Dr. Jozo Dujmović

3. BASIC DATA TYPES

Basic types of data are:

1. Unsigned integers
2. Signed integers (or simply integers)
3. Real numbers
4. Alphanumeric characters

These data types are supported by all computers, and are internally binary coded.

1. UNSIGNED INTEGERS

For simplicity let us study the case of 3-bit integers:

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

Note that $7 = 2^3 - 1$ (the largest unsigned number)

In the general case of n bits the range of numbers is:

Decimal	Binary
0	000...000
1	000...001
....	....
$2^n - 1$	111...111

$0 \le N \le 2^n - 1$

Basic arithmetic operations with unsigned integers are:

1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
5. Remainder (%)

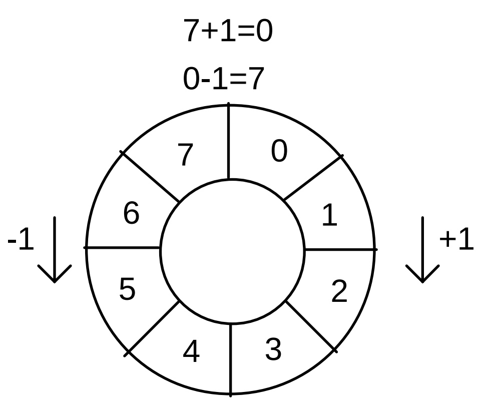
An example of binary addition in the case of 3-bit numbers:

```
          3   011       Note:
         +2   010           1+0 =01 (result=1, carry=0)
------------------          1+1 =10 (result=0, carry=1)
Result    5   101         1+1+1 =11 (result=1, carry=1)
------------------
Carry        010
```

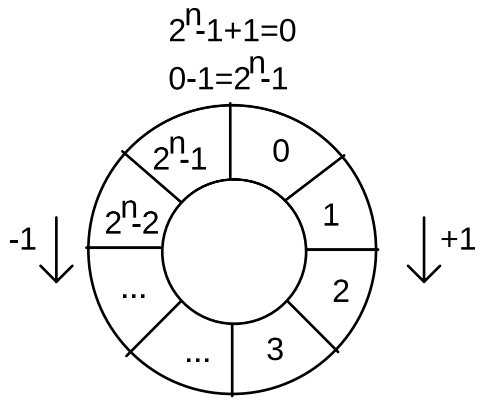
Examples of overflow:

```
         7  111
        +1  001
-----------------
Result   0  000
-----------------
Carry      111


         7  111
        +4  100
-----------------
Result   3  011
-----------------
Carry      100
```

We can define a "ring of unsigned integers":



The case with 3 bits



The general case with n bits

From this representation we see that the following "overflow errors" are possible:

6+3=1 (the sum of two numbers is less than each of the numbers)

3-5=6 (the result is greater than each of the numbers that are subtracted)

An example of binary multiplication:

```
2*3:                010 * 011
               --------------------
                   010
                  010
               --------------------
                   110   = 6
```

The division of unsigned integers generates only the integer part of the result:

5/2 = 2
3/2 = 1
1/2 = 0
6/7 = 0

The remainder is defined as the modulo operation:

k%m = k - (k/m)*m $\qquad$ ($m \le k$)

```
7%3 = 7 - (7/3)*3 = 7 - 2*3 = 7 - 6 = 1
7%4 = 7 - (7/4)*4 = 7 - 1*4 = 7 - 4 = 3
7%7 = 0
```

2. SIGNED INTEGERS (2's complement)

Signed integers use the same binary codes as unsigned integers, but the interpretation of codes is different. In the simple case of 3-bit integers:

Decimal	Binary	
3	011	Note that $3 = 2^2 - 1$
2	010	
1	001	
0	000	
-1	111	
-2	110	
-3	101	
-4	100	Note that $-4 = -2^2$

Therefore, the first bit can be interpreted as a sign bit: 1 denotes negative numbers, and 0 denotes nonnegative numbers.

