Informatica

Informatica significa Informazione automatica. Per fornire informazione è necessario:

- comprendere la richiesta. Es. Tra quanto passa l'espresso per Torino?
- avere i dati necessari a disposizione. L'espresso per Torino passa alle 12. Sono le 11.30
- essere in grado di elaborarli. Saper calcolare $12.00 - 11,30 = 0.30$
- comunicare la risposta. Tra 30 minuti

Informazione automatica

L'informazione automatica viene fornita da una macchina che:

- per comprendere la richiesta. deve avere un dispositivo che permetta di specificare la richiesta, ad es. attraverso dei tasti
- avere i dati necessari a disposizione. scritti su un dispositivo di memorizzazione
- essere in grado di elaborarli. Saper calcolare la funzione che in base alla richiesta e ai dati memorizzati calcola la risposta
- comunicare la risposta. Avere un dispositivo su cui comunicare il risultato del calcolo.

Un esempio: la calcolatrice

- La calcolatrice
  - ha 10 tasti per specificare le cifre
  - ha 4 tasti per specificare le operazioni (funzioni) da calcolare
  - una memoria su cui mantenere i risultati parziali
  - un foglio o un display sui cui stampare (o visualizzare) il risultato
- Analogamente posso definire altre macchine che calcolano altre funzioni su altri dati ma...
- ogni macchina è limitata ad un numero finito di operazioni mentre per i dati..

Funzioni calcolabili

Sorgono spontanee alcune domande:

- Quali sono le funzioni che si possono calcolare?
- Quante sono le funzioni calcolabili?
- Quali sono le funzioni che una macchina può calcolare?
- Posso costruire una macchina per ogni funzione calcolabile?
- È possibile costruire una macchina che calcoli tutte le funzioni calcolabili?

Funzioni calcolabili

Molto prima che venissero costruito il primo calcolatore i matematici avevano studiato il dominio delle funzioni calcolabili Ω

1. Le funzioni calcolabili sono quelle per le quali esiste un modo effettivo di calcolarle (algoritmo)
2. Le funzioni calcolabili sono infinite (numerabili)
3. Si! posso costruire una macchina per ogni funzione calcolabile
4. Ma la vera cosa interessante è che posso costruire una macchina che calcoli tutte le (infinite) funzioni calcolabili: questa macchina è il moderno calcolatore.

Importanti proprietà di Ω

- contiene tutte le funzioni i cui valori possono essere calcolati in modo effettivo: non solo concepiti (Es. razionali vs. reali).
- Ogni funzione calcolabile ha una (?) descrizione finita D. Es. la funzione fattorale ha la seguente descrizione:
- D(fatt)

  fatt (0) = 1

  fatt (n+1) = (n+1) * fatt (n)

Funzione Universale $\mathcal{U}_\Omega$

- Esiste una funzione universale $\mathcal{U}_\Omega$ che calcola tutte le funzioni calcolabili: $\forall f \in \Omega$, $\forall$ descrizione $D(f)$, $\forall$ valore n nel dominio di f $\mathcal{U}_\Omega(D(f), n) = f(n)$
- $\mathcal{U}_\Omega \in \Omega$ ovvero è calcolabile
- Il calcolatore è una realizzazione di $\mathcal{U}_\Omega$ cioè calcola prendendo come dati la descrizione della funzione da calcolare D(f) e i valori (n) su cui calcolare f.

Linguaggi di programmazione

Aspetto fondamentale è la descrizione della funzione e il linguaggio in cui tale descrizione è data.

- Linguaggi per descrivere funzioni sono chiamati linguaggi di programmazione
- Sono formalismi, con sintassi e semantica formalmente definite.
- Esempi: Caml, Haskel, Prolog, C, JAVA, Pascal, linguaggi macchina
- Le descrizioni delle funzioni da calcolare sono frasi in un linguaggio di programmazione e sono detti programmi.

Linguaggi di programmazione

- I linguaggi di programmazione sono tutti equivalenti: Se ho $D_{L1}(f)$ programma nel linguaggio L1 e $D_{L2}(f)$ programma nel linguaggio L2 che calcolano la stessa funzione abbiamo:

$$D_{L1}(f) \rightarrow D_{L2}(f)$$

- La traduzione è una funzione calcolabile e quindi descrivibile con un programma, che quindi può essere eseguito da un calcolatore.

Il linguaggio macchina

Il linguaggio direttamente eseguibile dal calcolatore (detto linguaggio macchina) è un linguaggio di programmazione poco comprensibile agli umani (programmatori) che devono scrivere il programma perchè:

- le operazioni sono molto semplici,
- sono specificate in binario: sequenze di 0 e 1 quindi
- specificare un'operazione complessa richiede la scrittura di un lungo programma, incomprensibile.

Linguaggi ad alto livello

- Tutti i programmi vengono scritti in un linguaggio ad alto livello (per voi il C) e sono molto più:
  - compatti,
  - comprensibili
  - modificabili
  - estendibili
  - correggibili
- i programmi di tali linguaggi devono pertanto essere tradotti in linguaggio macchina per poter essere eseguiti.
- Fortunatamente la traduzione (come abbiamo già detto) può essere fatta dalla macchina stessa.
- I programmi che effettuano la traduzione si chiamano compilatori

Le fasi della programmazione

Ad un primo livello di astrazione l'attività della programmazione può essere suddivisa in quattro (macro) fasi principali.

1. Definizione del problema (specifica): quale funzione si vuole calcolare e quali sono i dati di interesse
2. Individuazione di un procedimento risolutivo (algoritmo)
3. Codifica dell'algoritmo in un linguaggio di programmazione (codifica)
4. Esecuzione e messa a punto (esecuzione)

Specifica

- La prima fase della programmazione consiste nel comprendere e definire (specificare) il problema che si vuole risolvere.
- La specifica del problema può essere fatta in maniera più o meno rigorosa, a seconda del formalismo descrittivo utilizzato.
- La specifica di un problema prevede la descrizione dello stato iniziale (dati iniziali, input) e dello stato finale atteso (i risultati, output).
- La caratterizzazione degli stati iniziale e finale dipende dal particolare problema in esame e dagli oggetti di interesse.

Esempi di specifica informale

1. Dati due numeri, trovare il maggiore.
2. Dato un 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. Il testo di un esercizio: Si scriva un programma C che stampa "Ciao Nome Cognome", dove nome e cognome sono il proprio nome e cognome.

- la descrizione (es 3) non fornisce un metodo risolutivo
- la descrizione (es 2) del problema può essere ambigua o imprecisa (se Mario Rossi compare più volte)

Algoritmi

- Una volta specificato il problema, si determina un procedimento risolutivo dello stesso (algoritmo), ovvero un insieme di azioni da intraprendere per ottenere i risultati attesi.
- Il concetto di algoritmo ha origini molto lontane: l'uomo ha utilizzato spesso algoritmi per risolvere problemi di varia natura. Solo in era moderna, tuttavia, ci si è posti il problema di caratterizzare problemi e classi di problemi per i quali è possibile individuare una soluzione algoritmica e solo nel secolo scorso è stato dimostrato che esistono problemi per i quali non è possibile individuare una soluzione algoritmica.

Algoritmi (cont.)

- Utilizziamo algoritmi nella vita quotidiana tutte le volte che, ad esempio, seguiamo le istruzioni per il montaggio di una apparecchiatura, per la sostituzione della cartuccia di una stampante, per impostare il ciclo di lavaggio di una lavastoviglie, per prelevare contante da uno sportello Bancomat, ecc.

Un algoritmo è una sequenza di passi che, se intrapresa da un esecutore, permette di ottenere i risultati attesi a partire dai dati forniti.

Proprietà di un algoritmo

La descrizione di un procedimento risolutivo può considerarsi un algoritmo se rispetta alcuni requisiti essenziali, tra i quali:

Finitezza: un algoritmo deve essere composto da una sequenza finita di passi elementari.

Eseguibilità: il potenziale esecutore deve essere in grado di eseguire ogni singola azione in tempo finito con le risorse a disposizione

Non-ambiguità: l'esecutore deve poter interpretare in modo univoco ogni singola azione.

Tipici procedimenti che non rispettano alcuni dei requisiti precedenti sono:

- le ricette di cucina: *aggiungere sale q.b.* - non rispetta 3)
- le istruzioni per la compilazione della dichiarazione dei redditi (!)

Codifica

- Questa fase consiste nell'individuare una rappresentazione degli oggetti di interesse del problema ed una descrizione dell'algoritmo in un opportuno linguaggio noto all'esecutore.
- Nel caso in cui si intenda far uso di un elaboratore per l'esecuzione dell'algoritmo, quest'ultimo deve essere tradotto (codificato) in un opportuno linguaggio di programmazione. Il risultato in questo caso è un programma eseguibile al calcolatore.
- Quanto più il linguaggio di descrizione dell'algoritmo è vicino al linguaggio di programmazione scelto, tanto più semplice è la fase di traduzione e codifica. Se addirittura il linguaggio di descrizione coincide con il linguaggio di programmazione, la fase di traduzione è superflua.

Codifica (cont.)

I linguaggi di programmazione forniscono strumenti linguistici per rappresentare gli algoritmi sottoforma di programmi che possano essere compresi da un calcolatore.

In particolare dobbiamo rappresentare nel linguaggio di programmazione

- l'algoritmo $\Longrightarrow$ programma
- le informazioni iniziali $\Longrightarrow$ dati in ingresso
- le informazioni utilizzate dall'algoritmo $\Longrightarrow$ dati ausiliari
- le informazioni finali $\Longrightarrow$ dati in uscita

In questo corso impareremo a codificare algoritmi utilizzando il linguaggio di programmazione denominato C.

Esecuzione

- La fase conclusiva consiste nella:
  - compilazione del programma: cioè la traduzione in linguaggio macchina. Questa fase porta (quasi sempre) alla luce errori di vario genere (sintattici, o di tipo ma sempre statici).
  - esecuzione vera e propria del programma. Anche questa fase può portare alla luce errori, in questo caso di semantica (errori dinamici)
- In entrambi i casi la correzione degli errori può richiedere la revisione di una o piú fasi (dalla specifica, alla definizione dell'algoritmo, alla codifica di quest'ultimo).
- Nel caso dei linguaggi di programmazione moderni, vengono forniti strumenti (denominati di solito debugger) che aiutano nella individuazione degli errori e nella messa a punto dei programmi.

Esempi

Negli esempi che seguono, utilizziamo un linguaggio pseudo-naturale per la descrizione degli algoritmi.

Tale linguaggio utilizza, tra l'altro, le comuni rappresentazioni simboliche dei numeri e delle operazioni aritmetiche.

Problema 1: Calcolo del prodotto di due interi positivi

Specifica

Input: due valori interi positivi A e B

Output: il valore di $A \times B$

Algoritmo

Se l'esecutore che scegliamo è in grado di effettuare tutte le operazioni di base sui numeri, un semplice algoritmo è il seguente:

Passo 1.	Acquisisci il primo valore, sia esso A
Passo 2.	Acquisisci il secondo valore, sia esso B
Passo 3.	Ottieni il risultato dell'operazione $A \times B$

Problema 1 (cont.)

Codifica

Un esecutore che rispetta le ipotesi precedenti è una persona dotata di una calcolatrice tascabile. La codifica dell'algoritmo in questo caso può essere allora la seguente:

Passo 1.	Digita in sequenza le cifre decimali del primo valore
Passo 2.	Digita il tasto $*$
Passo 3.	Digita in sequenza le cifre decimali del secondo valore
Passo 4.	Digita il tasto $=$

Esecuzione

Problema 1 (cont.)

- Supponiamo ora che l'esecutore scelto sia in grado di effettuare solo le operazioni elementari di somma, sottrazione, confronto tra numeri.
- È necessario individuare un nuovo procedimento risolutivo che tenga conto delle limitate capacità dell'esecutore

Algoritmo 2

Passo 1.	Acquisisci il primo valore, sia esso A
Passo 2.	Acquisisci il secondo valore, sia esso B
Passo 3.	Associa 0 ad un terzo valore, sia esso C
Passo 4.	**Finché** B>0 **ripeti**
Passo 4.1.	Somma a C il valore A
Passo 4.2.	Sottrai a B il valore 1
Passo 5.	Il risultato è il valore C

Problema 1: (cont.)

Un esecutore che rispetta le ipotesi precedenti è un bimbo in grado di effettuare le operazioni richieste (somma, sottrazione e confronto) e di riportare i risultati di semplici calcoli su un quaderno.

Codifica

Passo 1. Scrivi il primo numero nel riquadro A

Passo 2. Scrivi il secondo numero nel riquadro B

Passo 3. Scrivi il valore 0 nel riquadro C

Passo 4. Ripeti i seguenti passi finché il valore nel riquadro B è maggiore di 0:

1. calcola la somma tra il valore in A e il valore in C e e scrivi il risultato ottenuto in C
2. calcola la differenza tra il valore in B ed il numero 1 e scrivi il risultato ottenuto in B

Passo 5. Il risultato è quanto contenuto nel riquadro C.

Il concetto di stato

Gli esempi visti finora consentono di fare le seguenti considerazioni di carattere generale:

- La specifica (astratta) di un problema consiste nella descrizione di uno stato iniziale (che descrive i dati del problema) e di uno stato finale (che descrive i risultati attesi).
- È necessario individuare una rappresentazione degli oggetti coinvolti (e dunque dello stato) che sia direttamente manipolabile dall'esecutore prescelto.
- Un algoritmo è una sequenza di passi elementari che, se intrapresi da un esecutore, comportano ripetute modifiche dello stato fino al raggiungimento dello stato finale desiderato.
- Le azioni di base che l'esecutore è in grado di effettuare devono dunque prevedere, tra le altre, azioni che hanno come effetto cambiamenti dello stato.

Un'astrazione dello stato

Introduciamo una possibile astrazione del concetto di stato, di cui faremo spesso uso in seguito.

Stato

Uno stato è un insieme di associazioni tra nomi simbolici e valori.

In uno stato, ad ogni nome simbolico è associato al più un valore.

Rappresentiamo una associazione tra il nome simbolico x ed il valore val con la notazione

$$x \rightsquigarrow val$$

Lo stato: esempi

Stati corretti

- $\{$nome $\rightsquigarrow$ Antonio, cognome $\rightsquigarrow$ Rossi, eta' $\rightsquigarrow$ 25$\}$
- $\{$importo $\rightsquigarrow$ \$1650, tasso $\rightsquigarrow$ 10%, interesse $\rightsquigarrow$ \$165 $\}$
- $\{$a $\rightsquigarrow$ 25, b $\rightsquigarrow$ 3, c $\rightsquigarrow$ 50$\}$

Stati non corretti

- $\{$nome $\rightsquigarrow$ Antonio, nome $\rightsquigarrow$ Paolo, eta' $\rightsquigarrow$ 25$\}$
- $\{$b $\rightsquigarrow$ 45, a $\rightsquigarrow$ 150, b $\rightsquigarrow$ 10$\}$

La soluzione facendo riferimento allo stato

- Scriviamo la soluzione precedente facendo riferimento alle modifiche dello stato:
-

Passo 1.	Associa il primo numero ad A (nello stato)
Passo 2.	Associa il secondo numero a B (nello stato)
Passo 3.	Associa il valore 0 a C (nello stato)
Passo 4.	Ripeti finché $B > 0$:
	1. calcola $A + C$ e associa il risultato a C (nello stato)
	2. calcola $B - 1$ e associa il risultato a B (nello stato)
Passo 5.	Il risultato è il valore associato a C (nello stato).

Gli stati per il calcolo di 3×2

Stati calcolati

- Passo 1 Scrivi il primo numero in A. $\{A \rightsquigarrow 3\}$
- Passo 2 Scrivi il secondo numero in B $\{A \rightsquigarrow 3, B \rightsquigarrow 2\}$
- Passo 3 Scrivi 0 in C. $\{A \rightsquigarrow 3, B \rightsquigarrow 2, C \rightsquigarrow 0\}$
- Passo 4.1 Calcola la somma di A e C e scrivila in C (B>0). $\{A \rightsquigarrow 3, B \rightsquigarrow 2, C \rightsquigarrow 3\}$
- Passo 4.2 Calcola la differenza tra B e 1 e scrivila in B (B>0) $\{A \rightsquigarrow 3, B \rightsquigarrow 1, C \rightsquigarrow 3\}$
- Passo 4.1 Calcola la somma di A e C e scrivila in C (B>0) $\{A \rightsquigarrow 3, B \rightsquigarrow 1, C \rightsquigarrow 6\}$
- Passo 4.2 Calcola la differenza tra B e 1 e scrivila in B (B>0) $\{A \rightsquigarrow 3, B \rightsquigarrow 0, C \rightsquigarrow 6\}$
- Passo 5 $\{A \rightsquigarrow 3, B \rightsquigarrow 0, C \rightsquigarrow 6\}$

Problema del prodotto

Sintetizziamo ulteriormente:

Passo 1. leggi (A);
Passo 2. leggi (B);
Passo 3. $C = 0$;
Passo 4. Ripeti finché $(B > 0)$:
1. $\{C = A + C$;
2. $B = B - 1;\}$

Passo 5. stampa (C);

- È quasi un programma in C!

Soluzione induttiva

La soluzione precedente utilizza, per calcolare, il paradigma *procedurale*, ovvero un algoritmo iterativo. Un modo diverso di calcolare è quello che utilizza definizioni induttive delle funzioni. Ad esempio per il prodotto abbiamo:

$$\text{prodotto}\ (A, 0) \quad = 0 \qquad (1)$$
$$\text{prodotto}\ (A, B) \quad = A + \text{prodotto}(A, B-1) \text{ se } B \neq 0 \qquad (2)$$

- In questo caso non c'e' lo stato, ma la computazione avviene per *riduzione*

  $\text{prodotto}(3, 2)$

  $\text{prodotto}(3, 2) = 3 + \text{prodotto}(3, 1)$ utilizzo (2) con $A = 3$ e $B = 2$

  $3 + \text{prodotto}(3, 1) = 3 + 3 + \text{prodotto}(3, 0)$ utilizzo (2) con $A = 3$ e $B = 1$

  $3 + 3 + \text{prodotto}(3, 0) = 3 + 3 + 0$ utilizzo (1) con $A = 3$ e $B = 0$

  $3 + 3 + 0 = 6$ calcolo il risultato il modo di esprimere il calcolo è diverso.

Il linguaggio C

I linguaggi di programmazione ad alto livello sono linguaggi formali (sintassi e semantica formalmente definite) però sono compatti e comprensibili. Le tipologie di linguaggi sono:

- procedurali o imperativi (C, Pascal, Fortran, ecc)
- object-oriented (Java, C++, ecc)
- funzionali (Caml, Haskel, ecc)
- logici (Prolog)

I linguaggi imperativi a cui appartiene il C sono quelli più simili come modello di calcolo al linguaggio macchina.

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

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

- Questo programma stampa sullo schermo una riga di testo:

```
Ciao mondo!
>
```

- Vediamo in dettaglio ogni riga del programma.

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

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

`main()`

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

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

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

`#include <stdio.h>`

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

Note:

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

Alcune varianti del programma

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

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

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

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

Un altro programma: area di un rettangolo

```
#include <stdio.h>

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

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

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

Quando viene eseguito stampa:

```
Area: 12
>
```

Le variabili

Servono a rappresentare, nei programmi, le associazioni (modificabili) dello stato

Una variabile è caratterizzata dalle seguenti proprietà:

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

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

Le variabili (cont.)

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

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

Area del rettangolo

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

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

Variabili numeriche

Variabili intere

- per dichiarare variabili intere si può usare il tipo `int`
- i valori di tipo `int` sono interi il cui valore massimo e minimo dipende dal compilatore che si utilizza. **Esempio:** per il compilatore gcc (usato in ambiente Unix) $+/-$ 2147483648
- riprenderemo il discorso quando parleremo della rappresentazione.
- in C esistono altri tipi per variabili intere (`short`, `long`) — li vedremo più avanti

Variabili razionali

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

Area del rettangolo

`base = 3;`
è un'istruzione di assegnamento

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

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

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

- una variabile all'interno di una espressione sta per il valore ad essa associato in quel momento

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

Area del rettangolo

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

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

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

Struttura dei programmi C

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

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

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

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

Un programma C deve contenere nell'ordine:

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

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

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

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

  - la parte esecutiva. Nell'esempio:

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

La parte dichiarativa

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

Esempio:

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

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

Dichiarazioni di variabili

- Abbiamo già visto esempi di dichiarazioni di variabili.

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

- Ad ogni variabile viene attribuito, al momento della dichiarazione, un tipo

  $\Longrightarrow$ specifica l'insieme dei valori che la variabile può assumere

- La dichiarazione può anche attribuire un valore iniziale alla variabile (inizializzazione)

```
int x = 0;
```

- Variabili dello stesso tipo possono essere dichiarate contemporaneamente

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

(ma inizializzate singolarmente)

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

Area di un rettangolo di dimensioni lette da tastiera

```
#include <stdio.h>

main()
{
  int base, altezza, area;

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

  area = base * altezza;

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

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

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

N.B. il precedente valore della variabile `base` va perduto

Esempio di esecuzione

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

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

Dichiarazioni di costanti (variabili *read-only*)

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

  Esempio:

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

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

  Esempio:

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

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

Uso di costanti

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

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

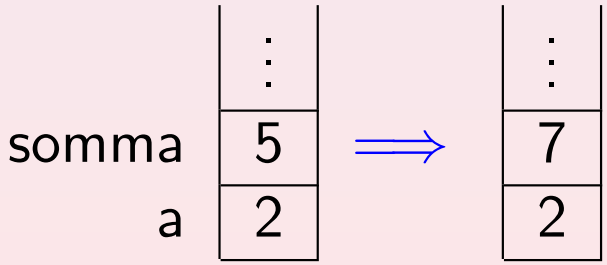
  Senza l'uso della costante si dovrebbero rimpiazzare nel codice tutte le occorrenze di `3.14` in `3.1415` ...

Assegnamento

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

Esempio:

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



Esempio:

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

Osservazioni sull'assegnamento

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

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

  Esempio: Quali istruzioni sono corrette e quali no?

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

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

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

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

- Tutti i programmi che abbiamo visto fino ad ora prevedono istruzioni che vengono eseguite in sequenza. Ma come si fa ad esempio a leggere 2 interi e a stampare il valore massimo?
- Cioè come si fa ad eseguire un istruzione solo se si verifica una determinata condizione?
- la risposta sono le istruzioni condizionali (esistono praticamente identiche in quasi tutti i linguaggi di programmazione).
- Per poter definire le istruzioni condizionali è necessario poter esprimere le condizioni.
- Le condizioni si esprimono attraverso le espressioni booleane.

Espressioni booleane

- In C non esiste un tipo Booleano $\Longrightarrow$ si usa il tipo **int** :

  falso $\Longleftrightarrow$ 0

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

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

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

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

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

Operatori logici

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

Semantica:

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

0 . . . falso

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

Esempio:

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

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

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

Esempio:

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

è equivalente a

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

I caratteri: il tipo char

Esempio:

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

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

Esempio:

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

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

Come va usato il codice dei caratteri

- Convertiamo una lettera minuscola in maiuscolo:

  Esempio:

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

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

  Esempio:

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

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

Istruzione **if-else**

Sintassi:

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

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

Semantica:

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

```
int temperatura;

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

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

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

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

Istruzione **if**

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

Sintassi:

```
if (espressione)
   istruzione
```

Semantica:

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

Esempio:

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

=> 18 ↩
Arrivederci

=> 30 ↩
Fa caldo
Arrivederci

Blocco

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

  Sintassi:

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

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

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

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

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

If annidati (in cascata)

- Si hanno quando l'istruzione del ramo then o else è un'istruzione **if o if-else.**

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

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

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

Osservazioni:

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

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

- non serve che la seconda condizione sia composta

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

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

- **Attenzione**: il seguente codice

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

  ha ben altro significato (quale?)

Ambiguità dell'else

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

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

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

    printf("b negativo");
```

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

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

  printf("a negativo");
```

Esercizio

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

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

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

Algoritmo: `determina tipo di triangolo`

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

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

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

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

Esercizio

Risolvere il problema del triangolo utilizzando il seguente algoritmo:

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

Iterazione determinata e indeterminata

- Le istruzioni iterative permettono di ripetere determinate azioni più volte:
  - un numero di volte fissato $\Longrightarrow$ iterazione determinata
    Esempio:
    ```
    fai un giro del parco di corsa per 10 volte
    ```
  - finchè una condizione rimane vera $\Longrightarrow$ iterazione indeterminata
    Esempio:
    ```
    finche' non sei sazio
    prendi una ciliegia dal piatto e mangiala
    ```

Istruzione **while**

Permette di realizzare l'iterazione in C.

Sintassi:

while `(espressione)`
`istruzione`

- `espressione` è la guardia del ciclo
- `istruzione` è il corpo del ciclo (può essere un blocco)

Semantica:

1. viene valutata l'`espressione`
2. se è vera si esegue `istruzione` e si torna ad eseguire l'intero `while`
3. se è falsa si termina l'esecuzione del `while`

- Nota: se `espressione` è falsa all'inizio, il ciclo non fa nulla.

Iterazione determinata

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

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

```
int conta, dato, somma;
printf("Immetti 10 interi:  ");
somma = 0;
conta = 0;
while (conta < 10) {
  scanf("%d", &dato);
  somma = somma + dato;
  conta = conta + 1;
}
printf("La somma e' %d\n", somma);
```

Esempio: Leggere un intero N seguito da N interi e calcolare la somma di questi ultimi.

- Simile al precedente: il numero di ripetizioni necessarie non è noto al momento della scrittura del programma ma lo è al momento dell'esecuzione del ciclo.

```
int lung, conta, dato, somma;
printf("Immetti la lunghezza della sequenza ");
printf("seguita dagli elementi della stessa:  ");
scanf("%d", &lung);
somma = 0;
conta = 0;
while (conta < lung) {
  scanf("%d", &dato);
  somma = somma + dato;
  conta = conta + 1;
}
printf("La somma e' %d\n", somma);
```

Esempio: Leggere 10 interi positivi e stamparne il massimo.

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

```
int conta, dato, massimo;
printf("Immetti 10 interi:  ");
massimo = 0;
conta = 0;
while (conta < 10) {
  scanf("%d", &dato);
  if (dato > massimo)
    massimo = dato;
  conta = conta + 1;
}
printf("Il massimo e' %d\n", massimo);
```

Esercizio

Leggere 10 interi arbitrari e stamparne il massimo.

Istruzione **for**

- I cicli visti fino ad ora hanno queste caratteristiche comuni:
  - utilizzano una variabile di controllo
  - la guardia verifica se la variabile di controllo ha raggiunto un limite prefissato
  - ad ogni iterazione si esegue un'azione
  - al termine di ogni iterazione viene incrementato (decrementato) il valore della variabile di controllo

Esempio: Stampare i numeri pari da 0 a N.

