Introduzione al linguaggio C

- Abbiamo già visto come un programma non sia altro che un algoritmo codificato in un linguaggio di programmazione.
- Problema: quale linguaggio scegliere per la codifica di un algoritmo?
  - Il linguaggio naturale sarebbe facilmente comprensibile ma non è eseguibile da una macchina.
  - Il linguaggio macchina che abbiamo brevemente illustrato è eseguibile ma di difficile comprensione.
- Due requisiti fondamentali di un qualsiasi linguaggio per la descrizione di algoritmi:
  - deve essere preciso per non lasciare adito a dubbi interpretativi
  - deve essere sintetico per non rendere difficile la comprensione dei programmi.

- Il linguaggio naturale e il linguaggio macchina si collocano in posizioni opposte, soddisfacendo uno solo dei requisiti.
- I linguaggi di programmazione ad alto livello sono progettati proprio per colmare tale gap.
  $\Longrightarrow$ sono linguaggi adatti a codificare algoritmi pur rimanendo comprensibili.
- La fatica di tradurre un programma nel linguaggio macchina è affidata a particolari programmi, i compilatori, che traducono programmi scritti nel linguaggio di più alto livello in programmi equivalenti nel linguaggio macchina.

- Vedremo il cosidetto ANSI C (standard del 1989, con successive aggiunte)
- Il primo programma C: ciao mondo

```
#include <stdio.h>
main()
  /* Stampa un messaggio sullo schermo. */
{
  printf("Ciao mondo!\n");
}
```

- Questo programma stampa sullo schermo una riga di testo:

```
Ciao mondo!
>
```

- Vediamo in dettaglio ogni riga del programma.


```
/* Stampa un messaggio sullo schermo.  */
```

- testo racchiuso tra "/*" e "*/" è un commento
- i commenti servono a chi scrive o legge il programma, per renderlo più comprensibile
- il compilatore ignora i commenti
- attenzione a non dimenticare di chiudere i commenti con */, altrimenti tutto il resto del programma viene ignorato

`main()`

- è una parte presente in tutti i programmi C
- le parentesi "(" e ")" dopo main indicano che main è una funzione
- i programmi C sono composti da una o più funzioni, tra le quali ci deve essere la funzione `main`
  $\Longrightarrow$ `main` è una funzione speciale, perché l'esecuzione del programma incomincia con l'esecuzione di `main`
- la parentesi "{" apre il corpo della funzione e "}" lo chiude
  - la coppia di parentesi e la parte racchiusa da esse costituiscono un blocco
  - il corpo della funzione contiene le istruzioni (e dichiarazioni) che costituiscono la funzione

`printf("Ciao mondo!\n");`

- è un'istruzione semplice (ordina al computer di eseguire un'azione) in questo caso visualizzare (stampare) sullo schermo la sequenza di caratteri tra apici
- ogni istruzione semplice deve terminare con ";"
- oltre alle istruzioni semplici, esistono anche istruzioni composte (che non devono necessariamente terminare con ";")
- la parte racchiusa in una coppia di doppi apici è una stringa (di caratteri)
- "\n" non viene visualizzato sullo schermo, ma provoca la stampa di un carattere di fine riga
  - "\" è un carattere di escape e, insieme al carattere che lo segue, assume un significato particolare (sequenza di escape)
- in realtà anche `printf` è una funzione, e l'istruzione di sopra è un'attivazione di funzione (le vedremo più avanti)

`#include <stdio.h>`

- è una direttiva di compilazione
- viene interpretata dal compilatore durante la compilazione
- la direttiva "`#include`" dice al compilatore di includere il contenuto di un file nel punto corrente
- `<stdio.h>` è un file che contiene i riferimenti alla libreria standard di input/output (dove è definita la funzione `printf`)
- il linguaggio C non prevede istruzioni esplicite di input/output. Queste operazioni sono definite tramite funzioni nella libreria standard di input/output.

