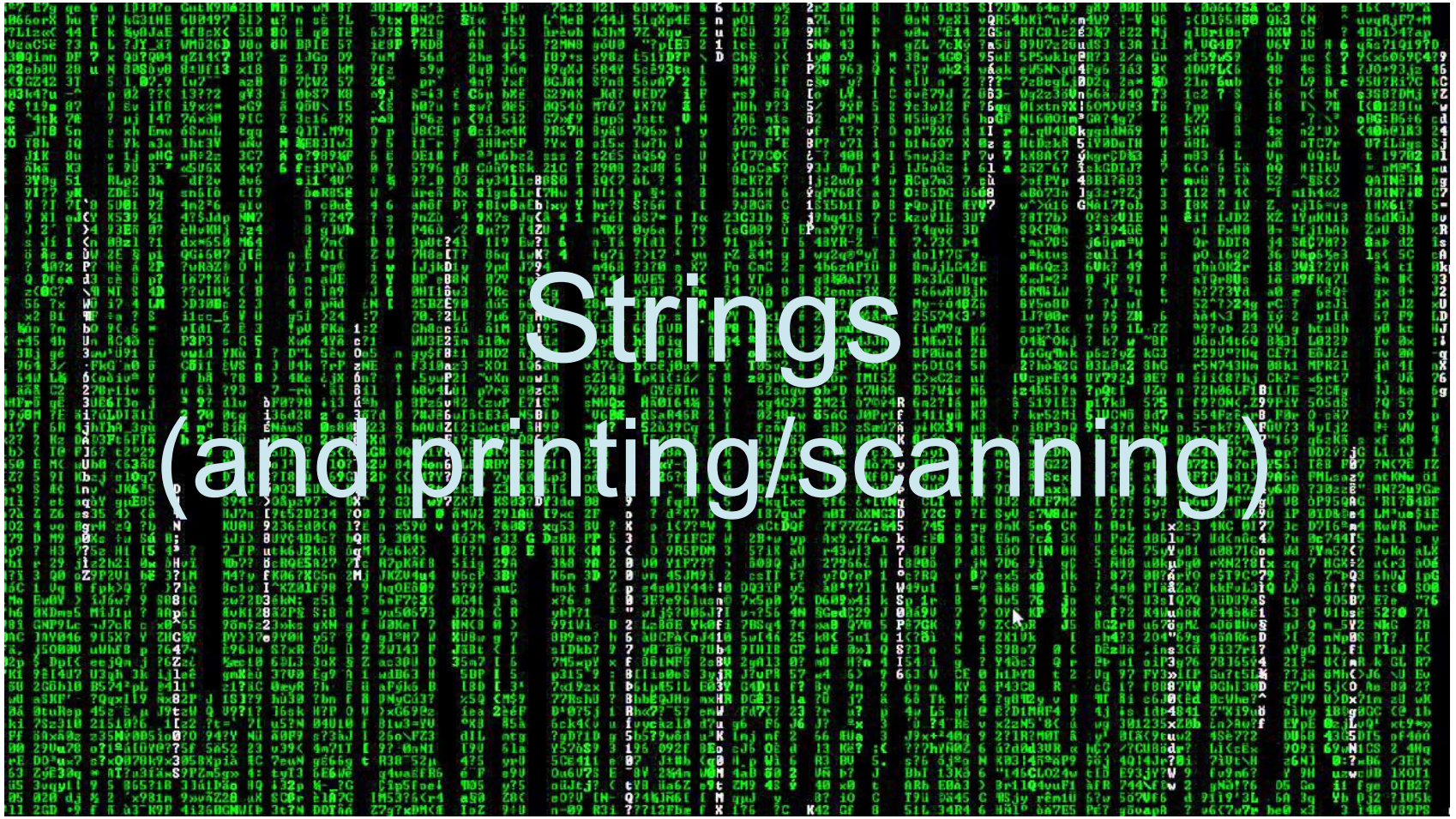
ESIGELEC
ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

C Programming



Strings (and printing/scanning)

Ángel Perles
angel.perles@upv.es
R04

Contents

Objectives
Introduction
ASCII encoding
Char data type and C strings
Escape sequences
Standard functions and macros
*printf() family
*scanf() family
Cajón de sastre

Objectives

- Text in C: ASCII, char, strings ended by '\0'
- Arrays vs. pointers to literals
- Escape sequences
- <string.h> and <ctype.h> functions
- Number ⇔ string conversion: strtol(), snprintf()
- Formatted output: *printf(), <inttypes.h>
- Safe formatting: snprintf() + truncation check
- Formatted input: *scanf(), return value, fgets() + sscanf()