```
i = 0; /* Inizializzazione della var.  di controllo */
while (i <= N) { /* guardia */
  printf("%d ", i); /* Azione da ripetere */
  i=i+2; /* Incremento var.  di controllo */
}
```

- L'istruzione **for** permette di gestire direttamente questi aspetti:

```
for (i = 0; i <= N; i=i+2)
  printf("%d", i);
```

Sintassi:

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

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

Semantica: l'istruzione **for** precedente è equivalente a

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

Esempio:

```
for (i = 1; i <= 10; i=i+1)      ⟹   i: 1, 2, 3, ..., 10
for (i = 10; i >= 1; i=i-1)      ⟹   i: 10, 9, 8, ..., 2, 1
for (i = -4; i <= 4; i = i+2)    ⟹   i: -4, -2, 0, 2, 4
for (i = 0; i >= -10; i = i-3)   ⟹   i: 0, -3, -6, -9
```

- In realtà, la sintassi del **for** è

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

  dove `espr-1`, `espr-2` e `espr-3` sono delle espressioni qualsiasi (in C anche l'assegnamento è un'espressione ...).
- È buona prassi:
  - 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` può 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**

Esempio: Leggere 10 interi positivi e stamparne il massimo.

```
int conta, dato, massimo;
printf("Immetti 10 interi:  ");
massimo = 0;
for (conta=0; conta<10; conta=conta+1)
{
  scanf("%d", &dato);
  if (dato > massimo)
    massimo = dato;
}
printf("Il massimo e' %d\n", massimo);
```

Iterazione indeterminata

- 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: Leggere una sequenza di interi che termina con 0 e calcolarne la somma.

  Input: $n_1, \ldots, n_k, 0$ (con $n_i \neq 0$)

  Output: $\sum_{i=1}^{k} n_i$

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

Rappresentazione binaria

- Per informazione intendiamo tutto quello che viene manipolato da un calcolatore:
  - numeri (naturali, interi, reali, ...)
  - caratteri
  - immagini
  - suoni
  - programmi
  - ...
- La più piccola unità di informazione memorizzabile o elaborabile da un calcolatore, il bit, corrisponde allo stato di un dispositivo fisico che viene interpretato come 1 o 0.
- 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 concetto 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.

Rappresentazione posizionale

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

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

  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 dalla base b:

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

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

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

Intervallo di rappresentazione

- I numeri rappresentabili in base b con n posizioni (cifre) vanno da 0 a $b^n - 1$.

3 cifre in base 10 :	da	0	a	$999 = 10^3 - 1$
8 cifre in base 2 :	da	0	a	$255 = 2^8 - 1$
16 cifre in base 2 :	da	0	a	$65\,535 = 2^{16} - 1$
32 cifre in base 2 :	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: da base b a base 10

- Usando direttamente

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

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.

Conversioni di base: da base 10 a base b

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

- Vogliamo determinare le cifre $c_0, c_1, \ldots, c_{k-1}$
- Consideriamo la divisione di N per b:

$$N = R + b \cdot Q \qquad (0 \leq R < b)$$
$$= c_0 + b \cdot (c_1 + b \cdot (\cdots))$$
$$\Downarrow$$
$$R = c_0 \quad \text{ovvero, il resto } R \text{ della divisione di } N \text{ per } b \text{ dà}$$
$$c_0 \quad \text{(cifra meno significativa)}$$
$$Q = c_1 + b \cdot (\cdots)$$

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

Conversione da base 10 a base b

```
i = 0;
while (num != 0) {
    c[i] = num % b;
    num = num / b;
    i = i+1; }
```

N.B. Le cifre vengono determinate dalla meno significativa alla più significativa.

Esempio: $(25)_{10} = (???)_2 = (11001)_2$

$N : b$	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

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

Rappresentazione di numeri interi

- Dobbiamo rappresentare anche il segno: 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 alla base

Rappresentazione in complemento alla base

In quanto segue b indica la base e n indica il numero complessivo di cifre.

- Con base b e n cifre, abbiamo a disposizione b^n configurazioni distinte.
- Utilizziamo metà delle configurazioni per rappresentare numeri positivi e l'altra metà per rappresentare numeri negativi.

$$||X|| = \begin{cases} X & \text{se } X \geq 0 \\ b^n - |X| & \text{se } X < 0 \end{cases}$$

- in questo modo si rappresentano gli interi relativi nell'intervallo $[-b^n/2,\ b^n/2)$
  - se $X \geq 0$: $||X||$ è compresa in $[0,\ b^n/2)$
  - se $X < 0$: $||X||$ è compresa in $[b^n/2,\ b^n)$
- lo 0 ha una sola rappresentazione

Rappresentazione in complemento alla base

N	$b = 10$ e $n = 1$	$b = 2$ e $n = 3$
-5	5	
-4	6	100
-3	7	101
-2	8	110
-1	9	111
0	0	000
1	1	001
2	2	010
3	3	011
4	4	

- se $b = 2 \Longrightarrow$ rappresentazione in complemento a 2
  - rappresentazione degli interi relativi nell'intervallo $[-2^{n-1}, 2^{n-1})$
  - positivi: la cifra più significativa è 0 (rappresentati nella parte inferiore dell'intervallo)
  - negativi: la cifra più significativa è 1 (rappresentati nella parte superiore dell'intervallo)

Operazione di complementazione

Vogliamo determinare un algoritmo per determinare la rappresentazione in complemento alla base di $-X$, data quella di X. Indipendentemente dal segno di X, abbiamo:

$$||X|| + || - X|| = |X| + b^n - |X| = b^n$$

per cui

$$|| - X|| = b^n - ||X||$$

o equivalentemente

$$|| - X|| = b^n - 1 - ||X|| + 1$$

Operazione di complementazione

- Supponiamo:

$$||X|| = \sum_{i=0}^{n-1} c_i \cdot b^i$$

e ricordiamo che la rappresentazione di $b^n - 1$ è

$$\sum_{i=0}^{n-1} (b-1) \cdot b^i$$

- Otteniamo:

$$\begin{aligned} ||-X|| &= b^n - 1 - ||X|| + 1 \\ &= (\sum_{i=0}^{n-1} (b-1) \cdot b^i) - (\sum_{i=0}^{n-1} c_i \cdot b^i) + 1 \end{aligned}$$

Operazione di complementazione

- Sia ora k la prima posizione significativa di $||X||$, ovvero la prima cifra (a partire da destra) diversa da 0. Abbiamo allora:

$$
\begin{aligned}
||-X|| &= (\sum_{i=0}^{n-1}(b-1)\cdot b^i) - (\sum_{i=0}^{n-1} c_i \cdot b^i) + 1 \\
&= (\sum_{i=0}^{n-1}(b-1)\cdot b^i) - (\sum_{i=k}^{n-1} c_i \cdot b^i) + 1 \\
&= (\sum_{i=k+1}^{n-1}((b-1)-c_i)\cdot b^i) + ((b-1)-c_k)\cdot b^k + \\
&\quad (\sum_{i=0}^{k-1}(b-1)\cdot b^i) + 1
\end{aligned}
$$

- Osserviamo ora che $(\sum_{i=0}^{k-1}(b-1)\cdot b^i) + 1 = b^k$.

Operazione di complementazione

- Sia ora k la prima posizione significativa di $||X||$, ovvero la prima cifra (a partire da destra) diversa da 0. Abbiamo allora:

$$
\begin{aligned}
||-X|| &= (\sum_{i=0}^{n-1}(b-1)\cdot b^i) - (\sum_{i=0}^{n-1} c_i \cdot b^i) + 1 \\
&= (\sum_{i=0}^{n-1}(b-1)\cdot b^i) - (\sum_{i=k}^{n-1} c_i \cdot b^i) + 1 \\
&= (\sum_{i=k+1}^{n-1}((b-1)-c_i)\cdot b^i) + ((b-1)-c_k)\cdot b^k + \quad b^k
\end{aligned}
$$

Operazione di complementazione

$$||-X|| = \left(\sum_{i=k+1}^{n-1}((b-1)-c_i)\cdot b^i\right) + ((b-1)-c_k)\cdot b^k + b^k$$

$$= \left(\sum_{i=k+1}^{n-1}((b-1)-c_i)\cdot b^i\right) + (b-c_k)\cdot b^k$$

- L'ultimo addendo è 0, poichè $c_i = 0$, per ogni $i = 0, \ldots, k-1$.
- Come possiamo leggere quanto ottenuto?

  La rappresentazione di $-X$ si ottiene da quella di X:

  1. ricopiando gli zeri meno significativi
  2. complementando alla base la prima cifra significativa
  3. complementando alla base meno uno le rimanenti cifre

Operazione di complementazione

$$||-X|| = \left(\sum_{i=k+1}^{n-1}((b-1)-c_i)\cdot b^i\right) + ((b-1)-c_k)\cdot b^k + b^k$$

$$= \left(\sum_{i=k+1}^{n-1}((b-1)-c_i)\cdot b^i\right) + (b-c_k)\cdot b^k + \left(\sum_{i=0}^{k-1} c_i \cdot b^i\right)$$

- L'ultimo addendo è 0, poichè $c_i = 0$, per ogni $i = 0, \ldots, k-1$.
- Come possiamo leggere quanto ottenuto?

  La rappresentazione di $-X$ si ottiene da quella di X:

  1. ricopiando gli zeri meno significativi
  2. complementando alla base la prima cifra significativa
  3. complementando alla base meno uno le rimanenti cifre

Operazione di complementazione

Esempio: $b = 3$, $n = 4$, $||X|| = 0210$ (dunque $X = (21)_{10}$)

$$||-X|| = 2020$$

Verifichiamo:

- $2020 = (60)_{10}$
- $3^4 - 60 = 81 - 60 = 21 = |-X|$

Nel caso del complemento a 2 abbiamo più semplicemente:

- si lasciano inalterate tutte le cifre fino al primo 1 compreso
- si invertono le rimanenti cifre

Complemento a 2

Esempio: Rappresentazione di -298 in complemento a 2:

- di quante cifre minimo abbiamo bisogno per rappresentare 298 ?
- $-2^{10-1} \leq 298 < 2^{10-1}$: 10 cifre
- $298 = 256 + 32 + 8 + 2$, $||298||$ in base $2 = 100101010$
- $||298||$ in complemento a 2 con 10 cifre$= 0100101010$
- $||-298||$ in complemento a 2 con 10 cifre $= 1011010110$
- $||298||$ in complemento a 2 con 11 cifre$= 00100101010$
- $||-298||$ in complemento a 2 con 11 cifre$= 11011010110$

Operazioni su interi relativi in complemento a 2

Somma di due numeri

- si effettua bit a bit
- non è necessario preoccuparsi dei segni
- il risultato sarà corretto (in complemento a 2 se negativo)
- può verificarsi trabocco (overflow) $\Longrightarrow$ il risultato non è corretto

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

Operazioni su interi relativi in complemento a 2

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

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

rip.	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 questi casi non si ha trabocco.

rip.	01000		10000
+9	01001	-9	10111
+8	01000	-8	11000
-15	10001	+15	01111
(e non +17)		(e non −17)	

Si ha trabocco quando il riporto sul bit di segno è diverso dall'ultimo riporto.

Operazioni su interi relativi in complemento a 2

Differenza tra due numeri: si somma al primo il complemento del secondo

Esempio: $n = 5$, intervallo di rappresentazione: da -16 a 15

rip.		11111
	+9	01001
	-9	10111
	0	00000

	11001
+9	01001
-3	11101
+6	00110

Numeri frazionari

- Numeri reali compresi tra 0 e 1: si rappresentano comunemente come

$$N = 0.c_{-1}c_{-2}\ldots c_{-n}$$

- Il peso delle cifre dipende, al solito, dalla loro posizione e dalla base prescelta

$$N_b = c_{-1}\cdot b^{-1} + c_{-2}\cdot b^{-2} + \cdots + c_{-n}\cdot b^{-n} = \sum_{i=-n}^{-1} c_i \cdot b^i$$

Esempio: Consideriamo $b = 10$ ed il numero 0.587

$$0.587_{10} = 5\cdot 10^{-1} + 8\cdot 10^{-2} + 7\cdot 10^{-3}$$

Numeri frazionari

$$N_b = \sum_{i=-n}^{-1} c_i \cdot b^i \qquad (\bullet)$$

- Nel caso di un numero frazionario in binario, possiamo usare la $(\bullet)$ per convertirlo in base 10

  Esempio: Convertiamo in base 10 il numero frazionario binario 0.1011_2

$$0.1011_2 = 1 \cdot 2^{-1} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 0.6875_{10}$$

- La rappresentazione dei numeri frazionari può introdurre approssimazioni dovute alla limitatezza delle cifre dopo la virgola.
- L'approssimazione è comunque inferiore a b^{-n} dove n è il numero di cifre utilizzate.

Conversione di un numero frazionario da base 10 a base 2

- Il metodo più semplice consiste nell'effettuare una sequenza di moltiplicazioni per 2 prendendo ad ogni passo la parte intera del risultato come cifra binaria della rappresentazione
- **Esempio:** Convertiamo 0.125 in base 2

$$\begin{array}{rcr|l} 0.125 \times 2 & = & 0.25 & 0 \\ 0.25 \times 2 & = & 0.5 & 0 \\ 0.5 \times 2 & = & 1.0 & 1 \end{array}$$

- In questo caso abbiamo una rappresentazione esatta su 3 cifre ($0.125 = 1/8$)

$$0.125_{10} = 0.001_2$$

Conversione di un numero frazionario da base 10 a base 2

- **Esempio:** Convertiamo 0.587_{10} in base 2

0.587×2	$=$	1.174	1
0.174×2	$=$	0.348	0
0.348×2	$=$	0.696	0
0.696×2	$=$	1.392	1
0.392×2	$=$	0.784	0
0.784×2	$=$	1.568	1
0.568×2	$=$	$\ldots$	

- Quindi la rappresentazione di 0.587_{10} in base 2 è:
  - 0.1001_2 con 4 cifre (approssimazione accurata entro 2^{-4})
  - 0.100101_2 con 6 cifre (approssimazione accurata entro 2^{-6})

L'aritmetica reale

- L'insieme dei reali (e dei razionali) è infinito $\Longrightarrow$ non è possibile rapprentarlo tutto

  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

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

L'aritmetica reale

- Limitazioni della rappresentazione:
  - k bit per la parte intera $\Longrightarrow (-2^k,\ 2^k)$
  - m bit per la parte frazionaria $\Longrightarrow$ precisione $\leq 2^{-m}$

Rappresentazione in virgola mobile (floating point)

Utilizza la notazione esponenziale. Si esprime il numero come prodotto di due parti

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

Esempio:

$$1150 = 1.15 \times 10^3$$

ma anche

$$1150 = 0.115 \times 10^4$$

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 $1/b \leq |m| < 1$
- **Esempio:** $1150 = \underbrace{0.115}_{\text{mantissa}} \times \underbrace{10^4}_{\text{caratteristica}}$
- In binario:

  Esempio: $101 = \underbrace{0.101}_{\text{mantissa}} \times \underbrace{2^{011}}_{\text{caratteristica}}$

Rappresentazione in virgola mobile

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

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

$$(0.1) \quad 1/2 \leq |m| \leq \sum_{i=1}^{k} 2^{-i} \quad (0.11\ldots1)$$

$$|e| \leq 2^{h-1} - 1$$

- 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)

Rappresentazione in virgola mobile

Insieme $\mathcal{F}$ dei numeri rappresentabili in virgola mobile

- sottoinsieme finito dei numeri razionali rappresentabili
- simmetrico rispetto allo 0
- gli elementi non sono uniformemente distribuiti sull'asse reale
  - densi intorno allo 0

    $m_1 = 0.10, m_2 = 0.11, e_1 = -5$
    $X_1 = 0.10 \times 10^{-5} = 0.0000010$
    $X_2 = 0.11 \times 10^{-5} = 0.0000011$
  - radi intorno al massimo rappresentabile

    $m_1 = 0.10, m_2 = 0.11, e_2 = 5$
    $X_1 = 0.10 \times 10^{5} = 10000$
    $X_2 = 0.11 \times 10^{5} = 11000$
- molti razionali non appartengono ad $\mathcal{F}$ (ed es. $1/3$, $1/5$, ...)
- non è chiuso rispetto ad addizioni e moltiplicazioni
- per rappresentare un reale X si sceglie l'elemento di $\mathcal{F}$ più vicino ad X
- la funzione che associa ad un reale X l'elemento di $\mathcal{F}$ 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

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

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

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

Istruzione **switch**

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

Sintassi:

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

Semantica:

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

Esempio:

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

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

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

Esempio:

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

Osservazioni sull'istruzione **switch**

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

Esempio: Il seguente frammento di codice è sbagliato:

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

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

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

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

Semantica:

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

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

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

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

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

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

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

```
int b;
printf("Immetti un numero tra 1 e 6:  ");
scanf("%", &b);

switch (b) {
  case 1:  case 2:  case 3:  case 5:  printf("Numero primo\n");
  case 4:  case 6:                    printf("Numero non primo\n");
  default :                           printf("Valore non valido!\n");
}
```

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

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

Iterazione indeterminata

- 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: Leggere una sequenza di interi che termina con 0 e calcolarne la somma.

  Input: $n_1, \ldots, n_k, 0$ (con $n_i \neq 0$)

  Output: $\sum_{i=1}^{k} n_i$

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

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
```

$\Longrightarrow$ una iterazione viene eseguita comunque.

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

```
main() {
int lunghezza = 0; /* lunghezza della sequenza */
int dato; /* dato letto di volta in volta */
printf("Inserisci una sequenza di interi (0 fine seq.)\n");
do {
  scanf("%d", &dato);
  lunghezza=lunghezza+1;
} while (dato != 0);
printf("La sequenza e' lunga %d\n", lunghezza - 1);
}
```

- Nota: lo 0 finale non è conteggiato (non fa parte della sequenza, fa da terminatore)

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

$$\begin{aligned} MCD(12, 8) &= 4 \\ MCD(12, 6) &= 6 \\ MCD(12, 7) &= 1 \end{aligned}$$

- 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 MCD 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 e' trovato un divisore comune
{
  if mcd divide sia m che n
    si e' 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 a 1)
- per verificare se si è trovato il MCD si utilizza una variabile booleana (nella guardia del ciclo)
- Implementazione in C ...

```
int m, n;                /* i due numeri letti */
int mcd;                /* il massimo comun divisore */
int trovato = 0;        /* var. booleana: inizialmente false */
if (m <= n)             /*inizializza mcd al minimo tra m e n*/
  mcd = m;
else
  mcd = n;
while (mcd > 1 && !trovato)
  if ((m % mcd == 0) && (n % mcd == 0))
    /* mcd divide entrambi */
    trovato = 1;
  else
    mcd = mcd -1;
printf("MCD di %d e %d:  %d", m, n, mcd);
```

Quante volte viene eseguito il ciclo?

- caso migliore: 1 volta (quando m divide n o viceversa)
  es. MCD(500, 1000)
- caso peggiore: *min*(m,n) volte (quando MCD(m,n)=1)
  es. MCD(500, 1001)
- l'algoritmo si comporta *male* se m e n sono grandi e MCD(m,n) è piccolo

Metodo di Euclide per il calcolo del MCD

- Permette di ridursi più velocemente a numeri più piccoli, sfruttando le seguenti proprietà:

  MCD(x, x) = x
  MCD(x, y) = MCD(x-y, y) se x>y
  MCD(x, y) = MCD(x, y-x) se y>x

- I divisori 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.

Esempio:

m	n	maggiore - minore
210	63	147
147	63	84
84	63	21
21	63	42
21	42	21
21	21	

Algoritmo: di Euclide per il calcolo del MCD

```
int m,n;
scanf("%d%d", &m, &n);
while (m != n)
  if (m > n)
    m = m - n;
  else
    n = n - m;

printf("MCD: %d\n", m);
```

- 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

- Per assicurarci che l'algoritmo venga eseguito su valori corretti, possiamo inserire una verifica sui dati in ingresso, attraverso un ciclo di lettura

  Proposte?

```
do {
  printf("Immettere due interi positivi:  ");
  scanf("%d%d", &m, &n);
  if (m <= 0 || n <= 0)
    printf("Errore:  i numeri devono essere > 0!\n");
} while (m <= 0 || n<= 0);
```

Metodo di Euclide con i resti per il calcolo del MCD

- Cosa succede se $m \gg n$?

Esempio:

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?
- Metodo di Euclide: sia

$$m = n \cdot k + r \quad (\text{con } 0 \leq r < m)$$

$$\begin{array}{lcll} MCD(m, n) & = & n & \text{se } r=0 \\ MCD(m, n) & = & MCD(r, n) & \text{se } r \neq 0 \end{array}$$

Algoritmo di Euclide con i resti per il calcolo del MCD

```
leggi m ed n
while m ed n sono entrambi ≠ 0
{ sostituisci il maggiore tra m ed n con
  il resto della divisione del maggiore per il minore
}
stampa il numero tra i due che e' diverso da 0
```

Esercizio

Tradurre l'algoritmo in C

Cicli annidati

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

  Esempio: Stampa della tavola pitagorica.

  Algoritmo

```
for ogni riga tra 1 e 10
  { for ogni colonna tra 1 e 10
    stampa riga * colonna
  stampa un a capo }
```

- Traduzione in C

```
int riga, colonna;
const int Nmax = 10; /* indica il numero di righe e di
colonne */
for (riga = 1; riga <= Nmax; riga=riga+1) {
  for (colonna = 1; colonna <= Nmax; colonna=colonna+1)
    printf("%d ", riga * colonna);
  putchar('\n'); }
```

Digressione sulle costanti: la direttiva `#define`

- Nel programma precedente, `Nmax` è una costante. Tuttavia la dichiarazione

  ```
  const int Nmax = 10;
  ```

  causa l'allocazione di memoria (si tratta duna dichiarazione di variabile *read only*)
- C'è un altro modo per ottenere un identificatore costante, che utilizza la direttiva **#define**.

  ```
  #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
- a differenza di **const** non alloca memoria

Assegnamento e altri operatori

- In C, l'operazione di assegnamento `x = exp` è un'espressione
  - il valore dell'espressione è il valore di `exp` (che è a sua volta un'espressione)
  - la valutazione dell'espressione `x = exp` ha un side-effect: quello di assegnare alla variabile `x` il valore di `exp`
- Dunque in realtà, "=" è un operatore (associativo a destra).
  Esempio: Qual'è l'effetto di `x = y = 4` ?
  - È equivalente a: `x = (y = 4)`
  - `y = 4` ... espressione di valore `4` con modifica (side-effect) di `y`
  - `x = (y = 4)` ... espressione di valore `4` con ulteriore modifica su `x`
- L'eccessivo uso di assegnamenti come espressioni rende il codice difficile da comprendere e quindi correggere/modificare.

Operatori di incremento e decremento

- Assegnamenti del tipo: `i = i + 1` / `i = i - 1` sono molto comuni.
  - operatore di incremento: `++`
  - operatore di decremento: `--`
- In realtà `++` corrisponde a due operatori:
- postincremento: `i++`
  - il valore dell'espressione è il valore di `i`
  - side-effect: incrementa `i` di `1`
- L'effetto di

```
int i,j;
i=6;
j=i++;
```

è j=6, i=7.

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

```
int i,j;
i=6;
j=++i;
```

è j=7, i=7.

(analogamente per `i--` e `--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 in questo modo!
  In un'istruzione di assegnamento non ci devono essere altri side-effect (oltre a quello dell'operatore di assegnamento) !!!
- Riscrivere, ad esempio, come segue:

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 quale è l'ordine di valutazione degli operandi nelle espressioni.

  Esempio:
  ```
  int x, y, z;
  x = 2;
  y = 4;
  z = x++ + (x * y);
  ```
- Quale è il valore di `z`?
  - se viene valutato prima `x++`: $2 + (3 * 4) = 14$
  - se viene valutato prima `x*y`: $(2 * 4) + 2 = 10$

Forme abbreviate dell'assegnamento

```
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;
```

Tipi di dato semplici

- Abbiamo visto nei primi esempi che il C tratta vari tipi di dato
  $\Longrightarrow$ le dichiarazioni associano variabili e costanti al corrispondente tipo
- Per tipo di dato si intende un insieme di valori e un insieme di operazioni che possono essere applicate ad essi.

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

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

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

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

Classificazione dei tipi

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

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

Tipi semplici built-in

- interi
- reali
- caratteri

Per ciascun tipo consideriamo i seguenti aspetti:

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

Tipi interi: interi con segno

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

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

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

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

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

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

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

Tipi interi: interi senza segno

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

Notazione per le costanti:

- decimale: come per interi con segno
- esadecimale: `0xA`, `0x2F4B`, ...
- ottale: `012`, `027513`, ... (almeno 2 cifre, prima cifra sempre 0)

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

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

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

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

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

Caratteri

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

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

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

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

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

Intervallo di definizione: dipende dal compilatore

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

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

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

Esempio:

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

Come non va usato il codice

- Confrontiamo:

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

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

- Non è sbagliato, però è pessimo stile di programmazione.
- Non è detto che il codice dei caratteri sia quello ASCII.
  $\Longrightarrow$ Il programma non sarebbe portabile.
- Vedremo presto un modo per sfruttare l'ordinamento tra caratteri molto utile.

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

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

Esempio:

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

Immetti due interi
> 18 25↩
18 25

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

Immetti due interi
> 18 25↩
18 32 25

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

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

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

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

Esempio:

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

Tipi reali

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

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

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

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

Costanti: con punto decimale o notazione esponenziale

Esempio:

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

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

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

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

Output con printf (per `float`):

- `%f` ... notazione in virgola fissa
  `%8.3f` ... 8 cifre complessive, di cui 3 cifre decimali

Esempio:

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

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

- `%e` (oppure `%E`) ... notazione esponenziale
  `%10.3e` ... 10 cifre complessive, di cui 3 cifre decimali

Esempio:

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

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

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

Riassunto degli specificatori di formato per i tipi reali:

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

Conversioni di tipo

Situazioni in cui si hanno conversioni di tipo

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

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

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

Conversioni implicite tra operandi di tipo diverso nelle espressioni

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

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

$$\texttt{int} \rightarrow \texttt{long} \rightarrow \texttt{float} \rightarrow \texttt{double} \rightarrow \texttt{long double}$$

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

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

Conversioni nell' assegnamento

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

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

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

  se si ha una retrocessione si può avere perdita di informazione

  Esempio:

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

- Se la conversione non è possibile si ha errore.

Conversioni esplicite (operatore di cast)

Sintassi: `(tipo) espressione`

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

  Esempio:

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

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

Input/output

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

Input/output (cont.)

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

  $\Longrightarrow$ il programmatore non si deve preoccupare di come ciò sia effettivamente realizzato
- Ogni volta che si effettua una operazione di lettura attraverso `getchar` viene acquisito il prossimo carattere dallo standard input e viene restituito il suo valore

  (analogamente per `scanf` che comporta l'acquisizione di uno o più caratteri a seconda delle specifiche di formato presenti . . .)
- Ogni volta che si effettua una operazione di scrittura (attraverso `putchar` o `printf`) tutti i valori coinvolti vengono convertiti in sequenze di caratteri e quest'ultime vengo accodate allo standard output.
- Tipicamente il sistema operativo consente di redirigere gli stream standard, ad esempio su uno o più file.

Formattazione dell'output con `printf`

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

Formattazione dell'input con `scanf`

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

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

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

Tipi di dato strutturati: Array

- I tipi di dato visti finora sono tutti semplici: `int`, `char`, `float`, ...
- ma i dati manipolati nelle applicazioni reali sono spesso complessi (o strutturati)
- 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 dell'array è detto lunghezza (o dimensione) dell'array
- Consentono di rappresentare tabelle, matrici, matrici n-dimensionali, ...

Array monodimensionali (o vettori)

- Supponiamo di dover rappresentare e manipolare la classifica di un campionato cui partecipano 16 squadre.
- È del tutto naturale pensare ad una tabella

Classifica

Squadra A	Squadra B	...	Squadra C
1^o posto	2^o posto		16^o posto

che evolve con il procedere del campionato

Classifica

Squadra B	Squadra A	...	Squadra C
1^o posto	2^o posto		16^o posto

Sintassi: dichiarazione di variabile di tipo vettore

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

Esempio: `int vet[6];`

dichiara un vettore di 6 elementi, ciascuno di tipo intero.

- All'atto di questa dichiarazione vengono riservate (allocate) 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).
- Ogni elemento del vettore è una variabile identificata dal nome del vettore e da un indice

  Sintassi: elemento di array `nome-array[espressione];`

  Attenzione: `espressione` deve essere di tipo intero ed il suo valore deve essere compreso tra 0 a lunghezza-1.

- **Esempio:**

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

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

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

- `vet[0]`, `vet[1]`, ecc. sono variabili intere come tutte le altre e dunque possono stare a sinistra dell'assegnamento (es. `vet[0] = 15`), cosi' come all'interno di espressioni (es. `vet[0] + a`).
- Come detto, l'indice del vettore è un'espressione.

