

# IL LINGUAGGIO C

---

- Un elaboratore è un **manipolatore di simboli (segni)**
- L'architettura fisica di ogni elaboratore è **intrinsecamente capace** di trattare vari domini di dati, detti **tipi primitivi**
  - dominio dei **numeri naturali e interi**
  - dominio dei **numeri reali**  
(con qualche approssimazione)
  - dominio dei **caratteri**
  - dominio delle **stringhe di caratteri**

# TIPI DI DATO

---

Il linguaggio C è un linguaggio tipizzato. Il concetto di *tipo di dato* viene introdotto per raggiungere due obiettivi:

- esprimere in modo sintetico
  - la loro rappresentazione in memoria, e
  - un insieme di operazioni ammissibili
- permettere di *effettuare controlli statici* (al momento della compilazione) sulla *correttezza* del programma.

# Tipi di dato: Specificatori e Modificatori (che vengono combinati)

- **Specificatori di tipo:**
  - `void`: indefinito o non specificato
  - `char`: caratteri
  - `int`: numeri interi
  - `float`: numeri reali (decimali) a precisione singola
  - `double`: numeri decimali a precisione doppia
- **Modificatori di tipo:**
  - `short`: riduce il numero di valori memorizzabili
  - `long`: aumenta il numero di valori memorizzabili
  - `signed`: dato con segno
  - `unsigned`: dato senza segno

# Memorizzazione delle Informazioni

- I dati vengono memorizzati nella memoria del calcolatore.
- Il calcolatore ha una memoria finita, sequenza finita di celle (si possono rappresentare un numero ristretto di valori possibili).
- Il bit (0-1) è l'unità basilare.
- Il byte (8 bit) è la più piccola quantità di memoria indirizzabile in molte architetture di calcolatori.
- Il numero di valori distinti che è possibile rappresentare in un tipo di dato dipende dal numero di bytes associati.

# TIPI DI DATO PRIMITIVI IN C

---

- **Caratteri (1 byte)**
  - signed char -128....127
  - unsigned char 0...255
- **interi con segno**
  - short (int) (2 bytes) -32768 ... 32767 (16 bit)
  - int ????????
  - long (int) -2147483648 .... 2147483647 (32 bit)
- **naturali (interi senza segno)**
  - unsigned short (int) 2 bytes 0 ... 65535
  - unsigned (int) ????????
  - unsigned long (int) 0 ... 4294967295 (32 bit)

Dimensione di `int` e `unsigned int` non fissa. **Dipende dal compilatore**



# Verifica della dimensione dei dati

- Le dimensioni esatte dei tipi di dato possono variare su computer differenti.
- Per verificare la dimensione dei dati:
  - Usare l'operatore `sizeof` che permette di verificare l'occupazione in bytes di un tipo o variabile
  - Utilizzare il file header `limits.h` (dati interi) e `float.h` (dati reali).

# •Nota sulla Rappresentazione dei Caratteri

---

Un tipo fondamentale di dato da rappresentare è costituito dai *singoli caratteri*

Idea base: associare *a ciascun carattere un numero intero (codice)* in modo convenzionale

 *Codice standard ASCII* (1968)

ASCII definisce univocamente i primi 128 caratteri (7 bit – vedi tabella nel lucido seguente)

I caratteri con codice superiore a 127 possono variare secondo la particolare codifica adottata (dipendenza da linguaggio naturale: ISO 8859-1 per alfabeto latino1, ...) (8- bit)

Visto che i caratteri hanno un codice intero, essi possono essere considerati un insieme ordinato (ad esempio: 'g' > 'O' perché 103 > 79)