INTRODUCTION

Introduction

- Machine thinking (digital) != human thinking (symbolic)
- So, a relation between human symbols and computer representation is a must
- This encoding is not immediate (different languages, different organisation of texts, etc.)
- Let's try to deal with it considering the limitations of C99



Merer papyrus

ASCII ENCODING

ASCII encoding

- ASCII: American Standard Code for Information Interchange
- A widely accepted character and control encoding format
- 7 bits encoding = 2^7 = 128 “American” symbols
- Extended to 8 bits (2^8 = 256) to support “code pages” and deal with international character sets
- But it is problematic for supporting international characters well
  - Alphabetical ordering
  - Accents, etc. support (“*Où est le garçon*?”)
  - Moving between international code pages (Niño (Spanish), Łukasz (Polsky)
- If feasible, use UNICODE encoding, e.g. UTF-8

ASCII encoding

- 7 bits table

Character Codes:

Dec	Hex	Sym	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Cha
0	0	NUL	32	20		64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	'	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	TAB	41	29	)	73	49	I	105	69	i
10	A	LF	42	2A	*	74	4A	J	106	6A	j
11	B	VT	43	2B	+	75	4B	K	107	6B	k
12	C	FF	44	2C	,	76	4C	L	108	6C	l
13	D	CR	45	2D	-	77	4D	M	109	6D	m
14	E	SO	46	2E	.	78	4E	N	110	6E	n
15	F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	□

ASCII Character & Control Codes:

Name	Char	Dec	Hex	Description
NUL	^@	00	00	null
SOH	^A	01	01	start of heading
STX	^B	02	02	start of text
ETX	^C	03	03	end of text
EOT	^D	04	04	end of transmission
ENQ	^E	05	05	enquiry
ACK	^F	06	06	acknowledge
BEL	^G	07	07	bell
BS	^H	08	08	backspace
HT	^I	09	09	horizontal tab
LF, NL	^J	10	0A	line feed, new line
VT	^K	11	0B	vertical tab
FF, NP	^L	12	0C	form feed, new page
CR	^M	13	0D	carriage return
SO	^N	14	0E	shift out
SI	^O	15	0F	shift in
DLE	^P	16	10	data link escape
DC1	^Q	17	11	device control 1
DC2	^R	18	12	device control 2
DC3	^S	19	13	device control 3
DC4	^T	20	14	device control 4
NAK	^U	21	15	negative acknowledge
SYN	^V	22	16	synchronous idle
ETB	^W	23	17	end of trans. block
CAN	^X	24	18	cancel
EM	^Y	25	19	end of medium
SUB	^Z	26	1A	substitute
ESC	^[	27	1B	escape
FS	^\	28	1C	file separator
GS	^]	29	1D	group separator
RS	^^	30	1E	record separator
US	^_	31	1F	unit separator

CHAR DATA TYPE AND C STRINGS

Char data type and C strings

- The data type *char* is the basic type to represent characters in C
- It is equivalent to a byte (integer), perhaps *signed*, perhaps *unsigned*, who knows
- E.g.

```
char symbol;

symbol = 'A';
symbol = symbol + 1;
```

- Note that the assigment *symbol = 'A'*; could be written

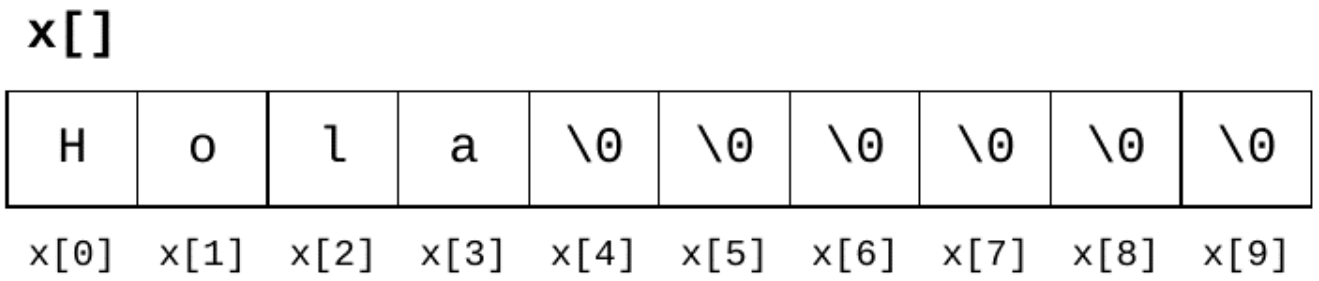
```
symbol = 65;     // silly ...
symbol = 0x41;   // ... programmer
```

