CIS-350
INFRASTRUCTURE TECHNOLOGIES

REPRESENTING NUMERICAL DATA – CHAPTER 5 (Excerpts)
(Covered Superficially)

You do not have to read chapter 5. You can just rely on these lecture notes that follow.

Representing Integer Data (Signed Numbers)

Integer (whole) numbers are stored exactly in the computer.

So far, we discussed unsigned #s

unsigned #s

1 byte

$(00000000)_2 = (0)_{10}$ - minimum # that can be stored in 1 byte
$(11111111)_2 = (2^8\text{-}1)_{10}=(255)_{10}$ - maximum # that can be stored in 1 byte

size limit or range = [0,255]
1 byte can store $2^8 = 256$ unique patterns

2 bytes

$(0000000000000000)_2 = (0)_{10}$ - minimum # that can be stored in 2 bytes
$(1111111111111111)_2 = (2^{16}\text{-}1)_{10} = (65535)_{10}$ - maximum # that can be stored in 2 bytes

size limit or range = [0,65535]
2 bytes can store $2^{16} = 65536$ unique patterns

Using unsigned #s, you can not represent a negative value.

In programming languages such as C++, C, Java or C# most integer #s are stored in 2 or 4 bytes. It depends on the implementation.

The declaration statement in C++/Java/C#

```
unsigned int i;     //will store an unsigned # (for addresses, counters), rare declaration
int j;              //will store a signed #, most common declaration
```

Chapter 5 describes four different methods of storing and manipulating negative and positive integer #s (signed #s).

1. BCD - (Binary Coded Decimal) representation for unsigned and signed numbers
2. Sign and magnitude representation (representation for signed integers)
3. One's (1's) Complement Representation
4. Two's (2's) Complement Representation

The first 3 methods are not used in the contemporary computers. Method 4 is used only. As a result, we will deal with this method only.

Two's (2's) Complement Representation

- very often used to represent signed #s
- one representation for 0
- fast and simple arithmetic

Integer numbers are represented in the computer in the 2's complement form. The form allows to represent positive and negative numbers.

Consider 1 byte:

$(10000000)_2 = -1*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 0*2^1 + 0*2^0 = (-128)_{10}$

$(11111111)_2 = -1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = (-1)_{10}$

$(00000000)_2 = 0*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 0*2^1 + 0*2^0 = (0)_{10}$

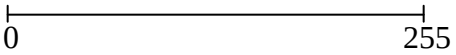
$(01111111)_2 = 0*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = (127)_{10}$

These are the 8-bit 2's complementary binary representations.

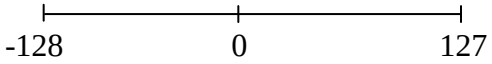
The leftmost bit is set aside for a sign: 1- negative, 0 - positive. The sign bit also contributes to the magnitude of the #.

The size limit (size range) for unsigned numbers stored in 1 byte: [0,255].
The size limit (size range) for signed numbers stored in 1 byte: [-128,127] (asymmetric).
1 byte can store 2^8=256 unique patterns.

Unsigned:



Signed:



Ex. (1-byte representation)

Given: $(35)_{10} = (00100011)_2$

Find: $(-35)_{10} = (?)_2$

1. compute the 1's complement by inverting each bit:

00100011
11011100

2. compute 2's complement by adding 1 (pay attention to a potential carry)

```
 11011100
 +        1
----------
```

$(11011101)_2 = (-35)_{10}$

Check: $= -1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = (-35)_{10}$

Ex. (Same with 2 bytes – 2-byte representation) – very similar to a problem in Homework 3.

Given: $(35)_{10} = (00000000\ 00100011)_2$ - note that we padded the most significant byte with 0s.

Find: $(-35)_{10} = (?)_2$

1. compute the 1's complement by inverting each bit:

00000000 00100011
11111111 11011100

2. compute the 2's complement by adding 1 (pay attention to a potential carry)

```
 11111111 11011100
 +                1
------------------------
```

$(11111111\ 11011101)_2 = (-35)_{10}$

Check: $= -1*2^{15} + 1*2^{14} + 1*2^{13} + 1*2^{12} + 1*2^{11} + 1*2^{10} + 1*2^9 + 1*2^8 + 1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = (-35)_{10}$

Ex.