```
index = 2;
vet[index+1] = 23;
```

Manipolazione di vettori

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

Esempio: Lettura e stampa di un vettore.

```
#include <stdio.h>
#define LUNG 5

main ()
{
int v[LUNG]; /* vettore di LUNG elementi, indicizzati da 0 a LUNG-1 */
int i;

for (i = 0; i < LUNG; i++) {
  printf("Inserisci l'elemento di indice %d:  ", i);
  scanf("%d", &v[i]);
}
printf("Indice Elemento\n");
for (i = 0; i < LUNG; i++) {
  printf("%6d %8d\n", i, v[i]); }

}
```

Inizializzazione di vettori

- Gli elementi del vettore possono essere inizializzati con valori costanti (valutabili a tempo di compilazione) 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};
  ```
  $\Longrightarrow$ errore!

- se i valori iniziali sono meno degli elementi, i rimanenti vengono posti a 0

  `int n[10] = {3};` azzera i rimanenti 9 elementi del vettore
  `float af[5] = {0.0};` pone a 0.0 i 5 elementi
  `int x[5] = {};` errore!

- se ci sono più inizializzatori di elementi, si ha un errore a tempo di compilazione

  Esempio: `int v[2] = {1, 2, 3};` errore!

- se si mette una sequenza di valori iniziali, si può omettere la lunghezza (viene presa la lunghezza della sequenza)

  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 ai singoli elementi.
- Ad esempio, non si possono effettuare direttamente delle assegnazioni tra vettori.

Esempio:

```
int a[3] = {11, 22, 33};
int b[3];
b = a;                          errore!
```

Esempi

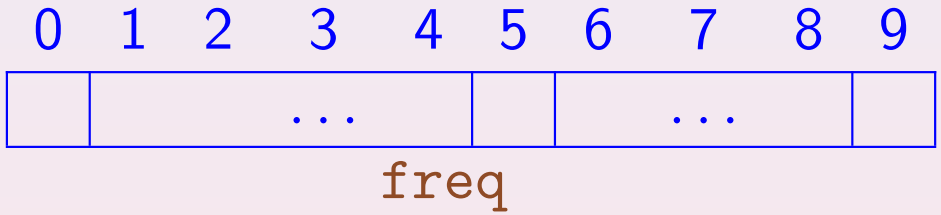
- Calcolo della somma degli elementi di un vettore.

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

- Leggere N interi e stampare i valori maggiori di un valore intero y letto in input.

```
#include <stdio.h>
#define N 4
main()   {
int ris[N];
int y, i;
printf("Inserire i %d valori:\n", N);
for (i = 0; i < N; i++) {
  printf("Inserire valore n.  %d:  ", i+1);
  scanf("%d", &ris[i]);     }
printf("Inserire il valore y:\n");
scanf("%d", &y);
printf("Stampa i valori maggiori di %d:\n", y);
for (i = 0; i < N; i++)
  if (ris[i] > y)
    printf("L'elemento %d:  %d e' maggiore di %d\n",
            i+1, ris[i],y);
}
```

- Leggere una sequenza di caratteri terminata dal carattere `\n` di fine linea e stampare le frequenze delle cifre da `'0'` a `'9'`.
- utilizziamo un vettore `freq` di 10 elementi nel quale memorizziamo le frequenze dei caratteri da `'0'` a `'9'`



`freq[0]` conta il numero di occorrenze di `'0'`

...

`freq[9]` conta il numero di occorrenze di `'9'`

- utilizziamo un ciclo per l'acquisizione dei caratteri in cui aggiorniamo una delle posizioni dell'array tutte le volte che il carattere letto è una cifra

```
int i; char ch;
int freq[10] = {0};
do {
  ch = getchar();
  switch (ch) {
    case '0':  freq[0]++; break;
    case '1':  freq[1]++; break;
    case '2':  freq[2]++; break;
    case '3':  freq[3]++; break;
    case '4':  freq[4]++; break;
    case '5':  freq[5]++; break;
    case '6':  freq[6]++; break;
    case '7':  freq[7]++; break;
    case '8':  freq[8]++; break;
    case '9':  freq[9]++; break;
    }
} while (ch != '\n');
printf("Le frequenze sono:\n");
for (i = 0; i < 10; i++)

  printf("Freq.  di %d:  %d\n", i, freq[i]);
```

- Nel ciclo **do-while**, il comando `switch` può essere rimpiazzato da un **if** come segue

```
if (ch >= '0' && ch <= '9')
  freq[ch - '0']++;
```

  Infatti:

- i codici dei caratteri da `'0'` a `'9'` sono consecutivi
- dato un carattere `ch`, l'espressione intera `ch - '0'` è la distanza del codice di `ch` dal codice del carattere `'0'`. In particolare:
  - `'0' - '0' = 0`
  - `'1' - '0' = 1`
  - ...
  - `'9' - '0' = 9`

- Leggere da tastiera i risultati (double) di 20 esperimenti. Stampare il numero d'ordine ed il valore degli esperimenti per i quali il risultato e' minore del 50% della media.

```
#include <stdio.h>
#define DIM 20
main() {
double ris[DIM], media;
int i;
/* inserimento dei valori */
printf("Inserire i %d risultati dell'esperimento:\n", DIM);
for (i = 0; i < DIM; i++) {
  printf("Inserire risultato n.  %d:  ", i);
  scanf("%g", &ris[i]); }
/* calcolo della media */
media = 0.0;
for (i = 0; i < DIM; i++)
    media = media + ris[i];
media = media/DIM;
printf("Valore medio:  %g\n", media);
/* stampa dei valori minori di media*0.5 */
printf("Stampa dei valori minori di media*0.5:\n");
for (i = 0; i < DIM; i++)
    if (ris[i] < media * 0.5)
      printf("Risultato n.  %d:  %g\n", i, ris[i]);        }
```

Array multidimensionali

Sintassi: dichiarazione

`tipo-elementi nome-array` $[\text{lung}_1][\text{lung}_2]\cdots[\text{lung}_n]$;

Esempio: `int mat[3][4];` $\Longrightarrow$ matrice 3×4

- Per ogni dimensione i l'indice va da 0 a lung_i-1.

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

Esempio: `int marketing[10][5][12]`

(indici potrebbero rappresentare: prodotti, venditori, mesi dell'anno)

Accesso agli elementi di una matrice

```
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.

```
#include <stdio.h>
#define RIG 2
#define COL 3
main()
{
int mat[RIG][COL];
int i, j;
/* lettura matrice */
printf("Lettura matrice %d x %d;\n", RIG, COL);
for (i = 0; i < RIG; i++)
  for (j = 0; j < COL; j++)
    scanf("%d", &mat[i][j]);
/* stampa matrice */
printf("La matrice e':\n");
for (i = 0; i < RIG; i++) {
   for (j = 0; j < COL; j++)
     printf("%6d ", mat[i][j]);
   printf("\n");      }   /* a capo dopo ogni riga */
}
```

Esempio: Programma che legge due matrici M×N (ad esempio 4×3) e calcola la matrice somma.

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    c[i][j] = a[i][j] + b[i][j];
```

Inizializzazione di matrici

```
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

Esercizio

Programma che legge una matrice A (M×P) ed una matrice B (P×N) e calcola la matrice C prodotto di A e B

- La matrice C è di dimensione M×N.
- Il generico elemento C_{ij} di C è dato da:

$$C_{ij} = \sum_{k=0}^{P-1} A_{ik} \cdot B_{kj}$$

Soluzione

```
#define M 3
#define P 4
#define N 2
int a[M][P], b[P][N], c[M][N];
...
/* calcolo prodotto */
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++) {
    c[i][j] = 0;
    for (k = 0; k < P; k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
  }
```

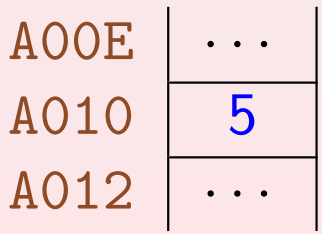
- Tutti gli elementi di `c` possono essere inizializzati a 0 al momento della dichiarazione:

```
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];
```

Cosa è una variabile?

Quando si dichiara una variabile, ad es. `int a;` si rende noto il nome e il tipo della variabile. Il compilatore

- alloca l'opportuno numero di byte di memoria per contenere il valore associato alla variabile (ad es. `4`).
- aggiunge il simbolo `a` alla tavola dei simboli e l'indirizzo del blocco di memoria ad esso associato (ad es. `A010` che è un indirizzo esadecimale)
- Se poi troviamo l'assegnamento `a = 5;` ci aspettiamo che al momento dell'esecuzione il valore `5` venga memorizzato nella locazione di memoria assegnata alla variabile `a`



Cosa è una variabile?

Alla variabile a si associa quindi:

- il valore della locazione di memoria, ovvero l'indirizzo `A010` e
- il valore dell'intero che vi viene memorizzato, ovvero `5`.

- Nell'espressione `a = 5;` con `a` ci riferiamo alla locazione di memoria associata alla variabile: il valore `5` viene copiato a quell'indirizzo.
- nell'espressione `b = a;` (dove `b` è ancora un intero) `a` si riferisce al valore: il valore associato ad `a` viene copiato all'indirizzo di `b`

  È ragionevole avere anche variabili che memorizzino indirizzi.

Puntatori

- Proprietà della variabile `a` nell'esempio:

  nome: `a`
  tipo: `int`
  valore: 5
  indirizzo: `A010` (che è fissato una volta per tutte)

- In C è possibile denotare e quindi manipolare gli indirizzi di memoria in cui sono memorizzate le variabili.
- Abbiamo già visto nella `scanf`, l'operatore indirizzo "`&`", che applicato ad una variabile, denota l'indirizzo della cella di memoria in cui è memorizzata (nell'es. `&a` ha valore `0xA010`).
- Gli indirizzi si utilizzano nelle variabili di tipo puntatore, dette semplicemente puntatori.

Tipo di dato: Puntatore

Un puntatore è una variabile che contiene l'indirizzo in memoria di un'altra variabile (del tipo dichiarato)

Esempio: dichiarazione **int** `*pi;`

- La varibile `pi` è di tipo puntatore a intero
- È una variabile come tutte le altre, con le seguenti proprietà:

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

- Più in generale:
  Sintassi `tipo *variabile;`
- Al solito, più variabili dello stesso tipo possono essere dichiarate sulla stessa linea

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

Esempio:

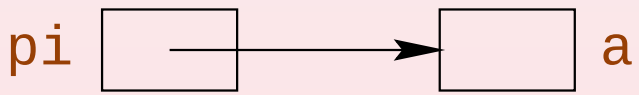
```
int *pi1, *pi2, i, *pi3, j;
float *pf1, f, *pf2;
```

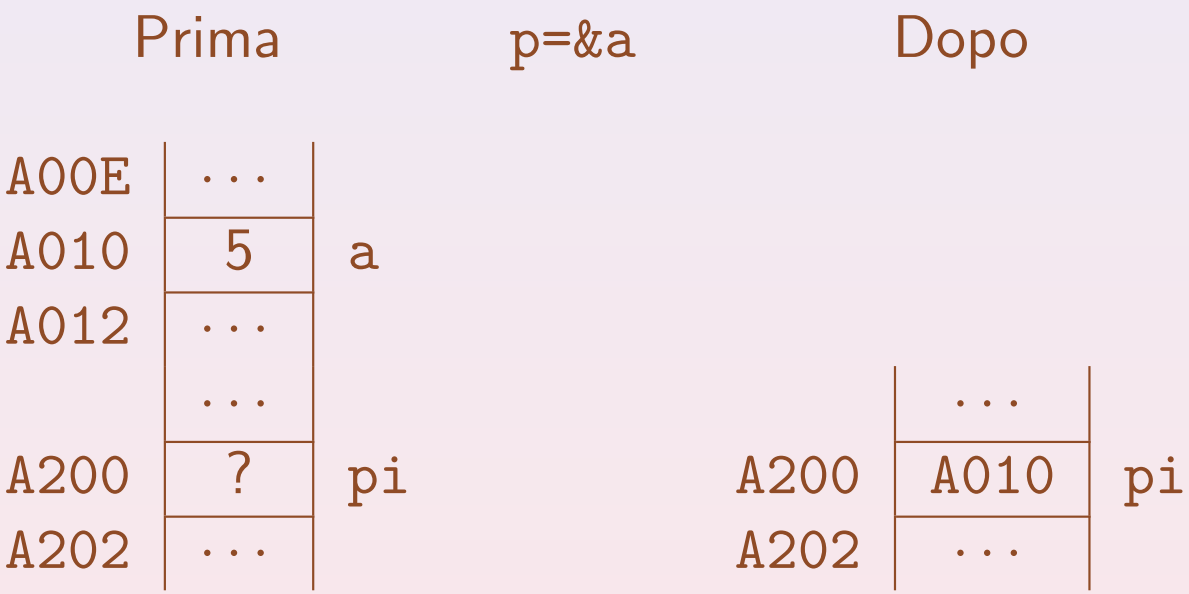
- Abbiamo dichiarato:

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

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

  Esempio: `pi = &a;`
  - il valore di `pi` viene inizializzato all'indirizzo della variabile `a`
  - si dice che `pi` punta ad `a` o che `a` è l'oggetto puntato da `pi`
  - lo rappresenteremo spesso cosi':




Prima
p=&a
Dopo
A00E
...
A010
5
a
A012
...
...
A200
?
pi
A202
...
...
A200
A010
pi
A202
...

Operatore di dereferenziamento "*"

- Applicato ad una variabile puntatore fa riferimento all'oggetto puntato. (mentre `&` fa riferimento all'indirizzo)
  Esempio:

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

pi = &a;   /* pi punta ad a ==> *pi sta per a */
b = *pi;  /* assegna a b il valore della var.puntata
             da pi, ovvero il valore di a:  5 */
*pi = 9; /* assegna 9 alla variabile puntata da pi,
            ovvero ad a */
```

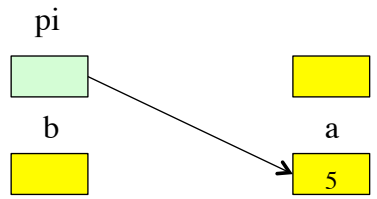
- N.B. Se `pi` è di tipo `int *`, allora `*pi` è di tipo `int`.
- Non confondere le due occorrenze di "*":
  - "*" in una dichiarazione serve per dichiarare una variabile di tipo puntatore, es. `int *pi;`
  - "*" in un'espressione è l'operatore di dereferenziamento, es. `b = *pi;`

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

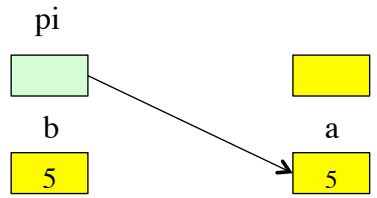
- hanno priorità più elevata degli operatori binari
- "*" è associativo a destra
  Es.: `**p` è equivalente a `*(*p)`
- "&" può essere applicato solo ad una variabile;
  `&a` non è una variabile $\Longrightarrow$ "&" non è associativo
- "*" e "&" sono uno l'inverso dell'altro
  - data la dichiarazione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

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

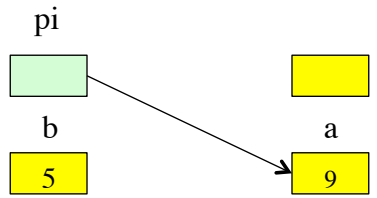
pi = &a



b= *pi



*pi = 9



Stampa di puntatori

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

Esempio:


```
int a = 5, *pi;
pi = &a;
printf("ind.  di a = %p\n", &a);   /* stampa 0xA010 */
printf("val.  di pi = %p\n", pi);   /* stampa 0xA010 */
printf("val.  di &*pi = %p\n", &*pi);   /* stampa 0xA010 */
printf("val.  di a = %d\n", a);   /* stampa 5 */
printf("val.  di *pi = %d\n", *pi);   /* stampa 5 */
printf("val.  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 */

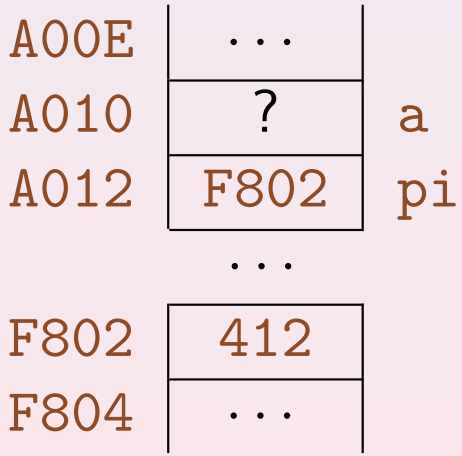
temp = *pa;
*pa = *pb;
*pb = temp;
```

Inizializzazione di variabili puntatore

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

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

  Esempio: `int a, *pi;`



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

  `*pi = 500;` $\Longrightarrow$ scrive 500 nella cella di indirizzo `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.

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

- Compilando si ottiene un warning:
  "assignment makes integer from pointer without a cast"
- Due variabili di tipo puntatore a tipi diversi sono incompatibili.

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

`x = pi;`	assegnazione `int*` a `int` warning: "assignment makes integer from pointer ..."
`pf = x;`	assegnazione `int` a `float*` warning: "assignment makes pointer from integer ..."
`pi = pf;`	assegnazione `float*` a `int*` warning: "assignment from incompatible pointer type"

- Perché il C distingue tra puntatori di tipo diverso?
- Se tutti i tipi puntatore fossero identici non sarebbe possibile determinare a tempo di compilazione il tipo di `*p`.

  Esempio:

  ```
  puntatore 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 (nota solo a tempo di esecuzione).
- Ad esempio, quale sarebbe il significato di `/` in `i/*p`: divisione intera o reale?

Funzione `sizeof` con puntatori

- La funzione `sizeof` restituisce l'occupazione in memoria in byte di una variabile (anche di tipo puntatore) o di un tipo.
- I puntatori occupano lo spazio di un indirizzo.
- L'oggetto puntato ha invece la dimensione del tipo puntato.

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

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

Operazioni con puntatori

Sui puntatori si possono effettuare diverse operazioni:

- di dereferenziamento

  Esempio:

  ```
  int *p, i;
  ...
  i = *p;
  ```

  Il valore della variabile intera `i` è ora lo stesso del valore dell'intero puntato da `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).

  Dopo l'assegnamento precedente, `p` punta allo stesso intero a cui punta `q`.

- di confronto

  Esempio:

  ```
  if (p == q) ...
  ```

  I due puntatori hanno lo stesso valore.

  Esempio:

  ```
  if (p > q) ...
  ```

  Ha senso? Con quello che abbiamo visto finora no. Vedremo che ci sono situazioni in cui ha senso.

Aritmetica dei puntatori

Sui puntatori si possono anche effettuare operazioni aritmetiche, con opportune limitazioni

- somma o sottrazione di un intero
- sottrazione di un puntatore da un altro

Somma e sottrazione di un intero

Se `p` è un puntatore a `tipo` e il suo valore è un certo indirizzo `ind`, il significato di `p+1` è il primo indirizzo utile dopo `ind` per l'accesso e la corretta memorizzazione di una variabile di tipo `tipo`.

Esempio:

```
int *p, *q;
....
q = p+1;
```

Se il valore di `p` è l'indirizzo `100`, il valore di `q` dopo l'assegnamento è `104` (assumendo che un intero occupi 4 byte).

- Il valore calcolato in corrispondenza di un'operazione del tipo `p+i` dipende dal tipo T di `p` (analog. per `p-i`):
  Op. Logica: p = p+1 Op.Algebrica: p = p +`sizeof(T)`

  Esempio:
  ```
  int *pi;
  *pi = 15;
  pi=pi+1;
  ```
  $\Longrightarrow$ `pi` punta al prossimo `int` (4 byte dopo)

  Esempio:
  ```
  double *pd;
  *pd = 12.2;
  pd = pd+3;
  ```
  $\Longrightarrow$ `pd` punta a 3 `double` dopo (24 byte dopo)

  Esempio:
  ```
  char *pc;
  *pc = 'A';
  pc = pc - 5;
  ```
  $\Longrightarrow$ `pc` punta a 5 `char` prima (5 byte prima)

- Possiamo anche scrivere: `pi++;` `pd+=3;` `pc-=5;`

Puntatore a puntatore

- Le variabili di tipo puntatore sono variabili come tutte le altre: in particolare hanno un indirizzo che può costituire il valore di un'altra variabile di tipo puntatore a puntatore.

Esempio:

```
int *pi, **ppi, x=10;
pi = &x;
ppi = &pi;
printf("pi = %p ppi = %p *ppi = %p\n", pi, ppi, *ppi);
printf("*pi = %d **ppi = %d x = %d\n", *pi, **ppi, x);
```

```
pi = 0x22ef34    ppi = 0x22ef3c    *ppi = 0x22ef34
*pi = 10         **ppi = 10         x = 10
```

Esempi

```
int a, b, *p, *q;
a=10;
b=20;
p = &a;
q = &b;
*q = a + b;
a = a + *q;
q = p;
*q = a + b;
printf("a=%d b=%d *p=%d *q=%d", a,b,*p,*q);
```

Quali sono i valori stampati dal programma?

Esempi (contd.)

```
int *p, **q;
int a=10, b=20;
q = &p;
p = &a;
*p = 50;
**q = 100;
*q = &b;
*p = 50;
a = a+b;
printf("a=%d   b=%d  *p=%d  **q=%d\n", a, b, *p, **q);
```

Quali sono i valori stampati dal programma?

Relazione tra vettori e puntatori

- 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 un puntatore, inizializzato all'inidirizzo dell'elemento di indice 0.

  `int vet[10];` $\Longrightarrow$ `vet` e `&vet[0]` hanno lo stesso valore (un indirizzo)

  $\Longrightarrow$ `printf("%p %p", vet, &vet[0]);` stampa 2 volte lo stesso indirizzo.

- Possiamo far puntare un puntatore al primo elemento di un vettore.

```
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 (il quarto elemento).
- 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]`
  - In C, `vet[3]` è solo un modo alternativo di scrivere `*(vet+3)`.
- Notazioni per gli elementi di un vettore:
  - `vet[3]` $\Longrightarrow$ notazione con puntatore e indice
  - `*(vet+3)` $\Longrightarrow$ notazione con puntatore e offset

- Un esempio che riassume i modi in cui si può accedere agli elementi di un vettore.

```
int vet[5] = {11, 22, 33, 44, 55};
int *pi = vet;
int offset = 3;

/* assegnamenti equivalenti */

vet[offset] = 88;
*(vet + offset) = 88;
pi[offset] = 88;
*(pi + offset) = 88;
```

- **Attenzione**: a differenza di un normale puntatore, il nome di un vettore è un puntatore costante
  - il suo valore non può essere modificato!
- ```
  int vet[10];
  int *pi;
  ```

  `pi = vet;` corretto

  `pi++;` corretto

  `vet++;` scorretto: `vet` e' un puntatore costante!
- È questo il vero motivo per cui non è possibile assegnare un vettore ad un altro utilizzando i loro nomi

  ```
  int a[3]={1,1,1}, b[3], i;
  for (i=0; i<3; i++)
      b[i] = a[i];
  ```

  ma non `b=a` (`b` è un puntatore costante!)
  ```

Modi alternativi per scandire un vettore

```
int a[LUNG]= {.......};
int i, *p=a;
```

- I seguenti sono tutti modi equivalenti per stampare i valori di a

```
for (i=0; i<LUNG; i++)
     printf("%d", a[i]);

for (i=0; i<LUNG; i++)
    printf("%d", *(a+i));

for (p=a; p<a+LUNG; p++)
    printf("%d", *p);
```

```
for (i=0; i<LUNG; i++)
      printf("%d", p[i]);

for (i=0; i<LUNG; i++)
     printf("%d", *(p+i));
```

- Non è invece lecito un ciclo del tipo

```
for ( ; a<p+LUNG; a++)
    printf("%d", *a);
```

perché? Perché a++ è un assegnamento sul puntatore costante a!.

Differenza tra puntatori

- Il parallelo tra vettori e puntatori ci consente di capire il senso di un'operazione del tipo `p-q` dove `p` e `q` sono puntatori allo stesso tipo.

