Tipi semplici built-in

- interi
- reali
- caratteri

Per ciascun tipo consideriamo i seguenti aspetti:

1. intervallo di definizione (se applicabile)
2. notazione (sintassi) per le costanti
3. operatori
4. predicati (operatori di confronto)
5. formati di ingresso/uscita

Tipi interi: interi con segno

- `int`: un intero rappresentabile sulla macchina (segnaposto `%d`)
- dimensione e gamma di valori rappresentabile dipende dalla macchina su cui viene compilato il programma.
- la funzione predefinita `sizeof()` fornisce la lunghezza in byte di un qualsiasi tipo o variabile C. Controllare sul proprio con il comando: `printf("%d\n",sizeof(int));`

Tipi interi: interi con segno (cont.)

- 3 tipi:
  `short`
  `int`
  `long`
- **Intervallo di definizione:** da -2^{n-1} a $2^{n-1}-1$, dove `n` dipende dal compilatore
- I valori limite sono contenuti nel file limits.h (da includere), che definisce le costanti:
  `SHRT_MIN`, `SHRT_MAX`, `INT_MIN`, `INT_MAX`, `LONG_MIN`, `LONG_MAX`
- Vale: `sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)`
  `sizeof(short)` $\geq$ 2 (ovvero, almeno 16 bit)
  `sizeof(long)` $\geq$ 4 (ovvero, almeno 32 bit)
- Compilatore gcc: `short`: 16 bit, `int`: 32 bit, `long`: 32 bit

Notazione per le costanti: in decimale: `0`, `10`, `-10`, ...

- Per distinguere `long` (solo nel codice): `10L` (oppure `10l`, ma `l` sembra `1`).

Operatori: `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `>`, `<=`, `>=`

N.B.: l'operatore di uguaglianza si rappresenta con `==` (mentre `=` è utilizzato per il comando di assegnamento!)

Ingresso/uscita: tramite `printf` e `scanf`, con i seguenti specificatori di formato (dove `d` indica "decimale"):

`%hd` per `short`
`%d` per `int`
`%ld` per `long` (con `l` minuscola)

Tipi interi: interi senza segno

- 3 tipi:
  `unsigned short`
  `unsigned int`
  `unsigned long`
- **Intervallo di definizione:** da 0 a 2^n-1, dove n dipende dal compilatore.
  Il numero n di bit è lo stesso dei corrispondenti interi con segno.
- Le costanti definite in limits.h sono:
  `USHRT_MAX`, `UINT_MAX`, `ULONG_MAX` (n.b. il minimo è sempre 0)

Notazione per le costanti:

- decimale: come per interi con segno
- esadecimale: `0xA`, `0x2F4B`, ...
- ottale: `012`, `027513`, ...

- Nel codice si possono far seguire le cifre del numero dallo specificatore `u` (ad esempio `10u`).

Ingresso/uscita: tramite `printf` e `scanf`, con i seguenti specificatori di formato:

`%u`	per numeri in decimale
`%o`	per numeri in ottale
`%x`	per numeri in esadecimale con cifre `0`, ..., `9`, `a`, ..., `f`
`%X`	per numeri in esadecimale con cifre `0`, ..., `9`, `A`, ..., `F`

Per interi `short` si antepone `h`
`long` si antepone `l` (minuscola)

Operatori: tutte le operazioni vengono fatte modulo 2^n.

Caratteri

- Servono per rappresentare caratteri alfanumerici attraverso opportuni codici, tipicamente il codice ASCII (American Standard Code for Information Interchange).
- Un codice associa ad ogni carattere un intero:
  Esempio: Codice ASCII:

carattere:	'0'	···	'9'	':'	';'	'<'
intero (in decimale):	48	···	57	58	59	60

carattere:	'a'	···	'z'	'{'	'|'	'}'
intero (in decimale):	97	···	122	123	124	125

carattere:	'A'	···	'Z'	'['	'\'	']'
intero (in decimale):	65	···	90	91	92	93

Caratteri (cont.)

- un singolo byte, in grado di contenere un carattere (codifica ASCII). Segnaposto %c.

```
char a='a'; // i caratteri si indicano tra apici
```

- In C i caratteri possono essere usati come gli interi (un carattere coincide con il codice che lo rappresenta).
  Infatti:

```
int a='a';
printf("%c\n",a);
printf("%d\n",a);
```

  stampa prima a e poi 97, che corrisponde alla codifica ASCII di a
- Nessuna relazione tra un carattere per rappresentare una cifra e la cifra stessa: '2' non è 2.

