Corso di Fondamenti di Informatica

Corso di Laurea in Ingegneria Elettronica

Prof. Diego Calvanese

Anno Accedemico 2001/2002

Introduzione all'elaborazione automatica delle informazioni

Algoritmi e programmi

Problemi da risolvere usando elaboratori possono essere di natura molto varia.

Esempi

1. Dati due numeri, trovare il maggiore.
2. Dato un elenco di nomi e numeri di telefono (rubrica o elenco telefonico) e un nome, trovare il numero di telefono corrispondente.
3. Data la struttura di una rete stradale e le informazioni sui flussi dei veicoli, determinare il percorso più veloce da A a B.
4. Scrivere tutti i numeri pari che non sono la somma di due numeri primi (Congettura di Goldbach).
5. Decidere per ogni programma C e per ogni dato in ingresso, se il programma C termina quando viene eseguito su quel dato.

Caratteristica comune ai problemi

informazioni in ingresso $\Longrightarrow$ informazioni in uscita

Osservazioni sulla formulazione dei problemi:

- descrizione non fornisce un metodo risolutivo (si pensi all'esempio 3)
- descrizione del problema è talvolta **ambigua** o **imprecisa** (ad esempio, 2 con Mario Rossi che compare più volte)
- per alcuni problemi **non è noto un metodo risolutivo** (ad esempio 4)
- esistono problemi per i quali è stato dimostrato che **non può esistere un metodo risolutivo** (ad esempio 5)

Noi consideriamo solo problemi per i quali è noto esistere un metodo risolutivo.

Per delegare ad un calcolatore la soluzione di un problema è necessario:

1. Individuare un **algoritmo** che risolve il problema, ovvero un insieme di passi che, eseguiti in ordine, permettono di calcolare i risultati a partire dalle informazioni a disposizione.

   Proprietà di un algoritmo:

   non-ambiguità: le istruzioni devono essere univocamente interpretabili dall'esecutore

   eseguibilità: ogni istruzione deve poter essere eseguita (in tempo finito) con le risorse a disposizione

   finitezza: l'esecuzione dell'algoritmo deve terminare in tempo finito per ogni insieme di dati in ingresso

2. **Rappresentare in un linguaggio di programmazione (LDP)**

(a) l'algoritmo		(a) programma
(b) le informazioni a disposizione		(b) dati in ingresso
(c) le informazioni utilizzate dall'algoritmo	$\Longrightarrow$	(c) dati ausiliari
(d) le informazioni fornite al termine		(d) dati in uscita

Riassumendo:

problema $\Longrightarrow$ **algoritmo** $\Longrightarrow$ **programma**

Individuare un algoritmo per un dato problema può essere molto complesso:
$\Longrightarrow$ Conviene operare per **livelli di astrazione**:

- si parte da una versione molto generale
- si **raffinano** via via le varie parti dell'algoritmo fino ad ottenere una descrizione dettagliata che può essere direttamente codificata in un LDP

$\Longrightarrow$ **metodo dei raffinamenti successivi**

Pseudocodice per la specifica di algoritmi

- si usano **frasi in linguaggio naturale** che esprimono operazioni o condizioni
- operazioni vengono eseguite in sequenza, tranne che per le . . .
- **strutture di controllo** dell'esecuzione delle operazioni (specificate usando parole chiave)

if condizione	**for** ogni valore compreso tra . . . e . . .
then operazione 1	**do** operazione
else operazione 2	
while condizione	**do** operazione
do operazione	**while** condizione

Lo pseudocodice si presta bene al metodo dei raffinamenti successivi: un **raffinamento** consiste nel sostituire un'operazione con

- una sequenza di operazioni, oppure
- una struttura di controllo

Esempio: Calcolo del massimo comun divisore di due interi positivi m ed n.

Algoritmo

1. calcola l'insieme I dei divisori di m
2. calcola l'insieme J dei divisori di n
3. calcola l'insieme K intersezione di I e J (divisori comuni)
4. restituisci il valore massimo tra quelli in K

Raffinamento del passo 1

1.1 inizializza I all'insieme vuoto
1.2 **for** ogni numero i compreso tra 2 ed m
do if i è divisore di m
then aggiungi i ad I

Esercizio: scrivere un algoritmo per il calcolo del MCD che si basi sulla seguente proprietà

$$MCD(m,n) = \begin{cases} m, & \text{se } m = n \\ MCD(m-n,n), & \text{se } m > n \\ MCD(m,n-m), & \text{se } m < n \end{cases}$$

Programmi: rappresentano algoritmi in un linguaggio di programmazione

Esempio: Programma che

1. legge due numeri da tastiera
2. ne calcola il MCD e
3. lo stampa

```
/* File: mcd2.c */
#include <stdio.h>
int main(void)
{ int m, n;
  printf("Immetti due interi positivi: ");
  scanf("%d%d", &m, &n);
  printf("Il massimo comun divisore di %d e %d e' ", m, n);
  while (m != n)
    if (m > n)
      m = m - n;
    else
      n = n - m;
  printf("%d\n", m);
  return 0;
}
```

I sistemi di elaborazione

Architettura del calcolatore

L'architettura è ancora quella classica sviluppata da **Von Neumann** nel 1947.

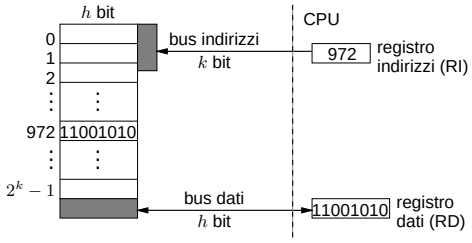
L'architettura di von Neumann riflette le funzionalità richieste da un elaboratore:

- memorizzare i dati e i programmi $\Longrightarrow$ **memoria principale**
- dati devono essere elaborati $\Longrightarrow$ **unità di elaborazione (CPU)**
- comunicazione con l'esterno $\Longrightarrow$ **unità di ingresso/uscita (periferiche)**
- le componenti del sistema di devono scambiarsi informazioni $\Longrightarrow$ **bus di sistema**

Tra le periferiche evidenziamo la **memoria secondaria**.

Memoria centrale (o RAM)

- è una sequenza di **celle di memoria** (dette **parole**), tutte della stessa dimensione
- ogni cella è costituita da una **sequenza di bit** (binary digit)
- il numero di bit di una cella di memoria (dimensione) dipende dall'elaboratore, ed è un multiplo di 8: 8, 16, 32, 64, . . .
- ogni cella di memoria è identificata in modo univoco dal suo **indirizzo**
- il numero di bit necessari per l'indirizzo dipende dal numero di celle di memoria
  k bit $\Longrightarrow$ 2^k celle



Operazione di lettura:

1. CPU scrive l'indirizzo della cella di memoria da cui leggere nel RI
2. esegue l'operazione ("apre i circuiti")
3. il valore della cella indirizzata viene trasferito nel RD

Operazione di scrittura: al contrario

Caratteristiche principali della memoria centrale

- è una memoria ad **accesso casuale**, ossia il tempo di accesso ad una cella di memoria è indipendente dalla posizione della cella $\Longrightarrow$ viene chiamata **RAM** (random access memory)
- può essere sia letta che **scritta**
  - scrittura distruttiva
  - lettura non distruttiva
- **alta velocità** di accesso (attualmente ca. 10ns, per lettura e scrittura)
- è **volatile** (si perde il contenuto quando si spegne il calcolatore)

Dimensione della memoria: misurata in byte

Kilobyte	$=$	2^{10}	$\sim$ 10^3	byte
Megabyte	$=$	2^{20}	$\sim$ 10^6	byte
Gigabyte	$=$	2^{30}	$\sim$ 10^9	byte
Terabyte	$=$	2^{40}	$\sim$ 10^{12}	byte

Tipi di memoria:

RAM random access memory (programmi e dati)

ROM read only memory (firmware)

PROM programmable ROM (possono essere scritte una sola volta)

EPROM erasable PROM (possono anche essere cancellate con luce UV)

EEPROM electrically EPROM (possono essere cancellate da appositi dispositivi)

Memoria secondaria

Caratteristiche:

- non volatile
- capacità maggiore della memoria centrale (decine di GB)
- tempo di accesso lento rispetto alla memoria centrale (ca. 10ms)
- accesso sequenziale e non casuale
- tipi di memoria secondaria: dischi rigidi, floppy, CDROM, CDRW, DVD, nastri, ...

Bus di sistema

Suddiviso in tre parti:

- bus indirizzi: k bit, unidirezionale
- bus dati: h bit, bidirezionale
- bus comandi: trasferisce i comandi da master a slave

$\Longrightarrow$ parallelismo (attualmente si arriva a 128 bit)

CPU (Central Processing Unit)

- coordina le attività di tutte le componenti del calcolatore
- interpreta le istruzioni che compongono il programma e le esegue
- 3 componenti principali:

  unità logico-aritmetica (ALU): effettua i calcoli aritmetici e logici

  unità di controllo: coordinamento di tutte le operazioni

  registri: celle di memoria ad accesso molto veloce all'interno della CPU
  - registro istruzione corrente (IR): contiene l'istruzione in corso di esecuzione
  - contatore di programma (PC): contiene l'indirizzo della prossima istruzione da eseguire
  - accumulatori: utilizzati dalla ALU per gli operandi ed il risultato
  - registro dei flag: memorizza alcune informazioni sul risultato dell'ultima operazione (carry, zero, segno, overflow, . . .)
  - registro interruzioni: utilizzato per la comunicazione con le periferiche

Tutte le attività interne alla CPU sono regolate da un orologio (**clock**) che genera impulsi regolari ad una certa frequenza (ad es. 800 MHz, 1 GHz, 2 GHz, . . .).

Ciclo dell'unità di controllo

Il **programma** è memorizzato in celle di memoria consecutive, sulle quali l'unità di controllo lavora eseguendo il ciclo di

prelievo — decodifica — esecuzione

Ogni istruzione è costituita da:

01001001	00110011
codice operativo	operandi

1. fase di **prelievo** (fetch)
   l'unità di controllo acquisisce l'istruzione indirizzata da PC e aggiorna PC in modo che indirizzi la prossima istruzione
2. fase di **decodifica**
   in base al codice operativo viene decodificato il tipo di istruzione per determinare quali sono i passi da eseguire per la sua esecuzione
3. fase di **esecuzione**
   vengono attivate le componenti che realizzano l'azione specificata

Esecuzione dei diversi tipi di istruzione

- ingresso dati: comanda all'unità di ingresso di trasferire i dati in memoria
- uscita dati: comanda all'unità di uscita di trasferire i dati dalla memoria
- operazione aritmetica o logica: comanda il trasferimento dei dati nell'ALU e l'esecuzione dell'operazione
- salto: aggiorna PC in modo opportuno

Le istruzioni dettano quindi il flusso del programma. Vengono eseguite in sequenza, a meno che non vi sia un'istruzione di salto:

do all'infinito
  preleva dalla memoria l'istruzione indirizzata da PC e caricala in IR
  decodifica l'istruzione
  esegui l'istruzione
  if l'istruzione non è un salto
  then incrementa il valore di PC

Dal codice sorgente al codice macchina

I concetti di algoritmo e di programma permettono di astrarre dalla reale struttura del calcolatore, che comprende sequenze di 0 e 1, ovvero un **linguaggio macchina**.

Livelli di astrazione ai quali possiamo vedere i programmi:

Linguaggio macchina (o codice binario): livello più basso di astrazione
- programma è una sequenza di 0 e 1 (suddivisi in parole) che codificano le istruzioni
- dipende dal calcolatore

Linguaggio assemblativo: livello intermedio di astrazione
- dipende dal calcolatore e le sue istruzioni sono in corrispondenza 1-1 con le istruzioni in linguaggio macchina
- istruzioni espresse in forma simbolica $\Longrightarrow$ comprensibile da un umano

Linguaggi ad alto livello: *Esempi:* C, Pascal, C++, Java, Fortran, Lisp, ...
- si basano su costrutti non elementari, comprensibili da un umano
- istruzioni sono più complesse di quelle eseguibili da un calcolatore (corrispondono a molte istruzioni in linguaggio macchina)
- in larga misura indipendenti dallo specifico elaboratore

Quindi, per arrivare dalla formulazione di un problema all'esecuzione del codice che lo risolve, bisogna passare attraverso **diversi stadi**:

problema
⇓ codifica del problema — *progettista*
algoritmo
⇓ codifica dell'algoritmo — *progettista*
codice sorgente (linguaggio ad alto livello)
⇓ compilazione — *compilatore*
codice oggetto (simile al codice macchina, ma contiene riferimenti simbolici)
⇓ collegamento tra diverse parti in codice oggetto — *collegatore (linker)*
codice macchina (eseguibile)
⇓ caricamento — *caricatore (loader)*
codice in memoria eseguito

Esempio: dati due interi positivi X ed Y, eseguire il loro prodotto usando solo le operazioni di somma e sottrazione

Algoritmo: moltiplicare X per Y, significa sommare X a se stesso per Y volte

leggi X ed Y
inizializza la somma a 0
for Y volte
do incrementa la somma del valore di X
stampa la somma

Raffinamento della parte ripetitiva

leggi X ed Y
inizializza la somma a 0
inizializza il contatore a 0
while contatore $< Y$
do incrementa la somma del valore di X
    incrementa il contatore di 1
stampa la somma

Codifica dell'algoritmo in C

```
#include <stdio.h>
int main (void)
{
  int x, y,
  int i = 0,
  int sum = 0;

  printf("Introduci due interi da moltiplicare\n");
  scanf("%d%d", &x, &y);
  while (i < y) {
    sum = sum + x;
    i = i + 1;
  }
  printf("La somma di %d e %d e' pari a %d\n", x, y, sum);
  return 0;
}
```


Compilazione in linguaggio macchina

Vediamo per comodità il codice in linguaggio assembler (corrisponde 1-1 al codice in linguaggio macchina, ma si legge meglio)

Un **esempio** di **linguaggio assembler**:

Operazione	**Codice assembler**	**Significato**
Caricamento di un dato	`LAOD R1 X` `LAOD R2 X`	Carica nel registro $R1$ (o $R2$) il dato memorizzato nella cella di memoria identificata dal nome simbolico X
Somma Sottrazione	`SUM R1 R2` `SUB R1 R2`	Somma (sottrae) il contenuto di $R2$ al contenuto di $R1$ e memorizza il risultato in $R1$
Memorizzazione	`STORE R1 X` `STORE R2 X`	Memorizza il contenuto di $R1$ ($R2$) nella cella di nome simbolico X
Lettura	`READ X`	Legge un dato e lo memorizza nella cella di nome simbolico X
Scrittura	`WRITE X`	Scrive il valore contenuto nella cella di nome simbolico X
Salto incondizionato	`JUMP A`	La prossima istruzione da eseguire è quella con etichetta A
Salto condizionato	`JUMPZ A`	Se il contenuto di $R1$ è uguale a 0, la prossima istruzione da eseguire è quella con etichetta A
Termine esecuzione	`STOP`	Ferma l'esecuzione del programma

- $R1$, $R2$ indicano i **registri** accumulatori della CPU
- X è un **variabile**, ovvero un nome per una locazione di memoria (a cui è associato un indirizzo)
- A indica l'**indirizzo** di una istruzione del programma (ovvero della cella di memoria che la contiene)

Programma per il prodotto in linguaggio assembler

	Etic.	Istr. assembler	Istruzione C	Significato
0		`READ X`	`scanf`	Leggi valore e mettilo nella cella identificata da X
1		`READ Y`	`scanf`	Leggi valore e mettilo nella cella identificata da Y
2		`LOAD R1 ZERO`	`i = 0`	Inizializzazione di I; metti 0 in $R1$
3		`STORE R1 I`		Metti il valore di $R1$ in I
4		`LOAD R1 ZERO`	`sum = 0`	Inizializzazione di SUM; metti 0 in $R1$
5		`STORE R1 SUM`		Metti il valore di $R1$ in SUM
6	`INIZ`	`LOAD R1 I`	`if (i == y)`	Esecuzione del test; metti in $R1$ il valore di I
7		`LOAD R2 Y`	`jump to FINE`	Metti in $R2$ il valore di Y
8		`SUB R1 R2`		Sottrai $R2$ (ossia Y) da $R1$
9		`JUMPZ FINE`		Se $R1 = 0$ (quindi $I = Y$) salta a $FINE$
10		`LOAD R1 SUM`	`sum = sum + x`	Somma parziale; metti in $R1$ il valore di SUM
11		`LOAD R2 X`		Metti in $R2$ il valore di X
12		`SUM R1 R2`		Metti in $R1$ la somma tra $R1$ ed $R2$
13		`STORE R1 SUM`		Metti il valore di $R1$ in SUM
14		`LOAD R1 I`	`i = i + 1`	Incremento contatore; metti in $R1$ il valore di I
15		`LOAD R2 UNO`		Metti 1 in $R2$
16		`SUM R1 R2`		Metti in $R1$ la somma tra $R1$ ed $R2$
17		`STORE R1 I`		Metti il valore di $R1$ in I
18		`JUMP INIZ`		Salta a $INIZ$
19	`FINE`	`WRITE SUM`	`printf`	Scrive il contenuto di SUM
20		`STOP`		Fine dell'esecuzione

Osservazioni sul codice assembler

- ad una istruzione C corrispondono in genere più istruzioni assembler (e quindi linguaggio macchina)

  Esempio: `sum = sum + x`

  $\Longrightarrow$
  1. carica il valore di X in un registro
  2. carica il valore di SUM in un altro registro
  3. effettua la somma tra i due registri
  4. memorizza il risultato nella locazione di memoria di SUM

- `JUMP` e `JUMPZ` interrompono la sequenzialità delle istruzioni

In realtà il compilatore (ed il linker) genera **linguaggio macchina**

- ogni istruzione è codificata come una sequenza di bit
- ogni istruzione occupa una (o più celle di memoria)
- istruzione costituita da 2 parti:
  - codice operativo
  - operandi

  (nel nostro caso abbiamo solo istruzioni a un solo operando)

Un esempio di linguaggio macchina

Istruzione assembler	Codice operativo
`LOAD R1` ind	0000
`LOAD R2` ind	0001
`STORE R1` ind	0010
`STORE R2` ind	0011
`SUM R1 R2`	0100
`SUB R1 R2`	0101
`JUMP` ind	0110
`JUMPZ` ind	0111
`READ` ind	1000
`WRITE` ind	1001
`STOP`	1011

	Indirizzo	Codice binario: Codice operativo	Codice binario: Indirizzo operando	Istr. assembler
0	00000	1000	10101	READ X
1	00001	1000	10110	READ Y
2	00010	0000	10111	LOAD R1 ZERO
3	00011	0010	11001	STORE R1 I
4	00100	0000	10111	LOAD R1 ZERO
5	00101	0010	11000	STORE R1 SUM
6	00110	0000	11001	LOAD R1 I
7	00111	0001	10110	LOAD R2 Y
8	01000	0101	-----	SUB R1 R2
9	01001	0111	10011	JUMPZ FINE
10	01010	0000	11000	LOAD R1 SUM
11	01011	0001	10101	LOAD R2 X
12	01100	0100	-----	SUM R1 R2
13	01101	0010	11000	STORE R1 SUM
14	01110	0000	11001	LOAD R1 I
15	01111	0001	11010	LOAD R2 UNO
16	10000	0100	-----	SUM R1 R2
17	10001	0010	11001	STORE R1 I
18	10010	0110	00110	JUMP INIZ
19	10011	1001	11000	WRITE SUM
20	10100	1011	-----	STOP
21	10101			X
22	10110			Y
23	10111	0000	00000	ZERO
24	11000			SUM
25	11001			I
26	11010	0000	00001	UNO

Rappresentazione binaria dell'informazione

Per informazione intendiamo tutto quello che viene manipolato da un calcolatore:

- numeri (naturali, interi, reali, . . .)
- caratteri
- immagini (statiche, dinamiche)
- suoni
- programmi

In un calcolatore tutte le informazioni sono rappresentate in **forma binaria**, come sequenze di **0** e **1**.

Per **motivi tecnologici**: distinguere tra due valori di una grandezza fisica è più semplice che non ad esempio tra dieci valori.

Rappresentazione di numeri naturali

- Un numero naturale è un oggetto matematico, che può essere **rappresentato** mediante una **sequenza di simboli** di un alfabeto fissato.
- È importante distinguere tra numero e sua rappresentazione: il **numerale** "234" è la rappresentazione del numero 234.
- Si distinguono **2 tipi di rappresentazione**:

  additiva: ad es. le cifre romane

  posizionale: una cifra contribuisce con un valore diverso al numero a seconda della posizione in cui si trova

Noi consideriamo solo la rappresentazione posizionale.

Un numero è rappresentato da una **sequenza finita di cifre** di un certo **alfabeto**:

$$c_{n-1}c_{n-2}\cdots c_1c_0 \;=\; N_b$$

c_0 ..... viene detta cifra **meno significativa**

c_{n-1} .... viene detta cifra **più significativa**

Il numero b di cifre diverse (dimensione dell'alfabeto) è detto **base** del sistema di numerazione. Ad ogni cifra è associato un valore compreso tra 0 e $b-1$.

Base	**Alfabeto**	**Sistema**
2	`0`, `1`	binario
8	`0`, . . . , `7`	ottale
10	`0`, . . . , `9`	decimale
16	`0`, . . . , `9`, `A`, . . . , `F`	esadecimale

Il significato di una sequenza di cifre (il numero N che essa rappresenta) dipende da b:

$$c_{n-1} \cdot b^{n-1} + c_{n-2} \cdot b^{n-2} + \cdots + c_1 \cdot b^1 + c_0 \cdot b^0 \quad = \quad \sum_{i=1}^{n} c_i \cdot b^i \quad = \quad N$$

Esempio: Il numerale `101` rappresenta numeri diversi a seconda del sistema usato:

Sistema	**Base** b	$\texttt{101})_b$	**Valore** (in decimale)
binario	2	$(\texttt{101})_2$	5
ottale	8	$(\texttt{101})_8$	65
decimale	10	$(\texttt{101})_{10}$	101
esadecimale	16	$(\texttt{101})_{16}$	257

Intervallo di rappresentazione con n cifre in base b: **da** 0 **a** $b^n - 1$

Esempio:

3 cifre in base 10 : da 0 a $999 = 10^3 - 1$

8 cifre in base 2 (1 byte) : da 0 a $255 = 2^8 - 1$

16 cifre in base 2 (2 byte) : da 0 a $65\,535 = 2^{16} - 1$

32 cifre in base 2 (4 byte) : da 0 a $4\,294\,967\,296 = 2^{32} - 1$

2 cifre in base 16 : da 0 a $255 = 16^2 - 1$

8 cifre in base 16 : da 0 a $4\,294\,967\,296 = 16^8 - 1$

Conversioni di base

Conversione da base b a base 10

Usando direttamente

$$c_{n-1} \cdot b^{n-1} + c_{n-2} \cdot b^{n-2} + \cdots + c_1 \cdot b^1 + c_0 \cdot b^0 \quad = \quad \sum_{i=1}^{n} c_i \cdot b^i \quad = \quad N$$

esprimendo le cifre e b in base 10 (e facendo i conti in base 10)

Esercizio: Scrivere l'algoritmo di conversione da base b a base 10.

Conversione da base 10 a base b

$$\begin{aligned} N &= c_0 + c_1 \cdot b^1 + c_2 \cdot b^2 + \cdots + c_{k-1} \cdot b^{k-1} \\ &= c_0 + b \cdot (c_1 + b \cdot (c_2 + \cdots + b \cdot (c_{k-1}) \cdots)) \end{aligned}$$

Vogliamo determinare le cifre $c_0, c_1, \ldots, c_{k-1}$

Consideriamo la divisione di N per b:

$$\begin{aligned} N &= R + b \cdot Q \qquad\qquad (0 \leq R < b) \\ &= c_0 + b \cdot (c_1 + b \cdot (\cdots)) \end{aligned}$$

$\Longrightarrow$ $R = c_0$ ovvero, il resto R della divisione di N per b dà c_0 (cifra meno significativa)
$Q = c_1 + b \cdot (\cdots)$

A partire dal quoziente Q si può iterare il procedimento per ottenere le cifre successive (fino a che Q diventa 0).

algoritmo converte N da base 10 a base b
  $i \leftarrow 0$
  while $N \neq 0$
  do $c_i \leftarrow N$ mod b
    $N \leftarrow N$ div b
    $i \leftarrow i + 1$

N.B. Le cifre vengono determinate dalla meno significativa a quella più significativa.

Esempio: $(25)_{10} = (???)_2$

$N : b$	Q	R	cifra
$25 : 2$	12	1	c_0
$12 : 2$	6	0	c_1
$6 : 2$	3	0	c_2
$3 : 2$	1	1	c_3
$1 : 2$	0	1	c_4

$(25)_{10} = (11001)_2$

N.B. servono 5 bit (con cui possiamo rappresentare i numeri da 0 a 31)

Conversione quando una base è potenza di un'altra

Da base b^k a base b (esempio: da base $8 = 2^3$ a base 2)

$$\begin{aligned} N = \; & c_{n-1} \cdot (b^k)^{n-1} + \\ & c_{n-2} \cdot (b^k)^{n-2} + \\ & \vdots \\ & c_0 \cdot (b^k)^0 \end{aligned}$$

Ogni cifra c_i è compresa tra 0 e $b^k - 1$ $\Longrightarrow$ possiamo rappresentarla in base b con al più k cifre

$$c_{i,k-1} \; c_{i,k-2} \; \cdots \; c_{i,0}$$

che rappresentano il numero

$$c_i = c_{i,k-1} \cdot b^{k-1} + c_{i,k-2} \cdot b^{k-2} + \cdots + c_{i,0} \cdot b^0$$

Sostituendo nell'equazione precedente si ottiene

$$\begin{aligned} N \;=\; & (c_{n-1,k-1}\cdot b^{k-1}+\cdots+c_{n-1,0}\cdot b^0)\cdot (b^k)^{n-1}+\\ & (c_{n-2,k-1}\cdot b^{k-1}+\cdots+c_{n-2,0}\cdot b^0)\cdot (b^k)^{n-2}+\\ & \quad\vdots\\ & (c_{0,k-1}\cdot b^{k-1}+\cdots+c_{0,0}b^0)\cdot (b^k)^0 \end{aligned}$$

Quindi i coefficienti in base b sono

$$c_{n-1,k-1}\; c_{n-1,k-2}\; \cdots\; c_{n-1,0}\; c_{n-2,k-1}\; \cdots\; c_{0,0}$$

$\Longrightarrow$ Si può convertire da base b^k a base b **convertendo ogni cifra separatamente**.

Esempio:

$$\begin{aligned} (542)_8 &= 5\cdot 8^2+4\cdot 8+2\\ &= (101)_2\cdot 2^6+(100)_2\cdot 2^3+(010)_2\\ &= (101\,100\,010)_2 \end{aligned}$$

Da base b a base b^k (esempio: da base 2 a base $16=2^4$):
Si può procedere in modo duale, raggruppando le cifre in gruppi di k a partire da destra.

Esempio: $(1100\,0101)_2=(C9)_{16}$

Rappresentazione di numeri interi

dobbiamo rappresentare anche il **segno** $\Longrightarrow$ si usa uno dei bit (quello più significativo)

Rappresentazione tramite modulo e segno

- il bit più significativo rappresenta il segno
- le altre $n-1$ cifre rappresentano il valore assoluto
- **problemi**:
  - doppia rappresentazione per lo zero ($00\cdots00$ e $10\cdots00$)
  - le operazioni aritmetiche sono complicate (analisi per casi)

$\Longrightarrow$ invece della rappresentazione tramite modulo e segno si usa una rappresentazione in complemento

Rappresentazione in complemento

In quanto segue:

- b indica la base
- n indica il numero complessivo di cifre
- consideriamo solo numeri in valore assoluto $\leq b^n$

Residuo modulo b^n di un intero X:

$$|X|_{b^n}=X-\lfloor X/b^n\rfloor\cdot b^n$$

dove $\lfloor Y\rfloor$ indica la parte intera inferiore di Y

Esempio: $b=10,\quad n=2$

$$\begin{aligned} |25|_{10^2} &= 25-\lfloor 25/10^2\rfloor\cdot 10^2=25-0=25\\ |-25|_{10^2} &= -25-\lfloor -25/10^2\rfloor\cdot 10^2=-25-(-1)\cdot 100=75 \end{aligned}$$

$\Longrightarrow$ per un numero positivo, il residuo è pari al numero stesso
per un numero negativo, il residuo è pari al complemento rispetto a b^n

- ad un numero corrisponde un unico residuo
- ad un residuo
  - corrispondono in generale due numeri, uno positivo ed uno negativo
  - corrisponde però un unico numero nell'intervallo $[-b^n/2,\ b^n/2)$

$\Longrightarrow$ **definizione semplificata di residuo**:

$$|X|_{b^n} = \begin{cases} X, & \text{se } 0 \leq X < b^n/2 \\ b^n - |X|, & \text{se } -b^n/2 \leq X < 0 \end{cases}$$

Il residuo viene utilizzato per la rappresentazione in complemento alla base.

Rappresentazione in complemento alla base (b con n cifre)

- è la rappresentazione di interi relativi nell'intervallo $[-b^n/2,\ b^n/2)$ tramite il residuo modulo b^n
  - se $X \geq 0$: RCB di X è compresa in $[0,\ b^n/2)$
  - se $X < 0$: RCB di X è compresa in $[b^n/2,\ b^n)$
- lo 0 ha una sola rappresentazione
- se $b = 2$ $\Longrightarrow$ **rappresentazione in complemento a 2**
  - positivi: cifra più significativa è 0 (rappresentati nella parte inferiore dell'intervallo)
  - negativi: cifra più significativa è 1 (rappresentati nella parte superiore dell'intervallo)

Esempio: $b = 2, \quad n = 6, \quad X = -9$

$$b^n - |X| = 1000000 - 1001 = 110111 (= 55_{10} = 64 - 9)$$

Osservazione:

$$\begin{array}{rcll} 2^n - |X| & = & 2^n - |X| + 1 - 1 & \\ & = & 2^n - 1 & n \text{ uni} \\ & & -|X| & \text{inverto i bit di } |X| \\ & & +1 & \text{sommo } 1 \end{array}$$

Esempio: $b = 2, \quad n = 6, \quad X = -9$

$$(9)_{10} = (001001)_2$$

inverto i bit: `110110`

sommo 1: `110111`

Metodo ancora più rapido:

1. si rappresenta $|X|$ con n bit
2. a partire da destra si lasciano inalterate tutte le cifre fino al primo 1 compreso
3. si invertono le rimanenti cifre

Operazioni su interi relativi in complemento a 2

Somma di due numeri

- si effettua **bit a bit**
- non è necessario preocuparsi dei segni
- il risultato sarà corretto (in complemento a 2 se negativo)
  (per la giustificazione vedi dispensa)
- può verificarsi **trabocco** (overflow) $\Longrightarrow$ il risultato non è corretto

  Si verifica quando il numero di bit a disposizione non è sufficiente a rappresentare il risultato.

Esempio: $n = 5, \quad \pm 9 \pm 3, \quad \pm 9 \pm 8$

intervallo di rappresentazione: da -2^4 a $2^4 - 1$ (ovvero, da -16 a 15)

riporto	00011		11001		00111		11111	
	$+9$	01001	$+9$	01001	-9	10111	-9	10111
	$+3$	00011	-3	11101	$+3$	00011	-3	11101
	$+12$	01100	$+6$	00110	-6	11010	-12	10100

In questo caso non si ha trabocco.

riporto	01000		10000	
	$+9$	01001	-9	10111
	$+8$	01000	-8	11000
	-15	10001	$+15$	01111
	(e non $+17$)		(e non -17)	

In questo caso si ha trabocco.

Si ha **trabocco** quando il riporto sul bit di segno è diverso dal riporto dal bit di segno verso l'esterno.