```
int *p, *q;
int a[10]={0};
int x;
...
x=p-q;
```

- Il valore di `x` è il numero di interi compresi tra l'indirizzo `p` e l'indirizzo `q`.
- Quindi se nel codice precedente `...` sono le istruzioni:

```
q = a;
p = &a[5];
```

il valore di `x` dopo l'assegnamento è 5.

Esempio

```
double b[10] = {0.0};
double *fp, *fq;
char   *cp, *cq;

fp = b+5;
fq = b;

cp = (char *) (b+5);
cq = (char *) b;

printf("fp=%p  cp=%p  fq=%p  cq=%p\n", fp, cp, fq, cq);
printf("fp-fq= %d,   cp-cq=%d\n", fp-fq, cp-cq);
```

```
fp=0x22fe3c  cp=0x22fe3c  fq=0x22fe14  cq=0x22fe14
fp-fq=5   cp-cq=40
```

Modularizzazione

- Quando abbiamo a che fare con un problema complesso spesso lo suddividiamo in problemi più semplici che risolviamo separatamente, per poi combinare insieme le soluzioni dei sottoproblemi al fine di determinare la soluzione del problema di partenza.
- **Esempio:** La moltiplicazione di numeri con molte cifre viene suddivisa in moltiplicazioni cifra per cifra e i risultati di queste ultime vengono combinate insieme mediante addizioni.
- Questo procedimento è applicabile anche alla programmazione.
  - si suddivide un problema complesso in problemi di volta in volta più semplici
  - una volta individuati (sotto)problemi sufficientemente elementari si risolvono questi ultimi direttamente
  - si combinano le soluzioni dei sottoproblemi per ottenere la soluzione del problema di partenza

- Approccio top-down: si parte dall'alto, considerando il problema nella sua interezza e si procede verso il basso per raffinamenti successivi fino a ridurlo ad un insieme di sottoproblemi elementari
- Approccio bottom-up: ci si occupa prima di risolvere singole parti del problema, senza averne necessariamente una visione d'insieme, per poi risalire procedendo per aggiustamenti successivi fino ad ottenere la soluzione globale.
- I linguaggi di programmazione mettono a disposizione dei meccanismi di astrazione che favoriscono un approccio modulare

  Astrazione sui dati - il programmatore può definire nuovi tipi di dato specifici per il particolare problema (tipi di dato astratti)
  - collezioni di valori
  - operazioni con le quali operare su tali valori

  Astrazione funzionale - il programmatore può estendere le funzionalità del linguaggio definendo sottoprogrammi che risolvono (sotto)problemi specifici.
  - i sottoprogrammi sono di solito parametrici
  - possono essere (ri)usati alla stessa stregua delle operazioni built-in del linguaggio

Funzioni

- In C i sottoprogrammi si realizzano attraverso le funzioni.
- Una funzione può essere vista come una scatola nera:

$$\text{parametri di ingresso} \longrightarrow \boxed{\text{F}} \longrightarrow \text{valore calcolato}$$

  - risolve un sottoproblema specifico
  - attraverso i parametri e il risultato scambia informazioni con il `main` e con altre funzioni

Esempio:

$$x \longrightarrow \boxed{\text{abs}} \longrightarrow |x|$$

$$x, y \longrightarrow \boxed{\text{mcd}} \longrightarrow mcd(x, y)$$

$$b, e \longrightarrow \boxed{\text{exp}} \longrightarrow b^e$$

$$x_1, \ldots, x_n \longrightarrow \boxed{\text{sum}} \longrightarrow \sum_{i=1}^{n} x_i$$

Esempio: Definizione di abs in C

```
int abs(int x)
{
int ris;
if (x<0)
   ris = -x;
else
   ris = x;
return ris;  }
```

- Uso della funzione

```
main()
{
int x1, x2, z, w;
  ...
z = abs(x1);
  ...
printf("%d\n", w + abs(x2));
  ...
}
```

- Il linguaggio deve mettere a disposizione strumenti per
  - definire nuove operazioni astratte (funzioni)
  - usare le nuove operazioni definite
- Distinguiamo due momenti diversi:
  - la definizione della funzione
    definisce il codice che realizza l'operazione astratta
  - e la chiamata della funzione
    corrisponde all'utilizzo della funzione

- Ad una stessa definizione possono corrispondere diverse chiamate (come `z = abs(x1)` e `w + abs(x2)` nell'esempio precedente).
- Nella definizione della funzione, il codice fa riferimento agli argomenti o parametri formali della funzione (nell'esempio `x`)

  $\Longrightarrow$ un parametro formale non corrisponde ad un valore vero e proprio: è semplicemente un riferimento simbolico (ad un argomento della funzione)

Esempio:

```
int exp(int base, int esponente)
{
int ris = 1;
while (esponente > 0)
  {
    ris = ris * base;
    esponente = esponente - 1;
  }
return ris;
}
```

- I parametri formali sono base ed esponente

- Al momento della chiamata, alla funzione vengono forniti i valori degli argomenti, o parametri attuali, rispetto ai quali effettuare il calcolo

```
int exp(int base, int esponente)
{...}

main()  {
int b, e, r1, r2;
  ...
r1 = exp(2,5);
  ...
scanf("%d  %d", &b, &e);
r2 = exp(b, e);
  ... }
```

- Prima chiamata `exp(2,5)`
  - `2` è il parametro attuale corrispondente a `base`
  - `5` è il parametro attuale corrispondente a `esponente`
- Seconda chiamata `exp(b,e)`
  - `b` è il parametro attuale corrispondente a `base`
  - `e` è il parametro attuale corrispondente a `esponente`

Funzioni: definizione

Sintassi:

`intestazione blocco`

dove

- `blocco` è il corpo della funzione
- `intestazione` è l'intestazione della funzione ed ha la seguente forma:

  `id-tipo identificatore (parametri-formali)`

- `id-tipo` specifica il tipo del risultato calcolato dalla funzione
- `identificatore` specifica il nome della funzione ed è un qualsiasi identificatore C valido
- `parametri-formali` è una sequenza (eventualmente vuota) di dichiarazioni di parametro (tipo e nome) separate da virgola

Esempi: intestazioni di funzione

- `int abs (int x)`
  `int MassimoComunDivisore(int a, int b)`
- `double Potenza(double x, double y)`
  `float media (int vet[], int lung)`

Funzioni: chiamata (invocazione, attivazione)

Sintassi:

`identificatore` (`parametri-attuali`)

- `identificatore` è 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

Esempi: chiamate di funzioni

```
int mcd, x, y1, y2;
double exp, w, v, z;
  ...
mcd = MassimoComunDivisore(x+1, y1+y2);
exp = Potenza(z, 3.0);
  ...
exp = Potenza(z, Potenza(v,w));
```

Semantica (informale) di una chiamata di funzione

- Dentro il corpo di una funzione F compare una chiamata di un'altra funzione G
  - F viene detta funzione chiamante
  - G viene detta funzione chiamata

    Esempio: nel `main` c'è un assegnamento `x = abs(x);`

    $\Longrightarrow$ `main` è il chiamante, `abs` il chiamato
- Una chiamata di funzione è un'espressione, la cui valutazione avviene come segue:
  - viene sospesa l'esecuzione di F e viene "ceduto il controllo" a G, dopo aver opportunamente associato i parametri attuali ai parametri formali (passaggio dei parametri, fra poco . . .)
  - vengono eseguite le istruzioni di G, a partire dalla prima
  - l'esecuzione di G termina con l'esecuzione di un'istruzione speciale (istruzione `return`) che calcola il risultato della chiamata (è il valore dell'espressione corrispondente alla chiamata)
  - al termine dell'esecuzione di G il controllo ritorna a F, che prosegue l'esecuzione a partire dal punto in cui G era stata attivata

Valore di ritorno di una funzione: istruzione `return`

- **Esempio:** Funzione che restituisce il massimo tra due interi.

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

- Chiamata di `max`, ad esempio da `main`:

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

- La funzione `main` tramite i parametri attuali comunica alla funzione `max` i valori sui quali calcolare la funzione (il valore delle variabili `i`,`j`).
- La funzione `max` tramite il valore di ritorno comunica il risultato al `main`.

- Nel corpo deve esserci l'istruzione `return espressione;` la cui esecuzione comporta:
  - il calcolo del valore di `espressione`: questo valore viene restituito al chiamante come risultato dell'esecuzione della funzione
  - la cessione del controllo alla funzione chiamante

Osservazioni

- in `return espressione`, il tipo di `espressione` deve essere lo stesso del tipo del risultato della funzione dichiarato nella definizione
- l'esecuzione di `return espressione` comporta la terminazione dell'esecuzione della funzione

Esempio:

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

Dichiarazioni di funzione (o prototipi)

- I parametri attuali nella chiamata di una funzione devono corrispondere in numero e tipo (in ordine) ai parametri formali.
- Dobbiamo permettere al compilatore di fare questo controllo
  $\Longrightarrow$ prima della chiamata deve essere nota l'intestazione
- Due possibilità:
  1. la funzione è stata definita prima
  2. la funzione è stata dichiarata prima

Sintassi della dichiarazione di funzione (o prototipo)

```
intestazione;
```

ovvero:

```
id-tipo identificatore (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

- Bisogna dichiarare o definire ogni funzione prima di usarla (chiamarla)
- È pratica comune specificare in quest'ordine:
  1. dichiarazioni (prototipi) di tutte le funzioni (tranne `main`)
  2. definizione di `main`
  3. definizioni delle funzioni
- In questo modo ogni funzione è stata dichiarata prima di essere usata
- L'ordine in cui mettiamo le definizioni non deve necessariamente corrispondere a quello delle dichiarazioni.

Esempio:

```
int max(int, int);

int foo(char, int);

main()  ...

int max(int m, int n)  ...    /* OK. definizione coerente
 con il prototipo */

int foo (int z, char c)  ...    /* NO! definizione non coerente
 con il prototipo */
```

Nella definzione di `foo` i parametri formali non sono nell'ordine specificato dal prototipo.

Passaggio dei parametri

- Abbiamo visto che le funzioni utilizzano parametri
  - permettono uno scambio di dati tra chiamante e chiamato
  - nell'intestazione/prototipo: lista di parametri formali (con tipo associato) – sono delle variabili
  - nell'attivazione: lista di parametri attuali — possono essere delle espressioni
- Al momento della chiamata ogni parametro formale viene inizializzato al valore del corrispondente parametro attuale.
- Il valore del parametro attuale viene copiato nella locazione di memoria del corrispondente parametro formale.
- Questo meccanismo di passaggio dei parametri viene comunemente detto passaggio per valore.

Esempio:

Definizione della funzione succ

```
int succ (int);   /* prototipo di succ */

   main()                                  int succ (int x)
   {  int z, w;                               { x = x + 1;
      z = 10;                                   return x;
      w = succ(z);                           }
      printf("%d", w);
   }
```

- L'effetto della chiamata `succ(z)` può essere simulato dall'esecuzione della seguente porzione di codice:

```
x = 10;          /* il parametro formale viene inizializzato con
                    il valore del parametro attuale */
x = x + 1;       /* esecuzione del corpo della funzione
return x;           succ */
```

- Chiamate diverse corrispondono ad inizializzazioni diverse delle variabili corrispondenti ai parametri formali

```
w = succ(20);          x = 20;
                  ⟹    x = x + 1;
                       return x;
```

- In questo caso il valore assegnato alla variabile `w` è `21`.

```
z = 10;
w = succ(z+3);          x = 13;
                   ⟹    x = x + 1;
                        return x;
```

- In questo caso il valore assegnato alla variabile `w` è `14`.
  - Se non vi è corrispondenza perfetta tra il tipo del parametro formale e quello del parametro attuale, viene effettuata una conversione implicita di tipo secondo le regole già viste.
  - Il passaggio dei parametri di tipo array non comporta la copia dei valori dell'array (fra poco ...)

Procedure

- Non sempre le operazioni astratte di cui abbiamo bisogno possono essere descritte in modo naturale come funzioni matematiche.

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

  1. la forma della figura
  2. la dimensione
  3. il carattere di riempimento
  4. ...

- In questo caso il compito dell'operazione astratta non è (o non è soltanto) produrre un valore, ma è produrre effetti di altro tipo, tipicamente modifiche di stato.

  $\Longrightarrow$ in questi casi possiamo utilizzare procedure

  - le procedure sono un'astrazione delle istruzioni
  - le funzioni sono un'astrazione degli operatori

Le procedure in C

- Una procedura è una funzione avente come tipo del risultato il tipo speciale `void`.
- La definizione/dichiarazione di procedure e la loro chiamata è analoga al caso delle funzioni

  Esempio:

```
void emoticon (int n)
{  /* stampa n volte la sequenza -:) */
   int i;
   for (i=0; i<n; i++)
     {putchar('-');  putchar(':'); putchar(')'); putchar(' ');}
}

main()  {
   ...;
   emoticon(3);
   ...  }
```

- Le procedure non contengono di solito un'istruzione `return` (se la contengono è del tipo `return;` che non comporta il calcolo di alcun valore, ma solo la cessione del controllo al chiamante)

- La semantica di una chiamata di procedura P da una funzione/procedura F è analoga a quella della chiamata di funzione, ma una chiamata di procedura è un'istruzione
- In particolare, il passaggio dei parametri avviene per valore come nel caso delle funzioni
- il controllo viene restituito al chiamante al termine dell'esecuzione del blocco che costituisce il corpo della procedura (o in corrispondenza dell'esecuzione di un'istruzione del tipo `return;`)
- Il C non distingue tra funzioni e procedure (queste ultime sono casi particolari di funzioni)
  $\Longrightarrow$ concettualmente, però, è bene vedere le funzioni come astrazioni di operazioni e le procedure come astrazioni di istruzioni.

Esempio:

Procedura che stampa una cornice di asterischi di "altezza" parametrica

```
void stampaCornice(int altezza)
{
  int i;
  printf("**********************\n");
  for (i=1; i<=altezza; i++)
     printf("*                    *\n");
  printf("**********************\n");
  return;
}
```

- Come astrazione delle istruzioni, le procedure possono dover modificare lo stato.
  Esempio: Procedura `abs` che assegna ad una variabile intera il suo valore assoluto
  - il chiamante deve comunicare alla procedura la variabile in questione
  - la procedura deve analizzare il valore della variabile e, se necessario, effettuare il rimpiazzamento
- La seguente realizzazione della procedura non è corretta

```
void abs(int x)
{    if (x < 0)
        x = -x;   }
```

- Simuliamo il comportamento di una chiamata della procedura (come visto in precedenza)

```
int z = -5;
abs(z);        ⟹    x = -5;
                     if (x < 0) x = -x;
```

- La modifica del parametro formale non si ripercuote sul parametro attuale (si ricordi che il passaggio dei parametri è per valore).

Passaggio dei parametri per indirizzo

- Per ottenere l'effetto di modificare il valore dei parametri attuali, alcuni linguaggi (es. Pascal) prevedono un'ulteriore modalità di passaggio dei parametri, il passaggio per indirizzo
  - informalmente: il passaggio per indirizzo fa in modo che, al momento della chiamata, il parametro formale costituisca un modo alternativo per accedere al parametro attuale (che deve essere una variabile e non può essere una generica espressione)
  - durante l'esecuzione del corpo, ogni riferimento (e in particolare modifica) al parametro formale è di fatto un riferimento al parametro attuale.
- In C esiste solo il passaggio per valore, ma quello per indirizzo si può realizzare come segue:
  1. si utilizza un parametro formale di tipo puntatore
  2. all'interno del corpo della procedura ogni riferimento al parametro formale avviene attraverso l'operatore di dereferenziazione `*`
  3. al momento della chiamata, si utilizza come parametro attuale un indirizzo di una variabile (utilizzando, se necessario, l'operatore di indirizzo `&`).

Esempio:

Riprendiamo l'esempio del valore assoluto.

```
void abs(int *x)
{ if (*x < 0)
      *x = -(*x);
}
```

- Nel chiamante si utilizzano chiamate del tipo `abs(&z)` per ottenere l'effetto di rimpiazzare il valore della variabile `z` con il suo valore assoluto.

```
int z = -5;
abs(&z);        ⟹    x = &z;
                      if (*x < 0) *x = -(*x);
```

- N.B. Il passaggio dei parametri è sempre per valore, ma questa volta viene passato un valore puntatore che consente di accedere alla variabile del chiamante.
- Nell'esempio, l'assegnamento `*x = -(*x);` ha come effetto la modifica della variabile `z` del chiamante, in quanto il valore di `x` è `&z` e dunque `*x` è proprio `z`.

Esempio: Procedura per lo scambio di due variabili intere

Procedura per lo scambio di due variabili intere

```
void swap(int *x, int *y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

- Esempio di utilizzo: programma che legge due valori interi e li stampa ordinati.

programma che legge due valori interi e li stampa ordinati.

```
main()   {
int a, b;
scanf("%d", &a);
scanf("%d", &b);
if (a > b) swap(&a, &b);
printf("Valore minimo: %d\n", a);
printf("Valore massimo: %d\n", b);   }
```

Parametri di tipo vettore

- Il meccanismo del passaggio per valore di un indirizzo consente il passaggio di vettori come parametri di funzioni/procedure.
- 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 anche la dimensione del vettore in un ulteriore parametro.

Esempio:

```
void stampaVettore(int *v, int dim)
{ int i;
  for (i = 0; i < dim; i++)
    printf("v[%d]: %d\n", i, v[i]);
}

main()
{ int vet[5] = {1, 2, 3, 4, 5};
  ...
  stampaVettore(vet, 5);    ...  }
```

- Per evidenziare che il parametro formale è un vettore (ovvero l'indirizzo dell'elemento di indice 0), si può utilizzare la notazione `nome-parametro[]` invece di `*nome-parametro`.
  Esempio: `void stampa(int v[], int dim) { ... }`
- Si può anche specificare la dimensione nel parametro, ma questa viene ignorata.
  Esempio: `void stampa(int v[5], int dim) { ... }`
- Come al solito, nel prototipo della funzione il nome del parametro (vettore) può anche mancare.
  Esempio: `void stampa(int [], int);`

- Il passaggio di un vettore è un passaggio per indirizzo.

  $\Longrightarrow$ La funzione può modificare gli elementi del vettore passato.

  Esempio: Lettura di un vettore.

```
void leggiVettore(int v[], int dim)
{
  int i;
  for (i = 0; i < dim; i++)
  {
    printf("Immettere l'elemento di indice %d: ", i);
    scanf("%d", &v[i]);
  }
}
```

Esempio: Programma che legge, inverte e stampa un vettore di interi

Programma che legge, inverte e stampa un vettore di interi

```
#include <stdio.h>
#define LUNG 5

void leggiVettore(int [], int);
void stampaVettore(int [], int);
void invertiVettore(int [], int);

main()
{
  int vett[LUNG];
  leggiVettore(vett, LUNG);
  printf("Vettore prima dell'inversione\n");
  stampaVettore(vett, LUNG);

  invertiVettore(vett, LUNG);
  printf("Vettore dopo l'inversione\n");
  stampaVettore(vett, LUNG);
}
```

- La definizione della procedura `void invertiVettore(int [], int);` è lasciata per esercizio.

Passaggio di matrici come parametri

- Quando passiamo un vettore ad una funzione, passiamo in realtà il puntatore (costante) all'elemento di indice 0.

  $\Longrightarrow$ non serve specificare la dimensione del vettore nel parametro formale.
- Quando passiamo una matrice ad una funzione, per poter accedere correttamente agli elementi, 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) { ... }`
- Il motivo è semplice: per accedere ad un generico elemento della matrice, `mat[i][j]`, la funzione deve calcolare l'indirizzo di tale elemento `mat + offset`. Per calcolare correttamente `offset` è necessario sapere quante sono le colonne.
- L'indirizzo di `mat[i][j]` è infatti:

  `mat + (i · C · sizeof(int)) + (j · sizeof(int))`

  dove `C` è il numero di colonne (ovvero gli elementi in ciascuna riga).

Riassumendo:

- 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
- In generale, in un parametro di tipo array vanno specificate tutte le dimensioni, tranne eventualmente la prima.
  1. vettore: non serve specificare il numero di elementi
  2. matrice: bisogna specificare il numero di colonne, ma non serve il numero di righe

Esercizio

Definire le funzioni/procedure utilizzate nel seguente programma e completare con gli opportuni parametri attuali la chiamata di `swap` in modo che il suo effetto sia di scambiare gli elementi minimo e massimo del vettore.

```
#include <stdio.h>
#define LUNG 10

void leggivet (int [] vet, int dim);
void stampavet (int [] vet, int dim);
int indice_minimo (int vet[], int dim);
int indice_massimo (int vet[], int dim);
void swap (int *, int *);

main()
{
  int vettore[LUNG], pos_min, pos_max;

  leggivet(vettore, LUNG);
  pos_min = indice_minimo(vettore, LUNG);
  pos_max = indice_massimo(vettore, LUNG);
  swap (?, ?);   /* scambio degli elementi minimo e massimo */
  printf("Vettore dopo lo scambio dell'elemento minimo e massimo:\n");
  stampavet(vettore, LUNG);
}
```

Variabili locali

- Il blocco che costituisce il corpo di una funzione/procedura può contenere dichiarazioni di variabili.

  Esempio:

```
void leggiVettore(int v[], int dim)
{
  int i;              /* i E' UNA VARIABILE LOCALE */
  for (i = 0; i < dim; i++) { ... }
}
```

  - sono variabili proprie della funzione
  - hanno tempo di vita limitato alla durata della chiamata
  - più in generale: un identificatore dichiarato nel corpo di una funzione è detto locale alla funzione e non è visibile all'esterno della funzione (ad esempio nel `main`), ma solo nel corpo della stessa

- In realtà, ciò non è altro che un caso particolare di regole generali che governano la visibilità e il tempo di vita degli identificatori di un programma.

Struttura generale di un programma C

- parte direttiva
- parte dichiarativa globale che comprende:
  - dichiarazioni di costanti
  - dichiarazioni di tipi (li vedremo . . .)
  - dichiarazioni di variabili (variabili globali)
  - prototipi di funzioni/procedure
- il programma principale (`main`)
- le definizioni di funzioni/procedure

Esempio

```
#include <stdio.h>        /* parte direttiva */
#define LUNG 10

int i = 1;               /* variabili globali */
int j = 2;

int Q(int);              /* prototipi di funzioni e procedure */
void P(int *);

main()                  /* programma principale */
{
int x = 10;
char c = 'a';
x = Q(x);
P(&x);
}

int Q(int v) { ... }     /* definizioni di funzioni e procedure */
void P(int *z) { ... }
```

Blocchi

- il corpo di una funzione/procedura, cosi' come il corpo del programma principale, è un blocco.
- In C un blocco è costituito da
  - una parte dichiarativa (può non esserci)
  - una parte esecutiva (sequenza di istruzioni)
- Nel `main` o nel corpo delle funzioni possono comparire diversi blocchi, che possono essere
  - annidati: un blocco è una delle istruzioni di un altro blocco
  - paralleli: blocchi che fanno parte della medesima sequenza di istruzioni

```
{
   int x;
   x = 10;
   {
     int z;
     z = 20 ;
     ...
   }
   ...
}
```

```
{
   int x;
   x = 10;
   ...
}
{
   int z;
   z = 20;
   ...
}
```

- Anche la parte esecutiva del programma principale e di una funzione/procedura è un blocco
- Gli identificatori dichiarati nella parte dichiarativa di un blocco sono detti nomi locali del blocco e devono essere tutti diversi tra loro
  - nel caso di una funzione/procedura, fanno parte dei nomi locali anche gli identificatori utilizzati per i parametri formali

Esempio:

```
{
int x;   /* NO! identificatore x dichiarato   */
char x;  /* due volte nello stesso blocco     */
...
}

void p(int x, char y)
{
int x;      /* NO! identificatore x gia' usato per un parametro formale */
...
}
```

- In blocchi diversi possono essere utilizzati gli stessi identificatori

Esempio:

```
main()
{
int x;      /* x, y: variabili locali del main */
int y;
...
   {
     char x;     /* x: variabile locale del blocco annidato */
     ...
   }
...
}

void p(int x)
{
int y;          /*x,y: variabili locali della procedura p  */
...
}
```

- Un programma C può avere una struttura molto complessa a seguito dell'uso di funzioni, procedure e blocchi.
- È necessario definire regole precise per regolamentare l'uso dei nomi utilizzati all'interno di un programma.
- A questo scopo introduciamo alcune definizioni utili.

  Ambiente globale: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa globale del programma

  Ambiente locale di una funzione: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa della funzione e nella sua intestazione

  Ambiente locale di un blocco: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa del blocco
- Quanto detto informalmente in precedenza può essere meglio precisato:

  $\Longrightarrow$ è possibile dichiarare più volte lo stesso identificatore (anche con significati diversi) purché in ambienti diversi
- Se ciò evita il proliferare di identificatori, causa il problema di stabilire il significato di un riferimento ad un identificatore in un generico punto del programma

Esempio: Riprendiamo l'esempio precedente

```
main()
{
int x;      /* x, y: variabili locali del main */
int y;
...
   {
     char x;     /* x: variabile locale del blocco annidato */
     ...
   }
...
}

void p(int x)
{
int y;          /*x,y: variabili locali della procedura p  */
...
}
```

- Se in un punto del programma viene eseguita l'istruzione `x = ...`, a quale delle tre dichiarazioni di `x` ci si riferisce?
- Dipende dal punto in cui si trova tale assegnamento e dalle regole di visibilità (o regole di scoping).

Regole di visibilità

- Gli identificatori presenti nell'ambiente globale sono visibili in tutte le funzioni e in tutti i blocchi del programma.
  Se un identificatore è definito in più punti (in blocchi e/o funzioni), la definizione valida è quella dell'ambiente più vicino al punto di utlizzo.
  N.B. Gli identificatori predefiniti del linguaggio si intendono parte dell'ambiente globale.
- Gli identificatori presenti nell'ambiente locale di una funzione sono visibili nel corpo della funzione (ivi compresi eventuali blocchi in esso contenuti).
  Se un identificatore è definito in più punti del corpo, la definizione valida è quella dell'ambiente più vicino al punto di utlizzo.
- Gli identificatori presenti nell'ambiente locale di un blocco sono visibili nella parte esecutiva del blocco (ivi compresi eventuali blocchi in essa contenuti).
  Se un identificatore è definito in più punti di un blocco, la definizione valida è quella dell'ambiente più vicino al punto di utlizzo.

- Detto altrimenti, l'ambito di visibilità di un identificatore è determinato dalla posizione della sua dichiarazione:
  - gli identificatori dichiarati all'interno di un blocco hanno ambito di visibilità a livello di blocco
    $\Longrightarrow$ una variabile dichiarata in un blocco è visibile solo in quel blocco (compresi eventuali blocchi annidati)
  - gli identificatori dichiarati all'interno di una funzione (compresi quelli nell'intestazione) hanno ambito di visibilità a livello di funzione
    $\Longrightarrow$ una variabile dichiarata in una funzione è visibile solo nel corpo della funzione (compresi eventuali blocchi annidati)
  - gli identificatori dichiarati all'esterno delle funzioni e del main hanno ambito di visibilità a livello di programma
    $\Longrightarrow$ una variabile globale è visibile ovunque nel programma

Esempio:

```
int x1=10, x2=20;
char c='a';