Intervallo di definizione: dipende dal compilatore

- Vale: `sizeof(char)` $\leq$ `sizeof(int)`
  Tipicamente i caratteri sono rappresentati con 8 bit.

  Operatori: sono gli stessi di `int` (operazioni effettuate utilizzando il codice del carattere).

  Costanti: `'A'`, `'#'`, ...

  Esempio:

```
char x, y, z;
x = 'A';
y = '\n';
z = '#';
```

- In C l'apice singolo ' delimita singoli caratteri, mentre l'apice doppio " delimita stringhe di caratteri.

Come non va usato il codice

- Confrontiamo:

```
char x, y, z;
x = 'A';
y = '\n';
z = '#';
```

```
char x, y, z;
x = 65;  /* codice ASCII di 'A' */
y = 10;  /* codice ASCII di '\n' */
z = 35;  /* codice ASCII di '#' */
```

- Non è sbagliato, però è pessimo stile di programmazione.
- Non è detto che il codice dei caratteri sia quello ASCII.
  $\Longrightarrow$ Il programma non sarebbe portabile.

Ingresso/uscita: tramite `printf` e `scanf`, con specificatore di formato `%c`

Attenzione: in ingresso non vengono saltati gli spazi bianchi e gli a capo

Esempio:

```
int i, j;
printf("Immetti due interi\n");
scanf("%d%d", &i, &j);
printf("%d %d\n", i, j);
```

Immetti due interi
> 18 25↩
18 25

```
int i, j;
char c;
printf("Immetti due interi\n");
scanf("%d%c%d", &i, &c, &j);
printf("%d %d %d\n", i, c, j);
```

Immetti due interi
> 18 25↩
18 32 25

- 32 è il codice ASCII del carattere ' ' (spazio)

Attenzione:

- è necessario specificare i salti degli spazi o di altri caratteri simili solo nel caso si leggano caratteri (ovvero quando si usa lo specificatore '%c'). Ad esempio in scanf("%c % c", &x, &y) dove voglio leggere due caratteri separati da uno spazio bianco non significativo.
- non è necessario se si desidera leggere interi, reali o stringhe. Ad esempio in scanf("%d%d", &i, &j)

- Funzioni per la stampa e la lettura di un singolo carattere (da usare in alternativa a `printf` e `scanf`):

  `putchar(c);` ... stampa il carattere memorizzato in `c`

  `c = getchar();` ... legge un carattere e lo assegna alla variabile `c`

Esempio:

```
char c;
putchar('A');
putchar('\n');
c = getchar();
putchar(c);
```

Tipi reali

- `float`, `double`,`long double`: sono i tipi usati per rappresentare i reali, con diversi gradi precisione. Si usa la notazione in virgola mobile (floating point).

```
float x=123.34;
double y=100.1e5; //anche notazione scientifica
```

- segnaposto `%f` e `%L`;

Tipi reali (cont.)

I reali vengono rappresentati in virgola mobile (floating point).

- 3 tipi:
  `float`
  `double`
  `long double`
- **Intervallo di definizione:**

	sizeof	cifre significative	min esp.	max esp.
`float`	4	6	−37	38
`double`	8	15	−307	308
`long double`	12	18	−4931	4932

- Le grandezze precedenti dipendono dal compilatore e sono definite nel file `float.h`.
- Deve comunque valere la relazione:
  `sizeof(float)` $\leq$ `sizeof(double)` $\leq$ `sizeof(long double)`

Costanti: con punto decimale o notazione esponenziale

Esempio:

```
double x, y, z, w;
x = 123.45;
y = 0.0034;     /* oppure y = .0034 */
z = 34.5e+20;   /* oppure z = 34.5E+20 */
w = 5.3e-12;
```

- Nei programmi, per denotare una costante di tipo
  - `float`, si può aggiungere `f` o `F` finale
    Esempio: `float x = 2.3e5f;`
  - `long double`, si può aggiungere `L` o `l` finale
    Esempio: `long double x = 2.34567e520L;`

Operatori: come per gli interi (tranne "%")

