Programmazione 1

Lezione 2

Vincenzo Marra

vincenzo.marra@unimi.it

Dipartimento di Matematica *Federigo Enriques*
Università degli Studi di Milano

13 marzo 2019

La funzione `printf` della libreria standard

- La funzione `printf` serve a stampare stringhe di caratteri sul terminale. Permette quindi di produrre dati in uscita, o output.

La funzione `printf` della libreria standard

- La funzione `printf` serve a stampare stringhe di caratteri sul terminale. Permette quindi di produrre dati in uscita, o output.
- Essa è definita nel file di intestazione (in inglese *header*, da cui il suffisso `.h`) denominato `stdio.h`, per `standard input and output`.

La funzione `printf` della libreria standard

- La funzione `printf` serve a stampare stringhe di caratteri sul terminale. Permette quindi di produrre dati in uscita, o output.
- Essa è definita nel file di intestazione (in inglese *header*, da cui il suffisso `.h`) denominato `stdio.h`, per `standard input and output`.
- Per usarla occorre quindi includere il file di intestazione nel programma, con la direttiva per il preprocessore

`#include` <*nomefile*>

La funzione `printf` della libreria standard

- La funzione `printf` serve a stampare stringhe di caratteri sul terminale. Permette quindi di produrre dati in uscita, o output.
- Essa è definita nel file di intestazione (in inglese *header*, da cui il suffisso `.h`) denominato `stdio.h`, per `standard input and output`.
- Per usarla occorre quindi includere il file di intestazione nel programma, con la direttiva per il preprocessore

  `#include` <*nomefile*>
- Più in generale, le righe del sorgente che cominciano col simbolo # denotano direttive per il preprocessore; ne parleremo in dettaglio più avanti nel corso.

- La funzione `printf` può avere come argomento una *costante di tipo stringa*, racchiusa fra doppi apici: ad esempio,

```
printf("pippo")
```

scriverà `pippo` sul terminale.

- La funzione `printf` può avere come argomento una *costante di tipo stringa*, racchiusa fra doppi apici: ad esempio,

  ```
  printf("pippo")
  ```

  scriverà `pippo` sul terminale.
- All'interno della stringa argomento si possono inserire dei caratteri di controllo che regolano la formattazione dei dati. Tali caratteri di controllo sono noti come *sequenze di escape* (in inglese, *escape sequences*, letteralmente "sequenze di fuga").

- La funzione `printf` può avere come argomento una *costante di tipo stringa*, racchiusa fra doppi apici: ad esempio,

  ```
  printf("pippo")
  ```

  scriverà `pippo` sul terminale.
- All'interno della stringa argomento si possono inserire dei caratteri di controllo che regolano la formattazione dei dati. Tali caratteri di controllo sono noti come *sequenze di escape* (in inglese, *escape sequences*, letteralmente "sequenze di fuga").
- Un esempio che abbiamo già visto è la sequenza di escape `\n` (per *new line*, ossia "nuova riga'), che indica un ritorno a capo.

- La funzione `printf` può avere come argomento una *costante di tipo stringa*, racchiusa fra doppi apici: ad esempio,

  ```
  printf("pippo")
  ```

  scriverà `pippo` sul terminale.
- All'interno della stringa argomento si possono inserire dei caratteri di controllo che regolano la formattazione dei dati. Tali caratteri di controllo sono noti come *sequenze di escape* (in inglese, *escape sequences*, letteralmente "sequenze di fuga").
- Un esempio che abbiamo già visto è la sequenza di escape `\n` (per *new line*, ossia "nuova riga'), che indica un ritorno a capo.
- Così,

  ```
  printf("pippo\n")
  ```

  scriverà `pippo` sul terminale, seguito da un a capo.

Altre utili sequenze di escape:

Escape Sequence	Character
\a	Bell (speaker beeps)
\b	Backspace (non-erase)
\f	Form feed/clear screen
\n	New line
\r	Carriage Return
\t	Tab
\v	Vertical tab
\\	Backslash
\?	Question mark
\'	Single quote
\"	Double quote
\x*nn*	Hexadecimal character code *nn*
\o*nn*	Octal character code *nn*

- La funzione `printf` può avere anche altri argomenti, e impareremo a usarla in modo completo solo col tempo. Per esempio, la riga

```
printf("pippo ha %d anni", 12)
```

scriverà `pippo ha 12 anni`.

