Exercise sheet 2 – Data representation

Goals:

- Codes: Unicode and UTF
- Number representation and formats

Exercise 2.1: Data representation (recapitulation semester 1–3)

(a) Represent the number *-16* in hex-format for an 32-bit architecture.

Proposal for solution:

```
00000000 00000000 00000000 00010000
11111111 11111111 11111111 11101111
                                 +1
-----------------------------------
11111111 11111111 11111111 11110000 = 0xFFFFFFF0
```

Exercise 2.2: Data representation: Unicode and UTF

(a) Find the unicode character U+20214 in the unicode table. Hint: With Linux Mint Mate (the VM), you can use the graphical tool „Character Map“ (Menu -> type: Character Map). Another way is to use `https://unicode-table.com` to perform the search.

(b) On which plane is unicode character U+20214?

Proposal for solution: On the plane with index 2: SIP: Supplementary Ideographic Plane

(c) Translate (encode) the unicode character U+20214 into UTF-8.

Proposal for solution:

```
2    0    2    1    4
0010 0000 0010 0001 0100
++** **** xxxx xx-- ----
=> 18 bits for code point -> 4 byte variant required!

11110000 10100000 10001000 10010100 = 0xF0A08894
     _++   ******   xxxxxx   ------
```

(d) Translate (encode) the unicode character U+20214 into UTF-16.

Proposal for solution:

```
2    0    2    1    4
0010 0000 0010 0001 0100
=> 18 bits for code point -> 4 byte variant required!

correction:
0x20214 - 0x10000 = 0x10214

1    0    2    1    4
```

```
0001 0000 0010 0001 0100
xxxx xxxx xx-- ---- ----

HS (high surrogate)  LS (low surrogate)
11011000 01000000    11011110 00010100
      xx xxxxxxxx          -- --------
0xD840               0xDE14

=> 0xD840DE14
```

(e) Translate (encode) the unicode character U+20214 into UTF-32.

Proposal for solution:

```
Nothing to convert or correct:
0    0    0    2    0    2    1    4
0000 0000 0000 0010 0000 0010 0001 0100

=> 0x00020214
```

Exercise 2.3: Number representation (theoretical)

Given are the decimal numbers 50.5 and 0.80.
Hint: Represent the numbers initially as binary fraction. Use appropriate scaling.

(a) State the bit pattern for the **binary fixed point** format. *Hint: You may define the position of the fixed point.*

Proposal for solution:
16 bit; point on bit 8:

```
50.5  00110010.10000000
 0.8  00000000.11001100...
```

(b) State the bit pattern for the **decimal fixed point** format and add them. *Hint: You may represent each digit with BCD.*

Proposal for solution:
16 bit; point on bit 8:

```
50.5  01010000.01010000
 0.8  00000000.10000000

 01010000.01010000
+00000000.10000000
------------------
 01010000.11010000 (BCD only from 0-9 => 13-10=3)
 01010001.00110000
```

(c) State the bit pattern for the **binary floating point** format.

Proposal for solution:

```
50.5: c = e + 127 =  5 + 127 = 132 -> 0|10000100|10010100000000000000000|
 0.8: c = e + 127 = -1 + 127 = 126 -> 0|01111110|10011001100110011001100|
                                                    ----
```

(d) List some pros and cons of **fixed point** numbers: **binary fixed point** vs. **decimal fixed point**. *Hint: You may think about performance, cost, accuracy, programming language support, ….*

Proposal for solution:

binary fixed point	**decimal fixed point**
+ low cost (it's just an imaginary point)	+ exact visualisation of decimal fractions (e.g. 0.8
+ performance compared to software-emulation	+ arbitrary number of digits (BCD)
+ good for measured values (eg. current, voltage)	- memory consumption
- restricted accuracy (it's still binary)	- performance
- programming language support	0 programming language support

Exercise 2.4: Binary floating point number (coding)

(a) Write a C program with a for loop (from 1 to 500). In every loop it adds 0.8 to a `float` variable. At the end, print the result with a precision of 7 digits.
Use `RA_exercises/sheet_02/binary_floating_point/binary_floating_point.c` as a starting template.