Note:

- è importante distinguere i caratteri maiuscoli da quelli miniscoli `Main`, `MAIN`, `Printf`, `PRINTF` non andrebbero bene
- si è usata l'indentazione per mettere in evidenza la struttura del programma ▸)

Alcune varianti del programma

```
#include <stdio.h>
main()
  /* Stampa un messaggio sullo schermo. */
{
  printf("Ciao");
  printf(" mondo!\n");
}
```

- produce lo stesso effetto del programma precedente
- la seconda invocazione di `printf` incomincia a stampare dal punto in cui aveva smesso la prima
- Cosa viene stampato se usiamo

```
printf("Ciao");
printf("mondo!\n");
```

```
printf("Ciao\n");
printf("mondo!\n");
```

Un altro programma: area di un rettangolo

```
#include <stdio.h>

main() {
 int base;
 int altezza;
 int area;

 base = 3;
 altezza = 4;
 area = base * altezza;

 printf("Area: %d\n", area);
}
```

Quando viene eseguito stampa:

```
Area: 12
>
```


Le variabili

Servono a rappresentare, nei programmi, le associazioni (modificabili) dello stato
$\Longrightarrow$ cf. $x \rightsquigarrow val$ nello pseudo-linguaggio

Una variabile è caratterizzata dalle seguenti proprietà:

1. nome: serve a identificarla — esempio: `area`
2. valore: valore associato nello stato corrente — Esempio: 4 (può cambiare durante l'esecuzione)
3. tipo: specifica l'insieme dei possibili valori — Esempio: `int` (numeri interi)
4. indirizzo: della cella di memoria a partire dal quale è memorizzato il valore.

Nome, tipo e indirizzo non possono cambiare durante l'esecuzione.

Le variabili (cont.)

- Il nome di una variabile è un identificatore C
  $\Longrightarrow$ sequenza di lettere, cifre, e _ che inizia con una lettera o con _
  Esempio: `Numero_elementi`, `x1`, ma non `1_posto`
  - può avere lunghezza qualsiasi, ma solo i primi 31 caratteri sono significativi
  - lettere minuscole e maiuscole sono considerate distinte
- Ad ogni variabile è associata una cella di memoria o più celle consecutive, a seconda del suo tipo. Il suo indirizzo è quello della prima cella.
- Analogia con una scatola di scarpe etichettata in uno scaffale
  - nome $\Longrightarrow$ etichetta
  - valore $\Longrightarrow$ scarpa che c'è nella scatola
  - tipo $\Longrightarrow$ capienza (che tipo di scarpe ci metto dentro)
  - indirizzo $\Longrightarrow$ posizione nello scaffale (la scatola è incollata)

  N.B.
  - non tutte le variabili sono denotate da un identificatore
  - non tutti gli identificatori sono identificatori di variabile (ad es. funzioni, tipi, parole riservate, ...)

Area del rettangolo

- `int base;` — è una dichiarazione di variabile
  - viene creata la scatola e incollata allo scaffale
  - ha tipo `int` $\Longrightarrow$ può contenere interi
  - ha nome `base`
  - ha un indirizzo (posizione nello scaffale), che è quello della cella di memoria associata alla variabile
  - ha un valore iniziale, che però non è significativo (è casuale)
    $\Longrightarrow$ la scatola viene creata piena, però con una scarpa scelta a caso, ovvero
    $\Longrightarrow$ l'associazione nello stato è del tipo nome $\rightsquigarrow$?

- `int altezza;`
  `int area;`
  $\Longrightarrow$ come per `base`

Variabili numeriche

Variabili intere

- per dichiarare variabili intere si può usare il tipo `int`
- i valori di tipo `int` sono rappresentati in C con almeno 16 bit
- il numero effettivo di bit dipende dal compilatore
  Esempio: 32 bit per il compilatore gcc (usato in ambiente Unix)
- in C esistono altri tipi per variabili intere (`short`, `long`) — li vedremo più avanti

Variabili reali

- per dichiarare variabili reali si può usare il tipo `float`
  Esempio: `float temperatura;`

Area del rettangolo

`base = 3;`
è un'istruzione di assegnamento (come nel nostro pseudo-linguaggio)

- in C l'operatore di assegnamento è denotato dal simbolo "="
- come già sappiamo, l'effetto è di modificare una associazione nello stato
  $\Longrightarrow$ in questo caso il valore 3 viene associato a base, come?
  $\Longrightarrow$ il nuovo valore viene scritto nella spazio associato alla variabile
- a questo punto la variabile `base` ha un valore significativo
  $\Longrightarrow$ da base $\rightsquigarrow$? a base $\rightsquigarrow$ 3

`altezza = 4;` $\Longrightarrow$ come sopra

`area = base * altezza;`
a destra di "=" possono comparire espressioni $\Longrightarrow$ il valore assegnato è quello dell'espressione calcolata nello stato corrente

- una variabile all'interno di una espressione sta per il valore ad essa associato in quel momento (cf. pseudo-linguaggio)

Nota: operatori aritmetici tra interi del C `+`, `-`, `*`, `/`, `%`, ...

