Text Encoding

How computers handle text

Eric Zhao

October 29, 2022

It’s just text, right?

A memory refresher

Bits and bytes

Computers store information in bits (0 or 1 / TRUE or FALSE)

...	1	1	0	1	0	1	1	0	...

Table 1: Memory sketch

Bits and bytes

We can do binary logic and arithmetic with these:

a	b	op	result
0	1	AND	0
0	1	OR	1
1	1	NAND	0
0	0	NOR	1
1	1	XOR	0

Table 2: Logic binop inputs and results

Bits and bytes

A **byte** is equal to 8 bits

This means possible values are in the interval $[0_{10}, 255_{10}]$

1	0	0	1	1	0	1	0

Table 3: A byte representing the value of 154_{10}, or 0x9A

We can (and will) use 2 hexadecimal (base 16) numbers to represent a single byte

Bits and bytes

type	description	bits	bytes
boolean	true / false flag	1	1[1]
char	16-bit character	16	2
short	16-bit integer	16	2
int	32-bit integer	32	4
long	64-bit integer	64	8
float	32-bit floating-point	32	4
double	64-bit floating-point	64	8

Table 4: Sizes of Java data types

[1]In Java, booleans usually take up a byte for speed

Big and little endianness

MSB → Most Significant Byte
LSB → Least Significant Byte

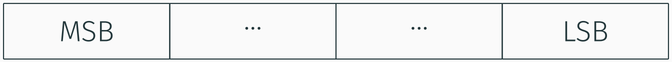


Figure 1: Most and least significant bytes in a 4-byte word

Big and little endianness

168496141_{10} = 0x0A0B0C0D

0A (MSB)	0B	0C	0D (LSB)

Figure 2: Most and least significant bytes in a 0x0A0B0C0D

Big and little endianness

168496141_{10} = 0x0A0B0C0D

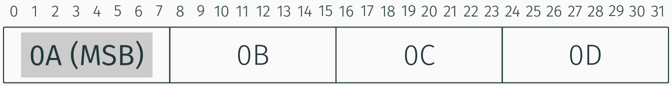


Figure 3: Big endian memory layout

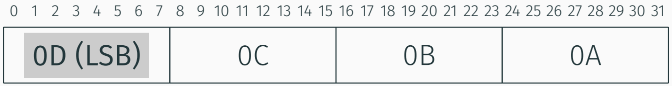


Figure 4: Little endian memory layout

Big and little endianness

Endianness – the order in which memory is written and read

Big endianness – most significant byte to least significant byte

Little endianness – least significant byte to most significant byte

So what about text?

We need to convert strings of text into numbers or sequences of numbers

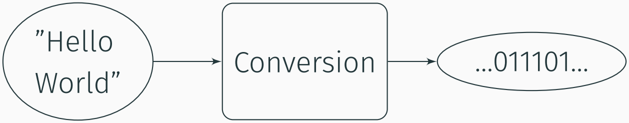


Figure 5: Sketch of text memory storage

ASCII

Character set vs. character encoding

Character set – collection of characters in a written language

Coded character set – function that maps characters to code points

Code point – a value denoting a character

Character encoding – mapping of code points to bytes[2]

[2]More precisely, there's the character encoding form and the character encoding sequence

1963 – ASCII

ASCII – American Standard Code for Information Interchange

Now referred to officially as US-ASCII by the IANA

1963 – ASCII

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	16	10	DLE	32	20		48	30	0
1	01	SOH	17	11	DC1	33	21	!	49	31	1
2	02	STX	18	12	DC2	34	22	”	50	32	2
3	03	ETX	19	13	DC3	35	23	#	51	33	3
4	04	EOT	20	14	DC4	36	24	$	52	34	4
5	05	ENQ	21	15	NAK	37	25	%	53	35	5
6	06	ACK	22	16	SYN	38	26	&	54	36	6
7	07	BEL	23	17	ETB	39	27	’	55	37	7
8	08	BS	24	18	CAN	40	28	(	56	38	8
9	09	HT	25	19	EM	41	29	)	57	39	9
10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<
13	0D	CR	29	1D	GS	45	2D	-	61	3D	=
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>
15	0F	SI	31	1F	US	47	2F	/	63	3F	?