Ingresso/uscita: tramite `printf` e `scanf`, con diversi specificatori di formato

Output con printf (per `float`):

- `%f` ... notazione in virgola fissa

  `%8.3f` ... 8 cifre complessive, di cui 3 cifre decimali

  Esempio:

```
float x = 123.45;
printf("|%f| |%8.3f| |%-8.3f|\n", x, x, x);
```

```
|123.449997| | 123.450| |123.450 |
```

  dove il `-` indica l'allineamento a sinistra

- `%e` (oppure `%E`) ... notazione esponenziale

  `%10.3e` ... 10 cifre complessive, di cui 3 cifre decimali

  Esempio:

```
double x = 123.45;
printf("|%e| |%10.3e| |%-10.3e|\n", x, x, x);
```

```
|1.234500e+02| | 1.234e+02| |1.234e+02 |
```

Input con scanf (per `float`):
si può usare indifferentemente `%f` o `%e`.

Riassunto degli specificatori di formato per i tipi reali:

	float	double	long double
printf	`%f`, `%e`	`%f`, `%e`	`%Lf`, `%Le`
scanf	`%f`, `%e`	`%lf`, `%le`	`%Lf`, `%Le`

Booleani ed operatori

- In C non esiste un tipo Booleano. Si usa il tipo `int`:
  - `0` rappresenta FALSO;
  - `diverso da 0` (di solito 1) rappresenta VERO.
- Operatori logici:
  - `!`: NOT (operatore unario). Esempio: `!a`;
  - `&&`: AND (operatore binario). Esempio: `a && b`;
  - `||`: OR (operatore binario). Esempio: `a || b`;

  Restituiscono un valore intero pari a 0 o 1 a seconda del valore (falso/vero) dell'espressione.
  Altri operatori lavorano sui singoli bit: ad esempio operatori di shift (<<, >>), AND (&), OR (|), XOR (^) ...

Conversioni di tipo nelle espressioni

- Le espressioni aritmetiche hanno un valore ed un tipo dettato da quello delle variabili in gioco.
- In un'operazione gli operandi di tipo diverso vengono convertiti nello stesso tipo applicando alcune regole automatiche. In genere si amplia il campo dei valori rappresentabili.

Esempio

Se `x` ha tipo `int` e `y` ha tipo `float`, il risultato dell'espressione `x+y` viene automaticamente convertito a `float`

Conversioni di tipo

Situazioni in cui si hanno conversioni di tipo

- quando in un'espressione compaiono operandi di tipo diverso
- durante un'assegnamento `x = y`, quando il tipo di `y` è diverso da quello di `x`
- esplicitamente, tramite l'operatore di cast
- nel passaggio dei parametri a funzione (più avanti)
- attraverso il valore di ritorno di una funzione (più avanti)

Una conversione può o meno coinvolgere un cambiamento nella rappresentazione del valore.

da `short` a `long` (dimensioni diverse)
da `int` a `float` (anche se stessa dimensione)

Conversioni implicite tra operandi di tipo diverso nelle espressioni

Quando un'espressione del tipo x *op* y coinvolge operandi di tipo diverso, avviene una conversione implicita secondo le seguenti regole:

1. ogni valore di tipo `char` o `short` viene convertito in `int`
2. se dopo il passo 1. l'espressione è ancora eterogenea si converte l'operando di tipo inferiore facendolo divenire di tipo superiore secondo la seguente gerarchia:

   `int` → `long` → `float` → `double` → `long double`

Esempio: `int x; double y;`
Nel calcolo di `(x+y)`:

1. `x` viene convertito in `double`
2. viene effettuata la somma tra valori di tipo `double`
3. il risultato è di tipo `double`

Conversioni nell'assegnamento

Si ha in `x = exp` quando i tipi di `x` e `exp` non coincidono.

- La conversione avviene sempre a favore del tipo della variabile a sinistra:

  se si tratta di una promozione non si ha perdita di informazione

  se si ha una retrocessione si può avere perdita di informazione

  Esempio:

```
int i;
float x = 2.3, y = 4.5;
i = x + y;
printf("%d", i); /* stampa 6 */
```

- Se la conversione non è possibile si ha errore.

Conversioni esplicite (operatore di cast)

Sintassi: `(tipo) espressione`

- Converte il valore di `espressione` nel corrispondente valore del `tipo` specificato.