Area del rettangolo

```
printf("Area: %d\n", area);
```

- è un'istruzione di stampa
- il primo argomento è la stringa di formato che può contenere specificatori di formato
- lo specificatore di formato `%d` indica che deve essere stampato un intero in notazione decimale (`d` per decimal)
- ad ogni specificatore di formato nella stringa deve corrispondere un valore che deve seguire la stringa di formato tra gli argomenti di `printf`

  Esempio: `printf("%d%d···%d", i1, i2, ..., in);`
- nel caso di `printf("Ciao mondo!\n");` la stringa di formato non conteneva specificatori e quindi non vi erano altri argomenti.

Struttura dei programmi C

- Nel semplice programma che abbiamo appena analizzato possiamo già vedere la struttura generale di un programma C.

```
/* DIRETTIVE DI COMPILAZIONE */
#include <stdio.h>
 main() {

 /* PARTE DICHIARATIVA */
  int base;
  int altezza;
  int area;

 /* PARTE ESECUTIVA */
  base = 3;
  altezza = 4;
  area = base * altezza;
  printf("Area: %d\n", area);
}
```

Un programma C deve contenere nell'ordine:

- Una parte contenente direttive per il compilatore. Nel nostro programma la direttiva

```
#include <stdio.h>
```

- l'identificatore predefinito `main` seguito dalle parentesi `()`.
- due parti racchiuse tra parentesi graffe
  - la parte dichiarativa. Nell'esempio:

```
int base;
int altezza;
int area;
```

  - la parte esecutiva. Nell'esempio:

```
base = 3;
altezza = 4;
area = base * altezza;
printf("Area: %d\n", area);
```


La parte dichiarativa

- È posta prima della codifica dell'algoritmo e obbliga il programmatore a dichiarare i nomi simbolici che saranno presenti nello stato e di cui farà uso nella parte esecutiva. Contiene i seguenti elementi:
  - la sezione delle dichiarazioni di variabili;
  - la sezione delle dichiarazioni di costanti.
- Le dichiarazioni:
  - rendono più pesante la fase di costruzione dei programmi, ma
  - consentono di individuare e segnalare errori in fase di compilazione.

Esempio:

```
int x;
int alfa;
alfa = 0;
x=alfa;
alba=alfa+1;
```

- Nell'ultima linea abbiamo erroneamente scambiato una **b** con una **f** $\Longrightarrow$ il compilatore individua `alba` come variabile non dichiarata.

Dichiarazioni di variabili

- Abbiamo già visto esempi di dichiarazioni di variabili.

```
float x;
int base;
int altezza;
```

- Ad ogni variabile viene attribuito, al momento della dichiarazione, un tipo
  $\Longrightarrow$ specifica l'insieme dei valori che la variabile può assumere
- La dichiarazione può anche attribuire un valore iniziale alla variabile (inizializzazione)

```
int x = 0;
```

- Variabili dello stesso tipo possono essere dichiarate contemporaneamente

```
int base, altezza, area;
```

(ma inizializzate singolarmente)

Esempio: `int x, y, z=0;` solo `z` è inizializzata a `0`.

Dichiarazioni di costanti (variabili *read-only*)

- Una dichiarazione di costante crea un'associazione non modificabile
  $\Longrightarrow$ associa in modo permanente un valore ad un identificatore.

  Esempio:

```
const float PiGreco=3.14;
const int N=100;
```

- L'associazione tra il nome `PiGreco` ed il valore `3.14` non può essere modificata durante l'esecuzione.
- Come per le dichiarazioni di variabili, più costanti dello stesso tipo possono essere dichiarate insieme

  Esempio:

```
const float PiGreco=3.14, e=2.718;
const int N=100, M=200;
```

- **N.B.** cosa succede quando si modifica una variabile read-only non è specificato dallo standard ANSI C, dipende dal compilatore.