- Please explain to me why?
- In any case, use *char* exclusively for text, and *int8_t/uint8_t* for numbers

Char data type and C strings

- A standard C string is a “null-terminated” array of chars.
- E.g., storing the string “Hola” in an array of chars will become

```
char x[10] = "Hola";
```



- Note:
  - The special char *’\0’* marks the end of the string
  - You can access the elements of the string. E.g. *x[3] = ’o’;* generates the string “Holo”, assigning *x[3] = ’\0’;* produces the string “Hol”
  - In C, *’A’* means a character, and *”A”* means the string formed by char ’A’ and ’\0’

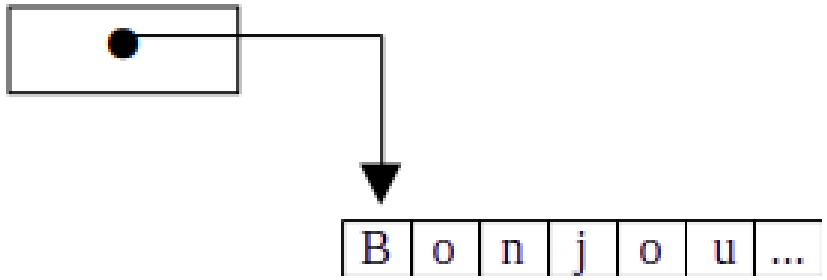
Char data type and C strings

- **Always** reserve an array size of at least n+1 for a string length of n chars
- E.g. you could write

```
char x[]   = "Hola";  // array of 5 chars
char x[10] = "Hola";  // array of 10 chars, right-filled with \0
char x[4]  = "Hola";  // this compiles without warnings and it's incorrect
```

- You can also create strings through a pointer

```
const char *p_msg = "Bonjour, M. Dupont";
```



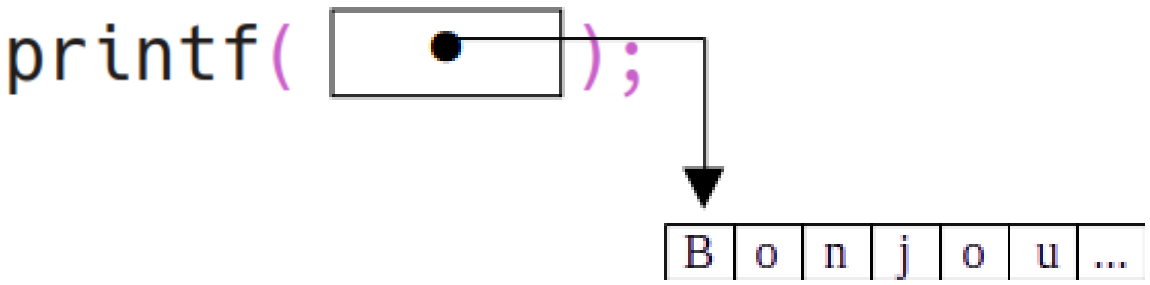
- The array is a modifiable copy; the pointer points to read-only text”

Char data type and C strings

- In fact, when you call a C function using a string as a parameter, the real behaviour is that the function receives a pointer to the string
- E.g., you see

```
printf("Bonjour, M. Dupont");
```

- but the reality is



- the function receives only a pointer, no size; the '\0' tells where the string ends

Char data type and C strings

- For Java
  - The *char* data type in Java is 16 bits wide
  - The encoding is unsigned UTF-16
  - *String* and *length()* are not available in C, only the \0 mark delimitates the string

Char data type and C strings

- Activity: Write a program that counts, in an integer variable, the number of 'a's in a character array. Bear in mind that the array does not necessarily have to be completely full.
- Modify the previous program so that it replaces the 'a's with 'o's in the string stored in the variable.
- Consider pointers for exploring the array

```
char text[1000] = "banana bread";
```



ESCAPE SEQUENCES

Escape sequences

- Some characters in the ASCII table are not symbols, and some of the symbols can't be directly written in your source code files
- Escape sequences solve this issue. They are based on using a backslash ('\') followed by something
- Typical ones are in the following table

Escape sequence	Meaning
\n	newline
\t	tab
\b	backspace
\r	carriage return
\f	form feed
\\	backslash
\'	single quote
\"	double quote
\0	octal escape of 0, end of C string
\xH...H	hexadecimal code of ASCII char

Escape sequences

- For example, to print double quotes, we can do

```
printf("Robespierre a été \"guillotiné\"");
```