- La funzione `printf` può avere anche altri argomenti, e impareremo a usarla in modo completo solo col tempo. Per esempio, la riga

```
printf("pippo ha %d anni", 12)
```

  scriverà `pippo ha 12 anni`.
- Questo modo di usare `printf` è essenziale in congiunzione con le *variabili*, di cui parleremo diffusamente più avanti. Ma vedremo subito qualche esempio in cui la forma sintattica qui sopra è utile.

Più

Piu.c

```c
/* Sommare due numeri interi */

#include <stdio.h>

int main(void)
{
        printf("So fare le somme: %d+%d=%d\n",17,25,17+25);

        return 0;
}
```

Meno

Meno.c

```c
/* Sottrare un intero da un altro */

#include <stdio.h>

int main(void)
{
        printf("So fare le sottrazioni: %d-%d=%d\n",17,25,17-25);

        return 0;
}
```

Opposto

Opp.c

```c
/* Calcolare l'opposto di un intero */

#include <stdio.h>

int main(void)
{
        printf("So calcolare gli opposti: -(%d) = %d\n",17,-(17));

        return 0;
}
```

Per

Per.c

```c
/* Moltiplicare due interi */

#include <stdio.h>

int main(void)
{
        printf("So fare le moltiplicazioni: %dx%d=%d\n",17,25,17*25);

        return 0;
}
```

Dati interi $a \in \mathbb{Z}$ e $b \in \mathbb{Z} \setminus \{0\}$, non si può sempre eseguire la divisione esatta di a per b, a meno che b, appunto, non divida a.

Dati interi $a \in \mathbb{Z}$ e $b \in \mathbb{Z} \setminus \{0\}$, non si può sempre eseguire la divisione esatta di a per b, a meno che b, appunto, non divida a. Possiamo però sempre definire la divisione intera:

$$\begin{aligned} \div : \mathbb{Z} \times \mathbb{Z} \setminus \{0\} &\longrightarrow \mathbb{Z} \\ (a, b) &\longmapsto \operatorname*{trunc}_{\to 0} \frac{a}{b}, \end{aligned} \tag{♭}$$

dove $\operatorname*{trunc}_{\to 0} \frac{a}{b}$ denota il troncamento verso lo zero del numero razionale $\frac{a}{b} \in \mathbb{Q}$, ossia:

$$\operatorname*{trunc}_{\to 0} \frac{a}{b} := \begin{cases} \left\lfloor \frac{a}{b} \right\rfloor & \text{se } \frac{a}{b} \geqslant 0 \\ \left\lceil \frac{a}{b} \right\rceil & \text{se } \frac{a}{b} < 0. \end{cases}$$

Dati interi $a \in \mathbb{Z}$ e $b \in \mathbb{Z} \setminus \{0\}$, non si può sempre eseguire la divisione esatta di a per b, a meno che b, appunto, non divida a. Possiamo però sempre definire la divisione intera:

$$\begin{aligned} \div : \mathbb{Z} \times \mathbb{Z} \setminus \{0\} &\longrightarrow \mathbb{Z} \\ (a, b) &\longmapsto \operatorname*{trunc}_{\to 0} \frac{a}{b}, \end{aligned} \tag{♭}$$

dove $\operatorname*{trunc}_{\to 0} \frac{a}{b}$ denota il troncamento verso lo zero del numero razionale $\frac{a}{b} \in \mathbb{Q}$, ossia:

$$\operatorname*{trunc}_{\to 0} \frac{a}{b} := \begin{cases} \left\lfloor \frac{a}{b} \right\rfloor & \text{se } \frac{a}{b} \geqslant 0 \\ \left\lceil \frac{a}{b} \right\rceil & \text{se } \frac{a}{b} < 0. \end{cases}$$

Attenzione.

Questa convenzione sul valore del quoziente è valida a partire solo dallo standard C99. Gli standard precedenti lasciavano maggiore libertà all'implementazione del compilatore.

Diviso

Div.c

```c
/* Dividere un intero per un altro */

#include <stdio.h>

int main(void)
{
        printf("So fare le divisioni intere: %d : %d = %d\n",17,25,17/25);
        printf("So fare le divisioni intere: %d : %d = %d\n",25,17,25/17);
        printf("So fare le divisioni intere: %d : %d = %d\n",-25,17,(-25)/17);
        printf("So fare le divisioni intere: %d : %d = %d\n",25,-17,25/(-17));
        printf("So fare le divisioni intere: %d : %d = %d\n",-25,-17,(-25)/(-17));

        return 0;
}
```

Diviso per Zero?