Uso di costanti

- Con la dichiarazione `const float PiGreco=3.14;`
  l'istruzione
  `AreaCerchio=PiGreco*RaggioCerchio*RaggioCerchio;`
  è equivalente a
  `AreaCerchio=3.14*RaggioCerchio*RaggioCerchio`
- Maggiore leggibilità dei programmi, dovuta all'uso di nomi simbolici
- Maggiore adattabilità dei programmi che usano costanti

  Esempio:
  Per aumentare la precisione, basta cambiare la dichiarazione in
  `const float PiGreco = 3.1415;`

  Senza l'uso della costante si dovrebbero rimpiazzare nel codice tutte le occorrenze di `3.14` in `3.1415` ...

Area di un rettangolo di dimensioni lette da tastiera

```
#include <stdio.h>

main()
{
  int base, altezza, area;

  printf("Immetti base del rettangolo e premi INVIO\n");
  scanf("%d", &base);
  printf("Immetti altezza del rettangolo e premi INVIO\n");
  scanf("%d", &altezza);

  area = base * altezza;

  printf("Area: %d\n", area);
}
```

Nuova istruzione: `scanf("%d", &base);`

- `scanf` è la funzione duale di `printf`
- legge da input (tastiera) un valore intero e lo assegna alla variabile `base`
- `"%d"` è la stringa di controllo del formato (in questo caso viene letto un intero in formato decimale)
- "`&`" è l'operatore di indirizzo
  - `&base` indica (l'indirizzo del)la locazione di memoria associata a `base`
  - `scanf` memorizza in tale locazione il valore letto
- quando viene eseguita `scanf` il programma si mette in attesa che l'utente immetta un valore. Quando l'utente digita Invio
  1. la sequenza di caratteri immessa viene convertita in un intero (formato `%d`) e
  2. l'intero ottenuto viene assegnato alla variabile `base` (viene cioè scritto nella/e cella/e di memoria a partire dall'indirizzo passato a `scanf`)

N.B. il precedente valore della variabile `base` va perduto (cf. Input(base) nell pseudo-linguaggio.)

Esempio di esecuzione

- Vediamo cosa avviene durante l'esecuzione (indichiamo in rosso ciò che l'utente digita e in particolare con $\hookleftarrow$ il tasto Invio).

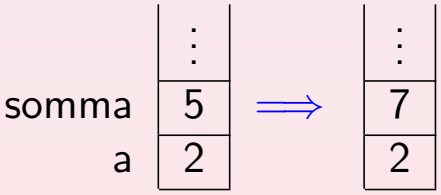
```
Immetti base del rettangolo e premi INVIO
5 ↩
Immetti altezza del rettangolo e premi INVIO
4 ↩
Area: 20
```

Assegnamento

- Ricordiamo che l'esecuzione di `x = exp` corrisponde a:
  1. valutare il valore dell'espressione `exp` a destra di "=" (usando i valori correnti delle variabili);
  2. assegnare poi tale valore alla variabile `x` a sinistra di "=".

Esempio:

```
somma = 5;
a = 2;
somma = somma + a;
```



Esempio:

	a	b
`int a, b;`	?	?
`a = 2;`	2	?
`b = 3;`	2	3
`a = b;`	3	3
`a = a + b;`	6	3
`b = a + b;`	6	9

Osservazioni sull'assegnamento

- **Attenzione:** A sinistra di "=" ci deve essere un identificatore di variabile

  $\Longrightarrow$ denota la corrispondente associazione modificabile nello stato.

  Esempio: Quali istruzioni sono corrette e quali no?

  `a = a;` SI corretta (anche se poco significativa ...)

  `a = 2 * a;` SI corretta (il valore associato ad `a` viene raddoppiato)

  `5 = a;` NO, `5` non denota una associazione modificabile nello stato ma un valore costante

  `a + b = c;` NO, `a+b` è un'espressione, non una variabile!

Tipi di dato semplici

- Abbiamo visto nei primi esempi che il C tratta vari tipi di dato

  $\Longrightarrow$ le dichiarazioni associano variabili e costanti al corrispondente tipo
- Per tipo di dato si intende un insieme di valori e un insieme di operazioni che possono essere applicate ad essi.

