Computers

Data Representation

Grau en Ciència i Enginyeria de Dades

Facultat d’Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

2021-2022 Q2
Creative Commons License

This work is under a Creative Commons Attribution 4.0 Unported License

The details of this license are publicly available at https://creativecommons.org/licenses/by-nc-nd/4.0
Table of Contents

• Basic Concepts
  • Representation of several data types

• Data Type representation
  • Impact of accuracy error
  • Impact of compatibility issues
  • Dependencies on Hardware and Software
Basic Concepts

• Need to represent data in the Computer

- Programs (sentences, expressions, operations, numbers...)
- Strings (quoted by “ “)
- Characters (quoted by ‘ ’)
- Many other
- Natural language

Hello? Are you there?

2*π*r

2*3.141592654*r

“2*3.141592654*r”

‘2’ ‘*’ ‘3’ ‘.’ ‘1’ ‘4’ ‘...’
Basic Concepts

• The computer is based on transistors (circuits can be open or closed)
  • Any data should be represented by 0s and 1s

Instructions: 0010 1010 1100 1010 1101
0010 1010 110010010000111 1010 1101

Strings: “0110 1001 0111 1000 0101 1100 ...”

Characters: 0110 1001 0111 1000 0101 1100 ...

Natural language, sound, images...
Basic Concepts

• Binary Digit: it is a number that can adopt two values: 0 or 1
  • Many interpretations: true/false, positive/negative, on/off, etc

• Bit: contraction from “Binary Information Digit” (John W. Tukey, 1947)
  • It is the maximum amount of information that can be conveyed by a binary digit
    • A binary digit can convey between 0 and one bit of information

• Byte: it is a unit of digital information, usually comprising 8 bits
  • The smallest addressable unit of memory in many, but not all, computer architectures
    • Introduced in the ‘60s, and popularized by Intel (8008) in the ‘70s
    • It was used to encode a text character
Basic Concepts – character encoding

• Byte: unit of digital information, usually comprising 8 bits
  • The smallest addressable unit of memory in many, but not all, computer architectures
    • Introduced in the ‘60s, and popularized by Intel (8008) in the ‘70s
    • It is used to encode text characters (ASCII)

  01001101 (interpreted as an ASCII character)  ‘M’

  01001110 (interpreted as an ASCII character)  ‘N’

  01001110 (interpreted as an ASCII character)  ‘O’
Basic Concepts – 8-bit (byte) binary number

- **LSB**: Least Significant Bit - gives the value 0 or \(1 \times 2^0 = 0 \text{ or } 1\)
- **MSB**: Most Significant Bit - (on a byte) gives the value 0 or \(1 \times 2^7 = 0 \text{ or } 128\)

**Example**

\[
\begin{array}{cccccc}
0 & 1 & 0 & 0 & 0 & 1 \\
64 & 32 & 16 & 8 & 4 & 1
\end{array}
\]

\[64 + 4 + 2 + 1 = 71\]

- Interpreted as a “number” \(71\)
- Interpreted as a character ‘G’ (ASCII)

**Observation**: for now, we can only represent Natural numbers [0...255]
Signed Number Representations

• A binary number can represent negative values using different methods
  • MSB = 1 represents negative numbers
  • Optimize subtractions
    • E.g. add negative numbers

• The most known and used methods are:
  • Sign and magnitude
    • Commonly used to represent the mantissa of floating points (see later slides)
  • One’s Complement
    • Early computers and other current particular usages
  • Two’s Complement
    • Most of current computers
  • Biased representation
    • Exponent of floating points (see later slides)
Two’s Complement

- Range:
  - From $-2^{(N-1)}$ to $+2^{(N-1)-1}$

- Advantage compared to one’s complement
  - Disregard overflow bit in operations (e.g. subtraction)

- Negative number:
  - Decimal number -> invert bits -> add +1

- Exercise:
  - Decimal number: -23
    - Two’s Complement representation?