- **BUT, NOTE**: Only use English in your source code, including comments. Use specialised libraries for dealing with multilingual capabilities
- Probably the accents will fail in this *printf()* example
- And the maximum flexibility is obtained using the *\xH..H* escape sequence
  - E.g., another way of expressing the 'A' char is

```
char symbol;

symbol = '\x41'; //'A'
```

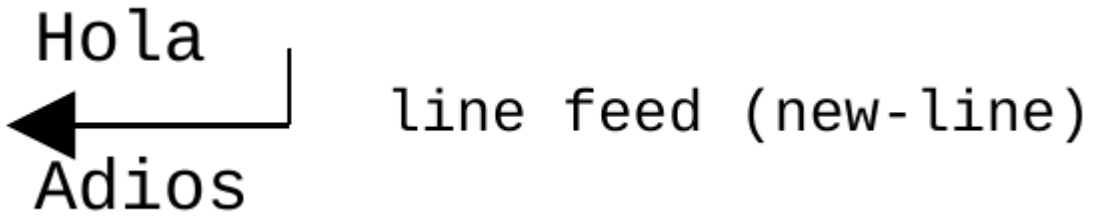
Escape sequences

- An **especially problematic** sequence is the *'\n'* one.
- Its meaning can be different between platforms, instruments, terminals, etc. … a hell
- Let's illustrate its behaviour using the following example on different platforms. Imagine the following "*printf*"

```
printf("Hola\nAdios");
```

Escape sequences

- Assume now ***nix derivatives**, Linux, Android, macOS, iOS …
- One character is required to represent a new line



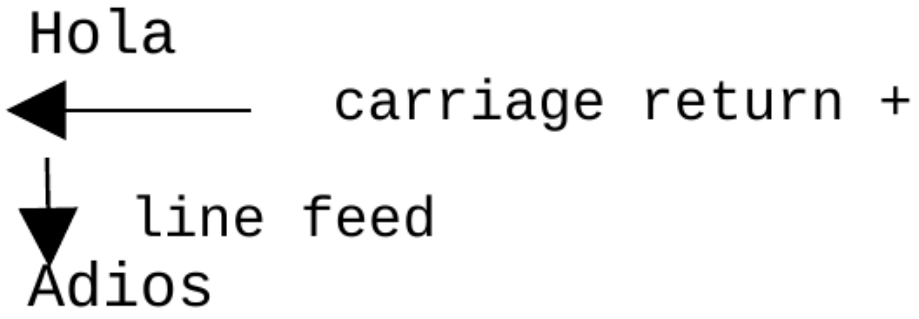
- The real string created by your compiler is

H	o	l	a	\x0A	A	d	i	o	s	\0

```
printf("Hola\nAdios");
```

Escape sequences

- Assume now **Microsoft Windows XX** (and old MS-DOS)
- printf() (text mode) translates it to \r\n when writing



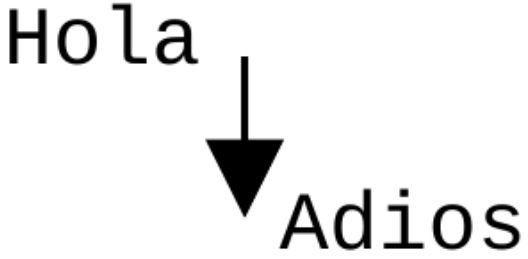
- The real string created by your compiler continues being

H	o	l	a	\x0A	A	d	i	o	s	\0

```
printf("Hola\nAdios");
```

Escape sequences

- So, if this text is generated on Linux, it can be displayed incorrectly in a terminal (e.g., PuTTY)



- It is recommended to avoid the *'\n'* sequence when dealing with embedded platforms such as modems, industrial control devices, etc.
- E.g., we could use the following two options in this case

```
printf("Hola\x0D\x0A" "Adios");

printf("Hola\r\nAdios");
```