## Tabella ASCII standard

Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char
00000000	0	Null	00100000	32	Spc	01000000	64	@	01100000	96	`
00000001	1	Start of heading	00100001	33	!	01000001	65	A	01100001	97	a
00000010	2	Start of text	00100010	34	"	01000010	66	B	01100010	98	b
00000011	3	End of text	00100011	35	#	01000011	67	C	01100011	99	c
00000100	4	End of transmit	00100100	36	\$	01000100	68	D	01100100	100	d
00000101	5	Enquiry	00100101	37	%	01000101	69	E	01100101	101	e
00000110	6	Acknowledge	00100110	38	&	01000110	70	F	01100110	102	f
00000111	7	Audible bell	00100111	39	'	01000111	71	G	01100111	103	g
00001000	8	Backspace	00101000	40	(	01001000	72	H	01101000	104	h
00001001	9	Horizontal tab	00101001	41	)	01001001	73	I	01101001	105	i
00001010	10	Line feed	00101010	42	*	01001010	74	J	01101010	106	j
00001011	11	Vertical tab	00101011	43	+	01001011	75	K	01101011	107	k
00001100	12	Form Feed	00101100	44	,	01001100	76	L	01101100	108	l
00001101	13	Carriage return	00101101	45	-	01001101	77	M	01101101	109	m
00001110	14	Shift out	00101110	46	.	01001110	78	N	01101110	110	n
00001111	15	Shift in	00101111	47	/	01001111	79	O	01101111	111	o
00010000	16	Data link escape	00110000	48	0	01010000	80	P	01110000	112	p
00010001	17	Device control 1	00110001	49	1	01010001	81	Q	01110001	113	q
00010010	18	Device control 2	00110010	50	2	01010010	82	R	01110010	114	r
00010011	19	Device control 3	00110011	51	3	01010011	83	S	01110011	115	s
00010100	20	Device control 4	00110100	52	4	01010100	84	T	01110100	116	t
00010101	21	Neg. acknowledge	00110101	53	5	01010101	85	U	01110101	117	u
00010110	22	Synchronous idle	00110110	54	6	01010110	86	V	01110110	118	v
00010111	23	End trans. block	00110111	55	7	01010111	87	W	01110111	119	w
00011000	24	Cancel	00111000	56	8	01011000	88	X	01111000	120	x
00011001	25	End of medium	00111001	57	9	01011001	89	Y	01111001	121	y
00011010	26	Substitution	00111010	58	:	01011010	90	Z	01111010	122	z
00011011	27	Escape	00111011	59	;	01011011	91	[	01111011	123	{
00011100	28	File separator	00111100	60	<	01011100	92	\	01111100	124	
00011101	29	Group separator	00111101	61	=	01011101	93	]	01111101	125	}
00011110	30	Record Separator	00111110	62	>	01011110	94	^	01111110	126	~
00011111	31	Unit separator	00111111	63	?	01011111	95	_	01111111	127	Del

# COSTANTI DI TIPI PRIMITIVI

---

- **interi** (in varie basi di rappresentazione, se preceduto da 0 in ottale, se 0x esadecimale)

<i>base</i>	<i>2 byte</i>	<i>4 byte</i>
decimale	12	70000, 12L
ottale	014	0210560
esadecimale	0xFF	0x11170

- **reali**

– in doppia precisione (default)

24.0      2.4E1      240.0E-1

– in singola precisione

24.0F      2.4E1F      240.0E-1F

# COSTANTI DI TIPI PRIMITIVI

---

## caratteri

- singolo carattere racchiuso fra apici

`'A'`    `'C'`    `'6'`

- caratteri speciali:

`'\n'`    `'\t'`    `'\''`    `'\\'`    `'\"'`

# STRINGHE

---

- Una *stringa* è una *sequenza di caratteri* delimitata da virgolette

"ciao"

"Hello\n"

- In C le stringhe sono semplici sequenze di caratteri di cui l'ultimo, *sempre presente in modo implicito*, è '\0'

"ciao" = {'c', 'i', 'a', 'o', '\0'}

# ESPRESSIONI

---

- Il C è un linguaggio basato su ***espressioni***
- Una ***espressione*** è una *notazione che denota un valore* mediante un processo di ***valutazione***
- Una espressione può essere *semplice* o *composta* (tramite aggregazione di altre espressioni)

# ESPRESSIONI SEMPLICI

---

## Quali espressioni elementari?

- **costanti**

- ‘A’ 23.4 -3 “ciao” ....

- **simboli di variabile**

- x pippo pigreco ....

- **simboli di funzione**

- $f(x)$

- `concat("alfa", "beta")`

- ...

# OPERATORI ED ESPRESSIONI COMPOSTE

---

- Ogni linguaggio introduce un **insieme di operatori**
- che permettono di **aggregare altre espressioni (operandi)**
- per formare **espressioni composte**
- con riferimento a diversi **domini / tipi di dato** (numeri, testi, ecc.)

## Esempi

`2 + f(x)`

`4 * 8 - 3 % 2 + arcsin(0.5)`

`strlen(strcat(buf, "alfa"))`

`a && (b || c)`

...

