



# I caratteri e le stringhe

---

Ver. 3

# Codice ASCII

- Per memorizzare i simboli grafici corrispondenti ai caratteri bisogna associare un numero intero a ciascuno di essi
- Il codice ASCII (pron. /'æski/), American Standard Code for Information Interchange, è una *tabella* che elenca le corrispondenze tra simboli grafici e numeri (es. A → 65, B → 66)
- I numeri del *Codice ASCII standard* sono a 7 bit →  $2^7=128$  caratteri diversi

# Codice ASCII standard

127		Altri caratteri
123		
122	z	Lettere minuscole (a-z)
97	a	
96		Altri caratteri
91		
90	Z	Lettere maiuscole (A-Z)
65	A	
64		Altri caratteri
58		
57	9	Cifre (0-9)
48	0	
47		Altri caratteri
33		
32		Carattere SPAZIO
31		<i>Caratteri di controllo</i>
0		

# Codice ASCII standard

---

- I primi 32 caratteri (0-31) sono detti *caratteri di controllo*, non sono visualizzati, ma producono un *effetto* (es. inseriscono un ritorno a capo, fanno emettere un beep, rappresentano il carattere EOF, ecc.)
- I successivi caratteri (tranne il 127) sono invece visualizzabili, ossia hanno un aspetto grafico

# Codice ASCII standard

- I caratteri visualizzabili sono suddivisi in 4 sezioni rilevanti separate da altri caratteri generici (punteggiatura, simboli matematici, ecc.)

```
!"#$%&'()*+,-./  
0123456789:;<=>?  
@ABCDEFGHIJKLMNO  
PQRSTUVWXYZ[\]^_  
`abcdefghijklmnop  
qrstuvwxyz{|}~
```

- In ordine crescente di valore numerico le sezioni sono:
  - Spazio (è il caract. visibile di codice più basso: 32)
  - Cifre (0-9 in ordine crescente, '0' ha codice 48)
  - Maiuscole (A-Z in ordine cresc., 'A' ha codice 65)
  - Minuscole (a-z in ordine cresc., 'a' ha codice 97)

# Codice ASCII standard

- I seguenti caratteri:
  - spazio (ASCII 32)
  - tabulazione orizzontale (ASCII 9)
  - tabulazione verticale (ASCII 11)
  - new line '\n' (ASCII 10)
  - carriage return '\r' (ASCII 13)
  - salto pagina (ASCII 12)

sono collettivamente chiamati dallo Standard con il termine *white spaces*, si noti che a parte lo spazio sono tutti codici di controllo (quindi hanno valore numerico inferiore a quello dello spazio)

# Codice ASCII standard

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	\$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(ff)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c		124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

# Codice ASCII standard

- È utile considerare quanto segue:
  - il codice ASCII delle cifre 0-9 si può ottenere dal corrispondente valore binario su 4 bit facendolo precedere dal valore **011**, ossia sommando  $0110000_2 = 48$ :  
 $7 = 0111 \rightarrow 0110111 = 55$  (codice ASCII di 7)  
ossia  $7 + 48 = 55$
  - Analogamente sottraendo 48 dal codice ASCII di una cifra si ottiene il valore di quella cifra:  
 $55 - 48 = 7$
  - La differenza tra una lettera minuscola e la corrispondente maiuscola è sempre pari a 32



# Codice ASCII esteso

- Poiché l'unità base di memorizzazione è il byte e il codice ASCII standard richiede solo 7 bit, i caratteri del codice ASCII standard hanno sempre il primo bit a sinistra (MSB – Most Significant Bit) pari a 0 (codici da 0 a 127)
- Per usare tutti i 256 possibili valori di un byte, ai 128 caratteri del codice ASCII standard vengono aggiunti altri 128 caratteri non standard, questi hanno MSB pari a 1 e costituiscono il *codice ASCII esteso*

# Codice ASCII esteso

- I 128 caratteri aggiuntivi (codici da 128 a 255) vengono utilizzati per altri simboli grafici meno comuni (es. le lettere accentate, à, ë, î)
- La parte aggiuntiva:
  - non è standard
  - è diversa per ciascun sistema operativo
  - può essere diversa anche nello stesso sistema operativo a seconda della lingua
  - è talvolta indicata con il termine *codepage*

# Caratteri

- Nel Linguaggio C simboli dei caratteri sono memorizzati con il tipo `char`, quantità di tipo intero (se `signed` o `unsigned` dipende dal compilatore)
- Il valore numerico dei caratteri è quello indicato nella tabella del codice ASCII, ad es.
  - il carattere `A` viene memorizzato come 65
  - il carattere `a` viene memorizzato come 97
  - il carattere `0` viene memorizzato come 48
  - il carattere spazio viene memorizzato come 32

# Costanti carattere

- Le costanti di tipo carattere vengono anche chiamate *letterali carattere* (*character literal*)
- Le costanti carattere sono indicate tra apici *singoli*, es. 'A', il valore numerico corrispondente è quello del codice ASCII
- Essendo numeri interi, i caratteri possono essere usati nelle operazioni aritmetiche, quindi in un'espressione scrivere 'A' è del tutto equivalente a scrivere il valore 65, ma si ricordi che nei calcoli i caratteri vengono convertiti in `int` per effetto delle *promozioni integrali*

# Costanti carattere

- Riprendendo l'esempio precedente, poiché  $'0' = 48$  invece di scrivere  $7 + 48 = 55$  si può scrivere  $7 + '0'$  ottenendo sempre 55 che è il codice ASCII del carattere  $'7'$
- Utilizzo tipico è quello opposto, ossia ricavare il valore numerico da una variabile  $c$  contenente il codice ASCII di una cifra:  
se  $c = '7'$  ( $=55$ ), allora  $c - '0'$  viene calcolato come  $55 - 48$  dando il valore 7, questo è di tipo `int` per effetto delle promozioni integrali

# Sequenze di escape

- Alcuni caratteri, essendo utilizzati nella sintassi del linguaggio, devono essere gestiti in modo particolare, questi sono:
  - l'apice ' che serve per racchiudere singoli caratteri,
  - le virgolette " che servono per racchiudere le *stringhe costanti* (sequenze di caratteri),
  - il punto interrogativo ? che serve per definire le *espressioni condizionali*
  - la barra rovesciata (backslash) \ che serve per definire le *sequenze di escape*