Differenza tra due numeri: sommo al primo il complemento del secondo

L'aritmetica reale

L'insieme dei reali (e dei razionali) è infinito $\Longrightarrow$ non è possibile rapprentarli tutti

Rappresentazione in virgola fissa

Si rappresentano separatamente, usando un numero fissato di cifre

- parte intera e
- parte frazionaria

(si usa una virgola per separare le due parti)

$$N_b = c_{n-1}\ c_{n-2}\ \cdots\ c_1\ c_0\ ,\ c_{-1}\ c_{-2}\ \cdots\ c_{-m}$$

rappresenta il numero

$$N = c_{n-1} \cdot b^{n-1} + \cdots + c_0 \cdot b^0 + c_{-1} \cdot b^{-1} + \cdots + c_{-m} \cdot b^{-m}$$

Rappresentazione in virgola mobile

Rappresentazione in **forma normalizzata in base** b

$$X = m \cdot b^e$$

- e è la **caratteristica** in base b di X: intero relativo
- m è la **mantissa** in base b di X: numero frazionario tale che $1/b \leq |m| < 1$
  Se è rappresentata dalla sequenza di cifre

  $$c_1\ c_2\ c_3\ \cdots$$

  allora rappresenta il valore

  $$c_1 \cdot b^{-1} + c_2 b^{-2} + \cdots$$

Esempio: $X = (5)_{10} = (101)_2$

$$\begin{aligned} m &= |m| = (0.101\cdots0000)_2 \\ e &= (11)_2 \end{aligned}$$

Fissati

- k bit per mantissa
- h bit per caratteristica
- 1 bit per il segno

l'**insieme di reali rappresentabili** è fissato (e limitato)

$$\begin{aligned} 1/2 &\leq |m| \leq \textstyle\sum_{i=1}^{k} 2^{-i} \\ |e| &\leq 2^{h-1} - 1 \end{aligned}$$

Questo fissa anche massimo e minimo (in valore assoluto) numero rappresentabile.

Assunzione realistica: reali rappresentati con 32 bit:

- 24 bit per la mantissa
- 7 bit per la caratteristica (in complemento)
- 1 bit per il segno della mantissa (0 positivo, 1 negativo)

Insieme F dei numeri rappresentabili in virgola mobile

- sottoinsieme finito dei numeri razionali rappresentabili (con n bit)
- simmetrico rispetto allo 0
- gli elementi **non** sono uniformemente distribuiti sull'asse reale
  - densi intorno allo 0
  - radi intorno al massimo rappresentabile
- molti razionali non appartengono ad F (ed es. $1/3$, $1/5$, ...)
- non è chiuso rispetto ad addizioni e moltiplicazioni
- per rappresentare un reale X si sceglie l'elemento di F più vicino ad X
- la funzione che associa ad un reale X l'elemento di F più vicino ad X è detta funzione di arrotondamento

Limitazioni aritmetiche

Dovute al fatto che il numero di bit usati per rappresentare un numero è limitato $\Longrightarrow$

- perdita di precisione
- **arrotondamento**: mantissa non è sufficiente a rappresentare tutte le cifre significative del numero
- **errore di overflow**: caratteristica non è sufficiente (numero troppo grande)
- **errore di underflow**: numero troppo piccolo viene rappresentato come 0

Formati standard proposti dalla IEEE (Institute of Electrical and Electronics Engineers)

- singola precisione: 32 bit
- doppia precisione: 64 bit
- quadrupla precisione: 128 bit

La programmazione nel linguaggio C

Introduzione ai programmi C

Vedremo il cosidetto **ANSI C** (standard del 1989, con piccole aggiunte del 1994)

Il primo programma C: ciao mondo

File: `base/ciao.c`

```
#include <stdio.h>
int main(void)
  /* Stampa un messaggio sullo schermo. */
{
  printf("Ciao mondo!\n");
  return 0;
}
```

Questo programma stampa sullo schermo una riga di testo:

```
Ciao mondo!
#
```

("#" denota la posizione del cursore)

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 `*/`

`int main(void)`

- è una parte presente in tutti i programmi C
- le parentesi "`(`" e "`)`" dopo main indicano che main è una **funzione**
- i programmi C contengono una o più funzioni, tra le quali ci deve essere la funzione `main`
- `main` è una **funzione speciale**, perché l'esecuzione del programma incomincia con l'esecuzione di `main`
- le funzioni C (come quelle matematiche) prendono un insieme (eventualmente vuoto) di **argomenti** e restituiscono un **valore** (oppure nulla)
  - `void` specifica che `main` non prende alcun argomento
  - `int` specifica che il valore restituito da `main` è di **tipo** intero

  Vedremo più avanti la nozione di "tipo" e i tipi del C.
- la parentesi graffa "`{`" 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** "`;`"
- vedremo più avanti che, 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)

`return 0;`

- se usato nella funzione `main` fa terminare l'esecuzione del programma
- 0 indica che il programma è terminato con successo
- vedremo più avanti qual'è l'effetto dell'istruzione `return` quando viene usata in altre funzioni

`#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`)

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 `ciao.c`

```
#include <stdio.h>
int main(void)
{
  printf("Ciao");
  printf(" mondo!\n");
  return 0;
}
```

- produce lo stesso effetto del programma precedente
- la seconda `printf` incomincia a stampare dal punto in cui aveva smesso la prima

Cosa viene stampato se usiamo

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

e se usiamo

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


Un altro semplice programma: area di un rettangolo

File: `base/arearet1.c`

```
#include <stdio.h>
int main(void)
{
  int base;    int altezza;    int area;

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

  printf("Area: %d\n", area);
  return 0;
}  /* main */
```

Quando viene eseguito stampa:

```
Area: 21
#
```

Le variabili: servono a denotare i dati all'interno dei programmi.

Una variabile è caratterizzata dalle seguenti **proprietà**:

1. **nome**: serve a identificarla — *Esempio:* `altezza`

   È un **identificatore** C: sequenza di lettere, cifre, e "_" che comincia con una lettera o con "_" (non con una cifra)
   - può avere lunghezza qualsiasi, ma solo i primi 31 caratteri sono significativi (ANSI)
   - lettere minuscole e maiuscole sono considerate distinte
2. **tipo**: specifica il tipo di dato che può memorizzare
   Esempio: `int` (può memorizzare interi)
3. **indirizzo**: della cella di memoria che contiene il dato denotato;

   Ad ogni variabile è associata una **cella di memoria** (o più celle di memoria consecutive, a seconda del tipo).
4. **valore**: dato che la variabile denota in un certo istante dell'esecuzione
   Esempio: 4

   Può cambiare durante l'esecuzione.

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

Analogia con una scatola di scarpe etichettata in uno scaffale

- nome $\Longrightarrow$ etichetta
- tipo $\Longrightarrow$ capienza (che tipo di scarpe ci metto dentro)
- indirizzo $\Longrightarrow$ posizione nello scaffale (la scatola è incollata)
- valore $\Longrightarrow$ scarpa che c'è nella scatola

N.B.

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

Ritorniamo al programma per l'area del rettangolo

```
int altezza;
```

è una **dichiarazione di variabile**

- viene creata la scatola e incollata allo scaffale
- ha **tipo** `int` $\Longrightarrow$ può contenere interi
- ha **nome** `altezza`
- 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) — la scatola viene creata piena, però con una scarpa scelta a caso

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

- come per `altezza`

Variabili **intere**

- per dichiarare variabili intere si può usare il tipo `int`
- 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`
- per immettere un reale si può usare la notazione con il punto decimale
- specificatore di formato: `%g`


```
base = 3;
```

è un'**istruzione di assegnazione**

- "`=`" è l'**operatore di assegnazione**
- il suo effetto è quello di **assegnare** il valore a destra di "`=`" (in questo caso 3) alla variabile a sinistra (in questo caso `base`)
  $\Longrightarrow$ il valore viene scritto nella cella di memoria associata alla variabile
- a questo punto la variabile `base` ha un valore significativo

```
altezza = 4;
area = base * altezza;
```

è un'istruzione di assegnazione, in cui a destra di "`=`" abbiamo un'**espressione**

- vengono presi i valori di `base` (3) e `altezza` (4) e viene calcolato il loro prodotto (12)
- tale valore viene assegnato alla variabile `area`

Nota: il C mette disposizione gli **operatori aritmetici** tra interi: `+`, `-`, `*`, `/`, ...


```
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`

```
printf("%d%d···%d", i1, i2, ..., in);
```

Area di un rettangolo di dimensioni lette da tastiera

File: `base/arearet2.c`

```
#include <stdio.h>

int main(void)
{
  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);

  return 0;
}  /* main */
```


`int base, altezza, area;`

- posso dichiarare contemporaneamente più variabili **dello stesso tipo**
- il tipo viene specificato un'unica volta
- le variabili devono essere separate con una virgola "`,`"
- la dichiarazione deve essere terminata con "`;`"

Quindi, la **sintassi** di una dichiarazione di variabili è

```
tipo variabile-1, variabile-2, ..., variabile-n;
```

In generale, la **sintassi** di un linguaggio di programmazione specifica le regole per la scrittura corretta di un programma e delle sue parti.

In informatica vengono usati dei formalismi appositi per specificare la sintassi dei linguaggi di programmazione. Noi lo faremo attraverso esempi significativi.

La **semantica** di un linguaggio di programmazione specifica invece qual'è il significato dei diversi costrutti di un programma.

`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`
  - la funzione `scanf` utilizza tale locazione per metterci il valore letto da tastiera
- 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 e
  2. l'intero ottenuto viene assegnato alla variabile `base` (viene cioè scritto nella cella di memoria il cui indirizzo è stato passato a `scanf`)
- N.B. il precedente valore della variabile `base` va perduto

Cosa appare **sullo schermo**:

```
Immetti base del rettangolo e premi INVIO
# => 5^
Immetti altezza del rettangolo e premi INVIO
# => 4^
Area: 20
#
```

(quello che segue "=>" è l'input dell'utente
"^" denota la pressione del tasto *Invio*)

Esercizio: Scrivere un programma che legge da tastiera un valore (reale) in Euro e stampa il controvalore in Lire.

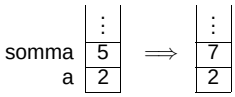
Osservazioni sull'istruzione di assegnazione

Nell'istruzione `x = e` viene

1. prima valutato il valore dell'espressione `e` a destra di "=" (usando i valori correnti delle variabili);
2. poi tale valore viene assegnato alla variabile `x` a sinistra di "=".

Esempio:

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



Esempio:

```
int a, b;

a = 2;
b = 3;
a = b;
a = a + b;
b = a + b;
```

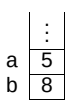
a	b
?	?
2	?
2	3
3	3
6	3
6	9

A sinistra di "=" ci deve essere una **variabile** (ci vuole una locazione di memoria in cui scrivere il valore).

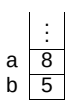
Esempio: Quali istruzioni sono corrette e quali no?

`a = a;`	SI corretta, ma ha effetto nullo
`a = 2 * a;`	SI corretta
`5 = a;`	NO, quale è la locazione di memoria
`5 = 6;`	NO, quale è la locazione di memoria nella quale scrivere il valore di `a`
`a + b = c;`	NO, `a+b` non è una variabile

Esempio: **Scambio del valore** di due variabili: prima:



dopo:



Il seguente codice non funziona (si perde il valore di `a`): `a = b; b = a;`

$\Longrightarrow$ prima di eseguire `a = b` bisogna copiare il valore di `a` in una **variabile temporanea** per poterla poi assegnare a `b`:

```
temp = a;
a = b;
b = temp;
```

a	b	temp
5	8	?
5	8	5
8	8	5
8	5	5

Gli operatori aritmetici del C

- in ordine di priorità:
  - `-` unario — priorità alta
  - `*` (moltiplicazione), `/` (divisione), `%` (modulo)
  - `+` (somma), `-` (sottrazione) — priorità bassa
- la **moltiplicazione** "`*`" va denotata esplicitamente
- "`/`" tra due interi indica la **divisione intera**

  Esempio: $24/6 = 4$ $\quad -24/6 = -4$
  $25/6 = 4$ $\quad -25/6 = -4$
- "`%`" può essere usato solo con operandi interi
- le espressioni vengono valutate **da sinistra a destra** tenendo conto delle **priorità degli operatori** (esattamente come in algebra)

  Esempio: `2 + 3 * 4` vale 14 (e non 20)

  Si possono sempre usare le **parentesi** per imporre un certo ordine di valutazione.

  Esempio: `(2 + 3) * 4` vale 20

Esercizio: Scrivere un programma che legge due interi e stampa quoziente e resto della divisione.

Es.: con 25 e 6 deve stampare "`25:6 = 4 con resto di 1`"

Esercizio: Leggere i coefficienti a, b, c di un'equazione di secondo grado

$$a \cdot x^2 + b \cdot x + c = 0$$

che si suppone essere a discriminante nonnegativo (ovvero vale che $b^2 - 4 \cdot a \cdot c \geq 0$) e calcolare gli zeri dell'equazione.

Istruzioni condizionali

- finora abbiamo visto come modificare il valore di variabili
- vediamo ora come **verificare il valore di variabili** e **compiere azioni diverse** a seconda del risultato della verifica

Istruzione `if-else`

Sintassi:

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

dove

- *espressione* è un'**espressione condizionale** il cui valore può essere vero o falso
- *istruzione1* rappresenta il **ramo then** (deve essere un'unica istruzione)
- *istruzione1* rappresenta il **ramo else** (deve essere un'unica istruzione)

Semantica: (ovvero qual'è il significato)

1. viene prima valutata *espressione*
2. se *espressione* è vera viene eseguita *istruzione1*
   altrimenti (ovvero se *espressione* è falsa) viene eseguita *istruzione2*
3. l'esecuzione procede con l'istruzione successiva all'istruzione `if-else`

Operatori relazionali del C

Tipicamente *espressione* coinvolge il **confronto** di due valori tramite gli **operatori relazionali**:

- `<`, `>`, `<=`, `>=` (minore, maggiore, minore o uguale, maggiore o uguale) — priorità alta
- `==`, `!=` (uguale, diverso) — priorità bassa

Esempio:

```
temperatura <= 0
velocita > velocita_max
voto == 30
anno != 2000
```

In C non esiste un tipo Booleano (unici valori: vero, falso) $\Longrightarrow$ 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)
`3 != 3`	ha valore 0 (ossia falso)

Attenzione: non confondere "`=`" con "`==`"

Esempio di istruzione `if-else`

```
int temperatura;

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

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

```
# => 32^
Fa caldo
Ciao
#
```

Esercizio: Leggere due interi e stampare il maggiore dei due.

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* e si procede con l'istruzione successiva
3. altrimenti si procede direttamente con l'istruzione successiva

Esempio di istruzione `if`

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


Blocco di istruzioni

La sintassi di `if-else` ci permette soltanto di avere un'unica istruzione nel ramo then (o nel ramo else).

Se nel ramo then (o nel ramo else) vogliamo eseguire più istruzioni dobbiamo usare un **blocco di istruzioni**.

Sintassi:

```
{
   dichiarazioni-di-variabili
   istruzione-1
   ...
   istruzione-n
}
```

Le variabili dichiarate all'interno di un blocco vengono dette **locali** al blocco.

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

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

if (mese == 12) {
  mesesucc = 1;
  annosucc = succ + 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`.

If annidati con condizioni mutuamente escludentisi

Esempio: Leggere una temperatura (intero) e stampare un messaggio secondo la seguente tabella:

temperatura t	messaggio
$30 < t$	molto caldo
$20 < t \leq 30$	caldo
$10 < t \leq 20$	gradevole
$t \leq 10$	freddo

File: `ifelse/temperat.c`

Osservazioni:

- si tratta di un'unica istruzione `if-else`
- non serve che la seconda condizione sia composta `(t <= 30)` e `(t > 20)`
- `else` si riferisce all'`if` più vicino

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");`

Ambiguità si risolve considerando che un `else` fa sempre riferimento all'`if` piu' vicino:

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

Perché un `else` si riferisca ad un `if` precedente, devo inserire l'ultimo `if` 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

Implementazione: file `ifelse/triang.c`

Esercizio: Si risolva 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
then è scaleno
else if le coppie uguali sono 1
then è isoscele
else è equilatero

Soluzione: file `ifelse/triang2.c`

Esercizio: Leggere i coefficienti a, b, c e calcolare gli zeri dell'equazione quadratica

$$a \cdot x^2 + b \cdot x + c = 0$$

A seconda del segno del discriminante $b^2 - 4 \cdot a \cdot c$ stampare le due soluzioni reali distinte, la soluzione reale doppia, o le due soluzioni complesse coniugate.

Soluzione: file `ifelse/equaquad.c`

Operatori logici (o booleani)

Permettono di combinare più condizioni ottenendo condizioni complesse.

In ordine di priorità:

- `!` (**not** logico) — priorità alta
- `&&` (**and** logico)
- `||` (**or** logico) — 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)` risulta vero (pari a 1) se `a` è compresa tra 10 e 20
`(b <= -5) || (b >= 5)` risulta vero se il valore assoluto di `b` è ≥ 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
((a+2) == (3*b)) || ((!trovato) && (c < (a/3)))
```

è equivalente a

Esercizio: Determinare il tipo di un triangolo usando condizioni composte.

Soluzione: file `ifelse/triang3.c`

Istruzione `switch`

Può essere usate 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 prima valutata *espressione*
2. viene cercato il primo *i* per cui il valore di *espressione* è pari a *valore-i*
3. se si è trovato tale *i*, allora vengono eseguite *istruzioni-i*
   altrimenti vengono eseguite *istruzioni-default*
4. l'esecuzione procede con l'istruzione successiva all'istruzione `switch`

Se abbiamo più valori per cui eseguire le stesse istruzioni, si possono raggruppare omettendo `break`:

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

Esempio: calcolo dei giorni del mese
Implementazione: file `ifelse/giormese.c`

Esercizio: Calcolo della data del giorno successivo, tenendo conto anche degli anni bisestili.
Soluzione: file `ifelse/datasucc.c`

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 intere** (ovvero interi noti a tempo di compilazione). In particolare, non possono essere espressioni che fanno riferimento a variabili.

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*/
}
```

$\Longrightarrow$ L'utilità dell'istruzione `switch` è **limitata**.

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 è trovato tale *i*, allora vengono eseguite 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*
4. l'esecuzione procede con l'istruzione successiva all'istruzione `switch`

Esempio: più `case` di uno `switch` eseguiti in sequenza

```
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("\nImmetti un valore <= 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. $\Longrightarrow$

È buona norma mettere `break` come ultima istruzione di ogni `case`.

Istruzioni iterative (o cicliche)

Esempio: Leggi 5 interi, calcolane la somma e stampala.

Variante non accettabile: 5 variabili, 5 istruzioni di lettura, 5 . . .

```
int i1, i2, i3, i4, i5;
scanf("%d", &i1):
...
scanf("%d", &i5);
printf("%d", i1 + i2 + i3 + i4 + i5);
```

Variante migliore che utilizza solo 2 variabili:

```
int somma, i;
somma = 0;
scanf("%d", &i);
somma = somma + i;
...                    /* per 5 volte */
scanf("%d", &i);
somma = somma + i;
printf("%d", somma);
```

$\Longrightarrow$ conviene però usare un'**istruzione iterativa**

Le **istruzioni iterative** permettono di ripetere determinate azioni più volte:

- un numero di volte fissato $\Longrightarrow$ **iterazione (o ciclo) definita**

  Esempio:

  for 10 volte
  do fai un giro del parco di corsa

- finchè una condizione rimane vera $\Longrightarrow$ **iterazione (o ciclo) indefinita**

  Esempio:

  while non sei ancora sazio
  do prendi una ciliegia dal piatto
  mangiala

Istruzione `while`

Permette di realizzare l'iterazione in C.

Sintassi:

```
while (espressione)
  istruzione
```

dove

- *espressione* viene detta **condizione** del ciclo
- *istruzione* viene detta **corpo** del ciclo

Semantica:

- viene valutata l'*espressione*
- se è vera si esegue *istruzione* e si torna a valutare *espressione* procedendo così fino a quando *espressione* diventa falsa
- a questo punto si passa all'istruzione successiva

Nota: se *espressione* è già falsa all'inizio, *istruzione* non viene eseguita per niente

Iterazione definita

Esempio: Stampa 100 asterischi.

Si utilizza un **contatore** per contare il numero di asterischi stampati.

algoritmo stampa di 100 asterischi
inizializza il contatore a 0
while il contatore è minore di 100
do stampa un "*"
incrementa il contatore di 1

Implementazione:

```
int i;
i = 0;
while (i < 100) {
  printf("*");
  i = i + 1;
}
```

$\Longrightarrow$ si parla anche di **ciclo controllato da contatore**

Il contatore viene detto **variabile di controllo** del ciclo.

Esercizio: Leggere 10 interi, calcolarne la somma e stamparla.

algoritmo somma di 10 numeri letti da tastiera
  inizializza la somma a 0
  for 10 volte
  do leggi un intero
    incrementa la somma dell'intero letto

Si utilizza un contatore per contare il numero di interi letti.

Soluzione: file `cicli/somma.c`

Esempio: Leggi 10 interi **positivi** e stampane il massimo.

Si utilizza un **massimo corrente** con il quale si confronta ciascun numero letto.

algoritmo massimo di 10 interi positivi
  inizializza il massimo a 0
  for 10 volte
  do leggi un intero
    if l'intero letto è maggiore del massimo
    then aggiorna il massimo all'intero letto
  stampa il massimo

Implementazione: file `cicli/massimo.c`

Esercizio: Leggere 10 interi (qualunque) e stamparne il massimo.
Soluzione: file `cicli/massimoi.c`

Il limite di conteggio può anche essere letto da tastiera.

Esercizio: Leggere un intero N e stampare i primi N numeri pari.
Soluzione: file `cicli/pari.c`

Operatori di incremento, decremento e assegnazione

Operazioni del tipo `i = i + 1` / `i = i - 1` sono molto comuni. $\Longrightarrow$

- operatore di **incremento**: `++`
- operatore di **decremento**: `--`

In realtà `++` corrisponde a due operatori:

- **postincremento**: `i++`
  - valore dell'espressione è il valore di `i`
  - side-effect: incrementa `i` di 1
- **preincremento**: `++i`
  - valore dell'espressione è il valore di `i+1`
  - side-effect: incrementa `i` di 1

(analogamente per `i--` e `--i`)

Per **side-effect** si intende la modifica del contenuto di una locazione di memoria.

È l'operazione di base nei linguaggi imperativi, nei quali il concetto fondamentale è lo **stato** del programma (dato dal contenuto di tutte le locazioni di memoria).

*Operazione di **assegnazione***

`x = y` è un'espressione

- valore dell'espressione è il valore di `y` (che è un'espressione)
- **side-effect**: assegna alla variabile `x` il valore di `y`

L'operatore "=" è **associativo a destra**.

Esempio: Qual'è il significato di `x = y = 4` ?

È equivalente a: `x = (y = 4)`

- `y = 4` ... espressione di valore 4 con side-effect su `y`
- `x = (y = 4)` ... espressione di valore 4 con ulteriore side-effect su `x`

Le seguenti **espressioni** sono equivalenti:

```
i = i + 1
++i
```

(valore dell'espressione è `i+1`,
come side-effect incrementa `i` di 1)

Le seguenti **istruzioni** sono equivalenti:

```
i = i + 1;
i++;
++i;
```


Nota sull'uso degli operatori di incremento e decremento

Esempio:

	Istruzione	x	y	z
1	`int x, y, z;`	?	?	?
2	`x = 4;`	4	?	?
3	`y = 2;`	4	2	?
4a	`z = (x + 1) + y;`	4	2	7
4b	`z = (x++) + y;`	5	2	6
4c	`z = (++x) + y;`	5	2	7

N.B.: **Non usare mai così!**
In un'istruzione di assegnazione non ci devono essere altri side-effect (oltre a quello dell'operatore di assegnazione) !!!

Riscrivere così: 4b: `z = (x++) + y;` $\Longrightarrow$

```
z = x + y;
x++;
```

4c: `z = (++x) + y;` $\Longrightarrow$

```
x++;
z = x + y;
```


Ordine di valutazione degli operandi

In generale il C **non** stabilisce qual'è l'ordine di valutazione degli operandi nelle espressioni.

Esempio:

```
int x, y, z;
x = 2;
y = 4;
z = x++ + (x * y);
```

Qual'è il valore di `z`?

- se viene valutato prima `x++`: $2 + (3 * 4) = 14$
- se viene valutato prima `x*y`: $(2 * 4) + 2 = 10$

$\Longrightarrow$ Se una variabile compare più volte e inoltre le viene applicato un operatore di incremento (decremento) allora il **risultato dell'espressione è indeterminato**.

Forme abbreviate dell'assegnazione

```
a = a + b;   ⟹   a += b;
a = a - b;   ⟹   a -= b;
a = a * b;   ⟹   a *= b;
a = a / b;   ⟹   a /= b;
a = a % b;   ⟹   a %= b;
```

Inizializzazione di variabili

L'assegnazione di un valore iniziale ad una variabile può essere effettuata contestualmente alla sua dichiarazione $\Longrightarrow$ **inizializzazione di variabile**

Esempio:

```
int a, b = 5, c;
float pi = 3.14152, x;
```

`b` e `pi` sono inizializzate, `a`, `c` e `x` non lo sono

Istruzione `for`

I seguenti elementi sono comuni ai cicli controllati da contatore:

- variabile di controllo (contatore)
- inizializzazione della variabile di controllo
- incremento (decremento) della variabile di controllo ad ogni iterazione
- verifica se si è raggiunto il valore finale della variabile di controllo

Esempio: Stampa i numeri da 1 a 100.

```
int i;                    /* 1 */
i = 1;                    /* 2 */
while (i <= 100) {        /* 4 */
  printf("%d", i);
  i++;                    /* 3 */
}
```

L'istruzione `for` permette di gestire automaticamente questi aspetti:

```
int i;
for (i = 1; i <= 100; i++)
  printf("%d", i);
```


Sintassi:

```
for (espr-1; espr-2; espr-3)
  istruzione
```

dove

- *`espr-1`* serve a inizializzare la variabile di controllo
- *`espr-2`* è la verifica di fine ciclo
- *`espr-3`* serve a incrementare la variabile di controllo
- *`istruzione`* è il corpo del ciclo

Semantica: l'istruzione `for` di sopra è equivalente a

```
espr-1;
while (espr-2) {
  istruzione
  espr-3;
}
```

(c'è un'eccezione che riguarda l'istruzione `continue`, che però noi non vediamo)

Esempi:

```
for (i = 1; i <= 10; i++)
for (i = 10; i >= 1; i--)
for (i = -4; i <= 4; i += 2)
for (i = 0; i >= -10; i -= 3)
```

$\Longrightarrow$ `i`: $1, 2, 3, \ldots, 10$

$\Longrightarrow$ `i`: $10, 9, 8, \ldots, 2, 1$

$\Longrightarrow$ `i`: $-4, -2, 0, 2, 4$

$\Longrightarrow$ `i`: $0, -3, -6, -9$

La sintassi del `for` permette che le *`espr-i`* siano delle espressioni qualsiasi.

Buona norma:

- usare ciascuna *`espr-i`* in base al significato descritto prima
- non modificare la variabile di controllo nel corpo del ciclo

Ciascuna delle tre *`espr-i`* puo' anche mancare:

- i "`;`" vanno messi lo stesso
- se manca *`espr-2`* viene assunto il valore vero
- se manca una delle tre *`espr-i`* è meglio usare un'istruzione `while`

Esercizio: Riscrivere tutti i programmi con ciclo visti finora utilizzando l'istruzione `for`.

Esercizio: Scrivere un programma che legge un intero N e calcola e stampa il fattoriale di N.

Soluzione: file `cicli/fattiter.c`

Esercizio: Scrivere un programma che legge un intero N ed una sequenza di interi di lunghezza N, e stampa la somma dei positivi e la somma dei negativi nella sequenza.

Soluzione: file `cicli/sompone.c` (senza `for`) e file `cicli/sompone2.c` (con `for`)

Esercizio: Il valore di π può essere calcolato con la serie

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Scrivere un programma che legge un intero N e calcola il valore di π approssimato ai primi N termini della serie.

Iterazione indefinita

In alcuni casi il numero di iterazioni da effettuare non è noto prima di iniziare il ciclo, perché dipende dal verificarsi di una **condizione**.

Esempio: Leggi interi e sommali, fermandoti quando leggi 0.