Table 5: ASCII table, produced with `ascii | tail -17`

1963 – ASCII

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
64	40	@	80	50	P	96	60	‘	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	|
77	4D	M	93	5D	]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

Table 6: ASCII table, cont.

7-bit ASCII encoding

128 characters → 2^7 → 7 bits

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	1	0	0	0	1	1	0	1	0	0	1

Figure 6: 7-bit ASCII encoding for "Hi"

8-bit ASCII encoding

8-bit encoding → 1 parity or empty bit + 7 bits

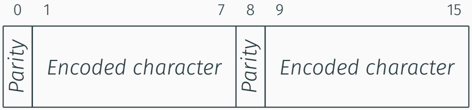


Figure 7: 8-bit ASCII encoding for two characters

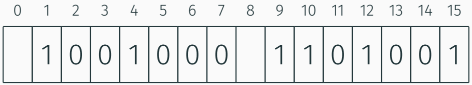


Figure 8: 8-bit ASCII encoding for “Hi”

Writing an ASCII decoder

```rust
// Pretend that the imported function from_codepoint()
// converts decimal code point values to their string
// representation.
use table::from_codepoint;

fn decode_ascii(memory: &[u8]) -> String {
    // Convert each codepoint to its corresponding ASCII
    // character.
    let codepoints = memory.iter().map(|n| from_codepoint(*n));

    // Collect each ASCII character string to one string.
    let string: String = codepoints.collect();

    // Return the string.
    string
}
```

What about all the other characters out there?

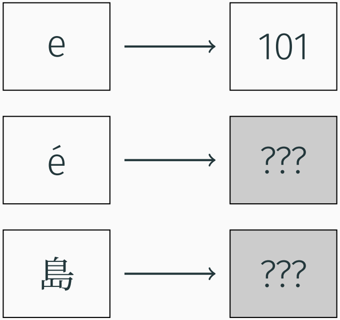


Figure 9: What the heck do we do about these?

Extended ASCII character sets

ISO/IEC 8859–1 – ASCII + characters sufficient for most western European languages
ISO/IEC 8859–2 – like ISO 8859–1, but for *eastern* European languges

char	À	Á	Ç	È	ä	æ	ñ	ø
dec	192	193	199	200	228	230	241	248
hex	C0	C1	C7	C8	E4	E6	F1	F8

Table 7: Select characters from ISO 8859–1

Multi-byte extended ASCII encodings

Shift JIS – Shift Japanese Industrial Standards, for encoding the Japanese language
UTF-8 – Unicode Transformation Set – 8-bit

Unicode

The problems with ASCII

Let’s review:

- Minimal language support
- Variants that are widespread but incompatible
- Too naïve for many types of languages

Unicode

Unicode – standard for consistent digital text handling across the world

Unicode Consortium – non-profit organization designated to maintain and publish the Unicode standard

Problem: character coverage

The Universal Character Set

Universal Character Set – character set that maps characters for almost all modern natural scripts, mathematics, music, etc.

- 1,112,064 possible codepoints
- Superset of ISO 8859-1 (extended ASCII with Latin-1 supplement)