  Esempio:

```
int somma, n;
float media;
...
media = somma / n;          /* divisione tra interi */
media = (float)somma / n;   /* divisione tra reali */
```

- L'operatore di cast "`(tipo)`" ha precedenza più alta degli operatori binari e associa da destra a sinistra. Dunque
  `(float) somma / n`
  equivale a
  `((float) somma) / n`

Input/output

- Come già detto, input e output non sono parte integrante del C
- L'interazione con l'ambiente è demandato alla libreria standard
  $\Longrightarrow$ un insieme di funzioni a uso dei programmi C
- La libreria `stdio.h` implementa un semplice modello di ingresso e uscita di dati testuali
- un testo è trattato come un successione (stream) di caratteri, ovvero
  $\Longrightarrow$ una sequenza di caratteri organizzata in righe, ciascuna terminata da "`\n`"
- al momento dell'esecuzione, al programma vengono connessi automaticamente 3 stream:
  - standard input: di solito la tastiera
  - standard output: di solito lo schermo
  - standard error: di solito lo schermo

Input/output (cont.)

- Compito della libreria è fare in modo che tutto il trattamento dei dati in ingresso e uscita si conformi a questo modello

  $\Longrightarrow$ il programmatore non si deve preoccupare di come ciò sia effettivamente realizzato

- Ogni volta che si effettua una operazione di lettura attraverso `getchar` viene acquisito il prossimo carattere dallo standard input e viene restituito il suo valore

  (analogamente per `scanf` che comporta l'acquisizione di uno o più caratteri a seconda delle specifiche di formato presenti . . .)

- Ogni volta che si effettua una operazione di scrittura (attraverso `putchar` o `printf`) tutti i valori coinvolti vengono convertiti in sequenze di caratteri e queste ultime vengo accodate allo standard output.

- Tipicamente il sistema operativo consente di redirigere gli stream standard, ad esempio su uno o più file.

Funzioni `printf` e `scanf`

- Le funzioni `printf` e `scanf` sono le funzioni di libreria per l'output e per l'input. La lettera `f` alla fine dei nomi delle due funzioni sta per "formatted", ovvero "con formato"
- Sia `printf` che `scanf` ricevono una stringa di controllo, che può contenere le specifiche di conversione indicate con il simbolo % (segnaposto), e una serie di parametri, che possono essere ad esempio le variabili da stampare o leggere.

```
printf(stringa di controllo, lista di variabili)
scanf(stringa di controllo, lista di variabili)
```

- Per stampare delle variabili di un determinato tipo dobbiamo utilizzare i segnaposto relativi (`%d` per `int`,`%c` per `char`,...)
- NOTA: come per i comandi della shell, anche per le funzioni di libreria standard possiamo usare il comando `man`.

Formattazione dell'output con `printf`

- Riepilogo specificatori di formato principali:
  - interi: `%d`, `%o`, `%u`, `%x`, `%X`
    per `short`: si antepone `h`
    per `long`: si antepone `l` (minuscola)
  - reali: `%e`, `%f`, `%g`
    per `double`: non si antepone nulla
    per `long double`: si antepone `L`
  - caratteri: `%c`
  - stringhe: `%s` (le vedremo più avanti)
  - puntatori: `%p` (li vedremo più avanti)
- Flag: messi subito dopo il "%"
  - "-": allinea a sinistra
  - altri flag (non ci interessano)
- Sequenze di escape: `\%`, `\'`, `\"`, `\\`, `\a`, `\b`, `\n`, `\t`, ...
- Per stampare il carattere '%' è necessario raddoppiarlo: `%%`

Formattazione dell'input con `scanf`

- Specificatori di formato: come per l'output, tranne che per i reali
  - `double`: si antepone `l`
  - `long double`: si antepone `L`
- **Soppressione dell'input:** mettendo "*" subito dopo "%"
  Non ci deve essere un argomento corrispondente allo specificatore di formato: il campo deve essere letto, ma il valore non deve essere assegnato ad alcuna variabile.