DivZero.c

```c
/* Dividere un intero per zero? */

#include <stdio.h>

int main(void)
{
        printf("Non so fare le divisioni per zero: %d  :  %d = %d\n",17,0,17/0);

        return 0;
}
```

Diviso tabulato

DivForm.c

```c
/* Dividere un intero per un altro */

#include <stdio.h>

int main(void)
{
        printf("So fare le divisioni intere:\t%d\t:\t%d\t=\t%d\n",17,25,17/25);
        printf("So fare le divisioni intere:\t%d\t:\t%d\t=\t%d\n",25,17,25/17);
        printf("So fare le divisioni intere:\t%d\t:\t%d\t=\t%d\n",-25,17,(-25)/17);
        printf("So fare le divisioni intere:\t%d\t:\t%d\t=\t%d\n",25,-17,25/(-17));
        printf("So fare le divisioni intere:\t%d\t:\t%d\t=\t%d\n",-25,-17,(-25)/(-17));

        return 0;
}
```

Tipi, costanti, variabili

- Le variabili sono il principale costrutto linguistico col quale si fa riferimento ai *dati* elaborati dal programma.

Tipi, costanti, variabili

- Le variabili sono il principale costrutto linguistico col quale si fa riferimento ai *dati* elaborati dal programma.
- Il nome "variabile" deriva dal fatto che il valore assunto da una variabile può cambiare durante l'esecuzione del programma: non arbitrariamente, però, ma solo all'interno di un insieme determinato dal "tipo" della variabile.

Tipi, costanti, variabili

- Le variabili sono il principale costrutto linguistico col quale si fa riferimento ai *dati* elaborati dal programma.
- Il nome "variabile" deriva dal fatto che il valore assunto da una variabile può cambiare durante l'esecuzione del programma: non arbitrariamente, però, ma solo all'interno di un insieme determinato dal "tipo" della variabile.
- Infatti, in C le variabili hanno un tipo che determina (1) i possibili valori che la variabile può assumere, e (2) la quantità di memoria necessaria a memorizzare un valore di quel tipo. Ad esempio, se una variabile è di tipo "intero", essa non potrà assumere il valore $\frac{3}{2}$. Anche le costanti che compaiono nel codice del programma, come le variabili, hanno sempre un tipo.

Tipi, costanti, variabili

- Le variabili sono il principale costrutto linguistico col quale si fa riferimento ai *dati* elaborati dal programma.
- Il nome "variabile" deriva dal fatto che il valore assunto da una variabile può cambiare durante l'esecuzione del programma: non arbitrariamente, però, ma solo all'interno di un insieme determinato dal "tipo" della variabile.
- Infatti, in C le variabili hanno un tipo che determina (1) i possibili valori che la variabile può assumere, e (2) la quantità di memoria necessaria a memorizzare un valore di quel tipo. Ad esempio, se una variabile è di tipo "intero", essa non potrà assumere il valore $\frac{3}{2}$. Anche le costanti che compaiono nel codice del programma, come le variabili, hanno sempre un tipo.
- Il linguaggio C mette a disposizione un insieme di tipi predefiniti, detti tipi primitivi. È possibile definire dei nuovi tipi (non primitivi), ma di questo ci occuperemo più avanti.

I 4 tipi primitivi fondamentali

Parola chiave	Nome	Descrizione	Dim. minima	Range minimo
`char`	Carattere	Codici ASCII	8 bit	$[-127, 127]$
`int`	Intero	Intero	16 bit	$[-32767, +32767]$
`float`	Reale	Precisione singola	32 bit	$[-10^{37}, 10^{37}]$
`double`	Reale	Precisione doppia	32 bit	$[-10^{37}, 10^{37}]$

I 4 tipi primitivi fondamentali

Parola chiave	Nome	Descrizione	Dim. minima	Range minimo
`char`	Carattere	Codici ASCII	8 bit	$[-127, 127]$
`int`	Intero	Intero	16 bit	$[-32767, +32767]$
`float`	Reale	Precisione singola	32 bit	$[-10^{37}, 10^{37}]$
`double`	Reale	Precisione doppia	32 bit	$[-10^{37}, 10^{37}]$