Example

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
\text{-27} & 2^6 & 64 & + & 4 & + & 2 & + & 1 & = & 71 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
\text{-27} & 2^6 & -128 & + & 64 & + & 4 & + & 2 & + & 1 & = & -57 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
\text{-27} & 2^6 & -128 & + & 32 & + & 16 & + & 8 & + & 1 & = & -71 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
\text{-27} & 2^6 & 32 & + & 16 & + & 8 & + & 1 & = & 57 \\
\end{array}
\]
Basic Concepts

• Data type is a classification that lets the interpreter/compiler know how the data is going to be used

• Direct impact on size and format

• Software and hardware support
  • Compiler/Interpreter and architecture
    • 32-bit vs 64-bit
#include <stdio.h>
int main(int argc, char * argv[]) {
    printf ("Hello, World!\n");
    return 0;
}

Compile & execute

$ gcc -o hello hello.c
$ ./hello

Interpret

$ python hello.py

These concepts will be completed in the next sessions/chapters
Basic Classification

• Scalar data types: a single value
  • Arithmetic (numbers), symbols and characters, boolean, enumeration, pointers

• Special data type: Void -- equivalent to no-data (of size 0), or datatype not specified

• Compound/Aggregate data types: built combining one or more scalar types
  • Arrays, structures, unions
Basic Background

- **Endianness**
  - The order of bytewise values in memory

- **Big-Endian**
  - Byte with **most** significant value: stored first (lowest memory address)
  - Data networking and mainframes

- **Little-Endian**
  - Byte with **least** significant value: stored first (lowest memory address)
  - x86 Intel processor family and most microprocessors

- Some architectures support both
  - E.g. Arm and IBM POWER in full, recent x86 and x86-64 have limited support (movbe)
Basic Background

• Big Endian: the location address points to the Big end of the number
  
  (Like writing the bytes left-to-right)

<table>
<thead>
<tr>
<th>&lt;num&gt;: 0x00010000</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

num = 0x42103278;

• Little Endian: the location address points to the Little end of the number

  (Like writing the bytes right-to-left)

<table>
<thead>
<tr>
<th>&lt;var2&gt;: 0x00010004</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>num = 0x42103278;</td>
</tr>
</tbody>
</table>

https://en.wikipedia.org/wiki/Endianness
Scalar Data Type: Arithmetic (numbers)

- Integer and real numbers
  - Complex numbers are held as a structure

- Signed and unsigned data types

- Numeral Systems
  - Binary, Octal, Decimal, Hexadecimal

https://en.wikipedia.org/wiki/Number
Integers

- **Char: 1 Byte**
  - Signed: from -128 to 127
  - Unsigned: from 0 to 255

- **Short: 2 Bytes**
  - Signed: from −32,768 to 32,767
  - Unsigned: from 0 to 65,535

- **Integer: 4 Bytes**
  - Signed: from -2,147,483,648 to 2,147,483,647
  - Unsigned: from 0 to 4,294,967,295

- **Long Long: 8 Bytes (if possible)**
  - Unsigned: from 0 to 18,446,744,073,709,551,615

- They can be expressed using different base notations:
  - Typically, decimal, octal, hexadecimal
**Base notations**

- Depending on the use/meaning, one may be more useful than the others

<table>
<thead>
<tr>
<th>Binary</th>
<th>Octal</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>07</td>
<td>7</td>
<td>0x7</td>
</tr>
<tr>
<td>1000</td>
<td>010</td>
<td>8</td>
<td>0x8</td>
</tr>
<tr>
<td>1001</td>
<td>011</td>
<td>9</td>
<td>0x9</td>
</tr>
<tr>
<td>1010</td>
<td>012</td>
<td>10</td>
<td>0xA (also 0xa, or 0x000a)</td>
</tr>
<tr>
<td>1011</td>
<td>013</td>
<td>11</td>
<td>0xB (0xb)</td>
</tr>
<tr>
<td>1100</td>
<td>014</td>
<td>12</td>
<td>0xC (0xc)</td>
</tr>
<tr>
<td>1101</td>
<td>015</td>
<td>13</td>
<td>0xD (0xd)</td>
</tr>
<tr>
<td>1110</td>
<td>016</td>
<td>14</td>
<td>0xE (0xe)</td>
</tr>
<tr>
<td>1111</td>
<td>017</td>
<td>15</td>
<td>0xF (0xf)</td>
</tr>
<tr>
<td>10000</td>
<td>020</td>
<td>16</td>
<td>0x10</td>
</tr>
<tr>
<td>10001</td>
<td>021</td>
<td>17</td>
<td>0x11</td>
</tr>
<tr>
<td>11100110</td>
<td>0346</td>
<td>118</td>
<td>(11100110) 0xE6</td>
</tr>
</tbody>
</table>
Integer versus Long Integer