```
int i, somma = 0;

scanf("%d", &i);
while (i != 0) {
  somma = somma + i;
  scanf("%d", &i);
}
printf("%d", somma);
```

0 gioca il ruolo di **sentinella** $\Longrightarrow$ si parla anche di **ciclo controllato da sentinella**

N.B. la sentinella **non deve essere compresa** tra i dati di ingresso

Esercizio: Leggere una sequenza di interi terminata da 0 e stamparne la lunghezza.

Soluzione: file `cicli/lung1.c`

Istruzione `do-while`

Nell'istruzione `while` la condizione di fine ciclo viene controllata all'inizio di ogni iterazione.

L'istruzione `do-while` è simile all'istruzione `while`, ma la **condizione viene controllata alla fine di ogni iterazione**.

Sintassi:

```
do
  istruzione
while (espressione);
```

Semantica: è equivalente a

```
istruzione
while (espressione) {
  istruzione
}
```


Note sulla sintassi di `do-while`:

- non serve racchiudere il corpo del ciclo tra "`{`" e "`}`"
- c'è un "`;`" dopo "`while espressione`"
- per evitare di confondere "`while espressione;`" con un'istruzione `while` con corpo vuoto conviene scrivere l'istruzione `do-while` in ogni caso come

```
do {
  istruzione
} while (espressione);
```

Esempio: Lunghezza di una sequenza di interi terminata da 0, usando `do-while`.

Implementazione: file `cicli/lung2.c`

Esercizio: Leggere una sequenza di interi terminata da 0 e stampare la somma dei positivi e la somma dei negativi nella sequenza.

Soluzione: file `cicli/sompone3.c`

Esempio: Leggere due interi positivi e calcolare il **massimo comun divisore**.

Es.: $MCD(12, 8) = 4$
$MCD(12, 6) = 6$
$MCD(12, 7) = 1$

1) Sfruttando direttamente la definizione di MCD

- osservazione: $1 \leq MCD(m, n) \leq min(m, n)$
  $\Longrightarrow$ si provano i numeri compresi tra 1 e $min(m, n)$
- conviene iniziare da $min(m, n)$ e scendere verso 1

algoritmo stampa massimo comun divisore di due interi positivi letti da tastiera
  leggi *m* ed *n*
  inizializza *mcd* al minimo tra *m* ed *n*
  while *mcd* > 1 e non si è trovato un divisore comune
  do if *mcd* divide sia *m* che *n*
    then si è trovato un divisore comune
    else decrementa *mcd* di 1
  stampa *mcd*

Osservazioni:

- il ciclo termina sempre perché ad ogni iterazione
  - o si è trovato un divisore
  - o si decrementa *mcd* di 1 (al più si arriva ad 1)
- per verificare se si è trovato il MCD si utilizza una variabile booleana (usata nella condizione del ciclo)

Implementazione: file `cicli/mcd1.c`

Quante volte viene eseguito il ciclo?

- caso migliore: 1 volta (quando m divide n o viceversa)
  p.es. $MCD(500, 1000)$
- caso peggiore: $min(m, n)$ volte (quando $MCD(m, n) = 1$)
  p.es. $MCD(500, 1001)$

$\Longrightarrow$ algoritmo si comporta male se m e n sono grandi e $MCD(m, n)$ è piccolo

2) Metodo di Euclide per il calcolo del massimo comun divisore

Permette di ridursi più velocemente a numeri più piccoli, sfruttando la seguente proprietà:

$$MCD(m, n) = \begin{cases} m \quad (\text{o } n), & \text{se } m = n \\ MCD(m - n, n), & \text{se } m > n \\ MCD(m, n - m), & \text{se } m < n \end{cases}$$

Dimostrazione: per **esercizio**
(mostrando che i divisore comuni di m ed n, con $m > n$, sono anche divisori di $m - n$)

Es.: $MCD(12, 8) = MCD(12 - 8, 8) = \ MCD(4, 8 - 4) = 4$

Come si ottiene un algoritmo?
Si applica ripetutamente il procedimento fino a che non si ottiene che $m = n$.

Es.:

m	n	$maggiore - minore$
210	63	147
147	63	84
84	63	21
21	63	42
21	42	21
21	21	$\Longrightarrow MCD(21, 21) = MCD(21, 42) = \cdots = MCD(210, 63)$

algoritmo di Euclide per il MCD di due interi positivi
leggi *m* ed *n*
while $m \neq n$
do sostituisci il maggiore tra *m* ed *n* con la differenza tra il maggiore ed il minore
stampa *m* (oppure *n*)

Implementazione: file `cicli/mcd2.c`

Cosa succede se $m = n = 0$? $\Longrightarrow$ il risultato è 0

E se $m = 0$ e $n \neq 0$ (o viceversa)? $\Longrightarrow$ si entra in **un ciclo infinito**

Se si vuole tenere conto del fatto che l'utente possa immettere una qualsiasi coppia di interi, è necessario inserire una verifica sui dati in ingresso.

Usiamo un ciclo di lettura e verifica dei dati in ingresso (fa uso di `do-while`)
Implementazione: file `cicli/mcd3.c`

3) Metodo di Euclide con i resti per il calcolo del massimo comun divisore

Cosa succede se $m >> n$?

Es.: $MCD(1000, 2)$

1000	2
998	2
996	2
...	
2	2

$MCD(1001, 500)$

1001	500
501	500
1	500
...	
1	1

Come possiamo comprimere questa lunga sequenza di sottrazioni?

Quello che in fondo si calcola è il resto della divisione intera. $\Longrightarrow$

Metodo di Euclide: sia $m = n \cdot k + r$ (con $0 \leq r < m$)

$$MCD(m, n) = \begin{cases} n, & \text{se } r = 0 \quad \text{(ovvero, } m \text{ è multiplo di } n\text{)} \\ MCD(r, n), & \text{se } r \neq 0 \end{cases}$$

algoritmo di Euclide con i resti per il calcolo del MCD
  leggi *m* ed *n*
  while *m* ed *n* sono entrambi $\neq 0$
  do sostituisci il maggiore tra *m* ed *n* con
    il resto della divisione del maggiore per il minore
  stampa il numero tra i due che è diverso da 0

Implementazione: per **esercizio**: file `cicli/mcd4.c`

Esempio: Leggere da input una sequenza di 0 e 1 (separati da spazi), terminata da 2, e calcolare la lunghezza della più lunga sottosequenza di soli 0.

Es.: $0\ 0\ 1\ \underbrace{0\ 0\ 0}\ 1\ 1\ 1\ 1\ 0\ 0\ 2$ $\Longrightarrow$ stampa 3

Variabili utilizzate:
- *bit* ... valore letto
- *cont* ... lunghezza sequenza corrente
- *maxlung* ... lunghezza massima sequenza di soli 0 (temporanea)

algoritmo lunghezza massima sottosequenza di soli 0
  inizializza *cont* e *maxlung* a 0
  do leggi un *bit*
    if *bit* è uguale a 0
    then incrementa *cont* di 1
      if *cont* $>$ *maxlung*
      then poni *maxlung* pari a *cont* (oppure: incrementa *maxlung* di 1)
    else poni *cont* pari a 0
  while *bit* è diverso da 2
  stampa *maxlung*

Implementazione: file `cicli/sequenz1.c`

Esercizio: Migliorare l'algoritmo (e l'implementazione) in modo da aggiornare *maxlung* solo al termine di una nuova sequenza di 0 di lunghezza maggiore delle precedenti.

Soluzione: file `cicli/sequenz2.c`

Cicli annidati

Il corpo di un ciclo può contenere a sua volta un ciclo.

Esempio: Stampa della tavola pitagorica.

algoritmo stampa della tavola pitagorica
  for ogni *riga* tra 1 e 10
  do for ogni *colonna* tra 1 e 10
    do stampa *riga* $*$ *colonna*
    stampa un a capo

Implementazione: file `cicli/pitagor1.c`

```
int riga, colonna;
int Nmax = 10;         /* indica il numero di righe e di colonne */

for (riga = 1; riga <= Nmax; riga++) {
  for (colonna = 1; colonna <= Nmax; colonna++)
    printf("%d ", riga * colonna);
  printf("\n");
}
```


Direttiva di compilazione `#define`

Nel programma precedente, `Nmax` non viene mai modificato (è uguale a 10), tuttavia abbiamo allocato una variabile.

Si può evitare?

- usiamo esplicitamente 10 nel programma? **NO!**
  - in un programma complesso non sappiamo più quale'è il significato di 10 (**magic number**)
  - se vogliamo modificare il valore 10 (ad es. in 15) dobbiamo farlo in molti punti del programma
- definiamo un **identificatore costante**

  `#define Nmax 10`
  - `#define` è una **direttiva di compilazione**
  - dice al compilatore di sostituire ogni occorrenza di `Nmax` con 10 prima di compilare il programma

Output formattato

`printf("x%4dy", 10);` stampa 10 su un campo di ampiezza 4, allineato a destra

```
x  10y#
```

`printf("x%-4dy", 10);` stampa 10 su un campo di ampiezza 4, allineato a sinistra

```
x10  y#
```

Esercizio: Stampare la tavola pitagorica.

1. aggiungendo la riga e la colonna di intestazione della tabella, e
2. usando output formattato per allineare le colonne

Soluzione: file `cicli/pitagor2.c`

`printf("x%6.3gy", 1.238);` stampa 1.238 su un campo di ampiezza 6, arrotondato a 3 cifre significative, allineato a destra

```
x  1.24y#
```

Il numero di iterazioni del ciclo più interno può dipendere dall'iterazione del ciclo più esterno.

Esempio: Stampa una piramide di asterischi di altezza letta in input.

Es.: con *altezza* 4:

```
   *
  ***
 *****
*******
```

riga	blank	*
1	3	1
2	2	3
3	1	5
4	0	7

$\Longrightarrow$ stampa: $(\mathit{altezza} - \mathit{riga})$ blank $(2 \cdot \mathit{riga} - 1)$ asterischi

algoritmo stampa piramide di asterischi
  leggi *altezza*
  for *riga* che va da 1 ad *altezza*
  do stampa $(\mathit{altezza} - \mathit{riga})$ spazi bianchi
    stampa $(2 \cdot \mathit{riga} - 1)$ asterischi
    vai a capo

Implementazione: file `cicli/piramid1.c`

Esercizio: Scrivere un programma che stampa una piramide di numeri (di altezza ≤ 9).

Es.: con altezza 4:

```
   1
  121
 12321
1234321
```

Soluzione: file `cicli/piramid2.c`

Esercizio: Scrivere un programma che legge un intero N e stampa il fattoriale di tutti i numeri compresi tra 1 ed N.

Soluzione: file `cicli/fatttab.c`

Istruzione `break`

Abbiamo visto che l'istruzione `break` permette di uscire da una istruzione `switch`.

In generale, `break` permette di uscire prematuramente da un'istruzione `switch`, `while`, `for` o `do-while`.

Esempio:

```
float a;
int i;

for (i = 0; i < 10; i++) {
  scanf("%g", &a);
  if (a >= 0.0)
    printf("%g\n", sqrt(a));
  else {
    printf("Errore\n");
    break;
  }
}
```

N.B. L'esecuzione di un `break` fa uscire di un solo livello.

`break` altera il flusso di controllo. $\Longrightarrow$ Quando viene usata nei cicli:

- si perde la strutturazione del programma
- si guadagna in efficienza rispetto ad implementare lo stesso comportamento in modo strutturato

Esempio: Codice precedente senza `break`:

```
float a;
int i;
int errore = 0;

for (i = 0; (i < 10) && !errore; i++) {
  scanf("%g", &a);
  if (a >= 0.0)
    printf("%g\n", sqrt(a));
  else {
    printf("Errore\n");
    errore = 1;
  }
}
```


Funzioni

Modularizzazione

Quando il progetto diviene complesso allora, per poter essere gestito, è necessario che venga **modularizzato**:

- il progetto viene strutturato in **parti separate**
- si stabiliscono **relazioni precise** tra le parti

Qualità di una modularizzazione

- **livello di dettaglio** dei sottoproblemi deve scaturire da scelte di progetto
- ogni sottoproblema deve essere **ben caratterizzabile** e risolvibile in modo indipendente
- le soluzioni dei sottoproblemi devono essere **combinabili** in modo semplice

Una buona modularizzazione si ottiene utilizzando il concetto di **astrazione**:

- ci si focalizza sugli aspetti essenziali del problema
- si ignorano aspetti non rilevanti rispetto all'obiettivo

Tipi di astrazione

Astrazione sui dati: attraverso uso di **tipi di dato astratti** (li vediamo più avanti)

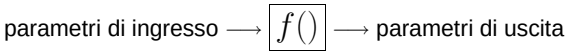
- collezioni di oggetti singoli
- operazioni con le quali operare su questi oggetti

Astrazione funzionale: ci si concentra sul "cosa" e non sul "come"

Noi trattiamo soprattutto la **modularizzazione per astrazione funzionale**

- supportata dai linguaggi imperativi tradizionali (C, Pascal, Fortran)
- realizzata in C attraverso la nozione di **funzione**

Una funzione può essere vista come una **scatola nera**:



- una funzione risolve un sottoproblema specifico
- attraverso i parametri la funzione scambia informazioni con altre funzioni

Le funzioni C

Esempio: Progettare un'interfaccia utente per la stampa di figure geometriche, in cui l'utente può scegliere:

1. la forma della figura $\Longrightarrow$ una funzione per ogni figura
2. la dimensione
3. il carattere di riempimento
4. di quanto spostare a destra la figura

Il programma può essere realizzato a diversi **livelli di generalità**, legati ad un'**astrazione crescente** del concetto di figura.

A questo corrisponde un livello crescente di **parametrizzazione** delle funzioni di stampa delle figure:

- livello (1) non è parametrico
- livello (2) è parametrico rispetto alla dimensione
- livello (3) è parametrico anche rispetto al carattere
- livello (4) è parametrico anche rispetto allo spostamento

Consideriamo prima solo il livello (1)

algoritmo stampa di figure a livello (1)
  do stampa un messaggio
    leggi un carattere
    switch carattere letto
      case 't' : stampa un triangolo
      case 'q' : stampa un quadrato
      case 'f' : stampa un saluto
  while il carattere letto è diverso da 'f'

Caratteri in C: si utilizza il tipo primitivo `char`

- ogni carattere è rappresentato dal suo codice
- i caratteri possone essere **usati come gli interi** (un carattere coincide con il codice che lo rappresenta)
- nei programmi C, un carattere viene racchiuso tra una coppia di apici singoli
  Es.: `'A'`, `'X'`, `'b'`, `'3'`, `'0'`, `';'`, `' '`
- per l'input/output di un carattere si usa lo specificatore di formato "%c"

Implementazione: stampa messaggio invece della figura: file `funzioni/figure0.c`

*Sintassi della **definizione di funzione***

intestazione blocco

dove

- *blocco* costituisce il **corpo della funzione**
- *intestazione* costituisce l'**intestazione della funzione** ed ha la seguente forma:

  identificatore-tipo identificatore (lista-parametri-formali)

  - *identificatore-tipo* specifica il **tipo del valore di ritorno**, ovvero il tipo del risultato restituito alla funzione chiamante (se manca viene assunto `int`)
  - *identificatore* specifica il **nome** della funzione ed è un qualsiasi identificatore C valido
  - *lista-parametri-formali* serve a passare informazioni dalla funzione chiamante a quella chiamata e viceversa:
    - è una lista di dichiarazioni di parametri (tipo e nome) separate da virgola
    - ogni parametro è una **variabile**
    - la lista di parametri può essere vuota

Esempi di intestazioni di funzione

```
char LeggiCarattereNonSpazio() { ... }
int MassimoComunDivisore(int a, int b) { ... }
double Potenza(double x, double y) { ... }
```

N.B. **Non** ci deve essere un "`;`" tra l'intestazione ed il corpo.

Se si omette il tipo di un parametro viene assunto per default il tipo `int`.

Attenzione: `double Potenza(double a, b) { ... }`
equivale a `double Potenza(double a, int b) { ... }`
e non a `double Potenza(double a, double b) { ... }`

N.B. **Non** si possono definire funzioni all'interno di altre funzioni. Quindi tutte le funzioni sono definite allo stesso livello.

*Sintassi della **attivazione di funzione** (detta anche invocazione o chiamata)*

`identificatore (lista-parametri-attuali)`

- *`identificatore`* specifica il nome della funzione
- *`lista-parametri-attuali`* è una lista di **espressioni** separate da virgola
- i parametri attuali devono corrispondere in numero e tipo ai parametri formali

Semantica di una attivazione di funzione B da una funzione A

- una attivazione di funzione è un'espressione
- viene sospesa l'esecuzione di A e si passa ad eseguire le istruzioni di B (a partire dalla prima)
- quando termina l'esecuzione di B, prosegue l'esecuzione di A dal punto in cui B era stata attivata

N.B. La definizione di una funzione non comporta la sua attivazione.

Prima di poter essere usata (ovvero attivata) una funzione deve essere stata **definita** (o **dichiarata** — vediamo più avanti cosa vuol dire dichiarare una funzione).

Variabili locali

Il **corpo della funzione** è un blocco $\Longrightarrow$ può contenere dichiarazioni di variabili:

- sono **locali** alla funzione (nozione a compile-time)
- hanno **tempo di vita** limitato alla durata dell'attivazione (nozione a run-time)

Regole di visibilità degli identificatori

- un identificatore dichiarato nel corpo di una funzione è detto **locale** alla funzione e **non è visibile all'esterno** della funzione ma solo nel corpo
- in realtà vale una regola più generale: un identificatore dichiarato in un blocco B è visibile
  - nel blocco B (ovvero fino a "`}`")
  - e in tutti i blocchi interni a B, a meno che non venga ridichiarato.
- un indentificatore dichiarato fuori da qualsiasi blocco è visibile nel file

N.B. La **visibilità** di un identificatore è un concetto rilevante a **compile time**.

Esempio: file `funzioni/scope.c`

Tempo di vita (o esistenza) di una variabile

Le variabili locali (ovvero le locazioni di memoria associate) vengono
- create al momento dell'attivazione di una funzione
- distruttre al momento dell'uscita dall'attivazione

Segue che:
- la funzione chiamante non può fare riferimento ad una variabile locale alla funzione chiamata
- ad attivazioni successive corrispondono variabili (locazioni di memoria) diverse

N.B. Il **tempo di vita** di una variabile è un concetto rilevante a **run time**.

Esempio: Stampa figure a livello (1): file `funzioni/figure1.c`

Esercizio: Scrivere un programma che stampa un rettangolo di asterischi a larghezza fissa e altezza variabile, utilizzando una funzione per la stampa di una riga di asterischi a lunghezza fissa.

Soluzione: file `funzioni/rettang1.c` (senza variabili locali) e `funzioni/rettang2.c` (con variabili locali)

Parametri

Esempio: Progettare un'interfaccia per la stampa di figure a livello (2) (utente può specificare la dimensione).

algoritmo stampa di figure a livello (2), (3) e (4)
  do stampa un messaggio
    leggi un carattere
    if il carattere letto è $\neq$ 'f'
    then acquisisci ulteriori informazioni
      (ad es. per (3), leggi la dimensione e il carattere di riempimento)
      switch carattere letto
        case 't' : stampa un triangolo usando le ulteriori informazioni
        case 'q' : stampa un quadrato usando le ulteriori informazioni
      else stampa un saluto
  while il carattere letto è diverso da 'f'

Per passare le ulteriori informazioni da `main` alle funzioni di stampa è necessario utilizzare dei **parametri**:
- permettono uno scambio di dati da chiamante a chiamato (e viceversa)
- nell'intestazione: lista di **parametri formali** (con tipo associato) — sono simili a variabili locali, ma vengono inizializzati
- nell'attivazione: lista di **parametri attuali** — possono essere delle espressioni

Al momento dell'attivazione ogni **parametro formale viene inizializzato al valore del corrispondente parametro attuale**.

$\Longrightarrow$ Il valore del parametro attuale viene **copiato** nella locazione di memoria del corrispondente parametro formale.

Esempio: Interfaccia per la stampa di figure a livello (2): file `funzioni/figure2.c`

Esercizio: Scrivere un programma per la stampa di figure geometriche a livello (4).

Soluzione: file `funzioni/figure4.c`

Esercizio: Scrivere un programma per la stampa di un rettangolo di altezza, larghezza e carattere di riempimento variabile. Il programma deve utilizzare una funzione di stampa di una riga del rettangolo.

Soluzione: file `funzioni/rettang3.c`

Esercizio: Scrivere un programma per la stampa di figure geometriche a livello (4), utilizzando una funzione per la stampa di una sequenza di caratteri (con lunghezza e carattere da stampare come parametri).

Soluzione: file `funzioni/figure5.c`

Funzioni che restituiscono un valore

Una funzione che restituisce un valore ha tipo di ritorno diverso da `void`.

Esempio: Funzione che restituisce il massimo tra due interi.

```
int max(int m, int n)
{
  if (m >= n)
    return m;
  else
    return n;
}
```

Attivazione di `max`, ad esempio da `main`:

```
int main(void)
{
  int i, j, massimo;
  scanf("%d%d", &i, &j);
  massimo = max(i,j);
  printf("massimo = %d\n");
  return 0;
}
```


Nel corpo **deve** esserci l'istruzione `return` *espressione*`;`

- restituisce il valore calcolato dalla funzione, che deve essere del tipo del valore di ritorno della funzione
- ritorna il controllo alla funzione chiamante

Esempio:

```
int max(int m, int n)
{
  if (m >= n)
    return m;
  else
    return n;
  printf("pippo");   /* non viene mai eseguita */
}
```

L'istruzione `return` può essere usata anche per funzioni void.

```
void f(int i)
{ ...
  if (i >= 0) return;
  printf("valore negativo");   /* non viene eseguita se i>=0 */
}
```

Esempio: Conversione da numero romano a intero.

- ingresso: sequenza di “cifre romane” terminata da ’\n’
- uscita: intero corrispondente

Facciamo uso di una funzione `Romano2Intero` che converte una singola cifra romana.

Variante 1: assumiamo che le cifre romane compaiano solo in ordine decrescente
Es.: `MMCLXVII` va bene, mentre `MCMX` no (perché `CM` e‘ decrescente)

Implementazione: file `funzioni/romani2.c`

Variante 2: sequenza di cifre romane qualsiasi (purché corretta)
Quando leggiamo una cifra

- dobbiamo aggiungerla alla somma corrente se è $\geq$ della cifra successiva
  Es.: `MMC`
- dobbiamo sottrarla dalla somma corrente se è $<$ della cifra successiva
  Es. `MCM`

$\Longrightarrow$ prima di decidere dobbiamo leggere la cifra successiva

Nell’algoritmo usiamo una *somma* corrente e due variabili *cifra_corrente* e *cifra_successiva* che mantengono le ultime due cifre romane lette.

algoritmo conversione da numero romano a intero
  inizializza *somma* a 0
  leggi *cifra_corrente*
  if *cifra_corrente* $\neq$ ’\n’
  then leggi *cifra_successiva*
    while *cifra_successiva* $\neq$ ’\n’
    do if valore di *cifra_corrente* $\geq$ valore di *cifra_successiva*
      then aggiungi valore di *cifra_corrente* a *somma*
      else sottrai valore di *cifra_corrente* da *somma*
      poni *cifra_corrente* pari a *cifra_successiva*
      leggi *cifra_successiva*
    aggiungi valore di *cifra_corrente* a *somma*
  stampa *somma*

Implementazione: per **esercizio**: file `funzioni/romani3.c`

Dichiarazioni di funzione (o prototipi)

I parametri attuali nell’attivazione di una funzione devono corrispondere in numero e tipo (in ordine) ai parametri formali.

Dobbiamo permettere al compilatore di fare questo controllo
$\Longrightarrow$ prima dell’attivazione deve conoscere l’intestazione della funzione.

Due possibilità:

1. la funzione è stata **definita** prima
2. la funzione è stata **dichiarata** prima

Sintassi di una **dichiarazione di funzione** (o **prototipo**): *`intestazione;`*

ovvero: *`tipo-di-ritorno nome-funzione (lista-parametri-formali);`*

- c’e‘ un “`;`” finale al posto del blocco
- nella lista di parametri formali può anche mancare il nome dei parametri — interessa solo il tipo
- il compilatore usa la dichiarazione per controllare che l’attivazione sia corretta
- dopo deve esserci una definizione della funzione coerente con la dichiarazione

Ordine di dichiarazioni e funzioni

Ogni funzione deve essere stata dichiarata o definita prima di essere usata.

È pratica comune specificare in quest'ordine:

1. dichiarazioni di tutte le funzioni (tranne `main`)
2. definizione di `main`
3. definizioni delle rimanenti funzioni

In questo modo ogni funzione è stata dichiarata prima di essere usata e l'**ordine** in cui mettiamo dichiarazioni e definizioni è **irrilevante**.

Esempio:

```
int Romano2Intero(char);

int main(void) { ... }

int Romano2Intero(char ch) { ... }
```


File header (o di intestazione)

Ogni libreria standard ha un corrispondente file header, che contiene

- definizioni di costanti
- definizioni di tipo
- dichiarazioni di tutte le funzioni della libreria

Esempi:

`<stdio.h>`	input/output
`<stdlib.h>`	allocazione della memoria, numeri casuali, utilità generali
`<string.h>`	manipolazione di stringhe
`<limits.h>`	limiti del sistema per valori interi
`<float.h>`	limiti del sistema per valori reali
`<math.h>`	funzioni matematiche
...	

Possono essere scritti anche per funzioni sviluppate da noi (hanno estensione .h).

Funzioni della libreria matematica

Per poterle utilizzare bisogna specificare `#include <math.h>`

Sia argomenti che valore di ritorno sono reali in doppia precisione, ovvero di tipo `double` (e non `float`).

Funzioni disponibili:

`sqrt(x)`	radice quadrata
`exp(x)`	e^x
`log(x)`	logaritmo naturale
`log10(x)`	logaritmo in base 10
`fabs(x)`	valore assoluto
`ceil(x)`	arrotonda all'intero più piccolo $\geq x$
`floor(x)`	arrotonda all'intero più grande $\leq x$
`pow(x,y)`	x^y
`fmod(x,y)`	resto di x/y (in virgola mobile)
`sin(x)`, `cos(x)`, `tan(x)`	trigonometriche (x espresso in radianti)

Esercizio: Calcolo del numero delle combinazioni di n oggetti presi r ad r, ovvero

$$\frac{n!}{r! \cdot (n-r)!}$$

utilizzando una funzione per il calcolo del fattoriale.

Soluzione: file `funzioni/combi.c`

Esercizio: Modularizzare i programmi visti finora a lezione o dati come esercizio attraverso l'introduzione di opportune funzioni.

Gestione della memoria a run-time

Codice macchina e dati entrambi in RAM, ma in zone separate:

- memoria per il codice macchina fissata a tempo di compilazione
- memoria per i dati locali alle funzioni (variabili e parametri) cresce e descresce dinamicamente durante l'esecuzione: viene gestita a **pila**

Una **pila** (o **stack**) è una struttura dati con accesso LIFO: Last In First Out = ultimo entrato è il primo a uscire (Es.: pila di piatti).

A run-time viene gestita automaticamente la **pila dei record di attivazione** (RDA) in memoria centrale:

- per **ogni attivazione di funzione** viene creato un nuovo RDA in cima alla pila
- al termine dell'attivazione della funzione il RDA viene rimosso dalla pila

Ogni RDA contiene:

- le locazioni di memoria per i parametri formali (se presenti)
- le locazioni di memoria per le variabili locali (se presenti)
- l'indirizzo di ritorno = indirizzo della prossima operazione da eseguire nella funzione chiamante

Esempio: file `funzioni/stack.c`

codice sorgente $\overset{\text{compilazione}}{\Longrightarrow}$ codice macchina (caricato in RAM al momento dell'esecuzione)

Supponiamo (per semplicità) che ad ogni istruzione del codice sorgente corrisponda una singola istruzione in linguaggio macchina:

	main()	
	...	
0A00	m1	
0A01	m2	
0A02	m3	
0A03	m4	
0A04	m5	⇒ A(s)
0A05	m6	
0A06	return	
0A07	...	

	A()	
	...	
0B00	a1	
0B01	a2	
0B02	a3	
0B03	a4	
0B04	a5	⇒ B(loc)
0B05	a6	
0B06	return	
0B07	...	

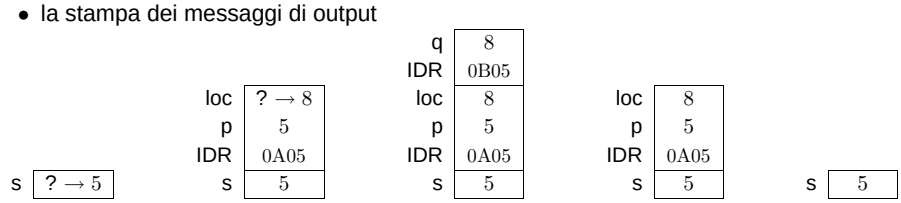
	B()
	...
0C00	b1
0C01	return
0C02	...

Due concetti fondamentali che determinano l'esecuzione:

- **program counter** (PC), che indica la prossima istruzione da eseguire
- **pila dei RDA**, con un RDA per ogni attivazione di funzione

Per seguire l'esecuzione del programma vediamo come evolvono in parallelo:

- la pila dei RDA (con indirizzo di ritorno)
- il program counter (in particolare durante le chiamate di funzione e i relativi ritorni)
- la stampa dei messaggi di output




```
Sono main()
Inserisci un intero: 5
Ora chiamo A(), con parametro attuale pari a 5
Sono A(). Il mio parametro p vale 5
Inserisci un intero: 8
Ora chiamo B(), con parametro attuale pari a 8
Sono B(). Il mio parametro q vale 8
Sono di nuovo A()
Sono di nuovo main()
```


Variabili automatiche e statiche

Tempo di vita di una variabile =

= periodo in cui esiste la cella di memoria associata alla variabile

= periodo in cui esiste il RDA corrispondente all'attivazione (per variabili **locali automatiche**)

Una variabile può anche essere **statica** $\Longrightarrow$ esiste per tutto il tempo di esecuzione del programma:

- se è dichiarata all'esterno di qualsiasi funzione, oppure
- se è locale ad una funzione e lo specificatore `static` precede la dichiarazione

  Es.: `void f(void) { static int x; ... }`
  - la variabile viene inizializzata alla prima attivazione della funzione
  - conserva il suo valore tra attivazioni successive
  - è locale, quindi visibile solo all'interno della funzione in cui è dichiarata

Esempio: Funzione che ritorna il numero di volte che è stata attivata.

Implementazione: file `funzioni/contatt.c`

Ricorsione

Una funzione che contiene al suo interno un'attivazione di sè stessa è detta **ricorsiva**.

Esempio: Programma che usa una funzione ricorsiva:

file `ricorsio/ricorsio.c`

Vediamo l'evoluzione della pila dei RDA per input 2. Output prodotto:

```
Sono main()
Inserisci un intero non negativo: 2
 - Attivo ricorsiva(2)
Sono ricorsiva(2) - Attivo ricorsiva(1)
Sono ricorsiva(1) - Attivo ricorsiva(0)
Sono ricorsiva(0) - Ho finito
Sono di nuovo ricorsiva(1) - Ho finito
Sono di nuovo ricorsiva(2) - Ho finito
Sono di nuovo main() - Ho finito
```

Cosa succede se si attiva `ricorsiva(-1)`?

Funzioni ricorsive sono convenienti per implementare funzioni matematiche che ammettono una **definizione induttiva**.

Esempio: fattoriale

- definizione iterativa: $fatt(n) = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$
- definizione induttiva:

$$fatt(n) = \begin{cases} 1, & \text{se } n = 0 \quad \text{(caso base)} \\ n \cdot fatt(n-1), & \text{se } n > 0 \quad \text{(caso ricorsivo)} \end{cases}$$

È essenziale che applicando ripetutamente il caso ricorsivo, ci riconduciamo prima o poi al caso base.

algoritmo ricorsivo per il calcolo del fattoriale di un intero nonnegativo *n*
  if $n = 0$
  then return 1
  else calcola il fattoriale di $n - 1$
    moltiplicalo per *n*
    restituisci il valore ottenuto

Implementazione: file `ricorsio/fattoria.c`

Esercizio: Implementare le operazioni di somma, prodotto, ed esponente, utilizzando le seguenti definizioni induttive di tali operazioni.

- definizione induttiva di somma tra due interi nonnegativi:

$$somma(x, y) = \begin{cases} x, & \text{se } y = 0 \\ 1 + (somma(x, y-1)), & \text{se } y > 0 \end{cases}$$

- definizione induttiva di prodotto tra due interi nonnegativi:

$$prodotto(x, y) = \begin{cases} 0, & \text{se } y = 0 \\ somma(x, prodotto(x, y-1)), & \text{se } y > 0 \end{cases}$$

- definizione induttiva di elevamento a potenza tra due interi nonnegativi:

$$esponente(x, y) = \begin{cases} 1, & \text{se } y = 0 \\ prodotto(x, esponente(x, y-1)), & \text{se } y > 0 \end{cases}$$

Si possono inserire le definizioni delle funzioni nel file `ricorsio/driveint.c`.

Soluzione: file `ricorsio/operindu.c`

Esempio: Leggere una sequenza di caratteri terminata da '\n' e stamparla invertita.

Es.: `paolo\n` $\Longrightarrow$ `oloap`

Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:

1. usando una struttura dati opportuna (array o lista) — più avanti
2. usando le celle di memoria della pila dei RDA come memoria temporanea

Implementazione: file `ricorsio/invertic.c`

Vediamo l'evoluzione della pila dei RDA con input "`abc\n`".

Esercizio: Leggere un intero e stamparne le cifre invertite (fornire una soluzione iterativa ed una ricorsiva).

Es.: `25138` $\Longrightarrow$ `83152`

Suggerimento: `25138 % 10` $=$ `8`
`25138 / 10` $=$ `2153`

Soluzione: file `ricorsio/invertin.c`

Esercizio: Leggere una sequenza di caratteri con un punto centrale, e verificare se è palindroma (ignorando gli spazi bianchi).

Una sequenza si dice **palindroma** se letta da sinistra a destra è identica a quando viene letta da destra a sinistra.

Es.: `i topi non av.vano nipoti`

Caratterizzazione induttiva di una sequenza palindroma:

- la sequenza costituita solo da '`.`' è palindroma
- una sequenza xsy è palindroma se lo è s e se x $=$ y.

Soluzione: file `ricorsio/palinric.c`

Cosa possiamo fare se la frase non contiene il '`.`' centrale?

Ricorsione multipla

Si ha ricorsione multipla quando un'attivazione di una funzione può causare **più di una attivazione ricorsiva** della stessa funzione.

Esempio: Funzione ricorsiva per il calcolo dell'n-esimo numero di Fibonacci.

Fibonacci: matematico pisano del 1200, interessato alla crescita di popolazioni.

Ideò un modello matematico per stimare il numero di individui ad ogni generazione:

$F(n)$ ... numero di individui alla generazione n-esima

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n+2) &= F(n) + F(n+1) \end{aligned}$$

$F(0)$, $F(1)$, $F(2)$, ... è detta sequenza dei numeri di Fibonacci:
$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ \ldots$

Implementazione: file `ricorsio/fibonacc.c`

Esercizio: Aggiungere il codice per contare il numero di attivazioni ricorsive di `fibonacci`.

Esercizio: Fornire un'implementazione iterativa per il calcolo dell'n-esimo numero di Fibonacci.

Esercizio: Implementare in C la funzione di Ackermann $A(m,n)$ definita come segue:

$$A(m,n) = \begin{cases} n+1, & \text{se } m = 0 \quad \text{(caso base)} \\ A(m-1,1), & \text{se } n = 0 \quad \text{(caso ricorsivo)} \\ A(m-1, A(m,n-1)), & \text{altrimenti} \quad \text{(caso ricorsivo)} \end{cases}$$

Attenzione: cresce **molto** rapidamente (non elementare): $A(x,x)$ cresce piu' rapidamente di qualsiasi catena di esponenziali $2^{2^{\cdots^{2^x}}}$.

Soluzione: file `ricorsio/ackerman.c`

Esercizio: Implementare funzioni ricorsive sfruttando le seguenti definizioni induttive:

- massimo comun divisore

$$mcd(x,y) = \begin{cases} x, & \text{se } y = 0 \\ mcd(y,r), & \text{se } y > 0 \text{ e } x = q \times y + r, \text{ con } 0 \le r < y \end{cases}$$

  Soluzione: file `ricorsio/mcdricor.c`

- verifica se due numeri interi positivi sono primi tra loro

$$primi(x,y) = \begin{cases} vero, & \text{se } x = 1 \text{ oppure } y = 1 \quad \text{(caso base)} \\ falso, & \text{se } x \neq 1,\ y \neq 1 \text{ e } x = y \quad \text{(caso base)} \\ primi(x, y-x), & \text{se } x \neq 1,\ y \neq 1 \text{ e } x < y \quad \text{(caso ricorsivo)} \\ primi(x-y, y), & \text{se } x \neq 1,\ y \neq 1 \text{ e } x > y \quad \text{(caso ricorsivo)} \end{cases}$$

  Soluzione: file `ricorsio/primrico.c`

- resto della divisione tra un intero ed un intero positivo

$$resto(x,y) = \begin{cases} resto(x+y, y), & \text{se } x < 0 \quad \text{(caso ricorsivo)} \\ x & \text{se } 0 \leq x < y \quad \text{(caso base)} \\ resto(x-y, y) & \text{se } x > y \quad \text{(caso ricorsivo)} \end{cases}$$

  Soluzione: file `ricorsio/restoric.c`

Esempio: Torri di Hanoi (leggenda Vietnamita).

- pila di dischi di dimensione decrescente su un perno 1
- vogliamo spostarla su un perno 2, usando un perno di appoggio 3
- condizioni:
  - possiamo spostare un solo disco alla volta
  - un disco più grande non può mai stare su un disco più piccolo
- secondo la leggenda: monaci stanno spostando 64 dischi;
  quando avranno finito, ci sarà la fine del mondo

Programma che stampa la sequenza di spostamenti da fare:

"muovi un disco dal perno x al perno y"

Idea: per spostare $n > 1$ dischi da 1 a 2, usando 3 come appoggio:

1. sposta $n-1$ dischi da 1 a 3
2. sposta l'n-esimo disco da 1 a 2
3. sposta $n-1$ dischi da 3 a 1

Implementazione: file `ricorsio/hanoi.c` (è un altro esempio di ricorsione multipla)

Visualizziamo l'albero delle attivazioni per 3 dischi.

Attenzione: quando si usa la ricorsione multipla, il numero di attivazioni ricorsive può essere **esponenziale** nella profondità delle chiamate ricorsive (cioè nell'altezza massima della pila dei RDA).

Esempio: Torri di Hanoi

$att(n)$ = numero di attivazioni di `muoviUnDisco` per n dischi
= numero di spostamenti di un disco

$$att(n) = \begin{cases} 1, & \text{se } n = 1 \\ 1 + 2 \cdot att(n-1), & \text{se } n > 1 \end{cases}$$

Senza 1 nel caso di $n > 1$ avremmo $att(n) = 2^{n-1}$.
$\Longrightarrow\ att(n) > 2^{n-1}$

È una caratteristica del problema (ovvero non esiste una soluzione migliore).

Esercizio: Contare il numero di attivazioni di `muoviUnDisco`.

Puntatori

Variabili di tipo puntatore

Esempio: `int a = 5;`

Proprietà della variabile `a`:

nome:	`a`
tipo:	`int`
valore:	5
indirizzo:	A010

A00E	···
A010	5
A012	···

Finora abbiamo usato solo le prime tre proprietà. Come si usa l'indirizzo?

`&a` . . . **operatore indirizzo** "`&`" applicato alla variabile `a`
$\Longrightarrow$ ha valore 0xA010 (ovvero, 61456 in decimale)

Gli indirizzi si utilizzano nelle variabili di tipo puntatore, dette anche **puntatori**.

Esempio: `int *pi;`

Proprietà della variabile `pi`:

nome:	`pi`
tipo:	**puntatore ad intero** (ovvero, indirizzo di un intero)
valore:	inizialmente casuale
indirizzo:	fissato una volta per tutte

Sintassi della dichiarazione di variabili puntatore

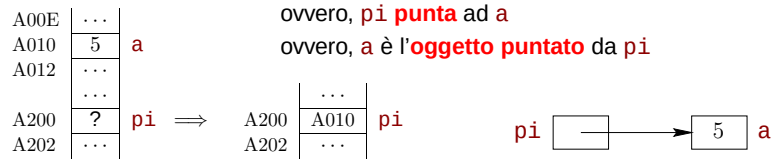
```
tipo *variabile, *variabile, ..., *variabile;
```

Esempio:
```
int *pi1, *pi2, i, *pi3, j;
float *pf1, f, *pf2;
```

`pi1`, `pi2`, `pi3`	sono di tipo puntatore ad `int`
`i`, `j`	sono di tipo `int`
`pf1`, `pf2`	sono di tipo puntatore a `float`
`f`	è di tipo `float`

Una variabile puntatore può essere inizializzata usando l'operatore di indirizzo.

Esempio: `pi = &a;` . . . il valore di `pi` viene posto pari all'indirizzo di `a`
ovvero, `pi` **punta** ad `a`
ovvero, `a` è l'**oggetto puntato** da `pi`



Operatore di dereferenziamento "*"

Applicato ad una variabile puntatore fa riferimento all'oggetto puntato.

Esempio:

```
int *pi;          /* dichiarazione di un puntatore ad intero */
int a = 5, b;     /* dichiarazione di variabili intere */

pi = &a;          /* pi punta ad a ==> *pi e' un altro modo di denotare a */
b = *pi;          /* assegna a b il valore della variabile puntata da pi,
                          ovvero il valore di a, ovvero 5 */
*pi = 9;          /* assegna 9 alla variabile puntata da pi, ovvero ad a */
```

N.B. Se `pi` è di tipo `int *`, allora `*pi` è di tipo `int`.

Non confondere le due occorrenze di "`*`":

- "`*`" in una dichiarazione serve per dichiarare una variabile di tipo puntatore
  Es.: `int *pi;`
- "`*`" in una espressione è l'operatore di dereferenziamento
  Es.: `b = *pi;`

Operatori di dereferenziamento "`*`" e di indirizzo "`&`":

- hanno priorità più elevata degli operatori binari
- "`*`" è associativo a destra
  Es.: `**p` è equivalente a `*(*p)`
- "`&`" può essere applicato solo ad una variabile;
  `&a` non è una variabile $\Longrightarrow$ "`&`" non è associativo
- "`*`" e "`&`" sono uno l'inverso dell'altro
  - data la dichiarazione `int a;`
    `*&a` è un alias per `a` (sono entrambi variabili)
  - data la dichiarazione `int *pi;`
    `&*pi` ha valore uguale a `pi`
    però: `pi` è una variabile
    `&*pi` non lo è (ad esempio, non può essere usato a sinistra di "`=`")

Stampa di puntatori

I puntatori si possono stampare con `printf` e specificatore di formato "`%p`" (stampa in formato esadecimale).

Esempio:

```
int a = 5;
int *pi;
```

A00E	···	
A010	5	a
A012	A010	pi
	···	

```
pi = &a;
printf("indirizzo di a = %p\n", &a);        /* stampa 0xA010 */
printf("valore di pi   = %p\n", pi);        /* stampa 0xA010 */
printf("valore di &*pi = %p\n", &*pi);      /* stampa 0xA010 */

printf("valore di a   = %d\n", a);          /* stampa 5      */
printf("valore di *pi = %d\n", *pi);        /* stampa 5      */
printf("valore di *&a = %d\n", *&a);        /* stampa 5      */
```

Si può usare `%p` anche con `scanf`, ma ha poco senso leggere un indirizzo.

Esempio: Scambio del valore di due variabili.

```
int a = 10, b = 20, temp;
temp = a;
a = b;
b = temp;
```

Tramite puntatori:

```
int a = 10, b = 20, temp;
int *pa, *pb;

pa = &a;         /* *pa diventa un alias per a */
pb = &b;         /* *pb diventa un alias per b */

temp = *pa;
*pa = *pb;
*pb = temp;
```

Inizializzazione di variabili puntatore

I puntatori (come tutte le altre variabili) devono venire inizializzati prima di poter essere usati.

$\Longrightarrow$ È un errore dereferenziare una variabile puntatore non inizializzata.

Esempio:

```
int a;
int *pi;
```

A00E	···	
A010	?	a
A012	F802	pi
	···	
F802	412	
F804	···	

`a = *pi;` $\Longrightarrow$ ad `a` viene assegnato il valore 412

`*pi = 500;` $\Longrightarrow$ scrive 500 nella cella di memoria di indirizzo $\mathrm{F802}$

Non sappiamo a cosa corrisponde questa cella di memoria!!!

$\Longrightarrow$ la memoria può venire corrotta

Tipo di variabili puntatore

Il tipo di una variabile puntatore è "puntatore a *`tipo`*". Il suo valore è un **indirizzo**.

I tipi puntatore sono **indirizzi** e **non interi**.

Esempio:

```
int a;
int *pi;
a = pi;
```

compilando si ottiene un warning: "assignment makes integer from pointer without a cast"

Due variabili di tipo **puntatore a tipi diversi non sono compatibili** tra loro.

Esempio:

```
int x;
int *pi;
float *pf;
```

`x = pi;` assegnazione `int*` a `int`
$\Longrightarrow$ warning: "assignment makes integer from pointer without a cast"

`pf = x;` assegnazione `int` a `float*`
$\Longrightarrow$ warning: "assignment makes pointer from integer without a cast"

`pi = pf;` assegnazione `float*` a `int*`
$\Longrightarrow$ warning: "assignment from incompatible pointer type"

Perché il C distingue tra puntatori di tipo diverso?

Se tutti i tipi puntatore fossero identici (ad es. puntatore a `void`), non sarebbe possibile determinare a tempo di compilazione il tipo di `*p`.

Esempio:

```
void *p;
int i;  char c;  float f;
```

Potrei scrivere:

```
p = &c;
p = &i;
p = &f;
```

Il tipo di `*p` verrebbe a dipendere dall'ultima assegnazione che è stata fatta!!!

Quale è il significato di `i/*p` (divisione intera oppure divisione reale)?

Il C permette di definire un puntatore a `void` (tipo `void*`)

- è compatibile con tutti i tipi puntatore
- **non** può essere dereferenziato (bisogna prima fare un cast esplicito)

Funzione `sizeof` con puntatori

La funzione `sizeof` restituisce l'occupazione in memoria in byte di una variabile. Può anche essere applicata anche ad un tipo.

Tutti i puntatori sono indirizzi $\Longrightarrow$ occupano lo spazio di memoria di un indirizzo.

L'oggetto puntato ha dimensione del tipo puntato.

Esempio: file `puntator/puntsize.c`

```
char *pc;
int *pi;
double *pd;

printf("%d %d %d  ", sizeof(pc),     sizeof(pi),    sizeof(pd));
printf("%d %d %d\n", sizeof(char *), sizeof(int *), sizeof(double *));

printf("%d %d %d  ", sizeof(*pc),  sizeof(*pi), sizeof(*pd));
printf("%d %d %d\n", sizeof(char), sizeof(int), sizeof(double));
```

```
4 4 4  4 4 4
1 2 8  1 2 8
```


Passaggio di parametri per indirizzo

Differenza tra copia del valore e copia dell'indirizzo di una variabile:

Esempio:

```
int b, x, *p;

b = 15;                         /* b vale 15 */
x = b;                          /* il valore di b viene copiato in x */
p = &b;                         /* l'indirizzo di b viene messo in p */

x = 23;
printf("b vale %d\n", b);       /* b non e' cambiato */
*p = 47;
printf("b vale %d\n", b);       /* b e' cambiato */
```

Se si ha una **copia dell'indirizzo** di una variabile questa copia **può essere usata per modificare la variabile** (il puntatore dereferenziato è un modo alternativo di denotare la variabile).

Sfruttando questa idea è possibile fare in modo che una **funzione modifichi una variabile della funzione chiamante**.

In C i **parametri** delle funzioni sono **passati per valore**:

- il parametro formale è una nuova variabile locale alla funzione
- al momento dell'attivazione il valore del parametro attuale viene copiato nel parametro formale

$\Longrightarrow$ le modifiche fatte sul parametro formale **non** si riflettono sul parametro attuale (come `b` e `x` dell'esempio precedente).

Però, se **passiamo** alla funzione **un puntatore ad una variabile**, la funzione può usare il puntatore per modificare la variabile.

Esempio: file `puntator/parametr.c`

Si tratta di un **passaggio per indirizzo**:

- la funzione chiamante passa l'indirizzo della variabile come parametro attuale
- la funzione chiamata usa l'operatore "`*`" per riferirsi alla variabile passata

(simula il **passaggio per riferimento** che esiste in molti linguaggi)

Esempio: Per passere un intero `i` per indirizzo:

il parametro formale si dichiara come: `int *pi` (di tipo: `int*`)
il parametro attuale è l'indirizzo di `i`: `&i`
nel corpo della funzione si usa: `*pi`

Il passaggio per indirizzo viene usato ogni volta che una funzione deve restituire più di un valore alla funzione chiamante.

Esempio: Funzione per lo scambio dei valori di due variabili, e funzione che stampa due valori in ordine crescente.
Implementazione: file `puntator/scambio.c` e file `puntator/ordina2.c`

Esempio: Scrivere una funzione che riceve come parametri giorno, mese, ed anno di una data, e li aggiorna ai valori per la data del giorno dopo.
Implementazione: file `puntator/datasuN1.c`

Esercizio: Utilizzare la funzione appena sviluppata per calcolare la data dopo *n* giorni:

- iterando *n* volte il calcolo della data del giorno successivo
  Soluzione: `puntator/datasuN1.c`
- versione ottimizzata, che passa direttamente al primo del mese successivo
  Soluzione: `puntator/datasuN2.c`

Allocazione dinamica della memoria

Un puntatore deve puntare ad una zona di memoria

- a cui il sistema operativo permette di accedere
- che non viene modificata inaspettatamente

Finora abbiamo visto un modo per soddisfare questi requisiti: assegnare ad un puntatore l'indirizzo di una delle variabili del programma.

Metodo alternativo: **allocazione dinamica della memoria**, attraverso una chiamata di funzione che crea una nuova zona di memoria e ne restituisce l'indirizzo iniziale.

- la zona di memoria è accessibile al programma
- la zona di memoria non viene usata per altri scopi (ad esempio variabili in altre funzioni)
- ad ogni chiamata della funzione viene allocata una nuova zona di memoria

Funzione `malloc`

La funzione `malloc` è dichiarata in `<stdlib.h>` con prototipo:

```
void * malloc(size_t);
```

- prende come parametro la dimensione (numero di byte) della zona da allocare (`size_t` è il tipo restituito da `sizeof` e usato per le dimensioni in byte delle variabili — ad esempio potrebbe essere `unsigned long`)
- alloca (riserva) la zona di memoria
- restituisce il puntatore iniziale alla zona allocata (è una funzione che restituisce un puntatore)

N.B. La funzione `malloc` restituisce un puntatore di tipo `void*`, che è compatibile con tutti i tipi puntatore.

Esempio:
```
float *p;
p = malloc(4);
```

Uso tipico di `malloc` è con `sizeof(`*`tipo`*`)` come parametro.

Esempio:

```
#include <stdlib.h>

int *p;
p = malloc(sizeof(int));
*p = 12;
(*p)++;                          /* N.B. servono le parentesi */
printf("*p vale %d\n", *p);
```

- attivando `malloc(sizeof(int))` viene allocata una zona di memoria adatta a contenere un intero; ovvero viene creata una nuova variabile intera
- il puntatore restituito da `malloc` viene assegnato a `p`
  $\Longrightarrow$ `*p` si riferisce alla nuova variabile appena creata

Lo heap (o memoria dinamica)

La zona di memoria allocata attraverso `malloc` si trova in un'area di memoria speciale, detta **heap** (o **memoria dinamica**).

$\Longrightarrow$ abbiamo **4 aree di memoria**:

- zona programma: contiene il codice macchina
- stack: contiene la pila dei RDA
- statica: contiene le variabili statiche
- heap: contiene dati allocati dinamicamente

Funzionamento dello heap:

- gestito dal sistema operativo
- le zone di memoria sono **marcate libere o occupate**
  - marcata libera: può venire utilizzata per la prossima `malloc`
  - marcata occupata: non si tocca

Potrebbe **mancare la memoria** per allocare la zona richiesta. In questo caso `malloc` restituisce il puntatore `NULL`.

$\Longrightarrow$ Bisogna sempre verificare cosa restituisce `malloc`.

Esempio:

```
p = malloc(sizeof(int));
if (p == NULL) {
  printf("Non ho abbastanza memoria per l'allocazione\n");
  exit(1);
}
...
```

La costante `NULL`

- è una costante di tipo `void*` (quindi compatibile con tutti i tipi puntatore)
- indica un puntatore che non punta a nulla $\Longrightarrow$ non può essere dereferenziato
- ha tipicamente valore 0
- definita in `<stdlib.h>` (ed in altri file header)

Deallocazione della memoria dinamica

Le celle di memoria allocate dinamicamente devono essere **deallocate** (o rilasciate) quando non servono più.

Si utilizza la funzione `free`, che è dichiarata in `<stdlib.h>`:

```
void * free(void *);
```

Esempio:

```
int *p;
...
p = malloc(sizeof(int));
...
free(p);
```

- il parametro `p` **deve** essere un puntatore ad una zona di memoria allocata precedentemente con `malloc` (altrimenti il risultato non è determinato)
- la zona di memoria **viene resa disponibile** (viene marcata libera)
- `p` non punta più ad una locazione significativa (ha **valore arbitrario**)

Prima del termine del programma **bisogna deallocare tutte le zone** allocate dinamicamente.

$\Longrightarrow$ Per ogni `malloc` deve essere eseguita una `free` corrispondente (sulla stessa zona di memoria, non necessariamente usando lo stesso puntatore).

Esempio: file `puntator/puntator.c`

```
int *pi;
int *pj;

pi = malloc(sizeof(int));    /* allocazione dinamica di memoria */
*pi = 150;                   /* ora *pi ha un valore significativo */
pj = pi;                     /* pi e pj PUNTANO ALLA STESSA CELLA DI MEMORIA */
free(pj);                    /* deallocazione di *pj, E QUINDI ANCHE DI *pi */
pi = malloc(sizeof(int));    /* allocazione dinamica di memoria */
*pi = 4000;                  /* ora *pi ha di nuovo un valore significativo */
pj = malloc(sizeof(int));    /* allocazione dinamica di memoria */
*pj = *pi;                   /* le celle contengono lo stesso valore */
pj = malloc(sizeof(int));    /* ERRORE METODOLOGICO:
                                HO PERSO UNA CELLA DI MEMORIA */
```


Tempo di vita di una variabile allocata dinamicamente

- dalla chiamata a `malloc` che la alloca fino alla chiamata a `free` che la dealloca
- indipendente dal tempo di attivazione della funzione che ha chiamato `malloc`

Attenzione: il puntatore ha tempo di vita come tutte le variabili locali

Esempio: file `puntator/vitadin.c`

Tipi primitivi del C

Un tipo è costituito da un insieme di **valori** ed un insieme di **operazioni** su questi valori.

Classificazione dei tipi primitivi del C

- scalari
  - aritmetici
    - interi: con segno, senza segno, caratteri, enumerati
    - reali
  - puntatori
- aggregati
  - array
  - strutture
  - unione
- funzione
- void (nessun valore e nessuna operazione)

Tipi aritmetici del C: per ciascun tipo consideriamo i seguenti **aspetti**:

1. intervallo di definizione
2. notazione per le costanti (nel codice, oppure in input/output)
3. operatori
4. predicati (operatori di confronto)
5. funzionalità di ingresso/uscita

Tipi interi

L'ANSI C richiede che gli **interi** siano **codificati in binario** (altrimenti molte delle operazioni a basso livello non sarebbero possibili).

Interi con segno

3 tipi: `short` (oppure `short int`, `signed short`, `signed short int`)
`int` (oppure `signed int`, `signed`)
`long` (oppure `long int`, `signed long`, `signed long int`)

Intervallo di definizione: da -2^{n-1} a $2^{n-1}-1$, dove n dipende dal compilatore

Vale che: `sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)`
`sizeof(short)` ≥ 2 (ovvero, almeno 16 bit)
`sizeof(long)` ≥ 4 (ovvero, almeno 32 bit)

LccWin32 e 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: `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `>`, `<=`, `>=`

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)

Esercizio: Compilare ed eseguire il file `tipi/intlim.c`.

Interi senza segno

3 tipi: `unsigned short` (oppure `unsigned short int`)
`unsigned int` (oppure `signed`)
`unsigned long` (oppure `unsigned long int`)

Intervallo di definizione: da 0 a $2^n - 1$, dove n dipende dal compilatore.
Il numero n di bit è come per i corrispondenti interi con segno.

Le costanti definite in `limits.h` sono:
`USHRT_MAX`, `UINT_MAX`, `ULONG_MAX` (si noti che il minimo è sempre 0)

Esercizio: Compilare ed eseguire il file `tipi/unsiglim.c`.

Notazione per le costanti:

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

Nel codice si può far seguire le cifre 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 operazione vengono fatte modulo 2^n.

Attenzione alla conversione tra `signed` e `unsigned` in operazioni miste: **il valore `signed` viene promosso a `unsigned`**.

Esempio:
```
unsigned int u;
if (u > -1) ...
```

La condizione `(u > -1)` è **sempre falsa** perchè `-1` viene convertito in `unsigned` e diventa il più grande `int` (numero con bit pari a $111\cdots111$).

Caratteri

Servono per rappresentare caratteri alfanumerici attraverso opportuni **codici**, tipicamente il codice **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange).

Un codice associa ad ogni carattere un intero:
Esempio: Codice ASCII:

carattere:	`'#'`	`'0'`...`'9'`	`';'`	`'A'`...`'Z'`	`'a'`...`'z'`	`{`	`}`	...
intero (in decimale):	35	$48\cdots57$	59	$65\cdots90$	$97\cdots122$	123	125	...

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

3 tipi: `char`, `signed char`, `unsigned char`.

I tipi `signed char` e `unsigned char` hanno lo stesso tipo di rappresentazione degli interi (rispettivamente complemento a 2 e binario).

Il tipo `char` può essere (dipende dall'implementazione):

- signed (come `signed char`)
- unsigned (come `unsigned char`)
- un misto tra i due (solo valori nonnegativi, ma come signed per le conversioni)

Esempio:

```
unsigned char uc = -1;    int i = uc;
signed char   sc = -1;    int j = sc;
char           c = -1;    int k = c;

printf("i = %d,  j = %d,  k = %d\n", i, j, k);
```

Se `char` equivale a `signed char` stampa: `i = 255,  j = -1,  k = -1`

Se `char` equivale a `unsigned char` stampa: `i = 255,  j = -1,  k = 255`

Esercizio: Stabilire se in LccWin32 i caratteri sono rappresentati con o senza segno.

Intervallo di definizione: dipende dal compilatore

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

Esercizio: Compilare ed eseguire il file `tipi/charlim.c`.

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

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

Esempio:

```
char x, y, z;        posso usare?
x = 'A';             x = 65;
y = '\n';            y = 10;
z = '#';             z = 35;
```

Non è sbagliato, però è **pessimo stile** di programmazione.
Non è detto che il codice sia 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;
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
```

```
Immetti due interi
# => 18a25^
18 97 25
```


Funzioni per la stampa e 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);
```

Esempio: Leggere da tastiera due interi x ed y ed un carattere op che rappresenta un operatore tra $+$, $-$, $*$, $/$, e stampare $x\ op\ y$.

Implementazione: file `tipi/charop.c`

Tipi reali

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

Intervallo di definizione: 3 tipi, le cui caratteristiche sono stabilite da uno standard ISO/IEEE:

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

Le caratteristiche sono descritte nel file `float.h`.

Esercizio: Compilare ed eseguire il file `tipi/floatlim.c`

Costanti: con punto decimale o notazione esponenziale

Esempio:
```
double x, y, z, w;
x = 123.45;
y = 0.0034;        y = .0034;
z = 34.5e+20;      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: stessi che 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:
  ```
  double 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 |
  ```
- `%g` (oppure `%G`)... forma ottimizzata
  - come `%e` se l'esponente è < -4 oppure il numero è $\geq$ precisione specificata
  - altrimenti come `%f`

  Però: `%8.3g` ... 8 cifre complessive, di cui 3 cifre significative (non necessariamente dopo la virgola)

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

Riassunto degli specificatori di formato per i tipi reali:

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

Conversioni di tipo

Situazioni in cui si hanno conversioni di tipo

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

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

Esempio: si ha cambiamento: da `short` a `long` (dimensioni diverse)
da `int` a `float` (anche se stessa dimensione)
non si ha cambiamento: da `int` a `unsigned int`

Conversioni implicite tra operandi di tipo diverso nelle espressioni

`short` → `int` → `long` → `float` → `double` → `long double`
`char` → `int`

Operando di tipo più a sinistra viene convertito nel tipo più a destra, che è anche il tipo del risultato.

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`

Inoltre: `signed` (`short`/`long`) viene convertito in `unsigned` (`short`/`long`)

N.B. Se il valore `signed` è negativo, il valore `unsigned` risultante sarà un numero positivo con bit più significativo pari a 1.

Conversioni sicure e non

- Le conversioni di sopra sono **sicure**: non si perde il valore.
  Eccezione: da `long` a `double` ci può essere perdita di precisione.
- Se invertiamo le conversioni sicure si può perdere il valore:
  `short` ← `int` ← `long` ← `float` ← `double` ← `long double`
  `char` ← `int`
  - ci può essere overflow
  - da reali a interi si ha troncamento della parte frazionaria

Conversioni in assegnazione

- Il risultato viene **sempre** convertito nel tipo della variabile a sinistra (con eventuale overflow o troncamento).
- Se la conversione non è possibile si ha errore.

Esempio:

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

*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.

Esempio riassuntivo su input/output per i tipi aritmetici e conversioni di tipo:
file `tipi/tipiarit.c`

Formattazione dell'input/output

Tutto l'input/output è eseguito attraverso gli stream.

- uno **stream** è una sequenza di caratteri organizzata in righe (terminate da "\n")
- ANSI C: righe di almeno 254 caratteri
- al momento dell'esecuzione, al programma vengono connessi automaticamente 3 stream:
  - standard input: di solito tastiera
  - standard output: di solito schermo
  - standard error: di solito schermo

  Tipicamente il sistema operativo consente di ridirigere gli stream standard, ad esempio su un file.

Formattazione dell'output con `printf`

Specificatori di formato:

- interi: `%d`, `%o`, `%u`, `%x`, `%X`
  per `short`: si antepone `h`
  per `long`: si antepone `l` (minuscola)
- reali: `%e`, `%E`, `%f`, `%g`, `%G`
  per `double`: non si antepone nulla
  per `long double`: si antepone `L`
- caratteri: `%c`
- stringhe: `%s` (le vedremo più avanti)
- puntatori: `%p`

Flag: messi subito dopo il "`%`"

- "`-`": allinea a sinistra
- altri flag: vedi libro

Sequenze di escape: `\%`, `\'`, `\"`, `\?`, `\\`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`

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`

Gruppo di scansione: `%[`*`gruppo-caratteri`*`]`

- utilizzato per la lettura di una sequenza di caratteri
- argomento corrispondente: vettore di caratteri (puntatore a `char`)
- vengono letti solo i caratteri compresi nel gruppo
- l'input si ferma al primo carattere fuori dal gruppo

Esempio:

```
char v[10];  int i;
scanf("%[aeiou]%d", v, &i);
printf("v = %s\n", v);
printf("i = %d\n", i);
```

```
# => aauei245^
v = aauei
i = 245
```

- un gruppo di scansione può contenere un intervallo
  Esempio: `%[0-9]` ... sequenza di cifre

- un gruppo di scansione può essere invertito: `%[^...]`

Esempio:

```
scanf("%[^0-9] %d", v, &i);
printf("v = %s\n", v);
printf("i = %d\n", i);
```

```
# => aa&;uei245
v = aa&;uei
i = 245
```

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", &g, &m, &a);
```

Esempio: Salta tutti i caratteri e poi salta "`\n`".

```
scanf("%*[^\n]");
getchar();
```


Array

Tipi di dato semplici e strutturati

- i tipi di dato visti finora erano tutti semplici: `int`, `char`, `float`, ..., puntatori
- dati manipolati sono spesso complessi (o **strutturati**) con componenti elementari o strutturate a loro volta

Gli **array** sono uno dei tipi di dato strutturati

- sono composti da **elementi omogenei** (tutti dello stesso tipo)
- ogni elemento è identificato all'interno dell'array da un **numero d'ordine** detto **indice** dell'elemento
- il numero di elementi del vettore è detto **lunghezza** (o **dimensione**) del vettore

Array monodimensionali (o vettori)

Dichiarazione di variabili di tipo vettore

```
tipo-elementi nome-array [lunghezza];
```

Esempio: `int vet[6];`

Alloca un vettore di 6 elementi, ovvero 6 locazioni di memoria consecutive, ciascuna contenente un intero. 6 è la **lunghezza** del vettore.

La **lunghezza di un vettore deve essere costante** (nota a tempo di compilazione).

In C l'**indice degli elementi** va sempre da 0 a *lunghezza* -1.

indice	elemento	variabile
0	?	`vet[0]`
1	?	`vet[1]`
2	?	`vet[2]`
3	?	`vet[3]`
4	?	`vet[4]`
5	?	`vet[5]`

`vet[i]` denota l'**elemento** del vettore `vet` **di indice** `i`.
Ogni elemento del vettore è una variabile.

Esempio:
```
int vet[6], a;
vet[0] = 15;
a = vet[0];
vet[1] = vet[0] + a;
printf("%d", vet[0] + vet[1]);
```


L'indice del vettore deve essere un intero.

Esempio:
```
...
a = 2;
vet[a] = 12;
vet[a+1] = 23;
```

In realtà, "`[]`" è un operatore (con priorità elevata – come quella di `()`).

Manipolazione di vettori

- avviene solitamente attraverso cicli `for`
- l'indice del ciclo varia in genere da 0 a *lunghezza* -1.
- conviene definire la lunghezza come una costante
  Es.: `#define LUNG 6`

N.B. `#define LUNG 6;` sarebbe sbagliato ("`LUNG`" verrebbe sostituito con "`6;`")

Esempio: Lettura e stampa di un vettore.

Implementazione: file `array/vettrw.c`

Inizializzazione di vettori

Gli elementi del vettore possono essere inizializzati con **valori costanti** (valutabili a compile-time) contestualmente alla dichiarazione del vettore .

Esempio: `int n[4] = {11, 22, 33, 44};`

- l'inizializzazione deve essere contestuale alla dichiarazione
  Esempio:
  ```
  int n[4];
  n = {11, 22, 33, 44};
  ```
  errore!
- se ci sono meno inizializzatori di elementi, quelli rimanenti vengono posti a 0
  Esempio:
  ```
  int n[10] = {3};
  ```
  azzera i rimanenti 9 elementi del vettore
  ```
  float af[5] = {0.0}
  ```
  pone i 5 elementi pari a 0.0
  ```
  int x[5] = {};
  ```
  errore!
- se ci sono più inizializzatori di elementi, si ottiene un errore di sintassi
  Esempio: `int v[2] = {1, 2, 3};` **errore!**
- se si mette una lista di inizializzatori, si può evitare di specificare la lunghezza (viene presa la lunghezza della lista)
  Esempio: `int n[] = {1, 2, 3};` equivale a `int n[3] = {1, 2, 3};`

In C l'unica operazione possibile sugli array è l'accesso agli elementi.
$\Longrightarrow$ Non si possono effettuare direttamente delle assegnazioni tra vettori.

Esempio:
```
int a[3] = {11, 22, 33};
int b[3];
b = a;
```
errore! (inoltre non farebbe quello che ci aspettiamo)

Esempio: Calcolare la somma degli elementi di un vettore.