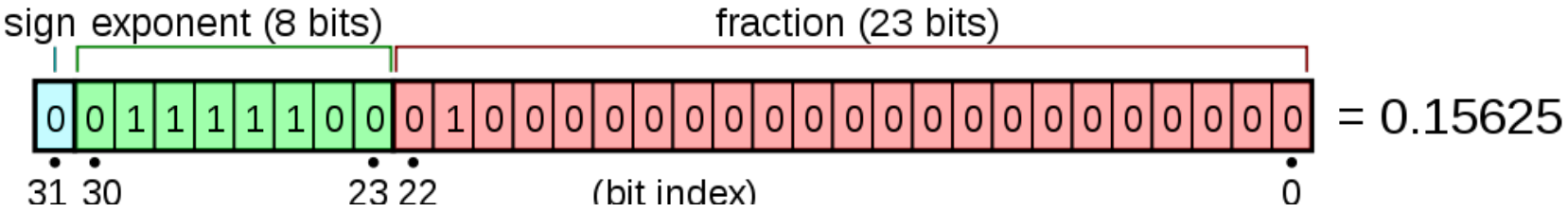
- I tipi `char` e `int` sono detti tipi integrali: i loro valori sono interi. In effetti, `char` è semplicemente un *sottotipo*, nel senso ovvio, di `int`. La differenza, oltre al range diverso, sta solo nel modo un cui i valori `char` sono da noi interpretati, come vedremo fra poco: ma dal punto di vista del compilatore si tratta pur sempre di interi. Vedremo, per esempio, che *in C si possono sommare due caratteri.*

I 4 tipi primitivi fondamentali

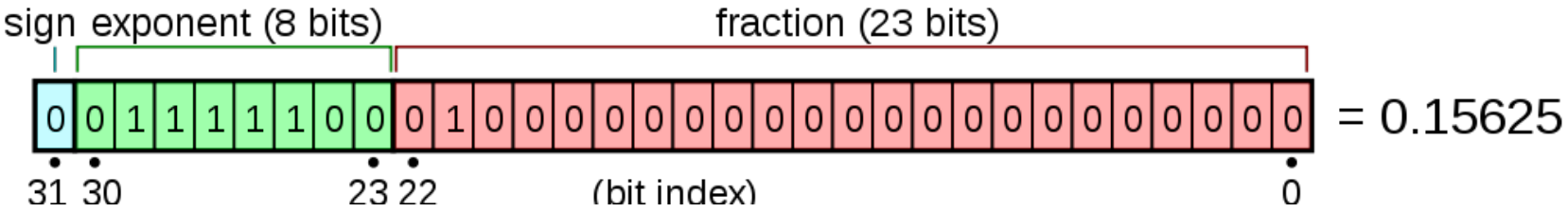
Parola chiave	Nome	Descrizione	Dim. minima	Range minimo
`char`	Carattere	Codici ASCII	8 bit	$[-127, 127]$
`int`	Intero	Intero	16 bit	$[-32767, +32767]$
`float`	Reale	Precisione singola	32 bit	$[-10^{37}, 10^{37}]$
`double`	Reale	Precisione doppia	32 bit	$[-10^{37}, 10^{37}]$

- I tipi `char` e `int` sono detti tipi integrali: i loro valori sono interi. In effetti, `char` è semplicemente un *sottotipo*, nel senso ovvio, di `int`. La differenza, oltre al range diverso, sta solo nel modo un cui i valori `char` sono da noi interpretati, come vedremo fra poco: ma dal punto di vista del compilatore si tratta pur sempre di interi. Vedremo, per esempio, che *in C si possono sommare due caratteri.*
- I tipi `float` e `double` sono detti tipi reali. La loro dimensione minima non è in realtà specificata dallo standard, perché essa dipende dal tipo di codifica in virgola mobile usata.

Virgola mobile (standard IEEE 754, “binary32”)

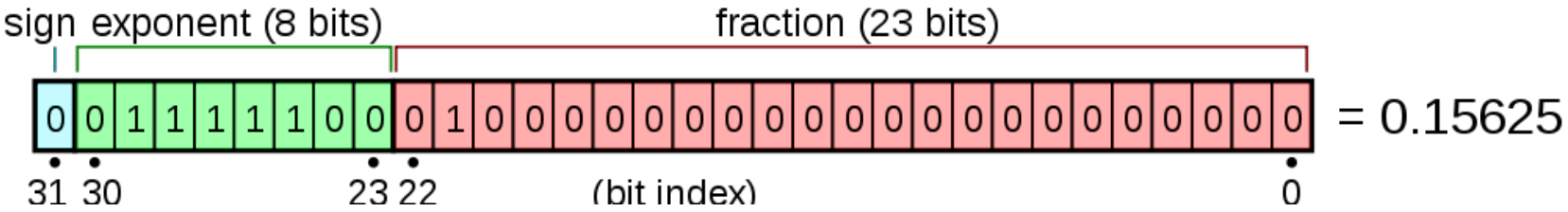


Virgola mobile (standard IEEE 754, "binary32")



