cisc1110 fall 2010 lecture IV.2

- *bases*
- storing numbers
- base conversion
- how different types of data are stored in the computer
- hexadecimal and octal constants

storing numbers

when your code says:

```
int x = 19;
```

and we draw the computer's memory to look like this:

x → [19]

what is really stored looks like this:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1

where each 0 or 1 is a *switch* that is either off (0) or on (1)

the set of switches can be interpreted as a *binary* or *base 2* number!

$19_{10} = 10011_2$

remember bases?

base 10:

$$\begin{aligned} 362 &= (2 * 1) + (6 * 10) + (3 * 100) \\ &= (2 * 10^0) + (6 * 10^1) + (3 * 10^2) \end{aligned}$$

base 2:

$$\begin{aligned} 1 &= 2^0 = 1 \\ 10 &= 2^1 = 2 \\ 100 &= 2^2 = 4 \\ 1000 &= 2^3 = 8 \\ 10000 &= 2^4 = 16 \\ &\ldots \end{aligned}$$

so

$$\begin{aligned} 10011_2 &= (1 * 2^0) + (1 * 2^1) + (0 * 2^2) + (0 * 2^3) + (1 * 2^4) \\ &= (1 * 1) + (1 * 2) + (0 * 4) + (0 * 8) + (1 * 16) \\ &= 19_{10} \end{aligned}$$

base conversion: 2 to 10

$1010100_2 =$

	=	=
$(0 * 2^0)$	$(0 * 1)$	0
$+ (0 * 2^1)$	$+ (0 * 2)$	$+ 0$
$+ (1 * 2^2)$	$+ (1 * 4)$	$+ 4$
$+ (0 * 2^3)$	$+ (0 * 8)$	$+ 0$
$+ (1 * 2^4)$	$+ (1 * 16)$	$+ 16$
$+ (0 * 2^5)$	$+ (0 * 32)$	$+ 0$
$+ (1 * 2^6)$	$+ (1 * 64)$	$+ 64$

$= 84_{10}$

base conversion: 10 to 2

$84_{10} =$

84 / 2	=	42	rem 0
42 / 2	=	21	rem 0
21 / 2	=	10	rem 1
10 / 2	=	5	rem 0
5 / 2	=	2	rem 1
2 / 2	=	1	rem 0
1 / 2	=	0	rem 1

$\Rightarrow 1010100_2$

two tricks

base 8 (octal):

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

base 16 (hexadecimal, "hex"):

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

- replace each octal (or hex) digit with the 3 (or 4) digit binary
- replace every 3 (or 4) binary digits with one octal (or hex) digit

back to storage and how different types of data are stored

x → 19

is really stored like this:

31	30	...	7	6	5	4	3	2	1	0
0	**0**	...	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**

- bits are numbered, from right to left, starting with 0
- the highest (rightmost, "most significant") bit (i.e., bit number 31) is the *sign* bit
- if the sign bit is 0, then the number is positive;
  if the sign bit is 1, then the number is negative
- negative numbers are encoding using a method called *two's complement*
- integers can also be *unsigned*
  which means that the sign bit is interpreted as another binary digit
- the largest signed integer value is: $2^{31} - 1 = 2,147,483,648$
- the largest unsigned integer value is: $2^{32} - 1 = 4,294,967,296$

storing characters: ASCII

- ASCII = American Standard Code for Information Interchange
- characters are stored as numbers
- standard table defines 128 characters
- for example, when you define:
  `char c = 'A';` the data is stored as a number:
  'A' $= 65_{10} = 01000001_2$
  like this:

c →

7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1

- sometimes it is handy to *convert* between integers and characters explicitly

```
char c = 'A';
int i;
i = (int)c;
```

in which case, the value of `i` will be 65.

the sizes of primitive data types

type	size	minimim value	maximum value
bool	1 bit	0	1
byte	8 bits	$-128 = -2^7$	$127 = 2^7 - 1$
char	8 bits	$-128 = -2^7$	$127 = 2^7 - 1$
short	16 bits	$-32,768 = -2^{15}$	$32,767 = -2^{15} - 1$
int	32 (or 16) bits	-2^{31} (2^{15})	$2^{31} - 1$ $(2^{15} - 1)$
long	32 bits	-2^{31}	$2^{31} - 1$
float	32 bits	$\approx -3.4E + 38$, 7 sig. dig.	$\approx 3.4E + 38$, 7 sig. dig.
double	64 bits	$\approx -1.7E + 308$, 15 sig. dig.	$\approx 1.7E + 308$, 15 sig. dig.

"sig. dig." = significant digits

(Note that the minimum and maximum values given above are based on using signed numbers.)

finding the size of things in a program

- C++ has a function called `sizeof()` which returns the size of its argument, in *bytes*
- for example, `sizeof( a )`, where verb+a+ is defined as an `int` variable, returns the value 4
- since there are 8 bits in a byte, then an `int` takes up $4 \times 8 = 32$ bits
- for example:

```
#include <iostream>
using namespace std;

int main() {
  int i;
  cout << "the number of bytes in an int is: " << sizeof( i )
       << endl;
  cout << "the number of bits in an int is: " << sizeof( i ) * 8
       << endl;
} // end of main()
```


hexademical and octal constants

- sometimes it is handy to use the hexadecimal (base 16) or octal (base 8) representation of a number in a program
- in C++, octal numbers are represented by using a leading zero in the number, which indicates that it is octal
- hexadecimal numbers are represented using `0x` before the value
- for example:

```
#include <iostream>
using namespace std;

int main() {

  int i = 10;   // set value using decimal notation (base 10)
  int o = 010;  // set value using octal (base 8)
  int h = 0x10; // set value using hexadecimal (base 16)

  // by default, cout will print all the numbers in decimal
  cout << "i = " << i << endl;
  cout << "o = " << o << endl;
  cout << "h = " << h << endl;
} // end of main()
```


want more?

- things to look up or research on your own:
  - how to print out numbers in octal and hexadecimal (see textbook page 251)
  - what is "two's complement"??
- note that there are appendices in the text book that list:
  - sizes of all the primitive data types
  - ASCII table
- for fun, have a look at `http://www.asciimation.co.nz/`...