In the general case of n bits the range of numbers is:

Decimal	Binary	
$2^{n-1} - 1$	011...111	(Largest positive number)
....	....	
1	000...001	
0	000...000	Range: $-2^{n-1} \le N \le 2^{n-1} - 1$
-1	111...111	
....	....	
-2^{n-1}	100...000	(Largest negative number)

Basic arithmetic operations with signed integers are the same as with unsigned:

1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
5. Remainder (%)

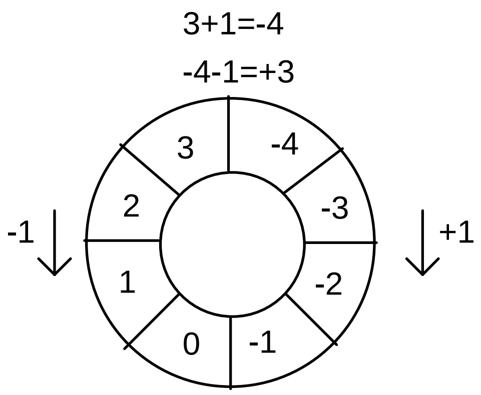
Binary addition is performed in the same way as with the unsigned numbers. However, all values are interpreted as 2's complement signed numbers, and subtraction is performed as addition of signed numbers. For example:

```
     3    011
    -2    110
------------------
     1    001
------------------
 Carry   110


    -1    111
    -2    110
------------------
    -3    101
------------------
 Carry   110
```

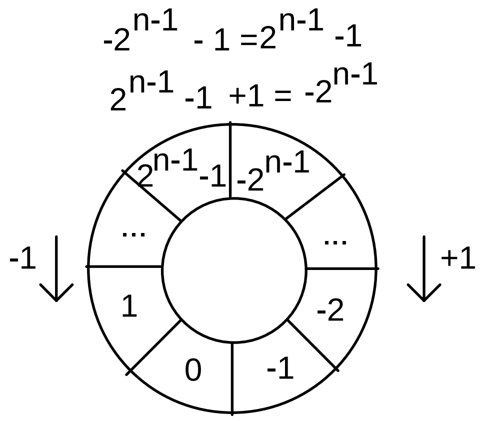
Examples of overflow errors:

```
     3    011
    +1    001
------------------
    -4    100      (Addition of positive numbers can yield a negative result)
------------------
 Carry   011


    -3    101
    -2    110
------------------
     3    011      (Addition of negative numbers can yield  a positive result)
------------------
Carry    100
```

These examples indicate that we can now define a "ring of (signed) integers":



The case with 3 bits



The general case with n bits

From this representation we see that the following integer overflow errors are possible:

1. The sum of two positive numbers can be negative
2. The sum of two negative numbers can be positive

An algorithm for conversion of positive binary integers to negative and vice versa:

1. Write the inverted number ($0 \rightarrow 1$, and $1 \rightarrow 0$)
2. Add 1 to the inverted number

Examples:

```
3:                    011
Inverted 3:           100
Add 1:                  1
--------------------------
-3:                   101


-3:                   101
Inverted 3:           010
Add 1:                  1
--------------------------
3:                    011
```

A fast algorithm:

1. Locate the rightmost bit 1
2. Keep the rightmost 1 and right zeros that follow the rightmost 1 (if they exist) unchanged
3. Invert the remaining bits ($0 \rightarrow 1$, and $1 \rightarrow 0$)

Example:

n = 0111100001000
-n = 1000011111000   (the rightmost 1000 remains unchanged)

The size of binary and decimal numbers:

If an unsigned binary number has b binary digits, how many decimal digits (d) will have the equivalent decimal number?

$$\underbrace{\boxed{101011100011111110101100010111}}_{b} \qquad \underbrace{\boxed{123456789}}_{d}$$