# Sequenze di escape

- I caratteri *virgolette* e *punto interrogativo* si possono indicare normalmente tra apici, ad esempio: `'"'` e `'?'`
- I caratteri *apice* e *punto interrogativo* *all'interno di stringhe costanti* (ossia tra virgolette) si possono indicare normalmente, ad esempio: `"L'aquila?"`
- Ma quando quei caratteri devono essere utilizzati sia per i costrutti del linguaggio sia come veri e propri caratteri, allora è necessario usare le *sequenze di escape*

# Sequenze di escape

- Le *sequenze di escape* sono composte da due o più caratteri di cui il primo è il carattere barra rovesciata o *backslash* \ detto anche *carattere di escape*
- Per indicare il carattere *apice* come costante carattere, dato che l'apice stesso è il delimitatore delle costanti carattere (come in ' **a** '), è necessario scrivere ' \ ' '
- Il *carattere di escape* \ indica che il carattere seguente (l'apice rosso) è un carattere normale e non un *delimitatore*, il carattere \ non viene memorizzato



# Sequenze di escape

- Analogamente, per indicare il carattere *virgolette* all'interno di una stringa costante, dato che le virgolette stesse sono i delimitatori delle stringhe costanti, è necessario scriverle precedute dal *carattere di escape* `\`, ad es. :  
`"premi \"invio\" per terminare"  
i carattere di escape \ non sono memorizzati`
- Lo stesso backslash `\`, se si deve inserire come carattere normale in una costante carattere o in una stringa costante deve essere sempre scritto raddoppiato, ad esempio:  
`'\\'` e `"C:\\esercizi\\file.txt"`  
i *carattere di escape* `\` non sono memorizzati

# Sequenze di escape

- Alcuni dei *caratteri di controllo* del codice ASCII possono essere ottenuti mediante *sequenze di escape*:
  - \a – quando viene visualizzato fa emettere un *beep*
  - \n – corrisponde al carattere *new line* (10)
  - \r – corrisponde al carattere *carriage return* (13)
  - \t – corrisponde al carattere *Tab*
- Gli altri non sono di uso comune, ma possono essere ottenuti tramite *sequenze di escape ottali o esadecimali* a partire dai codici ASCII corrispondenti

# Sequenze di escape oct/hex

- Tutti i caratteri possono essere indicati come *sequenze di escape ottali e esadecimali*, queste devono produrre un valore compatibile con un tipo `unsigned char` (ossia minori o uguali a  $255_{10}$ ,  $377_8$ ,  $FF_{16}$ )
- Una *sequenza di escape ottale* inizia con `\` ed è composta da 1 a 3 cifre ottali, ad es. `'\33'` o `'\033'` (lo 0 iniziale non è necessario, era necessario per le costanti intere)
- Una *sequenza di escape esadecimale* inizia con i caratteri `\x` (`x` minuscola e non `\0x`) ed è composta da 1 o 2 cifre esadecimali (maiuscole o minuscole), ad esempio `'\x4f'`

# Sequenze di escape oct/hex

- I seguenti modi sono equivalenti per esprimere la lettera A o il numero 65:
  - numero decimale: 65
  - carattere: 'A'
  - numero ottale: 0101
  - escape ottale: '\101'
  - numero esadecimale: 0x41
  - escape esadecimale: '\x41'
- La forma da usare di volta in volta dipende dall'utilizzo che si intende fare di tali valori

# Variabili carattere

- Le variabili carattere sono di tipo `char`, quantità di tipo intero (`signed` o `unsigned`, dipende dal compilatore) su 8 bit  
`char y;`
- Il valore può essere assegnato in tutte le forme viste precedentemente, in caso di valori costanti è preferibile siano assegnate costanti carattere: `y = 'A'` è più comprensibile di `y = 65` se si tratta di un carattere.  
In ogni caso `y` contiene il valore 65 che è il codice ASCII del carattere `'A'`

# Variabili carattere

- `y` è a tutti gli effetti un numero (nell'esempio 65), ma viene trattato come simbolo (`'A'`) da alcune funzioni

- Si consideri la seguente funzione:

```
printf("%c ha valore %d\n", y, y);
```

La specifica `%c` serve per visualizzare un carattere (ne considera il codice ASCII), mentre la specifica `%d` visualizza un numero intero, la `printf` visualizza la stessa variabile `y` in due modi diversi:

```
A ha valore 65
```

# Variabili carattere

---

- Il linguaggio C definisce anche:
  - le **sequenze trigrafiche** (*trigraph sequences*), combinazioni particolari di caratteri che permettono di rappresentare alcuni caratteri non presenti in alcune tastiere internazionali
- Non essendo di utilizzo comune, non sono trattate in queste slide e si rimanda alla bibliografia

# Stringhe

- Sono *vettori di char* terminati da un carattere **null** di valore 0, codice ASCII 0, si preferisce esprimerlo nella forma `'\0'` (ottale)  
Non è il carattere `'0'` (48 nel codice ASCII)

C	I	A	O	\0
---	---	---	---	----



67	73	65	79	0
----	----	----	----	---

U	N		P	O	'	\0
---	---	--	---	---	---	----



85	78	32	80	79	39	0
----	----	----	----	----	----	---

- Le stringhe terminate da uno 0 (`'\0'`) vengono anche dette *stringhe ASCIIZ*, il linguaggio C ha solo questo tipo di stringhe



# Stringhe costanti

- Sono sequenze di `char` racchiuse da *virgolette* (anche chiamate *doppi apici*), la stringa `"ciao ciao"` è composta da 9 caratteri, più uno non visibile ma presente in memoria: il **null** `'\0'` di terminazione

Come già detto, per includere in una stringa i caratteri `\` e `"` è necessario farli precedere dal carattere di escape `\`


- `"vero\\falso"` memorizza: `vero\falso`
- `"premi \"invio\""` per terminare"  
memorizza: `premi "invio" per terminare`

# Stringhe costanti

- Lo standard usa il termine *letterale stringa* (*string literal*) e non stringa costante perché, anche se viene normalmente utilizzata come stringa costante, esso non richiede che sia *realmente* immutabile, in generale:

- la modifica può portare a inconvenienti: è lecito per un compilatore memorizzare letterali uguali in unica copia, quindi la modifica di uno potrebbe cambiare gli altri (l'effetto è *indeterminato*)

```
puts("ciao");  
strlen("ciao");
```



The diagram illustrates that both the `puts` and `strlen` functions in the code above point to the same memory location. This memory location contains the characters 'c', 'i', 'a', 'o', and a null terminator '\0'.