• Long integer depends on both architecture Operating System

<table>
<thead>
<tr>
<th>OS</th>
<th>Arch</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>IA-32</td>
<td>4</td>
</tr>
<tr>
<td>Windows</td>
<td>x86-64</td>
<td>4</td>
</tr>
<tr>
<td>Linux</td>
<td>IA-32</td>
<td>4</td>
</tr>
<tr>
<td>Linux</td>
<td>x86-64</td>
<td>8</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>IA-32</td>
<td>4</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>x86-64</td>
<td>8</td>
</tr>
</tbody>
</table>

• Warning when porting code
  • The behavior may change
  • E.g.: IA-32 vs x86-64 Linux, x86-64 Win vs x86-64 Linux

Integer operations

- Binary operations: addition, substraction, multiplication, division...
  - shift, logical and, logical or, logical exclusive-or, logical not
- Bit by bit operation, like with the decimal system
- Implemented with transistors

### Addition

<table>
<thead>
<tr>
<th>0 0 1 1 0 0</th>
<th>0 0 0 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 + 4</td>
<td>4 + 1</td>
</tr>
<tr>
<td>= 12</td>
<td>= 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0 1 0 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 + 1</td>
</tr>
<tr>
<td>= 17</td>
</tr>
</tbody>
</table>

### Substraction

<table>
<thead>
<tr>
<th>0 0 1 1 0 0</th>
<th>0 0 0 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 + 4</td>
<td>4 + 1</td>
</tr>
<tr>
<td>= 12</td>
<td>=</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0 1 1 0 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(like /2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 1 0 1 1 1</th>
</tr>
</thead>
</table>

32 + 16 + 4 + 2 + 1 = 55
-32 + 16 + 4 + 2 + 1 = -9
Integer operations

• Flags may indicate specific conditions:
  • Zero: the result of the operation is 0 (saves time if there is a later check for 0)
  • Negative: the result of the operation is <0 (it may also save time)
  • Carry: applies to unsigned numbers, indicates that the result cannot be represented
    • Happens when the operation generates a “1” that does not fit in the number of bits used to represent the numbers
  • Overflow: applies to signed numbers, indicates that the result cannot be represented
    • Happens when the carry bit coming into position $2^{n-1}$ and the carry bit are different

\[
\begin{align*}
&\text{Subtraction} \\
8 + 4 & \quad = \quad 12 \\
16 + 4 + 1 & \quad = \quad 21 \\
32 + 16 + 4 + 2 + 1 & \quad = \quad 55 \\
-32 + 16 + 4 + 2 + 1 & \quad = \quad -9
\end{align*}
\]
Carry and overflow flags

- On substraction and addition

<table>
<thead>
<tr>
<th>Substraction</th>
<th>Addition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0</td>
<td>0 0 1 1 0 0</td>
</tr>
<tr>
<td>8 + 4</td>
<td>8 + 4</td>
</tr>
<tr>
<td>0 1 0 1 0 1</td>
<td>1 1 0 1 0 1</td>
</tr>
<tr>
<td>16 + 4 + 1</td>
<td>32 + 16 + 4 + 1</td>
</tr>
<tr>
<td>21</td>
<td>53</td>
</tr>
</tbody>
</table>

Carry is 1, meaning that the result (55) is not representable as a Natural (unsigned) number.

Carry is 1, meaning that the result (55) is not correct as a Natural number (it should be 65, but it does not fit).