The total number of possible codes with b binary digits is 2^b. The total number of possible codes with d decimal digits is 10^d. If binary and decimal notations are equivalent then

$$2^b = 10^d \qquad | \quad \log$$

$$b \cdot \log 2 = b \cdot 0.3 = d \cdot \log 10 = d$$

$$\boxed{\begin{array}{c} d = 0.3\ b \\ \\ b = 3.3\ d \end{array}}$$

Example:

$b = 24$
$d = 0.3 \cdot 24 = 7.2$ (all 7-digit numbers plus some [small] 8-digit numbers)

Decimal/binary conversion

Binary to decimal:

$$101011 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 32 + 8 + 2 + 1 = 43$$

$$101011 = ((((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1 = ((2 \cdot 2 + 1) \cdot 2 \cdot 2 + 1) \cdot 2 + 1 = 43$$

```
1 *2+ 0
      2 *2+ 1
            5 *2+ 0
                 10 *2+ 1
                        21 *2+ 1 = 43
```

```
1    0    1    0    1    1
  *2 + *2 + *2 + *2 + *2 +
     2    5   10   21   43
```

Decimal to binary:

```
43 /2= 21 /2= 10 /2= 5 /2= 2 /2= 1 /2= 0
        +      +     +     +     +     +
        1      1     0     1     0     1
```

```
43
------
21   1
10   1
 5   0
 2   1
 1   0
 0   1
```

Negative decimal to binary:

1. Convert the absolute value: 43 $\rightarrow$ 00101011
2. Change the sign: -43 $\rightarrow$ 11010101

Negative binary to decimal:

1. Compute the absolute value: 11010101 $\rightarrow$ 00101011
2. Convert the absolute value to decimal: 101011 $\rightarrow$ 43

3. REAL NUMBERS

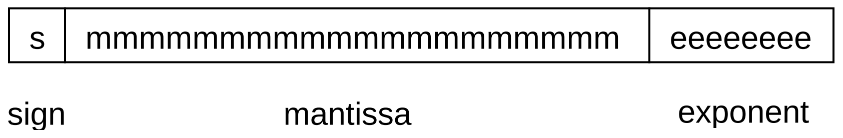
Decimal real numbers use decimal point: 123.456 . They can be written in the normalized form

$$123.456 = 0.123456 \cdot 10^3$$

In the normalized form the first digit after the decimal point must not be zero. Generally, the normalized decimal number can be written as follows:

$$0.mmmmmm \cdot 10^{eeee}$$

In this notation *mmmmmm* is called *mantissa*, and *eeee* is called *exponent*. Both mantissa and exponent are integers. Therefore, real numbers can be internally stored as two integers and a sign bit (s=0 for positive and s=1 for negative numbers) using the following format:



Computers use this notation for real numbers. Of course, both mantissa and exponent are binary coded, and the base is usually 2:

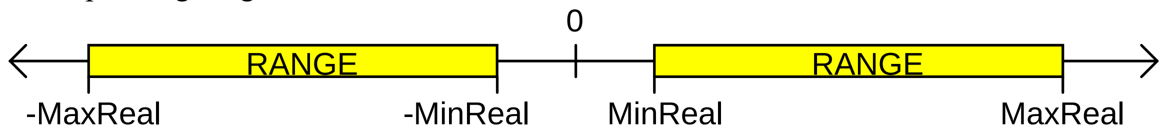
$$0.mmmmmm \cdot 2^{eeee}$$

Advantages of this notation:

1. The numbers have standard uniform *format*
2. Exponent determines the *range* of real numbers (minimum and maximum values)
3. Mantissa determines the *precision* of real numbers (number of significant digits)

Range

The *range* of numbers is determined by the smallest and the largest exponent. If the exponent field has 8 bits (the most frequent case) then -128 ≤ *eeee* ≤ 127. The corresponding range is



$$\text{MaxReal} = 2^{127} \cong 10^{38}$$
$$\text{MinReal} = 2^{-128} \cong 10^{-39}$$