  Esempio: Lettura di una data in formato gg/mm/aaaa oppure gg-mm-aaaa.

```
int g, m, a;
scanf("%d%*c%d%*c%d%*c", &g, &m, &a);
```

Espressioni booleane

- Il linguaggio deve consentire di descrivere espressioni booleane (guardie di condizionali e iterazione).
- Come già sappiamo, in C non esiste un tipo Booleano $\Longrightarrow$ si usa il tipo **int** :

  falso $\Longleftrightarrow$ 0

  vero $\Longleftrightarrow$ 1 (in realtà qualsiasi valore diverso da 0)

  Esempio: `2 > 3` ha valore 0 (ossia falso)

  `5 > 3` ha valore 1 (ossia vero)

- **Operatori relazionali del C**
  - `<`, `>`, `<=`, `>=` (minore, maggiore, minore o uguale, maggiore o uguale) — priorità alta
  - `==`, `!=` (uguale, diverso) — priorità bassa

  Esempio: `temperatura <= 0` `velocita > velocita_max`

  `voto == 30` `anno != 2000`

Operatori logici

- In ordine di priorità:
  - `!` (negazione) — priorità alta
  - `&&` (congiunzione)
  - `||` (disgiunzione) — priorità bassa

Semantica:

a	b	!a	a && b	a || b
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

0 . . . falso

1 . . . vero (o qualsiasi valore $\neq 0$)

Esempio:

`(a >= 10) && (a <= 20)` vero (1) se $10 \leq a \leq 20$

`(b <= -5) || (b >= 5)` vero se $|b| \geq 5$

- Le espressioni booleane vengono valutate da sinistra a destra:
  - con `&&`, appena uno degli operandi è falso, restituisce falso senza valutare il secondo operando
  - con `||`, appena uno degli operandi è vero, restituisce vero senza valutare il secondo operando
- **Priorità** tra operatori di diverso tipo:
  - not logico — priorità alta
  - aritmetici
  - relazionali
  - booleani (and e or logico) — priorità bassa

Esempio:

```
a+2 == 3*b || !trovato && c < a/3
```

è equivalente a

```
((a+2) == (3*b)) || ((!trovato) && (c < (a/3)))
```

Selezione doppia: istruzione **if-else**

Sintassi:

```
if   (espressione)
     istruzione1
else istruzione2
```

- `espressione` è un'espressione booleana
- `istruzione1` rappresenta il ramo then (deve essere un'unica istruzione)
- `istruzione2` rappresenta il ramo else (deve essere un'unica istruzione)

Semantica:

1. viene prima valutata `espressione`
2. se `espressione` è vera viene eseguita `istruzione1`
   altrimenti (ovvero se `espressione` è falsa) viene eseguita `istruzione2`

```
int temperatura;

printf("Quanti gradi ci sono?  "); scanf("%d", &temperatura);
if (temperatura >= 25)
  printf("Fa caldo\n");
else
  printf("Si sta bene\n");

printf("Arrivederci\n");
```

```
=> Quanti gradi ci sono? 30 ↩
Fa caldo
Arrivederci
=>
```

```
=> Quanti gradi ci sono? 18 ↩
Si sta bene
Arrivederci
=>
```

Selezione singola: istruzione **if**

- È un'istruzione **if-else** in cui manca la parte else.

Sintassi:

```
if (espressione)
   istruzione
```

Semantica:

1. viene prima valutata `espressione`
2. se `espressione` è vera viene eseguita `istruzione` altrimenti non si fa alcunché

Esempio:

```
int temperatura;
scanf("%d", &temperatura);
if (temperatura >= 25)
  printf("Fa caldo\n");

printf("Arrivederci\n");
```

```
=> 18 ↩
Arrivederci
```

```
=> 30 ↩
Fa caldo
Arrivederci
```

Blocco

- La sintassi di **if-else** consente di avere un'unica istruzione nel ramo then (o nel ramo else).
- Se in un ramo vogliamo eseguire più istruzioni dobbiamo usare un blocco.

  Sintassi:

```
{
    istruzione-1
    ...
    istruzione-n
}
```

- Come già sappiamo e come rivedremo più avanti, un blocco può contenere anche dichiarazioni.

