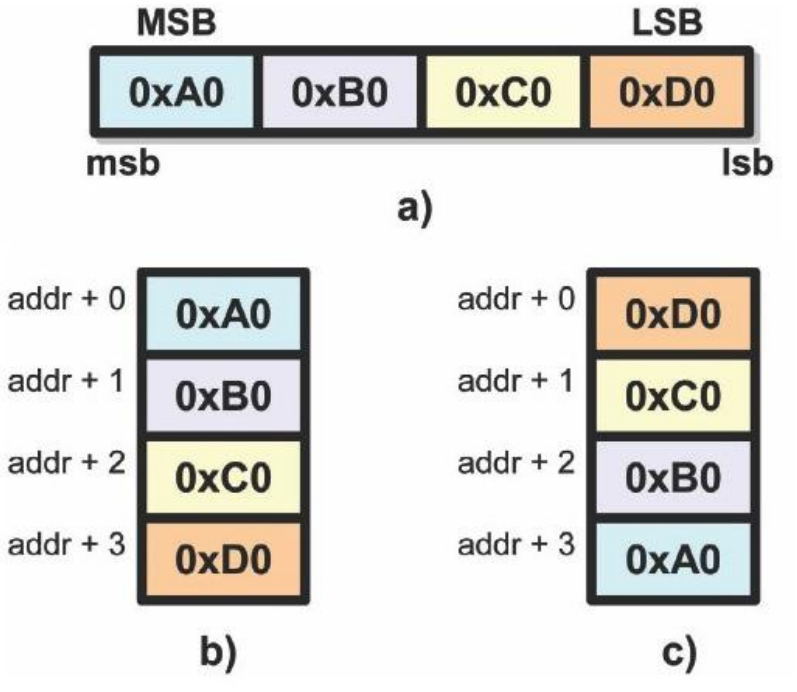
Il Problema dell'eterogeneità

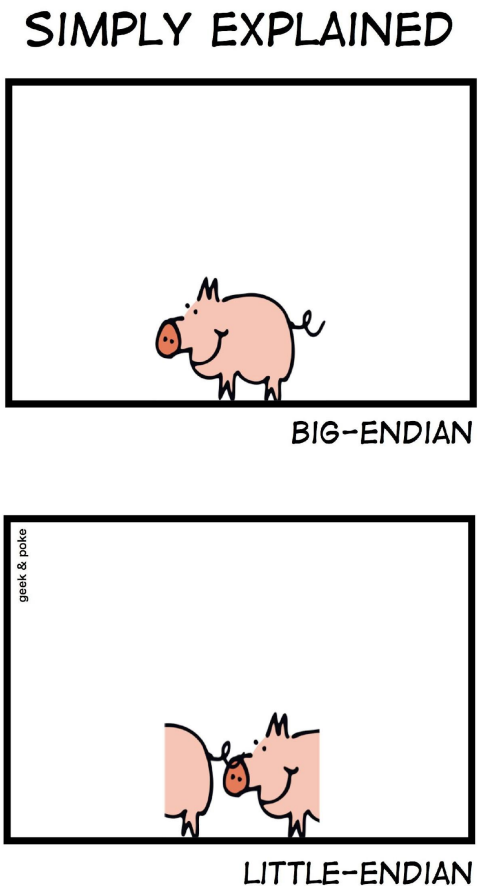
- Nelle reti di calcolatori vi è un'estrema eterogeneità di sistemi (hardware e software)
- Client e Server possono eseguire su architetture diverse che usano differenti rappresentazioni dei dati:
  - caratteri (ASCII, ISO 8859, Unicode, …)
  - interi (dimensione 4 o 8 byte, rappresentazione in complemento a 1 o a 2, ...)
  - reali (lunghezza exp e mantissa, formato, …)
  - ordine byte all'interno di una parola (little endian o big endian)
- Necessità di definire una rappresentazione comune dei dati e di implementare meccanismi per gestirla

Es. little endian o big endian



Big endian

Little endian



Caveat

- Attenzione! Il problema della rappresentazione eterogenea dei dati tra diverse piattaforme HW/SW presenta complessità che vanno ben oltre le comunicazioni di rete
- Ad esempio, anche la ricompilazione dello stesso codice sorgente su piattaforme diverse può presentare spiacevoli sorprese. Si veda l'articolo:

"Twice the Bits, Twice the Trouble:
Vulnerabilities Induced by Migrating to 64-Bit Platforms"
https://www.tu-braunschweig.de/Medien-DB/sec/pubs/2016-ccs.pdf

Rappresentazione dati

- Per comunicare tra nodi eterogenei sono possibili due tipi di soluzioni:
  - Ogni nodo converte i dati nel formato specifico del destinatario (prestazioni)
  - Si concorda un formato comune di rappresentazione dei dati che i nodi useranno per comunicare tra loro (flessibilità)
- Supponendo di avere N diversi tipi di nodi, nel primo caso avrò bisogno di N*(N-1) procedure di conversione dati, nel secondo caso solo di 2*N procedure

Ma quale ISO livello 6?!?!