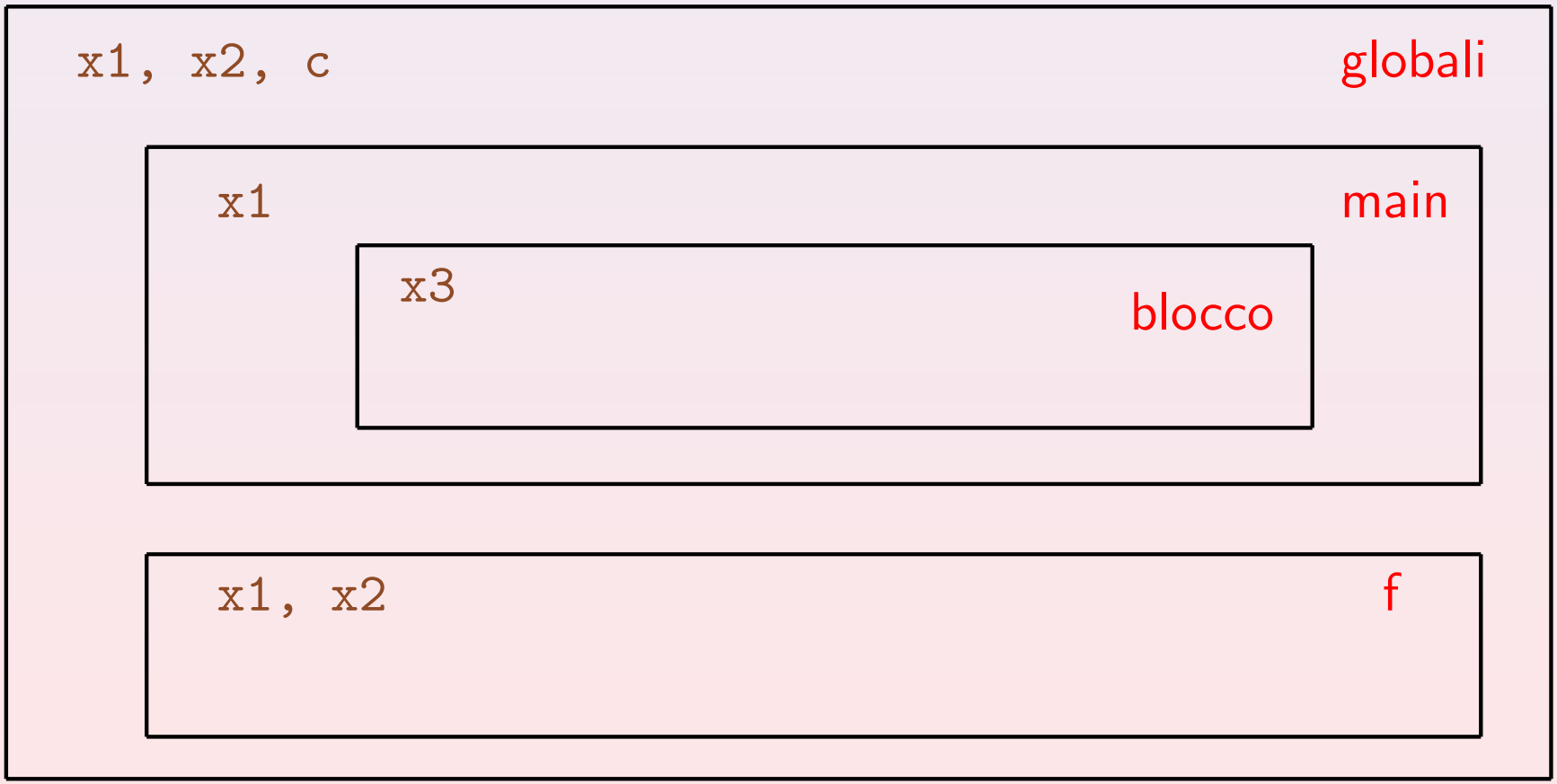
int f(int);

main()
{
int x1=30;    /* nasconde la variabile globale x1 */
x2 = x1+x2;   /* x1 e' quella locale, x2 e' globale */
printf("x1=%d   x2=%d\n", x1, x2);   /* stampa x1=30  x2=50  */
  { int x3=50;
    x1=f(x3); /* x1 e' quella locale al primo blocco */
    printf("x1=%d   x2=%d\n", x1, x2);   /* stampa x1=150  x2=50 */
  }
}

int f(int x1)  /* nasconde la variabile globale x1 */
{ int x2;       /* nasconde la variabile globale x2 */
  x2 = x1 + 100;  /* x1 e' il parametro formale, x2 la var. locale */
  return x2;
}
```

Rappresentazione Grafica: Modello a contorni

- Si rappresenta ogni ambiente mediante un rettangolo con gli identificatori in esso contenuti.



Durata delle variabili

- Una variabile ha un suo tempo di vita.
  - viene creata (ovvero ad essa viene riservata uno spazio di memoria)
  - viene (o può essere) distrutta (ovvero viene rilasciato il corrispondente spazio di memoria).
- Si distinguono due classi di variabili:
  - variabili automatiche: vengono create ogni volta che si entra nel loro ambiente di visibilità e vengono distrutte all'uscita di tale ambiente
    - es. variabili locali di un blocco: vengono create all'ingresso del blocco { distrutte all'uscita dal blocco }
    - es. variabili locali di una funzione: vengono create al momento della chiamata e distrutte all'uscita
  - variabili statiche: vengono create una sola volta e vengono distrutte solo al termine dell'esecuzione del programma (non ne faremo uso ...)
- N.B. nel caso di funzioni/blocchi eseguiti più volte (es. funzione chiamata in punti diversi, blocco all'interno di un ciclo):

  le variabili automatiche corrispondenti possono essere associate di volta in volta a locazioni di memoria diverse, quindi
  il loro valore non persiste tra una esecuzione e la successiva

Gestione della memoria a tempo di esecuzione (run-time)

- Il codice macchina e i dati risiedono entrambi in memoria, ma in zone separate:
  - la memoria per il codice macchina è fissata a tempo di compilazione
  - la memoria per i dati (in particolare per le variabili automatiche) cresce e decresce dinamicamente durante l'esecuzione: viene gestita a pila
- Una pila (o stack) è una struttura dati con accesso LIFO: Last In First Out = l'ultimo entrato è il primo ad uscire (es.: pila di piatti da lavare).
- Il sistema gestisce in memoria la pila dei record di attivazione (RDA)
  - per ogni chiamata di funzione viene creato un nuovo RDA in cima alla pila
  - al termine della chiamata 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)
  - altre informazioni che non analizziamo
- Anche gli ambienti locali dei blocchi vengono allocati/deallocati sulla pila.

Esempio:

```
int f(int);
main()
{
    int x, y, z;
    x=10;
    y=20;              /* PUNTO 1: blocco principale */
    z = f(x);          /* PUNTO 2: prima chiamata di f */
       {               /* PUNTO 3: uscita da f e ingresso nel blocco annidato*/
         int x=50;
         y=f(x);       /* PUNTO 4: seconda chiamata di f */
         z=y;          /* PUNTO 5: uscita da f */
       }
    ...                /* PUNTO 6: uscita dal blocco  */
}
int f(int a)
{
   int z;
   z = a + 1;
   return z;
}
```

Evoluzione della pila

PUNTO 1

x	10
y	20
z	?

PUNTO 2

a	10
z	?

x	10
y	20
z	?

PUNTO 3

x	50

x	10
y	20
z	11

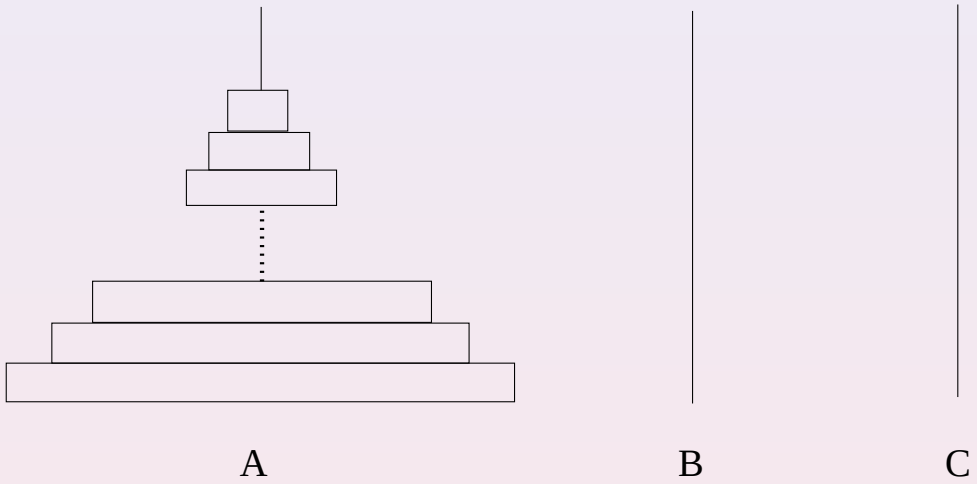
PUNTO 4

Programmazione ricorsiva: cenni

- In quasi tutti i linguaggi di programmazione evoluti è ammessa la possibilità di definire funzioni/procedure ricorsive: durante l'esecuzione di una funzione F è possibile chiamare la funzione F stessa.
- Ciò può avvenire
  - direttamente: il corpo di F contiene una chiamata a F stessa.
  - indirettamente: F contiene una chiamata a G che a sua volta contiene una chiamata a F.
- Questo può sembrare strano: se pensiamo che una funzione è destinata a risolvere un sottoproblema $\mathcal{P}$, una definizione ricorsiva sembra indicare che per risolvere $\mathcal{P}$ dobbiamo . . . saper risolvere $\mathcal{P}$!

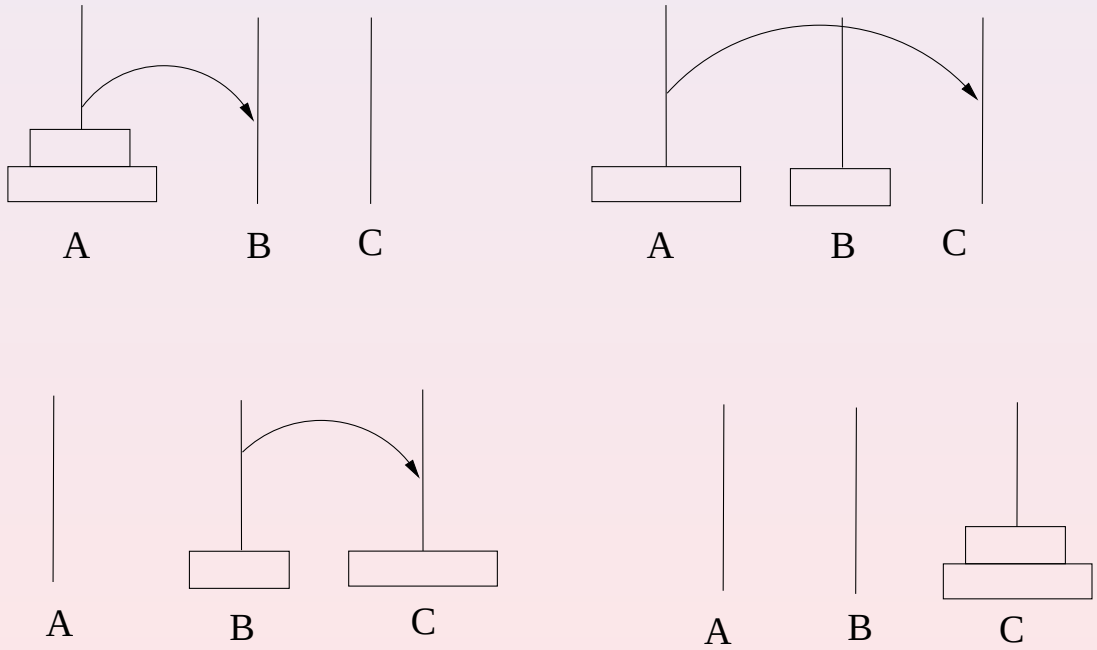
- In realtà, la programmazione ricorsiva si basa sull'osservazione che per molti problemi la soluzione per un caso generico può essere ricavata sulla base della soluzione di un altro caso, generalmente più semplice, dello stesso problema.
- La programmazione ricorsiva trova radici teoriche nel principio di induzione ben fondata che può essere visto come una generalizzazione del principio di induzione sui naturali
- La soluzione di un problema viene individuata supponendo di saperlo risolvere su casi più semplici.
- Bisogna poi essere in grado di risolvere direttamente il problema sui casi più semplici di qualunque altro.

Esempio: Torre di Hanoi (leggenda Vietnamita).



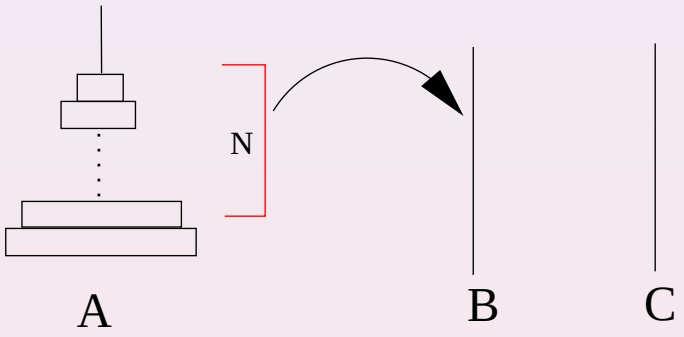
- pila di dischi di dimensione decrescente su un perno A
- vogliamo spostarla sul perno C, usando un perno di appoggio B
- vincoli:
  - possiamo spostare un solo disco alla volta
  - un disco più grande non può mai stare su un disco più piccolo
- secondo la leggenda: i monaci stanno spostando 64 dischi: quando avranno finito, ci sarà la fine del mondo

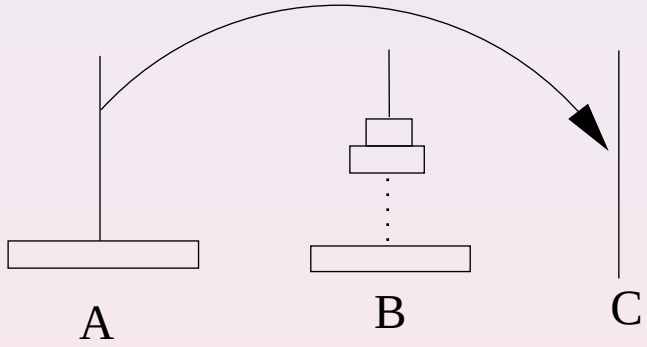
- Come individuare una soluzione per un numero N di dischi arbitrario?
  - per N=1 la soluzione è immediata: spostiamo l'unico disco da A a C
  - se sappiamo risolvere il problema per N=1 lo sappiamo risolvere anche per N=2: come?

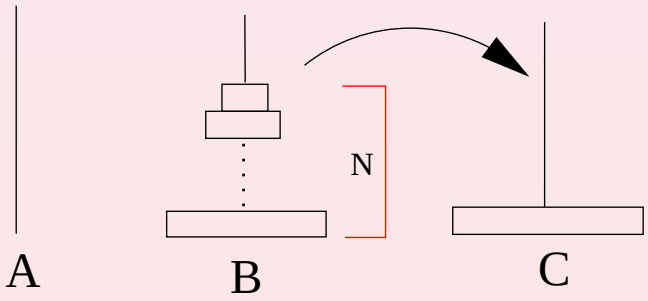


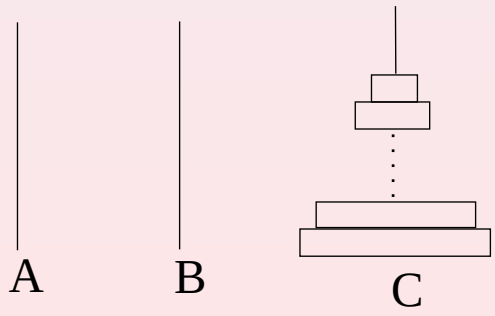
- Notiamo l'utilizzo del perno ausiliario B

- Possiamo generalizzare il ragionamento? Se sappiamo risolvere il problema per N dischi, possiamo individuare una soluzione per lo stesso problema ma con N+1 dischi?









- Formalizziamo il ragionamento
- Indichiamo con hanoi(N, P1, P2, P3) il problema: "spostare N dischi dal perno P1 al perno P2 utilizzando P3 come perno d'appoggio".

```
hanoi(N, P1, P2, P3)
     if (N=1)
         sposta  da P1 a P2;
     else
       {
         hanoi(N-1, P1, P3, P2);
         sposta da P1 a P2;
         hanoi(N-1, P3, P2, P1);
       }
```

Esempio: Soluzione di hanoi(3,A,C,B)

$$
\begin{aligned}
&\text{hanoi}(3,A,C,B) = \\
&\quad \text{hanoi}(2,A,B,C) = \\
&\qquad \text{hanoi}(1,A,C,B) = \text{sposta}(A,C) \\
&\qquad \text{sposta}(A,B) \\
&\qquad \text{hanoi}(1,C,B,A) = \text{sposta}(C,B) \\
&\quad \text{sposta}(A,\ C) \\
&\qquad \text{hanoi}(1,B,A,C) = \text{sposta}(B,A) \\
&\quad \text{hanoi}(2,B,C,A) = \\
&\qquad \text{sposta}(B,C) \\
&\qquad \text{hanoi}(1,A,C,B) = \text{sposta}(A,C)
\end{aligned}
$$

- Le funzioni ricorsive sono convenienti per implementare funzioni matematiche definite in modo induttivo.
  Esempio: Definizione induttiva di somma tra due interi non negativi:

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

  - La somma di x con 0 viene definita in modo immediato;
  - la somma di x con il successore di y viene definita come il successore della somma tra x e y.
- **Esempio:** somma di 3 e 2:

$$\begin{aligned} somma(3,2) = \quad & 1 + (\underline{somma(3,1)}) = \\ & 1 + (1 + (\underline{somma(3,0)})) = \\ & 1 + (1 + (3)) = \\ & 1 + 4 = \\ & 5 \end{aligned}$$

Esempio: Funzione fattoriale.

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

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

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

$$\begin{aligned} fatt(3) = \quad & 3 \cdot \underline{fatt(2)} = \\ & 3 \cdot (2 \cdot \underline{fatt(1)}) = \\ & 3 \cdot (2 \cdot (1 \cdot \underline{fatt(0)})) = \\ & 3 \cdot (2 \cdot (1 \cdot 1)) = \\ & 3 \cdot (2 \cdot 1) = \\ & 3 \cdot 2 = \\ & 6 \end{aligned}$$

Il codice delle due diverse versioni

- definizione iterativa:

```
int fatt(int n) {
    int i,ris;

    ris=1;
    for (i=1;i<=n;i++)
      ris=ris*i;
    return ris;
}
```

- definizione ricorsiva:

```
int fattric(int n) {
  if (n == 0)
    return 1;
  else
    return n * fattric(n-1);
 }
```

Esempio: Programma che usa una funzione ricorsiva.

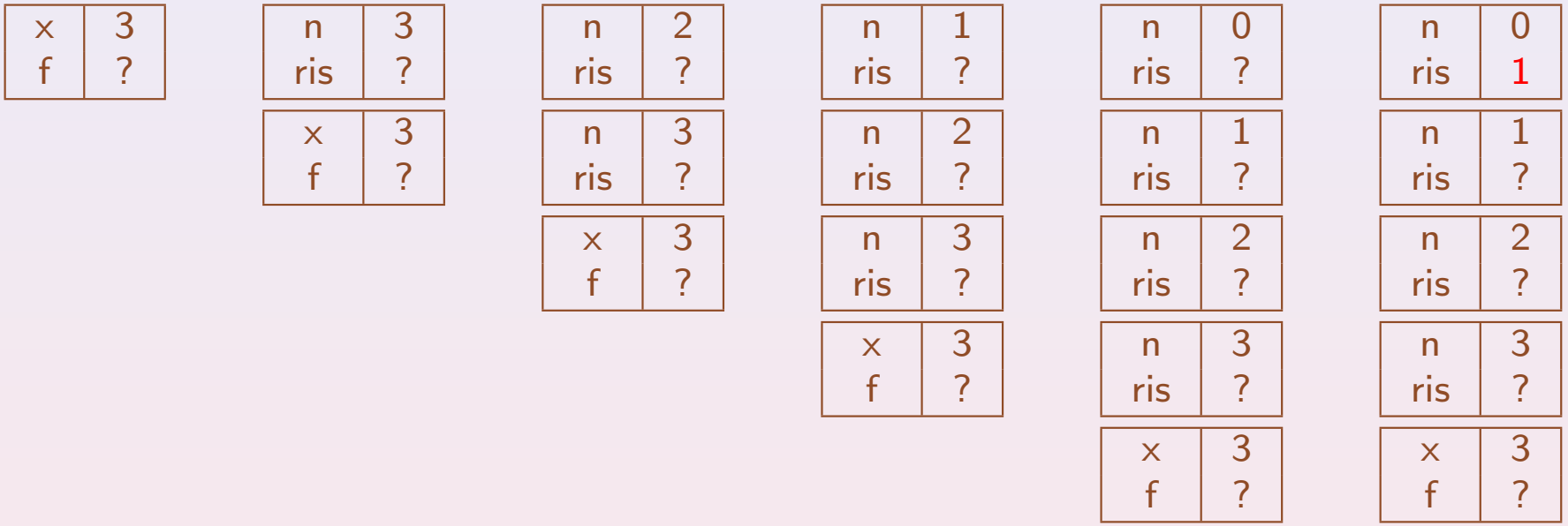
```
#include <stdio.h>

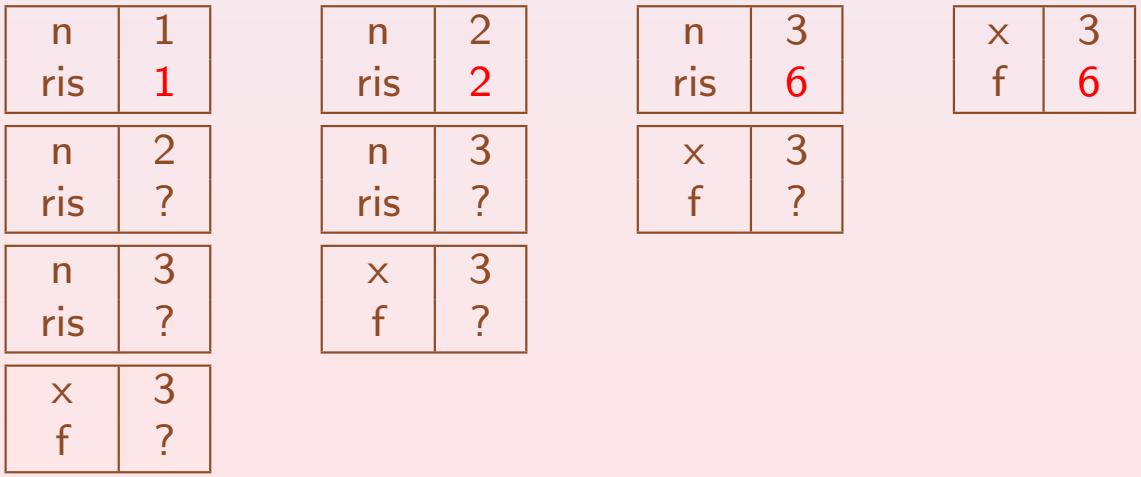
int fattric (int);

main()
{
int x, f;
scanf("%d", &x);
f = fattric(x);
printf("Fattoriale di %d:  %d\n", x, f);
}

int fattric(int n) {
  int ris;
  if (n == 0)
    ris = 1;
  else
    ris = n * fattric(n-1);
  return ris;
}
```

Evoluzione della pila (supponendo x=3).





Esempio: Leggere una sequenza di caratteri terminata da `'\n'` e stamparla invertita. Ad esempio: `casa` $\Longrightarrow$ `asac`

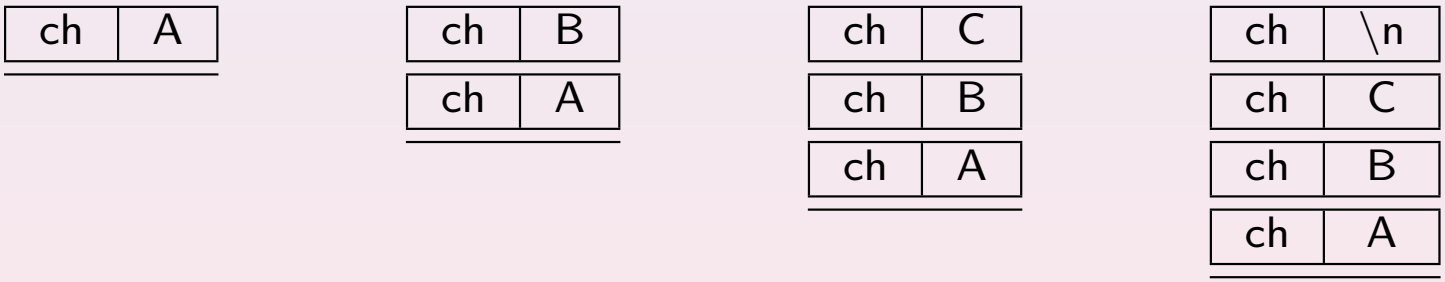
- Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  1. usando una struttura dati opportuna ma dinamica (liste, le vedremo più avanti)
  2. usando un procedimento ricorsivo.
     - leggiamo un carattere della sequenza, `c1`, leggiamo e stampiamo ricorsivamente il resto della sequenza `c2...cn` e infine stampiamo `c1`;
     - il caso base è rappresentato dalla lettura del carattere di fine sequenza.

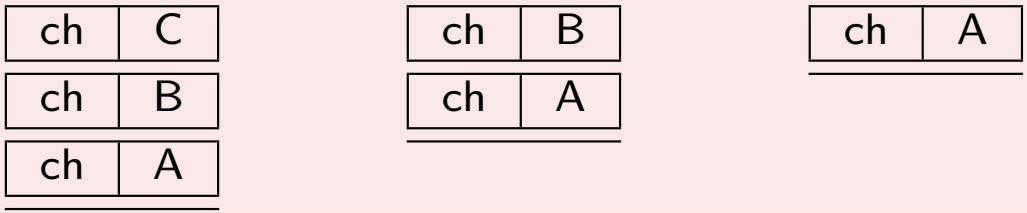
```
void invertInputRic()
 {  char ch;

  ch = getchar();
  if (ch != '\n')
    {
    invertInputRic();
    putchar(ch);
    }
  else
    printf("Sequenza invertita: ");
}
```

```
 main()
{
  printf("Immetti una sequenza di caratteri\n");
  invertInputRic();
  printf("\n");
}
```

Vediamo come evolve la pila per l'input `ABC\n`





L'output prodotto è il seguente

```
Sequenza invertita:  CBA
```

Ricorsione multipla

- Si ha ricorsione multipla quando un'attivazione di una funzione può causare più di una attivazione ricorsiva della stessa funzione (es. torre di Hanoi)

  Esempio: Definizione induttiva dei numeri di Fibonacci.

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-2) + F(n-1) \qquad \text{se } n > 1 \end{aligned}$$

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

```
 #include <stdio.h>

int fibonacci (int);

main() {
  int n;

  printf("Inserire un intero >= 0: ");
  scanf("%d", &n);
  printf("Numero %d di Fibonacci: %d\n", n, fibonacci(n));
}

 int fibonacci(int i)
  {
  int ris;
  if (i == 0)
    ris = 0;
  else if (i == 1)
    ris = ;
  else
    ris =  fibonacci(i-1) + fibonacci(i-2);
  return ris;
  }
```

Esempi di funzioni ricorsive

- Tradurre in C la definizione induttiva già vista:

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

```
int somma (int x, int y)
{
  int ris;
  if (y==0)
     ris = x;
  else
     ris = 1 + somma(x, y-1);
  return ris;
}
```

- Calcolo ricorsivo di x^y (si assume $y \geq 0$)

$$x^y = \begin{cases} 1 & \text{se } y = 0 \\ x \cdot x^{y-1} & \textit{altrimenti} \end{cases}$$

```
int exp (int x, int y)
{
  int ris;
  if (y==0)
     ris = 1;
  else
     ris = x * exp(x, y-1);
  return ris;
}
```

- Calcolare ricorsivamente la somma degli elementi nella porzione di un array v compresa tra gli indici from e to.
- Esprimiamo formalmente quanto richiesto:

$$sumVet(v, from, to) = \sum_{i=from}^{to} v[i]$$

- È evidente che:

$$\sum_{i=from}^{to} v[i] = \begin{cases} 0 & \text{se } from > to \\ v[from] + \sum_{i=from+1}^{to} v[i] & \text{se } from \leq to \end{cases}$$

- La traduzione in C è immediata.

Una soluzione