Overflow is 0, meaning that 1 is correct.
Logical operations

- Bit level operations only (no carry possible)
- Usually on Natural (unsigned) numbers (no overflow possible)

**Logical and**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8  + 4</td>
<td>= 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**And**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 + 4 + 1</td>
<td>= 21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>= 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Truth table (taula de veritat)**

<table>
<thead>
<tr>
<th>Src 0</th>
<th>Src 1</th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
<th>NOT (Src 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Real Numbers

• Numbers with a fractional component

• Two main representations
  • Fixed-point versus Floating-point
    • The radix point is fixed or can float anywhere
      • The symbol to separate integer and fractional parts of a real number
    • The lesson “computer and its elements” will introduce the cost difference among Hw supports

• Implementation: tradeoff between cost and precision
  • Lack of hardware resources
    • E.g.: Multimedia decoders
  • Boost performance although degraded precision
    • E.g. Playstations, Doom
Fixed-point numbers

• Bits = 1 + m + n
  • 1 bit for sign (if signed)
  • m bits for integer component
  • n bits for fraction component

• Notation: $Q_{m.n}$
  • Integer number without fraction component ($Q_{m.0}$)
  • Fractional number without integer component ($Q_n$)
Fixed-point numbers

• Value = \(-2^m b'_s + 2^{m-1} b'_{m-1} + \ldots + 2^1 b'_1 + 2^0 b'_0 + 2^{-1} b_{n-1} + 2^{-2} b_{n-2} + \ldots + 2^{-n} b_0\)

• Programming language support
  • C and C++ have no direct support, but can be implemented
    • Embedded-C supports it (implemented in GCC)
  • Python has direct support through a module
    • I.e. decimal module

Example: 1110

\[
\begin{align*}
Q3.0: & \quad -2^3 + 2^2 + 2^1 = -2 \quad (1110.) \\
Q1.2: & \quad -2^1 + 2^0 + 2^{-1} = -2 + 1 + 0.5 = -0.5 \quad (11.10) \\
Q3: & \quad -2^0 + 2^{-1} + 2^{-2} = -1 + 0.5 + 0.25 = -0.25 \quad (1.110)
\end{align*}
\]
Fixed-point numbers

- Bits = 1 + m + n
  - 1 bit for sign (if signed)
  - m bits for integer component
  - n bits for fraction component

- Example, 1 byte
  - Sign, 1 bit
  - Integer, 5 bits -> 0 .. 31, with sign -32 .. +31
  - Fraction, 2 bits -> 0, 0.25, 0.50, 0.75
  - Coverage: -31.75 .. +31.75

- Try
  $ ./float-to-fixed 5 2 <fp-number> to see the encoding possibilities
Fixed-point numbers: accuracy problems

- Precision loss and overflow

- Results can require more bits than operands
  - Round or truncate
  - Specify different size for result

- Boundary numbers to prevent overflow

- Exception: overflow flag
  - If supported by hardware
Floating-point numbers

• Bits = 1 + e + k
  • 1 bit for sign (if signed)
  • e bits for exponent \{1, \ldots, (2^e-1)-1\}
  • k bits for mantissa (fraction)
    • There is an implicit 1-bit (top left) equals to 1, unless exponent is equal to zero

• Most processors follow IEEE floating point standard
  • First version on 1985
  • Standardize formats
  • Special Values

Floating-point numbers

- Value = \((-1)^{\text{sign}} \times (1 + \sum_{k=1}^{k} b_{(k-i)} \times 2^{-i}) \times 2^{(e-(E_{\text{max}}))}\)

- Float: 4 Bytes
  - Sign bit, 8-bit exponent, 23-bit mantissa
- Double: 8 Bytes
  - Sign bit, 11-bit exponent, 52-bit mantissa
- Some languages support 10 bytes
  - Sign bit, 15-bit exponent, (1+63)-bit mantissa

Biased Exponent (bias = 127):
- 0 Denormal numbers 0.xxx
- 1 Smallest normal exponent
- 127 E+0
- 254 Larger normal exponent
- 255 Infinite / NaNs
# Floating-point numbers