  Esempio:
  I numeri interi `{..., -2, -1, 0, 1, 2, ...}` e le usuali operazioni aritmetiche (somma, sottrazione, ...)
- Ogni tipo di dato ha una propria rappresentazione in memoria (codifica binaria) che utilizza un certo numero di celle di memoria.
- Il meccanismo dei tipi ci consente di trattare le informazioni in maniera astratta, cioè prescindendo dalla sua rappresentazione concreta.

L'uso di variabili con tipo ha importanti conseguenze quali:

- per ogni variabile è possibile determinare a priori l'insieme dei valori ammissibili e l'insieme delle operazioni ad essa applicabili
- per ogni variabile è possibile determinare a priori la quantità di memoria necessaria per la sua rappresentazione
- è possibile rilevare a priori (a tempo di compilazione) errori nell'uso delle variabili all'interno di operazioni non lecite per il tipo corrispondente

  Esempio: Nell'espressione `y + 3` se la variabile `y` non è stata dichiarata di tipo numerico si ha un errore (almeno dal punto di vista concettuale) rilevabile a tempo di compilazione (cioè senza eseguire il programma).

Classificazione dei tipi

- Tipi semplici: consentono di rappresentare informazioni semplici
  Esempio: una temperatura, una misura, una velocità, ecc.
- Tipi strutturati: consentono di rappresentare informazioni costituite dall'aggregazione di varie componenti
  Esempio: una data, una matrice, una fattura, ecc.
- Un valore di un tipo semplice è logicamente indivisibile, mentre un valore di un tipo strutturato può essere scomposto nei valori delle sue componenti
  Esempio: un valore di tipo data è costituito da tre valori (semplici)
- Il C mette a disposizione un insieme di tipi predefiniti (tipi built-in) e dei meccanismi per definire nuovi tipi (tipi user-defined)

  Nota: con `T` identificatore di tipo, nel seguito indichiamo con `sizeof(T)` lo spazio (in byte) necessario per la memorizzazione di valori di tipo `T` (vedremo che `sizeof` è una funzione C).

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

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

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

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

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 = '#';
```


Come non va usato il codice

- Confrontiamo:

```
char x, y, z;          char x, y, z;
x = 'A';               x = 65;   /* codice ASCII di 'A' */
y = '\n';              y = 10;  /* codice ASCII di '\n' */
z = '#';               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.
- Vedremo presto un modo per sfruttare l'ordinamento tra caratteri molto utile.

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)

- Funzioni per la stampa e la lettura di un singolo carattere:

  `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

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 |
```

- `%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`

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 quest'ultime vengo accodate allo standard output.
- Tipicamente il sistema operativo consente di redirigere gli stream standard, ad esempio su uno o più file.

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`, ...

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.

  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

- Come già sappiamo, il linguaggio deve consentire di descrivere espressioni booleane (guardie di condizionali e iterazione).
- 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 (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)))
```


Come va usato il codice dei caratteri

Esempio:

```
char x, y, z;
x = 'a';
y = 'd';
```

- Posso valutare $(x \leq y)$.
- Cosa mi dice? Se x precede y nell'ordine alfabetico.

Esempio:

```
char ch1 = 'a';
char ch2 = 'c';
char ch3 = (char) ((ch1 + ch2)/2);
printf("%c'', ch3);
```

- Cosa stampa? Stampa il carattere `'b'`.

Come va usato il codice dei caratteri

- Convertiamo una lettera minuscola in maiuscolo:

  Esempio:
  ```
  char lower = 'k';
  char upper = (char) (lower - 'a' + 'A');
  printf("%c", upper);
  ```

- Convertiamo un carattere numerico (una cifra) nell'intero corrispondente:

  Esempio:
  ```
  char ch1 = '9';
  int num = ch1-'0';
  ```

- Questi frammenti di programma sono completamente portabili (non dipendono dal codice usato per la rappresentazione dei caratteri).

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
=>

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;
}
```

If annidati (in cascata)

- Si hanno 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
```


Istruzione **switch**

- Può essere usata per realizzare una selezione a più vie.

  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`

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");
           break;
  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 e' una costante*/
  case 0:  printf("nullo\n");
  case a>0:  printf("positivo\n");
             /* ERRORE: a>0 non e' 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** di uno **switch** 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!
=>