- il compilatore può avere impostazioni per controllare se sia possibile o no modificarle

# Stringhe costanti

- Nota questa distinzione, per semplicità nel seguito verranno utilizzati con lo stesso significato sia il termine *costante stringa*, sia il termine *letterale stringa*
- Lo standard C89 indica che una stringa costante deve poter contenere almeno 509 char
- Una stringa costante rimane in memoria per tutta l'esecuzione del programma, il suo valore è definito una volta sola ed è inizializzata prima dell'esecuzione del programma (si dice che ha *storage duration* `static`)

# Stringhe costanti

- Più costanti stringa consecutive (separate da nulla, spazi, Tab, ritorni a capo o commenti (che sono considerati pari ad un unico spazio) vengono *concatenate* dal compilatore in un'unica stringa

Le 4 stringhe nelle seguenti 3 righe:

```
"ciao" ..... "ciao" .....  
..... "ciao"  
..... "ciao"
```

vengono memorizzate senza aggiungere spazi intermedi come se fossero un'unica stringa di 17 caratteri (con un unico \0 finale):

```
"ciaociaociaociao"
```

# Stringhe costanti

- Un secondo metodo per spezzare su più righe una stringa consiste nel terminare tutte le righe tranne l'ultima con il carattere `\` e continuare a capo *senza spazi iniziali di indentazione*:

```
printf("Ciao co\  
me stai?");
```
- Tale metodo (detto *splicing*) ha evidenti inconvenienti di formattazione e leggibilità ed è sconsigliabile

# Sequenze di escape in stringhe

- Le sequenze di escape **ottali** utilizzate in una stringa terminano dopo al massimo 3 caratteri ottali o al primo carattere non ottale, quindi:
  - `"\1234"` contiene i due caratteri: `'\123'` ottale e 4 decimale
  - `"\183"` contiene i 3 caratteri: `'\1'` ottale, 8 e 3 decimali

# Sequenze di escape in stringhe

- Le sequenze di escape **esadecimale** in una stringa invece non sono limitate a 1-2 caratteri come nelle costanti carattere, ma terminano solo quando trovano un carattere non hex:
  - `"Z\xfc rich"` contiene i 6 caratteri `'Z'`, `'ü'`, `'r'`, `'i'`, `'c'`, `'h'` (`"Zürich"`) in quanto `'\xfc'` corrisponde al carattere Unicode `'ü'` e il carattere successivo `'r'` non è cifra esadecimale
  - `"\xfcber"` NON corrisponde a `"über"` (che si ottiene invece concatenando `"\xfc"` e `"ber"`) ma ai 2 caratteri: `'\xfcbe'` (dopo la `'c'` ci sono `'b'` ed `'e'` che sono *anche* cifre esadecimale e quindi il tutto viene interpretato come un unico valore) ed `'r'` (non esadecimale) → errore

# Stringhe variabili

- Sono vettori di `char` di dimensione costante (ossia nota al *compile-time*) e contenuto modificabile

```
char nome[15];
```

definisce una variabile stringa di 15 `char`

- Una stringa variabile può contenere un numero di caratteri minore della sua capacità, l'importante è che dopo l'ultimo carattere ci sia il carattere terminatore **null** `'\0'`
- Nell'esempio precedente `nome` può contenere *fino a* 14 caratteri utili perché deve esserci spazio per il **null** terminale



# Stringhe variabili

- Il terminatore permette di occupare solo parzialmente una stringa (lo spazio relativo ai restanti caratteri esiste ma è inutilizzato)

nome :

C	I	A	O	\0	?	?	?	?	?	?	?	?	?	?
---	---	---	---	----	---	---	---	---	---	---	---	---	---	---

- La *lunghezza* di una stringa è il numero di caratteri contenuti (fino al **null** escluso), non la sua dimensione ("capienza"), nell'esempio sopra la lunghezza è 4 e la dimensione 15

# Stringhe variabili

- Come per i vettori, la dimensione (la lunghezza massima) di una stringa viene di solito indicata con una `#define`
- Nella definizione della stringa stessa talvolta si inserisce `+1` per rendere più esplicito che il numero di caratteri significativi che può contenere è quello indicato dalla direttiva `#define`, il `+1` riserva spazio per il necessario **null** terminale:

```
#define MAXSTR 80  
char mystring[MAXSTR+1];
```

# Stringhe variabili

- Una stringa, essendo un vettore, può essere *inizializzata* elencando i singoli elementi:  

```
char s[N]={'C','i','a','o'};
```
- Ma normalmente si preferisce il modo specifico delle stringhe, identico al precedente a tutti gli effetti, ma più semplice e chiaro:  

```
char s[N]="Ciao";
```
- Attenzione che si tratta di una *inizializzazione*, NON di un'assegnazione (non si può assegnare una stringa al run-time con un semplice = )

# Stringhe variabili

- In entrambi i casi, se  $N$  è maggiore del numero di caratteri di inizializzazione allora gli elementi non inizializzati esplicitamente ( $s[4]$  e successivi) vengono inizializzati automaticamente tutti a 0 (come già visto per gli altri vettori)
- Ma 0 e `'\0'` sono solo due modi diversi di indicare lo stesso valore (**null**), quindi la stringa è correttamente terminata da un **null**

# Stringhe variabili

- Se il numero di caratteri di inizializzazione è maggiore o uguale alla dimensione della stringa (in questo esempio  $N=4$ ), vengono memorizzati solo i primi  $N$  caratteri della stringa di inizializzazione e il terminatore **null** non viene aggiunto

```
char s[4]="Ciao";
```

```
char s[4]="CiaoABCD";
```

in entrambi questi esempi  $s$  contiene solo "Ciao" senza il terminatore **null**

# Stringhe variabili

- Si noti che nell'inizializzazione:  
`char s[4]="Ciao";`  
"Ciao" non è un letterale stringa, ossia al run-time non è possibile recuperarne l'indirizzo di memoria, non è neppure detto abbia un **null** di terminazione

# Stringhe variabili

- Essendo una stringa un vettore, i suoi elementi si indicano con la notazione dei vettori:  
nome[0] è il 1<sup>o</sup> carattere  
nome[1] è il 2<sup>o</sup> carattere  
ecc.
- I singoli elementi della stringa sono `char`, se la stringa `s` contiene "Ciao":  
s[2] vale 'a'  
s[3] = 'k'; modifica s in "Ciak"
- Per *assegnare* una stringa NON si può scrivere `s="Ciao"` ma è necessario usare una funzione (es. `strcpy`, vedere più avanti)