- **Intel encoding**

<table>
<thead>
<tr>
<th>s</th>
<th>e</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign</td>
<td>Biased exp.</td>
<td>Magnitude</td>
</tr>
<tr>
<td>Single precision</td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>Double precision</td>
<td>11 bits</td>
<td>52 bits</td>
</tr>
<tr>
<td>Double extended precision</td>
<td>15 bits</td>
<td>1+63 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Sign</th>
<th>Exp.</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>$+\infty$</td>
<td>0</td>
<td>1 ... 11 1 ... 00</td>
</tr>
<tr>
<td>+Normals</td>
<td>$+$</td>
<td>0</td>
<td>1 ... 10 1 ... 11</td>
</tr>
<tr>
<td>+3.40 E+38</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>+1.000 E+0</td>
<td>0</td>
<td>01 ... 11 00 ... 01</td>
<td></td>
</tr>
<tr>
<td>+1.17 E-38</td>
<td>0</td>
<td>00 ... 00</td>
<td></td>
</tr>
<tr>
<td>+Denormals</td>
<td>$0$</td>
<td>00 ... 00</td>
<td>0.11 ... 11</td>
</tr>
<tr>
<td>+1.17 E-38</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>+1.40 E-45</td>
<td>0</td>
<td>00 ... 00</td>
<td>0.00 ... 01</td>
</tr>
<tr>
<td>+Zero</td>
<td>0</td>
<td>00 ... 00</td>
<td>0.00 ... 00</td>
</tr>
<tr>
<td>Negative</td>
<td>$-\infty$</td>
<td>1</td>
<td>11 ... 11 1 ... 00</td>
</tr>
<tr>
<td>-Zero</td>
<td>$-$</td>
<td>1</td>
<td>00 ... 00 0.00 ... 00</td>
</tr>
<tr>
<td>-Denormals</td>
<td>$-$</td>
<td>1</td>
<td>00 ... 00</td>
</tr>
<tr>
<td>-1.40 E-45</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-1.17 E-38</td>
<td>1</td>
<td>00 ... 00</td>
<td>0.11 ... 11</td>
</tr>
<tr>
<td>-Normals</td>
<td>$-$</td>
<td>1</td>
<td>00 ... 01 01 ... 11</td>
</tr>
<tr>
<td>-1.000 E+0</td>
<td>1</td>
<td>00 ... 01</td>
<td></td>
</tr>
<tr>
<td>-3.40 E+38</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-$\infty$</td>
<td>1</td>
<td>11 ... 10 1 ... 11</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>X</td>
<td>11 ... 11</td>
<td>1.0X ... XX</td>
</tr>
<tr>
<td>SNaN</td>
<td>X</td>
<td>11 ... 11</td>
<td>1.0X ... XX</td>
</tr>
<tr>
<td>QNaN</td>
<td>1</td>
<td>11 ... 11</td>
<td>1.10 ... 00</td>
</tr>
</tbody>
</table>

## Double precision floating point

- Equivalent table for 64-bit fp numbers
- Compare with results from valsfp64.c in Lab S3 – Data Representation

<table>
<thead>
<tr>
<th>Class</th>
<th>Sign</th>
<th>Exp.</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>0</td>
<td>11...11</td>
<td>1.00...00</td>
</tr>
<tr>
<td>+Normals</td>
<td>0</td>
<td>11...10</td>
<td>1.11...11</td>
</tr>
<tr>
<td>+1.79 E+308</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>+1.000 E+0</td>
<td>0</td>
<td>01...11</td>
<td>1.00...00</td>
</tr>
<tr>
<td>+2.22 E-308</td>
<td>0</td>
<td>00...01</td>
<td>1.00...00</td>
</tr>
<tr>
<td>+Denormals</td>
<td>0</td>
<td>00...00</td>
<td>0.11...11</td>
</tr>
<tr>
<td>+2.22 E-308</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>+4.94 E-324</td>
<td>0</td>
<td>00...00</td>
<td>0.00...01</td>
</tr>
<tr>
<td>+Zero</td>
<td>0</td>
<td>00...00</td>
<td>0.00...00</td>
</tr>
<tr>
<td>Negative</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>1</td>
<td>00...00</td>
<td>0.00...00</td>
</tr>
<tr>
<td>-Denormals</td>
<td>1</td>
<td>00...00</td>
<td>0.00...01</td>
</tr>
<tr>
<td>-4.94 E-324</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-2.22 E-308</td>
<td>1</td>
<td>00...00</td>
<td>0.11...11</td>
</tr>
<tr>
<td>-Normals</td>
<td>1</td>
<td>00...01</td>
<td>1.00...00</td>
</tr>
<tr>
<td>-1.000 E+0</td>
<td>1</td>
<td>01...11</td>
<td>1.00...00</td>
</tr>
<tr>
<td>-1.79 E+308</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-∞</td>
<td>1</td>
<td>11...11</td>
<td>1.00...00</td>
</tr>
<tr>
<td>NaN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>X</td>
<td>11...11</td>
<td>1.0X...XX</td>
</tr>
<tr>
<td>QNaN</td>
<td>X</td>
<td>11...11</td>
<td>1.1X...XX</td>
</tr>
<tr>
<td>-QNaN</td>
<td>1</td>
<td>11...11</td>
<td>1.10...00</td>
</tr>
</tbody>
</table>
Floating-point numbers