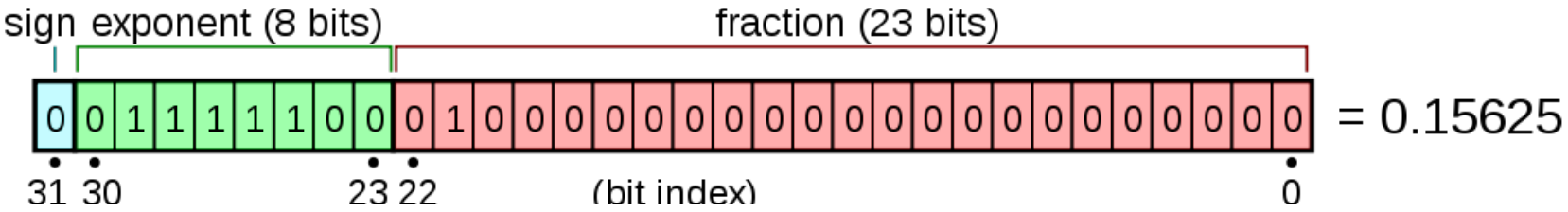
- *Sign bit*. "Bit di segno".

Virgola mobile (standard IEEE 754, "binary32")



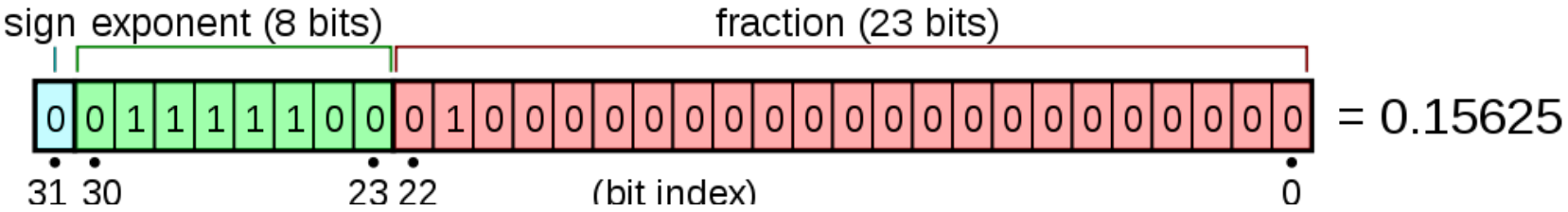
- *Sign bit*. "Bit di segno".
- *Fraction* or *Significand* or *Mantissa*. In italiano di solito "Mantissa".

Virgola mobile (standard IEEE 754, "binary32")



- *Sign bit*. "Bit di segno".
- *Fraction* or *Significand* or *Mantissa*. In italiano di solito "Mantissa".
- *Exponent*. "Esponente". (La base della rappresentazione è 2.)

Virgola mobile (standard IEEE 754, "binary32")



- *Sign bit*. "Bit di segno".
- *Fraction* or *Significand* or *Mantissa*. In italiano di solito "Mantissa".
- *Exponent*. "Esponente". (La base della rappresentazione è 2.)

Scrivendo b_i per il valore dell'i-esimo bit, ed e per il valore dell'esponente, si ha che il valore rappresentato è:

$$(-1)^{b_{31}} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right) \times 2^{e-127}$$

Costanti intere e in virgola mobile

- Le costanti intere si scrivono nel sorgente nel modo ovvio: 123, -123, 0, ecc.

Costanti intere e in virgola mobile

- Le costanti intere si scrivono nel sorgente nel modo ovvio: `123`, `-123`, `0`, ecc.
- Le costanti reali a precisione singola o doppia (`float` o `double`) si scrivono con il separatore decimale punto (=la virgola dei Paesi latini): `0.123`, `-0.123`, `12.345`, `0.0`, ecc.

Costanti intere e in virgola mobile

- Le costanti intere si scrivono nel sorgente nel modo ovvio: `123`, `-123`, `0`, ecc.
- Le costanti reali a precisione singola o doppia (`float` o `double`) si scrivono con il separatore decimale punto (=la virgola dei Paesi latini): `0.123`, `-0.123`, `12.345`, `0.0`, ecc.
- Se una costante reale sia `float` o `double` è implicitamente determinato dalla sua dimensione.

