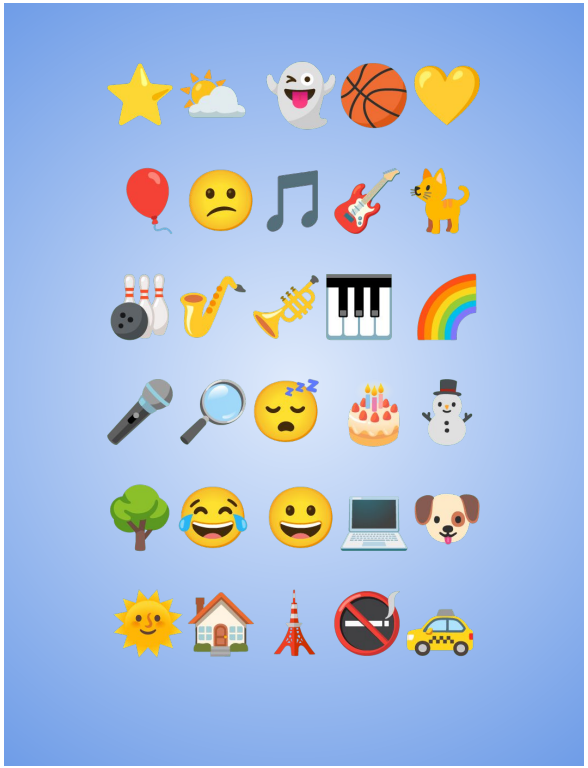
COMP1521 26T1

Week 8 Lecture 1

Files and Text Encoding and Unicode

Today's Lecture

- Files
  - Recap
  - Useful file functions
- Representing Text
  - ASCII
  - Unicode
    - UTF8 encoding



Recap Exercise: writing bytes

Assume I have opened 2 files for writing and have FILE * f1 and f2 variables.

- What would the following write to the files on our system? What would the return values of fwrite be?

```
uint8_t data = {0xAB, 0xCD};
size_t n = fwrite(&x, 1, 2, f1);

uint16_t x = 0xABCD;
size_t n = fwrite(&x, 2, 1, f2);
```

Recap Exercise: writing bytes

Assume I have a opened another file for writing and have a FILE * f3 variable.

- What would this code print to the file? Is this portable?

```
uint16_t x = 0xABCD;

uint8_t low_byte = x & 0xFF;
uint8_t high_byte = (x >> 8);
fputc(low_byte, f3);
fputc(high_byte, f3);
```

Recap Exercise: fseek

Write a program to print out the middle byte of a file.

Recap Exercise: Metadata and stat

If I ran the following on the command line:

```
chmod 755 f
```

A. Would “others” have the execute permission set for the file f?

B. How could I check this from my C code?

Making a directory

```
int mkdir(const char *pathname, mode_t mode);
```

returns 0 if successful, returns **-1** and sets **errno** otherwise

- for example: `mkdir("newDir", 0755)`

if **pathname** is e.g. `a/b/c/d`

- all of the directories `a`, `b` and `c` must exist
- directory `c` must be writable to the caller
- directory `d` must not already exist

the new directory contains two initial entries

- `.` is a reference to itself
- `..` is a reference to its parent directory

Demo: mkdir.c

Opening and Reading directories

```
// open a directory stream for directory name
DIR *opendir(const char *name);

// return a pointer to next directory entry
struct dirent *readdir(DIR *dirp);

// close a directory stream
int closedir(DIR *dirp);
```