Generally, let the size of exponent be *k* bits. Then the maximum exponent is

$$K = 2^{k-1} - 1$$

The range is determined as

$$\text{MaxReal} = 2^{K} \cong 10^{N}$$

Therefore, the maximum decimal exponent formula is

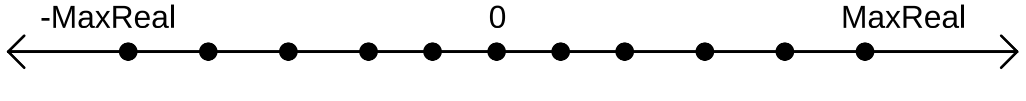
$$N = K \log_{10} 2 = 0.30103(2^{k-1} - 1)$$

For example, if k=8 then N=38.2 and if k=11 then N=308.

The only number between -MinReal and MinReal is zero. No numbers greater than MaxReal and less than -MaxReal can be presented (though, the IEEE standard includes also codes for +∞ and -∞). The distance between two adjacent real numbers is determined by the change in the last (least significant) bit of mantissa. An example of three adjacent numbers in the case of the 32-bit word is:

100101011111000010110000	10100101
100101011111000010110001	10100101
100101011111000010110010	10100101
24-bit mantissa	8-bit exponent

Using the 32-bit word we have approximately $2^{32} = 4.3 \cdot 10^{9}$ points representing real numbers. Therefore, the real axis with numbers can be visualized as follows:



There is an infinite number of real numbers (as well as integers) that cannot be represented using computers.

Precision

Precision is measured as the number of significant decimal digits M, which depends on the size of the binary mantissa *m*:

$$M = m\log_{10} 2 = 0.30103m$$

In the case of the 24-bit mantissa the number of significant decimal digits is M=7.2. If the result of a computation is displayed as 987.65432123456 than we know that the first 7 digits can be accurate while the rest is a meaningless sequence of random digits generated during the binary to decimal conversion:

$$\underbrace{987.6543}_{\text{accurate}}\;\underbrace{2123456}_{\text{random}}$$

Moreover, this is the ideal situation in which we assume that the method used for computing the resulting value is completely accurate. If the numeric method generates numeric errors (what is the usual case) than the situation might be as follows:

$$\underbrace{987.6}_{\text{accurate}}\;\underbrace{\boxed{543}\;\underbrace{2123456}_{\text{random}}}_{\text{inaccurate due to the numeric method errors}}$$

Normalization

In the case of decimal numbers, e.g. 7.25, the normalized notation is $0.725 \cdot 10^1$. In other words after the initial prefix "0." there must be a nonzero digit, and the correct value is adjusted using the power of the base 10. This concept can also be applied in the case of binary numbers. Without normalization we have

$7.25_{10} = 111.01_2$ (i.e. $1\cdot4 + 1\cdot2 + 1\cdot1 + 0\cdot0.5 + 1\cdot0.25 = 7.25$)

Similarly, we can write the normalized notation:

$$0.725 \bullet 10^1 = 0.11101_2 \bullet 2^3$$

Of course both the base and the exponent of binary equivalent should be written in binary:

$$7.25_{10} = (0.11101 \bullet 10^{11})_2$$

The binary mantissa is now 11101 and the binary exponent is 11. Consequently, the real number +7.25 can be stored in memory in the following format:

0	11101000000000000000000	00000011

Similarly, the negative number –7.25 can be stored as follows:

1	11101000000000000000000	00000011

The first digit of the normalized mantissa must be nonzero, and in the binary system the only such digit is 1. Since the first digit of the binary mantissa is always 1 it is reasonable to ask why this digit should be stored in memory. This fixed value might be assumed (hidden) and automatically generated by the arithmetic unit, giving the possibility for expanding mantissa for an additional bit (or to get a "free" sign bit). In such a notation the number +7.25 with the hidden/assumed leading 1 would be the following:

0	11010000000000000000000	00000011

Of course, the fields of sign, mantissa, and exponent are stored in memory as a continuous string of binary digits. In the case of the 32-bit memory word 8 bits can be used for exponent, 24 bits for mantissa (one is assumed and 23 are physically present in memory), and one bit for sign giving the following 32-bit memory word that represents +7.25:

01101000000000000000000000000011

Standard and double precision

Standard precision:

- Word size = 32 b
- Mantissa = 24 b
- Exponent = 8 b
- Range (MaxReal) = 10^{38}
- Precision (number of significant decimal digits) = 7.2

Double precision:

- Word size = 64 b
- Mantissa = 53 b
- Exponent = 11 b
- Range (MaxReal) = 10^{308}
- Precision (number of significant decimal digits) = 15.9

The double precision increases both the range and the precision of real numbers. However, some older systems may increase only precision:

- Word size = 64 b
- Mantissa = 56 b
- Exponent = 8 b
- Range (MaxReal) = 10^{38}
- Precision (number of significant decimal digits) = 16.9

Rounding, overflow, and underflow

The majority of arithmetic operations with real numbers (binary or decimal) include rounding:

- increase the last decimal digit in a field by 1 if the first decimal digit outside the field is 5 or more

- increase the last binary digit in a field by 1 if the first binary digit outside the field is 1 or more

For example:

	Field	Fraction outside the field
Before rounding:	$\boxed{123.4567}$	89123456

	Field	
After rounding:	$\boxed{123.4568}$	**DECIMAL ROUNDING**

	Field	Fraction outside the field
Before rounding:	$\boxed{0.110110}$	10011001100111

	Field	
After rounding:	$\boxed{0.110111}$	**BINARY ROUNDING**

If the absolute value of the result of an arithmetic operation is greater than MaxReal, this condition is called the overflow. If the absolute value of the result of an arithmetic operation is less than MinReal, this condition is called the underflow.

Decimal/binary conversion of real numbers

Binary to decimal:

$$10.11 = 1\cdot 2^1 + 0\cdot 2^0 + 1\cdot 2^{-1} + 1\cdot 2^{-2} = 2 + 0.5 + 0.25 = 2.75$$

Decimal fraction to binary:

```
0.1875 *2= 0.375 *2= 0.75 *2=  0.5 *2= 0
======     +         +         +       +
           0         0         1       1     →  0.0011
                                                 ======
```

Decimal to binary conversion can generate infinite periodical sequences:

```
0.3 *2= 0.6 *2= 0.2 *2= 0.4 *2= 0.8 *2= 0.6 *2=...
===     +       +       +       +       +
        0       1       0       0       1  →  0.0(1001)
                                             ==========
```

Since this sequence cannot fit in a finite mantissa, we have a frequent loss of accuracy during the decimal/binary/decimal data conversions. The numbers used by the computer users are regularly decimal numbers, while the internal presentation is always binary. Therefore, each numerical input/output operation causes data conversions.

Addition errors: truncation and associativity

Suppose that a machine has only 4 significant decimal digits. Let R=123.4 and r=0.049. Then

R + r = 123.4 + 0.049 = 123.4

because the value of r is too small to cause rounding:

```
     123.4|
    +  0.0|49
     -----|--
     123.4|49  =  123.4
```

This truncation phenomenon can be generally expressed as follows:

Let δ be the smallest number such that $1.0+\delta \neq 1$. Then, for all $x<\delta$, $1.0+x=1.0$.

Another consequence of the truncation phenomenon is that addition is sensitive to the order of performing addition operations. Here is an example based on R and r:

R+r+r+r = (((R+r)+r)+r) = 123.4
r+r+r+R = (((r+r)+r)+R) = 123.5

We have shown why 123.4+0.049 yields 123.4 and this explains the first of the above examples. In the second case we have

```
 0.049|
+0.049|
 -----|--
 0.098|     = r+r
+0.049|
 -----|--
 0.147|     = r+r+r


 123.4|
+  0.1|47
 -----|--
 123.5|47   = 123.5
```

Generally, the addition of real numbers should always start with the smallest numbers and end with the largest. Here is an experiment performed using a computer program and standard precision real numbers:

1 + 1/2 + 1/3 + ... + 1/900 = 7.380173
1/900 + 1/899 + ... + 1/2 + 1 = 7.380166

Similarly, if we write a program for computing

1 - 1/2 + 1/3 -1/4 + ... = ln 2 = 0.6931472

then we can do it in four different ways:

1. left to right, all terms
2. right to left, all terms
3. left to right, separately positive, and separately negative terms
4. right to left, separately positive, and separately negative terms

If we sum a sufficiently large number of terms, we can get four different results!

Computation of powers

It is important to differentiate integer and real powers. In the case of integer powers, the power is usually reduced to multiplications. For example:

$$(-2)^9 = ((((-2)^2)^2)^2 * (-2)$$

In other words, to compute x^n we don't have to have n-1 multiplications (in the above example n=9 and the number of multiplications is 4). However, the power function is reduced to multiplications and x can be either positive or negative.

In the case where the power n is *not* an integer, we cannot multiply. In such cases the power function computes the power using the exponential and the logarithm functions:

$$x^n = \exp(n * \ln(x))$$

However, now x must not be zero or negative, since the logarithm function is defined only for positive arguments. Therefore, if the power is expressed as a real number, for example $(-2)^{9.0}$, this can in some cases activate the logarithmic version of the power function and cause errors (9.0 has an integer value, but it is internally coded as a real number).

Numerical problems

The number of elementary numerical problems is relatively small, but it is important that programmers know them, and are prepared to avoid them. The list of basic problems, explained in this chapter is:

1. The range of all numbers is limited: this holds for unsigned integers, signed integers, and real numbers of different sizes.
2. The sum of unsigned integers can be less than the numbers being added.
3. The sum of positive integers can be negative.
4. The sum of negative integers can be positive.
5. The precision of real numbers is limited.
6. Addition of real numbers is restricted by truncation errors (1+x can yield 1)
7. The sum of real numbers depends on the order of addition.
8. Power function uses multiplication for integer powers, and logarithms for real powers; this can cause problems when computing powers of negative numbers.

4. ALPHANUMERIC CHARACTERS

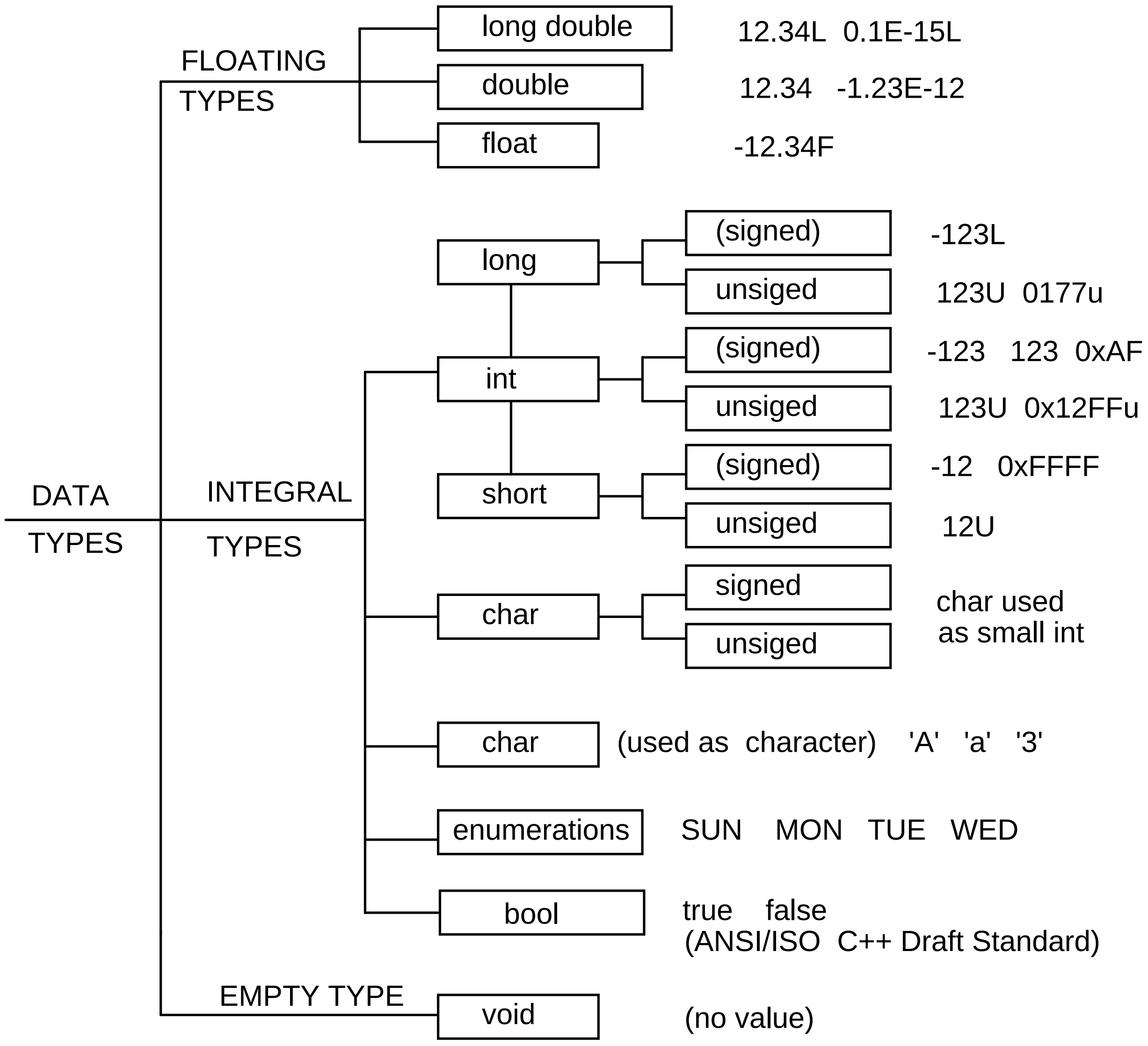
Alphanumeric characters are internally coded as small (8-bit) integers. The American Standard Code for Information Interchange (ASCII) defines 128 printable and non-printable characters that are coded using integers from 0 to 127. The non-printable characters are used for control purposes (e.g. sound signal, backspace, delete, etc.), and include codes 0-31, and 127. All other characters (except space) are printable. The complete ASCII table can be found in all programming texts. It is necessary, however, to memorize some special very important codes. They are:

Decimal value	Character	Use
0	Null	String terminator
7	Bell	Sound signal
8	Backspace	Delete character left of cursor
10	Line feed	Move cursor to the beginning of current line
13	Carriage return	Move cursor to the next line
27	Escape	Prefix of terminal commands
32		Space
48	0	Zero
49	1	One
...	...	...
65	A	First capital letter in the alphabet
66	B	Second capital letter
...	...	...
97	a	First lower case letter
98	b	Second lower case letter
...	...	...

It is useful to note the following:

1. Capital letters can be converted to lower-case letters by adding 32.
2. Lower-case letters can be converted to capital letters by subtracting 32
3. Both the lower-case alphabet and the upper-case alphabet are organized as ascending sequences of numeric codes. Consequently, the lexicographic sort can be performed as a numeric sort.
4. Numeric characters can be converted to integers by subtracting 48.

CLASSIFICATION AND SUMMARY OF BASIC DATA TYPES



Note: In C++ programs integers with prefix 0 are interpreted as octal numbers. Integers with prefix 0x are interpreted as hexadecimal numbers. This also holds for extraction of input values using cin >> n.

Examples of data declarations:
unsigned char ucmin=0, ucmax=255;
char a='A', nl='\n', t='\t';
unsigned short int usi1, usi2;
enum month {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
month m1, m2, m3;
long double d1, d2;