```
int a[10], i, somma = 0;
...
for (i = 0; i < 10; i++)
  somma += a[i];
printf("%d", somma);
```

Esempio: Calcolare il massimo di un vettore di 10 elementi interi.

```
int a[10], i, max;
...
max = a[0];
for (i = 1; i < 10; i++)
  if (a[i] > max) max = a[i];
printf("%d", max);
```


Esempio: Leggere 20 reali e stampare i valori inferiori al 50% della media.
2 aspetti da considerare

- le elaborazioni sui 20 elementi sono molto simili
- prima di poter iniziare a stampare i risultati bisogna avere letto tutti e 20 i valori

Implementazione: file `array/esperime.c`

Esercizio: Leggere da tastiera 20 interi e stamparli in sequenza, non stampando un numero se era già stato stampato prima.

Esempio: Leggere una sequenza di caratteri terminata da '\n' e stampare le frequenze delle cifre da '0' a '9'.
Implementazione: file `array/frequen1.c` (versione con `switch`)
file `array/frequen2.c` (versione migliorata)

Esercizio: Leggere una sequenza di caratteri terminata da '\n' e

- stampare la frequenza della lettera alfabetica maiuscola a frequenza massima tra tutte le lettere maiuscole
- stampare la frequenza della lettera alfabetica a frequenza massima tra tutte le lettere (ignorando la differenza tra maiuscole e minuscole)

Array multidimensionali

Dichiarazione di array multidimensionali

```
tipo-elementi nome-array [lung-1][lung-2]···[lung-n]
```

Esempio: `int mat[3][4];` $\Longrightarrow$ array bidimensionale di 3 righe per 4 colonne
(ovvero **matrice** 3×4)

Per ogni dimensione *i* l'indice va da 0 a *lung-i* $- 1$.

		colonne			
		0	1	2	3
righe	0	?	?	?	?
	1	?	?	?	?
	2	?	?	?	?

Esempio: `int marketing[10][5][12]`
(indici potrebbero rappresentare: prodotti, venditori, mesi dell'anno)

Accesso agli elementi di una matrice

Esempio:

```
int i, mat[3][4];
...
i = mat[0][0];              elemento di riga 0 e colonna 0 (primo elemento)
mat[2][3] = 28;             elemento di riga 2 e colonna 3 (ultimo elemento)
mat[2][1] = mat[0][0] * mat[1][3];
```

Come per i vettori, l'unica operazione possibile sulle matrici è l'accesso agli elementi tramite l'operatore `[]`.

Esempio: Lettura e stampa di una matrice.

Implementazione: file `array/matrici.c`

Esempio: Programma che legge due matrici $M \times N$ (ad esempio 4×3) e stampa la matrice somma.

Implementazione: file `array/matsomma.c`

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    c[i][j] = a[i][j] + b[i][j];
```

Inizializzazione di matrici

Esempio:

```
int mat[2][3] = {{1,2,3}, {4,5,6}};
int mat[2][3] = {1,2,3,4,5,6};
```

1	2	3
4	5	6

```
int mat[2][3] = {{1,2,3}};
int mat[2][3] = {1,2,3};
```

1	2	3
0	0	0

```
int mat[2][3] = {{1}, {2,3}};
```

1	0	0
2	3	0

Esempio: Programma che legge una matrice A $(M \times P)$ ed una matrice B $(P \times N)$ e calcola e stampa la matrice C prodotto delle due matrici.

La matrice C è di dimensione $M \times N$.

Il generico elemento C_{ij} di C è dato da:

$$C_{ij} = \sum_{k=0}^{P-1} A_{ik} \cdot B_{kj}$$

Implementazione: file `array/matprod.c`

In alternativa, si può inizializzare `c` contestualmente alla sua dichiarazione.

```
#define M 3
#define P 4
#define N 2
int a[M][P],  b[P][N],  c[M][N] = {0};
...
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < P; k++)
      c[i][j] += a[i][k] * b[k][j];
```

Esercizio: Programma che legge una matrice $A\ (M \times N)$ e:

1. stampa l'elemento massimo con i suoi indici di riga e di colonna;
2. costruisce il vettore degli elementi massimi di ogni riga, e lo stampa;
3. costruisce il vettore degli elementi massimi di ogni colonna, e lo stampa;
4. verifica se la matrice è diagonale (in questo caso $M = N$)
   (una matrice si dice diagonale se $A_{ij} = 0$ quando $i \neq j$);
5. verifica se la matrice è simmetrica
   (una matrice si dice simmetrica se $A_{ij} = A_{ji}$ per ogni coppia di indici i e j);
6. calcola la matrice $T\ (N \times M)$ trasposta di A, e la stampa
   (il generico elemento T_{ij} della trasposta T di A è pari ad Aji).

Esempio: Programma che legge una matrice $A\ (M \times N)$ e verifica se tutte le somme degli elementi di ogni riga coincidono.

algoritmo verifica se le somme delle righe di $A(M \times N)$ sono tutte uguali tra loro
  prima $\leftarrow$ somma degli elementi della prima riga di *A*
  for ogni riga *i* di *A*, finché le somme delle righe sono tutte uguali
  do *somma* $\leftarrow$ somma degli elementi della riga *i* di *A*
    if *somma* $\neq$ *prima*
    then le somme delle righe sono diverse
  stampa se le somme delle righe sono diverse o meno

Implementazione: file `array/matsomri.c`

Si poteva risolvere senza usare una matrice?
E se dobbiamo verificare se le somme degli elementi di ogni colonna coincidono?

Esercizio: Programma che legge da tastiera una matrice quadrata e verifica se è magica, ovvero se le somme delle righe, delle colonne e delle due diagonali coincidono.

Soluzione: file `array/magica.c` (fa uso di funzioni con parametro di tipo matrice)

Aritmetica dei puntatori

Sui puntatori si possono effettuare diverse **operazioni**:

- di **dereferenziamento**
  Esempio:
  ```
  int *p, i;
  i = *p;
  ```
- di **assegnamento**
  Esempio:
  ```
  int *p, *q;
  p = q;
  ```
  N.B. `p` e `q` devono essere dello stesso tipo (altrimenti bisogna usare l'operatore di cast).
- di **confronto**
  Esempio: `if (p == q) ···`
  Esempio: `if (p > q) ···` **Ha senso?** Con quello che abbiamo visto finora no. Vedremo tra poco che ci sono situazioni in cui ha senso.
- **aritmetiche**, con opportune limitazioni
  - incremento (`++`) o decremento (`--`)
  - somma (`+=`) o sottrazione (`-=`) di un intero
  - sottrazione di un puntatore da un altro

Significato delle operazioni aritmetiche sui puntatori

Il **numero di byte** di cui viene modificato il puntatore **dipende dal suo tipo**.

Esempio:
```
int *pi;
*pi = 15;
pi++;
*pi = 20;
```
$\Longrightarrow$ `pi` punta al prossimo `int` (4 byte dopo)

Esempio:
```
double *pd;
*pd = 12.2;
pd += 3;
```
$\Longrightarrow$ `pd` punta a 3 `double` dopo (24 byte dopo)

Esempio:
```
char *pc;
*pc = 'A';
pc -= 5;
```
$\Longrightarrow$ `pc` punta a 5 `char` prima (5 byte prima)

Si può anche scrivere:
```
pi = pi + 1;
pd = pd + 3;
pc = pc - 5;
```


Relazione tra vettori e puntatori

Attenzione: in generale non sappiamo cosa contengono le celle di memoria adiacenti ad una data cella.

L'unico caso in cui sappiamo quali sono le locazioni di memoria successive e cosa contengono è quando utilizziamo dei vettori.

In C il **nome di un vettore è** in realtà **l'indirizzo dell'elemento di indice** 0.

Esempio: `int vet[10];` `vet` e `&vet[0]` hanno lo stesso valore.
$\Longrightarrow$ `printf("%p %p", vet, &vet[0]);` stampa 2 volte lo stesso indirizzo.

Possiamo far puntare un puntatore al primo elemento di un vettore.

Esempio:
```
int vet[5];
int *pi;
pi = vet;
```
è equivalente a `pi = &vet[0];`

Accesso agli elementi di un vettore

Esempio:
```
int vet[5];
int *pi = vet;
*(pi + 3) = 28;
```
`pi+3` punta all'elemento di indice 3 del vettore.

3 viene detto **offset** (o scostamento) del puntatore.

N.B. Servono le "`()`" perchè "`*`" ha priorità maggiore di "`+`". Che cosa denota `*pi + 3` ?

Osservazione: `&vet[3]` equivale a `pi+3` equivale a `vet+3`
`*&vet[3]` equivale a `*(pi+3)` equivale a `*(vet+3)`

Inoltre, `*&vet[3]` equivale a `vet[3]`.

$\Longrightarrow$ In C, `vet[3]` è semplicemente un modo alternativo di scrivere `*(vet+3)`.

Notazioni per gli elementi di un vettore:
`vet[3]` .... notazione con **puntatore e indice**
`*(vet+3)` .... notazione con **puntatore e offset**

Riassumendo, si può accedere agli elementi di un vettore al seguente modo:

Esempio:

```
int vet[5] = {11, 22, 33, 44, 55};
int *pi = vet;
int offset = 3;

vet[offset] = 88;
*(vet + offset) = 88;
pi[offset] = 88;
*(pi + offset) = 88;
```

Esempio: Sono corrette le istruzioni nel seguente frammento di codice?

```
int vet[10];
int *pi;

vet = pi;
vet++;
vet += 2;
```

No, perché `vet` è un **puntatore costante**, e quindi non può essere modificato.

Passaggio di parametri di tipo vettore

Quando si passa un vettore come parametro ad una funzione, in realtà si sta passando l'indirizzo dell'elemento di indice 0.

Il parametro formale deve essere di tipo puntatore al tipo degli elementi del vettore.

Di solito si passa la dimensione del vettore in un ulteriore parametro.

Esempio:

```
void stampa(double *v, int dim)
{
  int i;
  for (i = 0; i < dim; i++)
    printf("%d: %g\n", i, v[i]);
}

int main(void)
{ ...
  double vet[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
  stampa(vet, 5);
  ...
}
```


Per evidenziare che il parametro formale è in realta un vettore (ovvero l'indirizzo dell'elemento di indice 0), di solito si usa la notazione *`nome-parametro`*`[]` invece di *`*nome-parametro`*.

Esempio: `void stampa(int v[], int dim) { ... }`

Si può anche specificare la dimensione nel parametro, ma viene ignorata.
Esempio: `void stampa(int v[5], int dim) { ... }`

Nel prototipo della funzione può anche mancare il nome del vettore.
Esempio: `void stampa(int [], int);`

Il passaggio di un vettore è in realtà un **passaggio per indirizzo**.
$\Longrightarrow$ La funzione può modificare gli elementi del vettore passato.

Esempio: Funzioni per la lettura e stampa di un vettore.
Implementazione: file `array/vettfunz.c`

Esempio: Programma che legge un vettore di dimensione N e lo inverte.
Implementazione: file `array/vettinv.c` (versioni iterativa e ricorsiva)

Utilizzo di `sizeof` con i vettori

Abbiamo visto che un vettore è un puntatore costante all'elemento di indice 0.

Esempio:
```
int vet[5];
int *pi = vet;
pi[3] = 12;
```
Possiamo usare un puntatore per accedere agli elementi del vettore.

Importante differenza tra vettori e puntatori:
- `sizeof(pi)` è equivalente a `sizeof(int*)`
- `sizeof(vet)` è equivalente a `5*sizeof(int)`
  $\Longrightarrow$ per i vettori, `sizeof` restituisce la dimensione dell'intero blocco

Il C usa la dimensione degli elementi di un vettore per calcolare l'offset di un elemento quando usiamo la notazione con indici.

Esempio: `int v[5];`

`v[i]` è equivalente a `*(v+i)`. Questo corrisponde alla cella di memoria a distanza `i*sizeof(int)` da quella a cui punta `v`.

Questo influenza il modo in cui il C gestisce le matrici.

Esempio: `int mat[2][3] = {{0, 1, 2}, {3, 4, 5}}`
corrisponde ad un vettore di 2 elementi, ognuno dei quali è un vettore di 3 interi.
Quindi `mat` è un puntatore ad un vettore di 3 `int`.

`mat[1][2]` è equivalente a
`*(mat[1] + 2)` che è equivalente a
`*(*(mat + 1) + 2)`

- nel valutare `*(mat+1)`, sul valore `1` viene effettuato uno scaling della dimensione dell'oggetto a cui punta `mat`, ovvero un array di 3 `int`
  $\Longrightarrow$ incremento di $1 \cdot 3 \cdot$ `sizeof(int)` byte
- nel valutare `*(mat[1] + 2)`, sul valore `2` viene effettuato uno scaling della dimensione dell'oggetto a cui punta `mat[1]`, ovvero un `int`
  $\Longrightarrow$ incremento di $2 \cdot$ `sizeof(int)` byte

Otteniamo: `mat` $+ 1 \cdot 3 \cdot 4 + 2 \cdot 4$ byte $=$ `mat` $+ 20$ byte
Più precisamente, `mat[1][2]` è equivalente a:
```
*(int*) ((char*)mat + (1*3*4) + (2*4)),   ovvero
*(int*) ((char*)mat + 20)
```


In generale:
```
#define R ...
#define C ...
int mat[R][C];
```

`mat[i][j]` è equivalente a
`*(mat[i] + j)` che è equivalente a
`*(*(mat + i) + j)`

L'indirizzo di `mat[i][j]` è: `mat` $+ ($`i` $\cdot$ `C` $\cdot$ `sizeof(int)` $+$ `j` $\cdot$ `sizeof(int)`$)$ byte

Quindi, per calcolare l'indirizzo dell'elemento `mat[i][j]` è necessario conoscere:
- il valore di `mat`, ovvero l'indirizzo del primo elemento della matrice
- l'indice di riga `i` dell'elemento
- l'indice di colonna `j` dell'elemento
- il numero `C` di colonne della matrice

Non è invece necessario conoscere il numero `R` di righe della matrice.

Nota: Se si specifica un indice in meno delle dimensioni, si ottiene un puntatore al tipo base dell'array.

Esempio: `mat[1]` coincide con `&mat[1][0]`, che fornisce un puntatore ad `int`.

Passaggio di matrici come parametri

Quando passiamo un vettore ad una funzione stiamo passando il puntatore (costante) all'elemento di indice 0. $\Longrightarrow$ **Non** serve specificare la dimensione del vettore nel parametro formale (è un puntatore al tipo degli elementi del vettore).

Quando passiamo una matrice ad una funzione, per poter calcolare l'offset corretto, la funzione deve **conoscere il numero di colonne** della matrice. $\Longrightarrow$
Non possiamo specificare il parametro nella forma `mat[][]`, come per i vettori, ma dobbiamo specificare il numero di colonne.

Esempio:
```
void stampa(int mat[][5], int righe) { ... }
```

In generale, in un parametro di tipo array vanno specificate tutte le dimensioni, tranne eventualmente la prima.

- vettore: non serve specificare il numero di elementi
- matrice: bisogna specificare il numero di colonne, ma non serve il numero di righe

Esempi ed **esercizi** sulle matrici: Usando opportune funzioni realizzare per ciascuno dei punti seguenti un programma che legge una matrice A $(M \times N)$ e

- stampa l'elemento massimo con i suoi indici di riga e colonna
  Soluzione: file `array/matmax.c`
- costruisce il vettore degli elementi massimi di ogni riga, e lo stampa
  Soluzione: file `array/matvetma.c`
- verifica se è diagonale (in questo caso $M = N$) (ovvero, $A_{ij} = 0$ quando $i \neq j$)
  Soluzione: file `array/matdiago.c`
- verifica se è simmetrica (ovvero, $A_{ij} = A_{ji}$)
  Soluzione: file `array/matsimm.c`
- calcola la matrice $N \times M$ trasposta
  Soluzione: file `array/mattrasp.c`

Esercizio: Fornire versioni di `array/matdiago.c` e `array/matsimm.c` in cui l'uscita dai cicli viene anticipata usando una variabile booleana e/o l'istruzione `break`.

Array dinamici

Un vettore è un puntatore costante e possiamo assegnarlo ad una variabile di tipo puntatore.

Esempio:
```
int vet[5];
int *pi = vet;
pi[3] = 18;
```

Invece di far puntare `pi` ad un vettore allocato attraverso una dichiarazione (quindi statico o sullo stack), possiamo farlo puntare ad una zona di memoria allocata dinamicamente:

Esempio:
```
int *pi, dim;
scanf("%d", &dim);
pi = malloc(dim * sizeof(int));
pi[dim-1] = 20;
```

Esempio: Progettare un programma che legge un intero N, alloca dinamicamente un vettore di N interi, legge N interi da tastiera e stampa gli N interi in ordine inverso.
Implementazione: file `array/allocdin.c`

Esercizio: È necessario gestire l'archivio di un deposito di autobus. Ciascun autobus è identificato da un codice numerico (di tipo `int`).

- Quando un autobus arriva al deposito, il suo codice deve essere inserito nell'archivio nella prossima posizione libera.
- Quando un autobus parte, il suo codice deve essere cancellato dall'archivio, e i codici dei rimanenti autobus devono essere scalati di una posizione, in modo da mantenere l'ordine di arrivo degli autobus.

Progettare un programma per la gestione dell'archivio, in cui l'interfaccia permetta di gestire:

- l'arrivo di un nuovo autobus (evitando duplicati nell'archivio);
- la partenza di un autobus;
- la stampa della lista di autobus nel deposito (in ordine di arrivo).

Si realizzi l'archivio tramite un vettore dinamico.

- Inizialmente l'archivio può contenere i codici di al più 5 autobus.
- Se è necessario inserire un nuovo autobus e l'archivio è pieno, allocare dinamicamente un nuovo vettore di dimensione doppia, ricopiare il contenuto del vecchio vettore nel nuovo, e deallocare il vecchio vettore.
- Se è necessario cancellare un autobus dall'archivio e l'archivio è pieno per meno di un terzo della sua capacità, allocare dinamicamente un nuovo vettore di metà dimensione, ricopiare il contenuto del vecchio vettore nel nuovo, e deallocare il vecchio vettore.

Vettori di puntatori

Gli elementi di un vettore possono essere puntatori.

Esempio: `char * vet1[20];` ... vettore di 20 puntatori a `char`
è equivalente a `char * (vet2[20]);` ... vettore di 20 puntatori a `char`
che è diverso da `char (* vet3)[20];` ... puntatore a vettore di 20 `char`

Esempio:

```
printf("%d %d\n", sizeof(vet1), sizeof(*vet1));     stampa 80 4
printf("%d %d\n", sizeof(vet2), sizeof(*vet2));     stampa 80 4
printf("%d %d\n", sizeof(vet3), sizeof(*vet3));     stampa 4 20
```

Nota: Ogni elemento di `vet1` (o `vet2`) è un puntatore a `char`.

Tipicamente i puntatori a `char` si usano per le stringhe.

Una **stringa** è un vettore di caratteri, terminato dal carattere `'\0'` (ovvero, in cui l'ultimo elemento da considerare per una generica operazione è il primo che si incontra contenente il carattere `'\0'`).

Qualificatore `const`

Il qualificatore `const` permette di specificare che una variabile deve essere **costante**, ovvero non può essere modificata dal programma.

N.B. Questo è diverso dalle costanti simboliche definite con `#define`.

Esempio:

```
const double pi = 3.1415;
pi = 5.2;
```

$\Longrightarrow$ dà errore di compilazione

Bisogna fare attenzione a dove si mette `const`:

```
int * const pi;   ... (puntatore costante) ad int
const int *pi;    ... puntatore ad (int costante)
```

Ha importanza soprattutto nel passaggio dei parametri alle funzioni.

Esempio: `void stampaValore(const int x) { ... }`
$\Longrightarrow$ la funzione non può modificare il parametro `x`.

Usare `const` permette al compilatore di rilevare errori di programmazione (dovuti alla modifica errata di un parametro).

Modi di utilizzare `const` quando si hanno parametri

- passaggio per valore: due modi
  - modificabile (senza `const`): la funzione può alterare la sua copia locale del valore passato come parametro
  - con `const`: la funzione non può alterare la sua copia locale del valore passato come parametro
- passaggio per indirizzo: quattro modi

  puntatore modificabile/costante a dati modificabili/costanti

Cosa dobbiamo usare?

Si applica il **criterio del minimo privilegio**: Si permette alla funzione di fare solo quello che deve, e niente di più.

1. Puntatore modificabile a dati modificabili

Dopo aver risolto il riferimento i dati possono essere modificati ed il puntatore può essere modificato per puntare ad altri dati.

Esempio: Stringa che deve essere modificata, ed in cui faccio l'incremento del puntatore per scorrere gli elementi.

```
void inMaiuscole(char *s)
{
  while (*s != '\0') {
    if (*s >= 'a' && *s <= 'z')
      *s = *s + 'A' - 'a';
    s++;
  }
}
```


2. Puntatore modificabile a dati costanti

Gli elementi del vettore non possono essere modificati, ma il puntatore può essere incrementato per scandire gli elementi.

Esempio: Stampa di una stringa.

```
void stampaStringa(const char *s)
{
  while (*s != '0') {
    putchar(*s);
    s++;
  }
}
```

Esercizio: Conteggio del numero di occorrenze di un carattere in una stringa.

3. Puntatore costante a dati modificabili

Questo è il caso dei vettori dichiarati con `[]` passati alle funzioni (un vettore è un puntatore constante), nel caso in cui **vogliamo** permettere la modifica degli elementi del vettore.

Esempio: Lettura di un vettore da tastiera.

```
void leggiVettore(int * const vet, int dim)
{
  int i;
  for (i = 0; i < dim; i++)
    scanf("%d\n", &vet[i]);
}
```


4. Puntatore costante a dati costanti

Questo è il caso dei vettori dichiarati con `[]` passati alle funzioni (un vettore è un puntatore constante), nel caso in cui **non vogliamo** permettere la modifica degli elementi del vettore.

Esempio: Stampa degli elementi di un vettore.

```
void stampaVettore(const int * const vet, int dim)
{
  int i;
  for (i = 0; i < dim; i++)
    printf("%d\n", vet[i]);
}
```

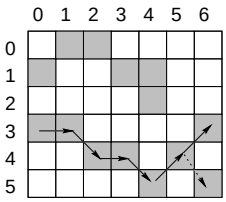

Esempio: Attraversamento di una palude.

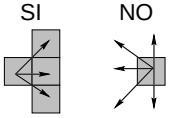
Una palude è rappresentata come una zona di $R \times C$ aree quadrate, ciascuna delle quali è – un'area di terra, o
– un'area di acqua.

Vogliamo attraversare la palude dal bordo sinistro a quello destro.

Sviluppare un programma che, usando una mappa della palude, trova un cammino attraverso la palude (**attraversamento**):

- solo su zone di terra
- assunzione semplificativa: ogni passo deve fare avanzare verso il bordo destro





Un attraversamento è un percorso

- da una zona nella colonna 0
- ad una zona nella colonna $C - 1$
- spostandosi ad ogni passo di una colonna in avanti.

Es.:	colonna:	0	1	2	3	4	5	6
	riga:	3	3	4	4	5	4	3

Strutture di dati

- mappa della palude: matrice $R \times C$ di 0 (acqua) o 1 (terra).
- cammino: vettore di C elementi, ognuno dei quali è un indice di riga (tra 0 e $R-1$)
  Passaggio per la zona (i, j) è rappresentato memorizzando il valore i nella componente di indice j del vettore.

  Es.:

colonna:	0	1	2	3	4	5	6
riga:	3	3	4	4	5	4	3

Algoritmo di ricerca di un attraversamento

- Utilizziamo una funzione ricorsiva che cerca un attraversamento a partire da una generica posizione (i, j).
- Per cercare un attraversamento della palude si deve cercarlo a partire dalle posizioni della prima colonna, ovvero $(0, 0), \ldots, (R-1, 0)$.

algoritmo cerca un attraversamento a partire dalla posizione (i, j)
  if (i, j) non è di terra
  then non esiste un attraversamento a partire da (i, j)
  else aggiungi (i, j) al cammino corrente
    if $j = C - 1$
    then l'attraversamento è stato trovato (passo base)
    else cerca un attraversamento a partire da una delle posizioni raggiungibili
      da (i, j), ovvero $(i-1, j+1)$ (solo se $i > 0$)
      $(i, j+1)$
      $(i+1, j+1)$ (solo se $i < R-1$)

algoritmo cerca un attraversamento della palude
  inizializza i a 0
  while l'attraversamento non è stato trovato e i è minore di R
  do cerca l'attraversamento a partire da $(i, 0)$
    $i \leftarrow i + 1$
  if l'attraversamento è stato trovato **then** restituisci il cammino trovato
  else restituisci che non esiste un attraversamento

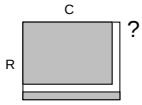
Implementazione: file `ricorsio/palude.c`

Visualizziamo l'evoluzione della pila dei record di attivazione per la palude di esempio.

- gli unici parametri di `cercaCammino` che cambiano sono `i` e `j`
- `palude` e `cammino` sono array, e quindi passati implicitamente per indirizzo

Osservazione: ogni zona di terra può venire esplorata più volte $\Longrightarrow$ inefficiente

Es.: Come si comporta il precedente programma sulla palude ?
  Molto male: $\sim R \cdot 3^C$ attivazioni ricorsive



Come si può ottimizzare, evitando di esplorare più volte la stessa posizione?
È sufficiente aggiungere prima delle chiamate ricorsive: `palude[i][j] = 0;`
Corrisponde a marcare una posizione già esplorata (come se fosse acqua) in modo che successivamente non venga più considerata (N.B. viene modificata la matrice originale).

Esercizio: Permettere anche movimenti nelle altre direzioni:

- è indispensabile marcare le posizioni già visitate in modo da evitare cicli infiniti
- serve un altro modo per rappresentare un cammino (può essere più lungo di C)

Stringhe

Una stringa è un vettore di caratteri.

Contiene la sequenza di caratteri che forma la stringa, seguita dal **carattere speciale di fine stringa**: `'\0'`.

Esempio: `char stringa1[10] = {'p', 'i', 'p', 'p', 'o', '\0'};`

Il seguente vettore di caratteri non è una stringa perchè non termina con `'\0'`.

Esempio: `char non_stringa1[2] = {'p', 'i'};`

Inizializzazione di stringhe

Una stringa può anche essere **inizializzata** utilizzando una **stringa letterale**:

Esempio:
```
char stringa2[] = "pippo";     oppure
char stringa2[6] = "pippo";
```

`stringa2` è un array **statico** di 6 caratteri: `'p'`, `'i'`, `'p'`, `'p'`, `'o'`, `'\0'`.

È possibile memorizzare una stringa in un **array di caratteri dinamico**.

Esempio: `char *buffer = malloc(80*sizeof(char));`

In questo caso (come per tutti gli array dinamici) **non** possiamo inizializzare l'array contestualmente alla sua creazione.

Possiamo anche "assegnare" ad una stringa una stringa letterale.

Esempio:
```
char *buffer2;
buffer2 = "pippo";
```

Con questa assegnazione abbiamo assegnato a `buffer2`, di tipo `char*`, la stringa costante `"pippo"`, di tipo `char*` costante.
$\Longrightarrow$ `buffer2` punta al primo carattere della **stringa costante** `"pippo"`.

L'istruzione `buffer2[0] = 't';` **non** dà errore di compilazione.

Dà però **errore in esecuzione** poiché stiamo cercando di cambiare un carattere dichiarato costante.

N.B. questo è diverso da
```
char buffer3[] = "pippo";      che è equivalente a
char buffer3[6] = "pippo";     che è equivalente a
char buffer3[] = {'p', 'i', 'p', 'p', 'o', '\0'};
```
In questo caso la stringa costante `"pippo"` viene usata per inizializzare il vettore `buffer3`.

Inizializzazione di un **vettore di stringhe**:

Esempio: `char *colori[4] = {"rosso", "giallo", "verde", "blu"};`

È un vettore di quattro puntatori a quattro stringhe costanti (di lunghezza $6, 7, 6, 4$).

È equivalente ad inizializzare i quattro puntatori separatamente:

```
char *colori[4];
colori[0] = "rosso";     colori[1] = "giallo";
colori[2] = "verde";     colori[3] = "blu";
```

Ingresso/uscita di stringhe

Stampa di una stringa: si deve utilizzare la specifica di formato "`%s`".

Esempio:
```
printf("%s\n", stringa1);
printf("%s\n", stringa2);
printf("%s\n", buffer2);
```

Vengono stampati tutti i caratteri fino al primo `'\0'` escluso.

Esempio: `printf("%s\n", non_stringa1);` stampa: `pi^D^H"^D^H"^D^A...`

Lettura di una stringa: si deve utilizzare la specifica di formato "`%s`".

Esempio:
```
char buffer[40];
scanf("%s", buffer);
```

1. Vengono letti da input i caratteri in sequenza fino a trovare il primo carattere di spaziatura (spazio, tabulazione, interlinea, ecc.).
2. I caratteri letti vengono messi dentro il vettore `buffer`.
3. Al posto del carattere di spaziatura, viene posto il carattere `'\0'`.

Note:
- il vettore **deve** essere sufficientemente grande da contenere tutti i caratteri letti
- non si usa `&buffer` ma direttamente `buffer` (questo perché `buffer` è di tipo `char*`, ovvero è già un indirizzo)

Manipolazione di stringhe

Per manipolare una stringa si deve accedere ai singoli caratteri singolarmente.

Esempio:
```
for (i = 0; buffer[i] != '\0'; i++) {
  /* fai qualcosa con buffer[i], ad esempio: */
  printf("%c\n", buffer[i]);
}
```

Esempio: Confronto di uguaglianza tra due stringhe.

N.B. **Non** si può usare "`==`" perché questo confronta i puntatori e non le stringhe.
$\Longrightarrow$ Si devono necessariamente scandire le due stringhe.
Implementazione: file `stringhe/strequal.c`

Esercizio: Lunghezza di una stringa.
Implementazione: file `stringhe/strlungh.c`

In realtà queste funzioni, come molte altre, sono disponibili nella libreria sulle stringhe.

Caratteri e stringhe

Caratteri e stringhe sono diversi e **non** vanno confusi:

- un carattere è in realtà un intero
  per denotare una costante di tipo carattere: `'x'`
- una stringa è un vettore di caratteri che termina con il carattere `'\0'`
  per denotare costanti di tipo stringa: `"un esempio di stringa"`
- una variabile di tipo stringa è in realtà un puntatore al primo carattere del vettore

Esempio:

`char c = 'a';`	carattere
`char *s = "a";`	puntatore alla stringa costante `"a"`
`char v[] = "a";`	vettore di 2 caratteri inizializzato a `{'a', '\0'}`

```
printf("%d %d %d\n", sizeof(c), sizeof(s), sizeof(v));
```

stampa:
```
1 4 2
```


Funzioni di libreria per la gestione dei caratteri

Per usare le funzioni per la gestione dei caratteri **è necessario**: `#include <ctype.h>`

Tutte le funzioni operano sui caratteri e su `EOF`.
Però: `char` può essere `unsigned`, mentre `EOF` normalmente è -1.

$\Longrightarrow$ Tutte le funzioni hanno parametri di tipo `int` (e non di tipo `char`).
Però possono prendere argomenti di tipo `char`, poiché `char` viene promosso ad `int`..

Funzioni di conversione maiuscolo - minuscolo

`int tolower(int c);`	se `c` è una lettera maiuscola, la converte in minuscolo altrimenti restituisce `c` invariata
`int toupper(int c);`	se `c` è una lettera minuscola, la converte in maiuscolo altrimenti restituisce `c` invariata

Funzioni che determinano il tipo di un carattere

- restituiscono vero (1) se il carattere è tra quelli indicati
- restituiscono falso (0) altrimenti

`int isdigit(int c);`	cifra
`int isxdigit(int c);`	cifra esadecimale
`int isalpha(int c);`	lettera alfabetica (minuscola o maiuscola)
`int isalnum(int c);`	cifra o lettera alfabetica
`int islower(int c);`	lettera alfabetica minuscola
`int isupper(int c);`	lettera alfabetica maiuscola
`int isspace(int c);`	carattere di spazio bianco (`' '`, `'\n'`, `'\r'`, `'\t'`, `'\v'`, `'\f'`)
`int iscntrl(int c);`	carattere di controllo (tra 0 e $0x1F$ più $0x7F$ = DEL) (`'\t'`, `'\v'`, `'\a'`, `'\b'`, `'\r'`, `'\n'`, ...)
`int isprint(int c);`	carattere stampabile (incluso lo spazio)
`int isgraph(int c);`	carattere stampabile diverso dallo spazio
`int ispunct(int c);`	carattere stampabile non alfanumerico o di spazio (ovvero un carattere di punteggiatura)

Funzioni di libreria per la conversione di stringhe

È necessario: `#include <stdlib.h>`

Convertono le strighe formate da cifre in valori interi ed in virgola mobile.

Funzioni `atoi`, `atol`, `atof`

```
int atoi(const char *str);
```

- converte la stringa puntata da `str` in un `int`
- la stringa deve contenere un numero intero valido
- in caso contrario il risultato è indefinito
- spazi bianchi iniziali vengono ignorati
- il numero può essere concluso da un qualsiasi carattere che non è valido in un numero intero (spazio, lettera, virgola, punto, . . .)

Esempio: file `stringhe/strtonum.c`

`atoi("123")`	restituisce l'intero 123
`atoi("123.45")`	restituisce l'intero 123 (`".45"` viene ignorato)
`atoi("  123.45")`	restituisce l'intero 123 (`"  "` e `".45"` ignorati)


```
long atol(const char *str);
```

- analoga ad `atoi`, solo che restituisce un `long int`

```
double atof(const char *str);
```

- analoga ad `atoi` ed `atol`, solo che restituisce un `double`
- il numero può essere concluso da un qualsiasi carattere non valido in un numero in virgola mobile

Esempio: file `stringhe/strtonum.c`

`atof("123.45")`	restituisce 123.45
`atof("1.23xx")`	restituisce 1.23
`atof("1.23.45")`	restituisce 1.23

Funzioni `strtod`, `strtol`, `strtoul`

```
double strtod(const char *inizio, char **fine);
```

- converte la stringa puntata da `inizio` in un `double`
- `*fine` viene fatto puntare al primo carattere successivo alla porzione di stringa che è stata convertita
- se il secondo argomento è `NULL`, allora viene ignorato
- se la conversione non è possibile, allora `*fine` viene posto uguale ad `inizio` e viene restituito 0

Esempio: file `stringhe/strtonum.c`

```
double d;
char *stringa = "123.45Euro";
char *resto;

d = strtod(stringa, &resto);
printf("%g\n", d);
printf("%s\n", resto);
```

Stampa

```
123.45
Euro
```

```
long strtol(const char *inizio, char **fine, int base);
```

- converte la stringa puntata da `inizio` in un `long`
- la stringa deve contenere un numero nella `base` specificata
- la `base` deve essere tra 2 e 36, oppure 0:
  - se è 0, la base da usare è determinata in base alla notazione per le costanti (ottale, decimale, esadecimale)
  - se è > 10, le cifre tra 10 e 35 sono rappresentate dalle lettere A–Z (oppure a–z)
- per il resto come `strtod`

```
unsigned long strtoul(const char *inizio, char **fine, int base);
```

- analoga a `strtol`, solo che restituisce un `unsigned long`

Esempio: file `stringhe/strtonum.c`

```
long i;
char *resto;

i = strtol("12000lire", &resto, 10);
printf("%ld\n", i);
printf("%s\n", resto);

i = strtol("FFGFF", &resto, 16);
printf("%ld\n", i);
printf("%s\n", resto);

i = strtol("0xFFGFF", &resto, 0);
printf("%ld\n", i);
printf("%s\n", resto);
```

Stampa

```
12000
lire

255
GFF

255
GFF
```

Esempio: Conversione di tutti i numeri in una stringa e memorizzazione in un vettore.

Implementazione: file `stringhe/strnums.c`

Funzioni di libreria per l'input/output di stringhe e caratteri

È necessario: `#include <stdio.h>`

Input e output di caratteri

```
int getchar(void);
```

- legge il prossimo carattere da standard input e lo restituisce

```
int putchar(int c);
```

- manda `c` in standard output
- restituisce `EOF` (-1) se si verifica un errore

Input e output di stringhe

`char *gets(char *str);`

- legge i caratteri da standard input e li inserisce nel vettore `str`
- la lettura termina con un `'\n'` o un `EOF`, che **non** viene inserito nel vettore
- la stringa nel vettore viene terminata con `'\0'`
- **Attenzione:** non c'è alcun modo di limitare la lunghezza della sequenza immessa
  $\Longrightarrow$ vettore allocato potrebbe non bastare

`int puts(const char *str);`

- manda la stringa `str` in standard output (inserisce un `'\n'` alla fine)

Input e output di stringhe da/su stringhe

`int sprintf(char *str, const char *format, ...);`

- come `printf`, solo che invece di scrivere su standard output, scrive nel vettore `str`

`int sscanf(char *str, const char *format, ...);`

- come `scanf`, solo che invece di leggere da standard input, legge dalla stringa `str`

Funzioni di libreria per la manipolazione di stringhe

È necessario: `#include <string.h>`

Funzioni di copia

`char *strcpy(char *dest, const char *src);`

- copia la stringa `src` nel vettore `dest`
- restituisce il valore di `dest`
- `dest` dovrebbe essere sufficientemente grande da contenere `src`
- se `src` e `dest` si sovrappongono, il comportamento di `strcpy` è indefinito

Esempio:

```
char a[10], b[10];
char *x = "Ciao";
char *y = "mondo";
strcpy(a, x);   strcpy(b, y);
puts(a);        puts(b);
```

Stampa:

```
Ciao
mondo
```


`char *strncpy(char *dest, const char *src, size_t n);`

- copia un massimo di `n` caratteri della stringa `src` nel vettore `dest` (`size_t` è il tipo del valore restituito da `sizeof`, ovvero `unsigned long` o `unsigned int`)
- restituisce il valore di dest
- **Attenzione:** il carattere `'\0'` finale di `src` viene copiato **solo** se `n` è $\geq$ della lunghezza di `src+1`

Esempio:

```
char a[10], b[10];
char *x = "Ciao";
char *y = "mondo";

strncpy(a, x, 5);
puts(a);                    stampa: Ciao
strncpy(b, y, 5);           b non è terminata da '\0'
b[5] = '0';
puts(b);                    stampa: mondo
```

Funzioni di concatenazione

```
char *strcat(char *dest, const char *src);
```

- accoda la stringa `src` a quella nel vettore `dest` (il primo carattere di `src` si sostituisce al `'\0'` di `dest`) e termina `dest` con `'\0'`
- restituisce il valore di `dest`
- `dest` dovrebbe essere sufficientemente grande da contenere tutti i caratteri di `dest`, di `src`, ed il `'\0'` finale
- se `src` e `dest` si sovrappongono, il comportamento di `strcat` è indefinito

Esempio:

```
char a[10], b[10];
char *x = "Ciao",  *y = "mondo";
strcpy(a, x);
strcat(a, y);
puts(a);
```

stampa: `Ciaomondo`

```
char *strncat(char *dest, const char *src, size_t n);
```

- accoda al più `n` caratteri di `src` alla stringa nel vettore `dest` (il primo carattere di `src` si sostituisce al `'\0'` di `dest`)
- termina in ogni caso `dest` con `'\0'`

Funzioni di confronto

```
int strcmp(const char *s1, const char *s2);
```

- confronta le stringhe `s1` ed `s2`
- restituisce:
  - 0 se `s1` $=$ `s2`
  - un valore < 0 se `s1` $<$ `s2`
  - un valore > 0 se `s1` $>$ `s2`
- il confronto è quello lessicografico: i caratteri vengono confrontati uno ad uno, ed il primo carattere diverso (o la fine di una delle due stringhe) determina il risultato
- per il confronto tra due caratteri viene usato il codice

Esempio:

```
char *s1 = "abc";
char *s2 = "abx";
char *s3 = "abc altro";

printf("%d\n", strcmp(s1, s1));
printf("%d\n", strcmp(s1, s2));
printf("%d\n", strcmp(s1, s3));
printf("%d\n", strcmp(s2, s1));
```

Stampa:

```
0
-1
-1
1
```


Attenzione: per verificare l'uguaglianza bisogna confrontare il risultato con 0

Esempio:

```
if (strcmp(s1, s2) == 0)
  printf("uguali\n");
else
  printf("diverse\n");
```

```
int strncmp(const char *s1, const char *s2, size_t n);
```

- confronta al più `n` caratteri di `s1` ed `s2`

Esempio:

```
char *s1 = "abc";
char *s3 = "abc altro";

printf("%d\n", strncmp(s1, s3, 3));
```

Stampa `0`, ovvero le due stringhe sono uguali.

Altre funzioni di libreria sulle stringhe

- lunghezza di una stringa (pari al numero di caratteri che precedono `'\0'`)
  `size_t strlen(const char *str);`
- di ricerca (di caratteri, di sottostringhe, ...)
- di manipolazione della memoria

Esercizio: Fornire un'implementazione delle funzioni di libreria su caratteri e stringhe.

Esercizio: Programma che legge da tastiera quattro stringhe che rappresentano interi, le converte in interi, le somma, e stampa la somma.

Esercizio: Programma che legge una data nel formato "`12/05/2002`" e la converte nel formato "`12 maggio 2002`".
Implementazione: file `stringhe/convdata.c`

Esercizio: Programma che legge un numero tra 0 e $999\,999$ e lo stampa in lettere.

Esercizio: Modificare il programma per la gestione del deposito di autobus tenendo conto che ogni autobus è identificato da un nome invece che da un codice numerico.

Programmi con argomenti passati da linea di comando

Quando compiliamo un programma viene generato un file eseguibile (estensione `.EXE` in Windows). I programmi possono prendere degli argomenti dalla riga di comando con la quale vengono eseguiti.
Esempio: `copy file1 file2`

I programmi visti finora non prendevano argomenti. La funzione `main` non prendeva parametri: `int main(void) { ... }`

Per poter usare gli argomenti con cui il programma è stato lanciato, la definizione di `main` deve diventare:

```
int main(int argc, char *argv[]) { ... }
```

- `argc` ... numero di argomenti con cui il programma è stato lanciato più 1
- `argv` ... vettore di `argc` stringhe che compongono la riga di comando
  `argv[0]` è il nome del programma stesso
  `argv[1]`, ..., `argv[argc-1]` sono gli argomenti (separati da spazio)
- ogni `argv[i]` è una stringa (terminata da `'\0'`)
- `argc` e `argv` vengono inizializzati dal sistema operativo (le stringhe di `argv` sono allocate dinamicamente)

Esempio: Stampa del numero di argomenti ricevuti da linea di comando.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("Ho ricevuto %d argomenti\n", argc-1);
  return 0;
}
```

Compilando si ottiene `argnum.exe`, che può essere eseguito:

```
> argnum primo secondo terzo
> Ho ricevuto 3 argomenti
```

Esempio: Stampa degli argomenti ricevuti da linea di comando.
Implementazione: file `stringhe/argprint.c`

Esempio: Stampa della somma di due numeri interi passati come argomenti.

N.B. I due numeri sono passati come stringhe e non come numeri.
$\Longrightarrow$ Vanno convertiti in numeri usando `atoi`.
Implementazione: file `stringhe/argsum.c`

Esempio: Eliminazione dell'estenzione dal nome di un file passato come argomento.

- il nome di un file è una stringa che può contenere un `'.'`
- vogliamo mantenere solo la parte che precede il `'.'` e memorizzarla in una nuova stringa
- il programma deve venire lanciato con un singolo argomento, che è la stringa da cui togliere l'estensione: possiamo accedere alla stringa con `argv[1]`
  i caratteri della stringa sono `argv[1][i]`
- per poter eliminare l'estensione, scandiamo la stringa
  - usiamo un indice `i` per la scansione
  - ci fermiamo quando incontriamo `'.'` oppure `'\0'`
    $\Longrightarrow$ la condizione di terminazione del ciclo deve essere:
    `argv[1][i] != '\0' && argv[1][i] != '.'`
- per memorizzare la stringa senza estensione usiamo un array dinamico

Implementazione: file `stringhe/argnoext.c`

Esercizio: Modificare i programmi visti a lezione che leggono dati da input, in modo che i dati vengano passati come argomenti al programma invece che letti da input.

Strutture e File

Strutture

Una **struttura** (o **record**) serve per **aggregare** elementi (anche di tipo diverso) sotto un unico nome.

Sono un tipo di dato **derivato**, ovvero costruito a partire da dati di altri tipi.

Definizione di una struttura

Esempio:
```
struct data {
  int giorno;
  int mese;
  int anno;
};
```

- parola chiave `struct` introduce la definizione della struttura
- `data` è l'**etichetta** della struttura, che attribuisce un nome alla definizione della struttura
- `giorno`, `mese`, `anno` sono i **campi** (o **membri**) della struttura

Campi di una struttura

- devono avere nomi univoci all'interno di una struttura
- strutture diverse possono avere campi con lo stesso nome
- i nomi dei campi possono coincidere con nomi di variabili o funzioni

  Esempio:
  ```
  int x;
  struct a { char x; int y; };
  struct b { int w; float x; };
  ```
- possono essere di tipo diverso (semplice o altre strutture)
- un campo di una struttura non può essere del tipo struttura che si sta definendo

  Esempio:
  ```
  struct s { int a;
             struct s next;  };
  ```
- un campo può però essere di tipo puntatore alla struttura

  Esempio:
  ```
  struct s { int a;
             struct s *next;  };
  ```
- la definizione della struttura non provoca allocazione di memoria, ma introduce un nuovo tipo di dato

Dichiarazione di variabili di tipo struttura

Esempio: `struct data oggi, appelli[10], *pd;`

- `oggi` è una variabile di tipo `struct data`
- `appelli` è un vettore di 10 elementi di tipo `struct data`
- `pd` è un puntatore a una `struct data`

Una variabile di tipo struttura può essere dichiarata contestualmente alla definizione della struttura.

Esempio:
```
struct studente {
  char nome[20];
  long matricola;
  struct data ddn;
} s1, s2;
```

In questo caso si può anche **omettere l'etichetta** di struttura.

Operazioni sulle strutture

Si possono assegnare variabili di tipo struttura a variabili **dello stesso tipo** struttura.

Esempio:
```
struct data d1, d2;
...
d1 = d2;
```

Nota: questo permette di assegnare interi vettori.

Esempio:
```
struct matrice { int elementi[10][10]; };
struct matrice a, b;
...
a = b;
```

Non è possibile effettuare il confronto tra due variabili di tipo struttura.

Esempio:
```
struct data d1, d2;
if (d1 == d2) ...
```
Errore!

Motivo: una struttura può contenere dei "**buchi**" dovuti alla necessità di allineare i campi con le parole di memoria.

L'equivalenza di tipo tra strutture è **per nome**.

Esempio:
```
struct s1 { int i; };
struct s2 { int i; };
struct s1 a, b;
struct s2 c;

a = b;      OK
a = c;      Errore, perché a e c non sono dello stesso tipo.
```

Si può ottenere l'indirizzo di una variabile di tipo struttura tramite l'operatore `&`.

Si può rilevare la dimensione di una struttura con `sizeof`.

Esempio: `sizeof(struct data)`

Attenzione: **non** è detto che la dimensione di una struttura sia pari alla somma delle dimensioni dei singoli campi (ci possono essere **buchi**).

Accesso ai campi della struttura

Avviene tramite l'**operatore punto**

Esempio:
```
struct data oggi;
oggi.giorno = 8;    oggi.mese = 5;   oggi.anno = 2002;
printf("%d %d %d", oggi.giorno, oggi.mese, oggi.anno);
```

Accesso tramite un puntatore alla struttura.

Esempio:
```
struct data *pd;
pd = malloc(sizeof(struct data));
(*pd).giorno = 8;   (*pd).mese = 5;   (*pd).anno = 2002;
```

N.B. Ci vogliono le `()` perché "`.`" ha priorità più alta di "`*`".

Operatore freccia: combina il dereferenziamento e l'accesso al campo della struttura.

Esempio:
```
struct data *pd;
pd = malloc(sizeof(struct data));
pd->giorno = 8;   pd->mese = 5;   pd->anno = 2002;
```


Inizializzazione di strutture

Può avvenire, come per i vettori, con un elenco di inizialzzatori.

Esempio:
```
struct data oggi = { 8, 5, 2002 };
```

Se ci sono meno inizializzatori di campi della struttura, i campi rimanenti vengono inizializzati a 0 (o `NULL`, se il campo è un puntatore).

Variabili di tipo struttura dichiarate esternamente alle definizioni di funzione vengono inizializzate a 0 per default.

Passaggio di parametri di tipo struttura

È come per i parametri di tipo semplice:

- il passaggio è per valore $\Longrightarrow$ viene fatta una **copia dell'intera struttura** dal parametro attuale a quello formale
- è possibile effettuare anche il passaggio per indirizzo

Per passare per valore ad una funzione un vettore (il vettore, non il puntatore iniziale) è sufficiente racchiuderlo in una struttura.

Una funzione può restituire un valore di tipo struttura.

Esempio: file `tipi/structvet.c`

typedef

Attraverso `typedef` il C permette di creare dei sinonimi di tipi definiti in precedenza.

Esempio:
```
struct data { int giorno, mese, anno; };
typedef struct data Data;
Data d1, d2;
Data appelli[10];
```

`Data` è un sinonimo di `struct data`.

N.B. `typedef` non crea un nuovo tipo, ma un **nuovo nome di tipo**, che può essere usato come sinonimo.

Esempio: file `tipi/complex.c`

In generale, una `typedef` ha la forma di una dichiarazione di variabile preceduta dalla parola chiave `typedef`, e con il nuovo nome di tipo al posto del nome della variabile.

Esempio:
```
typedef Data Appelli[10];
typedef int *PuntInt;
```


Elaborazione dei file

Un file è una **sequenza di byte** (o caratteri) memorizzata su disco (memoria di massa), alla quale si accede tramite un **nome**.

I byte corrispondono a dei dati di tipo `char`, `int`, `double`, ecc. (esattamente come la sequenza di byte che immettiamo da input o leggiamo da tastiera).

I file vengono gestiti dal sistema operativo, e il programma deve invocare le funzioni del SO per accedere ai file: viene fatto dalla libreria standard di input/output.

Principali operazioni sui file:

- lettura da file
- scrittura su file
- apertura di file: comunica al SO che il programma sta accedendo al file
- chiusura di file: comunica al SO che il programma rilascia il file

Per ogni programma vengono aperti automaticamente tre file:

- `stdin` (sola lettura): lettura è da tastiera
- `stdout` (sola scrittura): scrittura è su video
- `stderr` (sola scrittura): per messaggi di errore (scritti su video)

Apertura di un file: tramite la funzione `fopen`

Deve essere effettuata prima di poter operare su un qualsiasi file.

Esempio:
```
FILE *fp;
fp = fopen("pippo.dat", "r");
```

`FILE` è un tipo struttura definito in `stdio.h`.

Dichiarazione di `fopen`: `FILE * fopen(char *nomefile, char *modo);`

Parametri di `fopen`:

- il nome del file
- la modalità di apertura:
  - `"r"` (read) . . . sola lettura
  - `"w"` (write) . . . crea un nuovo file per la scrittura; se il file esiste già ne elimina il contenuto corrente
  - `"a"` (append) . . . crea un nuovo file, o accoda ad uno esistente per la scrittura
  - `"r+"` . . . apre un file per l'aggiornamento (lettura e scrittura), a partire dall'inizio del file
  - `"w+"` . . . crea un nuovo file per l'aggiornamento; se il file esiste già ne elimina il contenuto corrente
  - `"a+"` . . . crea un nuovo file, o accoda ad uno esistente per l'aggiornamento

`fopen` restituisce un puntatore ad una struttura di tipo `FILE`

- la struttura mantiene le informazioni necessarie alla gestione del file
- il puntatore (detto **file pointer**) viene utilizzato per accedere al file
- se si verifica un errore in fase di apertura, `fopen` restituisce `NULL`

Esempio:
```
FILE *fp;
if ((fp = fopen("pippo.dat", "r")) == NULL) {
  printf("Errore!\n");
  exit(1);
} else ...
```

Chiusura di un file: tramite `fclose`, passando il file pointer

Esempio: `fclose(fp);`

Dichiarazione di `fclose`: `int fclose(FILE *fp);`

Se si è verificato un errore, `fclose` restituisce `EOF` (-1).

La chiusura va **sempre** effettuata, appena sono terminate le operazioni di lettura/scrittura da effettuare sul file.

Scrittura su e lettura da file

Consideriamo solo file ad **accesso sequnenziale**: si legge e si scrive in sequenza, senza poter accedere agli elementi precedenti o successivi.

Scrittura su file: tramite `fprintf`

Dichiarazione di `fprintf`: `int fprintf(FILE *fp, char *formato, ...);`

- come `printf`, tranne che per `fp` (`printf(...);` è equivalente a `fprintf(stdout, ...);`)
- scrive sul file identificato da `fp` a partire dalla posizione corrente
- il file deve essere stato aperto in scrittura, append, o aggiornamento
- in caso di errore restituisce `EOF`, altrimenti il numero di byte scritti

Esempio:
```
int ris1, ris2;
FILE *fp;
if ((fp = fopen("risultati.dat", "w")) != NULL) {
  ris1 = ...;   ris2 = ...;
  fprintf(fp, "%d %d\n", ris1, ris2);
  fclose(fp);
}
```


Lettura da file: tramite fscanf

Dichiarazione di `fscanf`: `int fscanf(FILE *fp, char *formato, ...);`

- come `scanf`, tranne che per `fp` (`scanf(...);` è equivalente a `fscanf(stdin, ...);`)
- legge dal file identificato da `fp` a partire dalla posizione corrente
- il file deve essere stato aperto in lettura o aggiornamento
- restituisce il numero di assegnamenti fatti agli argomenti specificati nell'attivazione dopo la stringa di formato
- se il file termina o si ha un errore prima del primo assegnamento, restituisce `EOF`

Verifica di fine file: si può usare `feof`

Dichiarazione di `feof`: `int feof(FILE *fp);`

Restituisce vero (1) se per il file è stato impostato l'**indicatore di end of file**, ovvero:

- quando si tenta di leggere oltre la fine del file
- per lo standard input, quando viene digitata la combinazione di tasti
  - `ctrl-z` (per Windows)
  - `return ctrl-d` (per Unix)

Esempio: Conteggio del numero di caratteri e di linee in un file.

Implementazione: file `file/contaclf.c`

Esercizio: Conteggio del numero di parole in un file (uno o più spazi bianchi e/o `'\n'` sono equivalenti ad uno spazio bianco).

Soluzione: file `file/contapar.c`

Esempio: Calcolare la somma degli interi in un file.

```
int somma = 0, n;
FILE *fp;

if ((fp = fopen("pippo.txt", "r")) != NULL) {
  while (fscanf(fp, "%d", &n) == 1)
    somma += n;
  fclose(fp);
}
```


Esempio: Creazione di una copia di un file.

```
char ch;
FILE * fpr, *fpw;

if (((fpr = fopen("in.txt", "r")) != NULL) &&
    ((fpw = fopen("out.txt", "w")) != NULL))  {
  while (fscanf(fpr, "%c", &ch) == 1)
    fprintf(fpw, "%c", ch);
  fclose(fpr);
  fclose(fpw);
}
```

Esempio: Simulare 100 lanci di due dadi, memorizzando i risultati in un file. Leggere poi i risultati dal file, calcolarne e stamparne la media, e stampare sul file `frequenze.txt` le frequenze dei risultati dei lanci.

Implementazione: file `file/duedadif.c`

Esercizio: Calcolare le frequenze dei caratteri alfabetici in un file il cui nome è letto da tastiera.

Le liste collegate

Rappresentazione di liste

È molto comune dover rappresentare **sequenze di elementi** tutti dello stesso tipo e fare operazioni su di esse.

Esempi: sequenza di interi $(23\ 46\ 5\ 28\ 3)$
sequenza di caratteri `('x' 'r' 'f')`
sequenza di persone con nome e data di nascita

1. Rappresentazione sequenziale: tramite **array** (statici o dinamici)

Vantaggi:

- accesso agli elementi è diretto (tramite indice) ed efficiente
- l'ordine degli elementi è quello in memoria $\Longrightarrow$ non servono strutture dati addizionali che occupano memoria
- è semplice manipolare l'intera struttura (copia, ordinamento, ...)

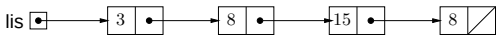
Svantaggi:

- inserire o eliminare elementi al centro è complicato ed inefficiente (bisogna spostare gli elementi che vengono dopo)

2. Rappresentazione collegata

- La sequenza di elementi viene rappresentata da una struttura di dati collegata, realizzata tramite **strutture e puntatori**.
- Ogni elemento è rappresentato con un una struttura C:
  - un campo per l'elemento (ad es. `int`)
  - un campo puntatore alla stuttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)
- L'ultimo elemento non ha un elemento successivo. $\Longrightarrow$ Il campo puntatore ha valore `NULL`.
- L'intera sequenza è **rappresentata** da un **puntatore al suo primo elemento** (vogliamo un'unica variabile per accedere a tutti gli elementi della sequenza).
  N.B. Il puntatore **non è** la sequenza, ma la rappresenta soltanto.

Graficamente:



Dichiarazioni di tipo

Esempio: Sequenze di interi.

```
struct nodoLista {
  int info;
  struct nodoLista *next;
};
typedef struct nodoLista NodoLista;
typedef NodoLista *TipoLista;
```

Esempio: Creazione di una lista di tre interi fissati: $(3, 8, 15)$
Implementazione: file `puntator/listeman.c`

```
TipoLista lis;                  /* puntatore al primo elemento della lista */

lis = malloc(sizeof(NodoLista));              /* allocazione primo elemento */
lis->info = 3;
lis->next = malloc(sizeof(NodoLista));       /* allocazione secondo elemento */
lis->next->info = 8;
lis->next->next = malloc(sizeof(NodoLista));        /* alloc. terzo elemento */
lis->next->next->info = 15;
lis->next->next->next = NULL;
```

Osservazioni:

- `lis` è di tipo `TipoLista`, quindi è un puntatore e **non** una struttura
- la zona di memoria per ogni elemento della lista (**non** per ogni variabile di tipo `TipoLista`) deve essere allocata esplicitamente con `malloc`

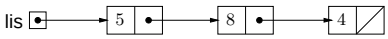
Esiste un modo più semplice di creare la lista di 3 elementi?

Esempio: Creazione di una lista di tre interi fissati, cominciando dall'ultimo: $(3, 8, 15)$

```
TipoLista aux, lis2 = NULL;

aux = malloc(sizeof(NodoLista));        /* allocazione dell'ultimo elemento */
aux->info = 15;    aux->next = lis2;
lis2 = aux;

aux = malloc(sizeof(NodoLista));        /* allocazione dell'ultimo elemento */
aux->info = 8;     aux->next = lis2;
lis2 = aux;

aux = malloc(sizeof(NodoLista));        /* allocazione dell'ultimo elemento */
aux->info = 3;     aux->next = lis2;
lis2 = aux;
```


Operazioni sulle liste

Supponiamo di aver creato una lista in memoria, con il puntatore iniziale in una variabile:



Stampa degli elementi di una lista

Vogliamo che venga stampato: `5 8 4`

Versione iterativa: file `puntator/listeman.c`

N.B. `lis = lis->next` fa puntare `lis` all'elemento successivo della lista

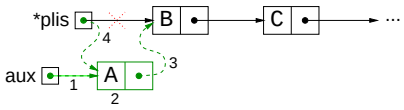
Versione ricorsiva

algoritmo stampa una lista
  if la lista non è vuota
  then stampa il dato del primo elemento
    stampa il resto della lista

Implementazione: file `puntator/listeman.c`

Inserimento di un nuovo elemento in testa

Esempio:



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegnamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista viene fatto puntare alla nuova struttura

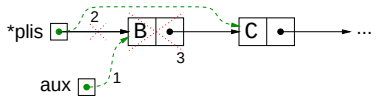
Implementazione: file `puntator/liste.c`, funzione `InserisciTestaLista`

- parametro di ingresso: elemento da inserire
- parametro di ingresso/uscita: lista $\Longrightarrow$ puntatore iniziale alla lista deve essere passato per indirizzo

Cancellazione del primo elemento

"Cancellare" significare deallocare la memoria occupata dall'elemento.

- se la lista è vuota non facciamo nulla
- altrimenti deallochiamo la struttura del primo elemento (con `free`) $\Longrightarrow$ la lista deve essere passata per indirizzo



Implementazione: file `puntator/liste.c`, funzione `CancellaPrimoLista`

```
void CancellaPrimoLista(TipoLista *plis)
{
  TipoLista aux;
  if (*plis != NULL) {
    aux = *plis;              /* 1 */
    *plis = (*plis)->next;    /* 2 */
    free(aux);                /* 3 */
  }
}
```


Cancellazione di tutta la lista

Versione iterativa

Per **esercizio**: file `puntator/liste.c`, funzione `CancellaLista`

Versione ricorsiva: file `puntator/listeric.c`, funzione `CancellaLista`

```
void CancellaLista(TipoLista *plis)
{
  TipoLista aux;
  if (*plis != NULL) {
    aux = (*plis)->next;   /* memorizza il puntatore all'elemento successivo */
    free(*plis);                              /* dealloca il primo elemento */
    CancellaLista(&aux);                    /* cancella il resto della lista */
    *plis = NULL;
  }
}
```

Attenzione: una volta fatto `free`, non possiamo più far riferimento al campo `next` della struttura.

In che ordine vengono cancellati gli elementi della lista?

Verifica se un elemento compare in una lista

Versione iterativa

Scandiamo la lista continuando fino a che:

- l'elemento non è stato ancora trovato, e
- non siamo giunti alla fine della lista (usiamo una variabile booleana `trovato`)

Implementazione: file `puntator/listecon.c`, funzione `EsisteInLista`

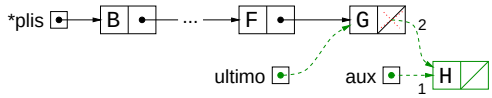
Versione ricorsiva

algoritmo verifica se *elem* compare in *lista*
  if *lista* è vuota
  then restituisci falso
  else if *elem* è il primo elemento di *lista*
    then restituisci vero
    else restituisci il risultato della verifica se *elem* compare nel resto di *lista*

Implementazione per **esercizio**: file `puntator/listeric.c`, funzione `EsisteInLista`

Inserimento di un elemento in coda — *Versione iterativa*

algoritmo inserisci *elem* in coda a *lista*
alloca una struttura per il nuovo elemento
if la *lista* è vuota
then restituisci la *lista* costituita dal solo elemento
else scandisci la *lista* fermandoti quando il puntatore corrente punta all'ultimo elem.
concatena il nuovo elemento con l'ultimo



Nota: la lista è passata per indirizzo $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione

Implementazione: file `puntator/liste.c`, funzione `InserisciCodaLista`

Inserimento di un elemento in coda — *Versione ricorsiva*

Caratterizzazione **induttiva** dell'inserimento in coda:

Sia *ris* la lista ottenuta inserendo *elem* in coda a *lis*. Allora:

1. se *lis* è la lista vuota, allora *ris* è costituita solo da *elem* (**caso base**)
2. altrimenti *ris* è ottenuta da *lis* facendo l'inserimento di *elem* in coda al resto di *lis* (**caso ricorsivo**)

Implementazione: file `puntator/listeric.c`, funzione `InserisciCodaLista`

```
void InserisciCodaLista(TipoLista *plis, TipoElemLista elem)
{
  if (*plis == NULL) {
    *plis = malloc(sizeof(NodoLista));
    (*plis)->info = elem;
    (*plis)->next = NULL;
  } else
    InserisciCodaLista(&(*plis)->next, elem);
}  /* InserisciCodaLista */
```

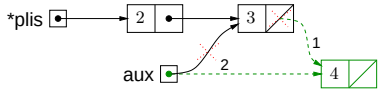
Vediamo l'evoluzione della pila dei RDA per l'inserimento di 3 in coda a $(1\ \ 2)$.

Creazione di una lista i cui elementi vengono letti da file

Vogliamo che gli elementi siano nell'ordine in cui compaiono nel file.
$\Longrightarrow$ Ogni elemento deve essere inserito in coda.

Per migliorare l'efficienza teniamo un puntatore all'ultimo elemento inserito in modo da non dover scandire tutta la lista per ogni elemento.

Esempio: contenuto del file: 2 3 4 5



Per evitare di dover trattare a parte il primo elemento usiamo la tecnica del **record generatore**:

- struct il cui campo `info` non contiene informazione significativa (record generatore viene detto anche **record fittizio**)
- viene creato e posto all'inizio della lista in modo che anche il primo elemento effettivo della lista abbia un elemento che lo precede
- prima di terminare la funzione bisogna rilasciare il record generatore

Implementazione: file `puntator/listeirw.c`, funzione `LeggiLista`

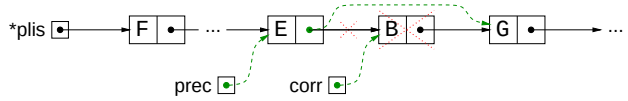
Cancellazione della prima occorrenza di un elemento

Versione iterativa

- si scandisce la lista alla ricerca dell'elemento
- se l'elemento non compare non si fa nulla
- altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
  1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo
  2. l'elemento non è ne il primo ne l'ultimo: si aggiorna il campo `next` dell'elemento che precede quello da cancellare in modo che punti all'elemento che segue
  3. l'elemento è l'ultimo: come (2), solo che il campo `next` dell'elemento precedente viene posto a `NULL`
- in tutti e tre i casi bisogna liberare la memoria occupata dall'elemento da cancellare

Osservazioni:

- per poter aggiornare il campo `next` dell'elemento precedente, bisogna **fermare la scansione sull'elemento precedente** (e non su quello da cancellare)



- per fermare la scansione dopo aver trovato e cancellato l'elemento, si utilizza una sentinella booleana
- per evitare di trattare a parte il caso del primo elemento, si può utilizzare di nuovo la tecnica del record generatore

Implementazione: file `puntator/listecon.c`, funzione `CancellaElementoLista`

Versione ricorsiva: per **esercizio**

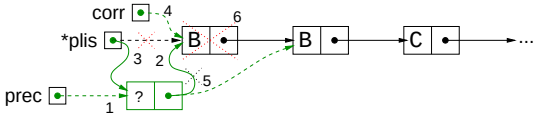
Implementazione: file `puntator/listeric.c`, funzione `CancellaElementoLista`

Cancellazione di tutte le occorrenze di un elemento — *Versione iterativa*

- analoga alla cancellazione della prima occorrenza di un elemento
- però, dopo aver trovato e cancellato l'elemento, bisogna continuare la scansione
- ci si ferma solo quando si è arrivati alla fine della lista

Osservazioni:

- in questo caso è decisamente meglio utilizzare il record generatore



- non serve la sentinella booleana per fermare la scansione

Implementazione: file `puntator/listecon.c`, funzione `CancellaTuttiLista`

Cancellazione di tutte le occorrenze di un elemento — *Versione ricorsiva*

Caratterizzazione **induttiva** della cancellazione di tutte le occorrenze:

Sia *ris* la lista ottenuta cancellando tutte le occorrenze di *elem* da *lis*. Allora:

1. se *lis* è la lista vuota, allora *ris* è la lista vuota (**caso base**)
2. altrimenti, se il primo elemento di *lis* è pari ad *elem*, allora *ris* è ottenuta da *lis* cancellando il primo elemento e tutte le occorrenze di *elem* dal resto di *lis* (**caso ricorsivo**)
3. altrimenti *ris* è ottenuta da *lis* cancellando tutte le occorrenze di *elem* dal resto di *lis* (**caso ricorsivo**)

Implementazione: file `puntator/listeric.c`, funzione `CancellaTuttiLista`

N.B. **Non ha alcun senso** applicare la tecnica del record generatore quando si implementa una funzione sulle liste in modo ricorsivo.

Inserimento di un elemento in una lista ordinata

Data una lista (ad es. di interi) già ordinata (in ordine crescente), si vuole inserire un nuovo elemento, mantenendo l'ordinamento.

Versione iterativa: per **esercizio**

Implementazione: file `puntator/listeord.c`, funzione `InserisciInListaOrdinata`

Versione ricorsiva

Caratterizzazione **induttiva** dell'inserimento in lista ordinata crescente:

Sia *ris* la lista ottenuta inserendo l'elemento *elem* nella lista ordinata *lis*. Allora

1. se *lis* è la lista vuota, allora *ris* è costituita solo da *elem* (**caso base**)
2. se il primo elemento di *lis* è maggiore o uguale a *elem*, allora *ris* è ottenuta da *lis* inserendovi *elem* in testa (**caso base**)
3. altrimenti *ris* è ottenuta da *lis* inserendo *elem* nel resto di *lis* (**caso ricorsivo**)

Implementazione: file `puntator/listeric.c`, funzione `InserisciInListaOrdinata`

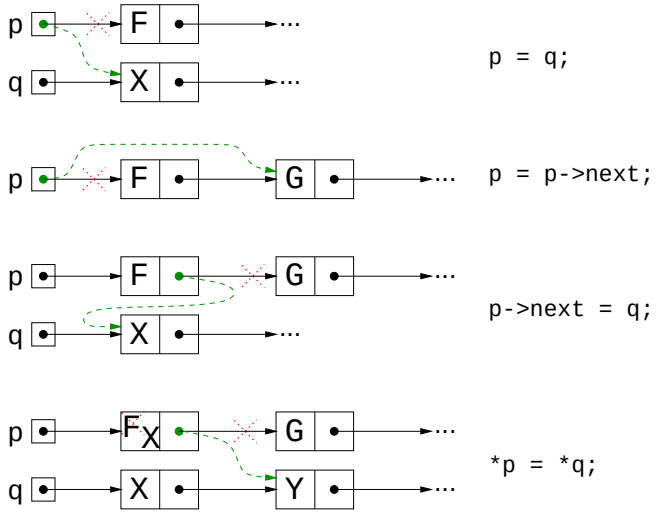
Sommario delle operazioni sulle liste

Funzione	Iterativa	Ricorsiva
`InserisciTestaLista`	lezione	
`CancellaPrimoLista`	lezione	
`StampaLista`	lezione	lezione
`EsisteInLista`	lezione	**esercizio**
`CancellaLista`	**esercizio**	lezione
`InserisciCodaLista`	lezione	lezione
`LeggiLista`	lezione (con rec. gen.)	**esercizio**
`CancellaElementoLista`	lezione (con rec. gen.)	**esercizio**
`CancellaTuttiLista`	lezione (con rec. gen.)	lezione
`InserisciInListaOrdinata`	**esercizio**	lezione
`CopiaLista`	**esercizio**	**esercizio**
`InvertiLista`	**esercizio**	**esercizio**

Implementazione: file `liste.c`, `listeric.c`, `listecon.c`, `listeord.c`, `listeirw.c`, `listerw.c` (tutti nella directory `puntator/`)

Tutte le funzioni sono discusse ed implementate nel libro di esercizi.

Riassunto delle operazioni sui puntatori



Esercizio di esame sulle liste

La Società Sportiva Olimpia dispone di un archivio in cui memorizza i farmaci che vengono assunti da ciascun atleta. L'archivio, realizzato attraverso strutture di dati C in memoria centrale, consiste di un vettore di 22 componenti, ciascuna delle quali è costituita da una struttura in cui sono memorizzati:

- la matricola dell'atleta (un intero) ed
- il puntatore ad una lista, rappresentata mediante strutture e puntatori, i cui elementi sono costittuiti da:
  - la sigla del farmaco (una stringa di al più 12 caratteri);
  - la data della prescrizione della terapia.

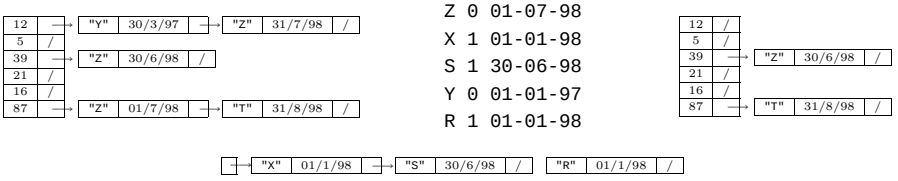
La Commissione Nazionale Antidoping ha aggiornato l'elenco delle sostanze vietate. In particolare, ha vietato alcune sostanze che precedentemente erano ammesse, ed ha ammesso alcune sostanze che erano vietate. La Commissione ha prodotto, e distribuito su floppy disk, un file in cui sono memorizzate le nuove informazioni. In particolare, ogni riga del file è costituita da tre campi, separati da uno spazio, che rappresentano rispettivamente:

- la sigla del farmaco (una striga di al più 12 caratteri);
- una cifra intera `0` o `1`, ad indicare se il farmaco è vietato (cifra `0`) o ammesso (cifra `1`);
- la data a partire dalla quale il farmaco è vietato o ammesso, a seconda dei casi, rappresentata nel formato `GG-MM-AA`.

Si richiede di:

1. Definire i tipi di dato C adeguati a risolvere i problemi di cui ai successivi punti (2) e (3).
2. Scrivere una funzione C che, dati come parametri l'archivio degli atleti della Società Olimpia ed il nome del file contenente le nuove informazioni sui farmaci vietati o ammessi, elimini dall'archivio tutte le prescrizioni dei farmaci vietati avvenute **non anteriormente** alla data del divieto.
3. Scrivere una funzione C che, dato come parametro il nome del file con le nuove informazioni sui farmaci, costruisca e fornisca in uscita la lista, rappresentata mediante strutture e puntatori, di tutti i farmaci ammessi, unitamente alle date di ammissione.

Esempio: Si consideri nella figura a sinistra l'archivio di atleti e al centro il file con le nuove informazioni sui farmaci vietati o ammessi. A destra è mostrato l'archivio dopo l'esecuzione della funzione richiesta al punto (2), ed in basso la lista fornita dalla funzione di cui al punto (3).



Strutture di dati

- per ogni atleta: – matricola
  – lista di prescrizioni: per ogni prescrizione sigla e data
  $\Longrightarrow$ vettore di strutture: – matricola
  – puntatore inziale alla lista di prescrizioni
- file: ogni riga: – sigla
  – se vietato (`0`) o ammesso (`1`)
  – data nel formato `GG-MM-AA`
- lista di farmaci ammessi: per ogni farmaco: – sigla
  – data di ammissione
  $\Longrightarrow$ stessi tipi della lista di prescrizioni

Implementazione: file `esami/doping.c`

Costruzione della lista di farmaci ammessi

Funzione che prende in ingresso il nome di un file e restituisce la lista dei farmaci ammessi:

- un parametro di ingresso (per valore), di tipo `char*`
- la lista viene restituita come valore di ritorno

algoritmo costruisci la lista di farmaci ammessi
  inizializza la lista alla lista vuota
  apri il file in lettura
  while non si è giunti alla fine del file
  do leggi un farmaco dal file
    if il farmaco è ammesso
    then inseriscilo (con sigla e data) nella lista
  chiudi il file

N.B. Non è richiesto un ordine specifico per gli elementi della lista.
$\Longrightarrow$ Adottiamo il modo più semplice per costruirla, cioè inserendo in testa.

Implementazione: file `esami/doping.c`

Cancellazione delle prescrizioni di farmaci vietati

Funzione che: prende in ingresso: – nome di un file – vettore di liste
restituisce: vettore di liste dalle quali sono stati tolti i farmaci non presenti prima del divieto

algoritmo cancella farmaci prescritti non prima del divieto
  for ogni elemento del file
  do if rappresenta un divieto
    then cancella le prescrizioni opportune dalle liste

Raffinamento successivo:

algoritmo cancella farmaci prescritti non prima del divieto
  apri il file in lettura
  while non si è giunti alla fine del file
  do leggi un elemento dal file
    if l'elemento rappresenta un divieto
    then for ogni atleta nel vettore di liste
      do cancella dalla lista di farmaci dell'atleta quelli richiesti dal divieto
  chiudi il file

Algoritmo ausiliario:

algoritmo cancella da una lista di farmaci quelli richiesti dal divieto
if la lista non è vuota
then cancella dal resto della lista i farmaci richiesti dal divieto
if il primo elemento
– ha una sigla che coincide con quella del divieto **e**
– ha una data di prescrizione non anteriore a quella del divieto
then eliminalo

Per confrontare due date usiamo una funzione ausiliaria (booleana).

Implementazione: file `esami/doping.c`

Esercizio: Gestione di un archivio di esami memorizzato su un file.

Le **informazioni** associate ad ogni esame sono:

- materia
- data in cui è stato sostenuto (giorno, mese, anno)
- voto

Le **operazioni di interesse** sono:

`A` ... inserimento di un nuovo esame
`B` ... cancellazione di un esame (per correggere eventuali errori)
`C` ... stampa della lista di esami su schermo
`D` ... calcolo media esami
`E` ... stampa esami con voto superiore/inferiore alla media
`F` ... stampa esami ordinati per data (difficile)
`G` ... stampa esami ordinati per voto (difficile)
`H` ... calcolo trimestre in cui sono stati sostenuti più/meno esami
...
`R` ... lettura dei dati da un file
`S` ... salvataggio dei dati su un file
`X` ... uscita dal programma, con eventuale richiesta di salvataggio dei dati

Interfaccia utente

- All'utente deve essere presentato un semplice menu delle operazioni, dal quale può scegliere l'operazione da effettuare digitando un carattere.
- Il menu viene presentato fino a quando l'utente non decide di uscire dal programma (operazione `X`).

Suggerimenti

- costruire (in memoria centrale) una lista collegata contenente i dati sugli esami letti da file (operazione `R`)
- effettuare le operazioni modificando la rappresentazione in memoria
- salvare su file i dati nella lista a richiesta dell'utente (operazione `S`)
- salvare su file i dati nella lista anche al momento dell'uscita dal programma (solo se sono state fatte modifiche dall'ultimo salvataggio), in modo da evitare di perdere eventuali modifiche fatte
- realizzare ciascuna operazione attraverso una funzione con opportuni parametri

I tipi di dato astratti

La nozione di tipo di dato astratto

Quando si affrontano problemi complessi è necessario procedere in due fasi separate.

1. **specifica** dell'algoritmo
2. **implementazione** dell'algoritmo in un linguaggio di programmazione

La fase (1) coinvolge la specifica di due aspetti:

- specifica dei dati che l'algoritmo deve manipolare
- specifica delle operazioni da eseguire per realizzare l'algoritmo

La fase (2) richiede di implementare nel LDP (realizzare):

- i dati, attraverso le strutture di dati del LDP
- le operazioni, attraverso le istruzioni del LDP

Per esprimere i dati in maniera independente dai vincoli imposti dal LDP si utilizzano i tipi di dato astratti.

Un **tipo di dato astratto** è costituito da tre componenti:

- il **dominio di interesse**, che rappresenta l'insieme di valori che costituiscono il tipo astratto, ed eventuali **altri domini**, che rappresentano altri valori coinvolti
- un insieme di **costanti**, che denotano valori del dominio di interesse
- un insieme di **operazioni** che si effettuano sugli elementi del dominio di interesse, utilizzando eventualmente elementi degli altri domini.

Tutte e tre le componenti sono indipendenti dalla rappresentazione e dall'uso del tipo stesso nei LDP (la fase di specifica è indipendente dai vincoli imposti dal LDP).

Concettualizzazione (o specifica) dei tipi di dato astratti

La concettualizzazione di un tipo di dato astratto richiede di specificare separatamente le tre componenti:

- gli insiemi che rappresentano il **dominio di interesse** e gli eventuali **altri domini**
- l'insieme delle **costanti** del tipo
- le **operazioni**, attraverso un insieme di **funzioni**:
  - la **segnatura** della funzione (ovvero il suo nome e il tipo dei parametri di ingresso ed uscita) specifica di cosa necessita un'operazione e cosa restituisce
  - il **significato** di un'operazione può venire specificato algebricamente in modo rigoroso. Noi descriveremo il significato delle operazioni in modo informale.

Esempio: Concettualizzazione del tipo astratto $Booleano$.

- dominio di interesse: $\{vero, falso\}$
- costanti: $true$ e $false$, che denotano rispettivamente $vero$ e $falso$
- operazioni:
$$\begin{aligned} and &: Booleano \times Booleano \rightarrow Booleano \\ or &: Booleano \times Booleano \rightarrow Booleano \\ not &: Booleano \rightarrow Booleano \end{aligned}$$

Esempio: Concettualizzazione del tipo astratto "insieme di lettere alfabetiche maiuscole".

- *dominio di interesse:* $InsLettere = \{\emptyset, \{A\}, \{B\}, \ldots, \{A, B, \ldots, Z\}\}$
  altri *domini:* $Lettera = \{A, B, \ldots, Z\}$
  $Booleano = \{vero, falso\}$
- *costanti:* $\emptyset$ (denota l'insieme vuoto)
- *operazioni:*
$$\begin{aligned} &VerificaInsiemeVuoto : InsLettere \rightarrow Booleano \\ &Inserisci : InsLettere \times Lettera \rightarrow InsLettere \\ &Cancella : InsLettere \times Lettera \rightarrow InsLettere \\ &VerificaAppartenenza : InsLettere \times Lettera \rightarrow Booleano \\ &Unione : InsLettere \times InsLettere \rightarrow InsLettere \\ &Intersezione : InsLettere \times InsLettere \rightarrow InsLettere \\ &\ldots \end{aligned}$$

  Le prime quattro sono le **operazioni di base**. Le altre operazioni possono essere ottenute componendo in modo opportuno le operazioni di base.

Realizzazione (o implementazione) dei tipi di dato astratti

Il tipo di dato astratto deve essere realizzato usando i costrutti del LDP:

1. rappresentazione dei **domini** usando i **tipi concreti** del LDP
2. codifica delle **costanti** attraverso i **costrutti** del LDP
3. realizzazione delle **operazioni** attraverso opportune **funzioni** del LDP

Esempio: Realizzazione del tipo astratto "insieme di lettere alfabetiche maiuscole".

1) $InsLettere$ rappresentato dal suo vettore caratteristico:

Es.: $\{B, C, Y\}$

0	1	1	0	0	$\cdots$	0	1	0
0	1	2	3	4	$\cdots$	23	24	25
A	B	C	D	E	$\cdots$	X	Y	Z

```
typedef int bool;
#define TRUE  1
#define FALSE 0

typedef char TipoLettera;
typedef bool TipoInsLettere['Z' - 'A' + 1];
```

2) Codifica della costante $\emptyset$

- per rappresentare $\emptyset$ si potrebbe usare un vettore di interi costanti; questo vettore non può però essere usato direttamente per effettuare inserimenti di elementi $\Longrightarrow$
- conviene definire una funzione che inizializza in modo opportuno un parametro passato per indirizzo

Es.:
```
void InitInsLettere(TipoInsLettere insieme)
{ TipoLettera ch;
  for (ch = 'A'; ch <= 'Z'; ch++)
    insieme[ch - 'A'] = FALSE;
}
```

3) Realizzazione delle operazioni

Es.:
```
void Inserisci(TipoInsLettere insieme, TipoLettera lettera)
{ insieme[lettera - 'A'] = TRUE; }
```

Esercizio: Fornire l'implementazione delle rimanenti operazioni.

Soluzione: file `tipidato/inslett.c`

Osservazioni:

- Per uno stesso tipo astratto si possono avere **più realizzazioni** diverse.

  Esempio: "Insieme di lettere alfabetiche maiuscole" può essere realizzato utilizzando il vettore caratteristico oppure una lista collegata di caratteri.

- Una realizzazione potrebbe avere delle **limitazioni** rispetto al tipo di dato astratto.

  Esempio: "Insieme di interi" potrebbe essere realizzato in modo tale da poter rappresentare come elementi solo gli interi rappresentabili in C.

Esercizio: Realizzare il tipo di dato astratto "Insieme di interi" (con domini, costanti e operazioni analoghe a $InsLettere$) usando liste concatenate.

Soluzione: file `tipidato/insint.c`

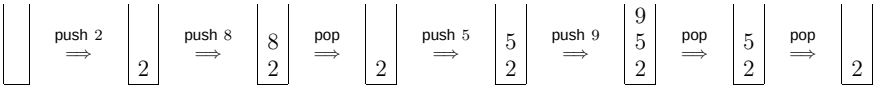
Il tipo di dato pila

Una **pila** è una sequenza di elementi (tutti dello stesso tipo) in cui l'inserimento e l'eliminazione di elementi avvengono secondo la regola:

> L'elemento che viene eliminato tra quelli presenti nella pila deve essere quello che è stato inserito per ultimo.

Si parla di gestione **LIFO** (per "**L**ast **I**n, **F**irst **O**ut").

Esempio:



- L'elemento in cima alla pila viene detto **elemento affiorante**.
- Una pila senza elementi viene detta **pila vuota**.

Utilizzo delle pile

Uno degli utilizzi delle pile si ha nella soluzione di problemi:

- che richiedono di effettuare più scelte in successione, e
- per cui può essere necessario ritrattare le scelte fatte per provare altre alternative
  $\Longrightarrow$ si deve tornare indietro sull'ultima scelta fatta (backtracking)

Si utilizza una pila per memorizzare la sequenza di scelte fatte:

- $Push$ di una nuova scelta;
- Pop ed eventuale $Push$ della scelta alternativa quando di deve fare backtracking.

Esempio: Ricerca di un cammino nella palude (file `ricorsio/palude.c`).
Nella soluzione ricorsiva viene utilizzata implicitamente la pila dei record di attivazione:

- quando si avanza, si effettua una chiamata ricorsiva ($Push$ di un nuovo RDA)
- quando non si può più proseguire, si esce dalla ricorsione per provare le scelte alternative (Pop del RDA ed eventuale $Push$ del RDA di una nuova chiamata ricorsiva)

Concettualizzazione del tipo di dato pila

- *domini:* $Pila$, $Elemento$, $Booleano$
- *costanti:* $PilaVuota$
- *operazioni:*
  - $TestPilaVuota : Pila \rightarrow Booleano$
    restituisce $vero$ se la pila è vuota, $falso$ altrimenti
  - $TopPila : Pila \rightarrow Elemento$
    restituisce l'elemento affiorante di una pila
  - $Push : Pila \times Elemento \rightarrow Pila$
    inserisce un elemento in cima ad una pila e restituisce la pila modificata
  - $Pop : Pila \rightarrow Pila \times Elemento$
    estrae l'elemento affiorante dalla cima della pila e lo restituisce (insieme alla pila modificata)

Realizzazione in C del tipo di dato pila

Indipendentemente da come verrà realizzato il tipo astratto, si possono specificare i prototipi delle funzioni che realizzano le diverse operazioni.

- *domini:*
```
typedef int bool;
typedef ... TipoElemPila;
typedef ... TipoPila;
```
- *costanti:*
```
void InitPila(TipoPila *pp);
```
- *operazioni:*
  - `bool TestPilaVuota(TipoPila p);`
  - `void TopPila(TipoPila p, TipoElemPila *pv);`
  - `void Push(TipoPila *pp, TipoElemPila v);`
  - `void Pop(TipoPila *pp, TipoElemPila *pv);`

Rappresentazione sequenziale delle pile

Una pila è rappresentata mediante:

- un **vettore di elementi** del tipo degli elementi della pila, riempito fino ad un certo indice con gli elementi della pila
- un **indice** che rappresenta la posizione nel vettore dell'elemento affiorante della pila

Esempio: Pila

8
34
12

rappresentata tramite:

`p.pila`

0	12
1	34
2	8
3	?
⋮	⋮
99	?

`p.pos`

2

Per la pila vuota: `p.pos` è pari a -1.

È necessario fissare la dimensione del vettore a tempo di compilazione. Si ha quindi un limite superiore al numero di elementi nella pila. $\Longrightarrow$

Si rende necessaria un'operazione di verifica di pila piena:
– `bool TestPilaPiena(TipoPila p);`

Implementazione: file `tipidato/pileseq.c`

Dichiarazioni di tipo:

```
#define MaxPila 100
typedef ... TipoElemPila;
struct tipoPila {
  TipoElemPila pila[MaxPila];
  int pos;
};
typedef struct tipoPila TipoPila;
```

Inizializzazione di una pila:

```
void InitPila(TipoPila *pp)
{
  (*pp).pos = -1;
}
```

Operazioni: sono tutte molto semplici e di **costo costante** (ovvero indipendente dal numero di elementi nella pila)

Esercizio: Implementare le operazioni sulle pile.

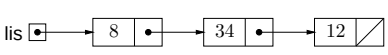
Rappresentazione collegata delle pile

Una pila è rappresentata mediante una **lista** nella quale il primo elemento è l'elemento affiorante della pila.

Esempio: Pila

8
34
12

rappresentata tramite:



Per rappresentare la pila vuota: lista vuota

Non è più necessario imporre un limite al numero massimo di elementi nella pila.
$\Longrightarrow$ `TestPilaVuota` non serve più.

Implementazione per **esercizio**: file `tipidato/pile.c`

Anche in questo caso le funzioni sono molto semplici e di costo costante:

- `Push`: tramite un inserimento in testa ad una lista
- `Pop`: tramite la cancellazione del primo elemento di una lista

Perchè non abbiamo usato una lista in cui l'elemento affiorante è l'ultimo della lista?

Rappresentazione sequenziale tramite vettori dinamici

Per utilizzare un vettore senza dover fissare la dimensione massima della pila a tempo di compilazione si può utilizzare un **vettore allocato dinamicamente**:

- si fissa una dimensione iniziale del vettore (che determina anche quella minima)
- la dimensione può crescere e decrescere a seconda delle necessità

Dichiarazioni di tipo:

```
#define DimInizialePila 10
typedef ... TipoElemPila;
struct tipoPila { TipoElemPila *pila;
                  int pos;
                  int dimCorrente; };
typedef struct tipoPila TipoPila;
```

Inizializzazione di una pila: richiede l'allocazione dinamica della memoria per il vettore

```
void InitPila(TipoPila *pp)
{ (*pp).pila = malloc(DimInizialePila * sizeof(TipoElemPila));
  (*pp).pos = -1;
  (*pp).dimCorrente = DimInizialePila;
}
```


Implementazione delle operazioni: file `tipidato/piledin.c`

- verifica di pila vuota e analisi dell'elemento affiorante:

  Come per la rappresentazione sequenziale statica.

- inserimento di un elemento:

  Se la pila è piena se ne raddoppia la dimensione.

  Si usa la funzione `realloc`, con prototipo:

  ```
  void *realloc(void *p, size_t dim);
  ```

  Alloca una nuova zona di memoria di dimensione specificata, copiandoci il contenuto di una zona di memoria precedentemente allocata dinamicamente.

- eliminazione dell'elemento affiorante:

  Se il numero di elementi nella pila è meno di un terzo della dimensione massima (e più della dimensione iniziale), allora si alloca una nuova zona di memoria di metà dimensione (usando `realloc`).

Confronto tra le rappresentazioni sequenziali e quella collegata

Rappresentazione sequenziale statica:

+ implementazione semplice
+ può essere usata anche con linguaggi privi di allocazione dinamica della memoria (ad ed. FORTRAN)
– è necessario fissare a priori il numero massimo di elementi nella pila
– l'occupazione di memoria è sempre pari alla dimensione massima

Rappresentazione sequenziale dinamica:

+ pila può crescere indefinitamente
– le operazioni di $Push$ e Pop possono richiedere la copia di tutti gli elementi presenti nella pila $\Longrightarrow$ non sono più a tempo di esecuzione costante

Rappresentazione collegata:

+ pila può crescere indefinitamente
+ occupazione di memoria è proporzionale al numero di elementi nella pila
– maggiore occupazione di memoria dovuta ai puntatori
– implementazione leggermente più complicata

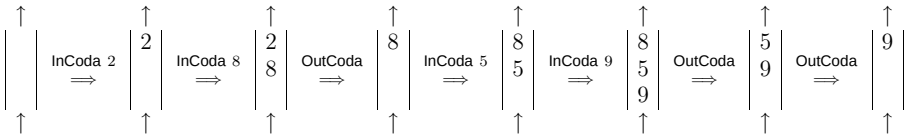
Il tipo di dato coda

Una **coda** è una sequenza di elementi (tutti dello stesso tipo) in cui l'inserimento e l'eliminazione di elementi avvengono secondo la regola:

> L'elemento che viene eliminato tra quelli presenti nella coda deve essere quello che è stato inserito per primo.

Si parla di gestione **FIFO** (per "**F**irst **I**n, **F**irst **O**ut").

Esempio:



Utilizzo delle code

Quando si vuole disaccoppiare il processamento di dati dalla loro generazione, ma si vuole mantenere l'ordinamento, ovvero processare i dato nello stesso ordine in cui sono stati generati.

Concettualizzazione del tipo di dato coda

- *domini:* $Coda$, $Elemento$, $Booleano$
- *costanti:* $CodaVuota$
- *operazioni:*
  - $TestCodaVuota : Coda \rightarrow Booleano$
    restituisce $vero$ se la coda è vuota, $falso$ altrimenti
  - $InizioCoda : Coda \rightarrow Elemento$
    restituisce il primo elemento di una coda
  - $InCoda : Coda \times Elemento \rightarrow Coda$
    inserisce un elemento in una coda e restituisce la coda modificata
  - $OutCoda : Coda \rightarrow Coda \times Elemento$
    estrae l'elemento dalla coda e lo restituisce (insieme alla coda modificata)

Realizzazione in C del tipo di dato coda

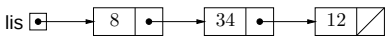
Indipendentemente da come verrà realizzato il tipo astratto, si possono specificare i prototipi delle funzioni che realizzano le diverse operazioni.

- *domini:*
```
typedef int bool;
typedef ... TipoElemCoda;
typedef ... TipoCoda;
```
- *costanti:*
```
void InitCoda(TipoCoda *pc);
```
- *operazioni:*
  - `bool TestCodaVuota(TipoCoda c);`
  - `void InizioCoda(TipoCoda c, TipoElemCoda *pv);`
  - `void InCoda(TipoCoda *pc, TipoElemCoda v);`
  - `void OutCoda(TipoCoda *pc, TipoElemCoda *pv);`

Rappresentazione collegata delle code

Come per le pile, si può rappresentare una coda tramite una **lista** nella quale il primo elemento è il primo elemento della coda (ovvero quello che deve uscire per primo).

Esempio: Coda $\uparrow 8\ 34\ 12 \uparrow$ rappresentata tramite:



Per rappresentare la coda vuota: lista vuota

Le operazioni $InitCoda$, $TestCodaVuota$, $InizioCoda$ e $OutCoda$ sono semplici da realizzare e di costo costante (ovvero, indipendente dalla lunghezza della coda).

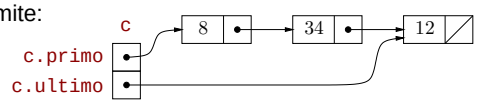
Per $InCoda$ bisogna fare un inserimento in coda ad una lista. $\Longrightarrow$
È necessaria una scansione di tutta la lista (costo non è più costante).

Si può modificare la rappresentazione in modo che tutte le operazioni vengano realizzate a costo costante?

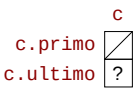
Si usa una rappresentazione mediante **lista e due puntatori**:

- un puntatore al primo elemento della coda
  (permette di realizzare $InizioCoda$ e $OutCoda$ a costo costante)
- un puntatore all'ultimo elemento della coda
  (permette di realizzare $InCoda$ a costo costante)

Esempio: Coda $\uparrow 8\ 34\ 12 \uparrow$ rappresentata tramite:



Per rappresentare la coda vuota:



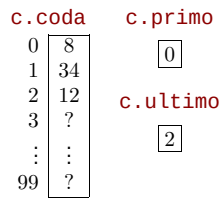
- vantaggio: tutte le operazioni hanno costo costante
- svantaggio: funzioni `InCoda` e `OutCoda` devono tenere conto dei due puntatori

Implementazione: file `tipidato/code.c`

Rappresentazione sequenziale delle code

Una coda è rappresentata mediante **un vettore** e **due indici**.

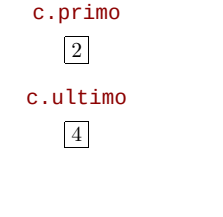
Esempio: Coda $\uparrow 8\ 34\ 12 \uparrow$ rappresentata tramite:



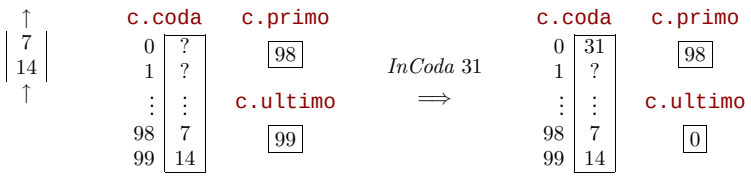
Per la coda vuota: `c.primo` è pari a -1.

Usando due indici si evita di spostare tutti gli elementi verso l'alto ad ogni $OutCoda$.

Esempio: $InCoda\ 25$, $InCoda\ 76$, $OutCoda$, $OutCoda$ $\uparrow 12\ 25\ 76 \uparrow$



Esempio: Dopo una serie di inserimenti ed estrazioni si potrebbe arrivare alla situazione:



$\Longrightarrow$ Il vettore deve essere **gestito in modo circolare**.

Verifica di coda piena: due possibilità

- `(c.primo - c.ultimo == 1)`
- `((c.ultimo - c.primo) == (MaxCoda-1))`

Implementazione delle operazioni: file `tipidato/codeseq.c`

Anche per le code si può usare una rappresentazione sequenziale tramite vettore dinamico.

Implementazione: per **esercizio**

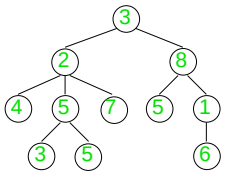
Gli alberi binari

Un **albero** è una struttura di data organizzata gerarchicamente.

È costituito da un insieme di **nodi** collegati tra di loro:

- ogni nodo contiene dell'informazione, detta **etichetta** del nodo
- i nodi collegati ad un certo nodo n vengono detti **figli** di n
- i nodi che non hanno figli sono detti **foglie** dell'albero
- c'è un unico nodo che non è figlio di alcun altro nodo, detto **radice** dell'albero (scendendo a partire dalla radice si raggiungono tutti i nodi dell'albero)

Esempio: Albero in cui le etichette sono degli interi.



la radice ha etichetta 3
i figli della radice hanno etichette 2 e 8
le foglie hanno etichette 4, 3, 5, 7, 5, 6

Albero vuoto: non ha nessun nodo (nemmeno la radice)

Cammino in un albero: sequenza di nodi, in cui ogni nodo è figlio del nodo che lo precede nella sequenza

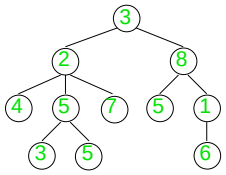
Esempio: (3, 8, 1, 6) è un cammino dalla radice ad una foglia
(8, 1) è un cammino

Livello (o **profondità**) di un nodo è la sua distanza dalla radice (quanto "in basso" si trova nell'albero).

Definizione formale (induttiva) di livello di un nodo:

- la radice ha livello 0
- se un nodo ha livello i, allora i suoi figli hanno livello $i + 1$

Esempio: Il nodo di etichetta 4 ha livello 2.
Il nodo di etichetta 6 ha livello 3.



Il **livello** i di un albero è formato da tutti i nodi a livello i.

Esempio: Il livello 2 è formato dai nodi con etichette 4, 5, 7, 5, 1.

Alberi binari

Un **albero binario** è un albero in cui ogni nodo ha **al massimo 2 figli**.

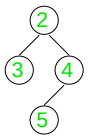
Esempio: Togliendo il nodo 7 dall'albero precedente otteniamo un albero binario.

- i due figli vengono detti **figlio sinistro** e **figlio destro**
- se consideriamo la parte di albero costituita dal figlio sinistro, dai suoi figli, dai figli dei suoi figli, . . . , questa è di nuovo un albero binario, detto **sottoalbero sinistro** (analogamente si definisce il **sottoalbero destro**)

Per rappresentare un albero binario in forma testuale si usa la **notazione parentetica**:

- albero vuoto: ( )
- albero non vuoto: (*radice* *sottoalbero-sinistro* *sottoalbero-destro*)
- se manca un figlio si mette l'albero vuoto come figlio

Esempio:




```
( 2  (3 () ())
     (4 (5 () ())
        ()
     )
)
```


Rappresentazione collegata degli alberi binari

L'elemento fondamentale è il nodo, che

- ha un'etichetta
- è collegato ai sottoalberi sinistro e destro (eventualmente vuoti)

$\Longrightarrow$ usiamo una struct a 3 campi:

- etichetta
- due puntatori ai sottoalberi sinistro e destro

```
typedef ... TipoInfoAlbero;                /* tipo dell'etichetta dei nodi */
struct nodoAlbero {
  TipoInfoAlbero info;
  struct nodoAlbero *destro, *sinistro;
}
typedef struct nodoAlbero NodoAlbero;
typedef NodoAlbero *TipoAlbero;
```


Stampa su schermo di un albero binario in rappresentazione parentetica

Implementazione: file `tipidato/albbin.c`, funzione `StampaAlbero`

Lettura da file di un albero binario in rappresentazione parentetica e creazione dell'albero in memoria

algoritmo lettura di un albero da file
  leggi '('
  leggi un carattere
  if il carattere è ')'
  then restituisci `NULL`
  else leggi la radice
    leggi il sottoalbero sinistro da file
    leggi il sottoalbero destro da file
    leggi ')'

Implementazione per **esercizio**: file `tipidato/albbin.c`, funzione `LeggiAlbero`

Visita di un albero

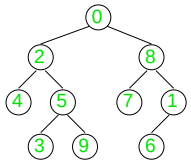
È l'analisi in sequenza di tutti i nodi dell'albero.

La maggior parte delle operazioni sugli alberi possono essere considerate delle varianti di visite.

Diversi **tipi di visita**, che differiscono per l'ordine in cui vengono visitati i nodi.

- visita in **preordine**: si analizza la radice, poi si visita in preordine il sottoalbero sinistro, poi il sottoalbero destro
- visita in **postordine**: si visita in postordine il sottoalbero sinistro, poi il sottoalbero destro, infine si analizza la radice
- visita **per livelli**: si visita prima la radice (livello 0), poi tutti i nodi a livello 1, poi tutti i nodi a livello 2, ...

Esempio:



preordine: 0 2 4 5 3 9 8 7 1 6
postordine: 4 3 9 5 2 7 6 1 8 0
per livelli: 0 2 8 4 5 7 1 3 9 6

Implementazione delle visite

Supponiamo di aver definito una funzione `Analizza`, a cui passiamo l'etichetta di un nodo, la quale effettua l'operazione richiesta dalla visita (assumiamo che sia semplicemente la stampa dell'etichetta).

- in preordine: banale, usando una funzione ricorsiva

  Implementazione: file `tipidato/albbin.c`, funzione `VisitaPreordine`

- in postordine: idem

  Implementazione: file `tipidato/albbin.c`, funzione `VisitaPostordine`

- per livelli:
  problema: dopo aver visitato un nodo, dobbiamo visitare i suoi fratelli, ma dobbiamo anche ricordarci i suoi figli (per visitarli dopo)
  $\Longrightarrow$ utilizziamo una coda, i cui elementi sono i nodi ancora da analizzare

  Implementazione: file `tipidato/albbin.c`, funzione `VisitaLivelli`

Esercizi:

- implementare le visite in preordine ed in postordine utilizzando una pila (invece che tramite una funzione ricorsiva)
- funzione che determina se l'albero contiene un nodo con una certa etichetta
- funzione che stampa tutte le foglie dell'albero
- per alberi con etichette di tipo `int`, funzione che stampa la somma delle etichette
  - di tutti i nodi dell'albero
  - di tutte le foglie dell'albero
  - di tutti i nodi ad un certo livello
  - dei nodi del sottoalbero la cui radice ha un'etichetta data
- funzione che costruisce la lista
  - dei nodi dell'albero visitati in preordine, postordine, per livelli
  - delle foglie dell'albero
  - dei nodi di un certo livello
- funzione che determina se due alberi sono uguali
- funzione che effettua la copia di un albero

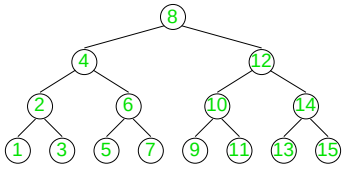
Alberi binari di ricerca

Consideriamo alberi binari per i quali è definito un **ordinamento** sull'informazione sui nodi (ad esempio interi, stringhe, ma anche strutture complesse con un campo chiave).

Un tale albero è un **albero binario di ricerca** se per **ogni** nodo N dell'albero:

- l'informazione associata ad **ogni** nodo nel sottoalbero sinistro di N è strettamente minore dell'informazione associata ad N
- l'informazione associata ad **ogni** nodo nel sottoalbero destro di N è strettamente maggiore dell'informazione associata ad N

Esempio:



L'albero di dice **bilanciato** se, detta P la profondità dell'albero, tutte le foglie stanno a livello P o $P-1$, e tutti i nodi interni hanno due figli, tranne al più un nodo al livello $P-1$ che ne ha uno solo.

La ricerca di un elemento *elem* in un albero binario di ricerca può essere fatta efficientemente (se l'albero è bilanciato):

algoritmo cerca *elem* nell'albero binario di ricerca *Alb*
  if *Alb* è vuoto
  then *elem* non è presente in *Alb*
  else if *elem* coincide con l'etichetta della radice di *Alb*
  then *elem* è stato trovato
  else if *elem* precede l'etichetta della radice di *Alb*
  then cerca *elem* nel sottoalbero sinistro di *Alb*
  else cerca *elem* nel sottoalbero destro di *Alb*

Implementazione: file `tipidato/albinric.c`

Costo: è **pari alla profondità** dell'albero. Sia n il numero di nodi dell'albero.

- caso migliore: l'albero è **bilanciato** $\Longrightarrow$ la profondità è proporzionale a $\log_2 n$
- caso peggiore: l'albero è una catena $\Longrightarrow$ la profondità è pari ad n

La complessità di calcolo

Valutazione dell'efficienza dei programmi

L'analisi della complessità di calcolo è lo studio dell'**efficienza** di algoritmi e programmi.

Obiettivo: trovare algoritmi (e quindi programmi) più efficienti per un dato problema.

Risorse considerate:

- tempo di elaborazione (è la risorsa più importante e l'unica che consideriamo)
- quantità di memoria occupata

Prima definizione (intuitiva) di efficienza: Un programma è più efficiente di un altro se la sua esecuzione richiede meno tempo di calcolo.

Problema: misurare il tempo (ad es. in secondi) non è attendibile. Bisogna tenere conto delle **condizioni** in cui si effettuano le prove per confrontare i due programmi:

- elaboratore (e sistema operativo)
- compilatore utilizzato
- dati in ingresso
- significatività dei dati in ingresso (più esecuzioni con dati diversi)

Non otteniamo un criterio oggettivo (persone diverse giungono a conclusioni diverse).

Funzione di costo

- Per ottenere un criterio oggettivo cerchiamo di esprimere il costo di un programma attraverso una funzione.
- Il costo dipende (quasi) sempre dai dati in ingresso.
- Il fattore determinante è la **dimensione dei dati in ingresso** (intesa come la memoria necessaria per memorizzarli).

  Esempio: Ordinamento di un vettore di 10 o 1000 elementi.

$\Longrightarrow$ L'argomento della funzione di costo è la dimensione dei dati in ingresso.

N.B. Tipicamente il costo di esecuzione dipende non solo dalla dimensione dei dati, ma anche dai particolari dati.

Assunzioni per calcolare il costo delle singole istruzioni:

- istruzione semplice (assegnazione, lettura, scrittura):
  costo 1 (*)
- istruzione composta:
  somma dei costi delle istruzioni componenti
- istruzione di ciclo:
  costo totale della condizione $+$ costo totale del corpo
  $=$ numero-iterazioni $*$ ($1+$ costo di una esecuzione del corpo)
- istruzione condizionale:
  costo della condizione $+$ $\begin{cases} \text{costo del ramo then, se la condizione è vera} \\ \text{costo del ramo else, se la condizione è falsa} \end{cases}$
- attivazione di funzione:
  costo di tutte le istruzioni che la compongono (eventualmente tenendo conto di attivazioni al suo interno)

L'ipotesi (*) è **semplificativa**.

Esempio: sia t_1 il costo di `x = x+1;`
sia t_2 il costo di `x = 2*(x+y) / (z*3*q);`

Non è realistico assumere che $t_1 = t_2 = 1$.
Però esiste una costante c tale che $c \cdot t_1 > t_2$.

Quindi, **(*) è vera a meno di un fattore moltiplicativo costante** (ovvero, indipendente dalla dimensione dei dati in ingresso).

Otteniamo una valutazione:

- indipendente dal calcolatore, compilatore, ...
- approssimata per un fattore moltiplicativo costante
- indipendente dal linguaggio di programmazione (si può fare riferimento direttamente all'algoritmo)

Esempio: Ricerca esaustiva di un elemento in un vettore

```
bool RicercaEsaustiva(TipoVettore A, TipoElemVettore elem, int n, int *posiz)
{ int i = -1;     bool trovato = FALSE;

  do i++; while (i < n-1 && A[i] != elem);
  if (A[i] == elem) { *posiz = i;
                      trovato = TRUE; }
  return trovato;
}
```

Costo di esecuzione del programma dipende dalla posizione dell'elemento cercato:

se è il 1^o: 1 iterazione $\Longrightarrow$ costo $= 2 + 2 \cdot 1 + 1 + 2 + 1 = 8$
se è il 2^o: 2 iterazioni $\Longrightarrow$ costo $= 2 + 2 \cdot 2 + 1 + 2 + 1 = 10$
se è l'i^o: i iterazioni $\Longrightarrow$ costo $= 2 + 2 \cdot i + 1 + 2 + 1 = 6 + 2 \cdot i$
se non c'è n iterazioni $\Longrightarrow$ costo $= 2 + 2 \cdot n + 1 + 1 = 4 + 2 \cdot n$

Si distinguono:

- il caso migliore (meno costoso) (Es.: è il 1^o)
- il caso peggiore (più costoso) (Es.: è l'ultimo o non c'è)

Per ottenere una valutazione del costo che dipende solo dalla dimensione dell'input e non dai particolari dati in ingresso **si fa riferimento al caso peggiore**.

Riassumendo, esprimiamo il **costo di un algoritmo** (o di un programma)

- come funzione della dimensione dell'input
- a meno di un fattore moltiplicativo costante
- facendo riferimento al caso peggiore

Classificazione degli algoritmi secondo il loro costo

- algoritmo **costante**: costo è c_0
- algoritmo **lineare**: costo è $c_1 \cdot n + c_0$
- algoritmo **quadratico**: costo è $c_2 \cdot n^2 + c_1 \cdot n + c_0$
- algoritmo **polinomiale**: costo è $c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \cdots + c_1 \cdot n + c_0$
- algoritmo **logaritmico**: costo è $c \cdot \log_2 n$
- algoritmo **esponenziale**: costo è $c \cdot 2^n$

dove: n è la dimensione dell'input
c, k, c_0, c_1, ..., c_k, sono costanti indipendenti da n

La notazione O

Definizione Un algoritmo (o un programma) ha **costo** (o **complessità**) $O(f(n))$ se il numero di istruzioni $t(n)$ che devono essere eseguite nel caso peggiore con input di dimensione n verifica

$$t(n) < c_1 \cdot f(n) + c_2$$

per ogni n e per opportune costanti c_1 e c_2 (indipendenti da n).

Quindi:

$O(1)$	... costo costante (indipendente dalla dimensione dell'input)
$O(n)$	... costo lineare
$O(n^2)$	... costo quadratico
$O(n^k)$	... costo polinomiale (k è una costante indipendente da n)
$O(\log_2 n)$	... costo logaritmico
$O(2^n)$	... costo esponenziale

Attenzione: **Non teniamo conto di costanti moltiplicative.**
Esempio: $O(n)$, sia se il costo è $2 \cdot n$ che se il costo è $1000 \cdot n$.

Quindi, stiamo facendo una **valutazione semplificata** che però funziona bene.

- motivo teorico: per input sufficientemente grandi la dimensione dell'input prevale sempre sulle costanti moltiplicative
- motivo pratico: le costanti moltiplicative sono quasi sempre piccole.

Questo significa che in pratica, per quanto riguarda il costo di un algoritmo che risolve un certo problema:

$O(\log_2 n)$	è molto meglio di $O(n)$
$O(n)$	è meglio di $O(n \cdot \log_2 n)$
$O(n \cdot \log_2 n)$	è meglio di $O(n^2)$
$O(n^2)$	è meglio di $O(n^3)$
...	
$O(n^k)$	è molto molto molto meglio di $O(2^n)$.

Istruzione dominante

Il fatto di trascurare le costanti moltiplicative permette di semplificare ulteriormente la valutazione del costo di un algoritmo.

Dato un algoritmo (o un programma) con costo $O(f(n))$, un'**istruzione** è **dominante** se, per ogni intero n, viene ripetuta nel caso peggiore di input di dimensione n, un numero di volte $d(n)$ tale che

$$c \cdot d(n) > f(n)$$

per un'opportuna costante c (che non dipende da n).

Intuitivamente, un'istruzione dominante è una di quelle che vengono eseguite più volte tra tutte le istruzioni dell'algoritmo (tipicamente, le istruzioni nei cicli più interni).

Esempio: Nella ricerca esaustiva (file `riceordi/ricesau.c`): `i++`

$\Longrightarrow$ Per valutare il costo di un algoritmo è sufficiente identificare un'istruzione dominante e valutare il costo di esecuzione di tale istruzione.

Ricerca e Ordinamento

Ricerca di un elemento in un vettore

Problema: Effettuare le seguenti **operazioni** su un insieme di dati omogenei:

- **ricerca** di un elemento: dati un insieme *A* ed un elemento *elem*, vogliamo sapere se *elem* è o meno presente in *A*
- **inserimento** di un elemento: dati un insieme *A* ed un elemento *elem*, se *elem* non compare in *A*, vogliamo inserirlo
- **eliminazione** di un elemento: dati un insieme *A* ed un elemento *elem*, se *elem* compare in *A*, vogliamo eliminarlo

Struttura di dati utilizzata: vettore, con una componente per ogni elemento

Operazione fondamentale: cercare un elemento nel vettore (è necessaria anche per inserimento ed eliminazione)

Ricerca esaustiva (in un vettore qualsiasi)

Se il vettore è costituito da elementi qualsiasi, non si può far di meglio che cercare l'elemento tra tutti gli elementi del vettore. $\Longrightarrow$

Ricerca esaustiva: si scandisce il vettore, confrontando gli elementi con quello da cercare, fino a che

- si sono confrontati tutti gli elementi, oppure
- si è trovato l'elemento cercato.

Implementazione: file `riceordi/ricesau.c`

Complessità: lineare (ovvero $O(n)$, dove n è il numero di elementi del vettore)

Ricerca binaria (in un vettore ordinato)

Se il vettore è ordinato, possiamo sfruttare l'ordinamento dimezzando ad ogni passo l'intervallo di elementi ancora da considerare per la ricerca.

Per cercare un elemento *elem* in un vettore ordinato *A*:

- usiamo due indici *inf* e *sup* per delimitare l'intervallo di ricerca
- confrontiamo *elem* con l'elemento di *A* al centro dell'intervallo, ovvero con *A[med]*, dove *med* $=$ *(inf*$+$*sup)*$/2$.

  Distinguiamo 3 casi:
  1. *elem* $=$ *A[med]* $\Longrightarrow$ trovato
  2. *elem* $<$ *A[med]* $\Longrightarrow$ cerchiamo nell'intervallo con indici da *inf* a *med*-1
  3. *elem* $>$ *A[med]* $\Longrightarrow$ cerchiamo nell'intervallo con indici da *med*$+1$ a *sup*
- se non troviamo *elem* ci fermiamo quando l'intervallo di ricerca si è annullato (ovvero, *inf* $>$ *sup*)

Esempio: Ricerca di 24 nel seguente vettore ordinato.

0	1	2	3	4	5	6	7	8	9	10
12	14	19	24	31	33	40	45	52	60	68

$\uparrow$ *inf* $= 0$ $\quad$ $\uparrow$ *med* $= 5$ $\quad$ $\uparrow$ *sup* $= 10$

$\uparrow$ *inf* $\quad$ $\uparrow$ *med* $\quad$ $\uparrow$ *sup*

inf $=$ *med* $\uparrow$ $\quad$ $\uparrow$ *sup* $\quad$ $\Longrightarrow$ trovato 24

se *elem* $= 31$ $\quad$ $\uparrow$ *inf* $=$ *med* $=$ *sup* $\quad$ $\Longrightarrow$ trovato 31

se *elem* $= 28$ $\quad$ *sup* $\uparrow$ $\quad$ $\uparrow$ *inf* $\quad$ $\Longrightarrow$ non compare 28

algoritmo ricerca *elem* nella parte di *A* compresa tra *inf* e *sup*
  if *sup* $<$ *inf*
  then l'elemento non compare in *A*
  else *med* $\leftarrow$ *(inf*$+$*sup)*$/2$
    if *elem* $=$ *A[med]*
    then l'elemento è stato trovato
    else if *elem* $<$ *A[med]*
    then ricerca *elem* nella parte di *A* tra *inf* e *med-1*
    else ricerca *elem* nella parte di *A* tra *med+1* e *sup*

Implementazione: file `riceordi/ricbinri.c`
Usa una funzione ricorsiva che prende come parametri anche gli estremi inferiore e superiore dell'intervallo di ricerca.

Esercizio: Fornire l'algoritmo e l'implementazione iterativa della ricerca binaria.
Soluzione: file `riceordi/ricbinit.c`

Complessità: ad ogni confronto dimezziamo la dimensione dell'intervallo di ricerca:

$$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{2^2} \rightarrow \cdots \rightarrow \frac{n}{2^m}$$

Nel caso peggiore (l'elemento non c'è) la ricerca termina dopo m confronti, con m tale che:

$$\frac{n}{2^m} \leq 1 \qquad \text{ovvero} \qquad n \leq 2^m \qquad \text{ovvero} \qquad m \geq \log_2 n$$

$\Longrightarrow$ La complessità della ricerca binaria è $O(\log_2 n)$.

Ordinamento di un vettore

Problema: Data una sequenza di elementi in ordine qualsiasi, ordinarla.

Questo è un problema fondamentale, che si presenta in moltissimi contesti, ed in diverse forme:

- ordinamento degli elementi di un vettore in memoria centrale
- ordinamento di una struttura collegata in memoria centrale
- ordinamento di informazioni memorizzate in un file su memoria di massa (file ad accesso casuale su disco, file ad accesso sequenziale su disco o su nastro)

Noi consideriamo solo la variante più semplice del problema dell'ordinamento.

Problema: Dato un vettore *A* di *n* elementi in memoria centrale non ordinato, ordinarne gli elementi in ordine crescente.

Usiamo le seguenti definizioni di tipo:

```
#define NumElementi 20
typedef int TipoElemVettore;
typedef TipoElemVettore TipoVettore[NumElementi];
```


Ordinamento per selezione del minimo (selection sort)

Esempio: Ordinamento di una pila di carte:

- seleziono la carta più piccola e la metto da parte
- delle rimanenti seleziono la più piccola e la metto da parte
- ...
- mi fermo quando rimango con una sola carta

Quando ho un vettore:

- per selezionare l'elemento più piccolo tra quelli rimanenti uso un ciclo
- "mettere da parte" significa scambiare con l'elemento che si trova nella posizione che compete a quello selezionato

Implementazione: file `riceordi/ordsel.c`

Complessità dell'ordinamento per selezione

Doppio ciclo $\Longrightarrow$ $O(n^2)$

Più in dettaglio: Istruzione dominante è il confronto `(A[j] < A[i_min])`.

Viene eseguita $(n-1)+(n-2)+\cdots+2+1=\frac{n\cdot(n-1)}{2}$ volte.
$\Longrightarrow$ $O(n^2)$ operazioni

La complessità è $O(n^2)$ in tutti i casi (anche se il vettore è già ordinato).

Numero di scambi:

- nel caso migliore: 0
- nel caso peggiore: $O(n)$

5	0	1	2	3	4
0	1	2	3	4	5

Ordinamento a bolle (bubble sort)

Idea: Si fanno "salire" gli elementi più piccoli ("più leggeri") verso l'inizio del vettore ("verso l'alto"), scambiandoli con quelli adiacenti.

L'ordinamento è suddiviso in $n-1$ fasi:

- fase 0: 0^o elemento (il più piccolo) in posizione 0
- fase 1: 1^o elemento in posizione 1
- $\cdots$
- fase $n-2$: $(n-2)^o$ elemento in posizione $n-2$, e quindi
  $(n-1)^o$ elemento in posizione $n-1$

Nella fase i: cominciamo a confrontare **dal basso** e portiamo l'elemento più piccolo (più leggero) in posizione i

Implementazione: file `riceordi/ordbub.c`

Osservazione: Se durante una fase non avviene alcuno scambio, allora il vettore è ordinato. Se invece avviene almeno uno scambio, non sappiamo se il vettore è ordinato o meno.

$\Longrightarrow$ Possiamo interrompere l'ordinamento appena durante una fase non avviene alcuno scambio.

Implementazione: per **esercizio**: file `riceordi/ordbubot.c`

Complessità della versione ottimizzata

- nel caso migliore (vettore già ordinato correttamente):
  $O(n)$ confronti, 0 scambi
- nel caso peggiore (vettore ordinato al contrario):
  $n-1$ fasi $\Longrightarrow$ $(n-1)+(n-2)+\cdots+2+1=\frac{n\cdot(n-1)}{2}$ confronti e scambi
  $O(n^2)$ confronti, $O(n^2)$ scambi

Ordinamento per fusione (merge sort)

Per ordinare 1000 elementi usando un algoritmo di complessità $O(n^2)$ (ordinamento per selezione o a bolle): sono necessarie 10^6 operazioni

Idea: Divido i 1000 elementi in due gruppi da 500 elementi ciascuno:

- ordino il primo gruppo in $O(n^2)$: $250\,000$ operazioni
- ordino il secondo gruppo in $O(n^2)$: $250\,000$ operazioni
- combino (fondo) i due gruppi ordinati: si può fare in $O(n)$ $\Longrightarrow$ 1000 operazioni

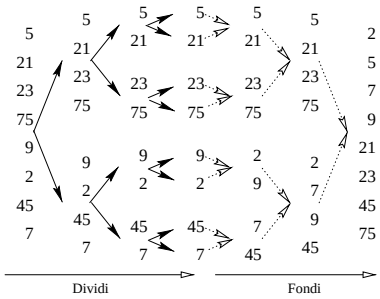
Totale: $501\,000$ operazioni (contro le 10^6)

Il procedimento può essere iterato: per ordinare le due metà non uso un algoritmo di complessità $O(n^2)$, ma applico lo stesso procedimento di divisione, ordinamento separato e fusione.

La suddivisione in due metà si ferma quando si arriva ad un gruppo costituito da un solo elemento (che è già ordinato).

algoritmo ordina per fusione gli elementi di *A* da *iniziale* a *finale*
  if *iniziale* $<$ *finale* (ovvero, c'è più di un elemento tra *iniziale* e *finale*, estremi inclusi)
  then *mediano* $\leftarrow$ (*iniziale* $+$ *finale*) $/2$
    ordina per fusione gli elementi di *A* da *iniziale* a *mediano*
    ordina per fusione gli elementi di *A* da *mediano* $+1$ *finale*
    fondi gli elementi di *A* da *iniziale* a *mediano* con
      gli elementi di *A* da *mediano* $+1$ a *finale*
      restituendo il risultato nel sottovettore di *A* da *iniziale* a *finale*

Esempio:



Per effettuare la **fusione** di due sottovettori ordinati e contigui ottenendo un unico sottovettore ordinato:

- si utilizzano un vettore di appoggio e due indici per scandire i due sottovettori
- il più piccolo tra i due elementi indicati dai due indici viene copiato nella prossima posizione del vettore di appoggio, e viene fatto avanzare l'indice corrispondente
- quando uno dei due sottovettori è terminato si copiano gli elementi rimanenti dell'altro nel vettore di appoggio
- alla fine si ricopia il vettore di appoggio nelle posizioni occupate dai due sottovettori

Implementazione: file `riceordi/ordmerge.c`

- funzione `MergeVettore` effettua la fusione
- funzione `MergeRicorsivo` effettua l'ordinamento di un sottovettore compreso tra due indici *iniziale* e *finale*
- funzione `MergeSort` chiama `MergeRicorsivo` sull'intervallo di indici da 0 a $n-1$

Complessità dell'ordinamento per fusione

Sia $T(n)$ il costo di ordinare per fusione un vettore di n elementi.

- se il vettore ha 0 o 1 elementi: $T(n) = c_1$
- se il vettore ha più di un elemento:

$$\begin{aligned} T(n) &= \text{costo dell'ordinamento della prima metà} \\ &\quad + \text{costo dell'ordinamento della seconda metà} \\ &\quad + \text{costo della fusione} \\ &= T(n/2) + T(n/2) + c_2 \cdot n \end{aligned}$$

Otteniamo un'equazione di ricorrenza la cui soluzione è la funzione di complessità cercata:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ T(n/2) + T(n/2) + c_2 \cdot n & \text{se } n > 1 \end{cases}$$

La soluzione, ovvero la complessità dell'ordinamento per fusione è $T(n) = O(n \cdot \log_2 n)$.

Questo è quanto di meglio si possa fare. Si può infatti dimostrare che nel caso peggiore sono necessari $n \cdot \log_2 n$ confronti per ordinare un vettore di n elementi.

Rissunto sulla complessità degli algoritmi di ordinamento

algoritmo	confronti	scambi	complessià totale
selection sort	$O(n^2)$ sempre	$O(n)$ 0 se già ordinato	$O(n^2)$ sempre
bubble sort (ottimizzato)	$O(n^2)$ $O(n)$ se già ordinato	$O(n^2)$ 0 se già ordinato	$O(n^2)$ $O(n)$ se già ordinato
insertion sort	$O(n^2)$ $O(n)$ se già ordinato	$O(n^2)$ 0 se già ordinato	$O(n^2)$ $O(n)$ se già ordinato
merge sort	$O(n \cdot \log_2 n)$ sempre	$O(n \cdot \log_2 n)$ sempre (serve vettore appoggio)	$O(n \cdot \log_2 n)$ sempre
quick sort			$O(n^2)$ caso peggiore $O(n \cdot \log_2 n)$ caso medio

Quick sort è un algoritmo molto utilizzato in pratica

- si comporta bene in media, anche se ci sono vettori per cui ha costo $O(n^2)$
- in pratica è più efficiente e più semplice da realizzare di merge-sort
- non richiede l'utilizzo di un vettore di appoggio