The seventeen Unicode planes

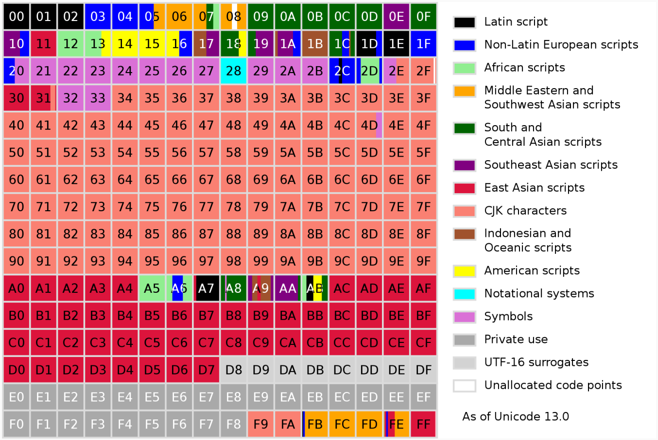


Figure 10: Basic Multilingual Plane

The seventeen Unicode planes

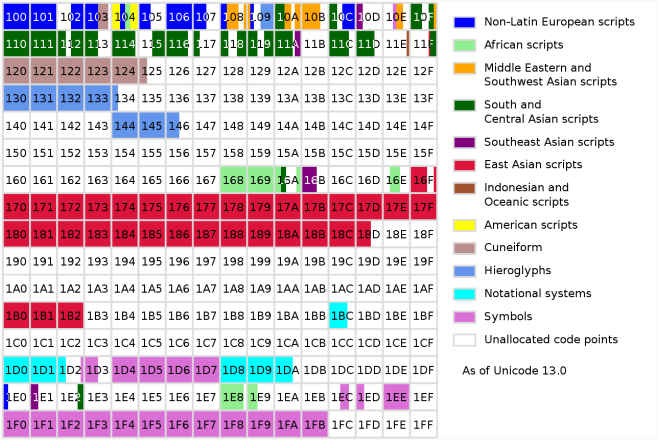


Figure 11: Supplementary Multilingual Plane

Problem: what *is* a character, anyway?

Characters: not so simple

If we continuously hit backspace[3]

… the shape of letters change.

الأبجدية العربية
الأبجدية العربي
الأبجدية العرب
الأبجدية العر
الأبجدية الع
الأبجدية ال
الأبجدية ا
الأبجدية

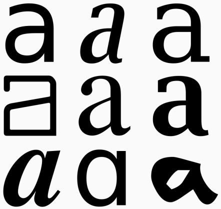
[3]Arabic is written right-to-left (deletion begins from the left)

Graphemes, not glyphs

Unicode maps codepoints to **graphemes** instead of **glyphs**

Grapheme – smallest functional unit of writing, represents an abstract concept (data)

Glyph – visual representation of a readable symbol (display)



Graphemes, not glyphs

μ (U+03BC)

Greek Small Letter Mu

Block: Greek and Coptic

μ

μ (U+00B5)

Micro Sign

Block: Latin-1 Supplement

µ

Why does this matter?

If we continuously hit $\boxed{\texttt{backspace}^{4}}$

… the shape of letters change.

الأبجدية العربية
الأبجدية العربي
الأبجدية العرب
الأبجدية العر
الأبجدية الع
الأبجدية ال
الأبجدية ا
الأبجدية

[4] Arabic is written right-to-left (deletion begins from the left)

Fonts: What stuff looks like

Fonts define how to render the characters

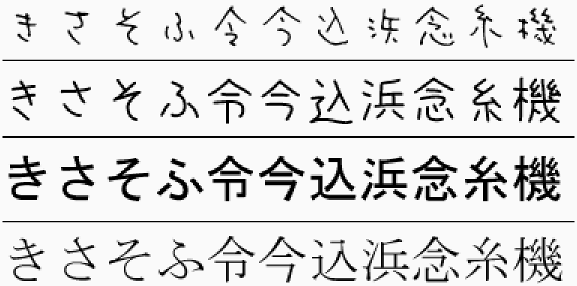


Figure 12: Japanese font comparison

Composing graphemes

Grapheme cluster – a sequence of multiple codepoints that together may represent a user-perceived character

NFC character	A	m	é		l	i	e
NFC codepoint	0041	006d	00e9		006c	0069	0065
NFD codepoint	0041	006d	0065	0301	006c	0069	0065
NFD character	A	m	e	◌́	l	i	e

Table 8: *Amélie* with two equivalent Unicode forms

characters == graphemes

UTF-8

Unicode encodings

The Unicode standard defines a number of encoding formats:

UTF-8 – 8-bit Unicode Transformation Format

UTF-16LE – 16-bit Unicode Transformation Format LE

UTF-16BE – UTF-16LE, but big-endian

UTF-32LE – 32-bit Unicode Transformation Format LE

UTF-32BE – UTF-32LE, but big-endian

How much space?

21 bits is enough for 2097152 possible codepoints:

$$2^{20} = 1048576 < \textbf{1112064 codepoints} < 2097152 = 2^{21}$$

Thus, 3 bytes (24 bits) minimum.

UTF-8 uses **4 bytes**.