Costanti intere e in virgola mobile

- Le costanti intere si scrivono nel sorgente nel modo ovvio: `123`, `-123`, `0`, ecc.
- Le costanti reali a precisione singola o doppia (`float` o `double`) si scrivono con il separatore decimale punto (=la virgola dei Paesi latini): `0.123`, `-0.123`, `12.345`, `0.0`, ecc.
- Se una costante reale sia `float` o `double` è implicitamente determinato dalla sua dimensione.
- Si può anche usare la notazione esponenziale:

$$\texttt{2.4E13} = 2.4 \times 10^{13}$$

$$\texttt{-2.4E-13} = -2.4 \times 10^{-13}$$

(Al posto di `E` si può anche scrivere `e`).

Costanti carattere: Il codice ASCII

- I caratteri che si digitano tramite terminale hanno una rappresentazione *numerica* interna al calcolatore. Una delle codifiche storiche, risalente al 1963, è nota come codice ASCII, acronimo di *American Standard Code for Information Interchange*. Oggi è comune l'uso di codifiche più estese, come lo standard Unicode, ma il codice ASCII è compatibile con i nuovi standard.

Costanti carattere: Il codice ASCII

- I caratteri che si digitano tramite terminale hanno una rappresentazione *numerica* interna al calcolatore. Una delle codifiche storiche, risalente al 1963, è nota come codice ASCII, acronimo di *American Standard Code for Information Interchange*. Oggi è comune l'uso di codifiche più estese, come lo standard Unicode, ma il codice ASCII è compatibile con i nuovi standard.
- In un sorgente C, un carattere racchiuso fra apici singoli denota il corrispondente codice ASCII.

Costanti carattere: Il codice ASCII

- I caratteri che si digitano tramite terminale hanno una rappresentazione *numerica* interna al calcolatore. Una delle codifiche storiche, risalente al 1963, è nota come codice ASCII, acronimo di *American Standard Code for Information Interchange*. Oggi è comune l'uso di codifiche più estese, come lo standard Unicode, ma il codice ASCII è compatibile con i nuovi standard.
- In un sorgente C, un carattere racchiuso fra apici singoli denota il corrispondente codice ASCII.
- Per esempio,

  ```
  'A'
  ```

  è un modo esotico per scrivere 65.

Costanti carattere: Il codice ASCII

- I caratteri che si digitano tramite terminale hanno una rappresentazione *numerica* interna al calcolatore. Una delle codifiche storiche, risalente al 1963, è nota come codice ASCII, acronimo di *American Standard Code for Information Interchange*. Oggi è comune l'uso di codifiche più estese, come lo standard Unicode, ma il codice ASCII è compatibile con i nuovi standard.
- In un sorgente C, un carattere racchiuso fra apici singoli denota il corrispondente codice ASCII.
- Per esempio,

  `'A'`

  è un modo esotico per scrivere 65.
- Ci si riferisce a `'c'`, dove *c* è un carattere, come a una costante (di tipo) carattere.

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Un esempio: ASCII e operazioni sui caratteri

ascii.c

```c
/* Visualizzazione di codici ascii, e somma di caratteri */

#include <stdio.h>

int main(void)
{
        printf("Il codice ASCII di %c e' %d.\n", 'A', 'A');

        printf("La somma di %c e %c e' %c.\n",'A','B','A'+'B');

        return 0;
}
```

Tabella B.1 Conversioni di `printf`.

CARATTERE	TIPO DELL'ARGOMENTO; SCRITTO COME
`d, i`	`int`; notazione decimale con segno.
`o`	`unsigned int`; notazione ottale priva di segno (senza lo zero in testa).
`x, X`	`unsigned int`; notazione esadecimale priva di segno (senza `0x` o `0X` in testa), dove `abcdef` (per `0x`) o `ABCDEF` (per `0X`) stanno per 10, ..., 15.
`u`	`unsigned int`; notazione decimale priva di segno.
`c`	`int`; carattere singolo, dopo la conversione in `unsigned char`.
`s`	`char *`; stampa i caratteri della stringa fino a raggiungere `'\0'`, o a esaurire il grado di precisione specificato.
`f`	`double`; notazione decimale della forma `[-]`*iii.ddd*, dove *iii* è la parte intera, *ddd* sono le cifre dopo il punto decimale, in numero dato dal grado di precisione specificato (di default è pari a 6); omette il punto decimale se il grado di precisione è zero.
`e, E`	`double`; notazione decimale della forma `[-]`*i.dddddd*`e`$\pm$*xx* o `[-]`*i.dddddd*`E`$\pm$*xx*, dove *xx* è l'esponente, e il numero di *d* è dato dal grado di precisione specificato (di default è pari a 6); omette il punto decimale se il grado di precisione è zero.
`g, G`	`double`; usa `%e` o `%E` se l'esponente è minore di -4 o maggiore o uguale al grado di precisione; altrimenti usa `%f`. Omette gli zeri in coda, e il punto decimale in coda.
`p`	`void *`; puntatore (rappresentazione dipendente dall'implementazione).
`n`	`int *`; il numero di caratteri scritti finora da questa chiamata a `printf` *è salvato nell'argomento*. Non ha luogo alcuna conversione.
`%`	non ha luogo alcuna conversione; visualizza un `%`.