# Stringhe variabili

- Si noti la differenza tra:
  - 'a' → il *carattere* 'a' (il *numero* 97)
  - "a" → la *stringa* contenente 'a' e '\0'
- Essendo una stringa un vettore di `char`, il nome di una stringa viene usato dal compilatore come sinonimo dell'*indirizzo di memoria* del primo carattere



# Stringhe variabili

- Per quanto non molto utilizzato, anche un letterale stringa può essere indicizzato
- Esempi
  - `hex = "0123456789ABCDEF"[val_decim];`  
dà la cifra esadecimale corrispondente al numero decimale `val_decim`: il valore `val_decim` identifica l'elemento della stringa (carattere) in quella posizione, quindi viene estratto e memorizzato in `hex`,  
ad es. se `val_decim` vale 10 restituisce 'A'
  - `printf("%d numer%c\n", a, "io"[a==1]);`  
se `a==1` dà risultato vero ossia 1 e con questo indicizza "io" dando "o", altrimenti dà "i"

# I/O di caratteri e stringhe

- Tutte le funzioni di input e output (I/O) di caratteri e stringhe sono contenute nella stessa libreria generica già vista per i valori numerici
- La sintassi delle funzioni e i valori simbolici necessari anche in questo caso sono contenuti nel file di intestazione `stdio.h`

# Output di caratteri

- `putchar (var) ;`
- `printf ("%c", var) ;`  
Entrambe le funzioni visualizzano il carattere [il cui codice ASCII è] contenuto in *var*
- *var* può essere una variabile, una costante o un'espressione di qualsiasi tipo intero (`char`, `int`, `short`, `long`, ecc.)
- `putchar` è più veloce della `printf` in quanto è una funzione molto meno complessa, talvolta realizzata con macro (vedere slide sul preprocessore)

# Input di caratteri

- `varInt = getchar();`  
`getchar` restituisce il carattere letto o `EOF` per segnalare la fine dell'input
- `varInt` deve essere di tipo `int` in quanto:
  - i caratteri sono di 8 bit e la `getchar` può restituire ciascuna delle 256 possibili combinazioni
  - per indicare la fine dell'input deve restituire un valore che non fa parte di quei 256, pertanto serve una variabile con più di 8 bit, di cui la `getchar` riempie solo il byte basso, mentre gli altri sono a 0
  - `EOF` è una costante simbolica definita con una `#define` in `stdio.h` e in genere vale `-1`, quindi è di tipo `int` e tutti i suoi bit sono 1 (in Compl. a 2)



# Caratteri ed EOF

- Se per errore si definisce la variabile per `getchar` di tipo `char`, spesso (ma non sempre) tutto *funziona*, ad esempio in:

```
if ((c=getchar()) !=EOF)
```
- Funziona SOLO SE contemporaneamente si verificano le seguenti condizioni:
  - EOF vale `-1` (in Compl. a 2 è `11...11`, es. 32 bit)
  - la `getchar` non riceverà mai in input il carattere corrispondente a `11111111` (`-1` in CA2 su 8 bit)
  - le variabili `char` sono in realtà `signed char` (dipende dal compilatore se `char` corrisponde a `signed char` o `unsigned char`)

# Caratteri ed EOF

- Allora nel caso le condizioni siano verificate:
  - se `c` è `11111111` (-1 in Complemento a 2 su 8 bit) questo non può che derivare dall'EOF
  - nel confronto `c == EOF`, prima del confronto stesso `c` viene convertito in un `int` e se vale `11111111` l'`int` viene ottenuto aggiungendo degli 1 a sinistra (estensione del segno), ossia producendo un valore composto di tutti 1 (`1111...111`) ossia -1, di fatto uguale a EOF

Quindi la condizione riconosce correttamente l'EOF.

# Caratteri ed EOF

- Ma se anche una sola delle condizioni non è vera, definire la variabile come `char` può portare a errori:
  - se l'input contiene il carattere `11111111`, ad es. non viene da tastiera, ma da file con la redirectione dell'input, questo viene interpretato erroneamente come EOF
  - Se `char` è un `unsigned char`, quando la `getchar` dà EOF, nel confronto `c==EOF` si ha che EOF viene memorizzato in `c` come `11111111` (255), ma prima del confronto `c` viene convertito in `int` aggiungendo degli 0 a sinistra (`000...00011111111`) e questo non corrisponde a `111...111`, l'EOF non viene riconosciuto



# Input di caratteri

- `scanf ("%c", &varChar) ;`  
La specifica `%c` legge 1 carattere (*anche se è uno spazio o un ritorno a capo*, ossia non li salta come le altre specifiche di conversione della `scanf`) e lo mette (o meglio mette il suo codice ASCII) in *varChar*
- *varChar* può essere una variabile intera di qualsiasi tipo, ma il tipo `char` è appropriato
- Per leggere un carattere saltando i white spaces iniziali si mette uno spazio iniziale nella specifica di conversione " `%c`" oppure si usa "`%1s`" (ma questo richiede una var. stringa)

# Input di numeri e caratteri

- Si sottolinea ancora che le specifiche di conversione diverse da `%c` saltano i white spaces iniziali, anche nel caso la `scanf` non riesca poi a leggere e assegnare un valore

Si abbia ad esempio il seguente codice:

```
x = scanf ("%d", &n) ;
```

```
y = scanf ("%d", &m) ;
```

```
z = scanf ("%c", &c) ;
```

e in input venga immessa la stringa:

```
" 12 a"
```

Allora...

