C: An On-Line Programming Course

2.1 Binary and Hexadecimal Integers

Before we continue with our study of C, we will need to have a basic understanding of the way computers store data. As ominous as the topic sounds, it really just comes down to two things: 0 and 1. Even a minimal knowledge of how computers use these two numbers will allow you to practice defensive programming.

Imagine an ordinary light switch. Its two positions allow the representation of two states. It is easy to see how the position of the switch can represent the two states ``light'' and ``dark.'' But the switch can also represent the two numbers 0 and 1: the ``off'' position, for example, might represent the number 0 and the ``on'' position can then represent the number ``1''. Now imagine a house with two rooms, r1 and r2. Assume, of course, that there is a light switch in each room. At night the house has 4 states:

r2	r1
dark	dark
dark	light
light	dark
light	light

Table 1

Now let us expand our table. We will number each of the four states of the house *starting at 0*. We will also represent each state numerically according to the position of the switches in creating that state. Remember that we are using 0 for ``off'' and 1 for ``on''. Our numerical representation will therefore be a string of 0's and 1's. The rightmost digit in the string corresponds to the state of room r1 and the one to its left represents r2:

State Number	r2	r1	Numerical Representation
0	dark	dark	00
1	dark	light	01
2	light	dark	10
3	light	light	11

Table 2

Of course with more rooms and switches there are more states. In general, if there are n switches then there

are 2^n different states. Here is a table for n = 3:

State Number	r3	r2	r1	Numerical Representation
0	dark	dark	dark	000
1	dark	dark	light	001
2	dark	light	dark	010
3	dark	light	light	011
4	light	dark	dark	100
5	light	dark	light	101
6	light	light	dark	110
7	light	light	light	111

Table 3

Now if we ignore the intermediate columns and just make the correspondence between the integer on the left and the string of 0's and 1's on the right, we have arrived at the basic principle of computer storage: each integer can be uniquely represented by a string of 0's and 1's. The representation is known as the ***binary representation*** of the integer. The numbers 0 and 1 are referred to as ***binary digits*** and each digit of the binary representation of a number is said to be a ***bit*** (a contraction of binary digit). Although some early computers did use mechanical switches to represent bits, the effectiveness of computers stems from the rapidity with which bits can be stored and manipulated electronically.

The binary representation of integers is actually very similar to the ordinary decimal representation. Consider the number 2837. What this decimal representation signifies is the number

$$2 \cdot 10^3 + 8 \cdot 10^2 + 3 \cdot 10 + 7$$

Compare this with the number 6 whose binary representation is given as 110 in Table 3. What the binary representation of 6 signifies is

$$1 \cdot 2^2 + 1 \cdot 2^1 + 0$$

Now let us write the binary representation of the decimal integer 2837. Small powers of 2 are tabulated below:

n	12	11	10	9	8	7	6	5	4	3	2	1	0
2^n	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

From the table we see that 2048 is the greatest power of 2 that is not greater than the given integer: 2837 = 2048 + 789. We continue with the remainder 789. The greatest power of 2 that is no larger than 789 is 512: 789 = 512 + 277. After repeating this process we obtain:

$$2837 = 2048 + 512 + 256 + 16 + 4 + 1$$

or

$$2837 = 1 \cdot 2^{11} + 0 \cdot 2^{10} + 1 \cdot 2^9 + 1 \cdot 2^8 + 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1$$

It follows that the binary representation of 2837 is

101100010101

As you can see, the binary representation of even relatively small numbers can require several bits. One way to facilitate working with binary numbers is to group them in successive blocks of four bits, starting at the right and padding with zeros on the left, if necessary, to complete the leftmost block of four bits. In this way, the binary representation of 2837 becomes

1011 0001 0101

Although writing the number in this way makes it two spaces longer, it also makes it easier to assimilate. Moreover, it prepares the binary number for a trick that we have up our sleeve.

In the paragraph above we have considered the binary (base 2) and decimal (base 10) representations of an integer. We have already discussed why computers use binary integers. Humans presumably settled on decimal integers because of our ten fingers. There is, however, no mathematical reason to prevent us from choosing any integer greater than 1 and using it as a base. In fact, we shall soon see 16 makes a convenient choice. The representation obtained by using 16 as a base is called the ***hexadecimal representation***.