Examining UTF-8 encoding

Considerations:

- ASCII compatibility
- Space efficiency (only use the number of bytes a coodepoint really needs)

Examining UTF-8 encoding

UTF-8 is a **fixed-order variable-width** encoding

Each codepoint is stored in **1 – 4 bytes**

Character	Codepoint	Bytes required
$	0x0024	1
¢	0x00A2	2
€	0x20AC	3
⍰	0x10348	4

Table 9: Number of bytes required to store certain characters

Examining UTF-8 encoding

This table illustrates how bits are structured in the encoding:

#	Range	Byte 1	Byte 2	Byte 3	Byte 4
1	U+0000–U+007F	0xxxxxxx			
2	U+0080–U+07FF	110xxxxx	10xxxxxx		
3	U+0800–U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	U+10000–U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 10: Layout of UTF-8 byte sequences

Examining UTF-8 encoding

Char	Hex	Binary	UTF-8
$	0x0024	010 0100	00100100
¢	0x00A2	000 1010 0010	11000010 10100010
€	0x20AC	0010 0000 1010 1100	11100010 10000010 10101100
?	0x10348	0 0001 0000 0011 0100 1000	11110000 10010000 10001101 10001000

Table 11: Representation of UTF-8 characters

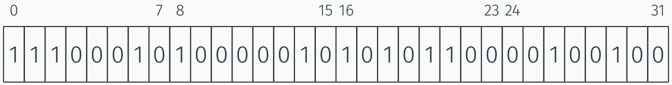


Figure 13: UTF-8 encoding of “€ $”

Writing a UTF-8 decoder

Let’s write a UTF-8 decoder in C++

We’ll convert a vector of bytes into Unicode codepoints

Our header file:

```
// Type alias for unsigned 8-bit integer.
using byte = unsigned char;
// Type alias for unsigned 32-bit integer.
using codepoint = unsigned int;
```

Writing a UTF-8 decoder

Our function signatures:

```
#include <optional>
#include <vector>

std::optional<codepoint>
next_codepoint(std::vector<byte>::const_iterator &iter,
               const std::vector<byte>::const_iterator &end);

std::vector<codepoint> decode(const std::vector<byte> &bytes);
```

Writing a UTF-8 decoder

Byte mask constants we'll use to help process each byte:

```
const byte B2_MASK = 0x1F; // 0001 1111
const byte B3_MASK = 0x0F; // 0000 1111
const byte B4_MASK = 0x07; // 0000 0111
const byte MB_MASK = 0x3F; // 0011 1111
```

Writing a UTF-8 decoder

Check if the iterator has reached the end; if so, return nothing:

```
std::optional<codepoint>
next_codepoint(std::vector<byte>::const_iterator &iter,
               const std::vector<byte>::const_iterator &end) {
  if (iter == end) { // Check if iterator has reached end;
    return {};       // return nothing.
  }
```

Writing a UTF-8 decoder

Handle the 1-byte case:

```
  auto b0 = *iter; // Get next byte from iterator.

  // Decode 1 byte case
  if (b0 < 128) {
    iter++;               // Consume this byte from iterator.
    return (codepoint)b0; // Convert byte value to 32-bit int.
  }
```

Writing a UTF-8 decoder

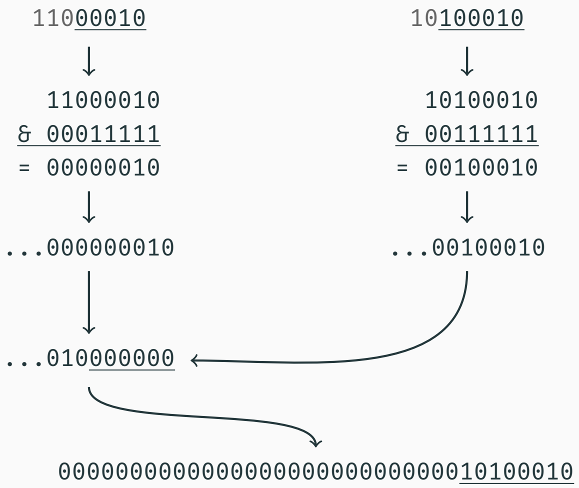
Handle the 2-byte case:

```
  if (++iter == end) { // Check if iterator reached end;
    return {};         // return nothing.
  }
  auto b1 = *iter; // Get next byte.

  if (b0 < 0xE0) { // Check if within range for 2-byte case.
    iter++;        // Consume this byte from iterator.

    // Decode the 2 bytes and return a int value.
    codepoint acc = ((codepoint)(b0 & B2_MASK)) << 6;
    return acc | (codepoint)(b1 & MB_MASK);
  }
```