STANDARD FUNCTIONS AND MACROS

Standard functions and macros

- A selection of functions to deal with standard C strings

Prototype	Header	Purpose
size_t strlen(const char *s);	string.h	Returns the number of characters in s, not counting the '\0'.
int strcmp(const char *s1, const char *s2);	string.h	Compares s1 and s2 character by character. Returns 0 if equal, <0 if s1 comes first, >0 otherwise.
char *strcpy(char *dest, const char *src);	string.h	Copies src, including the '\0', into dest. Returns dest. dest must be large enough.
char *strncpy(char *dest, const char *src, size_t n);	string.h	Copies at most n characters of src into dest. Returns dest. Warning: if src has n or more characters, no '\0' is added.
char *strcat(char *dest, const char *src);	string.h	Appends src to the end of dest. Returns dest. dest must be large enough.
void *memcpy(void *dest, const void *src, size_t n);	string.h	Copies n bytes from src to dest (any data, not only strings). Returns dest. The areas must not overlap.
char *strchr(const char *s, int c);	string.h	Returns a pointer to the first occurrence of character c in s, or NULL if not found.
char *strpbrk(const char *s1, const char *s2);	string.h	Returns a pointer to the first character in s1 that matches any character in s2, or NULL if none.
size_t strspn(const char *s1, const char *s2);	string.h	Returns the length of the initial part of s1 made only of characters in s2.
char *strstr(const char *s1, const char *s2);	string.h	Returns a pointer to the first occurrence of string s2 in s1, or NULL if not found.

Standard functions and macros

- Some random examples of use

```
char buffer[1000];

strcpy(buffer, "Hello");
strcat(buffer, " and goodbye");
```

Standard functions and macros

- Activity: Given the declarations of strings shown below, create the string “Astérix, Obélix et Idéfix” in *buffer*
- Calculate the length of the resulting string using *strlen()*

```
char buffer[1000];

const char *c1 = "Astérix";
const char *c2 = "Obélix";
const char *c3 = "Idéfix";
```



Standard functions and macros

- A selection of macros/functions to deal with standard C chars

Prototype	Header	Description
int isalpha(int c);	ctype.h	True if c is a letter: A-Z, a-z.
int isupper(int c);	ctype.h	True if c is an uppercase letter: A-Z.
int islower(int c);	ctype.h	True if c is a lowercase letter: a-z.
int isdigit(int c);	ctype.h	True if c is a decimal digit: 0-9.
int isxdigit(int c);	ctype.h	True if c is a hexadecimal digit: 0-9, A-F, a-f.
int isalnum(int c);	ctype.h	True if c is a letter or a digit: 0-9, A-Z, a-z.
int isspace(int c);	ctype.h	True if c is white space: ' ', '\t', '\n', '\v', '\f', '\r'.
int isblank(int c);	ctype.h	True if c is a blank: ' ', '\t'. (New in C99.)
int ispunct(int c);	ctype.h	True if c is a punctuation character: printable, but not a space, letter or digit (e.g. ! ? , _).
int isprint(int c);	ctype.h	True if c is printable, including the space (0x20-0x7E).
int isgraph(int c);	ctype.h	True if c is printable, excluding the space (0x21-0x7E).
int iscntrl(int c);	ctype.h	True if c is a control character (0x00-0x1F, 0x7F).
int toupper(int c);	ctype.h	Returns the uppercase version of c if c is a lowercase letter; otherwise returns c unchanged.
int tolower(int c);	ctype.h	Returns the lowercase version of c if c is an uppercase letter; otherwise returns c unchanged.

Standard functions and macros

- A selection of macros/functions to deal with standard C chars
- Notes:
  - In C99, these are functions, but the library may also implement them as macros.
  - The ranges shown apply to the default "C" locale (plain ASCII).
  - The *is...()* functions return an int: non-zero means true (not necessarily 1), zero means false.
  - c must be EOF or a value representable as unsigned char. With a plain char, cast it first: *isalpha((unsigned char)ch)*.

Prototype	Header	Description
int isalpha(int c);	ctype.h	True if c is a letter: A-Z, a-z.
int isupper(int c);	ctype.h	True if c is an uppercase letter: A-Z.
int islower(int c);	ctype.h	True if c is a lowercase letter: a-z.
int isdigit(int c);	ctype.h	True if c is a decimal digit: 0-9.
int isxdigit(int c);	ctype.h	True if c is a hexadecimal digit: 0-9, A-F, a-f.
int isalnum(int c);	ctype.h	True if c is a letter or a digit: 0-9, A-Z, a-z.
int isspace(int c);	ctype.h	True if c is white space: ' ', '\t', '\n', '\v', '\f', '\r'.
int isblank(int c);	ctype.h	True if c is a blank: ' ', '\t'. (New in C99.)
int ispunct(int c);	ctype.h	True if c is a punctuation character: printable, but not a space, letter or digit (e.g. ! ? , _).
int isprint(int c);	ctype.h	True if c is printable, including the space (0x20-0x7E).
int isgraph(int c);	ctype.h	True if c is printable, excluding the space (0x21-0x7E).
int iscntrl(int c);	ctype.h	True if c is a control character (0x00-0x1F, 0x7F).
int toupper(int c);	ctype.h	Returns the uppercase version of c if c is a lowercase letter; otherwise returns c unchanged.
int tolower(int c);	ctype.h	Returns the lowercase version of c if c is an uppercase letter; otherwise returns c unchanged.