If you maintain a web page, then you may have already encountered hexadecimal integers. That is because the color of the background and the colors of the various types of text are determined by a string of three two-digit hexadecimal integers. These three numbers control the mix and intensity of red, green, and blue (RGB). For example, black is 000000 and white is FFFFFF. Regular text on this page has RGB value 72001E. We have distinguished new terms by highlighting them in the yellowish color FF7300. What, then, are these hexadecimal numbers E and F?

Suppose that we are using the decimal number b as a base. (You should have 16 in mind to keep things concrete.) To represent the decimal number b in its base b representation, we express it as

$$b = 1 \cdot b^1 + 0$$

This expansion of b tells us that 10 is its base b representation. In other words, b is the smallest two-digit base b number. Thus, each integer 0, 1, 2, ... , b-1 is a one digit number. There must be a symbol to represent each of these numbers in base b. This presents no problems when b is not greater than 10. We already have commonly accepted symbols for all these numbers (namely 0, 1, 2, ... 9). However when b is greater than 10

we require new symbols. For example, for b = 16, we use A, B, C, D, E, and F for the decimal integers 10, 11, 12, 13, 14, and 15 respectively. Here is a table of some decimal integers with their hexadecimal equivalents:

Decimal	1	2	...	9	10	11	12	13	14	15	16	...	25	26	...	31	32	...	159	160	...	255
Hexadecimal	1	2	...	9	A	B	C	D	E	F	10	...	19	1A	...	1F	20	...	9F	A0	...	FF

Notice that 256 numbers, the decimal numbers 0 through 255, can be represented as two-digit hexadecimal numbers.

Example 1: Convert the decimal integer 2837 to a hexadecimal integer.

Solution: As we have seen, two digits will not suffice. We calculate that the cube of the decimal integer 16 is the decimal integer 4096. Since 2837 is less than 4096, we see that we may express 2837 as a three-digit hexadecimal integer rst where

$$2837 = r \cdot 16^2 + s \cdot 16 + t$$

We calculate

$$2837 = 11 \cdot 16^2 + 21 = 11 \cdot 16^2 + 1 \cdot 16 + 5$$

The hexadecimal representation of decimal 2827 is therefore **B15**.

■

You may recall that we mentioned that we had a trick up our sleeves for easily dealing with long binary integers. For example, we found that the decimal number 2837 has binary expansion 101100010101. We rewrote this in blocks of four bits starting from the right: 1011 0001 0101. Now observe that each block of four bits can represent numbers ranging from 0 (0000) to 15 (1111). These numbers are exactly the base 16 digits. If we represent each block of four bits by its hexadecimal digit, we obtain 1011 0001 0101 (binary) = B 1 5 (hexdecimal). That is precisely the hexadecimal representation of the decimal integer 2837:

2837 (decimal) = 1011 0001 0101 (binary) = B15 (hexadecimal)

or, using a standard notational device that is employed whenever there is a danger of confusing the base of an integer:

$$2837_{(10)} = 1011\ 0001\ 0101_{(2)} = B15_{(16)}.$$

We may state our observations in the form of a general rule. We leave it to the reader to convince himself of the validity of this rule:

Hexadecimal to Binary to Hexadecimal Rule

To convert a hexadecimal integer to a binary integer, replace each hexadecimal digit with its four bit binary representation, padded with 0's on the left if necessary. To convert a binary integer to a hexadecimal integer, start at the right most bit, group four bits at a time moving to the left, and replace each four bit binary group with its (one-digit) hexadecimal equivalent.

Example 2: Convert the hexadecimal integer E3F to a binary integer. Convert the binary integer 10100011101 to a hexadecimal integer.

Solution: Since $E_{(16)} = 14_{(10)} = (8 + 4 + 2 + 0)_{(10)} = 1110_{(2)}$, $3_{(16)} = 0011_{(2)}$, and $F_{(16)} = 15_{(10)} = (8 + 4 + 2 + 1)_{(10)} = 1111_{(2)}$, the hexadecimal integer E3F is $1110\ 0011\ 1111_{(2)}$. We have left spaces between the groups of four bits only as an aid to the eye (which otherwise has trouble taking in such a long bit string). In the other direction, $10100011101_{(2)} = 101\ 0001\ 1101_{(2)} = 51D_{(2)}$.