Writing a UTF-8 decoder



Writing a UTF-8 decoder

Handle the 3-byte case:

```
if (++iter == end) {
  return {};
}
auto b2 = *iter;

if (b0 < 0xF0) { // Check if within range for 3-byte case.
  iter++;

  // Decode the 3 bytes and return an int value.
  codepoint acc = ((codepoint)(b0 & B3_MASK)) << 12;
  acc = acc | ((codepoint)(b1 & MB_MASK) << 6);
  return acc | (codepoint)(b2 & MB_MASK);
}
```

Writing a UTF-8 decoder

Handle the 4-byte case:

```
  // Decode 4 byte case
  if (++iter == end) {
    return {};
  }
  auto b3 = *iter;

  // Assume first byte must start with 11110.
  iter++;

  // Decode 4 bytes and return an int value.
  codepoint acc = ((codepoint)(b0 & B4_MASK)) << 18;
  acc = acc | ((codepoint)(b1 & MB_MASK) << 12);
  acc = acc | ((codepoint)(b2 & MB_MASK) << 6);
  return acc | (codepoint)(b3 & MB_MASK);
}
```

Writing a UTF-8 decoder

Implementing `decode`:

```
// Pass in a vector of bytes to decode into codepoints.
std::vector<codepoint> decode(const std::vector<byte> &bytes) {
  // Initialize iterators pointing to start and end of bytes.
  auto iter = bytes.cbegin();
  auto end = bytes.cend();

  // Decode codepoints until there are no more bytes.
  std::vector<codepoint> codepoints = {};
  while (iter != end) {
    auto cp = next_codepoint(iter, end);
    codepoints.push_back(*cp);
  }

  return codepoints;
}
```

Writing a UTF-8 decoder

```
std::vector<unsigned int> test() {
  std::string str(u8"¢€한𐍈");
  std::vector<unsigned char> bytes(str.begin(), str.end());

  return decode(bytes);
}
```

The result:

```
[1]   162  8364 54620 66376
```

Try yourself: UTF-8 encoder

Implement a function `encode` that converts Unicode codepoints into UTF-8-encoded bytes:

```
// Java
int[] encode(int[] codepoints) {
  ...
}

# Python
def encode(codepoints: list[int]) -> list[int]:
  ...

// C++
std::vector<unsigned char>
encode(const std::vector<unsigned int> &codepoints) {
  ...
}
```

How to be Unicode-aware?

Understand your language

Consider:

- What is the size of the character data type?
- Do their values match Unicode’s codepoints?
- Are they stored internally as UTF-8 or some other encoding?
- How are strings stored?
- Do I need an external library?

Indexing / slicing strings

Limitation: We cannot index or slice strings

0	8	16	24	32	40
笑 (U+7B11)			w (U+0077)	¢ (U+00A2)	
E7	AC	91	77	C2	A2

Figure 14: UTF-8 for “笑 w¢”

What would we get if we tried to index the string?

```
let s = "笑w¢";
s[1]
s[1:2]
```

Iterating through strings

Iterating through strings is non-trivial.

What’s being printed?

```
for ch in "笑w¢" {
  print(ch + "\n");
}
```

Byte values?

Codepoint values?

Graphemes?

Grapheme clusters?

Category / property awareness

Many operations may require **awareness of Unicode character categories or properties**.

Unicode regular expression search and replace in ES6 JavaScript:

```
// Use the `u` flag to support Unicode in regular expression.
const regexp = /\p{Nd}/gu;

// "２" is the "Fullwidth Digit Two (U+FF12)".
const str = "Hello 07 and ２d."

str.replace(regexp, "");
// "Hello  and d."
```

Canonical equivalence

Some operations may require awareness of canonical equivalence.

```
let a = "Amélie";
let b = "Ame◌́lie";    // No space in here
assert!(a == b)       // Does this assertion pass or fail?
```

Takeaways

Text is complicated

Because natural language is complicated, the systems we develop to handle it are also complicated.

Text processing is non-trivial

Operations that may seem simple are often far more complex.

Unicode matters

All modern programs should support text input in any language—all modern programs should support Unicode.

Thanks!

Questions?

To get these slides:
`https://github.com/Dophin2009/textenc`