$(-65)_{10} = (\ ?\ )_2$

Find $(65)_{10} = (01000001)_2$

Find 16-bit representation of $(65)_{10} = (00000000\ 01000001)_2$

Find 1's complement of $(00000000\ 01000001)_2$

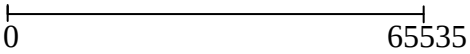
$(11111111\ 10111110)_2$

Find 2's complement:

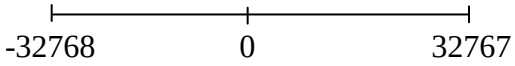
```
  11111111 10111110
+                1
 -------------------
  11111111 10111111 = (-65)10
```

The limit size (size range) for unsigned numbers stored in 2 bytes: [0, 65535].
The limit size (size range) for signed numbers stored in 2 bytes: [-32768, 32767] (asymmetric).
2 bytes can store 2^{16}=65536 unique patterns.

Unsigned:



Signed:



Real/Floating Point Numbers:

- #s with the decimal fraction
- many of them are stored in the computer in the approximate form (Ex. 0.51)
- have larger storage requirements
- consume more CPU time to perform arithmetic operations: +, -, *, /

Ex. C++/Java/C#

```
int a, b, c;            // 2 or 4 bytes each, stored in the exact form
long int d, e, f;       // 4 or 8 bytes
char g;                 // 1 byte, if stored in ASCII or EBCDIC code; 2 bytes if stored in Unicode
float h, i, j;          // 4 bytes, many floating point #s stored in the approximate form
double k, l, m;         // 8 bytes, approximate form, more precision, larger magnitude

a=2;  b=3;  c=a+b;      // operations on integers are done very fast
d=65358;
e=105312;
f=d+e;                  // you may need the declaration long int to perform this addition

h=31.2;
i=89.8;
j=h+i;                  // #s of type float handle up to 7 significant digits after the decimal point

k=87.3823285619;
l=1.1e190;
m=k+l;                  // type double - stores up to 15 significant digits after the decimal point
```

In math to represent a real # such as -0.0000003579, you may use a shorter scientific or exponential notation. To do this, you need to know 6 items:

$-0.3579 * 10^{-6}$

mantissa part exponent part

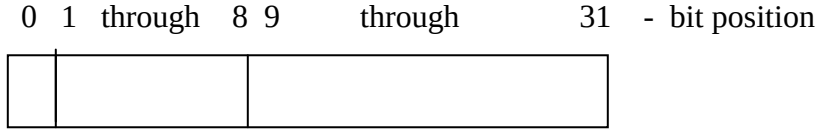
- sign of mantissa
- location of decimal point
- mantissa itself
- base
- exponent itself
- sign of exponent

The base and the location of decimal point are standardized in the computer and they do not have to be stored. The sign of exponent does not to be stored either since it is represented in the special notation called excess-N notation (Ex. excess-127 notation).

As a result, there are only 3 things to store in the computer:

- sign of mantissa
- exponent
- mantissa

Standard predefined format for storing a real # of type float - 4 bytes (Figure. 5.17, p. 166)



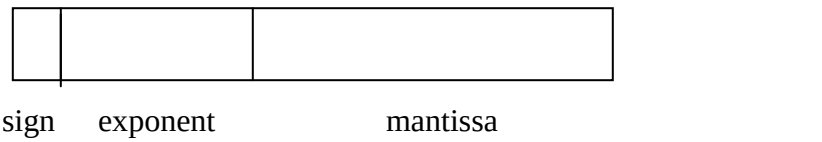
sign bit — exponent — mantissa

0 - positive, 1 - negative

exponent: determines the magnitude of a #

mantissa: determines the accuracy of a #

Standard predefined format for storing a real # of type double - 8 bytes



sign bit — exponent — mantissa

0 - positive, 1 - negative

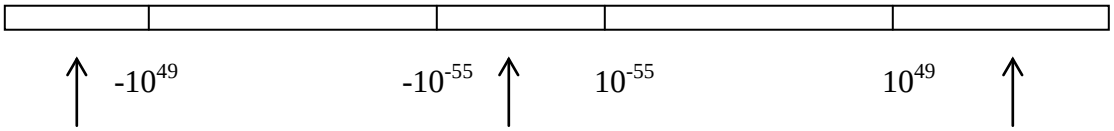
exponent: determines the magnitude of a #

mantissa: determines the accuracy of a #

When you perform arithmetic using real #s, you can create

- overflow: magnitude too large to be stored
- underflow: magnitude too small to be stored

Regions for type float (Fig. 5.15, p. 159)



overflow region (negative #s too large to be stored)

underflow region (negative and positive #s too small to be stored)

overflow region (positive #s too large to be stored)