# VL03 Scalar Data Types

## 6. December

# Scalar Data Types

- Integer
- Double
- Character, String
- Regular Expressions

# Integer

> Any of the natural numbers, the negatives of these numbers, or zero.
>
> — The Merriam Webster Dictionary

- Ranges
  - Signed: $-(2^{n-1})$ to $(2^{n-1} - 1)$
  - Unsigned: 0 to $(2^{n} - 1)$
- Fast hexadecimal ⇒ binary conversions because $\log_2(16) = 4$
- Fast operations: Comparison, Negation, Addition, Multiplication

### Table 1. int32_t examples, the prefix 0x denotes a hexadecimal value

| Real Value | Memory Value |
|---|---|
| 0x0 | 0x0 |
| 0x1 | 0x1 |

| Real Value | Memory Value |
|---|---|
| -0x1 | 0xFFFFFFFF |
| -0x10 | 0xFFFFFFF0 |
| -0x100 | 0xFFFFFF00 |
| -0x1000 | 0xFFFFF000 |
| -0x10000 | 0xFFFF0000 |
| -0x100000 | 0xFFF00000 |
| -0x1000000 | 0xFF000000 |
| -0x10000000 | 0xF0000000 |
| -0x80000000 | 0x80000000 |
| 0x7FFFFFFF | 0x7FFFFFFF |

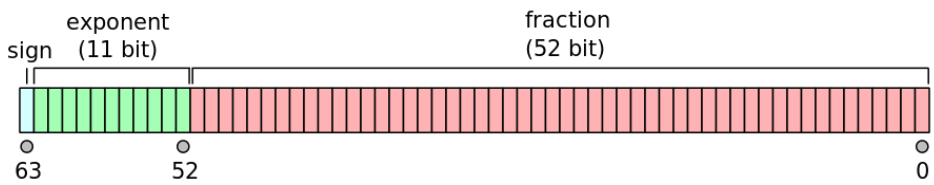
This representation mechanism is called **Two's Complement**. Sum of a positive number and its negative counterpart gives $2^n$ (except zero, n is number of bits). The conversion between negative and positive (multiplication with -1) can be done very fast as bit negation (ones' complement) and

increase by 1.

# Double

## IEEE-754 Standard for Floating-Point Arithmetic

```
 Specifies interchange and
arithmetic formats and methods
for binary and decimal floating-
point arithmetic in computer
programming environments.
IEEE = Institute of Electrical
and Electronics Engineers
```




## Representation

There are some reserved values for signed zero, infinity, and NaN (not a number). For non-reserved values, numeric representation can be obtained with the following formula:

$$(-1)^{\text{sign}}(1.b_{51}b_{50}\ldots b_0)_2 \times 2^{e-1023}$$

```c
#include <stdio.h>

int main()
{
    double a;
    double b;

    a = 1;
    b = a + 1;
    if(a == b) {
        printf("this would be
weird\n");
    }

    a = 10000000000000000.0;
    b = a + 1;
    if(a == b) {
        printf("yes, this is
perfectly normal\n");
    }
}
```

See https://en.wikipedia.org/wiki/Double-precision_floating-point_format for detailed

information.

# Character, String

A string is often implemented as an array of bytes that stores a sequence of characters, using some character encoding. String may also denote more general arrays or other sequence (or list) data types and structures.

## ASCII TABLE

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

**Figure 1. American Standard Code for Information Interchange**

# Character Encoding

```
string name = "Antonín Dvořák";
//name.Length == 14
```

## Abbreviations

```
UCS = Universal Coded Character
Set; UTF = UCS Transformation
Format
```

## UTF-8

UTF-8 encoded string occupies 17 bytes.

```
00000000: 41 6e 74 6f 6e c3 ad
6e 20 44 76 6f c5 99 c3 a1
Anton..n Dvo....
00000010: 6b
k
```

## UTF-32

UTF-32 encoded string occupies 56 bytes.

```
00000000: 00 00 00 41 00 00 00
6e 00 00 00 74 00 00 00 6f
...A...n...t...o
00000010: 00 00 00 6e 00 00 00
ed 00 00 00 6e 00 00 00 20
...n.......n...
00000020: 00 00 00 44 00 00 00
76 00 00 00 6f 00 00 01 59
...D...v...o...Y
00000030: 00 00 00 e1 00 00 00
6b
.......k...
```

- UTF-32 uses fixed four bytes
- UTF-8 uses a byte at the minimum in encoding the characters
- UTF-8 encoded file tends to be smaller
- UTF-8 is compatible with ASCII

Consider problems like sorting, error detection, length determination, and conversions between distinct encodings.

# Regular Expressions

A regular expression, regex or regexp is a string that defines a search pattern. Such patterns can be used for match or replace operations on strings. The control syntax uses a limited set of characters:
`*{}[]()^$.|*+?-` The implementations can vary, look at *Regular expressions in Java*, etc.

## Boolean "or"

```
A vertical bar separates
alternatives. apple|orange
```

## Grouping

```
Parentheses are used to define
the scope and precedence of the
operators.
For example, gray|grey and
gr(a|e)y are equivalent patterns
which both describe
the set of "gray" or "grey".
Grouping is also used to specify
```

```
and extract
specific data within the regex
match.
```

## Quantification

```
A quantifier how often the
previous element must precede to
match.
```

- `?` optional occurrence, colou?r matches both "color" and "colour", also used for greediness control
- `*` zero or more occurrence; ab*c matches "ac", "abc", "abbc", "abbbc", and so on.
- `+` at least one occurrence; ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac".
- `{n}` The preceding item is matched exactly `n` times.
- `{min,}` The preceding item is matched at least `min` times.

- `{min,max}` The preceding item is matched at least `min` times, but less than `max` times.

## Wildcards

- `^` beginning of a line (also used as a negation, see below)
- `$` end of line
- `.` match any character
- `[]` determines a group of characters, `-` interval, `^` is negation
  - `[a-z]` match any character between **a** and **z**
  - `[0123456789]` or `[0-9]` match any decimal digit
  - `[^A-Z]` match all characters except all between **A** and **Z**

## Implementation

```
The environment (programming
language, text editor) usually
defines some
```

```
specific abbreviations or
macros.
```

- `\d` decimal digit
- `\w`, `\W` word character, non-word character
- `\s`, `\S` space character, non-space character
- `\b`, `\<`, `\>` word boundary, beginning/end of a word

## Working with meta-characters

```
Meta-characters need to be
escaped if they should be
matched.
```

Regex `\\d` matches string `\d`, regex `\*\+\+\?` matches string `*++?`

# Regex Examples

Notation: example regular expressions are enclosed between two slashes /regex/.

```
/[a-z0-9_-]{3,16}/
# match a username
/[a-z0-9_-]{6,18}/
# match a password
/0x?([a-f0-9]+)/
# match a hexadecimal number
/(\d\d):(\d\d):(\d\d)/
# match a date in hh:mm:ss
format

# match a web address
/(https?:\/\/)?([\da-z\.-]+)
\.([a-z\.]{2,6})([\/\w
\.-]*)*\/?/
```

Experiment with your own regular expressions at https://regexr.com/

# Exercise

Consider a string containing multiple space separated expressions. An expression is either a single word or a compound expression consisting of multiple words between double quotes.

Example input:

```
apple orange banana "honey pie"
sun "high noon"
```

Example output:

```
all expressions: 6
compound expressions: 2
```