Esempio: Dati mese ed anno, calcolare mese ed anno del mese successivo.

```
int mese, anno, mesesucc, annosucc;

if (mese == 12)
{
  mesesucc = 1;
  annosucc = anno + 1;
}
else
{
  mesesucc = mese + 1;
  annosucc = anno;
}
```

Esempio: Dato un anno, controllare se si tratta di un anno bisestile. Ricordiamo che un anno è *bisestile* se lo si può dividere per 4, ma non per 100, ad eccezione degli anni divisibili per 400, che sono bisestili.

```
int anno;

if ((anno%4 == 0 && anno%100 !  = 0) || (anno%400 == 0))
printf(%d e' un anno bisestile \n, anno);
else
printf(%d e' un anno bisestile \n, anno);
```

Selezione multipla: If annidati (in cascata)

- Quando l'istruzione del ramo then o else è un'istruzione **if o if-else.**

  Esempio: Data una temperatura, stampare un messaggio secondo la seguente tabella:

temperatura t	messaggio
$30 < t$	Molto caldo
$20 < t \leq 30$	Caldo
$10 < t \leq 20$	Gradevole
$0 < t \leq 10$	Freddo
$t \leq 0$	Molto freddo

```
 if (temperatura > 30)
  printf("Molto caldo\n");
else if (temperatura > 20)
  printf("Caldo\n");
else if (temperatura > 10)
  printf("Gradevole\n");
else if (temperatura > 0)
  printf("Freddo\n");
else printf("Molto freddo\n");
```

Osservazioni:

- si tratta di un'unica istruzione **if-else**

```
if (temperatura > 30)
  printf("Molto caldo\n");
else ...
```

- non serve che la seconda condizione sia composta

```
if (temperatura > 30) printf("Molto caldo\n");
else /* il valore di temperatura e' <= 30 */
    if (temperatura > 20)
```

  Non c'è bisogno di una congiunzione del tipo
  `(t <= 30) && (t > 20)`
  (analogamente per gli altri casi).

- **Attenzione**: il seguente codice

```
if (temperatura > 30) printf("Molto caldo\n");
if (temperatura > 20) printf("Caldo\n");
```

  ha ben altro significato (quale?)

Ambiguità dell'else

```
if (a >= 0) if (b >= 0) printf("b positivo");
else printf("???");
```

- `printf("???")` può essere la parte **else**
  - del primo **if** $\Longrightarrow$ `printf("a negativo");`
  - del secondo **if** $\Longrightarrow$ `printf("b negativo");`
- L'ambiguità sintattica si risolve considerando che un **else** fa sempre riferimento all'**if** più vicino, dunque

```
if (a > 0)
  if (b > 0)
    printf("b positivo");
  else

    printf("b negativo");
```

- Perché un **else** si riferisca ad un **if** precedente, bisogna inserire quest'ultimo in un blocco

```
 if (a > 0)
  { if (b > 0) printf("b positivo"); }
else

  printf("a negativo");
```

Esercizio

Leggere un reale e stampare un messaggio secondo la seguente tabella:

gradi alcolici g	messaggio
$40 < g$	superalcolico
$20 < g \leq 40$	alcolico
$15 < g \leq 20$	vino liquoroso
$12 < g \leq 15$	vino forte
$10.5 < g \leq 12$	vino normale
$g \leq 10.5$	vino leggero

Esempio: Dati tre valori che rappresentano le lunghezze dei lati di un triangolo, stabilire se si tratti di un triangolo equilatero, isoscele o scaleno.

Algoritmo: `determina tipo di triangolo`

```
leggi i tre lati
confronta i lati a coppie, fin quando non
  hai raccolto una quantità di informazioni
  sufficiente a prendere la decisione
stampa il risultato
```

```
main()   {
float primo, secondo, terzo;

printf("Lunghezze lati triangolo ?  ");
scanf("%f%f%f", &primo, &secondo, &terzo);

if (primo == secondo) {
  if (secondo == terzo)
    printf("Equilatero\n");
  else
    printf("Isoscele\n");
}
else {
  if (secondo == terzo)
    printf("Isoscele\n");
  else if (primo == terzo)
    printf("Isoscele\n");
  else
    printf("Scaleno\n");
}     }
```

Esercizio

Risolvere il problema del triangolo utilizzando il seguente algoritmo:

```
Algoritmo: determina tipo di triangolo con conteggio
leggi i tre lati
confronta i lati a coppie contando
  quante coppie sono uguali
if le coppie uguali sono 0
  è scaleno
  else if le coppie uguali sono 1
    è isoscele
    else è equilatero
```