Dichiarazioni: primi cenni

- Prima di usare una variabile in un programma è necessario dichiararla. Torneremo sulle dichiarazioni nella prossima lezione.

Dichiarazioni: primi cenni

- Prima di usare una variabile in un programma è necessario dichiararla. Torneremo sulle dichiarazioni nella prossima lezione.
- Per ora, diciamo che una dichiarazione di variabile ne stabilisce il tipo e il nome, secondo la sintassi:

tipo nome;

Dichiarazioni: primi cenni

- Prima di usare una variabile in un programma è necessario dichiararla. Torneremo sulle dichiarazioni nella prossima lezione.
- Per ora, diciamo che una dichiarazione di variabile ne stabilisce il tipo e il nome, secondo la sintassi:

  tipo nome;

- La dichiarazione deve avvenire prima del primo utilizzo della variabile, pena un errore in compilazione.

Dichiarazioni: primi cenni

- Prima di usare una variabile in un programma è necessario dichiararla. Torneremo sulle dichiarazioni nella prossima lezione.
- Per ora, diciamo che una dichiarazione di variabile ne stabilisce il tipo e il nome, secondo la sintassi:

tipo nome;

- La dichiarazione deve avvenire prima del primo utilizzo della variabile, pena un errore in compilazione.
- Ad esempio, l'istruzione

```
int x;
```

dichiara una variabile di nome `x` e tipo `int`.

Assegnamenti: primi cenni

- Per assegnare un valore a una variabile, si usa il fondamentale costrutto dell'assegnamento. Anche di questo parleremo a fondo la prossima volta.

Assegnamenti: primi cenni

- Per assegnare un valore a una variabile, si usa il fondamentale costrutto dell'assegnamento. Anche di questo parleremo a fondo la prossima volta.
- Per ora, diciamo che se si è già dichiarata una variabile `x` di tipo `int`, allora, per esempio,

```
x=5;
```

assegna ad `x` il valore intero `5`.

Assegnamenti: primi cenni

- Per assegnare un valore a una variabile, si usa il fondamentale costrutto dell'assegnamento. Anche di questo parleremo a fondo la prossima volta.
- Per ora, diciamo che se si è già dichiarata una variabile `x` di tipo `int`, allora, per esempio,

```
x=5;
```

  assegna ad `x` il valore intero `5`.
- Si può anche assegnare un valore alla variabile al momento della sua dichiarazione, come nell'esempio seguente:

```
int x=5;
```

Var.c

```c
/* Uso delle variabili */

#include <stdio.h>

int main(void)
{
        //Dichiara x di tipo intero
        int x;
        //Dichiara y di tipo double e le assegna -2.5
        double y=-2.5;
        //Assegna ad x il valore 5
        x=5;

        printf("x: %d\ny: %g\n",x,y);

        return 0;
}
```

Overflow & Underflow

- Abbiamo visto che i numeri (interi e "reali", o meglio razionali) sono rappresentati tramite un numero prefissato di bit. Può quindi succedere che, partendo da un numero rappresentabile n, operazioni aritmetiche aventi n come argomento producano un risultato non più rappresentabile.

Overflow & Underflow

- Abbiamo visto che i numeri (interi e "reali", o meglio razionali) sono rappresentati tramite un numero prefissato di bit. Può quindi succedere che, partendo da un numero rappresentabile n, operazioni aritmetiche aventi n come argomento producano un risultato non più rappresentabile.
- Esempio semplificato. Si usino 3 bit per rappresentare interi positivi. Allora l'insieme rappresentabile è

$$\{1, \ldots, 7, 8\}.$$

Se a è una variabile di questo tipo, e a=8, l'operazione di incremento a=a+1 conduce a un intero non rappresentabile.