# Input di numeri e caratteri

## ■ Allora:

- nella prima `scanf` la specifica `%d` consuma gli spazi iniziali, legge 12 e lo mette in `n`, si ferma al primo spazio dopo il 2, lo spazio resta da leggere; la `scanf` restituisce 1
- nella seconda `scanf` la specifica `%d` consuma gli spazi dopo il 12 fino al carattere 'a' e si ferma, non avendo trovato cifre da poter convertire in numero non assegna nulla a `m` che mantiene il valore che aveva già, il carattere 'a' resta da leggere; la `scanf` restituisce 0
- nella terza `scanf` la specifica `%c` legge il primo carattere lasciato nel buffer cioè la 'a'; la `scanf` restituisce 1

# Output di stringhe

- `printf("%s", stringa);`  
La specifica `%s` visualizza *stringa* (che può essere variabile o costante e contenere spazi, Tab, `'\n'`, ecc.) fino al carattere `'\0'`
- Per visualizzare i primi *m* caratteri (a meno che non ce ne siano di meno) si indica il campo *precisione* nella specifica di conversione (es. `"%.3s"`)
- Per visualizzare la stringa in un campo di almeno *n* caratteri si indica la *dimensione* nella specifica di conversione (es. `"%10s"`), un `'-'` la allinea a sinistra: (es. `"%-10s"`)
- Restituisce il numero di caratteri visualizzati

# Output di stringhe

- `puts (stringa) ;`  
visualizza *stringa* (che può essere variabile o costante e contenere spazi, Tab, '`\n`', ecc.) fino al carattere '`\0`'
- Aggiunge automaticamente un '`\n`' (ossia va a capo dopo la stampa di *stringa*)

# Input di stringhe

- `gets (varStringa) ;`  
legge tutti i caratteri che vengono immessi dalla tastiera (spazi e Tab inclusi) fino al ritorno a capo e li mette in *varStringa* aggiungendo `' \0 '` al fondo
- Legge ed scarta il ritorno a capo immesso con il tasto Invio per terminare l'input
- Dà `NULL` in caso di errore o di fine input

# Input di stringhe

- `gets` non controlla se *varStringa* è abbastanza ampia da contenere i caratteri introdotti, se è troppo corta sovrascrive i byte di memoria subito successivi a *varStringa* modificando inavvertitamente le variabili fisicamente allocate lì (*buffer overflow*)
- Per specificare il numero massimo di caratteri da leggere si usi la funzione:  
`fgets(varStringa, n, stdin);`  
che legge fino a  $n-1$  caratteri
- `fgets`, diversamente da `gets`, mette in *varStringa* anche il `'\n'` corrispondente dell'INVIO immesso per terminare l'input

# Input di stringhe

- `scanf ("%s", varStringa);` → **SENZA &**  
La specifica "`%s`" salta i *white spaces* iniziali, legge tutti i caratteri che incontra fino al prossimo *white space*, mette la stringa letta in *varStringa* e aggiunge il '`\0`' al fondo
- Il carattere *white space* che fa fermare la `scanf` viene lasciato nel buffer e reso quindi disponibile per la successiva funzione di lettura (questo può creare problemi se non gestito correttamente)



# Input di stringhe

- Per la `scanf` i *white spaces* sono caratteri ordinari, tutti equivalenti tra loro
- Un *qualsiasi* white space nella *stringa di formato* indica alla `scanf` che in quel punto dell'input deve leggere e scartare tutti gli eventuali consecutivi white spaces ("eventuali" perché possono anche non essercene)
- In particolare (errore tipico), il carattere `'\n'` nella *stringa di formato* non indica di scartare tutti i restanti caratteri in input e passare alla lettura della riga successiva: potrebbe semplicemente far saltare uno spazio

# Input di stringhe

- Tutte le specifiche di conversione tranne la `%c` eliminano i *white spaces* che precedono la sequenza di caratteri da leggere, quindi *di norma* nella stringa di formato di una `scanf` non serve includere spazi (white spaces in generale) ed è preferibile che non ci siano per evitare errori di lettura
- È però necessario che una stringa di formato come "`%s %c`" abbia lo spazio indicato per eliminare tutti gli spazi che eventualmente precedono il carattere da leggere, ad es. in:  
"abcde x"

# Input di stringhe

- La `scanf` con la semplice specifica `%s` non controlla se *varStringa* è sufficientemente ampia per contenere i caratteri introdotti
- Se la stringa in input è più grande si ha un *buffer overflow* e i caratteri letti che eccedono *varStringa* vanno a occupare lo spazio di memoria successivo ad essa, modificando eventualmente altre variabili in modo errato
- Per specificare il numero *massimo* di caratteri da leggere con `%s`, si indica la *dimensione*, ad es. "`%10s`" salta i white spaces iniziali e legge fino a 10 e caratteri o al primo white space

# Input di stringhe

- Si può leggere una stringa con la `scanf` anche utilizzando la specifica di conversione `"%c"` indicando la *dimensione* (es. `"%4c"`)
- In questo caso viene letta *una stringa* di esattamente quel numero di caratteri, inclusi anche eventuali white spaces iniziali, intermedi e finali, *ma non viene aggiunto lo '\0' finale*  
`scanf ("%4c", varStringa) ;`
- Per saltare gli spazi iniziali (non quelli intermedi e finali) si può usare lo spazio iniziale, ossia la stringa di formato `"_ %4c"`

# Input di stringhe con scanset

- Uno *scanset* come dice il nome è un insieme (set) di caratteri da scandire (scan)
- Gli scanset si usano come specifiche di conversione per stringhe per specificare quali caratteri considerare nella lettura
- Uno scanset viene denotato indicando i caratteri che fanno parte del set tra parentesi quadre

# Input di stringhe con scanset

- Una specifica di conversione della forma `%[abc]` è quindi simile a `%s` salvo che la lettura si ferma solo quando trova un carattere *diverso* da quelli dell'insieme ('a', 'b' e 'c')
- Uno scanset non salta i white-spaces iniziali e aggiunge il **null** al fondo (come `%s`)
- Se si dà in input "abbaino" alla funzione `scanf ("%[abcno]", s);` si ferma alla 'i' che non c'è nello scanset, quindi `s` conterrà "abba" mentre "ino" resterà disponibile per le successive funzioni di input
- Uno scanset può includere lo spazio e il ritorno a capo (quindi può leggere più righe)

# Input di stringhe con scanset

- Il numero massimo di caratteri da leggere in uno scanset può essere specificato (in modo simile alla specifica `%3s`):
  - `%3[abc]` legge stringhe di al massimo 3 caratteri
- Per specificare un insieme consecutivo di caratteri si può usare la forma con il trattino:
  - `[a-z]` rappresenta tutte le lettere minuscole
  - `[A-Za-z]` rappresenta tutte le lettere
  - `[0-9]` rappresenta le cifre decimali
  - `[0-7]` rappresenta tutte le cifre ottali
  - Per includere il trattino, questo deve essere il primo carattere: `[-+*/]` rappresenta le 4 operazioni

# Input di stringhe con scanset

- Se il **primo** carattere dello scanset è '^', viene considerato lo scanset *complementare*, ossia quello composto da tutti i caratteri possibili *tranne* quelli indicati
- La lettura dunque si ferma solo quando trova un carattere *uguale* a uno di quelli indicati tra le parentesi quadre:

Se si dà in input "alber**o**" alla funzione:

```
scanf ("%[^nmrst]", s);
```

si ferma alla 'r' che c'è nello scanset, quindi *s* conterrà "albe" mentre "ro" resterà disponibile per le successive funzioni di input



# Input di stringhe con scanset

- Per leggere una stringa fino a fine riga utilizzando una `scanf` si può utilizzare `%[^\\n]*c` che legge tutti i caratteri che trova finché non incontra il ritorno a capo '\\n', la parte "`%*c`" legge e scarta il carattere di ritorno a capo
- In un ciclo di lettura di righe, si può utilizzare " `%[ ^ \\n ] "` per saltare i white spaces iniziali (incluse righe vuote)

# Buffer overflow

- Come già visto, le funzioni `gets` e `scanf` possono sovrascrivere “involontariamente” (viene usato per attacchi informatici) con i dati in input la memoria che segue la stringa
- Oltre alle preferibili soluzioni preventive (`fgets` e `sscanf` con l’indicazione della dimensione del campo), a scapito di un po’ di efficienza molti compilatori offrono la possibilità di aggiungere automaticamente dei controlli che verifichino, a ogni utilizzo della stringa, se l’operazione in corso farebbe sfiorare la stringa stessa (*boundary check*)

# Ritorno a capo

- A seconda del sistema operativo utilizzato, il ritorno a capo è in realtà costituito da una sequenza di uno o più caratteri:
  - MS-DOS/Win: CR+LF (codici ASCII 13 e 10: `\r\n`)
  - Unix/Linux/OSX: LF (codice 10: `\n`)
- Le funzioni di I/O viste considerano in carattere `'\n'` come generico "ritorno a capo":
  - **in input** la sequenza propria del sistema operativo per il ritorno a capo viene trasformata in `'\n'` (cioè legge e converte automaticamente 1 o 2 caratteri)
  - **in output** il carattere `'\n'` viene sostituito con la sequenza propria del sistema operativo (1 o 2 car.)

# Problemi di input

- Si abbiano le seguenti istruzioni in sequenza:  

```
scanf ("%d", &x) ;  
gets (s) ;
```

Se si inserisce un valore e si preme Invio, la `gets` sembra non essere eseguita ed `s` è vuota
- In realtà la `scanf` legge il valore, ma non il ritorno a capo `'\n'` che resta da leggere; la `gets` lo legge e termina subito mettendo in `s` una stringa vuota (cioè un carattere `'\0'`)
- Bisogna quindi "consumare" il ritorno a capo inserito dal tasto Invio

# Problemi di input

## ■ Alcune soluzioni:

1. si aggiunge dopo la specifica "%d" la specifica "%\*c" che legge e scarta il successivo carattere in input (il ritorno a capo): `scanf("%d%*c", &x);`
2. si aggiunge dopo la specifica %d uno spazio: "%d\_ "
3. si evita di mescolare `scanf` e `gets`:
  - si usano solo `scanf` con specifiche che scartano i white spaces iniziali (e quindi anche i '\n')
  - si usano solo `gets`: la `gets` legge una riga e scarta il '\n' alla fine della riga stessa
4. alcuni compilatori permettono di svuotare i buffer di input con `fflush(stdin)` tra la `scanf` e la `gets`, ma `fflush` su `stdin` non è standard e quindi non è consigliabile

# Problemi di input

- Si debbano introdurre due *caratteri* dalla tastiera (due input diversi) mediante il codice:  

```
puts("Inserisci 1o carattere: ");  
scanf("%c", &c);  
puts("Inserisci 2o carattere: ");  
scanf("%c", &d);
```

Non funziona correttamente perché la specifica `%c` non salta i white space, quindi la prima `scanf` preleva il 1° carattere e lo mette in `c`, ma lascia nel buffer della tastiera il *ritorno a capo* (Invio) e questo viene prelevato dalla seconda `scanf` e messo in `d`.

# Problemi di input

## ■ Alcune soluzioni:

1. nella prima `scanf` si usa `"%c%c"`
2. nella prima `scanf` si usa `"%c_ "`
3. nella seconda `scanf` si usa `"_ %c"` (spazio iniziale)
4. si sostituisce la 2<sup>a</sup> `scanf` con le due righe:

```
scanf ("%1s", s) ;
```

```
d = s[0] ;
```

```
avendo definito: char s[2] ;
```

Diversamente dalla specifica `%c`, la specifica `%s` salta i *white spaces* iniziali (compresi eventuali `'\n'`), `%1s` indica che si deve prelevare **un solo** carattere.

Attenzione che `%1s` produce *una stringa*: cioè memorizza il carattere letto e aggiunge il `'\0'`, bisogna dimensionarla di almeno 2 caratteri

# Stringa di formato

- Si noti che la stringa di formato è di solito un letterale stringa, ma può anche essere una variabile stringa:

```
char fmt_d = "%d";
```

```
char fmt_f = "%f";
```

```
...
```

```
scanf(fmt_d, &x);
```

```
printf(fmt_f, x/2.0);
```

Questo è utile per cambiare al run-time la modalità di visualizzazione di una variabile



# Test sui caratteri

- L'header `<ctype.h>` fornisce funzioni per la classificazione dei caratteri, queste danno risultato `!=0` (NON necessariamente 1) se `c` è del tipo indicato, le più comuni sono:
    - `isdigit(c)` → *cifra decimale*
    - `isalpha(c)` → *lettera*
    - `isalnum(c)` → *carattere alfanumerico*
    - `isxdigit(c)` → *cifra esadecimale*
    - `islower(c)` → *lettera minuscola*
    - `isupper(c)` → *lettera maiuscola*
    - `isspace(c)` → *white space*
- ```
if (isdigit(c)) printf("cifra");
```

# Test sui caratteri

- Nelle funzioni indicate il tipo di  $c$  è indicato come `int`, pertanto valori non `int` vengono convertiti in `int`
- Le funzioni richiedono che  $c$  sia un valore positivo (o `EOF`), se si passa un valore `char`:
  - se contiene un codice ASCII standard funzionano sempre correttamente
  - se i `char` sono `unsigned` funzionano sempre correttamente
  - se i `char` sono `signed` e  $c$  appartiene al *codice ASCII esteso* possono esserci problemi, è bene applicare a  $c$  un `cast` a `unsigned char`

```
if (isdigit((unsigned char)c))  
    printf("cifra");
```

# Case mapping

- L'header `<ctype.h>` dichiara anche funzioni che convertono il carattere `c`:
  - `toupper(c)` → *in maiuscolo*
  - `tolower(c)` → *in minuscolo*

Restituiscono un valore `int` contenente il codice ASCII del carattere `c` convertito (non modificano `c`)

Se a `toupper` si passa un carattere che non è una lettera minuscola, la funzione lo restituisce senza alcuna modifica (qualsiasi carattere sia), lo stesso dicasi per `tolower`

# Case mapping

- Poiché le due funzioni restituiscono un valore ma non lo cambiano, per convertire tutta una stringa è necessario applicare la funzione a ciascuno dei caratteri:

```
char s[10] = "Ciao! Come va?";  
for (i=0; s[i]!='\0'; i++)  
    s[i] = (char) toupper(s[i]);
```

memorizza in s:

```
"CIAO! COME VA?"
```

Si noti il cast a `char` di `toupper`, dovendo assegnare il valore prodotto di tipo `int` a un carattere è necessario per evitare il Warning

# Conversione di stringhe

- Le seguenti funzioni di `<stdlib.h>` convertono una stringa in numero. La stringa può anche avere altri char al fondo. La stringa deve iniziare con una cifra altrimenti dà 0 (!!!).
- $varInt = atoi(stringa)$   
converte *stringa* in un `int`  
`x=atoi("123"); → 123`
- $varLong = atol(stringa)$   
converte *stringa* in un `long`  
`y=atol("123xyz"); → 123L`
- $varLongLong = atoll(stringa) → 123LL$
- $varDouble = atof(stringa)$   
converte *stringa* in un `double`

# Libreria funzioni sulle stringhe

- L'header `<string.h>` contiene la definizione di funzioni per la gestione delle stringhe: lunghezza, copia, concatenazione, confronto, estrazione di parti (detto *parsing*)
- I parametri che non saranno modificati dalle funzioni sono dichiarati `const` (descritto in altre slide)
- Invece di scriversi le proprie funzioni di copia, ecc. conviene di solito usare quelle di libreria perché sono ottimizzate da professionisti e in alcuni casi realizzate in Assembly per avere la massima efficienza

# Lunghezza di una stringa

- La funzione `strlen(str)` restituisce un valore intero senza segno pari alla lunghezza di `str` (`'\0'` escluso)

```
int len;
```

```
char s[30]="ciao";
```

```
len=strlen(s); → len vale 4
```

- Più precisamente, `strlen()` restituisce un valore di tipo `size_t`, ma si può normalmente usare un intero di appropriata capacità (eventualmente con `cast`):

```
len=(int)strlen(s);
```

# Il tipo `size_t`

- `size_t` è un tipo intero senza segno definito dal compilatore (in `stddef.h` e `stdlib.h`) adatto a contenere la lunghezza di una qualsiasi stringa/vettore (o altro blocco di byte)
- In genere corrisponde a `unsigned long`
- Per stampare con una `printf` un valore di tipo `size_t` si può usare `"%lu"` e un cast a `unsigned long` sul valore



# Copia di una stringa

- La funzione `strcpy(str1, str2)` copia `str2` in `str1` e aggiunge lo `'\0'` finale
- È il modo corretto di assegnare un valore a una variabile stringa (può dare buffer overflow):

```
char s[30];
```

```
strcpy(s, "hello");
```

```
strcpy(s, "ciao");           → "ciao"
```

- Poiché `s` è una variabile stringa:

- non è corretto scrivere:

```
s = "ciao";
```

- **solo** nell'inizializzazione si può scrivere:

```
char s[30] = "ciao";
```

# Copia parziale di una stringa

- La funzione `strncpy(str1, str2, n)` copia i primi *n* caratteri di *str2* in *str1* (o meno se *str2* è più corta); può dare buffer overflow
- Attenzione: non aggiunge il '`\0`' finale se questo non è tra gli *n* caratteri di *str2* copiati:

```
char s[N]="hello";
```

```
strncpy(s, "ciao", 3); → "ciaoo"
```

```
strncpy(s, "hi", 2); → "hio"
```

- Per prevenire il problema si può usare:

```
strncpy(str1, str2, sizeof(str1) - 1);
```

```
str1[sizeof(str1) - 1] = '\0';
```

dove `sizeof(str1)` dà la dimensione di *str1*

# Concatenazione di stringhe

- La funzione `strcat(str1, str2)` concatena *str2* alla fine di *str1* (' \0 ' incluso)  
`char s[30]="Dove", t[30]="_vai?";  
strcat(s, t);` → **s vale "Dove vai?"**
- La funzione `strncat(str1, str2, n)` concatena i primi *n* caratteri di *str2* alla fine di *str1* (o meno se *str2* è più corta), non aggiunge il ' \0 ' finale se non è tra i primi *n* caratteri di *str2*
- Possono dare buffer overflow

# Confronto tra stringhe

- Avviene confrontando i caratteri di posizione corrispondente delle due stringhe secondo i loro codici ASCII ( " $<$ " significa "*precede*")

"cane" < "gatto" → "c" precede "g"

"cane" > "Gatto" → "G" precede "c"

"cane" < "cavallo" → "n" precede "v"

"cavallo" < "cavallone" → "\0" precede "n"

"cavallo" < "cavallo\_" → "\0" precede "\_"

"\_cavallo" < "cavallo" → "\_" precede "c"

"123" < "23" → "1" precede "2"

# Confronto tra stringhe

- La funzione `strcmp(str1, str2)` confronta *str1* e *str2* in base ai codici ASCII, restituisce un intero:
  - minore di 0 se  $str1 < str2$
  - uguale a 0 se  $str1 = str2$
  - maggiore di 0 se  $str1 > str2$
- La funzione `strncmp(str1, str2, n)` confronta i primi *n* caratteri di *str1* e *str2* (quelli presenti se almeno una è più corta)

# Ricerca in stringhe

- La funzione `strchr(str, carattere)` cerca *carattere* in *str* a partire dal suo primo carattere all'ultimo, dà `NULL` se non lo trova

```
if (strchr(s, 'z') != NULL)
    printf("Trovata una z!");
```
- `strrchr(str, carattere)` cerca *carattere* in *str* a partire dal suo ultimo carattere al primo, dà `NULL` se non lo trova
- `strstr(str1, str2)` cerca *str2* in *str1*, dà `NULL` se non la trova

```
if (strstr(s, "ciao") != NULL)
    printf("Trovata stringa ciao!");
```