Standard functions and macros

- A selection of macros/functions to deal with standard C chars (contd.)
- Remark the restrictions of these macros/functions dealing with non-"american" character sets (accents, ç, ñ, ü, …)
- You can, **more or less**, control some internationalisation aspects by changing the **"C" locale**
  - Include locale.h header file and use the setlocale() function
- E.g.

```
setlocale(LC_NUMERIC, "fr_FR"); // set locale French France
setlocale(LC_ALL, "C");         // set locale to standard C
```

- This also changes the behaviour of *printf()*, *scanf()*, etc.

Standard functions and macros

- Some random examples of use

```
if (isalnum('1'))
{
   printf("It is a letter or a digit\n"); // should show this
}
else
{
   printf("It is neither a letter nor a digit\n");
}
```

Standard functions and macros

- Activity: Analyse the string stored in the *char buffer[1000]* obtained from the previous exercise, and store in integer variables the number of uppercase letters, lowercase letters and spaces.
- Remember that it will contain the string “Astérix, Obélix et Idéfix”.
- Does the above string present any problems?



Standard functions and macros

- A selection of functions to deal with number <-> string conversions

Prototype	Header	Purpose
double atof(const char *s);	stdlib.h	Converts the initial part of s to a double. No error detection: returns 0.0 if s is not a number.
int atoi(const char *s);	stdlib.h	Converts the initial part of s to an int. No error detection: returns 0 if s is not a number; undefined behaviour if the value does not fit.
long atol(const char *s);	stdlib.h	Same as atoi(), but returns a long.
long strtol(const char *s, char **endptr, int base);	stdlib.h	Converts the initial part of s to a long in the given base (2-36, or 0 to accept the 0x and 0 prefixes). Stores in *endptr where the conversion stopped. On overflow sets errno to ERANGE.
unsigned long strtoul(const char *s, char **endptr, int base);	stdlib.h	Same as strtol(), but returns an unsigned long.
double strtod(const char *s, char **endptr);	stdlib.h	Converts the initial part of s to a double. Stores in *endptr where the conversion stopped. On overflow sets errno to ERANGE.
int snprintf(char *s, size_t n, const char *format, ...);	stdio.h	Writes formatted text, as printf() does, into s: at most n-1 characters plus the '\0'. Number to string: snprintf(buf, sizeof(buf), "%d", i). (New in C99.)

- Prefer the strto...() functions: unlike ato...(), they let you detect invalid input and overflow.
- endptr can be NULL if not needed. The whole string was a valid number if endptr != s and *endptr == '\0'.
- Before calling strto...(), set errno = 0 (errno.h) if you want to check it afterwards.

Standard functions and macros

- Some random examples of use

```
char *p_end;
strtol("12abc", &p_end, 10); // returns 12 and p_end pointing to 'a'
strtod("-2.5V", &p_end);     // returns -2.5 and p_end pointng to "V"
```

*PRINTF() FAMILY

*printf() family

- **IMPORTANT**: MISRA C:2012 Rule 21.6 “standard library input/output functions shall not be used”

Prototype	Header	Purpose
int printf(const char *format, ...);	stdio.h	Writes formatted text to the standard output (stdout).
int fprintf(FILE *stream, const char *format, ...);	stdio.h	Writes formatted text to a stream: stdout, stderr or a file opened with fopen().
int snprintf(char *s, size_t n, const char *format, ...);	stdio.h	Writes formatted text into the array s: at most n-1 characters plus the '\0'. Use it always to format into memory. (New in C99.)
int sprintf(char *s, const char *format, ...);	stdio.h	**Avoid**: like snprintf(), but it does not know the size of s, so it can write beyond the end of the array.

- Notes:
  - The arguments after the format are values (not pointers), one for each conversion specification (%d, %s, ...).
  - They return the number of characters written, or a negative value on error. *snprintf()* returns the number of characters it would have written: if the result is >= n, the text was truncated.
  - The compiler can check the arguments against the format only if the format is a string literal. Never write *printf(text);* write *printf("%s", text)*.