```
int sumVet(int *v, int from, int to)
{
  if (from > to)
    return 0;
  else
    return v[from] + sumvet(v,from+1,to);
}
```

Un'altra soluzione

```
int sumVet(int *v, int from, int to)
{
  int somma;
  if (from > to)
    somma = 0;
  else
    somma = v[from] + sumvet(v,from+1,to);
  return somma;
}
```

- Calcolare ricorsivamente il numero di occorrenze dell'elemento x nella porzione di un array v compresa tra gli indici from e to.

$$f(v, x, from, to) = \#\{i \in [from, to] \mid v[i] = x\}$$

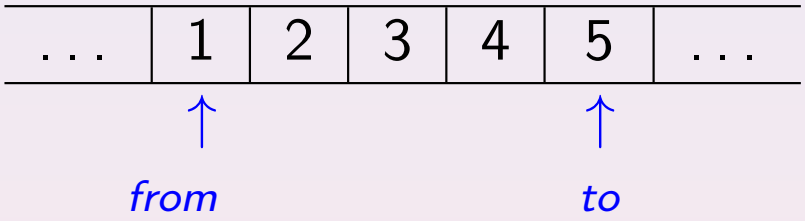
- Anche in questo caso ragioniamo induttivamente:

$$f(v, x, from, to) = \begin{cases} 0 & \text{se } from > to \\ f(v, x, from + 1, to) & \text{se } from \leq to \wedge v[from] \neq x \\ 1 + f(v, x, from + 1, to) & \text{se } from \leq to \wedge v[from] = x \end{cases}$$

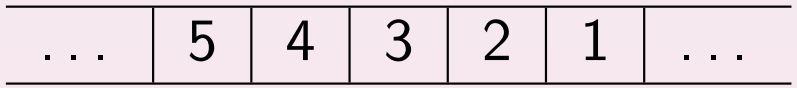
```
int occorrenze (int *v, int x, int from, int to)
{
  int occ;

  if (from > to)
    occ= 0;
  else
    if (v[from]!=x)
        occ = occorrenze(v,x,from+1,to);
    else
        occ = 1+occorrenze(v,x,from+1,to);
}
```

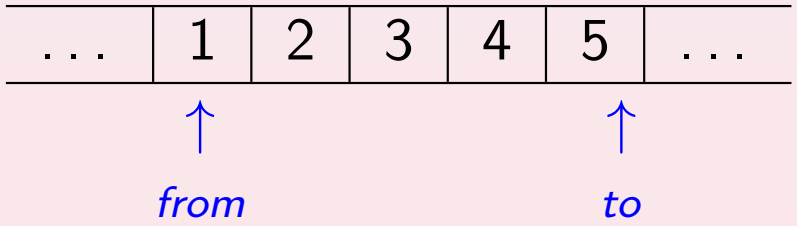
- Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici from e to.



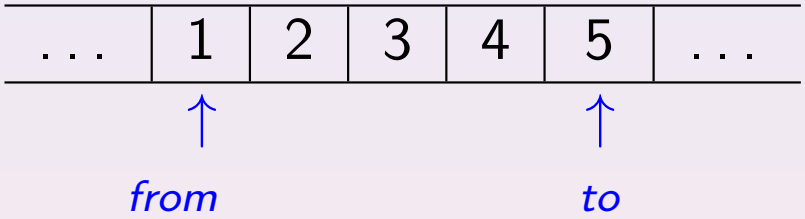
- Vogliamo ottenere:



- Induttivamente:

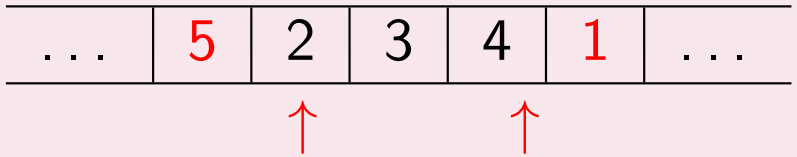


- Scrivere una procedura ricorsiva che inverte la porzione di un array individuata dagli indici from e to.



- Vogliamo ottenere:

- Induttivamente:



- Questa situazione corrisponde alla chiamata ricorsiva su una porzione più piccola del vettore

Programma

```
void swap(int *v, int i, int j)
{
    int temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

void invertiric (int *v, int from, int to)
{
  if (from < to)
    {
        swap(v, from, to);
        invertiric(v, from+1, to-1);
    }
}
```

- Si noti che la procedura non fa niente se la porzione individuata dal secondo e terzo parametro è vuota (from>to) o contiene un solo elemento (from=to)

Ricerca in una sequenza ordinata

Se la sequenza è ordinata posso sfruttare l'ordinamento per rendere più efficiente la ricerca, terminando se l'elemento corrente risulta maggiore dell'elemento cercato:

- Se, ad esempio, la sequenza è ordinata in senso crescente

```
int ricerca(int v[], int dim, int x, int *p)
{  int i;
    for =i;0; i<dim; i++)
       if (v[i] == x) {*p=i; return true; }
      else if (v[i]>x) return false;
      return false;
}
```

- Quanti elementi della sequenza devo esaminare per trovare l'elemento?
- Nel caso peggiore?
- Definire una funzione ricorsiva equivalente.

Ricerca in una sequenza ordinata con funzione ricorsiva

La versione ricorsiva della ricerca sequenziale :

- L'intestazione della funzione cambia

```
int ricercaRic(int v[], int from,int to, int x, int *p)
{  int i;
    if (from>to) return false;
    else if (v[i] == x) {*p=i; return true;}
    else if (v[i]>x) return false;
    else return ricercaRic(v,from+1, to, x,p);
}
```

- Una soluzione.

```
int ricercaOrd(int v[], int dim, int x, int *p)
{ return ricercaRic(v,0,dim,x, p);  }
```

Ricerca logaritmica in una sequenza ordinata

Un modo più efficiente è quello di mimare il nostro comportamento quando cerchiamo ad es. un numero telefonico in un elenco, in cui i numeri sono associati al nome e memorizzati seguendo l'ordine alfabetico:

- Non ci sogneremmo mai di partire dal primo e confrontare tutti i nomi finchè non ne troviamo uno che risulta uguale.
- Quello che facciamo è:
  - apriamo l'elenco ad un pagina più o meno casuale,
  - se l'elemento può essere in quella pagina, confrontiamo nomi con quello cercato
  - se l'elemento cercato si trova prima nell'ordinamento ripetiamo il procedimento nella parte dell'elenco che precede la pagina corrente.
  - se l'elemento cercato si trova dopo nell'ordinamento ripetiamo il procedimento nella parte dell'elenco che segue la pagina corrente.

Ricerca logaritmica in una sequenza ordinata

Adattiamo questo procedimento alla ricerca di un elemento in un vettore:

- calcoliamo l'elemento mediano e lo confrontiamo con il valore cercato
- se l'elemento mediano è uguale al valore cercato, terminiamo con true.
- se l'elemento mediano è maggiore dell'elemento corrente ripetiamo il procedimento nella porzione di vettore che precede l'elemento mediano.
- se l'elemento mediano è minore dell'elemento corrente ripetiamo il procedimento nella porzione di vettore che segue l'elemento mediano.
- Questo procedimento porta alla definizione di una funzione ricorsiva ma...
- quali sono le condizioni di terminazione?

Soluzione per la ricerca logaritmica

```
int ricercaLog(int v[], int x, int from, int to, int *p)

{ if (from>to) return false; // la porzione \'e vuota elemento non
  else {int med=(from+to)/2;  //calcolo l'indice del mediano
          if (v[med] == x)     //confronto il mediano con x
              {*p=i;
                return true;    // termina con successo
                }
          else if (v[med]>x)
           return ricercaLog(v, x, from, med, p);
          else return ricercaLog(v,x,med+1, to,p);
     }
```

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.
- Nel nostro caso formuliamo il problema in termini di ordinamento di vettori:

  Dato un vettore A di n elementi, ordinarlo in modo crescente
- Per semplicità faremo sempre riferimento a vettori di interi.

$$\begin{array}{ccc} & 5\ 2\ 4\ 6\ 1\ 3 \\ \Longrightarrow & \\ & 1\ 2\ 3\ 4\ 5\ 6 \end{array}$$

Ordinamento per inserzione (insertion sort)

- **Esempio:** Ordinamento di una mano di ramino
  - si inizia con la mano sinistra vuota e le carte coperte sul tavolo
  - si prende dalla tavola una carta alla volta e la si inserisce nella corretta posizione nella mano sinistra
  - ...
  - si termina quando si sono finite tutte le carte sul tavolo.
- Stesso procedimento per ordinare un vettore:
  - inizialmente il vettore rappresenta il mazzo sul tavolo
  - si usa un ciclo per analizzare uno alla volta gli elementi del vettore
  - Alla generica iterazione la situazione è la seguente

mano sinistra	carte ancora da scoprire

↑ nuova carta

  - per inserire la nuova carta al posto giusto nella mano sinistra dobbiamo
    - scorrere gli elementi che lo precedono per decidere la posizione che gli compete
    - spostare di un posto verso destra gli elementi maggiori per fargli spazio.

Esempio

In verde le carte ancora da esaminare, in rosso quelle già esaminate (mano sinistra). La nuova carta da esaminare è sottolineata.

$\underline{5}$	2	4	6	1	3
5	$\underline{2}$	4	6	1	3
2	5	$\underline{4}$	6	1	3
2	4	5	$\underline{6}$	1	3
2	4	5	6	$\underline{1}$	3
1	2	4	5	6	$\underline{3}$
1	2	3	4	5	6

- Una volta individuata la posizione k in cui inserire la nuova carta dobbiamo farle spazio, ovvero spostare verso destra di una posizione tutte le carte rosse da k in poi.

  Esempio:

1	2	4	5	6	3
1	2	4	5	6	3
1	2	4	5		6
1	2	4		5	6
1	2		4	5	6

- A questo punto possiamo piazzare la carta in modo ordinato

1	2	3	4	5	6

- Definiamo allora una procedura che sposta tutti gli elementi di un vettore verso destra di una posizione tra due indici dati `from` e `to`

```
void shiftR(int v[], int from, int to)
{
int i;
for (i=to-1; i>=from; i--)
  v[i+1] = v[i];
}
```

- L'elemento in posizione `to` viene perso
- Bisogna procedere da destra verso sinistra (perché?)
- se `to` è minore o uguale a `from` non succede nulla

Possiamo allora definire la procedura di di ordinamento per inserzione come segue

```
void sort(int v[], int dim)
{
int h, curr, j=1;
while (j<dim)
  { h=0;
    curr=v[j];
          /* curr e' l'elemento da piazzare */

    while((v[h]<curr) && (h<j))
       h++;
          /* curr va inserito in posizione h */

    shiftR(v,h,j);
     v[h]=curr;
    j++;
   }
}
```

Ordinamento per selezione del minimo (selection sort)

- **Esempio:** Ordinamento di un mazzo di carte
  - si seleziona la carta più piccola e si mette da parte
  - delle rimanenti si seleziona la più piccola e si mette da parte
  - ...
  - si termina quando rimane una sola carta
- Ordinamento di un vettore:
  - per selezionare l'elemento più piccolo tra quelli rimanenti si utilizza un ciclo
  - mettere da parte significa scambiare con l'elemento che si trova nella posizione che compete a quello selezionato

- in verde la parte che rimane da analizzare
  in blu l'elemento minimo selezionato
  in marrone lo scambio effettuato
  in rosso la parte ordinata

5	2	4	6	1	3	
	5	2	4	6	1	3
	1	2	4	6	5	3
1	2	4	6	5	3	
	1	2	4	6	5	3
	1	2	4	6	5	3
1	2	4	6	5	3	
	1	2	4	6	5	3
	1	2	3	6	5	4
1	2	3	6	5	4	
	1	2	3	6	5	4
	1	2	3	4	5	6
1	2	3	4	5	6	
	1	2	3	4	5	6
	1	2	3	4	5	6
1	2	3	4	5	6	

1 2 3 4 5 6

Implementazione

Ordinamento per selezione

```
int minPos(int v[], int from, int to);
/* calcola la posizione del minimo elemento di
   v nella porzione [from,to]                  */

void swap(int *p, int *q);
/* scambia le variabili puntate da p e q */

/** PROCEDURA DI ORDINAMENTO PER SELEZIONE **/

void sort(int v[], int dim)
{
  int i, min;
  for(i=0; i<dim-1; i++)
    {
      min = minPos(v, i, dim-1);
      swap(v+i, v+min);
    }
}
```

Scrivere per **esercizio** le procedure swap e minpos

```
int minPos(int v[], int from, int to) {
/* calcola la posizione del minimo elemento di
   v nella porzione [from,to]                */

    int i, pos;
    pos = from;
    for (i=from+1; i<=to; i++)
        if (v[i] < v[pos])
            pos = i;
    return pos;
}

void swap(int *p, int *q) {
/* scambia le variabili puntate da p e q */
    int temp = *p;
    *p = *q;
    *q = temp;
}
```

Ordinamento a bolle (bubble sort)

- 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
  - ...
  - 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

5	2	4	6	1	3	
	5	2	4	6	1	3
	5	2	4	1	6	3
	5	2	1	4	6	3
	5	1	2	4	6	3
	1	5	2	4	6	3
1	5	2	4	6	3	
	1	5	2	4	3	6
	1	5	2	3	4	6
	1	5	2	3	4	6
	1	2	5	3	4	6
1	2	5	3	4	6	
1	2	3	5	4	6	
1	2	3	4	5	6	
1	2	3	4	5	6	

```
/** PROCEDURA BUBBLE SORT **/


void sort(int v[], int dim)
{
  int temp,i,j;
  for (i = 0; i < dim-1; i++)      /* fase i-esima */

    for (j = dim-1; j > i; j--)    /* bolla piu' leggera in posizione i */
      if (v[j] < v[j-1])
        swap(v+j, v+j-1);
}
```

Ordinamenti ricorsivi

Selection Sort ricorsivo

- Il metodo del selection sort può essere facilmente realizzato in modo ricorsivo
- si definisce una procedura che ordina (ricorsivamente) la porzione di array individuata da due indici `from` e `to`
- il minimo elemento della porzione viene messo in posizione `from` per poi ordinare ricorsivamente la porzione tra `from+1` e `to`
- Il caso base corrisponde all'ordinamento di una porzione fatta da un solo elemento (è già ordinata)

```
void SelectionSort(int v[], int from, int to){
  if (from < to) {
    int min = minPos(v,from,to);
    swap(v+from, v+min);
    SelectionSort(v, from+1, to);
  } }

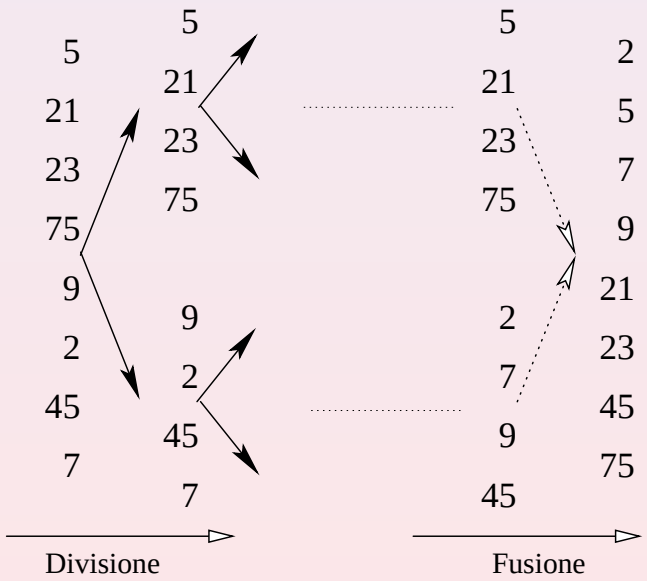
void sort(int v[], int dim) {
    SelectionSort(v,0,dim-1);
}
```

Merge sort

Si divide il vettore da ordinare in due parti:

- si ordina ricorsivamente la prima parte
- si ordina ricorsivamente la seconda parte
- si combinano (operazione di fusione, merge) le due parti ordinate

Esempio:



Esprimiamo il procedimento in uno pseudo-linguaggio

```
ordina per fusione gli elementi di A da from a to

IF from < to (c'è più di un elemento tra from e to)

THEN
      mid =  (from + to) /2
      ordina per fusione gli elementi di A da from a mid
      ordina per fusione gli elementi di A da mid +1 a to
      fondi
         gli elementi di A da from a mid con
         gli elementi di A da mid+1 a to
         restituendo il risultato nel sottovettore
          di A da from a to
```

Implementiamo l'algoritmo in C, definendo una procedura ricorsiva

```
void mergeRicorsivo(int A[], int from, int to)
```

che ordina la porzione dell'array A individuata dagli indici from e to.

Mergesort

```
void mergeRicorsivo(int A[], int from, int to)

{
  int mid;

  if (from < to) {              /* l'intervallo da mid a to, estremi
                                   inclusi, comprende almeno due elementi */
    mid = (from + to) / 2;

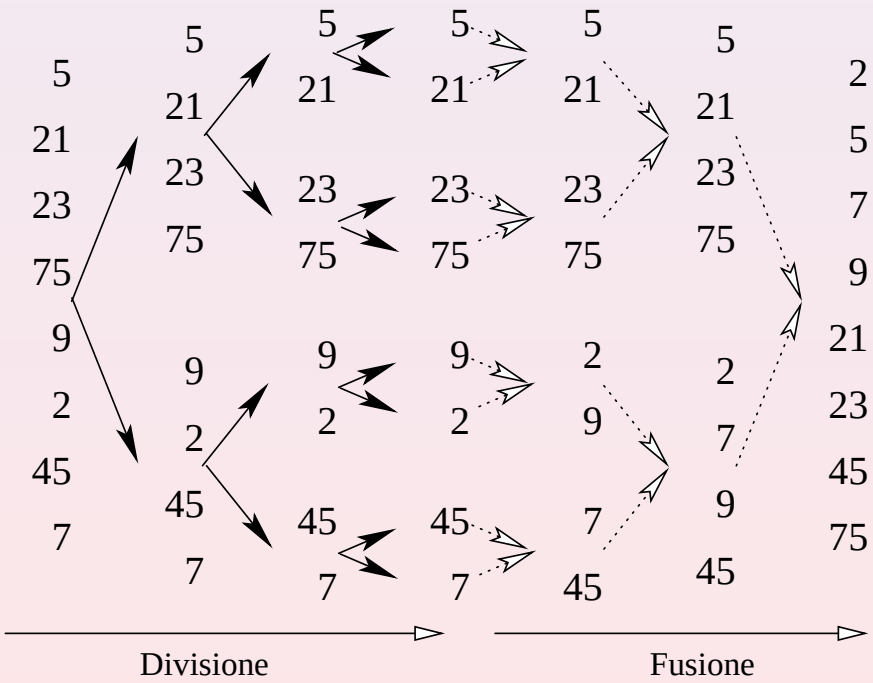
    mergeRicorsivo(A, from, mid);
    mergeRicorsivo(A, mid+1, to);

    merge(A, from, mid, to);    /* fonde le due porzioni ordinate [from, mid],
                                   [mid+1, to] nel sottovettore [from, to]  */
  }
}
```

La procedura `mergeSort` che ordina un array di interi è semplicemente

```
void sort(int v[], int dim)
{
mergeRicorsivo(v, 0, dim-1);
}
```

Esempio:



- Vediamo l'operazione di *fusione*, definendo la procedura

  `void merge(int A[], int from, int mid, int to)`

  che fonde le due porzioni dell'array `A` con indici compresi tra `from` e `mid` e tra `mid+1` e `to`.

- La procedura utilizza un array di supporto `B`: per semplicità, supponiamo di avere una costante `LUNG` che definisce la lunghezza degli array che stiamo trattando.

```
void merge(int A[], int from, int mid, int to)
 {
  int B[LUNG];                                /* vettore di appoggio */
  int primo, secondo, appoggio, da_copiare;

  primo = from;
  secondo = mid + 1;
  appoggio = from;

  while (primo <= mid && secondo <= to) { /* copia in modo ordinato      */
    if (A[primo] <= A[secondo]) {         /* gli elementi della prima e  */
      B[appoggio] = A[primo];             /* della seconda porzione in B */
      primo++;                            /* fino ad esaurire una delle due */
    }
    else {
      B[appoggio] = A[secondo];
      secondo++;
    }
    appoggio++;
  }
```

```
 if (secondo > to)              /* e' finita prima la seconda porzione */
 /* copia da A in B tutti gli elementi della
    prima porzione fino a mid */

    for (da_copiare = primo; da_copiare <= mid; da_copiare++) {
       B[appoggio] = A[da_copiare];
       appoggio++;
    }
  else                           /* e' finita prima la prima porzione */

    for (da_copiare = secondo; da_copiare <= to; da_copiare++) {
    /* copia da A in B tutti gli elementi della
    /* seconda porzione fino a to */

       B[appoggio] = A[da_copiare];
       appoggio++;
    }

/* ricopia tutti gli elementi da from a to da B ad A */
  for (da_copiare = from; da_copiare <= to; da_copiare++)
    A[da_copiare] = B[da_copiare];
}
```

Tipi user-defined

- Il C mette a disposizione un insieme di tipi di dato predefiniti (tipi built-in) e dei meccanismi per definire nuovi tipi (tipi user-defined)
- Vediamo le regole generali che governano la definizione di nuovi tipi e quindi i costrutti linguistici (costruttori) che il C mette a disposizione.
- Tutti i tipi non predefiniti utilizzati in un programma devono essere dichiarati come ogni altro elemento del programma. Una dichiarazione di tipo viene fatta di solito nella parte dichiarativa del programma.
  - parte dichiarativa globale:
    - dichiarazioni di costanti
    - dichiarazioni di tipi
    - dichiarazioni di variabili
    - prototipi di funzioni/procedure

Dichiarazione di tipo

- Una dichiarazione di tipo (type declaration) consiste nella parola chiave typedef seguita da:
  - la rappresentazione o costruzione del nuovo tipo (ovvero la specifica di come è costruito a partire dai tipi già esistenti)
  - il nome del nuovo tipo
  - il simbolo ; che chiude la dichiarazione

  Esempio: `typedef int anno;`
- Una volta definito e nominato un nuovo tipo, è possibile utilizzarlo per dichiarare nuovi oggetti (ad es. variabili) di quel tipo.

  Esempio:

```
float x;
anno a;
```

- **Nota:** In `C` si possono anche definire tipi senza usare `typedef`. Quest'ultima consente l'associazione di un nome (identificatore) a un nuovo tipo. Per uniformità e leggibilità del codice useremo spesso `typedef` per definire nuovi tipi.

Tipi semplici user-defined

Ridefinizione: Un nuovo tipo può essere definito rinominando un tipo già esistente (cioè creandone un alias)

typedef TipoEsistente NuovoTipo;

dove `TipoEsistente` può essere un tipo built-in o user-defined.

Esempio:

```
typedef int anno;
typedef int naturale;

typedef char carattere;
```

Enumerazione: Consente di definire un nuovo tipo enumerando i suoi valori, con la seguente sintassi

typedef enum {v1, v2, ... , vk} NuovoTipo;

Esempio:

```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
typedef enum {gen, feb, mar, apr, mag, giu,
              lug, ago, set, ott, nov, dic} Mese;

typedef enum {m, f} sesso;
```

- I valori elencati nella definizione di un nuovo tipo enumerato, sono identificatori che rappresentano costanti di quel tipo (esattamente come `0, 1, 2, ...` sono costanti del tipo `int`, o `'a', 'b', ...` sono costanti del tipo `char`).
- Dunque, se dichiariamo una variabile
  `Giorno g;`
  possiamo scrivere l'assegnamento
  `g = mar;`
- Le costanti dei tipi enumerati non vanno racchiuse tra virgolette o tra apici!

  N.B. Il compilatore associa ai nomi utilizzati per denotare le costanti dei tipi enumerati valori naturali progressivi.

  Esempio: il valore associato a `g` dopo l'assegnamento `g=mar` è il numero naturale (intero) `1`.

  $\Longrightarrow$ mancanza di astrazione: è possibile fare riferimento alla rappresentazione dei valori.

- La relazione tra interi e tipi enumerati consente di applicare a questi ultimi le seguenti operazioni:
  - operazioni aritmetiche: `+,-,*,/,%`
  - uguaglianza e disuguaglianza: `=, !=`
  - confronto: `<,<=,>,>=`
- Si noti che la relazione di precedenza tra i valori (che determina l'esito delle operazioni di confronto) dipende dall'ordine in cui vengono elencati i valori del tipo al momento della sua definzione.

  Esempio: Con le dichiarazioni viste in precedenza
  `lun < gio` è vero (un intero diverso da `0`) `apr <= feb` è falso (il valore intero `0`)
- Il C tratta questi tipi come ridefinizione di `int`

Tipi fai da te: i booleani

Soluzione 1

```
#define FALSE 0;
#define TRUE 1;...
typedef int Boolean;
Boolean b;
...
```

Soluzione 2

```
typedef enum {FALSE, TRUE} Boolean;...
Boolean b;
...
```

N.B. I valori vanno elencati come sopra, rispettando la convenzione adottata dal C: il valore 0 rappresenta falso.

Esempio:

```
typedef enum {false, true} boolean;

boolean even (int n)
{
if (n % 2 == 0)
    return true;
else
    return false;
}


boolean implies (boolean p, boolean q)
{
    if (p)
        return q;
    else
        return true;
}
```

Esempio: Uso del costrutto `switch` con tipi enumerati

```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
Giorno g;
...
switch (g) {
case lun: case mar: case mer: case gio: case ven:
        printf("Giorno lavorativo");
        break;
case sab: case dom:
        printf("Week-end");
        break;
}
```

```
void stampaGiorno(Giorno g) {
switch (g) {
case lun: printf("lun");
          break;
...

case dom: printf("dom");
          break;
}
```

Tipi strutturati user-defined

- Il C non possiede tipi strutturati built-in, ma fornisce dei costruttori che permettono di definire tipi strutturati anche piuttosto complessi.
- Array e puntatori possono essere visti come costruttori di tipo (definiscono un tipo di dato non semplice a partire da tipi esistenti).

Uso di `typedef` con array e puntatori

- In generale, una dichiarazione di tipo mediante `typedef` ha la forma di una dichiarazione di variabile preceduta dalla parola chiave `typedef`, e con il nome di tipo al posto del nome della variabile.
- Nel caso di array e puntatori:

```
typedef TipoElemento TipoArray[Dimensione];
typedef TipoPuntato *TipoPuntatore;
```

Esempio:

```
typedef int ArrayDieciInteri[10];
typedef int MatriceTreXQuattro[3][4];
typedef int *PuntIntero;
ArrayDieciInteri vet;           /*  int vet[10]; */
PunIntero p;                    /*  int *p; */
MatriceTreXQuattro mat, mat1;   /*  int mat[3][4]; int mat1[3][4]; */
```

Il costruttore struct

- Una struttura è un'aggregazione di elementi che possono essere eterogenei (di tipo diverso).
  Esempio:

```
struct persona  {
     char nome[15];
     char cognome[20];
     int eta;
     sesso s;   }
```

  - la parola chiave `struct` introduce la definizione della struttura
  - `persona` è l'etichetta della struttura, attribuisce un nome alla definizione della struttura
  - `nome`, `cognome`, `eta`, `s` sono detti campi della struttura
- È anche possibile definire strutture con campi omogenei

```
struct complex  {
 double real;
 double imag;  }
```

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 altri nomi già utilizzati (es. per 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
- un campo può però essere di tipo puntatore alla struttura

  Esempio:

```
struct s { int a;
           struct s *p;  };
```

Dichiarazione di variabili di tipo struttura

- La definizione di una struttura non provoca allocazione di memoria, ma introduce un nuovo tipo di dato.

  Esempio: `struct persona tizio, docenti[10], *p;`
  - `tizio` è una variabile di tipo `struct persona`
  - `docenti` è un vettore di 10 elementi di tipo `struct persona`
  - `p` è un puntatore a una `struct persona`
  - N.B.: `persona tizio;` Errore!
- Una variabile di tipo struttura può essere dichiarata contestualmente alla definizione della struttura.

  Esempio:

```
struct studente {          |      struct  {
  char nome[20];           |        char nome[20];
  long matricola;          |        long matricola;
  struct data ddn;         |        struct data ddn;
} s1, s2;                  |      } s1, s2;
```

- In questo caso si può anche omettere l'etichetta di struttura.

Uso di typedef con strutture

- Attraverso `typedef` è possibile associare un nome ad un tipo definito mediante il costruttore `struct`.

  Esempio:

```
struct data { int giorno, mese, anno; };

typedef struct data Data;
```

- `Data` è un sinonimo di `struct data`, che può essere utilizzato nelle dichiarazioni di variabili.

```
Data d1, d2;
Data appelli[10], *pd;
```

Operazioni sulle strutture

- Si possono assegnare variabili di tipo struttura a variabili dello stesso tipo struttura.

  Esempio:

```
Data d1, d2;
...
d1 = d2;
```

- Non è possibile invece effettuare il confronto tra due variabili di tipo struttura.

  Esempio:

```
struct data d1, d2;
if (d1 == d2) ...
```

Errore!

- 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` e `b` sono dello stesso tipo
`a = c;`	Errore!	`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.

Accesso ai campi di una struttura

- I campi di una struttura si comportano come variabili del tipo corrispondente. L'accesso avviene tramite l'operatore punto

```
Data oggi;
oggi.giorno = 11; oggi.mese = 5; oggi.anno = 2009;
printf("%d %d %d", oggi.giorno, oggi.mese, oggi.anno);
```

- Accesso tramite un puntatore alla struttura.

```
Data oggi, *pd;
pd = &oggi;
(*pd).giorno = 11; (*pd).mese = 5; (*pd).anno = 2009;
```

N.B. Ci vogliono le **()** perché "." ha priorità più alta di "*".

- Operatore freccia: combina il dereferenziamento e l'accesso al campo della struttura.

```
pd->giorno =11; pd->mese = 5; pd->anno = 2009;
```

- N.B.: `pd->giorno` è una abbreviazione per `(*pd).giorno`.

Esempio: Accesso al campo di una struttura che è a sua volta campo di un'altra struttura.

```
struct dipendente
   { Persona datiDip;
     Data dataAssunzione;
     int stipendio;
   };
typedef struct dipendente Dipendente;

Dipendente dip, *p;
...
dip.dataAssunzione.giorno = 3;
dip.dataAssunzione.mese = 4;
dip.dataAssunzione.anno = 1997;
...
(p->dataAssunzione).giorno = 5;
(p->stipendio) = (p->stipendio) + 120;
```

Inizializzazione di strutture

- Può avvenire, come per i vettori, con un elenco di inizializzatori.

  Esempio: `Data oggi = { 11, 5, 2009 }`

- Se ci sono meno inizializzatori di campi della struttura, i campi rimanenti vengono inizializzati a 0 (o al valore speciale `NULL`, se il campo è un puntatore).

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 comunque possibile simulare il passaggio per indirizzo attraverso un puntatore

Nota: per passare per valore ad una funzione un vettore (il vettore, non il puntatore al suo primo elemento) è sufficiente racchiuderlo in una struttura.

Esempio:

```
struct dipendente
   { Persona datiDip;
     Data dataAssunzione;
     int stipendio;
   };
typedef struct dipendente Dipendente;

void aumento(Dipendente *p, int percentuale)
{
    int incremento;
    incremento = (p -> stipendio) * percentuale / 100;
    p -> stipendio = p -> stipendio + incremento;
}
```

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

- Finora abbiamo usato gli array per realizzare tali strutture, nonostante ciò porti talvolta a un impiego inefficiente della memoria.
- Vediamo adesso un modo basato sull'allocazione dinamica di variabili, che ci permette di realizzare liste di elementi in maniera che la memoria fisica utilizzata corrisponda meglio a quella astratta, cioè al numero di elementi della sequenza che vogliamo rappresentare.

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite array
   - Vantaggi:
     - l'accesso agli elementi è diretto (tramite indice) ed efficiente
     - l'ordine degli elementi è quello in memoria $\Longrightarrow$ non servono strutture dati addizionali
     - è semplice manipolare l'intera struttura (copia, ordinamento, ...)
   - Svantaggi:
     - dobbiamo avere un'idea precisa della dimensione della sequenza
     - inserire o eliminare elementi è complicato ed inefficiente (comporta un numero di spostamenti che nel caso peggiore può essere dell'ordine del numero degli elementi della struttura)

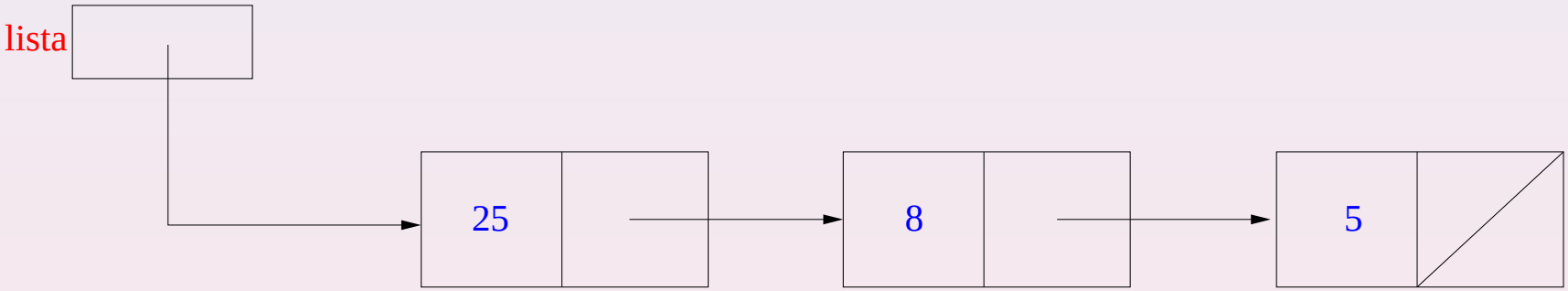
2. Rappresentazione collegata

- Una lista concatenata è una sequenza lineare di nodi, ciascuno dei quali memorizza un valore e contiene un riferimento (puntatore) al nodo successivo nella sequenza.
- Per aggiungere e cancellare nodi in qualunque posizione semplicemente aggiustando il sistema di puntatori senza operare sui nodi non interessati dalla aggiunta o dalla cancellazione.
- L'accesso agli elementi è di tipo sequenziale: per accedere al generico nodo, si deve scandire la lista, dato che l'accesso ad un elemento è possibile attraverso il puntatore contenuto nell'elemento precedente.

2. Rappresentazione collegata (continua)

- La sequenza di elementi viene rappresentata da una struttura di dati collegata, realizzata tramite strutture e puntatori.
- Ogni elemento è rappresentato con una struttura C:
  - un campo (o più campi se necessario) per l'elemento (ad es. `int`)
  - un campo puntatore alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)
- L'ultimo elemento non ha un elemento successivo
  - il campo puntatore ha valore `NULL` che assume quindi il significato di "fine lista".
- L'inizio della lista è individuato da una variabile del tipo dei puntatori ai vari elementi.
  - Sarà nostra abitudine attribuire a questa variabile il nome stesso della lista, identificando il concetto di "inizio lista" (o "testa della lista") con la lista stessa.
- l'accesso a una lista avviene attraverso il puntatore al primo elemento.

Graficamente



- La variabile lista, di tipo puntatore, è utilizzata per accedere alla sequenza.

Esempio: Sequenze di interi.

```
struct EL {
  int info;
  struct EL *next;
};
typedef struct EL ElementoLista;
typedef ElementoLista *ListaDiElementi;
```

1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;
2. la seconda dichiarazione utilizza `typedef` per ridenominare il tipo `struct EL` come `ElementoLista`;
3. la terza dichiarazione definisce il tipo `ListaDiElementi` come puntatore al tipo `ElementoLista`.

- A questo punto possiamo definire variabili di tipo lista:

```
ListaDiElementi Lista1, Lista2;
```

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

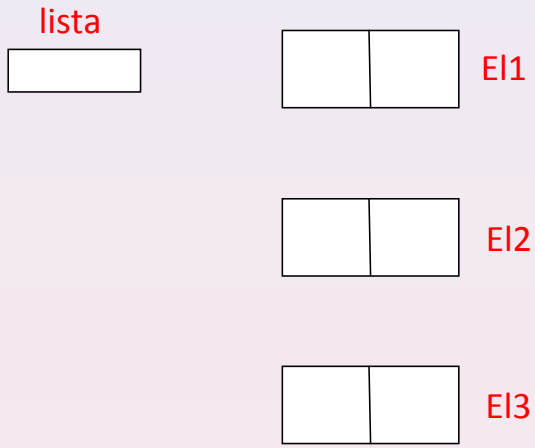
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Nota: per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

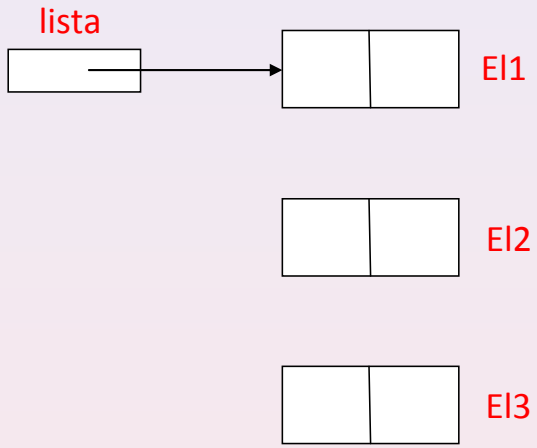
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Nota: per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

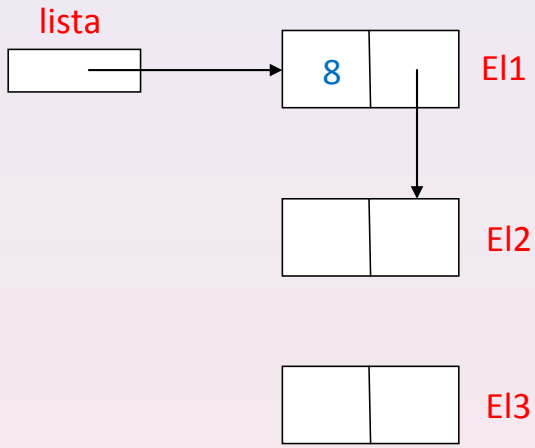
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Nota: per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

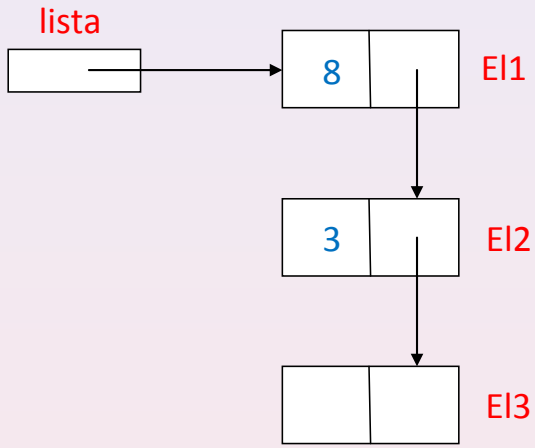
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Nota: per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

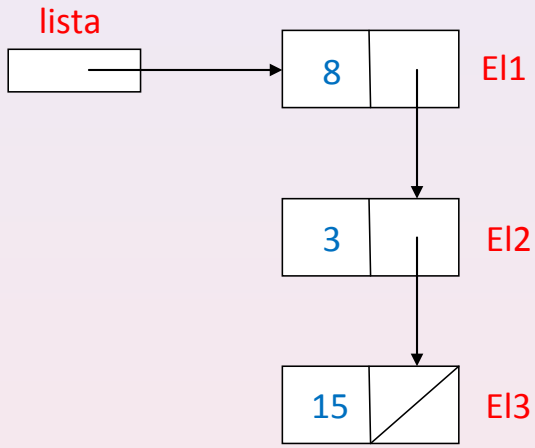
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Nota: per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

- Nel programma che abbiamo appena visto per creare una lista di tre elementi dobbiamo dichiarare tre variabili di tipo `ElementoLista`.
- Il numero degli elementi della lista deve essere deciso a tempo di compilazione.
- Non è possibile, per esempio, creare una lista con un numero di elementi letti in ingresso. Ma allora qual è il vantaggio rispetto all'array?
- Quello che abbiamo visto non è l'unico modo. . . .

Allocazione Dinamica della memoria

- L'allocazione dinamica della memoria è possibile in C grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard (standard library). Infatti è richiesta l'inclusione del file header `<stdib.h>`
- Le due funzioni principali sono
  - malloc: consente di allocare dinamicamente memoria per una variabile di un tipo specificato
  - free: consente di rilasciare dinamicamente memoria (precedentemente allocata con malloc)
- I tipi di dato sono ancora statici, ovvero hanno una dimensione fissata a priori. Le variabili di un certo tipo di dato possono invece essere create.

malloc

- La chiamata di funzione

  malloc(sizeof(TipoDato));

  crea in memoria una variabile di tipo TipoDato, e restituisce come risultato l'indirizzo della variabile creata.
- Se p è una variabile di tipo puntatore a TipoDato, l'istruzione

  p=malloc(sizeof(TipoDato));

  assegna l'indirizzo restituito dalla funzione malloc a p che punta quindi alla nuova variabile (p già esiste).
- Una variabile creata dinamicamente è necessariamente anonima: a essa si può fare riferimento solo tramite un puntatore

  a differenza di una variabile dichiarata mediante un proprio identificatore, che può essere riferita sia direttamente sia tramite un puntatore

free

- Se p è l'indirizzo di una variabile allocata dinamicamente, la chiamata

  free(p);

  rilascia lo spazio di memoria puntato da p

  la corrispondente memoria fisica è resa disponibile per qualsiasi altro uso.

- free deve ricevere come parametro attuale un puntatore al quale era stato assegnato come valore l'indirizzo restituito da una funzione di allocazione dinamica di memoria (cioè malloc).

Heap

- Poiché le variabili dinamiche possono essere create e distrutte in un qualsiasi punto del programma esse non possono essere allocate sullo stack.
- Vengono allocate in un'altra zona di memoria chiamata heap (mucchio). La loro gestione risulta molto più inefficiente.

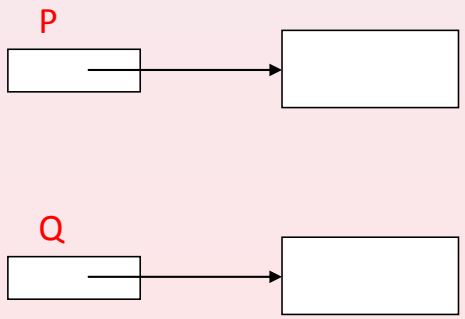
Produzione di garbage (spazzatura)

- Si verifica quando la memoria allocata dinamicamente risulta logicamente inaccessibile, e quindi sprecata, perché non esiste alcun riferimento ad essa.

  Esempio:

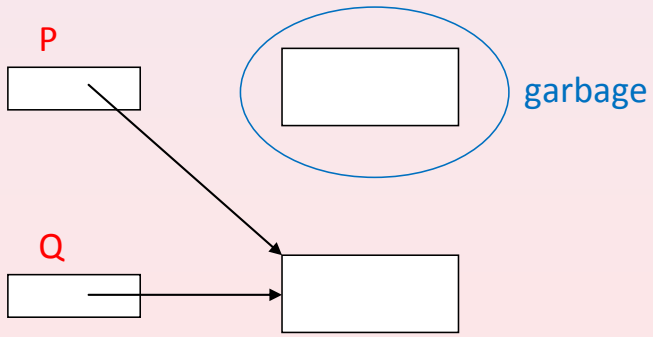
```
P=malloc(sizeof(TipoDato));
...
P=Q;
```

- In questo modo la cella puntata da `P` subito dopo l'assegnamento `P=Q` perde ogni possibilità di accesso (da cui il termine spazzatura).



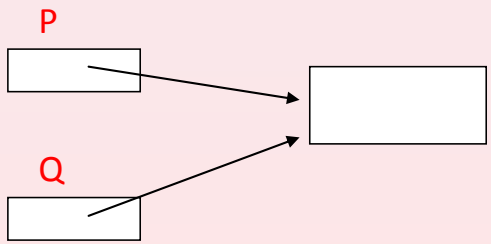
Produzione di garbage (spazzatura)

- Si verifica quando la memoria allocata dinamicamente risulta logicamente inaccessibile, e quindi sprecata, perché non esiste alcun riferimento ad essa.
  Esempio:
  ```
  P=malloc(sizeof(TipoDato));
  ...
  P=Q;
  ```
- In questo modo la cella puntata da `P` subito dopo l'assegnamento `P=Q` perde ogni possibilità di accesso (da cui il termine spazzatura).



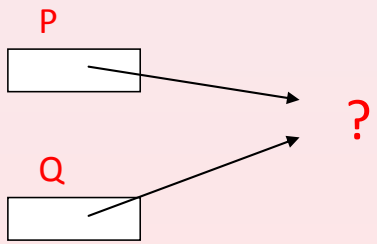
Riferimenti fluttuanti (dangling references)

- Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.
  Esempio:
  ```
  P=Q;
  free(Q);
  ```
- L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



Riferimenti fluttuanti (dangling references)

- Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.
  Esempio:
  ```
  P=Q;
  free(Q);
  ```
- L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



- Produzione di garbage e riferimenti fluttuanti hanno svantaggi simmetrici:
  - la prima comporta spreco di memoria
  - la seconda comporta risultati imprevedibili e scorretti.
- La seconda è più pericolosa della prima e in alcuni linguaggi non è prevista l'istruzione `free`.
- Viene lasciato al supporto del linguaggio l'onere di effettuare garbage collection ("raccolta rifiuti").

Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP

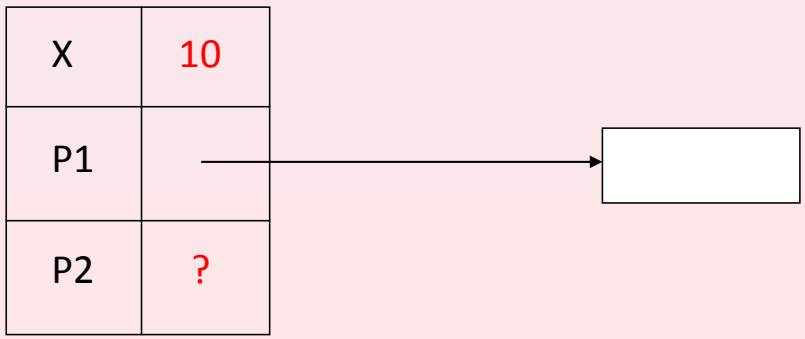
X	10
P1	?
P2	?

Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP

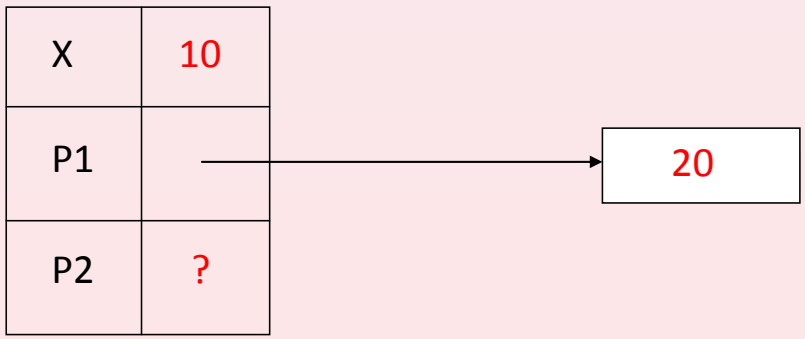


Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP

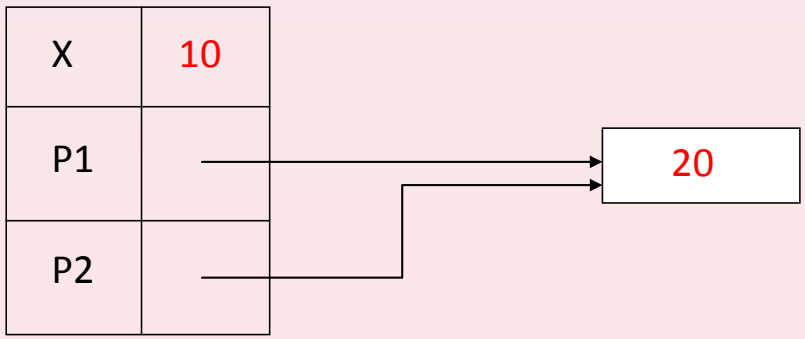


Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP

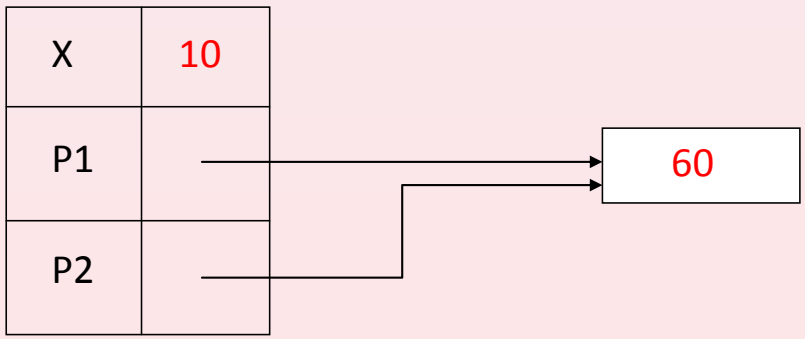


Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP

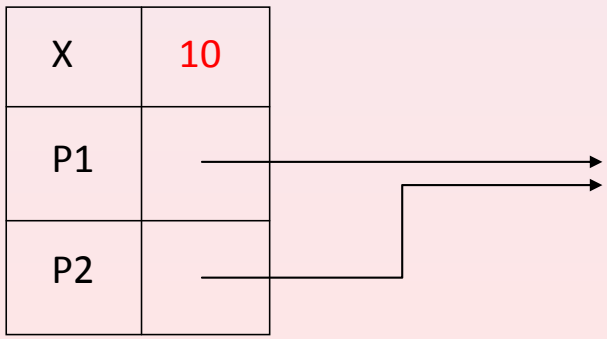


Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP



Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ListaDiElementi lista;              /* puntatore al primo elemento della lista */

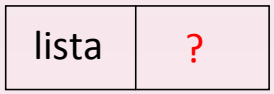
lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```

PILA HEAP



Creazione di una lista di tre interi fissati: (8, 3, 15)

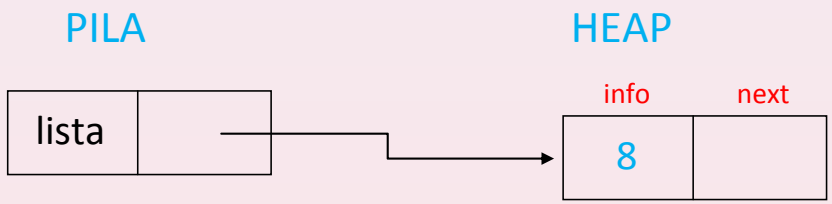
```
ListaDiElementi lista;              /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Creazione di una lista di tre interi fissati: (8, 3, 15)

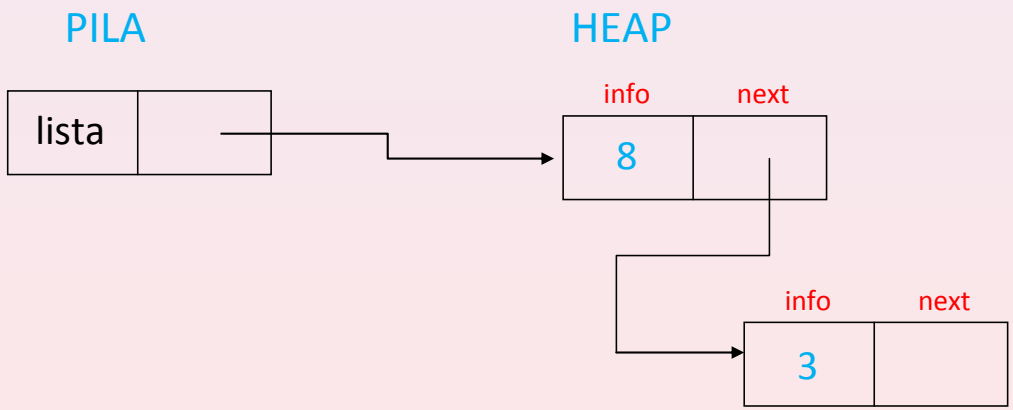
```
ListaDiElementi lista;              /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Creazione di una lista di tre interi fissati: (8, 3, 15)

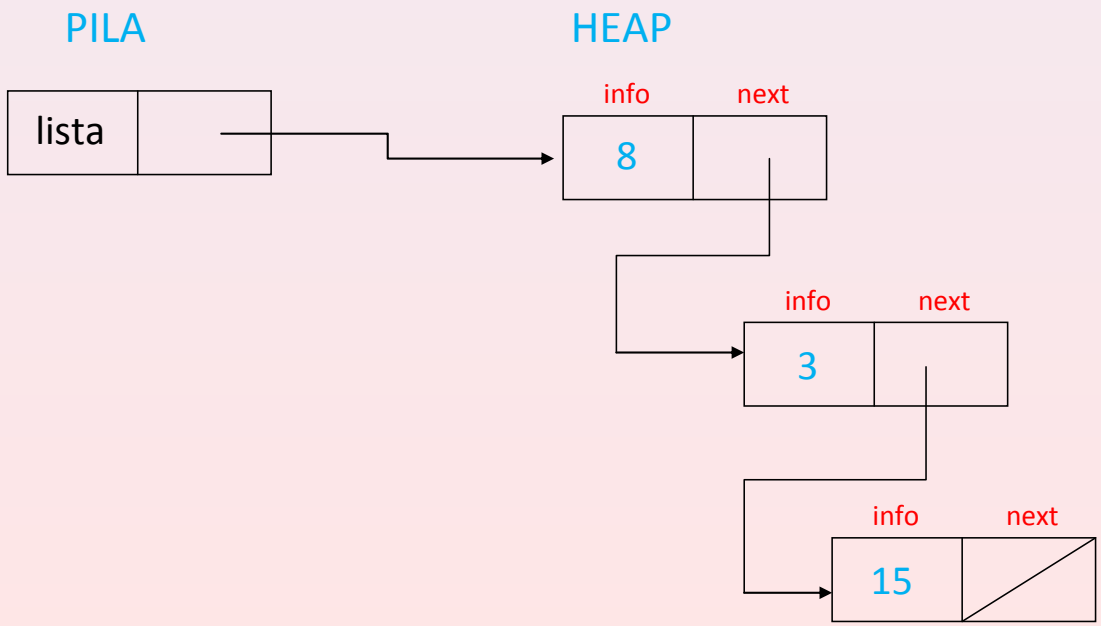
```
ListaDiElementi lista;              /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Osservazioni:

- `lista` è di tipo `ListaDiElementi`, quindi è un puntatore e non una struttura
- la zona di memoria per ogni elemento della lista (non per ogni variabile di tipo `ListaDiElementi`) deve essere allocata esplicitamente con `malloc`
- Esiste un modo più semplice di creare la lista di 3 elementi?
- Creiamo la lista a partire dal fondo!

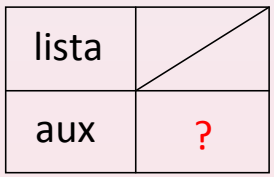
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;    aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;     aux->next = lista;
lista = aux;
```

PILA HEAP

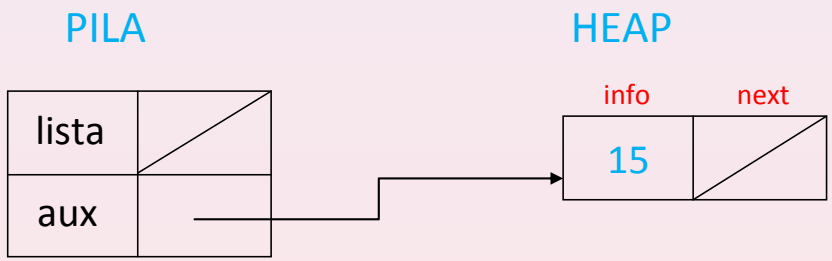


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;    aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;     aux->next = lista;
lista = aux;
```

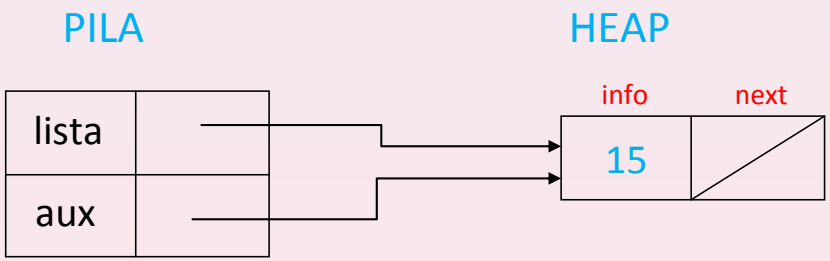


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

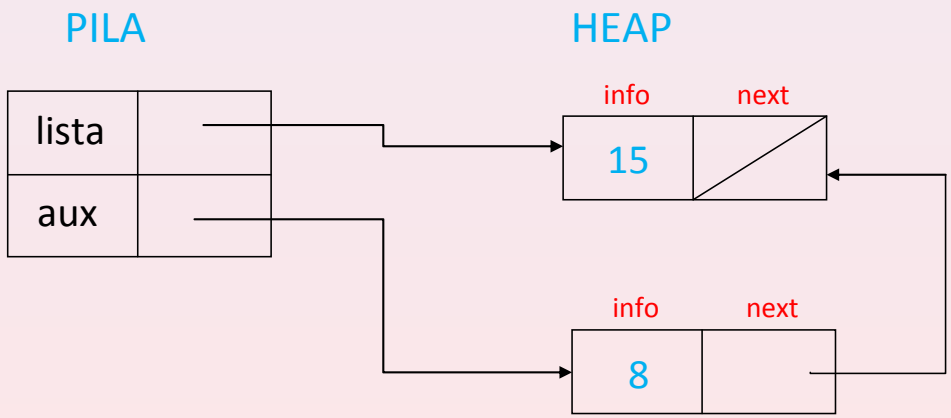


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

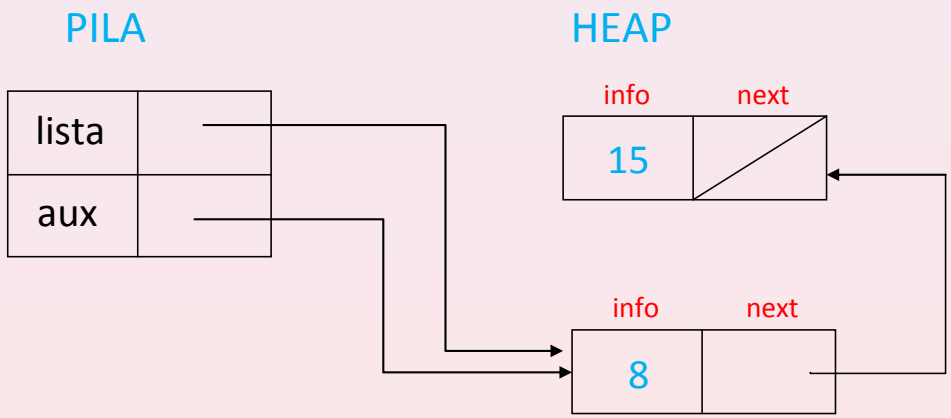


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

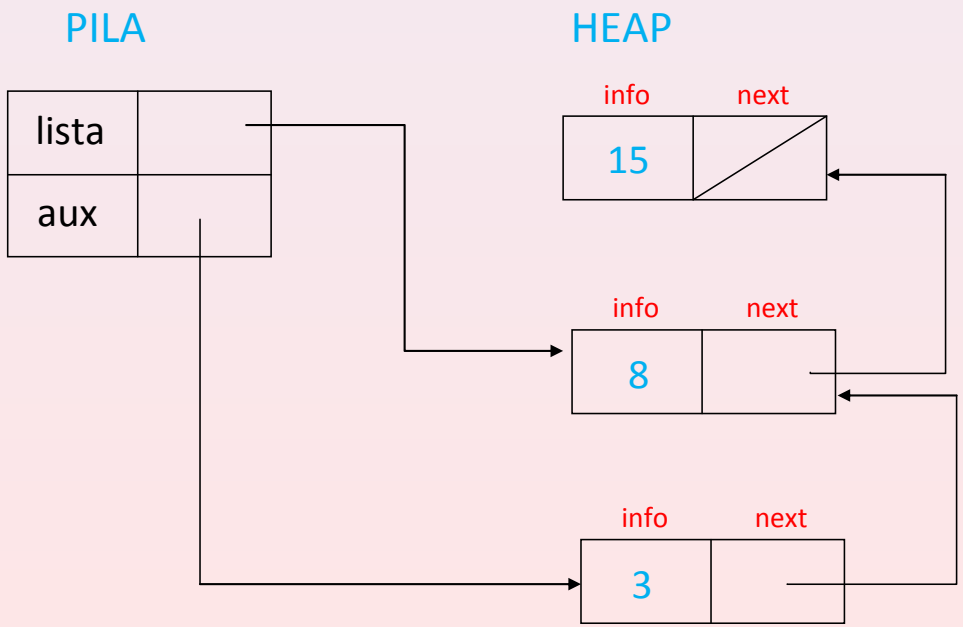


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

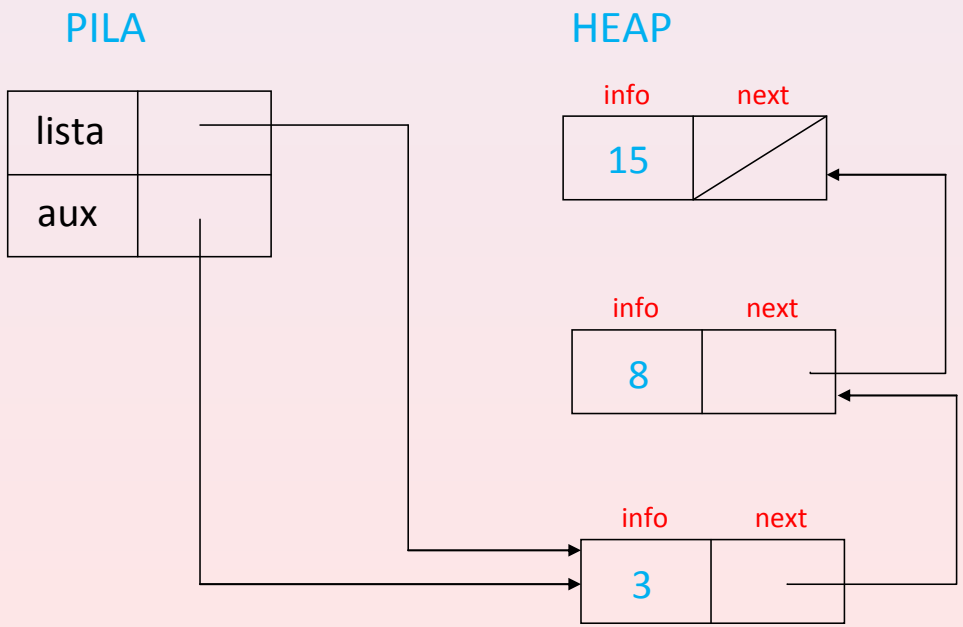


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



Operazioni sulle liste

- Definiamo una serie di procedure e funzioni per operare sulle liste.
- Usiamo liste di interi per semplicità, ma tutte le operazioni sono realizzabili in modo del tutto analogo su liste di altro tipo (salvo rare eccezioni)
- Facciamo riferimento alle dichiarazioni dei tipi `ElementoLista` e `ListaDiElementi` viste in precedenza

Inizializzazione

- Definiamo una procedura che inizializza una lista assegnando il valore `NULL` alla variabile testa della lista.
- Tale variabile deve essere modificata e quindi passata per indirizzo.
- Ciò provoca, nell'intestazione della procedura, la presenza di un puntatore a puntatore.

```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

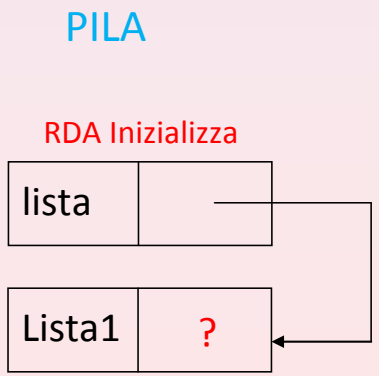
PILA

Lista1	?

```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

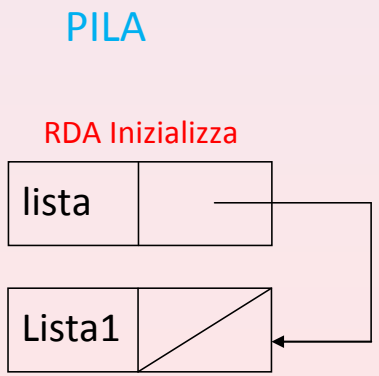
```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```



```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

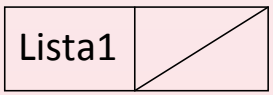


```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

PILA

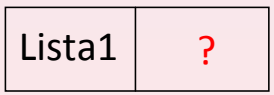


Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

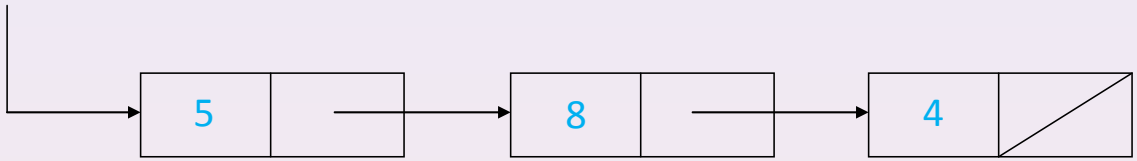
main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA



Stampa degli elementi di una lista

- Data la lista



vogliamo che venga stampato:

5 –> 8 –> 4 –> //

Versione iterativa:

Stampa Lista

```
void StampaLista(ListaDiElementi lis)
{
   while (lis != NULL)
    {
     printf("%d -->", lis->info);
     lis = lis->next;
    }
printf("//");
}
```

- N.B.: `lis = lis->next` fa puntare `lis` all'elemento successivo della lista
- Attenzione: Possiamo usare `lis` per scorrere la lista perché, avendo utilizzato il passaggio per valore, le modifiche a `lis` non si ripercuotono sul parametro attuale.

```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

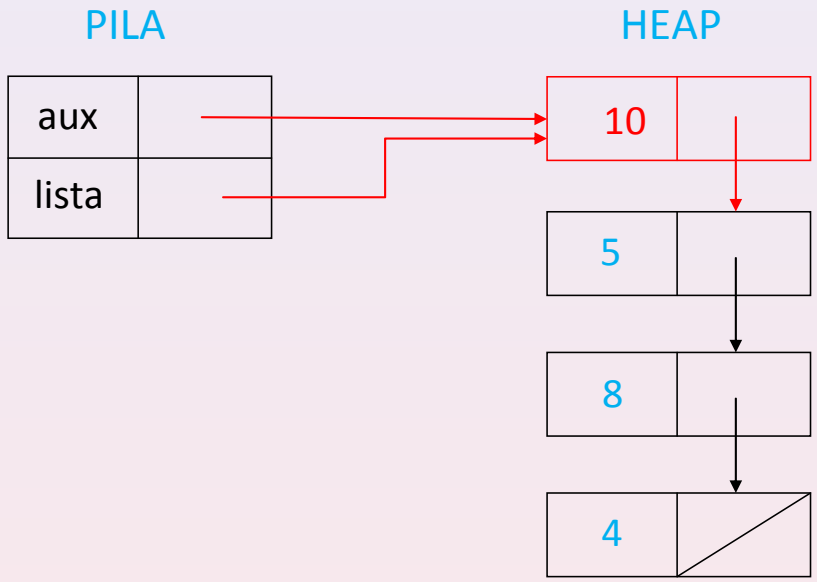
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

Versione ricorsiva

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```

Inserimento di un nuovo elemento in testa



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
   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

```
void InserisciTestaLista(ListaDiElementi *lista, int elem)
{
  ListaDiElementi aux;

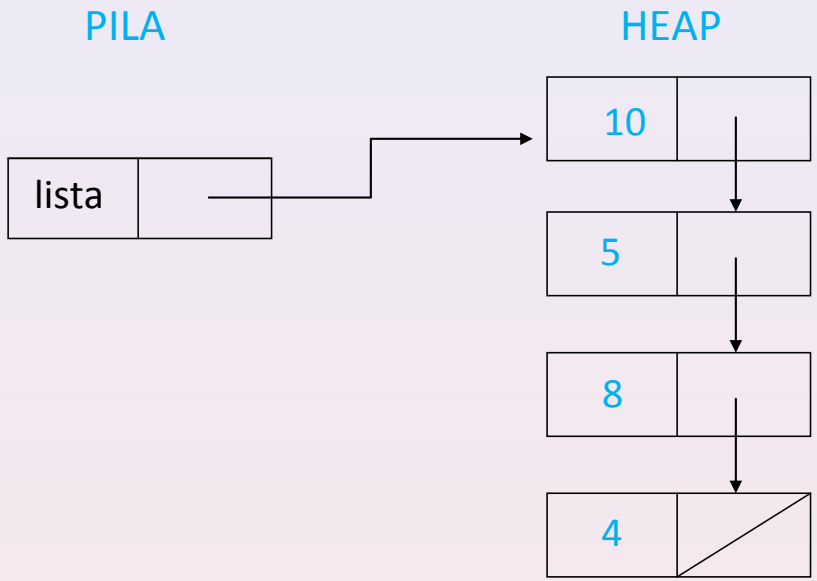
  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = *lista;
  *lista = aux;
}
```

- il primo parametro è la lista da modificare (passata per indirizzo)
- il secondo parametro è l'elemento da inserire (passato per indirizzo)
  - Attenzione: nel caso di liste di tipo `TipoElemLista` la procedura può essere generalizzata se su tale tipo è definito l'assegnamento

Esercizio

Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

Cancellazione del primo elemento



- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}
```

Cancellazione di tutta una lista

Una prima soluzione

```
void CancellaLista(ListaDiElementi *lista)
{
  ListaDiElementi aux;

  while (*lista != NULL) {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
  }
}
```

- Osserviamo che il corpo del ciclo corrisponde alle azioni della procedura `CancellaPrimo`. Possiamo allora scrivere:

  Possiamo allora scrivere:

```
void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}
```

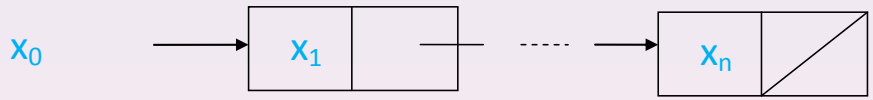
- Si noti il parametro attuale della chiamata a `CancellaPrimo`, che è `lista` (di tipo `ListaDiElementi *`) e non `&lista`

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

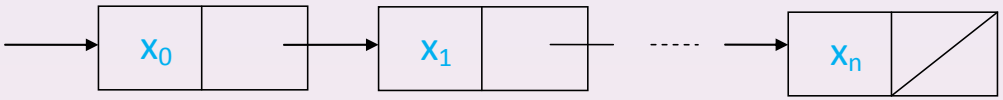
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

Visione ricorsiva delle liste



- Una lista di elementi è una struttura dati ricorsiva per sua natura
  1. data una lista L di elementi $x_1, \ldots, x_n$
  2. dato un ulteriore elemento x_0
  3. anche la concatenazione di x_0 e L è una lista
- Si noti che in 1. L può anche essere la lista vuota

Visione ricorsiva delle liste



- Una lista di elementi è una struttura dati ricorsiva per sua natura
  1. data una lista L di elementi $x_1, \ldots, x_n$
  2. dato un ulteriore elemento x_0
  3. anche la concatenazione di x_0 e L è una lista
- Si noti che in 1. L può anche essere la lista vuota

Cancellazione lista: versione ricorsiva

- Sfruttiamo la visione ricorsiva della struttura dati lista per realizzare la cancellazione in modo ricorsivo
  1. la cancellazione della lista vuota non richiede alcuna azione
  2. la cancellazione della lista ottenuta come concatenazione dell'elemento x e della lista L richiede l'eliminazione di x e la cancellazione di L
- la traduzione in C è immediata

```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

Appartenenza di un elemento ad una lista

- Ricordiamo la ricerca sequenziale sui vettori
- sostituiamo l'indice i con un puntatore p
- scorriamo la lista attraverso p
- l'elemento corrente è quello puntato da p
- Incapsuliamo questo codice in una funzione a valori booleani

Appartenenza di un elemento ad una lista

```
boolean Appartiene(TipoElementoLista elem, ListaDiElementi lista)
{
  boolean trovato = false;

  while (lista != NULL && !trovato)
    if (lista->info==elem)
      trovato = true;
    else
      lista = lista->next;
  return trovato;
}
```

- Non c'è bisogno di un puntatore ausiliario per scorrere la lista

  $\Longrightarrow$ il passaggio per valore consente di scorrere utilizzando il parametro formale!
- Abbiamo assunto che sul tipo `TipoElementoLista` sia definito l'operatore di uguaglianza ==

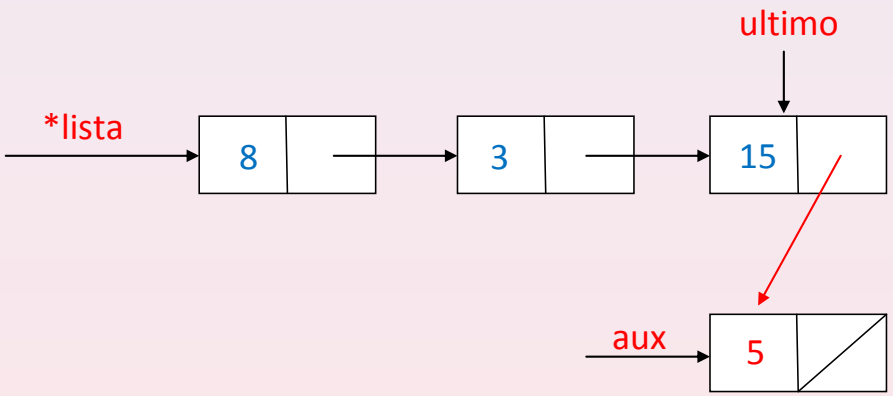
Versione ricorsiva

```
boolean Appartiene(TipoElementoLista elem, ListaDiElementi lis)
{
  if (lis == NULL)
    return false;
  else if (lis->info==elem)
    return true;
  else
    return (Appartiene(elem, lis->next));
}
```

- Un elemento elem
  - non appartiene alla lista vuota
  - appartiene alla lista con testa x se elem coincide con x
  - appartiene alla lista con testa x diversa da elem e resto L se e solo se appartiene a L

Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



Codice della versione iterativa

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi ultimo;   /* puntatore usato per la scansione */
  ListaDiElementi aux;
                            /* creazione del nuovo elemento */
  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = NULL;

  if (*lista == NULL)
    *lista = aux;
  else {
    ultimo = *lista;
    while (ultimo->next != NULL)
      ultimo = ultimo->next;
                        /* concatenazione del nuovo elemento in coda alla lista */
    ultimo->next = aux;
  }
}
```

Inserimento ricorsivo di un elemento in coda

Caratterizzazione induttiva dell'inserimento in coda

Sia nuovaLista la lista ottenuta inserendo elem in coda a lista.

- se lista è vuota, allora nuovaLista è costituita dal solo `elem` (caso base)
- altrimenti nuovaLista è ottenuta da lista facendo l'inserimento di elem in coda al resto di lista (caso ricorsivo)

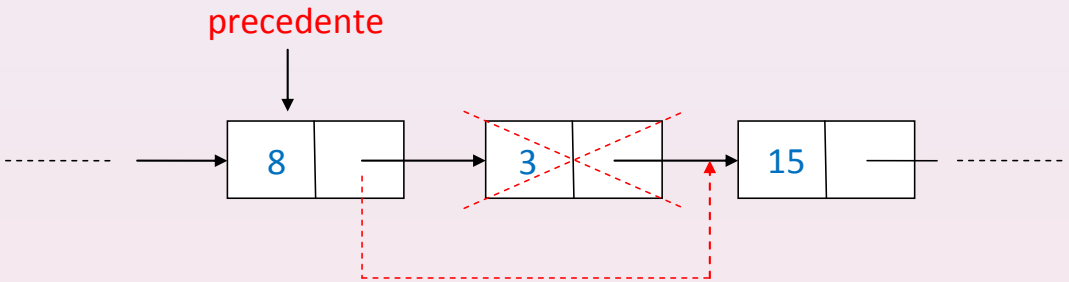
```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
  else
    InserisciCodaLista(       ??       , elem);}
```

Cancellazione della prima occorrenza di un elemento

- 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
     $\Longrightarrow$ passaggio per indirizzo!!
  2. l'elemento non è né il primo né 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

```
void CancellaElementoLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi prec;    /* puntatore all'elemento precedente */
  ListaDiElementi corr;    /* puntatore all'elemento corrente */
  boolean trovato;        /* usato per terminare la scansione */

  if (*lista != NULL)
    if ((*lista)->info==elem)
        { /* cancella il primo elemento */
            CancellaPrimo(lista);
        }
    else   { /* scansione della lista e cancellazione dell'elemento */
            prec = *lista; corr = prec->next; trovato = false;
            while (corr != NULL && !trovato)
              if (corr->info == elem)
                {   /* cancella l'elemento */
                    trovato = true;          /* provoca l'uscita dal ciclo */
                    prec->next = corr->next;
                    free(corr);  }
              else  {
                    prec = prec->next;    /* avanzamento dei due puntatori */
                    corr = corr->next;   }
}}
```

Versione ricorsiva:

```
void CancellaElementoLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  if (*lista != NULL)
    if ((*lista)->info== elem)
        { /* cancella il primo elemento */
            CancellaPrimo(lista);
        }
    else   /* cancella elem dal resto */
        CancellaElementoLista(&((*lista)->next), elem);
}
```

Cancellazione di tutte le occorrenze di un elemento

Versione iterativa

- analoga alla cancellazione della prima occorrenza
- però, dopo aver trovato e cancellato l'elemento, bisogna continuare la scansione
- ci si ferma solo quando si è arrivati alla fine della lista

  $\Longrightarrow$ non serve la sentinella booleana per fermare la scansione

Cancellazione di tutte le occorrenze di un elemento

Caratterizzazione induttiva

Sia ris la lista ottenuta cancellando tutte le occorrenze di elem da lista. Allora:

1. se lista è la lista vuota, allora ris è la lista vuota (caso base)
2. altrimenti, se il primo elemento di lista è uguale ad elem, allora ris è ottenuta da lista cancellando il primo elemento e tutte le occorrenze di elem dal resto di lista (caso ricorsivo)
3. altrimenti ris è ottenuta da lista cancellando tutte le occorrenze di elem dal resto di lista (caso ricorsivo)

Esercizio

Implementare le due versioni

Versione iterativa

```
void CancellaTuttiLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi prec;    /* puntatore all'elemento precedente */
  ListaDiElementi corr;    /* puntatore all'elemento corrente */
  boolean trovato = false;
  while ((*lista != NULL) && !trovato)  /* cancella le occorrenze */
    if ((*lista)->info!=elem)           /* di elem in testa       */
        trovato = true;
    else  CancellaPrimo(lista);

   if (*lista != NULL)
        {
            prec = *lista; corr = prec->next;
            while (corr != NULL)
              if (corr->info == elem)
                {   /* cancella l'elemento */
                    prec->next = corr->next;
                    free(corr);
                    corr = prec->next;}
              else  {
                    prec = prec->next;    /* avanzamento dei due puntatori */
                    corr = corr->next;   }
        }
}
```

Versione ricorsiva

```
void CancellaTuttiLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi aux;

  if (*lista != NULL)
    if ((*lista)->info==elem)
    {                                  /* cancellazione del primo elemento */
      CancellaPrimo(lista);
                               /* cancellazione di elem dal resto della lista */
      CancellaTuttiLista(lista, elem);
    }
    else
      CancellaTuttiLista(&((*lista)->next), elem);
}
```

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**

  Versione ricorsiva

- Caratterizziamo il problema induttivamente
- Sia ris la lista ottenuta inserendo l'elemento elem nella lista ordinata lista.
  1. se lista è la lista vuota, allora ris è costituita solo da elem (caso base)
  2. se il primo elemento di lista è maggiore o uguale a elem, allora ris è ottenuta da lista inserendo elem in testa (caso base)
  3. altrimenti ris è ottenuta da lista inserendo ordinatamente elem nel resto di lista (caso ricorsivo)

```
void InserzioneOrdinata(ListaDiElementi *lista, int elem)
{
if (*lista == NULL)
   InserisciTestaLista(lista, elem);
else
    if ((*lista) -> info >= elem)
        InserisciTestaLista(lista, elem);
    else
        InserzioneOrdinata(&((*lista)->next), elem);
}
```