[System Programming] Appendix of Assignment #1

Spring 2025

Seulki Lee
Yunseok Lee
Kyu Hwan Lee
Jiwoon Chang
Yeojin Lee

CONTACT
Ulsan National Institute of Science and Technology
Address 50 UNIST-gil, Ulju-gun, Ulsan, 44919, Korea
Tel. +82 52 217 0114 **Web.** www.unist.ac.kr

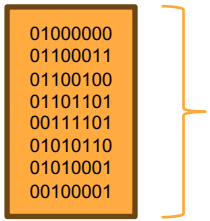
Computer Science and Engineering
106 3rd Engineering Bldg
Tel. +82 52 217 6333 **Web.** https://cse.unist.ac.kr/

Assignment #1

Overview of Storing Data and Printing Output

- Clarifying Output Format
  - The input stored in big endian format is converted into little endian format for memory storage, and then outputted according to each data type.
- Explanation Using an Example
  - The first line of the input example, "01000000011000110110010001101101001111010101011001010000100100001" , will be used to illustrate the process.

Input.txt

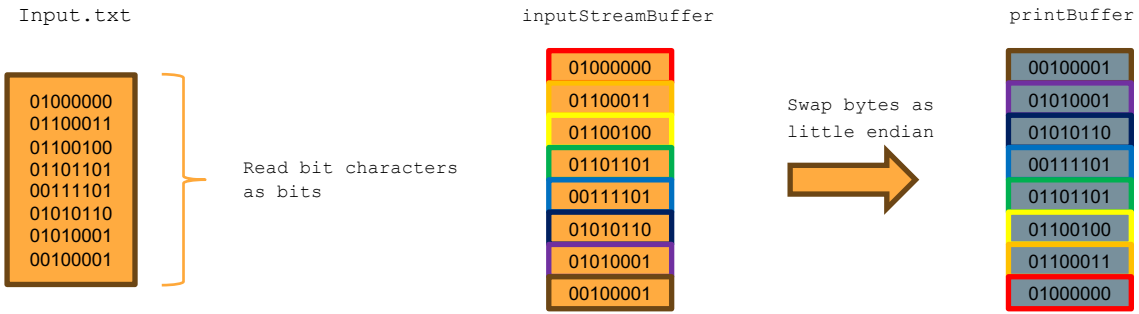


Input Processing

- Converting Bit Characters to Bytes
  - Group bit characters into 8-bit (1-byte) units and convert them to an unsigned char array.
  - Example: "01000000" → 0x40 (ASCII for '@').
  - Converted array example: "40 63 64 6d 3d 56 51 21".

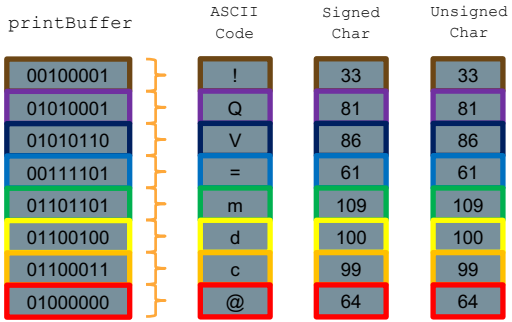
Byte Array Order Transformation

- Order Transformation Before Copying to Final Buffer
  - Bytes in inputStreamBuffer are reversed in order before being copied to the printBuffer.
  - Final memory storage format: { 21 51 56 3d 6d 64 63 40 }.



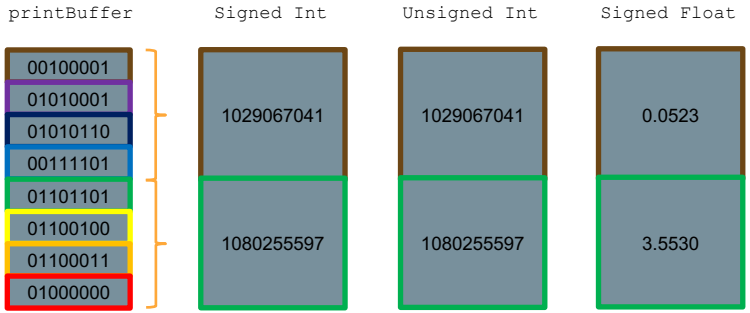
Output Process - ASCII and Characters

- ASCII Code Output
  - Bytes in printBuffer are interpreted and output as ASCII characters.
  - Valid ASCII characters are printed as is, others are printed as '.'.
- Signed Char Output
  - Bytes are interpreted as signed char and output as decimal numbers.
- Unsigned Char Output
  - Bytes are interpreted as unsigned char and output as decimal numbers.



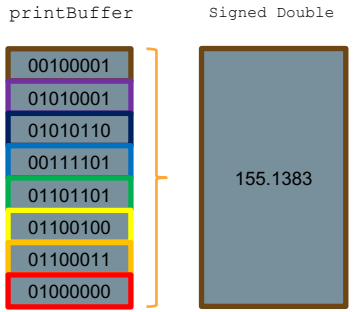
Output Process - Integers and Floating Points

- Signed and Unsigned Int Output
  - Bytes are grouped into 4-byte units and interpreted/output as signed and unsigned int.
  - Examples: 1029067041 (0x3d565121), 108025559 (0x4063646d).
- Signed Float Output
  - 4-byte units are interpreted and output as float.
  - Examples: 0.0523, 3.5530.