Overflow & Underflow

- Abbiamo visto che i numeri (interi e "reali", o meglio razionali) sono rappresentati tramite un numero prefissato di bit. Può quindi succedere che, partendo da un numero rappresentabile n, operazioni aritmetiche aventi n come argomento producano un risultato non più rappresentabile.
- Esempio semplificato. Si usino 3 bit per rappresentare interi positivi. Allora l'insieme rappresentabile è

$$\{1, \ldots, 7, 8\}.$$

  Se a è una variabile di questo tipo, e a=8, l'operazione di incremento a=a+1 conduce a un intero non rappresentabile.
- Si parla allora di overflow (aritmetico). In italiano, *traboccamento*.

- Nell'esempio, il valore della variabile a dopo l'incremento sarà, necessariamente, uno dei valori rappresentabili $\{1, \ldots, 8\}$. Esattamente *quale* valore sia dipende da diversi fattori, e principalmente dall'architettura della macchina su cui il programma è eseguito. Di solito, quindi, gli standard non prescrivono un valore determinato delle variabili in caso di overflow.

- Nell'esempio, il valore della variabile a dopo l'incremento sarà, necessariamente, uno dei valori rappresentabili $\{1, \ldots, 8\}$. Esattamente *quale* valore sia dipende da diversi fattori, e principalmente dall'architettura della macchina su cui il programma è eseguito. Di solito, quindi, gli standard non prescrivono un valore determinato delle variabili in caso di overflow.
- Il punto di fondo è che *nessun programma ben scritto dovrebbe andare in overflow*. Il codice sorgente deve essere scritto in modo tale da prevenire questa possibilità, o perlomeno da gestirla in modo appropriato. Alcuni dei peggiori bachi (=errori nascosti) nei programmi sono dovuti a una gestione carente del traboccamento.

- Nell'esempio, il valore della variabile a dopo l'incremento sarà, necessariamente, uno dei valori rappresentabili $\{1, \ldots, 8\}$. Esattamente *quale* valore sia dipende da diversi fattori, e principalmente dall'architettura della macchina su cui il programma è eseguito. Di solito, quindi, gli standard non prescrivono un valore determinato delle variabili in caso di overflow.
- Il punto di fondo è che *nessun programma ben scritto dovrebbe andare in overflow*. Il codice sorgente deve essere scritto in modo tale da prevenire questa possibilità, o perlomeno da gestirla in modo appropriato. Alcuni dei peggiori bachi (=errori nascosti) nei programmi sono dovuti a una gestione carente del traboccamento.
- Considerazioni analoghe valgono per l'underflow, cui adesso accenneremo.

- Si parla di underflow (aritmetico) quando una serie di operazioni su dati in virgola mobile produce un risultato non nullo, ma troppo piccolo per essere correttamente rappresentato.

- Si parla di underflow (aritmetico) quando una serie di operazioni su dati in virgola mobile produce un risultato non nullo, ma troppo piccolo per essere correttamente rappresentato.
- Si noti che l'underflow non ha senso per tipi a valori in $\mathbb{Z}$. Negli interi, infatti, esiste un più piccolo intorno non-singoletto dello zero – ossia, l'insieme $\{-1, 0, 1\}$ – i cui valori, se appartenenti al tipo in questione, sono certamente tutti rappresentabili.

- Si parla di underflow (aritmetico) quando una serie di operazioni su dati in virgola mobile produce un risultato non nullo, ma troppo piccolo per essere correttamente rappresentato.
- Si noti che l'underflow non ha senso per tipi a valori in $\mathbb{Z}$. Negli interi, infatti, esiste un più piccolo intorno non-singoletto dello zero – ossia, l'insieme $\{-1, 0, 1\}$ – i cui valori, se appartenenti al tipo in questione, sono certamente tutti rappresentabili.
- Il linguaggio C mette a disposizione due file di intestazione che contengono informazioni dettagliate sui numeri rappresentabili. Si tratta dei file `limits.h` e `float.h`. Fra le altre cose, i due file definiscono i valori costanti riportati nella tabella seguente.

Nome	Valore	Intestazione
`INT_MAX`	Max `int` rappresentabile	`limits.h`
`INT_MIN`	Min `int` rappresentabile	`limits.h`
`CHAR_MAX`	Max `char` rappresentabile	`limits.h`
`CHAR_MIN`	Min `char` rappresentabile	`limits.h`
`CHAR_BIT`	Dimensione in bit di `char`	`limits.h`
`FLT_MAX`	Max `float` rappresentabile	`float.h`
`FLT_MIN`	Min `float` positivo rappresentabile	`float.h`
`DBL_MAX`	Max `double` rappresentabile	`float.h`
`DBL_MIN`	Min `double` positivo rappresentabile	`float.h`

La libreria didattica Prog

(*Illustrazione al calcolatore dell'uso della libreria didattica* Prog.)