Capitolo 7

Tipi di base

Tipi di base

- I ***tipi di base*** (built-in) in C sono:
  - Numeri interi (`short int`, `int`, `long int` e `unsigned`)
  - Numeri in virgola mobile (`float`, `double`, e `long double`)
  - Caratteri `Char`

Notazione decimale

- Nella notazione decimale:
  - Si usano 10 cifre, i simboli 0,1,2,3,4,5,6,7,8 e 9
  - le cifre vengono "pesate" in funzione della posizione con una potenza di 10
  - 10 è la ***base***
- Esempio:

	10^3	10^2	10^1	10^0	(peso potenza base)
cifre	**3**	**2**	**0**	**7**	
	1000	100	10	1	(peso valore esplicito)

- Valore = $3 \times 1000 + 2 \times 100 + 7 \times 1 = \mathbf{3207}_{10}$

Notazioni binaria, ottale, esadecimale

- Notazione binaria: la **base** è 2, le cifre sono 0 e 1

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
b	*b*	*b*	*b*	*b*	*b*	*b*	*b*
128	64	32	16	8	4	2	1

- Esempio

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	**0**	**1**	**1**	**0**	**0**	**0**	**1**
128	64	32	16	8	4	2	1

- Valore in binario = $\mathbf{10110001}_2$
- Valore in decimale = 128+32+16+1 = $\mathbf{177}_{10}$

Notazioni binaria, ottale, esadecimale

- Notazione ottale: la **base** è 8, le cifre sono 0-7

8^3	8^2	8^1	8^0
c	c	c	c
512	64	8	1

- Esempio

8^3	8^2	8^1	8^0
2	**0**	**0**	**4**
512	64	8	1

- Valore in ottale = $\mathbf{2004}_8$
- Valore in decimale = $2 \times 512 + 4 \times 1 = \mathbf{1028}_{10}$

Notazioni binaria, ottale, esadecimale

- Notazione esadecimale: la **base** è 16
  - Si usano 0-9 e le lettere A,B,C,D,E,F
  - A vale 10, B vale 11, …., F vale 15

- Esempio

16^3	16^2	16^1	16^0
3	**B**	**0**	**A**
65536	256	16	1

- Valore in esadecimale = $\mathbf{3B0A}_{16}$
- Valore in decimale = $3 \times 65536 + 11 \times 256 + 10 \times 1 = \mathbf{133898}_{10}$

Interi senza segno

- Possono essere di varie "taglie":
  - `unsigned short int` (es. 2 byte)
  - `unsigned int` (es. 4 byte)
  - `unsigned long int` (es. 8 byte)

- La "taglia" non è stabilita dallo standard ma dipende dalla macchina (dalla CPU)

- Il valore è memorizzato ovviamente in binario

Interi senza segno

- Per un `unsigned int` di 2 byte il valore massimo è 2^{16}-1=65635, che corrisponde al valore binario

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- Conversione binario decimale:

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Interi senza segno

- Per un `unsigned int` di 4 byte il valore massimo è 2^{32}-1=4.294.967.295, che corrisponde al valore binario

  11111111 11111111 11111111 11111111

- Per un `unsigned int` di 8 byte il valore massimo è 2^{64}-1=18.446.744.073.709.551.615, che corrisponde al valore binario

  11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111

Interi con segno

- Come quelli senza segno, possono essere di varie "taglie":
  - `short int` (es. 2 byte)
  - `int` (es. 4 byte)
  - `long int` (es. 8 byte)

- Il primo bit (quello più a sinistra, cioè quello più significativo) determina il segno
  - 0 per i numeri positivi
  - 1 per i numeri negativi

Interi con segno

- Per gli interi con segno il bit più a sinistra (quello più significativo) ha un peso negativo
  - Anzichè contribuire per 2^{15} (2 byte), 2^{31} (4 byte) o 2^{63} (8 byte), contribuisce per -2^{15} (2 byte), -2^{31} (4 byte) o -2^{63} (8 byte)
  - Quindi vale 1 per i numeri negativi e 0 per quelli positivi (per questo viene detto ***bit di segno***)
- Conversione binario decimale:

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
-32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Interi con segno

- Per un `int` (con segno) di 2 byte il valore massimo è 2^{15}-1=32767, che corrisponde al valore binario

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

- Mentre il valore minimo è -2^{16}=-32768, che corrisponde al valore binario

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Interi con segno

- Per un `int` (con segno) di 4 byte il valore massimo è $2^{31}-1$=2.147.483.647, mentre il valore minimo è -2^{31}=-2.147.483.648

- Per un `int` (con segno) di 8 byte il valore massimo è $2^{63}-1$=9.223.372.036.854.775.807, mentre il valore minimo è -2^{63}=-9.223.372.036.854.775.808

Interi

- Gli specificatori `long` e `short`, come pure `signed` and `unsigned`, possono essere combinati con `int` per dichiarare variabili interi

- Quindi abbiamo 6 combinazioni:

```
short int          unsigned short int
int                unsigned int
long int           unsigned long int
```

- L'ordine è irrilevante. Anche la parola chiave `int` può essere omessa: `long int` può essere abbreviato in `long`
  - Ovviamente non si può omettere `int` per il tipo `int`

Taglia degli interi: CPU a 16 bit

- Taglie tipiche degli interi su CPU a 16 bit:

Tipo	byte	Minimo	Massimo
`short int`	2	–32,768 (-2^{15})	32,767 ($2^{15}-1$)
`unsigned short int`	2	0	65,535 ($2^{16}-1$)
`int`	2	–32,768 (-2^{15})	32,767 ($2^{15}-1$)
`unsigned int`	2	0	65,535 ($2^{16}-1$)
`long int`	4	–2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31}-1$)
`unsigned long int`	4	0	4,294,967,295 ($2^{32}-1$)

Taglia degli interi: CPU a 32 bit

- Taglie tipiche degli interi su CPU a 32 bit:

Tipo	byte	Minimo	Massimo
`short int`	2	–32,768 (-2^{15})	32,767 ($2^{15}-1$)
`unsigned short int`	2	0	65,535 ($2^{16}-1$)
`int`	4	–2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31}-1$)
`unsigned int`	4	0	4,294,967,295 ($2^{32}-1$)
`long int`	4	–2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31}-1$)
`unsigned long int`	4	0	4,294,967,295 ($2^{32}-1$)

Taglia degli interi: CPU a 64 bit

- Taglie tipiche degli interi su CPU a 64 bit:

Tipo	byte	Minimo	Massimo
`short int`	2	–32,768 (-2^{15})	32,767 ($2^{15}-1$)
`unsigned short int`	2	0	65,535 ($2^{16}-1$)
`int`	4	–2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31}-1$)
`unsigned int`	4	0	4,294,967,295 ($2^{32}-1$)
`long int`	8	-9.223.372.036.854.775.808 (-2^{63})	9.223.372.036.854.775.807 ($2^{63}-1$)
`unsigned long int`	8	0	18.446.744.073.709.551.615 ($2^{64}-1$)

Costanti intere

- Le ***costanti*** sono numeri che vengono scritti in modo esplicito nel testo sorgente di un programma
  - Il linguaggio C permette di specificare le costanti in decimale (base 10), ottale (base 8) oppure esadecimale (base 16)
- Decimali: iniziano con un numero diverso da 0

  `15 255 32767`

- Ottali: iniziano con 0: `017 0377 077777`

- Esadecimali: iniziano con 0x: `0xf 0xff 0x7fff`

  Maiuscolo o minuscolo non fa differenza:
  - `0xff 0xfF 0xFf 0xFF 0Xff 0XfF 0XFf 0XFF`

Overflow

- Quando eseguiamo un'operazione aritmetica con numeri interi è possibile che il risultato diventi **troppo grande** per essere rappresentato
- Ad esempio se facciamo un'operazione fra 2 interi di tipo `int`, il risultato deve essere anch'esso di tipo `int`.
- Se il risultato non può essere rappresentato come `int` (ad es. perchè il suo valore è maggiore del massimo rappresentabile con `int`), diciamo che si è verificato un ***overflow***

Overflow

- Cosa succede quando c'è un overflow dipende da se gli operandi sono con o senza segno
  - Quando l'overflow si verifica per un'operazione su numeri **con** segno, il comportamento del programma *non è definito*
  - Quando l'overflow si verifica per un'operazione su numeri **senza** segno, il comportamento del programma *è definito*: il risultato viene memorizzato in modulo 2^n, dove n è il numero di bit usati dalla rappresentazione:
    - In pratica si "buttano" i bit che vanno nell'overflow

Leggere e scrivere numeri interi

- La lettura e la scrittura di interi di vario tipo, richiede **nuovi specificatori** di conversione
- Quando si legge o si scrive un intero senza segno (*unsigned*) si usa la lettera `u` al posto di `d` nello specificatore di conversione. Le lettere `o` o `x` per ottale e esadecimale:

```
unsigned int u;

scanf("%u", &u);   /* reads  u in base 10 */
printf("%u", u);   /* writes u in base 10 */
scanf("%o", &u);   /* reads  u in base  8 */
printf("%o", u);   /* writes u in base  8 */
scanf("%x", &u);   /* reads  u in base 16 */
printf("%x", u);   /* writes u in base 16 */
```

Leggere e scrivere numeri interi

- Quando si scrive o si legge un intero `short`, si deve mettere la lettera `h` prima di `d`, `o`, `u`, o `x`:

```
short s;

scanf("%hd", &s);
printf("%hd", s);
```

- Per gli interi `long`, si usa la lettera `l` ("elle", non "uno") prima `d`, `o`, `u`, o `x`.
- Per gli interi `long long` (C99), si usano la lettere `ll` prima di `d`, `o`, `u`, o `x`.

Programa: somma di numeri (rivisitata)

- Il programma `sum.c` (Capitolo 6) somma una serie di numeri interi
- Un problema di questo programma è che la somma (o anche uno degli input) potrebbe creare un overflow per la variabile `int`
- Ecco un esempio di cosa potrebbe succedere su una macchina con `int` a 16 bit:

```
This program sums a series of integers.
Enter integers (0 to terminate): 10000 20000 30000 0
The sum is: -5536
```

- Si ricordi che quando c'è un overflow relativo ad interi con segno il comportamento del programma non è definito
- Possiamo migliorare il programma usando variabili `long`

sum2.c

```
/* Sums a series of numbers (using long variables) */

#include <stdio.h>

int main(void)
{
  long n, sum = 0;

  printf("This program sums a series of integers.\n");
  printf("Enter integers (0 to terminate): ");

  scanf("%ld", &n);
  while (n != 0) {
    sum += n;
    scanf("%ld", &n);
  }
  printf("The sum is: %ld\n", sum);

  return 0;
}
```

Numeri in virgola mobile

- Il C prevede 3 tipi per i numeri in virgola mobile (tipi ***floating***):

  - `float` Numeri in virgola mobile con precisione singola
  - `double` Numeri in virgola mobile con precisione doppia
  - `long double` Numeri in virgola mobile con precisione estesa

Specificatore di conversione

- Per i numeri in virgola mobile (`double`) si utilizzano gli specificatori di conversione `%e`, `%f`, e `%g`
- Per leggere un valore `double`, si mette la lettera `l` prima di `e`, `f`, o `g`:

```
double d;

scanf("%lf", &d);
```

- *Nota:* Usare `l` solo per `scanf` **non** per `printf`
- Nella `printf`, gli specificatori `e`, `f`, e `g` vanno bene sia per valori `float` che `double`
- Quando si legge o si scrive un valore `long double`, si mette la lettera `L` prima di `e`, `f`, o `g`.

Caratteri

- L'ultimo tipo di base è `char` che serve a rappresentare i caratteri
- I valori che può assumere una variabile di tipo `char` possono variare da computer a computer in quanto macchine diverse possono usare insiemi di caratteri diversi

Insiemi di caratteri

- L'insieme di caratteri più utilizzato è l'insieme ***ASCII*** (American Standard Code for Information Interchange), un codice a 7 bit che permette di definire 128 caratteri
  - Non contiene le vocali accentate

- Il codice ASCII viene spesso esteso ad un insieme di caratteri detto ***Latin-1*** che contiene 256 caratteri.
  - Contiene le vocali con l'accento

Codice ASCII

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	‘
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

Insiemi di caratteri

- Ad una variabile di tipo `char` si può assegnare un qualsiasi carattere, specificando direttamente il carattere:

```
char ch;

ch = 'a';    /* lower-case a */
ch = 'A';    /* upper-case A */
ch = '0';    /* zero         */
ch = ' ';    /* space        */
```

- Si noti che i caratteri "costanti" sono esplicitati usando le virgolette singole `' '`
  - non virgolette doppie " "

Operazioni con i caratteri

- È possibile fare "operazioni" con i caratteri in quanto *il linguaggio C tratta i caratteri come interi*
- Nel codice ASCII, i codici dei caratteri vanno da 0000000_2 a 1111111_2, che corrispondono agli interi decimali da 0 a 127.
- Il carattere `'a'` corrisponde al valore 97, `'A'` al valore 65, `'0'` al valore 48, e `' '` al valore 32.
- Le costanti-carattere di fatto sono di tipo `int`

Operazioni con i caratteri

- Quando un carattere viene usato in un'espressione esso viene valutato come numero intero
- Ad esempio, usando il codice ASCII:

```
char ch;
int i;

i = 'a';          /* i is now 97    */
ch = 65;          /* ch is now 'A'  */
ch = ch + 1;      /* ch is now 'B'  */
ch++;             /* ch is now 'C'  */
```

Operazioni con i caratteri

- I caratteri possono essere confrontati (visto che sono di fatto numeri)
- Ecco un'istruzione `if` che converte un carattere da minuscolo a maiuscolo:

```
if ('a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

- Confronti del tipo `'a' <= ch` sono eseguiti sui valori numerici del codice che rappresenta il carattere

Operazioni con i caratteri

- Per esempio è facile scrivere un ciclo `for` il cui controllo passa attraverso tutti i caratteri maiuscoli:

```
for (ch = 'A'; ch <= 'Z'; ch++) …
```

- Svantaggi:
  - Si possono fare errori che non vengono identificati dal compilatore
  - È possibile scrivere espressioni apparentemente senza senso `'a' * 'b' / 'c'`.
  - Si potrebbe minare la portabilità in quanto il reale valore numerico che corrisponde ad un carattere dipende dal codice che si usa
    - Normalmente è ASCII, ma può essere diverso

Sequenze di "escape"

- Una costante "carattere" di solito è un carattere racchiuso fra virgolette singole.
- Alcuni caratteri - come il newline - non posso essere esplicitati in questo modo perché non è possibile "stamparli" oppure perché il carattere ha un significato speciale
- Le ***sequenze di escape*** forniscono un modo per rappresentare questi caratteri
- Esistono due tipi di sequenze di escape: sequenze di escape con ***caratteri*** e con ***numeri***

Sequenze di “escape”

- Ecco la lista delle sequenze di escape con caratteri:

Nome	*Sequenza di escape*
Alert (bell)	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Backslash	`\\`
Question mark	`\?`
Single quote	`\'`
Double quote	`\"`

Funzioni per i caratteri

- La funzione di libreria `toupper` converte un carattere da minuscolo a maiuscolo:

```
ch = toupper(ch);
```

  - `toupper` restituisce la versione maiuscola del suo argomento
- Per usare la funzione `toupper` è necessario includere il file `ctype.h`

```
#include <ctype.h>
```

- La libreria standard del C fornisce molte altre funzioni utili per manipolare i singoli caratteri e le stringhe di caratteri

Usare `scanf` e `printf` per i caratteri

- Lo specificatore `%c` permette alle funzioni `scanf` e `printf` di leggere e stampare caratteri:

```
char ch;

scanf("%c", &ch);   /* reads one character */
printf("%c", ch);   /* writes one character */
```

- `scanf` non ignora i caratteri di spazio bianco (come fa per i numeri)
- Per forzare la `scanf` a saltare uno spazio prima di leggere il carattere è necessario esplicitare la presenza dello spazio nell'input prima dello specificatore `%c`:

```
scanf(" %c", &ch);
```

Usare `scanf` e `printf` per i caratteri

- Poiché `scanf` non ignora gli spazi bianchi è facile accorgersi della fine di una linea di input: basta controllare che il carattere appena letto sia il carattere "newline"
- Ecco un ciclo che legge ed ignora tutti i rimanenti caratteri fino al newline:

```
do {
   scanf("%c", &ch);
} while (ch != '\n');
```

- La prossima chiamata a `scanf` leggerà il primo carattere della prossima linea di input

Usare `getchar` e `putchar` per i caratteri

- Per l'input e l'output di caratteri singoli si possono usare le funzioni `getchar` e `putchar` al posto di `scanf` e `printf`
- `putchar` scrive un carattere:

```
putchar(ch);
```

- Invece `getchar` legge un nuovo carattere dall'input ad ogni chiamata; il carattere letto è anche il valore di ritorno

```
ch = getchar();
```

  - `getchar` restituisce un `int`, quindi non è raro vedere `ch` dichiarata di tipo `char`.
  - Come `scanf`, `getchar` non tralascia lo spazio bianco

Usare `getchar` e `putchar` per i caratteri

- Usare `getchar` e `putchar` (al posto di `scanf` e `printf`) rende i programmi più veloci.
  - `getchar` e `putchar` sono molto più semplici di `scanf` e `printf`, che sono progettate per leggere e scrivere diversi tipi di dati in molti formati

- `getchar` ha anche un altro vantaggio. Poichè restituisce il valore del carattere che legge, permette di semplificare il codice sorgente in molte situazioni tipiche

Usare `getchar` e `putchar` per i caratteri

- Consideriamo il ciclo con la `scanf` che abbiamo usato per ignorare i caratteri rimanenti fino al newline

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

- Riscrivendo il ciclo con `getchar` otteniamo:

```
do {
  ch = getchar();
} while (ch != '\n');
```

Usare `getchar` e `putchar` per i caratteri

- La chiamata a `getchar` può essere spostata nell'espressione di controllo del ciclo:

```
while ((ch = getchar()) != '\n')
   ;
```

- La variabile `ch` non è necessaria; è sufficiente il valore di ritorno di `getchar` che deve essere confrontato con il carattere newline:

```
while (getchar() != '\n')
   ;
```

Usare `getchar` e `putchar` per i caratteri

- `getchar` è utile in cicli che ignorano caratteri fino al raggiungimento di un particolare carattere
- Ecco un'istruzione che usa `getchar` per saltare un qualsiasi numero di spazi bianchi:

```
while ((ch = getchar()) == ' ')
    ;
```

- Quando il ciclo termina `ch` conterrà il primo carattere dell'input diverso dallo spazio

Usare `getchar` e `putchar` per i caratteri

- Fare attenzione quando si usa sia `getchar` che `scanf`.
- `scanf` normalmente lascia nell'input i caratteri che ha "controllato" ma non ha letto, come ad esempio il newline:

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

- In questo caso `getchar` leggerà il newline.

Programma: Lunghezza di un messagio

- Il programma `length.c` mostra la lunghezza di un messaggio inserito dall'utente

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

- `length2.c` è un programma più breve che non utilizza la variabile usata per memorizzare il carattere letto da `getchar`.

length.c

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  char ch;
  int len = 0;

  printf("Enter a message: ");
  ch = getchar();
  while (ch != '\n') {
    len++;
    ch = getchar();
  }
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```

length2.c

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  int len = 0;

  printf("Enter a message: ");
  while (getchar() != '\n')
    len++;
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```

Conversione di tipo

- Affinché il computer possa eseguire un'operazione aritmetica è necessario che gli operandi abbiano lo stesso tipo:
  - Devono essere memorizzati nello stesso modo e con lo stesso numero di bit
- Se si mischiano operandi di tipo diverso, il compilatore potrebbe dover modificare il tipo per valutare l'espressione
  - Se sommiamo uno `short` di 16 bit ad un `int` di 32 bit, il compilatore convertirà la rappresentazione dello `short` da 16 a 32 bit
  - Se sommiamo un `int` ad un `float`, il compilatore convertirà l'`int` nel formato del `float`.

Conversione di tipo

- Queste trasformazioni vengono eseguite in modo automatico senza che il programmatore faccia niente
- Vengono dette ***conversioni implicite***

- Le regole per la conversione implicita sono un po' complicate in quanto esistono molti tipi aritmetici

- Il C permette al programmatore di eseguire delle ***conversioni esplicite*** utilizzando l'***operatore di conversione (cast)***

Conversione esplicita (casting)

- Una conversione esplicita ha la seguente forma:

  (*type-name*) *expression*

  dove *type-name* specifica il tipo al quale convertire il risultato dell'espressione *expression*

Conversione esplicita (casting)

- Si può usare la conversione esplicita per calcolare la parte frazionaria di un valore `float`:

```
float f, frac_part;

frac_part = f - (int) f;
```

- La differenza fra `f` e `(int) f` è la parte frazionaria di `f`, che è stata persa nella conversione

- Possiamo usare la conversione esplicita anche quando questa avverrebbe implicitamente
  - per rendere chiaro che c'è una conversione

```
i = (int) f;  /* f is converted to int */
```

Conversione esplicita (casting)

- Le conversioni esplicite ci permettono di forzare il compilatore a fare una conversione
- Esempio:

```
float quotient;
int dividend, divisor;

quotient = dividend / divisor;
```

- Per evitare l'approssimazione ad intero del risultato si forza il compilatore a considere il risultato un `float`

```
quotient = (float) dividend / divisor;
```

- Il cast di `dividenda` `float` fa sì che il compilatore converta anche `divisora` `float` e, pertanto, il risultato sarà anch'esso `float`

Conversione esplicita (casting)

- Il C considera (*type-name*) come un operatore unario
- Gli operatori unari hanno precedenza maggiore rispetto agli operatori binari, quindi

  `(float) dividend / divisor`

  viene interpretato come

  `((float) dividend) / divisor`

Definizione di tipi

- La direttiva `#define` può essere usata per definire una macro per il tipo booleano:

```
#define BOOL int
```

- C'è un modo migliore usando una ***definizione di tipo:***

```
typedef int Bool;
```

- `Bool` adesso può essere usato nello stesso modo in cui si usano i tipi predefiniti
- esempio:

```
Bool flag;    /* same as int flag; */
```

Vantaggi della definizione di tipi

- Definire dei tipi specifici può rendere il programma più comprensibile
- Se le variabili `cash_in` e `cash_out` saranno usate per memorizzare somme di denaro, usare un tipo `Dollars` definito come

```
typedef float Dollars;
```

e scrivere

```
Dollars cash_in, cash_out;
```

dà più informazioni rispetto a

```
float cash_in, cash_out;
```

Vantaggi della definizione di tipi

- Inoltre modifiche al programma possono essere più semplici
- Ad esempio per ridefinire `Dollars` come `double`, ci basta cambiare la definizione del tipo

  ```
  typedef double Dollars;
  ```

- Senza la definizione del tipo avremmo dovuto trovare tutte le varibili di tipo `float` che utilizziamo per le somme di denaro e cambiare la loro dichiarazione

Definizioni di tipi e portabilità

- La definizione di tipi è importante anche per la portabilità
- Uno dei maggiori problemi che si incontra quando si porta il programma da un computer ad un altro è che i tipi possono essere diversi
- Se `i` è una variabile `int` l'assegnamento

  ```
  i = 100000;
  ```

  funziona su una macchina a 32 bit ma non su una macchina a 16 bit

Definizioni di tipi e portabilità

- Per migliorare la portabilità, si può usare `typedef` per definire nuovi nomi per i tipi interi
- Supponiamo di avere un programma che necessita di variabili con valori nell'intervallo 0–50.000.
- Potremmo usare variabili `long`, ma sarebbe meglio usare variabili `int` in quanto le operazioni aritmetiche su variabili `int` sono in genere più veloci. Inoltre le variabili `int` necessitano di meno spazio

Definizioni di tipi e portabilità

- Al posto di `int` possiamo definire un nuovo tipo:

  ```
  typedef int Quantity;
  ```

  ed usare questo tipo per le variabili del programma:

  ```
  Quantity q;
  ```

- Quando portiamo il programma su una macchina con interi troppo piccoli non dobbiamo far altro che cambiare la definizione del tipo:

  ```
  typedef long Quantity;
  ```

L'operatore `sizeof`

- Il valore dell'espressione

  `sizeof ( `*type-name*` )`

  è un intero senza segno che rappresenta il numero di byte necessari a memorizzare il tipo *type-name*.
- `sizeof(char)` è sempre 1, ma la grandezza di altri tipi può variare da macchina a macchina
- Su una macchina a 32 bit tipicamente `sizeof(int)` è 4
- Si può usare anche su costanti, espressioni e variabili

L'operatore `sizeof`

- Fare attenzione alla stampa di un valore `sizeof` perché il tipo di un'espressione `sizeof`, che è `size_t` dipende dall'implementazione

- In C89, è meglio convertire il valore prima di stamparlo:

```
printf("Size of int: %lu\n",
       (unsigned long) sizeof(int));
```

- La funzione `printf` del C99 gestisce direttamente il tipo `size_t` grazie allo specificatore di conversione `z`:

```
printf("Size of int: %zu\n", sizeof(int));
```

… arrivederci alla prossima lezione