# Ricerca in stringhe

- Per cercare a partire dalla posizione  $n$  della stringa si usi la seguente sintassi (diverrà chiara dopo aver visto i puntatori):

`strchr(str+n, carattere)`

`strstr(str1+n, str2)`

- Le funzioni di ricerca (`strchr`, `strrchr`, `strstr`, e altre presenti nella libreria standard come `strtok`, ecc.) in realtà restituiscono l'indirizzo di memoria ("il puntatore") a quanto trovato e `NULL` se non lo trovano

# Input da stringa

- La funzione `sscanf` (*stringa*, *formato*, *variabili*) ;  
è identica alla `scanf`, ma preleva i caratteri da *stringa* invece che dalla tastiera)
- Esempio  
se `str` contiene "12 ciao 23.2", la funzione:  
`sscanf(str, "%d%s%f", &a, s, &b);`  
legge e assegna: 12 ad `a`, "ciao" a `s`, 23.2 a `b`
- È utile per analizzare (*parsing*) una riga di cui non è noto a priori il numero di elementi che la compongono (per sapere quanti sono i valori letti si valuta il valore restituito dalla `sscanf`)



# Input da stringa

## ■ Esempio

```
gets (s) ;
```

```
n=sscanf (s, "%d%d", &a, &b) ;
```

- Se `s` contiene `"abcd"` (cioè non contiene valori numerici) nessuna delle variabili viene assegnata, tutto viene scartato, la `sscanf` restituisce 0
- Se `s` contiene `"12_abcd"`, 12 viene assegnato ad `a`, mentre `b` resta invariato e il resto viene scartato, la `sscanf` restituisce 1
- Se `s` contiene `"12_23_34_abcd"`, 12 viene assegnato ad `a`, 23 a `b`, il resto viene scartato, la `sscanf` restituisce 2
- Se `s` contiene `" \n"` (solo whitespaces) la `sscanf` restituisce EOF

# Output su stringa

- `sprintf(stringa, formato, variabili);`  
identica alla `printf`, ma "scrive" in *stringa* i caratteri che la `printf` manderebbe su `stdout` (video), aggiunge il **null** finale
- Esempio  
se `g` contiene 23 e `m` contiene "febbraio"  
`sprintf(str, "Il %d %s", g, m);`  
mette in `str`: "Il 23 febbraio\0"
- Utile per assemblare una stringa composta da parti di tipo diverso provenienti da variabili

# Altre funzioni di parsing

- Oltre alla `sscanf`, per la suddivisione (*parsing*) di una stringa composta da più parti (dette *token*) si possono usare le seguenti funzioni di `<string.h>` (la `strtok` è descritta nelle slide sui puntatori, per le altre si rimanda alla bibliografia): `strspn`, `strcspn`, `strpbrk`, `strtok`
- Per la conversione di stringhe in numeri oltre alle già descritte `atoi`, `atol` e `atof` esistono in `<stdlib.h>` le seguenti (preferibili perché informano in caso di errore): `strtod`, `strtol`, `strtoul`



# Vettori di stringhe

---

- Con questa definizione le stringhe sono tutte della stessa dimensione
- Con i puntatori si possono invece realizzare vettori di stringhe di lunghezze diverse (*jagged array*)

# Vettori di stringhe

- Se un inizializzatore è più lungo della dimensione della var. stringa, viene assegnata solo la parte iniziale di lunghezza pari a quella della var. (non deborda alla riga successiva):  
`char str[4][3] = { "unoduetre" };`  
inizializza la riga 0 con "uno" senza '\0'
- Al contrario le funzioni di input e di copia stringhe possono dare buffer overflow e i caratteri in più vanno a riempire le righe successive (con il '\0'):  
es. con `scanf("%s", str[0])` la lettura di "unodue" mette "uno" in `str[0]`, "due" in `str[1]` e "\0" in `str[2]`

# Vettori di stringhe

- Per memorizzare una sequenza di N stringhe in un vettore di stringhe si può utilizzare un ciclo:

```
char str[N][LEN];  
for (i=0; i<N; i++)  
    gets(str[i]);
```

- Si possono definire matrici multidimensionali anche di stringhe, ad es. una matrice di 4x6 stringhe di lunghezza N viene definita da:

```
char str[4][6][N];  
e ogni stringa è identificata da str[i][j]
```

# Esercizi

1. Scrivere un programma che date due stringhe in input stampi la **più lunga**. La prima se sono di uguale lunghezza.
2. Scrivere un programma che date due stringhe in input stampi la **maggiore**.
3. Scrivere un programma che chieda in input una stringa e calcoli da quanti caratteri è composta (senza usare la funzione `strlen` ma cercando il carattere `'\0'`)
4. Scrivere un programma che data una stringa in input, la converta tutta in maiuscolo.
5. Scrivere un programma che data una stringa in input verifichi se essa contiene almeno una 'A' tra i primi 10 caratteri.



# Esercizi

6. Scrivere un programma che richieda in input una stringa e conti quante cifre essa contiene.

Esempio

“Ciao2004! C6?” deve dare 5.

7. Scrivere un programma che richieda in input una stringa e conti di quante lettere maiuscole, lettere minuscole, cifre e altri caratteri è composta

Esempio

“Ciao2004! C6?” deve dare:

*maiuscole:2, minuscole: 3, cifre: 5, altri: 3.*

# Esercizi

8. Scrivere un programma che date in input due stringhe di lunghezza diversa indichi se la più corta è contenuta solo una volta nella più lunga.
9. Scrivere un programma che verifichi se la stringa data in input è palindroma o no ("kayak", "otto", "elle", "anilina").
10. Scrivere un programma che verifichi se la stringa data è composta di due parti uguali, trascurando il carattere centrale se la lunghezza è dispari (es. "CiaoCiao", "CiaoXCiao").

# Esercizi

11. Scrivere un programma che chieda all'utente di inserire una frase (max 128 caratteri), conti quante sono le *parole* (sequenze di lettere dell'alfabeto) che la compongono e la lunghezza media delle parole stesse.

Esempio

Se viene dato in input:

*Ieri... sono andato a mangiare all'una!*

il programma deve indicare che ci sono 7 parole e che la lunghezza media è 4.14 caratteri.