- Value = \((-1)^{\text{sign}} \times (1 + \sum_{1}^{k} b_{(k-i)} 2^{-i}) \times 2^{(e-(E_{\text{max}}))}\)
Floating-point numbers: support

• Float
  • C and C++: single-precision (32-bit)
  • Python: built-in double-precision (64 bit)

• Most 32-bit architectures comprises 64-bit support in FPU (floating-point unit)
  • IA-32 and x86-64 present 80-bit floating-point type (double-extended precision format)
    • From 1989: x87 FPU (80-bit)

• Quad-precision (128-bit)
  • Software support
  • Few architectures provide hardware support
    • E.g. IBM POWER9 processors (MareNostrum 4)
Floating-point numbers: accuracy problems

- Numbers that cannot be exactly represented as binary fractions
  - E.g. 1/10
    - 0.00011001100110011001100110011001100110011001100110011001...

- Conversion to integer loses accuracy due to truncate and roundoff
  - E.g. 56 / 7 = 8; 0,56 / 0,07 = can be 7
  - E.g. explosion of Ariane 5 rocket (1996)

- Commutative, but not necessary: associative and distributive
  - “(a + b) + c” could be not equal to “a + (b + c)”
  - “(a + b) * c” could be not equal to “a * c + b * c”
Scalar Data Type: Symbols and Characters

• Char data type
  • Encode alphanumeric data and symbols

• Several character set encoding
  • ASCII code (American Standard Code for Information Interchange)
    • Single byte using the bottom 7 bits. From 0 to 127
    • The standard for the early HTML
  • ISO-8859-1 code (Latin Alphabet)
    • 1 full byte (256 characters): extension to ASCII
    • The standard from HTML 2.0 to HTML 4.01
    • Problems with some symbols
  • Windows-1252 code (CP-1252): also called ANSI
    • It is a superset of ISO-8859-1 (more printable characters)
    • Used by default on legacy components of Microsoft Windows
    • ANSI comprises 8 bits: 1 additional bit compared to ASCII

• UNICODE
Chars: encoding

• Unicode Standard (created at late 80s by Xerox and Apple)
  • **UTF-8** (from 1 Byte up to 4 bytes, if necessary)
    • It is the dominant encoding
    • Support from Operating Systems (e.g. WindowsNT was the first OS that supported UTF-8)
    • It is the standard of HTML5 and websites
    • To support every language, even Klingon, and emojis
  • Several widths
    • UTF-16 (2-4 Bytes)
    • UTF-32 (4 Bytes)

Input Encoding = UTF-8
Chars: encoding issues

• Single Byte versus multiple bytes
  • Fixed-size versus variable-size characters
  • The need to represent character sets that cannot be represented in a single byte (e.g. Japanese, Chinese)
    • MBCSs: MultiByte Character Sets (old and legacy technology)
    • Unicode Standard

• Compatibility issues
  • Unicode-aware programs to manipulate data
    • E.g. fulfilled copy of null-terminated strings (correct process of zero bytes)
Chars: Binary-coded decimal (BCD)

• BCD encoding
  • Each digit is represented by a fix number of bits
• Some applications require exact precision – Financial
  • Digits (0..9) are encoded in 4 bits (1 nibble)
    • Bit combinations above 1001 (9) are not used; they can be used to encode +/- or the decimal point (.)
  • Unpacked version: 1 byte holds 1 BCD digit
    • 1 out of 2 nibbles is not used
  • Packet version: 1 byte holds 2 BCD digits
• Support
  • Support in IBM System/360, and (limited) in Intel CPUs
  • Native support in Ada, Cobol and PL/I
  • C/C++ and Python support through libraries
• Example: date and time
  • PS3 and some BIOS

https://en.wikipedia.org/wiki/Binary-coded_decimal
Displaying information

• Graphical
  • Resolution in pixels (e.g. FullHD 1920x1080)
  • Pixel: color encoding, usually... 24 bits
    • Red (R): 8 bits
    • Green (G): 8 bits
    • Blue (B): 8 bits
Scalar Data Type: enumeration and boolean

• Enumeration:
  • Ordered list of symbolic names bound to unique values
    • E.g.: colours {red, green, blue}
  • Integer size, by default

• Boolean:
  • True or false
  • Built-in data type in C++ and Python
    • But not in C (integer value: 0 vs 1)
  • Even it only needs one bit, it takes a byte
    • It must be addressable
Scalar Data Type: Pointer

- Value that refers to another value stored elsewhere (address == pointer)

```
int num = 0x0503;  //1283
int *p;
p = &num;
```

```
num = 0x00000503
&num = 0x08B00400
p = 0x08B00400
&p = 0x08B0040C
*p = 0x00000503
```
Special Data Type: void and void*

• The void data type is a keyword that refers to a placeholder to a data type to represent no data!!!!!!

• Different meanings
  • Void function (in C/C++): the function returns nothing
  • Void parameter (in C/C++): the function takes no parameters

• Void * is different...
  • It is a pointer that points to an unspecified data type...that is, anything!!!!
  • Really useful, but take care...

Aggregate Data Type: arrays

• A collection of items that can be selected/accessed using identifying key/s
  • E.g.: A collection of 5 integers in C/C++: int array[5];

• Implementation complexities
  • Elements of (same vs different) data type/size
  • Indexing keys: integer vs arbitrary values
  • Static vs dynamic array size
  • One vs multiple dimensions
Arrays: tradeoffs

• Memory layout for multidimensional arrays
  • Row-major versus column-major versus depth (for 3D) versus...

• Programming language based
  • Row major: C, C++, Python
  • Column major: Fortran, MatLab, R

• Hardware support to boost performance
  • Special registers
Pointer Arithmetics

• Pointers can point to subsequent mem @s based on the data type hold on
  • E.g. difference between char* (1@mem shift) and int* (4@mem shift)

• Pointers point to a given @ independent of the memory region
  • More details on memory @s in future lessons...
Aggregate Data Type: structures

• A record that groups several variables and place them in a particular memory block, accessed by a single pointer or variable name

```c
struct test {
    int enter;
    char caracter;
    float decimal;
};
```

• Padding bytes to align fields in memory aiming at boost performance
  • Processor architecture related (32 bits vs 64 bits)
Aggregate Data Type: unions

• Several values of different type can be accessed in the SAME mem @
  • Only one value at a time
  • Overwrite values

union test {
  int enter;
  char caracter;
  float decimal;
};

• Efficient way to use the same memory location for multiple purpose
• For type-less processing
Other aggregate data types related to OOP

- **Object Oriented Programming (OOP):** programming paradigm
  - Object is an instance of a class: a combination of variables, functions, and structs
    - E.g.: C++ and Python

- **Classes are an evolution of structs**
  - Contents (fields) with restricted access

- **String is different that an array of chars**
  - Class-based data type (C++)
    - With embedded functions and fields
  - **Sequence of characters ended with ‘\0’** (C/C++)
  - Array of chars is just an array of chars!!! (no ‘\0’ at the end)
Bibliography

• Computer Systems – A Programmer’s perspective (3rd Edition)
  • Randal E. Bryant, David R. O’Hallaron, Person Education Limited, 2016
  • https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004062589706711
    • Chapter 2: Representing and manipulating information

• Computer Organization and Design (5th Edition)
  • D. Patterson and J. Hennessy
  • http://cataleg.upc.edu/record=b1431482~S1*cat
    • Several chapters introduce different types of data

• Data Type Summary
  • MSDN (Microsoft)
    • Summary list of data types and their respective size in Windows based software