Found in man 3

Demo list_directory.c

Useful Linux (POSIX) functions

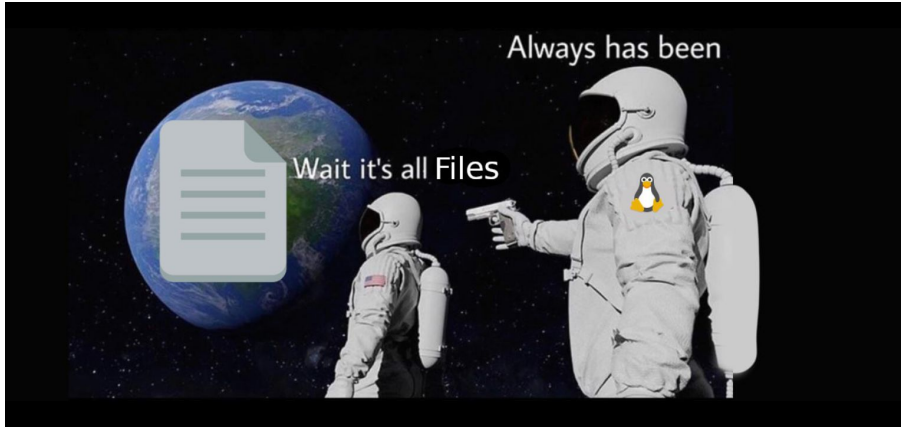
```
chmod(char *pathname, mode_t mode) // change permission of file/...
unlink(char *pathname) // remove a file...
rename(char *oldpath, char *newpath) // rename a file/directory
chdir(char *path) // change current working directory
getcwd(char *buf, size_t size) // get current working directory
link(char *oldpath, char *newpath) // create hard link to a file
symlink(char *target, char *linkpath) // create a symbolic link
```

Demo: chmod.c rm.c rename.c my_cd.c getcwd.c nest_directories.c many_links.c chain_links.c

Everything is a File

Originally the file system only managed data stored on a magnetic disk.
Unix philosophy is: **Everything is a File**
File system used to access:

- files
- directories (folders)
- storage devices (disks, SSD, ...)
- peripherals (keyboard, mouse, USB, ...)
- system information
- inter-process communication
- network



Text Representation

How should we represent text?

- We know how to represent unsigned integers, signed integers and real values in C.
- Text is arguably the most important data type
  - It can represent all other data types via serialization
    - E.g. JSON, XML, YAML, etc…
    - Advantage: no endianness issues like binary formats
- Text is sequences of characters
- So how can we represent characters?

So, how should we represent characters?

- By default in C and MIPS we have used ASCII
- Modern computers use something called “UNICODE” to represent the individual characters!
- But other things came before…
- Disclaimer: this timeline I am about to show is very **Western-centric**.
    - There are many other encoding schemes from around the world

A timeline of character representations

- 1828: First electronic Telegraph system (Pavel Schilling)
- 1837: Cooke and Wheatstone Telegraph
- 1844: Morse Code
- 1897: First radio transmission

many other encoding schemes that we won't cover

- 1943: First (modern) computer (Colossus)
- 1963: **ASCII**
- 1970s: **Extended ASCII**
- 1963: EBCDIC
- 1987: **Unicode**

ASCII: 1963

- **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
  - created by the American Standards Association (ASA)
  - later became the American National Standards Institute (ANSI)
    - the first organization to standardize the C programming language
- 7-bit (fixed-size) encoding
  - 128 possible values
  - all of the values are used
- One of the most common and influential encodings in computing

ASCII: Control Characters

- When ASCII was created, computers didn’t use monitors.
- Instead, they used teletypes
  - Electromechanical devices with a keyboard for input and a printer for output.
  - These could be controlled by a human (typing) or by a computer (printing).
- ASCII included control characters to
  - move the carriage
  - start a new line
  - ring the bell

ASCII: TTY (Teletypewriter)

ASCII:

USASCII code chart

Bits b_7 b_6 b_5 →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 ↓ Row / Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

ASCII Overview

- Uses values in the range `0x00` to `0x7F` (0..127)
- Characters partitioned into sequential blocks (sticks)
  - control characters (sticks 0 and 1) (codes 0x00 to 0x1F)
    - e.g. '\0', '\n'
  - Punctuation (stick 2, parts of sticks 3..7)
  - digits (stick 3) (codes 0x30-0x39)
    - e.g. '0'..'9'
- upper case alphabetic (65..90) \... 'A'..'Z'
- lower case alphabetic (97..122) \... 'a'..'z'

ASCII Patterns

- Sequential nature of groups allows for helpful things like
  - Converting character digits into integers
    - ‘4’ - ‘0’ gives us the integer 4
  - Iterating through the alphabet, comparing letters
    - ‘a’ + 1 gives us ‘b’ and also ‘a’ < ‘b’
  - Case conversion
    - ‘A’ + 32 gives us ‘a’
  - Some patterns are not so helpful…
    - ‘<’ + 2 gives ‘>’
    - ‘[‘ + 2 gives ‘]’
    - ‘{‘ + 2 gives ‘}’
    - ‘(‘ + 2 gives ‘*’