Proposal for solution:

```c
#include <stdlib.h> //EXIT_SUCCESS
#include <stdio.h>  //printf

int main()
{
    float inc = 0.8f;
    float val = 0.0f;

    for(int i = 1; i <= 500; ++i){
        val += inc;
    }

    printf("%.7f\n", val);

    return EXIT_SUCCESS;
}
```

(b) Compile and run the program with:

```
cd RA_exercises/sheet_02/binary_floating_point
make
./program
```

(c) What is the result and what have you expected?

Proposal for solution:
Expected Result: 400
Actual Result: 399.9984741

(d) Explain the behaviour.

Proposal for solution: Lost of precision due to usage of `float`. Also 0.8 is inaccurate and with += the errors accumulate.

(e) Use a `double` instead of the `float`. What do you observe?

Proposal for solution: The result is more precise, but the accumulation of the error is still visible.

Exercise 2.5: Binary fixed point number (coding)

We use the *Compositional Numeric Library (CNL)* library from `https://github.com/johnmcfarlane/cnl`. *Hint: The template contains already the library and the build is pre-configured within the* ***`Makefile`****.*

(a) Write a C++ program with a for loop (from 1 to 500). In every loop it adds 0.8 to a binary fixed point variable. At the end, print the result.
Use `RA_exercises/sheet_02/binary_fixed_point/binary_fixed_point.cpp` as a starting template.

(b) Follow the TODOs in `binary_fixed_point.c`.

Proposal for solution:

```cpp
#include <cstdlib>    //EXIT_SUCCESS
#include <iostream>   //std::cout
#include <cinttypes>  //int32_t
#include "cnl/include/cnl/all.h"

int main()
{
    cnl::fixed_point<int32_t, -7> inc{0.8};
    cnl::fixed_point<int32_t, -7> val{0.0};

    for(int i = 1; i <= 500; ++i){
        val += inc;
    }

    std::cout << val << std::endl;

    return EXIT_SUCCESS;
}
```

(c) Compile your program using the provided Makefile and run it.

Proposal for solution:

```bash
cd RA_exercises/sheet_02/binary_fixed_point
make
./numbers
```

(d) What is the result and what have you expected?

Proposal for solution: The result is 398.4375. Obviously the number is fixed to 7 digits as expected. The disadvantage is the loss in precision.

(e) What happens if you change the precision from 7 digits to 14 digits?

Proposal for solution: The result is 399.993896484375. Its precision is now higher, but it is still a fixed point format with limited precision that depends on the invested precision bits, and don't forget, the digits after the comma are represented by $2^{-1}, 2^{-2}, ..., 2^{-N}$ which can't accurately represent all fractions.

Exercise 2.6: Decimal fixed point number (coding)

We use the *Decimal data type for C++* library from `https://github.com/vpiotr/decimal_for_cpp`. *Hint: The template contains already the library and the build is pre-configured within the* ***`Makefile`****.*

(a) Write a C++ program with a for loop (from 1 to 500). In every loop it adds 0.8 to a decimal fixed point variable. At the end, print the result.
Use `RA_exercises/sheet_02/decimal_fixed_point/decimal_fixed_point.c` as a starting template.

(b) Follow the TODOs in `decimal_fixed_point_solution.c`.

Proposal for solution:

```cpp
#include <cstdlib>    //EXIT_SUCCESS
#include <iostream>   //std::cout
#include "decimal_for_cpp/include/decimal.h"

int main(void)
{
    dec::decimal<1> val(0.0);
    dec::decimal<1> inc(0.8);

    for(int i = 1; i <= 500; ++i){
        val += inc;
    }

    std::cout << val << std::endl;

    return EXIT_SUCCESS;
}
```

(c) Compile your program using the provided Makefile and run it.

Proposal for solution:

```bash
cd RA_exercises/sheet_02/decimal_fixed_point
make
./program
```

(d) What is the result and what have you expected?

Proposal for solution: The result is 400.0. Exactly what we expected!