- La Socket API è un'interfaccia di basso livello (tra ISO L4 e L5) che purtroppo non fornisce alcuno strumento di questo tipo
- Non esiste un unico standard per il livello di presentazione
  - Molte soluzioni, con caratteristiche molto diverse, sono stati sviluppate per ambiti specifici
- La soluzione giusta da adottare andrà valutata caso per caso

eXternal Data Representation (XDR)

- Sun XDR è una soluzione realizzata all'interno dello stack Sun/ONC RPC
- XDR fornisce un insieme di procedure di conversione per trasformare la rappresentazione nativa dei dati in una *rappresentazione esterna* (XDR) e viceversa
- XDR fa uso di uno stream (contenuto in un buffer) che permette di creare un messaggio con i dati in forma XDR
- I dati vengono inseriti/estratti nello/dallo stream XDR uno alla volta, tramite operazioni di serializzazione e/o deserializzazione

Esempio di serializzazione XDR

```
int i = 260;
char str[80] = “pippo”;
XDR *xdrs;
char buf[BUFSIZE]; /* buffer vuoto, da preparare */
...
/* Creazione stream XDR in memoria */
xdrmem_create(xdrs, buf, sizeof(buf), XDR_ENCODE);

/* Inserimento nello stream di un intero, convertito
in formato XDR */
xdr_int(xdrs, &i);

/* Inserimento nello stream di una stringa,
convertita in formato XDR */
xdr_string(xdrs, &str, strlen(str));

/* Scrittura su socket */
write(sd, buf, xdr_getpos(xdrs));
```

Esempio di deserializzazione XDR

```
int i;
char str[80];
XDR *xdrs;
char buf[BUFSIZE]; /* buffer vuoto, da preparare */
…
/* Lettura da socket */
read(sd, buf, sizeof(buf));

/* Creazione stream XDR in memoria */
xdrmem_create(xdrs, buf, sizeof(buf), XDR_DECODE);

/* Lettura di un intero dallo stream, convertito dal
formato XDR */
xdr_int(xdrs, &i);

/* Lettura di una stringa dallo stream, convertita
dal formato XDR */
memset(str, 0, sizeof(str));
xdr_string(xdrs, &str, sizeof(str)-1);
```

IDL

- Oltre ai tipi di dati primitivi, per cui fornisce già routine di serializzazione e di deserializzazione, XDR permette di gestire tipi di dati complessi
- In XDR, il formato delle strutture dati è definito attraverso un apposito linguaggio *IDL (Interface Definition Language)* simile al C
- Uso di *IDL compiler* per generare automaticamente le procedure di codifica e decodifica dei dati complessi

Esempio di IDL (Sun/ONC RPC)

```
/* definisci massima dimensione stringhe */
const MAXNAMELEN = 255;

/* parametro: nome directory */
typedef string nametype<MAXNAMELEN>;

/* valore di ritorno: lista di file */
typedef struct namenode *namelist;

struct namenode {
        nametype name; /* nome del file */
        namelist pNext; /* prossimo file */
};
```

Altre soluzioni

- CORBA Common Data Representation (CDR)
- ASN.1/X.680 (ITU-T/OSI)
- Soluzioni Web-oriented:
  - Google Protocol Buffers
  - Apache Avro
  - MessagePack
  - Apache Thrift

Protocolli Testuali

- Nella realizzazione di applicazioni distribuite, l'adozione di protocolli testuali si è spesso rivelata vincente
  - Facilità di testing e debugging
  - Estendibilità
  - Resilienza alla complessità (sistemi di successo di solito evolvono in sistemi più complessi)

Per approfondire si leggano i seguenti saggi di Eric S. Raymond:
How Not To Design a Wire Protocol, http://esr.ibiblio.org/?p=8254
The Art of Unix Programming, http://www.catb.org/~esr/writings/taoup/
"When you feel the urge to design a complex binary file format, or a complex binary application protocol, it is generally wise to lie down until the feeling passes."

US-ASCII

- Per decenni, lo standard per la rappresentazione del testo è stato US-ASCII.
- US-ASCII definisce un set di caratteri e una loro rappresentazione in formato binario a 8-bit.
- Solo i 7 bit meno significativi sono effettivamente utilizzati nello standard US-ASCII (127 caratteri). Il bit più significativo di ciascun byte è sempre settato a 0.

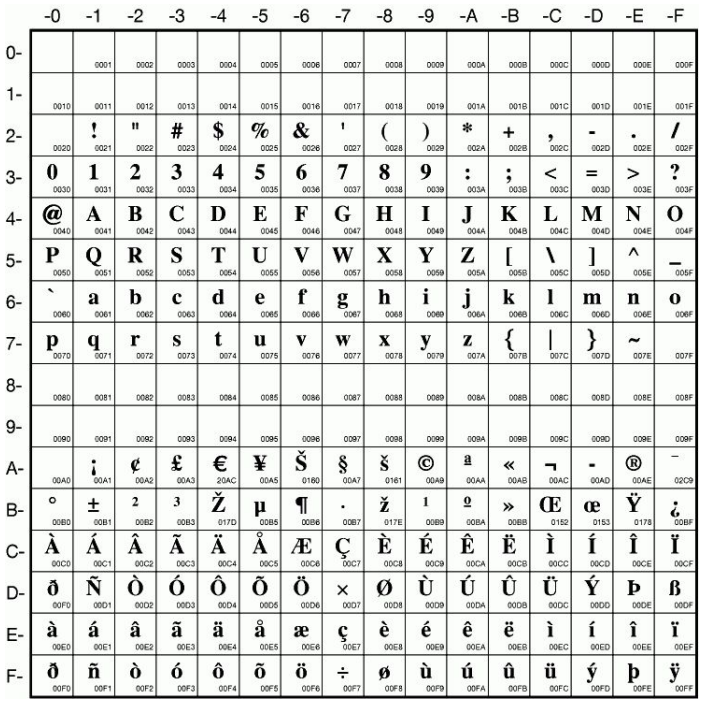
Tabella US-ASCII

b7 → b6 → b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits	b4 ↓	b3 ↓	b2 ↓	b1 ↓	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	FC	,	<	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

ISO 8859

- I caratteri della tabella US-ASCII sono sufficienti per la rappresentazione della lingua inglese, ma non per quella di molte altre lingue europee (non supportano accenti, umlaut, ecc.)
- Lo standard ISO 8859 estende US-ASCII utilizzando anche l'ottavo bit, portando così il set di caratteri supportati a 255
- Diverse mappe di caratteri *8859-n* per coprire le varie lingue (es. per l'italiano si hanno 8859-1 “Latin 1” e 8859-15 “Latin 9”)

Tabella ISO 8859-15



	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
1-	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
2-	0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3-	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4-	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5-	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6-	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7-	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	007F
8-	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	008A	008B	008C	008D	008E	008F
9-	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	009A	009B	009C	009D	009E	009F
A-	00A0	¡ 00A1	¢ 00A2	£ 00A3	€ 20AC	¥ 00A5	Š 0160	§ 00A7	š 0161	© 00A9	ª 00AA	« 00AB	¬ 00AC	- 00AD	® 00AE	¯ 02C9
B-	° 00B0	± 00B1	2 00B2	3 00B3	Ž 017D	µ 00B5	¶ 00B6	· 00B7	ž 017E	1 00B9	º 00BA	» 00BB	Œ 0152	œ 0153	Ÿ 0178	¿ 00BF
C-	À 00C0	Á 00C1	Â 00C2	Ã 00C3	Ä 00C4	Å 00C5	Æ 00C6	Ç 00C7	È 00C8	É 00C9	Ê 00CA	Ë 00CB	Ì 00CC	Í 00CD	Î 00CE	Ï 00CF
D-	Ð 00F0	Ñ 00D1	Ò 00D2	Ó 00D3	Ô 00D4	Õ 00D5	Ö 00D6	× 00D7	Ø 00D8	Ù 00D9	Ú 00DA	Û 00DB	Ü 00DC	Ý 00DD	Þ 00DE	ß 00DF
E-	à 00E0	á 00E1	â 00E2	ã 00E3	ä 00E4	å 00E5	æ 00E6	ç 00E7	è 00E8	é 00E9	ê 00EA	ë 00EB	ì 00EC	í 00ED	î 00EE	ï 00EF
F-	ð 00F0	ñ 00F1	ò 00F2	ó 00F3	ô 00F4	õ 00F5	ö 00F6	÷ 00F7	ø 00F8	ù 00F9	ú 00FA	û 00FB	ü 00FC	ý 00FD	þ 00FE	ÿ 00FF

Un mondo post US-ASCII

- Nel ventunesimo secolo, non è più possibile limitare le nostre applicazioni ai set di caratteri US-ASCII o ISO 8859
- Necessità di nuovi standard, che supportino anche:
  - Nuovi alfabeti (cinese, arabo, ebraico, cirillico, ecc.)
  - Caratteri composti
  - Modi di scrittura right-to-left

ISO 10646

- Lo *standard normativo ISO 10646* definisce lo *Universal Character Set (UCS)*, un set di caratteri che contiene tutti i caratteri universalmente noti
- Ciascun carattere ha un codice a esso associato, e viene rappresentato con una notazione esadecimale come U+12345678
- Separazione tra Basic Multilingual Plane (BMP) (caratteri da U+0000 a U+FFFF) e plane estesi (astrali)
- Supporto a caratteri composti (ad esempio il carattere Ä è rappresentabile tramite la coppia di caratteri U+0041 U+0308)

Unicode

- *Unicode* è uno *standard implementativo* sviluppato a partire dagli anni '80 da un consorzio di industrie che realizzavano software multilingua
- Inizialmente sviluppato parallelamente e indipendentemente da ISO 10646, si è allineato con quest'ultimo nel 1991
- Unicode *definisce degli standard (UTF-*/UCS-*) per l'encoding* dei caratteri dello UCS
  - Assunzione di lavoro (2003): i caratteri UCS hanno codici rappresentabili con al massimo 21 bit (UTF-16 non è in grado di rappresentare caratteri oltre U+10FFFF)

UTF-32 / UCS-4

- UTF-32 (anche noto come UCS-4) è l'encoding più semplice: ogni carattere viene rappresentato con 4 byte
- Molto spesso, questo tipo di codifica è troppo “onerosa”
  - I caratteri al di fuori del Basic Multilingual Plane sono talmente rari da poter essere ignorati per molte applicazioni
  - I caratteri all’interno del Basic Multilingual Plane sarebbero rappresentabili con 16 bit
- Codifica *non compatibile con US-ASCII*
- Encoding raramente adottato per le comunicazioni e praticamente utilizzato solo all'interno delle librerie di gestione del testo

UFT-16 / UCS-2

- UTF-16 (che estende il precedente UCS-2) è una **codifica a lunghezza variabile**:
  - 16 bit per i caratteri del Basic Multilingual Plane (U+0000-U+FFFF)
  - 32 bit per gli altri caratteri (U+10000-U+10FFFF)
- Rappresentazione piuttosto compatta
- Encoding standard di Java e Windows
- Codifica *non compatibile con US-ASCII*
- Due diverse versioni: UTF-16LE (little endian) e UTF-16BE (big endian)
  - Ove non specificato, uso di U+FEFF come Byte Order Mark

UTF-8

- UTF-8 è una **codifica a lunghezza variabile, che supporta tutto il set di caratteri UCS**
  - Ricordiamo che al momento si assume che i caratteri UCS abbiano codici rappresentabili con al massimo 21 bit
- La codifica UTF-8 associa a ciascun carattere una sequenza di byte di lunghezza variabile (da 1 a 4)
- Encoding standard di XML, di JSON, e della maggior parte dei sistemi Unix moderni

UTF-8 e retrocompatibilità

- UTF-8 è lo strumento messo a disposizione da UCS e Unicode per fornire un certo livello di compatibilità con il passato
  - I caratteri da U+0000 a U+007F sono identici ai caratteri della tabella US-ASCII
  - UTF-8 usa un solo byte per rappresentare i caratteri da U+0000 a U+007F
  - UTF-8 permette stringhe null-terminated
- Quindi, *l'encoding UTF-8 è del tutto compatibile con l'encoding US-ASCII* per il subset di caratteri da quest'ultimo supportati

Encoding UTF-8

Caratteri UCS	Rappresentazione binaria in UTF-8
Da U+000000 a U+00007F	0XXXXXXX
Da U+000080 a U+0007FF	110XXXXX 10XXXXXX
Da U+000800 a U+00FFFF	1110XXXX 10XXXXXX 10XXXXXX
Da U+010000 a U+10FFFF	11110XXX 10XXXXXX 10XXXXXX 10XXXXXX

- Velocità di decodifica: il primo byte mi dice quanti altri byte devo leggere per ottenere un carattere completo
- Auto-sincronizzazione: riesco a capire facilmente e velocemente dove inizia un carattere guardando in un intorno di 3 byte dalla posizione corrente

UTF-8 - Validazione

- **Attenzione!** Poiché in UTF-8 i caratteri hanno dimensione variabile, si deve fare particolare attenzione a verificare che un buffer di memoria non contenga dei caratteri incompleti prima di utilizzarlo!
  - Con una `write` da un file o da una IPC potrei leggere in un buffer solo una parte di una stringa UTF-8, che non contiene tutti i byte che codificano l'ultimo carattere ricevuto!
- **Attenzione!** Oltre a verificare che un buffer non contenga caratteri incompleti bisogna verificare che i dati rappresentati siano effettivamente validi!
  - I caratteri tra U+D800 e U+DFFF sono riservati per UTF-16 e non possono essere usati in UTF-8
  - Rappresentazioni UTF-8 di caratteri con un numero di bit significativi maggiore di 21 vanno scartate

UTF-8 - Validazione

UTF-8

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	NUL 0000 0	SOH 0001 1	STX 0002 2	ETX 0003 3	EOT 0004 4	ENQ 0005 5	ACK 0006 6	BEL 0007 7	BS 0008 8	HT 0009 9	LF 000A 10	VT 000B 11	FF 000C 12	CR 000D 13	SO 000E 14	SI 000F 15
1_	DLE 0010 16	DC1 0011 17	DC2 0012 18	DC3 0013 19	DC4 0014 20	NAK 0015 21	SYN 0016 22	ETB 0017 23	CAN 0018 24	EM 0019 25	SUB 001A 26	ESC 001B 27	FS 001C 28	GS 001D 29	RS 001E 30	US 001F 31
2_	SP 0020 32	! 0021 33	" 0022 34	# 0023 35	$ 0024 36	% 0025 37	& 0026 38	' 0027 39	(0028 40	) 0029 41	* 002A 42	+ 002B 43	, 002C 44	- 002D 45	. 002E 46	/ 002F 47
3_	0 0030 48	1 0031 49	2 0032 50	3 0033 51	4 0034 52	5 0035 53	6 0036 54	7 0037 55	8 0038 56	9 0039 57	: 003A 58	; 003B 59	< 003C 60	= 003D 61	> 003E 62	? 003F 63
4_	@ 0040 64	A 0041 65	B 0042 66	C 0043 67	D 0044 68	E 0045 69	F 0046 70	G 0047 71	H 0048 72	I 0049 73	J 004A 74	K 004B 75	L 004C 76	M 004D 77	N 004E 78	O 004F 79
5_	P 0050 80	Q 0051 81	R 0052 82	S 0053 83	T 0054 84	U 0055 85	V 0056 86	W 0057 87	X 0058 88	Y 0059 89	Z 005A 90	[005B 91	\ 005C 92	] 005D 93	^ 005E 94	_ 005F 95
6_	` 0060 96	a 0061 97	b 0062 98	c 0063 99	d 0064 100	e 0065 101	f 0066 102	g 0067 103	h 0068 104	i 0069 105	j 006A 106	k 006B 107	l 006C 108	m 006D 109	n 006E 110	o 006F 111
7_	p 0070 112	q 0071 113	r 0072 114	s 0073 115	t 0074 116	u 0075 117	v 0076 118	w 0077 119	x 0078 120	y 0079 121	z 007A 122	{ 007B 123	| 007C 124	} 007D 125	~ 007E 126	DEL 007F 127
8_	• +00 128	• +01 129	• +02 130	• +03 131	• +04 132	• +05 133	• +06 134	• +07 135	• +08 136	• +09 137	• +0A 138	• +0B 139	• +0C 140	• +0D 141	• +0E 142	• +0F 143
9_	• +10 144	• +11 145	• +12 146	• +13 147	• +14 148	• +15 149	• +16 150	• +17 151	• +18 152	• +19 153	• +1A 154	• +1B 155	• +1C 156	• +1D 157	• +1E 158	• +1F 159
A_	• +20 160	• +21 161	• +22 162	• +23 163	• +24 164	• +25 165	• +26 166	• +27 167	• +28 168	• +29 169	• +2A 170	• +2B 171	• +2C 172	• +2D 173	• +2E 174	• +2F 175
B_	• +30 176	• +31 177	• +32 178	• +33 179	• +34 180	• +35 181	• +36 182	• +37 183	• +38 184	• +39 185	• +3A 186	• +3B 187	• +3C 188	• +3D 189	• +3E 190	• +3F 191
2-byte C_	192	193	Latin 0080 194	Latin 00C0 195	Latin 0100 196	Latin 0140 197	Latin 0180 198	Latin 01C0 199	Latin 0200 200	IPA 0240 201	IPA 0280 202	IPA 02C0 203	accents 0300 204	accents 0340 205	Greek 0380 206	Greek 03C0 207
2-byte D_	Cyril 0400 208	Cyril 0440 209	Cyril 0480 210	Cyril 04C0 211	Cyril 0500 212	Armeni 0540 213	Hebrew 0580 214	Hebrew 05C0 215	Arabic 0600 216	Arabic 0640 217	Arabic 0680 218	Arabic 06C0 219	Syriac 0700 220	Arabic 0740 221	Thaana 0780 222	N'Ko 07C0 223
3-byte E_	Indic 0800 224	Misc. 1000 225	Symbol 2000 226	Kana, CJK 3000 227	CJK 4000 228	CJK 5000 229	CJK 6000 230	CJK 7000 231	CJK 8000 232	CJK 9000 233	Asian A000 234	Hangul B000 235	Hangul C000 236	Hangul D000 237	PUA E000 238	Forms F000 239
4-byte F_	SMP, SIP 10000 240	40000 241	80000 242	SSP, SPUA C0000 243	SPUA-B 100000 244	245	246	247	248	249	250	251	252	253	254	255

La tabella (presa da Wikipedia) mostra che alcuni byte in una sequenza di dati codificata in UTF-8 sono sicuramente (rosso) o possibilmente (rosa) rappresentazioni di caratteri non validi.

wchar_t, char16_t e char32_t

- Storicamente, C si basava su caratteri wide (tipo di dato `wchar_t`) per l'encoding di caratteri multibyte
- Sfortunatamente, la dimensione di `wchar_t` non è standardizzata (Windows usa 16 bit, Unix 32)
  - Solo implementazioni che usano `wchar_t` di dimensioni maggiori di 20 bit sono well behaved e possono dichiarare la macro `__STDC_ISO_10646__` con un valore maggiore o uguale a 200103L
- C11 introduce i tipi `char16_t` e `char32_t` (e le macro `__STDC_UTF_16__` e `__STDC_UTF_32__`) **ma non le funzioni per gestirli**

Manipolazione stringhe Unicode

- Chiaramente, non possiamo più usare `strlen` per contare il numero di caratteri in una stringa Unicode – nemmeno in quelle UTF-8 NULL-terminated
  - Le funzioni di libreria `str*` sono state progettate assumendo una codifica ASCII che usa un byte per carattere
  - È necessario convertire la stringa UTF-8 a un array di `wchar_t` e usare funzioni `wcs*` (e per piattaforme non compliant come Windows?)
- Inoltre, poiché UCS ammette caratteri composti, non è detto che la lunghezza di una stringa equivalga al numero effettivo di caratteri che essa stamperà a video
  - Uso di `wcswidth` per determinare il numero di colonne richieste per la stampa di un array di caratteri `wchar_t`
- In generale, **meglio usare librerie specifiche per la gestione del testo UTF-8**

Table 7-4: Narrow and Wide String Functions

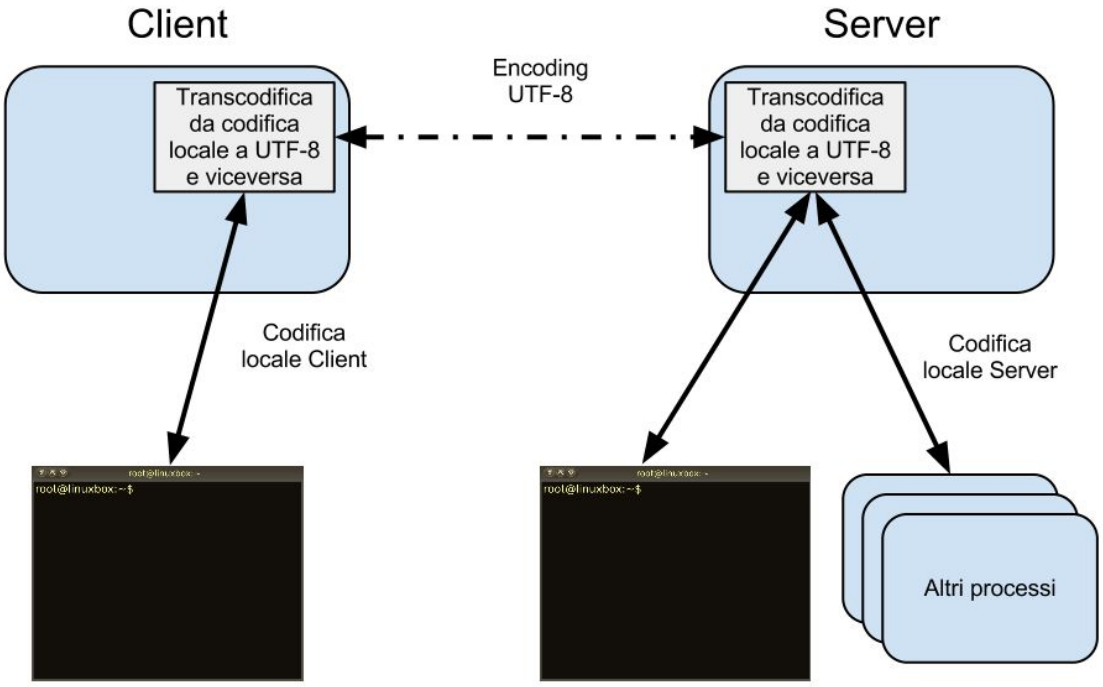
Narrow (char)	Wide (wchar_t)	Description
strcpy	wcscpy	String copy
strncpy	wcsncpy	Truncated, zero-filled copy
memcpy	wmemcpy	Copies a specified number of code units
memmove	wmemmove	Copies a specified number of (possibly overlapping) code units
strcat	wcscat	Concatenates strings
strncat	wcsncat	Concatenates strings with truncation
strcmp	wcscmp	Compares strings
strncmp	wcsncmp	Compares strings to a point
strchr	wcschr	Locates a character in a string
strcspn	wcscspn	Computes the length of a complementary string segment
strpbrk	wcspbrk	Finds the first occurrence of a set of characters in a string
strrchr	wcsrchr	Finds the first occurrence of a character in a string
strspn	wcsspn	Computes the length of a string segment
strstr	wcsstr	Finds a substring
strtok	wcstok	String tokenizer (modifies the string being tokenized)
memchr	wmemchr	Finds a code unit in memory
strlen	wcslen	Computes string length
memset	wmemset	Fills memory with a specified code unit

Corrispondenze tra funzioni per manipolazione stringhe con caratteri narrow (ASCII) e wide (wchar_t) nel linguaggio C. Courtesy: R. Seacord, “Effective C”, No Starch Press, 2020.

Unicode e Applicazioni Distribuite

- È bene quindi realizzare applicazioni distribuite che comunichino con l'esterno utilizzando UTF-8
- Nel caso il formato di rappresentazione delle stringhe nella nostra piattaforma di sviluppo non sia UTF-8 (ad esempio su Windows), dobbiamo prevedere una transcodifica da UTF-8 alla codifica locale e viceversa

Esempio



Transcodifica

- Per la transcodifica tra UTF-8 e codifica locale:
  - In Java si usano le funzioni di transcodifica fornite da InputStreamReader e OutputStreamWriter (il secondo parametro del costruttore specifica la codifica “esterna”).
  - In Unix/C lo standard è rappresentato dalla funzione *iconv* fornita dalle librerie di sistema, che però è piuttosto difficile da utilizzare. È pertanto consigliabile valutare l'uso di librerie di più alto livello, come *utf8proc* (https://julialang.org/utf8proc/), *libunistring* (https://www.gnu.org/software/libunistring/), *glib* (http://www.gtk.org), o addirittura *ICU* (http://site.icu-project.org/).

Esempi

- Validazione di una stringa UTF-8 a partire da buffer NULL-terminated con libunistring:

```
#include <unistring.h>
#include <inttypes.h>

/* mi assicuro che il buffer sia NULL-terminated */
char buffer[4096];
memset(buffer, 0, sizeof(buffer));
bytes_received = read(socket_fd, buffer, sizeof(buffer)-1);

/* verifico che il messaggio sia UTF-8 valido */
if (u8_check(buffer, bytes_received) != NULL) {
  /* stringa non (interamente) valida, posso provare
     a sanitizzarla o rigettarla: meglio la seconda opzione! */
  fprintf(stderr, "Error 3\n"); fflush(stderr);
  close(socket_fd);
  exit(EXIT_SUCCESS);
}
```

Esempi

- Validazione di una stringa UTF-8 a partire da buffer NULL-terminated con glib:

```
#include <glib.h>

/* mi assicuro che il buffer sia NULL-terminated */
char buffer[4096];
memset(buffer, 0, sizeof(buffer));
bytes_received = read(socket_fd, buffer,
sizeof(buffer)-1);

if (g_utf8_validate(buffer, NULL, NULL)) {
  /* stringa valida */
} else {
  /* stringa non (interamente) valida, posso provare
     a sanitizzarla o rigettarla: meglio la seconda
     opzione! */
  fprintf(stderr, "Error 3\n"); fflush(stderr);
  close(socket_fd);
  exit(EXIT_SUCCESS);
}
```

Esempi

- Estrazione sub-stringa UTF-8 valida da un buffer (non necessariamente NULL-terminated) con glib:

```
#include <glib.h>
#include <string.h>

char buff[4096]; int used_buf; char *valid_upto;

used_buf = read(fd, buffer, sizeof(buffer));
if (g_utf8_validate(buff, used_buf, &valid_upto)) {
  /* tutta la stringa è valida */
} else if (valid_upto > buff) {
  /* estraggo porzione di stringa valida e lascio quella non
     valida nel buffer */
  size_t valid_bytes = valid_upto - buff;
  /* devo ricordarmi di deallocare la stringa con free */
  char *valid_str = strndup(buff, valid_bytes);
  memmove(buff, valid_upto, used_buf - valid_bytes);
  used_buf -= valid_bytes;
}
```