Output Process - Double

- Signed Double Output
  - 8-byte units are interpreted and output as doubles.
  - Example: 155.1383 (0x4063646d3d565121).

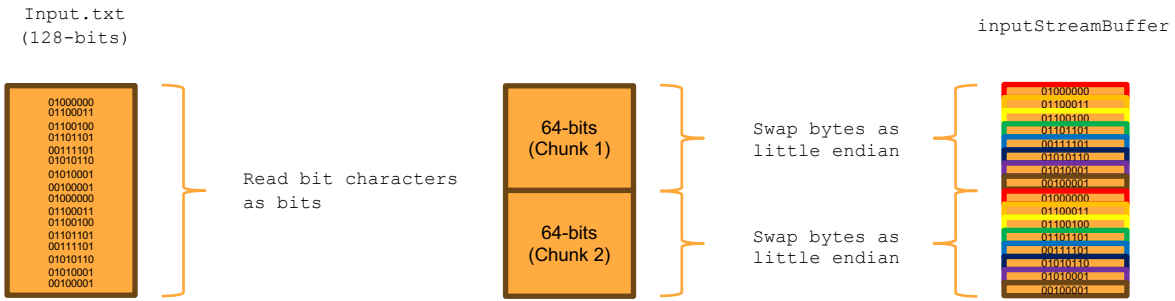


Details on Little Endian Conversion

- Little Endian Conversion in 64-bit Units
  - The conversion from big endian to little endian occurs in 64-bit chunks.
- Example with 128-bit Input
  - A 128-bit input is divided into two 64-bit chunks.

Details on Little Endian Conversion

- Byte Order Transformation Within Each Chunk
  - Within each chunk, the order of bytes is reversed for little endian conversion, as illustrated in previous slides. However, the order between the chunks does not change.
- Maintaining Chunk Order
  - Although the byte order within each 64-bit chunk is reversed due to little endian conversion, the sequence of Chunk 1 and Chunk 2 remains the same. This is because little endian conversion affects the byte order within specific-sized formats but does not alter the order of these formats relative to each other.



How to test with *random_hex_generator*

- Running the Command
  - Execute the *random_hex_generator* tool by using the command `./random_hex_generator_[platform]`, where `[platform]` can be `arm64_mac`, `x86-64_linux`, or `x86-64_mac`.
- Utilizing Output for Input
  - Use the last line's content, starting with "Bits:", from the output of random_hex_generator as input for your assignment program.

```
(base) TA@ubuntu Assignment1 % ./random_hex_generator_arm64_mac
Random 8-byte Hex Value: 3f7967463b6b4538
Float (Front 4 Bytes): 0.003590
Float (Back 4 Bytes): 0.974232
Double: 0.006202
Integer (Front 4 Bytes): 996885816
Integer (Back 4 Bytes): 1064920902
ASCII: 8 E k ; F g y ?
Bits: 00111111011110010110011101000110001110110110101101000010100111000
```

Preparing the Input File

- Collecting Bit Contents
  - Run the *random_hex_generator* once or multiple times to collect various "Bits" content and save these in an *input.txt* file.
- Executing the Assignment Program
  - Run your assignment program using the prepared *input.txt* as the input source.

```
(base) TA@ubuntu Assignment1 % ./random_hex_generator_arm64_mac
Random 8-byte Hex Value: 3f7967463b6b4538
Float (Front 4 Bytes): 0.003590
Float (Back 4 Bytes): 0.974232
Double: 0.006202
Integer (Front 4 Bytes): 996885816
Integer (Back 4 Bytes): 1064920902
ASCII: 8 E k ; F g y ?
Bits: 0011111101111001011001110100011000111011011010110100010100111000


(base) TA@ubuntu Assignment1 % ./random_hex_generator_arm64_mac
Random 8-byte Hex Value: 3f277c6e395a3a26
Float (Front 4 Bytes): 0.000208
Float (Back 4 Bytes): 0.654242
Double: 0.000179
Integer (Front 4 Bytes): 962214438
Integer (Back 4 Bytes): 1059552366
ASCII: & : Z 9 n | ' ?
Bits: 0011111100100111011111000110111000111001010110100011101000100110
```

```
(base) TA@ubuntu Assignment1 % vi input.txt
~
~
~
~
~
~
~
"input.txt" [noeol] 4L, 643B
```

```
0011111101111001011001110100011000111011011010110100010100111000
0011111100100111011111000110111000111001010110100011101000100110~
~
~
~
~
~
"input.txt" [noeol] 4L, 643B
```

```
0011111101111001011001110100011000111011011010110100010100111000
0011111100100111011111000110111000111001010110100011101000100110~
~
~
~
~
~
:wq
```

Comparing Results and Debugging

- Analyzing Program Output
  - Compare the results of your assignment program with the output of the *random_hex_generator*.
  - Note that the *random_hex_generator* does not limit decimal places, so consider rounding off the results beyond 4 digits after the decimal point.
- Debugging with a Debugger
  - You can debug your assignment program by comparing the hex values loaded into memory with the "Random 8-byte Hex Value:" from the *random_hex_generator* using debuggers like *gdb* or *lldb*.