■

Now that you have learned about positive integers, you might be wondering about negative numbers. On the other hand, if you have carefully followed the discussion so far, you might be thinking that signed integers are tailor made for the binary number system - there are, after all, two signs or states, positive and negative. Indeed it is possible to represent positive and negative integers by reserving one bit for the sign. The details, however, are not entirely straightforward and we defer a discussion until the next section.

If computers could only handle numbers they would still be useful to mathematicians, scientists, engineers, and the military. But ever since the early 1950's most computer applications have consisted of handling vast amounts of business and government data, data that has included non-numerical records. And nowadays one is as likely to turn to one's computer for e-mail or word processing as anything else. We even use text to write the programs that when compiled into machine language tell computers what to do. So how do we convert punctuation and the letters of the alphabet into bit strings? It turns out that that is very easy to do. Keep in mind that there are only 10 different decimal digits, 52 upper and lower case letters of the English alphabet, and a handful of commonly used written symbols (including punctuation marks). If we allocate 7 bits for the storage of printable characters, then we are able to store 128 different items - more than enough for the demands of the English language. In fact, the standard character set known as ASCII (American Standard Code for Information Interchange) does use 7 bits for each character. A table is presented below.

ASCII Character Set

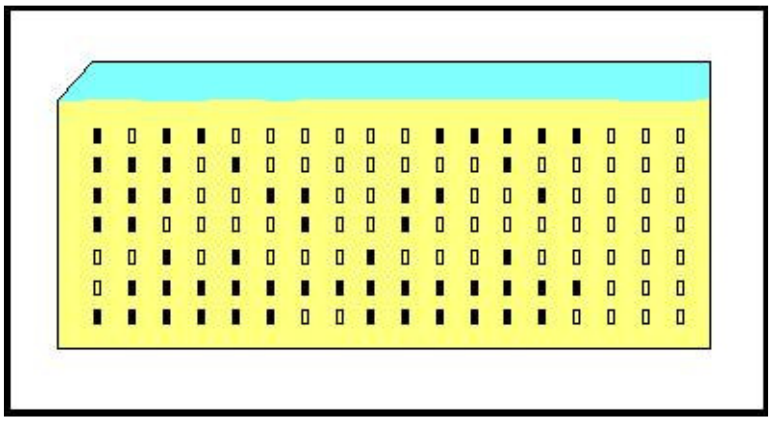
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y

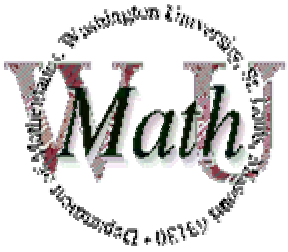
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Table 4

To find the ASCII code of a character appearing in this table, write the one or two digit number that begins its row. Append to that number the one digit number at the top of the character's column. The resulting two or three digit decimal integer is between 0 and 127. The ASCII code for the selected character is the 7-bit binary representation of this number. For example, the character `A' is in the row that begins with 6 and the column that is headed by 5. Therefore, the ASCII code for `A' is the 7 bit binary representation of 65: 1000001. Notice that the first 32 ASCII characters are control characters represented by mnemonics. They are: null, start of heading, start of text, end of text, end of transmit, enquiry, acknowledge, audible bell, backspace, horizontal tab, line feed, vertical tab, form feed, carriage return, shift out, shift in, data link escape, device control 1 (through 4), negative acknowledge, synchronous idle, end transmission block, cancel, end of medium, substitution, escape, figures shift, group separator, record separator, unit separator, and blank space.

In the author's early days of computing, that is to say, quite some time ago, console input was relatively rare. Programs were punched out at keypunches on *Hollerith cards.* The two bits of binary representation were achieved by the two states ``punched out hole'' and ``no hole.'' As an exercise in binary communication, decode the message below by turning the card 90 degrees clockwise so that the notch is at the top right corner. The depiction is not historically accurate because the message below has been encoded with ASCII. In the days of the Hollerith card, IBM's EBCDIC encoding was the norm. In the future, the enormous Unicode character set may prevail. But as of the time of this writing, ASCII files are the *lingua franca* of computing. They make it possible to compose a web page on one's PC and upload it to a UNIX server so that it may be downloaded by a Mac user half way around the world.





Brian E. Blank
Department of Mathematics
Washington University in St. Louis
St. Louis, MO 63130
e-mail: brian@math.wustl.edu

Last Updated: February 04 1998