ASCII: Bit Patterns

- The digits have values of **0b011** followed by the digits binary value
  - Allows for fast conversion between ASCII and binary numbers
- Uppercase and Lowercase letters are placed such that:
  - the only difference between them is the **5th** bit
  - this allows for very fast case conversion and case insensitive string comparison

ASCII Demo

- ASCII_to_DEC.c
  - Convert from ascii character digit to a numeric decimal digit
- ASCII_case_insensitive.c
  - Convert to and from upper case and lower case characters

ASCII Limitations

- ASCII works well for English (American English)
- And is fairly decent for British English.
  - Unless you use the pound sign (£)
- But it doesn’t work well for other european languages
  - And doesn’t work at all for other languages (e.g Asian languages).
- The solution (for other European languages at least) was to use the 8th bit to extend the encoding.

Extended ASCII

EASCII is not standardized! So there are many different encodings

- All legitimate “Extended ASCII”
- KOI-8: Russian encoding
- ISO 8859-1 (aka Latin-1): Western European encoding
- Code page 899: DOS mathematical
- symbols etc…

(wikipedia lists 100s of different Code Pages)

This made EASCII was the perfect recipe for mojibake disasters

Mojibake

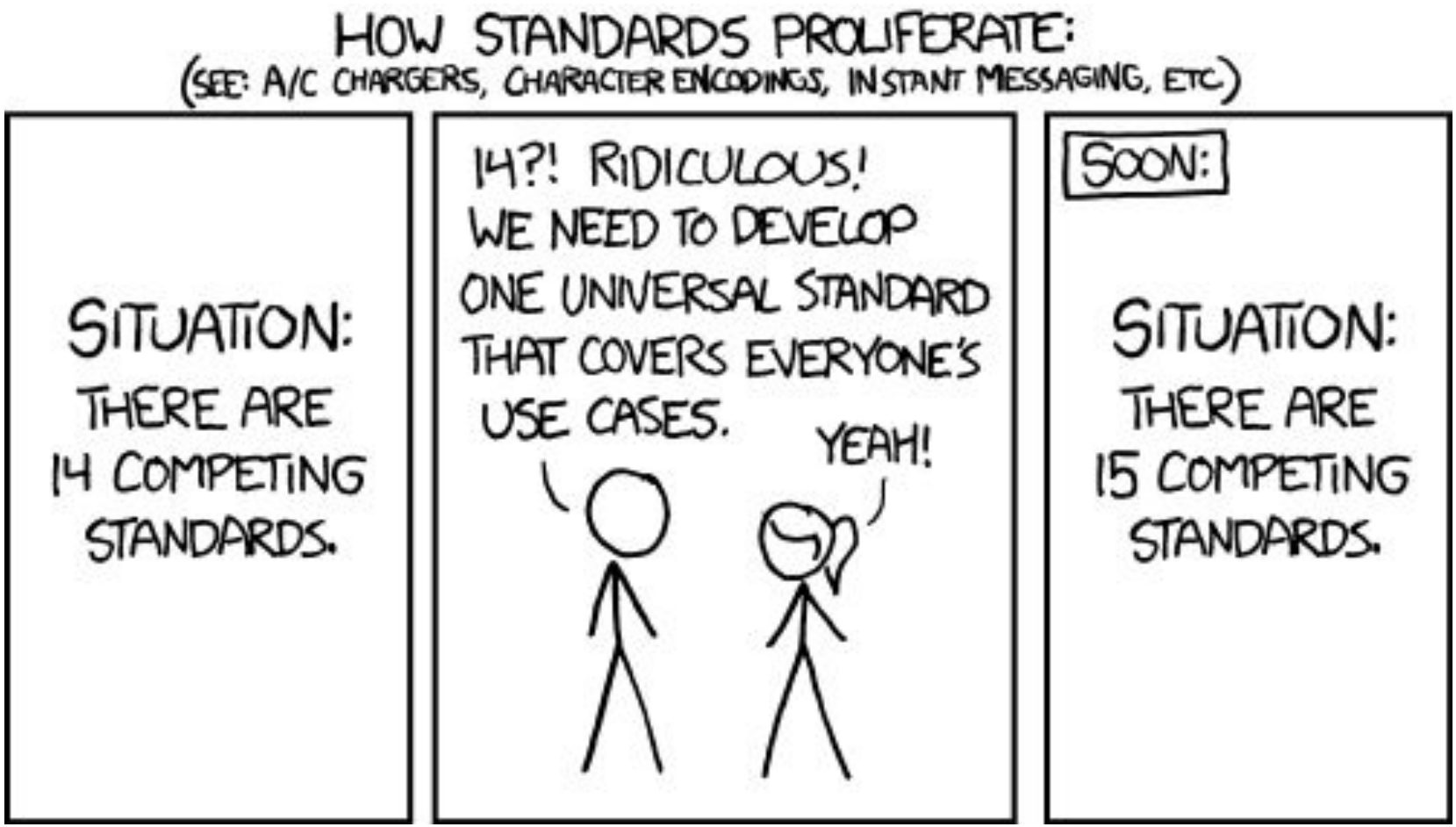
Mojibake occurs when:

- a byte string is decoded using the **wrong character encoding**, or
- two byte strings encoded in **different encodings** are concatenated

This results in garbled, unreadable characters

Examples:

Text	Encoded to	Decoded from	Result
Noël	UTF-8	ISO-8859-1	NoÃ«l
Русский	KOI-8	ISO-8859-1	òÕÓÓËÉÊ


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION:
THERE ARE
14 COMPETING
STANDARDS.
14?! RIDICULOUS!
WE NEED TO DEVELOP
ONE UNIVERSAL STANDARD
THAT COVERS EVERYONE'S
USE CASES.
YEAH!
SOON:
SITUATION:
THERE ARE
15 COMPETING
STANDARDS.

UNICODE

- The goal of UNICODE is to create a single encoding that can represent all of the characters in all of the languages in the world.
- https://en.wikipedia.org/wiki/List_of_Unicode_characters

UNICODE: Codespace

- The Unicode Standard defines a codespace, (ie “The encoding”)
  - The Unicode codespace ranges from **0x0000** to **0x10FFFF**
  - Each hex value represents a code point (ie a character)
- This gives a total of 1,114,112 code points
  - (293,168 are currently assigned) - approximately 25%.

Storing UNICODE characters: UTF-32

- Since the code points range from **0x0000** to **0x10FFFF**
  - So we need at least 21 bits to represent them.
- We can use 32 bits to represent a single character.
- **UTF-32** is a fixed width encoding
  - Simply take the UNICODE code point and store it in 32 bits.

UTF-32: Example

A → U+0041 → 0b00000000000000000000000001000001
€ → U+20AC → 0b00000000000000000010000010101100
字 → U+5B57 → 0b00000000000000000101101101010111
😀 → U+1F600 → 0b00000000000000011111011000000000

U+XXXX is the representation of a raw UNICODE code point

- code points are always at least 4 hex digits

UTF-32: is very very inefficient

- Representing the largest code point U+10FFFF
  - Wastes 11 bits!
- Many characters only need 16 bits
  - Wastes 16 bits bits per character
- Characters that correspond to ASCII only need 7 bits
  - Wastes 25 bits per character!!

UTF-32: is very very inefficient

“Hello 思语” ==

```
0x00000068
0x00000065
0x0000006c
0x0000006c
0x0000006f
0x00000020
0x0000601D
0x00008BED
```

8 x 4 = 32 bytes total - Look at all those leading zeros!!

UTF-8 Variable Length Encoding

UTF-8

- Goal of UTF-8 to increase efficiency
  - Waste less bits!
- Use variable width encoding
  - Why use 4 bytes for every character if we don’t have to?
- The most common characters should use less bits!

UTF-8 Layout

#bytes	#bits	Byte 1	Byte 2	Byte 3	Byte 4
1	7	**0**xxxxxxx	-	-	-
2	11	**110**xxxxx	**10**xxxxxx	-	-
3	16	**1110**xxxx	**10**xxxxxx	**10**xxxxxx	-
4	21	**11110**xxx	**10**xxxxxx	**10**xxxxxx	**10**xxxxxx

- A single **UTF-8** character can be anywhere from **1 to 4 bytes** long

UTF-8 Layout

- All ASCII characters can be stored in:
  - 1 byte with zero wasted bits
- Every UNICODE character can be stored in
  - 4 bytes, which is the same as UTF-32 in the worst case

UTF-8 Layout

#bytes	#bits	Byte 1	Byte 2	Byte 3	Byte 4
1	7	**0**xxxxxxx	-	-	-
2	11	**110**xxxxx	**10**xxxxxx	-	-
3	16	**1110**xxxx	**10**xxxxxx	**10**xxxxxx	-
4	21	**11110**xxx	**10**xxxxxx	**10**xxxxxx	**10**xxxxxx