*printf() family

- Format specifiers

Specifier	Argument type	Output
%d	int	Signed decimal: -42
%u	unsigned int	Unsigned decimal: 42
%o	unsigned int	Octal: 255 → 377
%x %X	unsigned int	Hexadecimal, lower/upper case: 255 → ff / FF
%c	int (a char is promoted)	One character: 'A' → A
%s	const char *	The characters up to the '\0': Hola
%f	double (a float is promoted)	Decimal notation, 6 decimals by default: 3.141593
%e %E	double	Scientific notation: 3.141593e+00 / 3.141593E+00
%g	double	%f or %e, whichever is shorter, without trailing zeros: 0.5, 1e-07
%p	void *	Pointer value (address), implementation-defined format: 0x7ffe29c29380
%zu	size_t	Unsigned decimal for sizeof and strlen() results: 4 (New in C99.)
%%	(none)	A literal % character

- Notes:
  - Each specifier must match the type of its argument. gcc checks it (-Wall) when the format is a string literal: treat these warnings as errors.
  - For long, long long, int32_t, uint64_t, etc., see the length modifiers and the <inttypes.h> macros.
  - Cast pointers to (void *) when printing them with %p.

*printf() family

- Format specifiers (contd.)
- Syntax

```
%[flags][width][.precision][length]conversion
```

- [width] -> minimum number of chars to fill

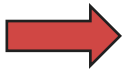
```
int i = 37;
printf("@%d@%4d@", i, i);
```



```
@37@  37@
```

- [flag] - -> aligns left

```
int i = 37;
printf("@%-4d@", i);
```



```
@37  @
```

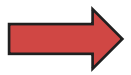
*printf() family

- Format specifiers (contd.)
- Syntax

```
%[flags][width][.precision][length]conversion
```

- [flag] 0 -> left fill with 0

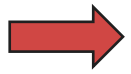
```
int i = 37;
printf("@%04d@", i);
```



```
@0037@
```

- [.precision] -> number of decimals

```
double lambda = 12.34567;
printf("Lambda is: %6.2f", lambda);
```



```
Lambda is: 12.35
```

*printf() family

- Format specifiers (contd.)
- Syntax

```
%[flags][width][.precision][length]conversion
```

- [length] -> size of the integer argument (long, long long, size_t)

Modifier	Argument type (signed / unsigned)	Use and example (output after →)
`l`	long / unsigned long	%ld %lu %lx `printf("%ld", -100000L) → -100000`
`ll`	long long / unsigned long long (New in C99.)	%lld %llu %llx `printf("%lld", 3000000000LL) → 3000000000`
`z`	size_t (New in C99.)	%zu `printf("%zu", strlen("Hola")) → 4`

*printf() family

- [length] … **is platform dependent** -> use *inttypes.h* to automate the selection

Macro	Argument type	Use and example (output after →)
PRIdN	intN_t (signed)	Signed decimal `printf("%" PRId32, temp) → -100000`
PRIuN	uintN_t (unsigned)	Unsigned decimal `printf("%" PRIu16, raw) → 4095`
PRIxN PRIXN	uintN_t (unsigned)	Hexadecimal, lower/upper case `printf("0x%08" PRIX32, reg) → 0x00000FFF`

- Notes:
  - N is the width in bits: 8, 16, 32 or 64 (e.g. PRId8, PRIu64, PRIX16). Include <inttypes.h>, which also includes <stdint.h>.
  - Each macro is a string literal with the right length modifier and conversion for your platform (e.g. "u", "lu" or "ld"). The % and any flags or width stay outside, in the format string: "%" PRIu32, "%08" PRIX32. The compiler joins the adjacent literals.
  - ~~For scanf(), the equivalent macros are SCNdN, SCNuN and SCNxN (e.g. SCNu8).~~

*printf() family

- Recommended programming pattern for *snprintf()*

```c
#include <inttypes.h>

uint32_t value = 4000000000UL;
char buffer[16];

int len = snprintf(buffer, sizeof(buffer), "Value: %" PRIu32, value);
if ((len < 0) || ((size_t)len >= sizeof(buffer)))
{
    /* error or truncated */
}
```

- It should generate "Value: 40000000“ and *len* = 17 (truncated string)
- For our particular configuration GCC, this problem is also detected at compile time

* printf() family

- Activity: Complete the code below to generate the string indicated in the comments using the function *snprintf()* on the *formatted* array

```
const double pi = 3.141592;
const int i = 3;
const char *p_word = "three";
char formatted[100];
/* Goal: "PI is 3.14, and the word three is the number 0003" */
```



* printf() family

- Activity: Implement a simple hex dump format function using the prototype shown below, where:
  - The output format is **:XXAABBCCDDEE...NN$**
  - **XX** is the number of bytes in hex left filled with 0
  - **AA, BB, …, NN** are "bytes" represented in 2-digit hex
  - Tip: call *snprintf()* several times, advancing a position in *p_dest*

```
int dump_hex(char *p_dest, size_t dest_size, const uint8_t *p_data, uint8_t num_data);
/* returns 0 on success, -1 if p_dest is too small */
```