Esempi

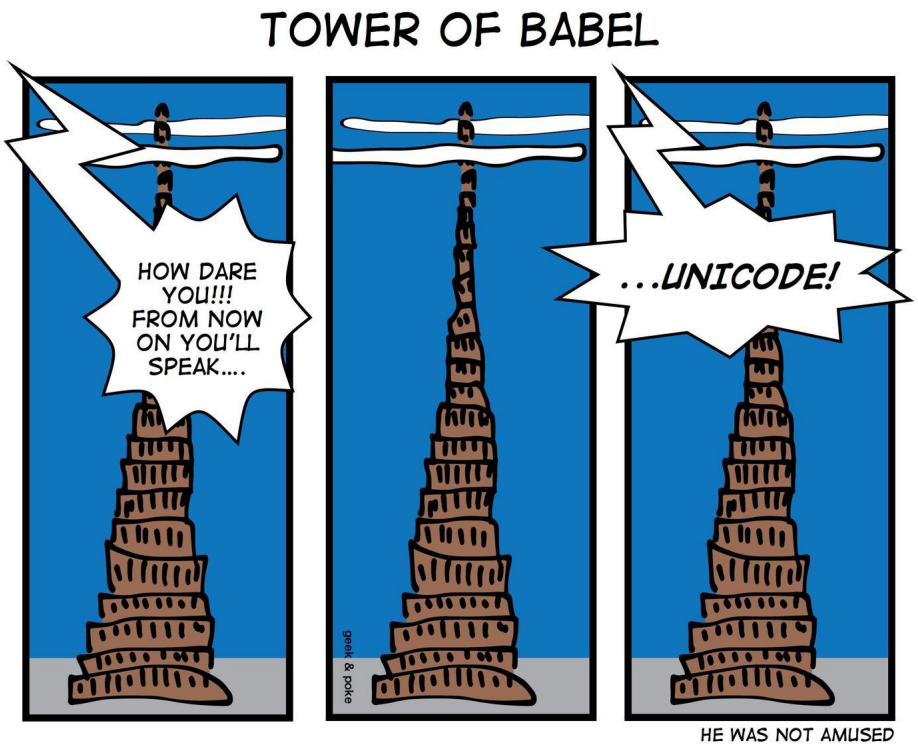
- Sanitizzazione di testo UTF-8 in un buffer (non necessariamente NULL-terminated) con glib:

```
#include <glib.h>

char buff[4096];
int used_buf;
char *sanitized_str;

used_buf = read(socket_fd, buffer, sizeof(buffer));

/* ricordarsi di deallocare la stringa con free */
sanitized_str = g_utf8_make_valid(buff, used_buf);
```

Attenzione! La sanitizzazione dell'input è un'operazione molto delicata dal punto di vista della sicurezza! Lasciarla a una funzione di libreria di cui non si conosce perfettamente l'implementazione è una pessima idea.

Per approfondire

- Per ulteriori informazioni su Unicode e UTF-8 si vedano:
  - http://www.joelonsoftware.com/articles/Unicode.html
  - http://utf8everywhere.org/
  - http://www.cl.cam.ac.uk/~mgk25/unicode.html
  - http://hackaday.com/2013/09/27/utf-8-the-most-elegant-hack/
  - https://eev.ee/blog/2015/09/12/dark-corners-of-unicode/
  - http://nullprogram.com/blog/2017/10/06/
  - http://bjoern.hoehrmann.de/utf-8/decoder/dfa/



Dati strutturati

- È possibile scambiare dati di tipo strutturato anche al di sopra di protocolli di tipo testuale, utilizzando standard come XML e JSON
- Di solito, la perdita di performance legata alla trasformazione da rappresentazione binaria a rappresentazione testuale (e viceversa) è più che ripagata dalla flessibilità, dalla robustezza e dalla facilità di debug dei protocolli testuali

XML

- XML è un linguaggio di descrizione specializzabile per settori specifici (ovverosia un metalinguaggio)
- Standardizzato da W3C, è molto utilizzato anche al di fuori del Web
- Tecnologia sviluppata in ottica machine-oriented, per facilitare la generazione automatica di codice che valida e/o manipola tipi di dati strutturati rigorosamente definiti
- XML è usato sia per la rappresentazione di dati e messaggi scambiati che per la definizione del loro formato

Esempio XML

```
<? xml version="1.0" encoding="utf-8" ?>
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber type="home">212 555-1234</phoneNumber>
    <phoneNumber type="fax">646 555-4567</phoneNumber>
  </phoneNumbers>
</person>
```

XML Schema

- XML Schema è un linguaggio derivato da XML che consente di definire tipi di documento XML contenenti tipi di dati con strutture complesse o non regolari
- XML Schema permette di:
  - definire tipi di dati complessi, basandosi su tipi di dati predefiniti
  - specificare l'ordine in cui gli elementi di un dato devono comparire
  - definire delle regole che specificano il numero di volte che ciascun elemento può o deve comparire
- Uso di XML Namespace per evitare ambiguità

Esempio XML Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Person" type="PersonType"/>
  <xsd:complexType name="PersonType">
    <xsd:sequence>
      <xsd:element name="firstName" type="xsd:string"
           minOccurs="0" MaxOccurs="1"/>
      <xsd:element name="lastName" type="xsd:string"
           minOccurs="1" MaxOccurs="1"/>
      <xsd:element name="age" type="AgeType" />
           minOccurs="0" MaxOccurs="1"/>
    ...
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="AgeType">
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="0"/>
      <xsd:maxInclusive value="100"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

JSON

- JSON è un formato di rappresentazione dati particolarmente leggero e molto utilizzato sul Web
  - Molto più compatto e significativamente più performante e facile da processare rispetto a XML
- Pensato per applicazioni Web 2.0 e per la manipolazione dei dati in JavaScript
- Compatibile con hash in linguaggio JavaScript

Esempio JSON

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021"
    },
    "phoneNumber": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```