- A single **UTF-8** character can be anywhere from **1 to 4 bytes** long
- Exercise: How many UTF-8 encoded characters would this represent
    - **110**10111 **10**101111 **1110**1110 **10**111100 **10**001011 **0**1001101

Conversion to UTF-8 (1/2)

€ (U+20AC)

- Convert to UTF-32 (raw 32 bit representation of the code point)
  0x000020AC
  0b00000000000000**10000010101100**
  - Look at all those leading zeros!
- remove leading 0s from the UTF-32 encoding
  0b**10000010101100**
- Split into 6 bit chunks from right to left

Conversion to UTF-8 (2/2)

€ (U+20AC)

- `0b 10 000010 101100`
- Match with appropriate multi-byte encoding (in this case, 3 chunks)

  `0b 1110xxxx 10xxxxxx 10xxxxxx`

  `0b 10 000010 101100`
- Replace the x values with the appropriate bits (0 if none)

  `0b 11100010 10000010 10101100`
- And in hex it looks like

  `0b 1110 0010 1000 0010 1010 1100`

  `0x E 2 8 2 A C`
- We saved a byte of storage! 😀

Exercise: Convert to UTF-8

字 → U+5B57

UTF-8: More Examples

A → U+0041 → 0b01000001 → 0x41

€ → U+20AC → 0b10 000010 101100

→ 0b11100010 10000010 10101100

→ 0xE282AC

字 → U+5B57 → 0b101 101101 010111

→ 0b11100101 10101101 10010111

→ 0xE5AD97

😀 → U+1F600 → 0b 11111 011000 000000

→ 0b11110000 10011111 10011000 10000000

→ 0xF09F9880

UTF-32: is very very inefficient

“Hello 思语” ==

```
0x00000068
0x00000065
0x0000006c
0x0000006c
0x0000006f
0x00000020
0x0000601D
0x00008BED
```

8 x 4 = 32 bytes total - Look at all those leading zeros!!

UTF-8 - much more efficient

“Hello 思语” ==

```
0x68
0x65
0x6c
0x6c
0x6f
0x20
0xE6809D
0xE8AFAD
```

12 bytes only - and no more leading zeros!

Writing C that uses Unicode

hello_unicode.c

unicode_strings.c

utf8_strlen.c

Summary of UTF-8

- Compact, but not minimal encoding
  - ASCII is a subset of UTF-8 - complete backwards compatibility!
  - No byte of multi-byte UTF-8 encoding is valid ASCII
  - No byte of multi-byte UTF-8 encoding is 0
    - can still use store UTF-8 in null-terminated strings.
  - 0x2F (ASCII /) and 0x00 can not appear in multi-byte characters
    - hence can use UTF-8 for Linux/Unix filenames
- C programs can treat UTF-8 similarly to ASCII
  - **Beware: number of bytes in UTF-8 string != number of characters**

What we learnt Today

- File recap
- Handy file functions
- ASCII
- Unicode
  - UTF-8 Encoding

Next Lecture

- Processes!

Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/0u5PAXfKtD

Reach Out

Content Related Questions:

Forum

Admin related Questions email:

cs1521@cse.unsw.edu.au

Student Support | I Need Help With…

My Feelings and Mental Health
Managing Low Mood, Unusual Feelings & Depression

Mental Health Connect
student.unsw.edu.au/**counselling**
Telehealth

Mind HUB
student.unsw.edu.au/**mind-hub**
Online Self-Help Resources

In Australia Call Afterhours UNSW Mental Health Support Line
1300 787 026
5pm-9am

Outside Australia Afterhours 24-hour Medibank Hotline
+61 (2) 8905 0307

Uni and Life Pressures
Stress, Financial, Visas, Accommodation & More

Student Support
Indigenous Student Support
student.unsw.edu.au/**advisors**

Reporting Sexual Assault/Harassment

Equity Diversity and Inclusion (EDI)
edi.unsw.edu.au/**sexual-misconduct**

Educational Adjustments
To Manage my Studies and Disability / Health Condition

Equitable Learning Service (ELS)
student.unsw.edu.au/**els**

Academic and Study Skills

Academic Language Skills
student.unsw.edu.au/**skills**

Special Consideration
Because Life Impacts our Studies and Exams

Special Consideration
student.unsw.edu.au/**special-consideration**