Selezione a più vie: istruzione **switch**

- **Sintassi:**

```
switch (espressione) {
  case valore-1:  istruzioni-1
                  break;
  ...
  case valore-n:  istruzioni-n
                  break;
  default :  istruzioni-default
}
```

- **Semantica:**
  1. viene valutata `espressione`
  2. viene cercato il primo `i` per cui il valore di `espressione` è uguale a `valore-i`
  3. se si trova tale `i`, allora vengono eseguite `istruzioni-i` altrimenti vengono eseguite `istruzioni-default`

Equivale a `if (espressione == valore-1) istruzioni-1 else if (espressione == valore-2) istruzione-2 else...  istruzioni-default`

Esempio:

```
int giorno;
...
switch (giorno) {
  case 1:  printf("Lunedi'\n");
           break;
  case 2:  printf("Martedi'\n");
           break;
  case 3:  printf("Mercoledi'\n");
           break;
  case 4:  printf("Giovedi'\n");
           break;
  case 5:  printf("Venerdi'\n");
           break;
  default :  printf("Week end\n");
}
```

- Se abbiamo più valori a cui corrispondono le stesse istruzioni, possiamo raggrupparli come segue:

```
case valore-1:  case valore-2:
                        istruzioni
                        break;
```

Esempio:

```
int giorno;
...
switch (giorno) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:  printf("Giorno lavorativo\n");
           break;
  case 6:
  case 7:  printf("Week end\n");
  default :  printf("Giorno non valido\n");
}
```

Osservazioni sull'istruzione **switch**

- L'`espressione` usata per la selezione può essere una qualsiasi espressione C che restituisce un valore intero.
- I valori specificati nei vari **case** devono invece essere costanti (o meglio valori noti a tempo di compilazione). In particolare, non possono essere espressioni in cui compaiono variabili.

  Esempio: Il seguente frammento di codice è sbagliato:

```
int a;
switch (a) {
  case a<0:  printf("negativo\n");
             /* ERRORE: a<0 non è una costante*/
  case 0:  printf("nullo\n");
  case a>0:  printf("positivo\n");
             /* ERRORE: a>0 non è una costante*/
}
```

- In realtà il C non richiede che nei **case** di un'istruzione **switch** l'ultima istruzione sia **break**.

  Quindi, in generale la **sintassi** di un'istruzione **switch** è:

```
switch (espressione) {
  case valore-1:  istruzioni-1
  ...
  case valore-n:  istruzioni-n
  default :  istruzioni-default
}
```

Semantica:

1. viene prima valutata `espressione`
2. viene cercato il primo `i` per cui il valore di `espressione` è pari a `valore-i`
3. se si trova tale `i`, allora si eseguono in sequenza `istruzioni-i`, `istruzioni-(i+1)`, ..., fino a quando non si incontra **break** o è terminata l'istruzione **switch**,
   altrimenti vengono eseguite `istruzioni-default`

Esempio: più **case** di uno **switch** eseguiti in sequenza (corretto)

```
int lati;
printf("Immetti il massimo numero di lati del poligono (al piu' 6):  ");
scanf("%d", &lati);
printf("Poligoni con al piu' %d lati:  ", lati);

switch (lati) {
  case 6:  printf("esagono, ");
  case 5:  printf("pentagono, ");
  case 4:  printf("rettangolo, ");
  case 3:  printf("triangolo\n");
           break;
  case 2:  case 1:  printf("nessuno\n");
                    break;
  default :  printf("\nErrore:  valore immesso > 6.\n");
}
```

- N.B. Quando si omettono i **break**, diventa rilevante l'ordine in cui vengono scritti i vari **case**. Questo può essere facile causa di errori.

  È buona norma mettere **break** come ultima istruzione di ogni **case**

Esempio: più case eseguiti in sequenza (scorretto)

```
int b;
printf("Immetti un numero tra 1 e 6:  ");
scanf("%", &b);
switch (b) {
  case 1:  case 2:  case 3:  case 5:  printf("Numero primo\n");
  case 4:  case 6:                     printf("Numero non primo\n");
  default :                            printf("Valore non valido!\n");
}
```

=> 3↩
Numero primo
Numero non primo
Valore non valido!
=>

=> 4 ↩
Numero non primo
Valore non valido!
=>