# CLASSIFICAZIONE DEGLI OPERATORI

---

## Due criteri di classificazione:

- in base al *tipo* degli operandi
- in base al *numero* degli operandi

in base al <i>tipo</i> degli operandi	in base al <i>numero</i> di operandi
<ul style="list-style-type: none"><li>• aritmetici</li><li>• relazionali</li><li>• logici</li><li>• condizionali</li><li>• ...</li></ul>	<ul style="list-style-type: none"><li>• unari</li><li>• binari</li><li>• ternari</li><li>• ...</li></ul>



# Ordine di valutazione degli operandi

- In C, come in altri linguaggi l'ordine di valutazione degli operandi è generalmente non specificato.
- Si vuole lasciare libertà ai compilatori per meglio ottimizzare il codice.
- Per questo motivo non possiamo assumere ad esempio nella seguente espressione:  $(i * j) + (w - 3)$  se  $i$  verrà valutato prima di  $j$  o  $(i * j)$  prima di  $(w - 3)$  o viceversa.
- Se non ci sono side-effects la cosa non ha importanza, ma può produrre a situazioni indefinite se sono coinvolti side-effects (assegnamento).

# OPERATORI ARITMETICI

---

<i>operazione</i>	<i>operatore</i>	<i>C</i>
inversione di segno	<i>unario</i>	-
somma	<i>binario</i>	+
differenza	<i>binario</i>	-
moltiplicazione	<i>binario</i>	*
divisione fra interi	<i>binario</i>	/
divisione fra reali	<i>binario</i>	/
modulo (fra interi)	<i>binario</i>	%

**NB:** la divisione  $a/b$  è fra interi se sia  $a$  sia  $b$  sono interi,  
è fra reali in tutti gli altri casi

# OPERATORI: OVERLOADING

---

In C (come in Pascal, Fortran e molti altri linguaggi) **operazioni primitive associate a tipi diversi possono essere denotate con lo stesso simbolo**. Ad esempio, le operazioni aritmetiche su reali o interi

In realtà **l'operazione è diversa e può produrre risultati diversi**

```
int X, Y;  
se X = 10 e Y = 4;  
X/Y vale 2
```

```
int X; float Y;  
se X = 10 e Y = 4.0;  
X/Y vale 2.5
```

```
float X, Y;  
se X = 10.0 e Y = 4.0;  
X/Y vale 2.5
```

# CONVERSIONI DI TIPO

---

In C è possibile combinare tra di loro operandi di tipo diverso:

- espressioni **omogenee**: tutti gli operandi sono dello stesso tipo
- espressioni **eterogenee**: gli operandi sono di tipi diversi

**Come è possibile gestire espressioni che coinvolgono tipi di dati differenti (e con diversa rappresentazione quindi)?**

La **conversione di tipo** permette di convertire una variabile da un tipo ad un altro tipo

# CONVERSIONI DI TIPO

---

In C abbiamo due differenti modalità di conversione di tipo:

- **Implicita (coercion)** effettuata dal compilatore in presenza di espressioni eterogenee
- **Esplicita (typecasting)** ottenuta dal programmatore utilizzando uno specifico operatore (cast)

## Regola adottata in C:

- sono eseguibili le espressioni eterogenee in cui tutti i tipi referenziati risultano ***compatibili*** (ovvero che risultano omogenei dopo l'applicazione della regola automatica di conversione implicita di tipo del C).

# COMPATIBILITÀ DI TIPO

---

- Consiste nella **possibilità di usare, entro certi limiti, oggetti di un tipo al posto di oggetti di un altro tipo**
- **Un tipo T1 è compatibile con un tipo T2 se il dominio D1 di T1 è contenuto nel dominio D2 di T2**
  - `int` è compatibile con `float` perché  $\mathbb{Z} \subset \mathbb{R}$
  - ma `float` *non è compatibile* con `int`

# COMPATIBILITÀ DI TIPO - NOTA

---

- $3 / 4.2$   
è una divisione *fra reali*, in cui il primo operando è convertito automaticamente da **int** a **double**
- $3 \% 4.2$   
è una operazione *non ammissibile*, perché 4.2 non può essere convertito in **int**

# CONVERSIONI DI TIPO

---

In pratica il tipo di dimensione minore viene convertito in quello di dimensione maggiore.

Data una espressione  $x \text{ op } y$  (*descrizione sintetica non completa, manca unsigned, evitare di utilizzarlo in espressioni miste, spesso comportamenti non definiti*)

- **1.** Ogni variabile di tipo **char** o **short** viene convertita nel tipo **int**;
- **2.** Se dopo l'esecuzione del passo 1 l'espressione è ancora eterogenea, rispetto alla seguente gerarchia

**int < long < float < double < long double**

si converte temporaneamente l'operando di tipo *inferiore* al tipo *superiore* (***promotion***)

- **3.** A questo punto l'espressione è **omogenea** e viene eseguita l'operazione specificata. Il risultato è di tipo uguale a quello prodotto dall'operatore effettivamente eseguito (in caso di overloading, quello più alto gerarchicamente)



# CONVERSIONI DI TIPO

---

```
int x;  
char y;  
double r;  
(x+y) / r
```



La valutazione dell'espressione procede da sinistra verso destra

- **Passo 1**

**(x+y)**

- *y* viene convertito nell'intero corrispondente
- viene applicata la somma tra interi
- **risultato intero** *tmp*

- **Passo 2**

- *tmp* / *r*      *tmp* viene convertito nel double corrispondente
- viene applicata la divisione tra reali
- **risultato reale**

# COMPATIBILITÀ DI TIPO: Assegnamento

---

In un *assegnamento*, *l'identificatore di variabile e l'espressione* devono essere dello *stesso tipo*

- Nel caso di tipi diversi, se possibile si effettua la conversione implicita, altrimenti l'assegnamento può generare perdita di informazione (troncamenti o overflow).

```
int x;  
char y;  
double r;  
x = y;      /* char -> int*/  
x = y+x;  
r = y;      /* char -> int -> double*/  
x = r; /* troncamento*/
```

# COMPATIBILITÀ IN ASSEGNAIMENTO

---

- In generale, negli assegnamenti sono **automatiche** le conversioni di tipo che non provocano perdita d'informazione
- Espressioni che **possono** provocare perdita di informazioni non sono però illegali

## Esempio

```
int i=5; float f=2.71F; double d=3.1415;  
f = f+i; /* int convertito in float */  
i = d/f; /* double convertito in int !*/  
f = d; /* arrotondamento o troncamento */
```

Possibile Warning: *conversion may lose significant digits*

# Conversione esplicita di tipo: CAST

---

In qualunque espressione è possibile **forzare una particolare conversione** utilizzando l'*operatore di cast*

( <tipo> ) <espressione>

## Esempi

```
int i=3,k=2; long double x=7.77; double  
y=7.1;
```

```
i = (int) sqrt(384);
```

```
i = (int) x % (int)y;
```

```
y = (double) i / k; // y = 1,5
```

**Nota:** L'operatore cast ha precedenza rispetto all'operatore / e viene forzato prima della divisione

# OPERATORI RELAZIONALI

---

Sono operatori binari per il confronto dei valori di due espressioni:

<i>relazione</i>	<i>C</i>
<b>uguaglianza</b>	<b>==</b>
<b>diversità</b>	<b>!=</b>
<b>maggiore di</b>	<b>&gt;</b>
<b>minore di</b>	<b>&lt;</b>
<b>maggiore o uguale a</b>	<b>&gt;=</b>
<b>minore o uguale a</b>	<b>&lt;=</b>

# OPERATORI RELAZIONALI

---

Attenzione:

**non esistendo il tipo *boolean*, in C le espressioni relazionali *denotano un valore intero***

- **0 denota *falso***  
**(condizione non verificata)**
- **1 denota *vero***  
**(condizione verificata)**
- **Attenzione:  $3 > 2 > 1$  è  $(3 > 2) > 1$  cioè  $1 > 1 = 0$  cioè falso.**

# OPERATORI LOGICI

---

<i>connettivo logico</i>	<i>operatore</i>	<i>C</i>
not (negazione)	<i>unario</i>	!
and	<i>binario</i>	&&
or	<i>binario</i>	

- Anche le espressioni logiche *denotano un valore intero*
- da interpretare come vero (1) o falso (0)

# OPERATORI LOGICI

---

- Anche qui sono possibili espressioni miste, utili in casi specifici

5 && 7    0 || 33    !5

- Valutazione in *corto-circuito*
  - A differenza degli altri operatori hanno un preciso ordine di valutazione da sinistra a destra
  - la valutazione dell'espressione cessa *appena si è in grado di determinare il risultato*
  - il secondo operando è valutato *solo se necessario*



# VALUTAZIONE IN CORTO CIRCUITO

---

– **22 || x**

già vera in partenza perché 22 è vero

– **0 && x**

già falsa in partenza perché 0 è falso

– **a && b && c**

se **a&&b** è falso, il secondo **&&** non viene neanche valutato

– **a || b || c**

se **a||b** è vero, il secondo **||** non viene neanche valutato

# ESPRESSIONI CONDIZIONALI

---

Una espressione condizionale è introdotta dall'operatore ternario

*condiz* ? *espr1* : *espr2*

L'espressione denota:

- o il valore denotato da *espr1*
  - o quello denotato da *espr2*
  - in base al valore della espressione *condiz*
- 
- **se *condiz* è vera**, l'espressione nel suo complesso denota il valore denotato da *espr1*
  - **se *condiz* è falsa**, l'espressione nel suo complesso denota il valore denotato da *espr2*

# ESPRESSIONI CONDIZIONALI: ESEMPI

---

–  $3 \text{ ? } 10 \text{ : } 20$

denota sempre 10 (3 è sempre vera)

–  $x \text{ ? } 10 \text{ : } 20$

denota 10 se  $x$  è vera (diversa da 0),  
oppure 20 se  $x$  è falsa (uguale a 0)

–  $(x > y) \text{ ? } x \text{ : } y$

denota il maggiore fra  $x$  e  $y$

# ESPRESSIONI CONCATENATE

---

Un' espressione concatenata è introdotta dall'**operatore di concatenazione** (la virgola)

*espr1, espr2, ..., esprN*

- tutte le espressioni vengono valutate da sinistra a destra e l'espressione globale esprime il valore denotato da *esprN*
- Supponiamo che
  - i valga 5
  - k valga 7
- Allora l'espressione: *i + 1, k - 4* denota il valore denotato da *k-4*, cioè 3

# ESPRESSIONI CONCATENATE

---

L'operatore binario di concatenazione (la virgola) ha la più bassa priorità

*espr1, espr2, ..., esprN*

- Supponiamo che
  - i valga 0
  - k valga 0
- L'espressione:  **$i = (k = 2, k + 2)$**   
denota il valore 4 con  $i = 4$  e  $k = 2$
- L'espressione:  **$i = k = 2, k + 2$**   
denota il valore 4 con  $i = 2$  e  $k = 2$

# OPERATORI INFISSI, PREFISSI E POSTFISSI

---

- Le espressioni composte sono **strutture** formate da **operatori** applicati a uno o più **operandi**
- **Ma... *dove* posizionare l'operatore rispetto ai suoi operandi?**

# OPERATORI INFISSI, PREFISSI E POSTFISSI

---

## Tre possibili scelte:

- **prima** → notazione *prefissa*  
Esempio: **+ 3 4**
- **dopo** → notazione *postfissa*  
Esempio: **3 4 +**
- **in mezzo** → notazione *infissa*  
Esempio: **3 + 4**



È quella a cui siamo abituati,  
perciò è adottata *anche in C*

# OPERATORI INFISSI, PREFISSI E POSTFISSI

---

- Le notazioni *prefissa* e *postfissa* non hanno problemi di *priorità e/o associatività* degli operatori
  - non c'è mai dubbio su *quale* operatore vada applicato a *quali* operandi
- La notazione *infissa* richiede *regole di priorità e associatività*
  - per identificare univocamente *quale* operatore sia applicato a *quali* operandi



# OPERATORI INFISSI, PREFISSI E POSTFISSI

---

- Notazione prefissa:

**\* + 4 5 6**

- si legge come  $(4 + 5) * 6$
- denota quindi 54

- Notazione postfissa:

**4 5 6 + \***

- si legge come  $4 * (5 + 6)$
- denota quindi 44

# PRIORITÀ DEGLI OPERATORI

---

- **PRIORITÀ (o precedenza) : specifica l'ordine di valutazione degli operatori quando in una espressione compaiono *operatori (infissi) diversi***

**Esempio:**     $3 + 10 * 20$

– si legge come  $3 + (10 * 20)$  perché l'operatore  $*$  è prioritario rispetto a  $+$

- NB: operatori diversi possono comunque avere *uguale priorità*

# ASSOCIATIVITÀ DEGLI OPERATORI

---

- **ASSOCIATIVITÀ**: specifica *l'ordine di valutazione* degli operatori quando in una espressione compaiono **operatori (infissi) di uguale priorità**
- Un operatore può quindi essere **associativo a sinistra o associativo a destra**

**Esempio:**    **3 - 10 + 8**

- si legge come  $(3 - 10) + 8$  perché gli operatori - e + sono equiprioritari e **associativi a sinistra**

# PRIORITÀ E ASSOCIATIVITÀ

---

Priorità e associatività predefinite possono essere **alterate mediante l'uso di parentesi**

**Esempio:**  $(3 + 10) * 20$   
– denota 260 (anziché 203)

**Esempio:**  $3 - (10 + 8)$   
– denota -15 (anziché 1)

# INCREMENTO E DECREMENTO

---

Gli operatori di incremento e decremento sono *usabili in due modi*

- **come pre-operatori: ++v**  
*prima incremento e poi uso nell'espressione*
- **come post-operatori: v++**  
*prima uso nell'espressione poi incremento*

*Formule equivalenti:*

`v = v + 1;`

`v +=1;`

`++v;`

`v++;`

`Y = ++X` equivale a:

`X = X + 1;`

`Y = X;`

Mentre

`Y = X++` equivale a:

`Y = X;`

`X = X + 1;`

# Incremento/decremento con semantica non chiara

- Presenza di side-effects dovuti all'assegnamento.
- Semantica non chiara
- Esempio:
  - `int x =1; int y = ++x * x;`
  - Nota che l'ordine degli operatori è non specificato in C
  - Risultato: `x=2` ma `y` sarà `=4` se valutato prima `++x`, mentre `y=2` se valutato prima `x`.

# CHE COSA STAMPA?

---

```
main()  
{ int c;  
  c=5;  
  printf("%d\n", c);  
  printf("%d\n", c++);  
  printf("%d\n\n", c);  
  c=5;  
  printf("%d\n", c);  
  printf("%d\n", ++c);  
  printf("%d\n", c); }
```

Soluzione:

5

5

6

5

6

6

# CHE COSA SUCCEDE?

---

```
a=0;  
printf("ciao e %d\n", a=1);
```

```
a=0;  
printf("ciao e %d\n", a==1);
```

```
a=0;  
printf("ciao e %d\n", a==0);
```

```
a=0;  
printf("ciao e a=%d\n", a);
```

```
a=1;  
if (a=4)  
    printf("ciao e a=%d\n", a);
```



# CHE COSA SUCCEDE?

---

```
a=0;  
printf("ciao e %d\n", a=1);
```

ciao e 1

```
a=0;  
printf("ciao e %d\n", a==1);
```

ciao e 0

```
a=0;  
printf("ciao e %d\n", a==0);
```

ciao e 1

```
a=0;  
printf("ciao e a=%d\n", a);
```

ciao e a=0

```
a=1;  
if (a=4)  
    printf("ciao e a=%d\n", a);
```

ciao e a=4

# ESEMPI

---

- `int i, k = 5;`

`i = ++k /* i vale 6, k vale 6 */`

- `int i, k = 5;`

`i = k++ /* i vale 5, k vale 6 */`

- `int i=4, j, k = 5;`

`j = i + k++; /* j vale 9, k vale 6 */`

- `int j, k = 5;`

`j = ++k - k++; /* DA NON USARE */`

`/* j vale 0, k vale 7 ? Indefinito */`

# RIASSUNTO OPERATORI DEL C (1)

---

<b>Priorità</b>	<b>Operatore</b>	<b>Simbolo</b>	<b>Associatività</b>
1 (max)	chiamate a funzione selezioni	() []    ->    .	a sinistra
2	operatori unari: op. negazione op. aritmetici unari op. incr. / decr. op. indir. e deref. op. sizeof	!    ~ +    - ++    -- &    * <b>sizeof</b>	a destra
3	op. moltiplicativi	*    /    %	a sinistra
4	op. additivi	+    -	a sinistra

# RIASSUNTO OPERATORI DEL C (2)

Priorità	Operatore	Simbolo	Associatività
5	op. di shift	>> <<	a sinistra
6	op. relazionali	< <= > >=	a sinistra
7	op. uguaglianza	== !=	a sinistra
8	op. di AND bit a bit	&	a sinistra
9	op. di XOR bit a bit	^	a sinistra
10	op. di OR bit a bit		a sinistra
11	op. di AND logico	&&	a sinistra
12	op. di OR logico		a sinistra
13	op. condizionale	? . . . :	a destra
14	op. assegnamento e sue varianti	= += -= *= /= %= &= ^=  = <<= >>=	a destra
15 (min)	op. concatenazione	,	a sinistra

# La libreria math.h

- Gli operatori del C presentati non consentono calcoli complessi (es. radice quadrata)
- Le funzioni matematiche più comuni (trigonometriche, esponenziali, logaritmiche ecc..) sono fornite dalla libreria standard `math.h`
- Prendono ingresso di tipo `double` e uscita di tipo `double`.
- `HUGE_VAL` rappresenta infinito (- meno)