- Usage example generating the formatted string: *:05000F107FFF$*

```
const uint8_t data[] = {0, 15, 16, 127, 255};
char formatted[100];

if (dump_hex(formatted, sizeof(formatted), data, sizeof(data) != 0)
{
    // problems!!!
}
```



*SCANF() FAMILY

*scanf() family

- Read formatted data: the inverse of **printf()*

Prototype	Header	Purpose
int scanf(const char *format, ...);	stdio.h	Reads formatted data from the standard input (stdin, usually the keyboard).
int fscanf(FILE *stream, const char *format, ...);	stdio.h	Reads formatted data from a stream: stdin or a file opened with fopen().
int sscanf(const char *s, const char *format, ...);	stdio.h	Reads formatted data from the string s in memory.
char *fgets(char *s, int n, FILE *stream);	stdio.h	Reads one line (at most n-1 characters, keeping the '\n') into s and adds the '\0'. Returns NULL at the end of the input or on error.

- Notes:
  - The arguments after the format are **pointers**: &value (or the array name for %s).
  - They return the **number of items assigned** (or EOF if the input ends before the first conversion) -> always check it
  - Always give %s a width smaller than the array: *char word[6]* → "*%5s*" (prevents writing beyond the end of the array).
  - Keyboard input: read the line with *fgets()* and then analyse it with *sscanf()*

*scanf() family

- Format matching
  - A space in the format matches any amount of whitespace (even none)
  - *%d %x %f %s* skip leading white space, *%c* does not
  - Any other character in the format must appear in the input
  - A conversion stops at the first character that does not fit
  - * in the format lets “jump” matches, e.g., %*d reads a value but does not assign it

```
sscanf("19 32", "%d%d", &a, &b);     /* 2: a = 19, b = 32 */
sscanf("19,32", "%d,%d", &a, &b);    /* 2: a = 19, b = 32 */
sscanf("19;32", "%d,%d", &a, &b);    /* 1: ';' is not ',' */
sscanf("12 34", "%*d %d", &a);       /* 1: a = 34 */
```

*scanf() family

- Example: extract the signal quality of an AT-compatible modem using the command AT+CSG (E.g. a cellular modem, a satellite modem …)

```
const char *p_resp = "+CSQ: 21,99";     /* signal quality */

int rssi;
int ber;

if (sscanf(p_resp, "+CSQ: %d,%d", &rssi, &ber) != 2)
{
    /* unexpected answer, e.g. "+CSQ: ERROR" */
}
```

*scanf() family

- Typical pitfalls
  - **%f** → float *, **%lf** → double * (*printf()* does not care, ***scanf()*** **does!**)
  - *uint8_t*, *uint16*_t…: use SCNu8, SCNx16… (**inttypes.h**)
  - %s reads one word and needs a width: char unit[4] → "%3s"
  - %c reads one char: **no whitespace skipped**, **no '\0' added**

```
/* p_text = "3.27 V ch 2" */
double  voltage;
char    unit[4];
uint8_t channel;

n = sscanf(p_text, "%lf %3s ch %" SCNu8, &voltage, unit, &channel);
/* n = 3: 3.27, "V", 2 */
```

*scanf() family

- Suggested pattern for reading from keyboard
  - *scanf() leaves the '\n' in the input and gets stuck on invalid data*
  - *Robust pattern: read the whole line with fgets(), then analyse it with sscanf()*

- E.g. enhancement to the lab's cycles read

```
if (fgets(line, (int)sizeof(line), stdin) == NULL)
{
    /* end of input or error */
}
else if (sscanf(line, "%d", &cycles) != 1)
{
    /* not a number: ask again */
}
```

*scanf() family

- Activity: analyse the string `"a03451"` with a single *sscanf()* call so that:
  - the 1st character goes to a `char` variable
  - the 2nd to 5th characters, read as a decimal number, go to an `int`
  - the 6th character, read as a number, goes to another `int`
  - check that *sscanf()* returns 3
- Hint: a width limits how many characters a conversion reads: `%4d`



*scanf() family

- Activity: write a function that asks for the number of cycles (1–6) until the user types a valid value
  - use *fgets()* + *sscanf()* (no *scanf()*)
  - reject text (`abc`) and out-of-range values (`9`)
  - return -1 at the end of the input

```
int read_cycles(void);
```



CAJÓN DE SASTRE

Cajón de sastre

- To consider on next courses?
- Encode source files using UTF-8 encoding

ESIGELEC

ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA