Programmazione in linguaggio C

Stefano Lodi

28 febbraio 2007

Algoritmi, linguaggi, programmi, processi

trasformazione di un insieme di dati iniziali in un insieme di risultati finali mediante istruzioni	algoritmo
strumento o formalismo per rappresentare le istruzioni di un algoritmo e la loro concatenazione	linguaggio di programmazione
algoritmo scritto in un linguaggio di programmazione al fine di comunicare al calcolatore elettronico le azioni da intraprendere	programma
programma in esecuzione su un calcolatore	processo

Struttura di un programma C

- Un programma C esegue *operazioni* in successione su un insieme di *variabili*
- Dichiarazioni di variabili e descrizione delle operazioni compaiono all'interno di funzioni
- Un programma C comprende una o piú funzioni, delle quali una è sempre la funzione principale (*main*)
- La struttura di un programma C è la seguente

```
main() {
  <dichiarazioni di dati>
  <istruzioni>
}
```

Programma sorgente

- Programma sorgente $\Longleftrightarrow$ sequenza di caratteri che soddisfa determinate regole
- Si distinguono regole lessicali e regole sintattiche
- Lessico $\Longleftrightarrow$ insieme di vocaboli utilizzabili per la costruzione del programma
- Il lessico fornisce i "mattoni da costruzione" da combinare secondo regole sintattiche per formare programmi formalmente corretti
- Solo i programmi formalmente corretti sono accettati dal compilatore

Lessico del C

parole riservate	significato speciale; non utilizzabili in modi diversi da quanto previsto, es. `main`, `while`, `for`
identificatori	nomi univoci attribuiti agli oggetti con cui il programma opera
costanti	valori numerici o sequenze di caratteri da utilizzare nelle operazioni
simboli speciali	caratteri non alfanumerici usati per comporre i vari oggetti del programma, es. ; [ ] ( ) ==
separatori	caratteri che determinano la fine di una parola riservata, di un identificatore o di una costante; comprendono i simboli speciali, lo spazio bianco, il fine linea

Tipi di dato

- I valori manipolabili da un programma si raggruppano in alcune categorie, con operazioni specifiche della categoria
  - interi, reali, sequenze di caratteri
- Un tipo di dato è definito da
  - dominio di valori D
  - funzioni $f_1, \ldots, f_n$ e predicati $p_1, \ldots, p_m$ *definiti sul dominio* (operatori)
  - costanti $c_1, \ldots, c_q$
- Se $v_1, \ldots, v_{k_i} \in D$ allora $f_i(v_1, \ldots, v_{k_i}) \in D$, dove k_i è il numero di argomenti di f_i (per $i = 1, \ldots, n$).
- Se $v_1, \ldots, v_{k_i} \in D$ allora $p_i(v_1, \ldots, v_{k_i})$ è vero oppure falso, dove k_i è il numero di argomenti di p_i (per $i = 1, \ldots, m$).

Tipi di dato

- rappresentazione di un tipo di dato in un linguaggio $\Longleftrightarrow$ descrizione con strutture linguistiche per *definirlo* e *manipolarlo*
- Esistono
  - Tipi semplici e strutturati
    - Nei tipi strutturati il dominio è un prodotto cartesiano di domini
  - Tipi predefiniti e definiti dal programmatore
- Qualche convenzione terminologica
  - Dati $\Longleftrightarrow$ sia valori che variabili
  - Operatori $\Longleftrightarrow$ simboli di predicato o funzione
  - Operandi $\Longleftrightarrow$ dati a cui sono applicati operatori

Il tipo `int`

- Numeri interi tra un minimo e un massimo
- Può essere qualificato come
  - `signed` oppure `unsigned`, rappresentando cosí interi con segno o senza segno, e
  - `short` oppure `long`, per rappresentare due differenti campi di variazione corrispondenti all'impiego di 16 bit o 32 bit per la rappresentazione
  - La dimensione di `int` senza qualificatori non è fissa e varia a seconda del compilatore (16 o 32 bit)

Il tipo `int`

Tipo	abbreviazione	campo di valori
signed short int	short int	da -32768 a 32768
signed int	int	dipende dal compilatore
signed long int	long int	da -2147483648 a 2147483648
unsigned short int		da 0 a 65535
unsigned int		dipende dal compilatore
unsigned long int		da 0 a 4294967295

Il tipo char

- L'insieme dei caratteri disponibili per la comunicazione
  - A rigore, dipende dal compilatore
  - In pratica, oggi la conformità allo standard ASCII è assai comune
- 256 caratteri (numerati da 0 a 255) di cui
  - da 0 a 31 non stampabili, codici di controllo
  - da 32 a 127 set di caratteri standard
  - da 128 a 255 nazionali
- Subsequenze notevoli
  - da 48 a 57, cifre, ordine crescente di significato
  - da 65 a 90, maiuscole, in ordine alfabetico
  - da 97 a 122, minuscole, in ordine alfabetico
- $\longrightarrow$ proprietà utili per le conversioni

I tipi `float`, `double`

- Per le applicazioni numeriche è indispensabile un tipo decimale
- Si utilizzano i numeri in virgola mobile (*floating point*), con un numero finito prefissato di cifre $\rightarrow$ sottoinsieme dei razionali $\mathbb{Q}$
- `float`, singola precisione
- `double`, doppia precisione
- `long double`, quadrupla precisione

Tipo	numero di byte	campo di valori	cifre signif.
`float`	4 byte	$3.4 \times 10^{-38} \div 3.4 \times 10^{38}$	6
`double`	8 byte	$1.7 \times 10^{-308} \div 1.7 \times 10^{308}$	15
`long double`	16 byte	$1.1 \times 10^{-4932} \div 1.1 \times 10^{4932}$	19

Costanti

- Costante $\Longleftrightarrow$ dato che non può cambiare valore per tutta la durata del programma
- Costante intera: `123`
- Costante reale: `3.14`, `314E-2`, `0.314E1`
- Costante di tipo carattere: un singolo carattere scritto tra apici, es. `'a'`
  - Alla costante si associa il valore nell'insieme di caratteri considerato, (es. in ASCII `'a'` $\rightarrow 65$)
  - Alcuni caratteri non stampabili si rappresentano con un carattere preceduto da barra (*escape sequence*); ad esempio, a capo si rappresenta con `'\n'`, tabulazione con `'\t'`, a inizio riga con `'\r'`
- Costante di tipo stringa: zero o piú caratteri tra doppi apici: `""`, `"ciao"`, `"continuare?  (digitare \"si\"per continuare)"`

Dichiarazioni di costanti

- Associa un identificatore a una stringa di caratteri
- Sintassi

$$\texttt{\#define}\ \langle\text{identif}\rangle\ \langle\text{stringa-di-caratteri}\rangle$$

- Tutte le occorrenze di $\langle\text{identif}\rangle$ sono sostituite con $\langle\text{stringa-di-caratteri}\rangle$
- ```
  #define Pi 3.1415926
  #define StringaVuota ""
  ```
- `#define` non è una instruzione ma una direttiva, elaborata dal preprocessore
  ```

Variabili

- Variabile $\Longleftrightarrow$ dato che può essere usato e modificato dal programma
- La dichiarazione di variabile
  - associa un identificatore ad un tipo
  - determina l'allocazione di un'area di memoria adatta a contenere valori del tipo associato
- Nella dichiarazione è possibile specificare il valore che deve essere assunto dalla variabile all'inizio dell'esecuzione del programma (valore di inizializzazione)

Dichiarazione di variabile

- Sintassi semplificata

$$\langle\text{dich-variabile}\rangle \longrightarrow \langle\text{nome-tipo}\rangle\,\langle\text{lista-variabili}\rangle$$

$$\langle\text{lista-variabili}\rangle \longrightarrow \langle\text{variabile}\rangle \mid \langle\text{variabile}\rangle\ \texttt{,}\ \langle\text{lista-variabili}\rangle$$

$$\langle\text{variabile}\rangle \longrightarrow \langle\text{identif}\rangle \mid \langle\text{identif}\rangle = \langle\text{espr}\rangle$$

-
```
int X=0;  /* X è inizializzato a 0 */
char C,K; /* equiv. a char C; char K; */
```

Operatori aritmetici

- Gli operatori aritmetici in C:

operatore	tipo	significato
-	unario	cambio segno
+	binario	addizione
-	binario	sottrazione
*	binario	moltiplicazione
/	binario	divisione tra interi o reali
%	binario	modulo (tra interi)

- / è un simbolo unico per la divisione reale e intera
  - se `x`, `y` sono entrambi interi, `x/y` è la divisione tra interi
  - altrimenti è la divisione reale
- Operatore modulo: `A%B` = `A - (A/B)*B`

Assegnamento

- Semplificando, l'assegnamento permette di calcolare il valore di una espressione e attribuire tale valore a una variabile
- L'assegnamento causa la scrittura del valore nell'area di memoria associata alla variabile, distruggendo il precedente valore
- Sintassi semplificata

$$\langle \text{assegnamento} \rangle \longrightarrow \langle \text{nome-var} \rangle \texttt{=} \langle \text{espr} \rangle$$

-
```
int X,Y;
X = 0;
Y = X + 1;
Y = Y + 1;
```

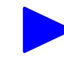
Assegnamento

- Piú generalmente, l'assegnamento permette di specificare
  - a destra dell'uguale un'espressione, detta rvalue, da valutare,
  - a sinistra dell'uguale una espressione, detta lvalue, utilizzata per indirizzare l'area di memoria in cui scrivere il valore risultato della valutazione della espressione sulla destra
- Sintassi

$$\langle \text{assegnamento} \rangle \longrightarrow \langle \text{lvalue} \rangle \; \texttt{=} \; \langle \text{rvalue} \rangle$$

- Una variabile può essere sia $\langle \text{lvalue} \rangle$ che $\langle \text{rvalue} \rangle$
- Una costante può essere solo $\langle \text{rvalue} \rangle$

Assegnamento



- L’assegnamento è una particolare espressione!
  - Il valore dell’assegnamento è il valore ottenuto dalla valutazione della parte destra
  - L’effetto dell’assegnamento è la scrittura del valore nell’area di memoria denotata dalla parte sinistra
- Pertanto è lecito scrivere

```
4 - (X = 1)
```

espressione che vale 3, in quanto l’assegnamento `X = 1` ha valore 1

Assegnamento

- Il linguaggio C fornisce come operatori comode abbreviazioni per alcuni assegnamenti notevoli
- `++` operatore di incremento, `--` operatore di decremento:
  - forma prefissa: `++X`, `--X`
  - forma postfissa: `X++`, `X--`
  - `++X`, `X++` hanno su `X` lo stesso effetto di `X = X + 1`
  - `--X`, `X--` hanno su `X` lo stesso effetto effetto di `X = X - 1`
  - Se compaiono in una espressione, vengono valutati diversamente
    - la forma prefissa modifica `X` prima dell'utilizzo del valore di `X` nel calcolo dell'espressione
    - la forma postfissa modifica `X` dopo l'utilizzo del valore di `X` nel calcolo dell'espressione

Assegnamento

- Esempi

```
X = 5;
Y = X++;     /* X vale 6 , Y vale 5 */
Y = ++X;     /* X vale 7, Y vale 7 */

Z = 5;
Y = 10+ ++Z /* Z vale 6, Y vale 16 */
```

- Operatori +=, -=, *=, /=, %=

```
Y += Z;     /* Y vale 21 */
```

Espressioni

- Espressione $\iff$ regola per il calcolo di un valore
- Si compone di
  - Operandi
    - valori costanti
    - valori correnti di variabili
    - risultati di funzioni
    - ...
  - Operatori
- Un'espressione si valuta secondo le regole di precedenza degli operatori
- A parità di precedenza, la valutazione procede da sinistra a destra
- Le parentesi alterano la precedenza come consueto

Espressioni

- Precedenza e associatività per alcuni operatori

Operatori	Categoria	Associatività	Precedenza
`* / %`	prodotto e divisione	$\leftarrow$	alta
`+ -`	somma e sottrazione	$\leftarrow$	
`= +=` $\cdots$	assegnamento	$\rightarrow$	bassa

- Tutti gli operandi hanno un tipo e gli operatori richiedono specifici tipi e restituiscono tipi determinati $\rightarrow$ il compilatore è sempre in grado di stabilire il tipo di una espressione

- Si noti la associatività dell'assegnamento a destra: sono cosí possibili *assegnamenti multipli* quali

```
Y = X = 3;
```

il cui effetto è di porre sia `X` che `Y` a 3

Espressioni

- Sintassi delle espressioni (parziale)

$$
\begin{aligned}
\langle\text{espr}\rangle &\longrightarrow \langle\text{costante}\rangle \mid \langle\text{nome-var}\rangle \mid \texttt{(}\ \langle\text{espr}\rangle\ \texttt{)} \mid \\
&\quad\ \langle\text{espr-assegn}\rangle \mid \langle\text{espr-incr-decr}\rangle \\
\langle\text{espr-assegn}\rangle &\longrightarrow \langle\text{nome-var}\rangle\ \langle\text{oper-assegn}\rangle\ \langle\text{espr}\rangle \\
\langle\text{oper-assegn}\rangle &\longrightarrow \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=} \\
\langle\text{espr-incr-decr}\rangle &\longrightarrow \texttt{++}\ \langle\text{nome-var}\rangle \mid \texttt{--}\ \langle\text{nome-var}\rangle \mid \\
&\quad\ \langle\text{nome-var}\rangle\ \texttt{++} \mid \langle\text{nome-var}\rangle\ \texttt{--}
\end{aligned}
$$

Input e Output

- C non ha istruzioni predefininte per input e output
- Esiste tuttavia una Libreria Standard di funzioni, definite in `stdio.h`
- Un programma che svolge input e output deve includere la *direttiva*

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

- Funzioni per input/output di caratteri, linee, formattato
- Principali funzioni
  - `printf`
  - `scanf`

Output

- Semplice programma che visualizza dati in output

```
#include <stdio.h> /*include la libreria standard */
main () {
  int base, altezza, area;

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

  printf("L’area è: %d",area);
}
```

- la funzione `printf` ha argomenti (anche detti *parametri*) di due tipi
  - il primo è una costante stringa, la stringa di formato
  - i successivi, se presenti, sono espressioni il cui valore viene visualizzato

Output

- La stringa di formato contiene
  - Testo da visualizzare (`L'area è: `)
  - numero e tipo dei dati da visualizzare (`%d` → decimale)
- Esempi
  - `printf("%d %d %d", base, altezza, area)`
    ```
    3 4 12
    ```
  - `printf("%4d%4d%6d", base, altezza, area)`
    ```
       3   4    12
    ```
  - `printf("%-5d%-5d%-10d", base, altezza, area)`
    ```
    3    4    12
    ```
  - con sequenze di escape
    `printf("%d\n%d\n%d", base, altezza, area)`
    ```
    3
    4
    12
    ```

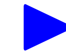
Output

- `printf`: *converte*, *formatta*, *stampa* i suoi argomenti sotto il controllo delle indicazioni della stringa di formato
- La stringa di formato contiene due tipi di caratteri
  - ordinari: copiati sull'output
  - specifiche di conversione: ognuna provoca la conversione e stampa del successivo argomento

carattere	tipo valore	stampato come
`d`	`int`	numero dec.
`c`	`int`	carattere singolo
`s`	`char *`	stampa tutta la stringa
`f`	`double`	$x \ldots x.dddddd$
`e`,`E`	`double`	$x \ldots x.dddddd\mathtt{e}\pm zz$, $x \ldots x.dddddd\mathtt{E}\pm zz$

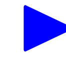
Output

- Tra il % e il carattere di conversione:
  - `-` $\mapsto$ allineamento a sinistra
  - numero $\mapsto$ ampiezza minima del campo di output
  - punto $\mapsto$ separatore tra ampiezza e precisione
  - numero $\mapsto$ precisione

Input 

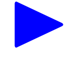
- Funzione `scanf`: *legge* caratteri da input, li *interpreta* secondo quanto indicato dalla stringa di formato, *memorizza il risultato* nelle variabili riportate come argomenti
- `scanf("%d", &base);`

  acquisisce un intero decimale e memorizza nella variabile `base`
- primo argomento: stringa di formato, contiene solo specifiche di conversione
- successivi argomenti: nomi di variabili a cui assegnare i valori acquisiti
- I nomi delle variabili devono essere preceduti dal simbolo `&` (estrae l'indirizzo della variabile consecutiva)

Input 

- Stringa di formato
  - Spazi e tabulazioni sono ignorati
  - i caratteri normali (non %) devono corrispondere al successivo carattere non bianco in input
  - Le specifiche di conversione riflettono quelle della funzione `printf`
- Conversione per le variabili numeriche
  - gli spazi bianchi precedenti le cifre o il punto decimale sono ignorati
  - le cifre sono accettate fino al carattere di terminazione: `spazio` o `invio`

Input



- Tra `%` e il carattere di conversione possono essere presenti nell'ordine:

l'ampiezza massima del campo: un intero decimale

- se non è presente si assume ampiezza infinita
- altrimenti, `scanf` legge un numero massimo di caratteri pari all'intero specificato
  - lo spazio bianco non influisce sul conteggio dei caratteri da leggere

caratteri flag:

- `l` (elle) tra `%` e il carattere di conversione indica che il carattere di conversione sarà `f`, `e`, o `E` e l'argomento corrispondente sarà `double` e non `float`
- `h` tra `%` e il carattere di conversione indica che il carattere di conversione sarà `d` e l'argomento corrispondente sarà `short int` e non `int`

Memoria di transito della tastiera ▶

- I caratteri digitati alla tastiera sono memorizzati in un'area di memoria temporanea, la memoria di transito, o buffer, di tastiera
  - Il buffer di tastiera si trova sotto il completo controllo del sistema operativo
- L'esecuzione della funzione `scanf` legge i caratteri da convertire dal buffer di tastiera; se il buffer è vuoto, attende che vengano premuti tasti seguiti da un invio
  - Il numero di caratteri letti in presenza di una data sequenza di caratteri dipende dalla stringa di formato
  - ogni carattere letto è "consumato", nel senso che i caratteri sono letti nella successione in cui si trovano nel buffer (la stessa con cui sono stati digitati)
- Esiste una finestra di lettura che indica la posizione nel buffer del prossimo carattere che viene passato a `scanf`

Memoria di transito della tastiera

Esempio.
```
main() {
    char Ch;
    scanf("%c", &Ch);
}
```

- Inizialmente il buffer è vuoto e `scanf` si arresta per mancanza di caratteri da convertire
- L'utente digita `abc` e i caratteri sono memorizzati nel buffer nell'ordine in cui sono digitati
- Ora la finestra è sul primo carattere `a`
- L'utente digita $\hookleftarrow$ e il primo carattere `a` è consumato da `scanf`; il carattere `'a'` è memorizzato nella variabile `Ch`
- Per effetto della lettura, la finestra avanza al carattere successivo
- Ogni successiva lettura riceverà `b` come primo carattere letto

Memoria di transito della tastiera

istruzione	tastiera	stato di buffer e variabili
`scanf("%c",&Ch)`		buffer: [ ] — Ch: [ ] — finestra di lettura
	abc↵	buffer: [a b c ↵] — Ch: [a] — finestra di lettura (su a)
		buffer: [a b c ↵] — Ch: [a] — finestra di lettura (su b)

Memoria di transito della tastiera ▶

Osservazione (Lettura di un singolo carattere).

```
#include <stdio.h>
main () {
  int I;
  char C;
  printf("Inserire intero e premere enter ->");
  scanf("%d",&I);
  printf("Inserire carattere e premere enter ->");
  scanf("%c" ,&C);
  printf("\nIntero %5d Carattere %c (ASCII %d)",I,C,C);
}
```

- Input: 123↩
- Output:

  ```
  Intero   123 Carattere
   (ASCII 10)
  ```
- ?

Istruzioni semplici e composte

- Una *istruzione semplice* è un'espressione seguita da ;
- Una *istruzione composta* è una sequenza di istruzioni (non necessariamente semplici) racchiuse tra { e }
  - Le istruzioni componenti la sequenza vengono eseguite nell'ordine di comparizione nella sequenza;
  - Ogni istruzione viene eseguita solo dopo il termine della esecuzione della precedente
  - La sequenza vuota {} è ammessa
  - In generale, in C il codice può essere scritto in formato libero. L'incolonnamento aiuta la lettura, e, nella pratica, è sempre utilizzato

$\texttt{\{}\ \langle\text{istr}\rangle_1\texttt{;}\ \langle\text{istr}\rangle_2\texttt{;}\ \langle\text{istr}\rangle_3\texttt{;}\ \texttt{\}}$

```
{
   ⟨istr⟩₁;
   ⟨istr⟩₂;
   ⟨istr⟩₃;
}
```

Istruzioni semplici e composte: scambio del valore di due variabili

- **Esempio.** Date due variabili `x`, `y` dello stesso tipo, scrivere una istruzione che ne scambi il valore.
- Due assegnamenti simmetrici non risolvono il problema, indipendentemente dal loro ordine!

Istruzione	x	y	Istruzione	x	y
	4	7		4	7
`x=y;`			`y=x;`		
	7	7		4	4
`y=x;`			`x=y;`		
	7	7		4	4

Istruzioni semplici e composte: scambio del valore di due variabili

- Si ricorre ad una variabile di utilizzo temporaneo per memorizzare il valore della prima variabile che viene assegnata, che altrimenti andrebbe perso

Istruzione	`temp`	`x`	`y`
	?	4	7
`temp=x;`			
	4	4	7
`x=y;`			
	4	7	7
`y=temp;`			
	4	7	4

Blocco

- Generalizza l'istruzione composta permettendo l'inserimento di dichiarazioni prima delle istruzioni componenti

$$\langle\text{blocco}\rangle \longrightarrow \texttt{\{}\,\langle\text{dichiarazioni}\rangle\,\langle\text{istruzioni}\rangle\,\texttt{\}}$$

```
{ int temp;
  temp=x;
  x=y;
  y=x;
}
```

- Un blocco è una istruzione; pertanto può essere contenuto (*nidificato*) in altri blocchi

Istruzioni ripetitive

- Costituite da
  - Una *istruzione da ripetere*, o *corpo del ciclo*
  - Una *logica di controllo* della ripetizione
- L'istruzione da ripetere può in particolare essere una istruzione composta
- La logica di controllo consiste nella valutazione di una espressione
  - La prima valutazione dell'espressione può avvenire inizialmente o dopo la prima ripetizione
  - Se il valore dell'espressione risulta diverso da zero, la ripetizione prosegue, diversamente il controllo passa alla prima istruzione seguente l'istruzione ripetitiva

Inizializzazione di una variabile ▶

- L'esecuzione di una istruzione ripetitiva comporta spesso l'utilizzo di una variabile sia come parte del $\langle \text{rvalue} \rangle$ che come $\langle \text{lvalue} \rangle$ di un assegnamento

- **Esempio.** Somma di n numeri

  Algoritmo.

  ```
  - finché sono presenti elementi in input
    - leggi un elemento
    - aggiungi elemento alla variabile di accumulo
  ```

- Se `s` è la variabile di accumulo e `x` l'elemento letto, l'istruzione da ripetere sarà quindi `s = s + x;`

Inizializzazione di una variabile

- Alla prima ripetizione `s` non ha un valore definito
- È necessario che essa abbia ricevuto un valore prima dell'inizio della ripetizione
- L'assegnamento del valore iniziale si dice inizializzazione
- Quale valore deve essere assegnato? Dipende dal problema, ma una regola empirica è

  > che valore deve assumere la variabile se la ripetizione avviene zero volte?

-

  Esempio. somma di n numeri: se $n = 0$ il risultato deve essere 0,quindi la corretta inizializzazione è

$$s = 0$$

Operatori relazionali

- Gli operatori relazionali permettono il confronto tra valori
- Il loro risultato è vero o falso

Operatore	Significato
==	uguale
!=	diverso
>	maggiore
>=	maggiore o uguale
<	minore
<=	minore o uguale

Valori e operatori logici

- Algebra di Boole
  - Valori vero, falso
  - operatori logici and, or, not
- Funzionamento (tabella di verità)

P	Q	P and Q	P or Q	not P
falso	falso	falso	falso	vero
falso	vero	falso	vero	vero
vero	falso	falso	vero	falso
vero	vero	vero	vero	falso

Valori e operatori logici

- In C non esiste un tipo predefinito per i valori logici
- Si rappresentano con valori interi con la codifica:
  - il valore 0 denota falso
  - il valore 1 denota vero (ogni altro valore diverso da 0 denota vero)
- Operatori logici in C

simbolo	operatore logico
!	not
&&	and
||	or

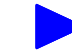
P	Q	P && Q	P || Q	!P
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Espressioni

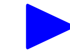
- Rivisitiamo la sintassi delle espressioni, aggiungendo espressioni relazionali e logiche

$$
\begin{array}{rcl}
\langle \text{espr} \rangle & \longrightarrow & \langle \text{costante} \rangle \mid \langle \text{nome-var} \rangle \mid \texttt{(} \, \langle \text{espr} \rangle \, \texttt{)} \mid \langle \text{espr-relazionale} \rangle \mid \\
 & & \langle \text{espr-assegn} \rangle \mid \langle \text{espr-logica} \rangle \mid \langle \text{espr-incr-decr} \rangle \\
\langle \text{espr-relazionale} \rangle & \longrightarrow & \langle \text{espr} \rangle \; \langle \text{oper-relazionale} \rangle \; \langle \text{espr} \rangle \\
\langle \text{oper-relazionale} \rangle & \longrightarrow & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\langle \text{espr-assegn} \rangle & \longrightarrow & \langle \text{nome-var} \rangle \; \langle \text{oper-assegn} \rangle \; \langle \text{espr} \rangle \\
\langle \text{oper-assegn} \rangle & \longrightarrow & \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=} \\
\langle \text{espr-logica} \rangle & \longrightarrow & \langle \text{espr} \rangle \; \langle \text{oper-logico} \rangle \; \langle \text{espr} \rangle \\
\langle \text{oper-logico} \rangle & \longrightarrow & \texttt{!} \mid \texttt{\&\&} \mid \texttt{||} \\
\langle \text{espr-incr-decr} \rangle & \longrightarrow & \texttt{++} \, \langle \text{nome-var} \rangle \mid \texttt{--} \, \langle \text{nome-var} \rangle \mid \\
 & & \langle \text{nome-var} \rangle \, \texttt{++} \mid \langle \text{nome-var} \rangle \, \texttt{--}
\end{array}
$$

Istruzione `while`



- Esegue ripetutamente quanto segue: Calcola il valore della espressione, se è diverso da zero, esegue il blocco; altrimenti passa il controllo all'istruzione successiva
- Sintassi: $\langle$istr-while$\rangle \longrightarrow$ `while (` $\langle$espr$\rangle$ `)` $\langle$istr$\rangle$
- In totale si potranno avere $n+1$ valutazioni (test) e n ripetizioni, con $n \geq 0$
- caso particolare: il test fallisce alla prima valutazione $\rightarrow$ non si ha alcuna ripetizione
- In presenza di istruzioni di ripetizione, un programma può non terminare
- perché una istruzione ripetitiva possa terminare è necessario che l'istruzione ripetuta modifichi almeno una delle variabili coinvolte nella espressione di controllo.

Istruzione `while` 

Esempio. Calcolare la divisione fra due numeri interi non negativi usando solo somme algebriche e sottrazioni

- Si sottrae di volta in volta il divisore dal dividendo, si incrementa un contatore del numero di sottrazioni effettuate
- Le sottrazioni termineranno quando il dividendo residuo sarà più piccolo del divisore e il dividendo residuo sarà il resto della divisione
- Istruzione da ripetere

```
Resto = Resto - Divisore;
Quoziente = Quoziente + 1;
```

- Condizione per eseguire la ripetizione

```
Resto >= Divisore
```

Istruzione `while`

- Entrambi `Resto` e `Quoziente` compaiono a destra negli assegnamenti → vanno inizializzati
- La ripetizione avverrà zero volte se il divisore è maggiore del dividendo in questo caso deve risultare il resto uguale al dividendo e il quoziente zero
- Inizializzazione

```
Resto = Dividendo;
Quoziente = 0;
```

- Vediamo il programma completo...

Istruzione `while`

- Si consiglia di progettare sempre prima un algoritmo in linguaggio naturale e poi tradurlo nel linguaggio di programmazione scelto

  Algoritmo.

```
- acquisisci gli operandi
- inizializza resto e quoziente
- finchè il resto è non inferiore al divisore
  - incrementa il quoziente
  - sottrai divisore da resto
- visualizza quoziente e resto
```

Istruzione `while` ▶

```
- acquisisci gli operandi



- inizial. resto e quoziente

- finchè resto >= divisore
  - incrementa il quoziente
  - sottrai divisore da resto

- visualizza quoziente e resto
```

```
#include <stdio.h>
main() {
  int Dividendo, Divisore,
      Quoziente, Resto;
  printf("Dividendo ? ");
  scanf("%d", &Dividendo) ;
  printf("\nDivisore ?" );
  scanf("%d", &Divisore) ;
  Resto = Dividendo;
  Quoziente = 0;
  while (Resto >= Divisore) {
    Quoziente = Quoziente + 1;
    Resto = Resto - Divisore;
  };
  printf ("\n%d, %d",Quoziente,Resto) ;
}
```

Istruzione `do...while`

- Ripete quanto segue: esegue il blocco; calcola il valore della espressione, e, se è zero, passa il controllo all'istruzione successiva
- Sintassi: ⟨istr-do-while⟩ ⟶ `do` ⟨istr⟩ `while` `(` ⟨espr⟩ `)`
- Differisce da `while` in quanto la valutazione dell'espressione viene fatta dopo la esecuzione del blocco; pertanto il blocco viene eseguito almeno una volta
- Si potranno quindi avere n test e n ripetizioni, con $n \geq 1$.
- Ogni `do...while` si può riscrivere con `while`; tuttavia se l'istruzione deve essere eseguita almeno una volta perché l'espressione sia definita, è piú leggibile `do...while`

Istruzione `do...while`

Esempio. Leggere da input caratteri finchè non compare un punto

- si richiede che vengano letti da input n caratteri ed eseguiti n test, di cui $n-1$ falliscono (i caratteri diversi da punto) e uno ha successo (il punto)

```
do scanf("%c",&Ch) while (Ch != '.') ;
```

Utilizzando `while` il test compare prima della prima lettura

è necessario un assegnamento fittizio a `Ch` per garantire l'entrata nel ciclo almeno una volta

```
Ch = '#';
while (Ch != '.') scanf ("%c" ,&Ch) ;
```

Istruzione `do...while`

- **Esempio.** Conta il numero di parole della frase digitata in input. La frase inizia con un carattere diverso dallo spazio ed è terminata da un punto.

Pseudocodice	Codice C
	`#include <stdio.h>`
	`main() {`
	`char C;`
	`int n=0;`
- esegui	`do {`
- esegui	`do`
- acquisisci C	`C = getchar();`
- finché C non è ' ' o '.'	`while (C != ' ' && C != '.');`
- incrementa il numero di parole	`n++;`
- finché C è ' '	`while (C == ' ')`
- acquisisci C	`C =getchar();`
- finché il carattere non è '.'	`} while (C != '.');`
- visualizza il risultato	`printf("%d",n);`
	`}`

```
#include <stdio.h>
main() {
  char C;
  int n=0;
  do {
    do
      C = getchar();
    while (C != ' ' && C != '.');
    n++;
    while (C == ' ')
      C =getchar();
  } while (C != '.');
  printf("%d",n);
}
```

Esempi comparati `while`, `do...while`

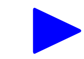
Problema. Sommare una sequenza di numeri interi positivi letti da input

- È necessaria una variabile `Somma` per memorizzare il risultato, inizializzata a zero
- continuare a memorizzare un numero acquisito da input quando è stato già sommato è inutile → una sola variabile `I` per contenere i numeri acquisiti da input è sufficiente
- Su `I` si riscriverà di volta in volta il nuovo numero dopo aver effettuato la somma del precedente

Esempi comparati `while`, `do...while`

- È conveniente costruire un programma interattivo $\Longleftrightarrow$ un programma che interagisce con l'utente,
  - sollecitandolo all'introduzione di dati al momento opportuno
  - visualizzando risultati
- Occorre visualizzare messaggi di output che sollecitano l'utente a digitare i numeri
- Corpo del ciclo

```
- visualizzazione di un messaggio per sollecitare
  l'introduzione di un numero
- acquisizione di un numero in I
- somma di I a Somma
```

Esempi comparati `while`, `do...while` 

Soluzione. Supponiamo non nota la lunghezza della sequenza

- Come conveniamo di indicare la terminazione? Per esempio, con un numero non positivo (è ammessa una sequenza di zero numeri positivi)

- ad ogni iterazione

  - un test sul numero letto, se è positivo va sommato, altrimenti le iterazioni vengono arrestate

  - Se la sequenza è di n numeri positivi più un numero negativo per la terminazione, si avranno $n+1$ messaggi, $n+1$ acquisizioni, $n+1$ test, l'ultimo test fallisce:

    messaggio, lettura, test,

    somma, messaggio, lettura, test,

    ..., somma, messaggio, lettura, test

Esempi comparati `while`, `do...while` ▶

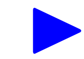
- adottiamo come corpo del ciclo

  somma,messaggio,lettura

- Si ricava l'algoritmo seguente

Algoritmo. - *inizializza Somma a zero*
- *visualizza messaggio*
- *acquisisci un numero I*
- *fino a quando I è maggiore di zero*
  - *aggiungi I a Somma*
  - *visualizza messaggio*
  - *acquisisci un numero I*
- *visualizza il risultato*

Esercizio. Implementare l'algoritmo in C

Esempi comparati `while`, `do...while` 

Programma.

```
                       #include <stdio.h>
                       main() {
- iniz. Somma a 0        int I, Somma=0;
- visual.messaggio       printf("Intero in input\n");
                         printf("(negativo o zero per terminare) \n");
- acq. numero I          scanf("%d", &I);
- fino a quando I>0      while (I > 0) {
  - agg. I a Somma         Somma += I;
  - vis. messaggio         printf("Intero in input \n");
                           printf("(negativo o zero per terminare) \n");
  - acq. numero I          scanf("%d", &I);
                         }
- vis. risultato         printf("\n Somma dei numeri positivi: %d",
                                Somma);
                       }
```

Esempi comparati `while`, `do...while` ▶

- Supponiamo ora che
  - si vogliano sommare numeri qualunque
  - sia garantito almeno un numero valido
- Dopo ogni acquisizione si chiede all'utente se vuole continuare
  - $\Longrightarrow$ occorre una variabile di tipo carattere `ch` che assume valori ricevuti in input `'S'` e `'N'`
  - È garantito almeno un numero valido $\Longrightarrow$ la sequenza di operazioni è

    `messaggio, lettura, somma, domanda, test,`
    `...messaggio, lettura, somma, domanda, test`
  - n messaggi, n letture, n domande, n test
  - corpo del ciclo

    `messaggio, lettura, somma, domanda`

Esempi comparati `while`, `do...while` □

Algoritmo.

```
- inizializza Somma a zero
- finchè la risposta è "continuazione"
  - visualizza messaggio
  - acquisisci un numero I
  - aggiungi I a Somma
  - domanda se continuare
- visualizza il risultato
```

- si domanda se continuare solo all'interno del corpo del ciclo $\Longrightarrow$ l'algoritmo si presta ad essere implementato con `do...while`

Esempi comparati `while`, `do...while` ▶

Programma.

```
#include <stdio.h>
main() {
  int I, Somma=0;
  char Ch;
  do {
    printf("Intero in input \n");
    scanf("%d", &I);
    Somma+=I;
    printf("Continua (S/N) ? \n");
    scanf(" %c", &Ch);
  } while (Ch !='N');
  printf("\n Somma dei numeri positivi: %d", Somma);
}
```

Istruzione `for`

- Sintassi dell'istruzione `for`:

$$\langle\text{istr-for}\rangle \longrightarrow \texttt{for}\ \texttt{(}\ \langle\text{espr}\rangle\ ;\ \langle\text{espr}\rangle\ ;\ \langle\text{espr}\rangle\ \texttt{)}\ \langle\text{istr}\rangle$$

- Il significato dell'istruzione `for`:

```
for (espressione1; espressione2; espressione3)
  istruzione
```

è riconducibile alla seguente istruzione `while`:

```
espressione1;
while (espressione2) {
  istruzione
  espressione3;
}
```

Istruzione `for`

- Per ora ci limitiamo ad un utilizzo particolare dell'istruzione `for`, volta al controllo del numero di ripetizioni
- In tale forma limitata
  - una variabile di controllo assume in successione tutti i valori compresi tra un minimo e un massimo
  - *espressione1* assegna di un valore iniziale alla variabile di conteggio
  - *espressione2* esprime il valore finale della variabile di conteggio
  - *espressione3* incrementa o decrementa la variabile di conteggio

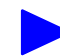
Esempio. Visualizza i cubi degli interi da 1 a 10

```
for (J = 1; J<= 10; J++)
  printf("%d\n",J*J*J);
```

Istruzione `for`

- Se il valore della variabile di controllo non è modificato esplicitamente con istruzioni di assegnamento nel corpo del ciclo allora
  - la variabile di controllo assume tutti i valori compresi fra l'espressione iniziale e quella finale
  - al termine delle ripetizioni il valore della variabile di controllo contiene il valore successivo o precedente al valore finale

Ripetizioni nidificate



- Una ripetizione nidificata è un'istruzione contenuta in una istruzione di ripetizione che è a sua volta una istruzione di ripetizione

  Esempio. Per ogni numero fornito da input, calcolare una potenza letta da input (una sola volta, in anticipo)

  Algoritmo.
  ```
  - acquisisci l'esponente E
  - visualizza messaggio
  - acquisisci un numero B
  - fino a quando B è diverso da zero
    - calcola la potenza B elevato a E
    - acquisisci un numero B
    - visualizza il risultato
  ```

- Il passo in cui si calcola la potenza è una ulteriore ripetizione, in cui il numero di ripetizioni è noto a priori

Ripetizioni nidificate

- Consideriamo allora il sottoproblema

  Problema. Elevare B alla potenza E

  Algoritmo.
  ```
  - inizializza P a uno
  - per tutti i valori di I da 1 a E
    - moltiplica P per B
  ```

- Nidificando il secondo algoritmo nel primo

  Algoritmo.
  ```
  - acquisisci l'esponente E
  - visualizza messaggio
  - acquisisci un numero B
  - fino a quando B è diverso da zero
    - inizializza P a uno
    - per tutti i valori di I da 1 a E
      - moltiplica P per B
    - acquisisci un numero B
    - visualizza il risultato
  ```

Ripetizioni nidificate

- Il programma implementato in C contiene una ripetizione realizzata con `for` all'interno di una ripetizione realizzata con `while`

Programma.

```
#include <stdio.h>
main () {
  int B,P,E,I;
  printf("Inserire l'esponente non negativo > ");
  scanf("%d", &E) ;
  printf("Inserire la base (zero per terminare) ");
  scanf("%d", &B) ;
  while (B!= 0) {
    P=1;
```

(continua)

Ripetizioni nidificate

- *(continua dalla pagina precedente)*

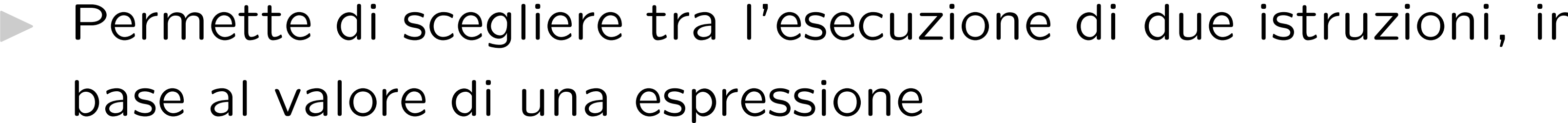
```
      for(I=1;I<=E;I++)
        P=P*B;
      printf("%d elevato a %d = %d\n",B,E,P);
      printf("Inserire la base (zero per terminare) ");
      scanf("%d", &B) ;
    }
  }
```

- L'inserimento del valore zero per `B` determina la fine dell'esecuzione senza che nemmeno una ripetizione sia eseguita

Istruzioni di controllo ▶

- Una istruzione di controllo permette di scegliere fra una o più istruzioni da eseguire, secondo il valore di una espressione
- Le istruzionifra cui si sceglie possono essere semplici o composte
- Caso particolare: è specificata una sola istruzione
  - la scelta è tra eseguire e non eseguire quella istruzione
- In C esistono due istruzioni di scelta
  - `if`
    - scelta binaria
  - `switch`
    - scelta multipla

Istruzione `if`



- Permette di scegliere tra l'esecuzione di due istruzioni, in base al valore di una espressione

- Sintassi:

  $\langle \text{istr-if} \rangle \longrightarrow$ `if (` $\langle \text{espr} \rangle$ `)` $\langle \text{istr} \rangle$ | `if (` $\langle \text{espr} \rangle$ `)` $\langle \text{istr} \rangle$ `else` $\langle \text{istr} \rangle$

  - **prima variante** valuta l'espressione; se non è zero, esegui l'istruzione

    seconda variante valuta l'espressione; se non è zero, esegui la prima istruzione, altrimenti esegui la seconda istruzione

  - In caso di nidificazione di piú istruzioni `if`, ogni eventuale `else` si riferisce sempre alla parola `if` piú vicina

interpretazione errata: `if` $\langle \text{espr} \rangle$ `if` $\langle \text{espr}_1 \rangle$ $\langle \text{istr}_1 \rangle$ `else` $\langle \text{istr}_2 \rangle$

interpretazione corretta: `if` $\langle \text{espr} \rangle$ `if` $\langle \text{espr}_1 \rangle$ $\langle \text{istr}_1 \rangle$ `else` $\langle \text{istr}_2 \rangle$

Istruzione `if`

- Nidificazione di istruzioni `if`

```
if (E1)
  S1
else
  if (E2)
    S2
  else
    S3
```

E1	E2	Istruzione eseguita
vero	non valutata	S1
falso	vero	S2
falso	falso	S3

Istruzione `if`

- Nidificazione di istruzioni `if`

```
if (E1)
  if (E2)
    S1
  else
    S2
```

E1	E2	Istruzione eseguita
vero	vero	S1
vero	falso	S2
falso	non valutata	-

Istruzione `if`

- Nidificazione di istruzioni `if`

```
if (E1) {
  if (E2)
    S1
}
else
  S3
```

E1	E2	Istruzione eseguita
vero	vero	S1
falso	falso	-
falso	non valutata	S3

Istruzione `if`

Esempio. Dati tre interi positivi in ordine non decrescente, trovare il tipo di triangolo (equilatero, isoscele, scaleno) da essi definito

Confronto fra i lati (A,B,C)	Analisi
$A + B < C$	non è un triangolo
$A = B = C$	triangolo equilatero
$A = B$ o $B = C$	triangolo isoscele
$A \neq B \neq C$	triangolo scaleno

Istruzione `if`

- È sufficiente eseguire una serie di test tra coppie di lati per isolare il caso che si è verificato

Algoritmo.

```
- acquisisci la terna di misure A,B,C
- se A+B < C allora non è un triangolo
- altrimenti
  - se A=C allora è equilatero (sono in ordine)
  - altrimenti
    - se A=B o B=C allora è isoscele
    - altrimenti è scaleno
```

Istruzione `if` ▶

Programma.

```
#include <stdio.h>
main () {
  int A, B, C;
  /*Legge e stampa i dati */
  do {
    printf("Lunghezza lati, in ordine non decrescente? ");
    scanf("%d%d%d", &A, &B, &C);
  }
  while( A>B || B>C);
  printf("%d %d %d\n", A, B, C);
```

(continua)

Istruzione `if`

(continua dalla pagina precedente)

```
  /* Esegue l'analisi e stampa i risultati */
  if (A + B < C)
    printf("non e' un triangolo");
  else {
    printf("e' un triangolo ");
    /* determina il tipo di triangolo */
    if (A == C)
      printf("equilatero");
    else
      if ((A == B) || (B == C))
        printf("isoscele");
      else
        printf("scaleno");
  }
}
```

Istruzione `break`

- La prossima istruzione da eseguire diventa la prima istruzione seguente il blocco che contiene `break`
- Spesso utilizzato nel blocco di una ripetizione per determinare un'uscita immediata al verificarsi di una condizione
  - Es. verificare una condizione con al piú 10 tentativi

```
⋮
for (I=0; I<10; I++) {
  scanf ("%d", &K) ;
  if (I==K){
    break;
  }
}
⋮
```

Istruzione `break`

- L'utilizzo di `break` ha l'effetto di nascondere una condizione di controllo all'interno del corpo del ciclo
- La lettura dei programmi risulta piú difficile
- Non se consiglia l'uso quando esistono alternative piú semplici

Istruzione `break`

Esempio (deprecato utilizzo di break). Calcolare la somma dei numeri in input, terminando al primo numero negativo, che non deve essere aggiunto alla somma.

```
int n,s;

while (1)
  scanf("%d", &n);
  if (n < 0)
    break;
  s += n;
```

```
scanf("%d", &n);
while ( n >= 0 )
  s += n;
  scanf("%d", &n);
```

Istruzione `continue`

- Interrompe l'iterazione corrente e avvia l'iterazione successiva
- Può essere utilizzata solo all'interno di `while`, `do...while`, `for`, con il seguente comportamento:

  `while` viene valutata l'espressione; se non ha valore non zero, viene eseguita l'istruzione

  `do...while` stesso comportamento

  `for` viene eseguita l'espressione di incremento e si valuta l'espressione; se non ha valore non zero, viene eseguita l'istruzione

Istruzione `continue`

Esempio. Acquisire un numero da input e determinare se è divisibile per `I`, con `I` variabile da 1 a 10

- *acquisisci dividendo K*
- *per I=1,...,10*
  - *se K/I non dà resto zero*
    - *avvia nuova iterazione*
  - *visualizza dividendo, divisore*

```
#include <stdio.h>
main() {
  int I,K;
  scanf("%d",&K);
  for (I=1; I<=10; I++) {
    if (K%I != 0)
      continue;
    printf("%d è divisibile per %d\n",K,I)
  }
}
```

Istruzione `continue`

- Riscritto senza `continue`

```
#include <stdio.h>
main () {
  int I,K;
  scanf (" %d" , &K) ;
  for (I=1; I<=10; I++) {
    if (K%I==0)
    printf( "%d e' divisibile per %d\n", K, I);
  }
}
```

Istruzione `switch`

- L'istruzione `switch` permette la scelta tra piú di due alternative, in base al confronto tra il valore di una espressione di tipo `int` o `char` e un insieme di valori costanti
- Sintassi:

$$\langle\text{istr-switch}\rangle \longrightarrow \texttt{switch(}\,\langle\text{espr-intera}\rangle\,\texttt{)}\,\langle\text{corpo-switch}\rangle$$

$$\langle\text{corpo-switch}\rangle \longrightarrow \langle\text{lista-case}\rangle\,\langle\text{sequenza-istr}\rangle$$

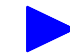
$$\langle\text{sequenza-istr}\rangle \longrightarrow \langle\text{istr}\rangle \mid \langle\text{istr}\rangle\,\langle\text{sequenza-istr}\rangle$$

$$\langle\text{lista-case}\rangle \longrightarrow \langle\text{case}\rangle \mid \langle\text{case}\rangle\,\langle\text{lista-case}\rangle$$

$$\langle\text{case}\rangle \longrightarrow \langle\text{espr-intera-costante}\rangle\,\texttt{:} \mid \texttt{default:}$$

Istruzione `switch`

- La sintassi di `switch` è soggetta ai seguenti ulteriori vincoli:
- ⟨espr-intera⟩, ⟨espr-intera-costante⟩ sono espressioni di tipo `int` o `char`
- Non sono ammessi piú ⟨case⟩ con lo stesso valore di ⟨espr-intera-costante⟩
- `default:` può comparire una volta sola

Istruzione `switch` 

- Funzionamento dell'istruzione `switch`:
  - L'espressione ⟨espr-intera⟩ viene valutata e confrontata con le espressioni costanti
  - Se il valore di una (e quindi una sola) delle espressioni costanti è uguale al valore dell'espressione intera, allora
    - si esegue la prima istruzione in posizione successiva ai due punti che seguono l'espressione costante; sia I tale istruzione
  - altrimenti
    - se è presente `default`, si esegue la prima istruzione in posizione successiva ai due punti seguenti `default`
  - <u>tutte</u> le istruzioni seguenti I fino al termine dell'istruzione `switch` vengono eseguite in sequenza

Istruzione `switch`

Osservazione. L'istruzione `break` è utile per evitare l'esecuzione delle istruzioni presenti in ⟨case⟩ diversi da quello contenente la ⟨espr-intera-costante⟩ che ha verificato l'uguaglianza con ⟨espr-intera⟩. Pertanto un forma spesso usata dell'istruzione `switch` è la seguente

```
switch (e) {
case c1 :
  ...
  break;
case c2 :
  ...
  break;
  ⋮
default :
  ...
}
```

Istruzione `switch` ▶

Esempio.

```
/*  http://www.its.strath.ac.uk/courses/c */
main() {
  int n;
  do {
    printf("Intero non negativo: ");
    scanf("%d", &n);
  }
  while (n<0);
  switch(n) {
  case 0 :
    printf("Nessuno\n");
    break;
  case 1 :
    printf("Uno\n");
```

(continua)

Istruzione `switch`

(continua)

```
      break;
    case 2 :
      printf("Due\n");
      break;
    case 3 :
    case 4 :
    case 5 :
      printf("Alcuni\n");
      break;
    default :
      printf("Molti\n");
      break;
    }
}
```

Istruzione `switch`

Esempio. Costruire un simulatore di calcolatore tascabile: un programma che legge in input un'espressione aritmetica, cioè una sequenza di valori numerici, separati da operatori + - * /, terminata da =, e calcola il valore dell'espressione

```
                              #include <stdio.h>
                              main()
                              {
                                float Risultato, Valore;
                                int Op;
- acquis. primo operando        printf("Espressione: ");
  memor. nel risultato          scanf("%f",&Risultato);
                                do
- acquis. primo operatore,        Op = getchar();
  scartando caratteri           while (Op!='+' && Op!='-' &&
  non validi                           Op!='*' && Op!='/' &&
                                       Op!='=');
```

Istruzione `switch`

- *finché l'operat. non è =*
  - *acquisisci operando*
  - *esegui operazione tra operando e risult.*

```
while (Op !='=') {
  scanf("%f",&Valore);
  switch(Op) {
  case '+':
    Risultato = Risultato+Valore;
    break;
  case '-':
    Risultato = Risultato-Valore;
    break;
  case '*':
    Risultato = Risultato*Valore;
    break;
  case '/':
    Risultato = Risultato/Valore;
    break;
  }
```

Istruzione `switch`

- *acquisisci operatore, scartando caratteri non validi*
- *visualizza risultato*

```
      do
        Op=getchar();
      while (Op!='+' &&
             Op!='-' &&
             Op!='*' &&
             Op!='/' &&
             Op!='=');
    }
    printf("Risultato: %f",Risultato);
  }
```

Astrazione e raffinamento

- La soluzione di problemi complessi è ricercata mediante decomposizione del problema e soluzione separata dei sottoproblemi
  - Ciascun sottoproblema viene associato a una opportuna *astrazione*, che consiste in una specifica dei dati e del risultato
  - Ciascun sottoproblema può essere a sua volta analizzato a un livello di *raffinamento* maggiore e decomposto
- I linguaggi di programmazione forniscono *procedure* e *funzioni* per la implementazione di soluzioni ai problemi in termini di astrazioni

procedura costruisce una istruzione definita dal programmatore; estende la nozione di istruzione

funzione costruisce una funzione definita dal programmatore; estende la nozione di operatore

Vantaggi della programmazione per astrazioni

facilità di sviluppo Consente di concentrare l'attenzione su un minore numero di aspetti in ogni fase del progetto

chiarezza e facilità di collaudo Il collaudo può essere compiuto separatamente per le soluzioni ai sottoproblemi

possibilità di riutilizzo Il codice può essere riutilizzato piú volte, abbattendo i costi

chiara definizione della comunicazione I dati necessari alla soluzione di un sottoproblema, forniti dal codice chiamante, e i dati modificati dalla soluzione, restituiti al chiamante, sono chiaramente specificati

Parametri delle unità di programma

- Quando, in un programma strutturato in unità, il controllo di esecuzione passa da una unità *chiamante* ad un'altra *chiamata*, i dati necessari alla corretta esecuzione della chiamata e i risultati da essa calcolati sono passati tramite

  variabili non locali: sono variabili visibili sia all'unità chiamante che all'unità chiamata; oppure tramite

  parametri: esiste una *intestazione* della unità chiamata in cui i parametri sono dichiarati;

  - l'unità chiamante collega ai parametri proprie variabili (che possono variare ad ogni attivazione)
  - i parametri dichiarati nella intestazione sono detti *formali*, mentre le variabili collegate in una specifica esecuzione sono i parametri *attuali* o *effettivi*

Legame per valore

- Utilizzato quando la chiamante deve fornire un valore alla chiamata; pertanto la comunicazione è *solo di ingresso* alla chiamata
- Il parametro attuale è una espressione
- Il parametro formale è una variabile locale
- È facoltà della chiamata utilizzare il parametro formale nella sua qualità di variabile. In ogni caso le variabili facenti parte del parametro attuale non mutano di valore.
- È richiesta al compatibilità per assegnamento tra parametro formale e parametro attuale

Legame per riferimento

- Il parametro attuale è una variabile della chiamante
- Il parametro formale si riferisce direttamente alla locazione del parametro attuale; pertanto è un altro nome per il parametro attuale, per tutta e sola la esecuzione della chiamata
- Parametro attuale e formale devono avere lo stesso tipo

Astrazione in C: la funzione

- Il linguaggio C ammette come unico meccanismo di astrazione la *funzione*, costituita da una *intestazione* e un *blocco*
  - Una procedura si realizza come una funzione particolare che non restituisce alcun valore
- Sintassi

$$
\begin{array}{rcl}
\langle \text{def-funzione} \rangle & \longrightarrow & \langle \text{intest} \rangle \; \langle \text{blocco} \rangle \\
\langle \text{intest} \rangle & \longrightarrow & \langle \text{nome-funz} \rangle \; \langle \text{param-formali} \rangle \\
 & & \mid \langle \text{tipo-risult} \rangle \; \langle \text{nome-funz} \rangle \; \langle \text{param-formali} \rangle \\
\langle \text{tipo-risult} \rangle & \longrightarrow & \langle \text{nome-tipo-risult} \rangle \mid \texttt{void} \\
\langle \text{param-formali} \rangle & \longrightarrow & \texttt{(} \; \langle \text{lista-param} \rangle \; \texttt{)} \mid \texttt{()} \mid \texttt{(void)} \\
\langle \text{lista-param} \rangle & \longrightarrow & \langle \text{param} \rangle \mid \langle \text{param} \rangle \; \texttt{,} \; \langle \text{lista-param} \rangle \\
\langle \text{param} \rangle & \longrightarrow & \langle \text{nome-tipo-param} \rangle \; \langle \text{nome-param} \rangle
\end{array}
$$

Astrazione in C: la funzione

- ⟨tipo-risult⟩ indica il tipo del risultato della funzione. Può essere
  - `void`: assenza di risultato, cioè la funzione rappresenta una procedura
  - un altro tipo ad eccezione del tipo array
- ⟨param-formali⟩ rappresenta i parametri formali della funzione
  - `void` indica assenza di parametri
  - ogni parametro è indicato con tipo e nome

Il blocco della funzione

- La parte $\langle$dichiarazioni$\rangle$ del blocco è detta *parte dichiarativa locale*; la parte $\langle$istruzioni$\rangle$ è detta anche *parte esecutiva*, o *corpo della funzione*
- La parte esecutiva spesso comprende una istruzione `return` con la sintassi

  $\langle$istr-return$\rangle$ $\longrightarrow$ `return` $\langle$espr$\rangle$

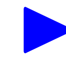
  dove $\langle$espr$\rangle$ ha il tipo $\langle$nome-tipo-risult$\rangle$
- `return` restituisce il risultato della funzione come valore di $\langle$espr$\rangle$ e restituisce immediatamente il controllo alla chiamante; pertanto, se è eseguita, è sempre l'ultima istruzione eseguita dalla funzione
- Il valore assunto da $\langle$espr$\rangle$ viene comunque convertito a $\langle$nome-tipo-risult$\rangle$

Chiamata delle funzioni ▶

- Sintassi

$$\langle\text{chiamata-funzione}\rangle \longrightarrow \langle\text{nome-funz}\rangle \; \texttt{(} \; \langle\text{param-attuali}\rangle \; \texttt{)}$$
$$\mid \langle\text{nome-funz}\rangle \; \texttt{()}$$
$$\langle\text{param-attuali}\rangle \longrightarrow \langle\text{espr}\rangle \mid \langle\text{espr}\rangle \; \texttt{,} \; \langle\text{param-attuali}\rangle$$

- Il collegamento tra parametri attuali e parametri formali
  - è stabilito secondo l'ordine di comparizione
  - è sempre per valore
- $\langle\text{chiamata-funzione}\rangle$ è una $\langle\text{espr}\rangle$: quindi può essere parametro attuale—è consentita la composizione di funzioni

Chiamata delle funzioni 

Esempio.

```
int MinimoInteri(int i, int j) {
/* I, J "parametri formali" */
  if (i<j)
    return i;
  else
    return j;
}
main() { /* chiamante */
  int M;
  int A=5, B=6;
  M = MinimoInteri(A,B);
  /* la funzione MinimoInteri e‘ la chiamata
  A, B sono parametri attuali, compatibili per assegnamento con i
  parametri formali */
}
```

Chiamata delle funzioni ▶

- Una chiamata di funzione è una espressione $\rightarrow$ può essere usata come parametro attuale nella chiamata di un'altra funzione, ad esempio `printf`

```
printf("%d",MinimoInteri(A,B);
```

oppure della stessa `MinimoInteri`

```
main() {
  int M;
  int A=5, B=6;
  M = MinimoInteri(A,MinimoInteri(B,3));
}
```

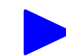
Chiamata delle funzioni ▶

Esempio. Scrivere una funzione per il calcolo del prodotto di due numeri interi non negativi X e Y utilizzando l'operatore $+$

```
                              int ProdottoInteri1(int X, int Y) {
 - inizializza P a 0            int P=0; /* X deve essere >= 0 */
 - inizializza W a X            int W=X;
 - finché W è diverso da 0      while (W>0) {
   - P=P+Y                        P = P + Y;
   - W=W-1                        W = W -1;
                                }
   - restituisci P              return P;
                              }
```

- X, Y non sono modificati nel corpo della funzione

Chiamata delle funzioni



- Non utilizziamo la variabile di appoggio `W` ma decrementiamo direttamente `X`

	`int ProdottoInteri2(int X, int Y) {`
– *inizializza P a 0*	`int P=0;`
– *finché X è diverso da 0*	`while (X>0) {`
– *P=P+Y*	`P = P + Y;`
– *X=X-1*	`X = X -1;`
	`}`
– *restituisci P*	`return P;`
	`}`

```
int ProdottoInteri2(int X, int Y) {
  int P=0;
  while (X>0) {
    P = P + Y;
    X = X -1;
  }
  return P;
}
```

Chiamata delle funzioni

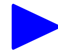
Esempio. Visualizzare il minimo di tre interi acquisiti da input, utilizzando una funzione che calcola il minimo di due interi

```
int MinimoInteri(int i, int j)
{
  if (i<j)
    return i;
  else
    return j;
}
main()
{
  int a,b,c;
  printf("Primo intero="); scanf("%d",&a);
  printf("Secondo intero="); scanf("%d",&b);
  printf("Terzo intero="); scanf("%d",&c);
  printf("%d\n",MinimoInteri(a,MinimoInteri(b,c)));
}
```

Struttura di un programma C

- Un programma C è costituito da
  - Una parte dichiarativa globale opzionale
  - la definizione della funzione `main()`
  - un insieme di definizioni di funzioni, eventualmente vuoto
- `main()` è abbreviazione di `void main(void)`: è l'intestazione della funzione `main`
  - `main` può avere parametri di input per comunicare con il sistema operativo: infatti è possibile eseguire il programma con argomenti che vengono passati come parametri di `main`
- Non si può definire una funzione all'interno di un'altra

Ricorsione



- In matematica sono frequenti le definizioni per induzione su $\mathbb{N}$
  - Successioni

$$a_1 = \frac{1}{2}$$
$$a_{n+1} = \frac{1}{2 + a_n}, \qquad \text{per } n > 0$$

  - La funzione fattoriale

$$0! = 1$$
$$n! = n \cdot (n-1)!, \qquad \text{per } n > 0$$

  - Soluzione di integrali

$$\int e^x dx = e^x + c$$
$$\int x^n e^x dx = x^n e^x - n \int x^{n-1} e^x dx, \qquad \text{per } n > 0$$

Ricorsione

- Una funzione $f\colon \mathbb{N} \to \mathbb{N}$ è definita per *ricorsione* quando può essere scritta come

$$f(0) = g_0$$
$$f(n) = h(n, f(n-1))$$

- Gli esempi precedenti ricadono in questo schema con

$g_0 =$	$\frac{1}{2}$	1	$e^x + c$
$h(\alpha, \beta) =$	$\frac{1}{2+\beta}$	$\alpha \cdot \beta$	$x^\alpha e^x - \alpha \cdot \beta$

Ricorsione

- La gestione a pila dei record di attivazione in C permette a una funzione di chiamare se stessa
- Ogni chiamata genera un nuovo record di attivazione
- La programmazione di una funzione definita per ricorsione è immediata

	`void f( `*parametri*` ) {`
	`if ( `*caso n = 0*` )`
g_0	*tratta caso n = 0*
	`else`
$h(n, f(n-1))$	*combina n con chiamata* `f(`*n-1*`)`
	`}`
	`main()`
	`{`
	`f( `*n*` )`
	`}`

Ricorsione

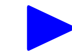
Esempio. Calcolare il fattoriale di un numero naturale con una funzione ricorsiva.

```
int Fattoriale(int n) {
  if(n==0)
    return 1;
  else
    return n*Fattoriale(n-1);
}
main()
{
  int n;
  printf("Digitare numero ");
  scanf("%d",&n);
  printf("Il fattoriale di %d è %d\n",n,Fattoriale(n));
}
```

Ricorsione

Esempio. Leggere da input a, b, n e calcolare $\int_a^b x^n e^x dx$. Impiegare una procedura ricorsiva per il calcolo della potenza.

```
#include <stdio.h>
#include <math.h>
float Potenza(float x, int n)
{
  if (n==0)
    return 1.0;
  else
    return x*Potenza(x,n-1);
}
float Integrale(float a, float b, int n)
{
  if (n==0)
    return exp(b)-exp(a);
  else
    return (Potenza(b,n)*exp(b)-Potenza(a,n)*exp(a))-n*Integrale(a,b,n-1);
}
main()
{
  int n;
  float a,b;
  printf("Digitare estremi e ordine "); scanf("%f %f %d",&a,&b,&n);
  printf("%f\n",Integrale(a,b,n));
}
```

Vettori e matrici 

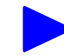
Problema. Dato un testo, si determini la frequenza assoluta (il numero di comparizioni) di ogni carattere.

- Possibile soluzione

```
- inizializza le variabili di conteggio a zero
- finché ci sono linee di testo da trattare
  finché ci sono caratteri sulla riga
  - considera il carattere corrente e incrementa
    la variabile corrispondente
  - scarica la riga
- stampa i valori di tutte le variabili di conteggio
```

- Inizializzazione e stampa comprendono 2*N istruzioni uguali, che differiscono solo per la variabile
- Due istruzioni ripetitive svolgerebbero lo stesso compito, se fosse possibile raccogliere le variabili sotto un nome unico e trattare ciascuna variabile in una ripetizione

Vettori e matrici



- Matrice
  - Corrispondenza tra un insieme di *indici* e un insieme di valori di un dominio DV
- Definizione del *tipo di dato matrice*
  - $indice$: un tipo numerabile ordinale, prodotto cartesiano di n tipi semplici numerabili
  - DV: un insieme di valori di tipo qualunque
  - Operazioni
    - $accedi: matrice \times indice \rightarrow V \in DV$ Data una coppia $(M, I) \in matrice \times indice$ restituisce il valore $V \in DV$ corrispondente a (M, I)
    - $memorizza: matrice \times indice \times DV \rightarrow matrice$ Data una terna $(M, I, V) \in matrice \times indice \times DV$ produce una matrice M' che ha il valore V nella posizione I, ed è identica a M altrove
  - Per $n = 1$ la matrice è chiamata *vettore*

Vettori e matrici

- Rivisitiamo il problema iniziale

  Problema. Dato un testo, si determini la frequenza assoluta (il numero di comparizioni) di ogni carattere alfabetico

  - *per ogni indice I*
    - *poni la variabile di indice I a zero*
  - *finché ci sono righe da trattare*
    - *finché ci sono caratteri sulla riga*
      - *leggi un carattere in C*
      - *se C è un carattere alfabetico minuscolo allora*
        - *determina l'indice corrispondente al valore di C*
        - *incrementa il valore corrispondente di uno*
  - *per ogni indice I*
    - *visualizza la variabile di indice I*

Il costruttore array [ ]

- Vettori e matrici si dichiarano con il *costruttore array* `[ ]`
- Sintassi

$$\langle\text{var-array}\rangle \longrightarrow \langle\text{tipo}\rangle\ \langle\text{identif}\rangle\ \langle\text{array}\rangle\ \texttt{;}$$

$$\langle\text{tipo-array}\rangle \longrightarrow \texttt{typedef}\ \langle\text{tipo}\rangle\ \langle\text{identif}\rangle\ \langle\text{array}\rangle\ \texttt{;}$$

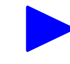
$$\langle\text{array}\rangle \longrightarrow \langle\text{costr-array}\rangle \mid \langle\text{costr-array}\rangle\ \langle\text{array}\rangle$$

$$\langle\text{costr-array}\rangle \longrightarrow \texttt{[}\ \langle\text{espress-costante-intera}\rangle\ \texttt{]}$$

  - $\langle\text{espress-costante-intera}\rangle$ stabilisce il numero di elementi del vettore ed è chiamato *dimensione* del vettore
  - Se n è la dimensione, l'indice può assumere valori tra 0 e $n-1$
- Operazioni sul tipo di dato array

$memorizza(M, I, V)$	`M[I]=V`
$accedi(M, I)$ e memorizza in VV	`VV=M[I]`

Il costruttore array [ ]



- L'inizializzazione si realizza elencando i valori tra graffe

```
int V[5] = {3,1,2,4,6};
int V[5] = {3,1};          /*primi due elem. inizializzati*/
int V[5] = {3,1,2,4,6,8}; /*non corretta,troppi elementi*/
int V[]  = {3,1,2};        /*array di tre elementi*/
```

- Non si può manipolare un array nel suo insieme; in particolare la operazione di copia deve essere programmata esplicitamente:

```
for (i=0; i<dimensione; i++)
  V2[i] = V1[i];
```

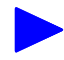
Il costruttore array [ ] ▶

Esempio. Acquisire da input un vettore di N interi e visualizzare il massimo

```
#include <stdio.h>
#define N 10
main() {
  int V[N];
  int Max;
  int i;
  for(i=0; i<N; i++) {
    printf("Digitare l'elemento %d: ",i);
    scanf("%d",&V[i]);
  }
  Max=V[0];        - considerare il primo elemento
                     come massimo

  for(i=1; i<N; i++)      - per le posizioni da 1 a N-1
    if (V[i]>Max)         - se l'elemento corrente
                            è maggiore del massimo
      Max = V[i];         - allora il massimo
                            è l'elemento corrente

  printf("Il valore massimo è %d",Max);
}
```

Il costruttore array [ ] 

- La dichiarazione di una matrice si ottiene applicando 2 volte il costruttore array

  ```
  int M[2][4]
  ```

  dichiara una matrice di 2 righe e 4 colonne
- Una matrice è considerata come un array di array—un array in cui ogni elemento è un array a sua volta:

  ```
  typedef int TV[4];
  TV M[2]
  ```
- Memorizzazione e accesso sono analoghi al vettore

$memorizza(M,(I,J),V)$	`M[I][J]=V`
$accedi(M,(I,J))$ e memorizza in VV	`VV=M[I][J]`

- Inizializzazione

```
int M[2][4] = { {1,2,3,4},
                {5,6,7,8} };
int M[2][4] = { 1,2,3,4,5,6,7,8 }; /*equivalente*/
```

Il costruttore array [ ] ▶

Esempio. Date le matrici

`int M1[2][3] = {1,2,3,4,5,6};`

`int M2[3][4] = {12,11,10,9,8,7,6,5,4,3,2,1};`

calcolare e visualizzare il loro prodotto.

```
- per ogni riga di M1
  - per ogni colonna di M2
    - inizializza M1XM2 a zero
    - per ogni colonna di M1
      - accumula comb. lin.

- visualizza matrice prodotto
```

```
#include <stdio.h>
#define NR1 2
#define NC1 3
#define NR2 3
#define NC2 4
main() {
  int M1[NR1][NC1] = {1,2,3,4,5,6};
  int M2[NR2][NC2] = {12,11,10,9,8,7,6,5,4,3,2,1};
  int M1XM2[NR1][NC2];
  int i,j,k;
  for(i=0; i<NR1; i++)
    for(j=0; j<NC2; j++) {
      M1XM2[i][j]=0;
      for(k=0; k<NC1; k++)
        M1XM2[i][j]=M1XM2[i][j]+M1[i][k]*M2[k][j];
    }
  for(i=0; i<NR1; i++) {
    for(j=0; j<NC2; j++)
      printf("%6d",M1XM2[i][j]); printf("\n");
  }
}
```

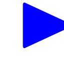
Il costruttore array [ ]

- Rivisitiamo il problema del conteggio dei caratteri

Problema. Dato un testo, si determini la frequenza assoluta di ogni carattere alfabetico.

```
                                          #include <stdio.h>
                                          #define NL 'z'-'a'+1
                                          main() {
                                            int Conteggio[NL];
                                            char Ch;
                                            int i;
- inizializza il vettore a zero             for (Ch=0; Ch<NL; Ch++)
                                              Conteggio[Ch]=0;
- finché ci sono caratteri                  do {
  su una riga
  - leggi carattere in Ch                     scanf("%c",&Ch);
  - se Ch è alfabetico minuscolo              if ((Ch >= 'a') && (Ch <= 'z'))
    - incrementa il valore                      Conteggio[Ch-'a']=Conteggio[Ch-'a']+1;
      Conteggio[Ch-'a']                     }
                                            while (Ch!='\n');
- visualizza vettore                        for(Ch=0; Ch<NL; Ch++)
                                              if (Conteggio[Ch]!=0)
                                                printf("%c = %d\n",Ch+'a',Conteggio[Ch]);
                                          }
```

Il costruttore array [ ]



Problema. Dato un vettore V di N elementi, stabilire se l'elemento D è memorizzato in una delle posizioni del vettore.

```
                                             #include <stdio.h>
                                             #define N 5
                                             main() {
                                               int V[N] = {1,6,7,3,21};
                                               int D;
                                               int i;

                                               printf("Digitare l'elemento da cercare: ");
                                               scanf("%d",&D);
                                               /* Ricerca lineare */
- inizializza la pos. corrente a 0             i=0;
- finché la pos. corrente non supera N-1       while((i<N-1) && (V[i]!=D))
  e l'elemento in pos. corrente
  è diverso da D
  - incrementa pos. corrente                     i++;
- se l'elemento corrente è uguale a D          if (V[i]==D)
  - visualizza messaggio di successo             printf("%d è nel vettore.\n",D);
  altrimenti                                   else
  - visualizza messaggio di fallimento           printf("%d non è nel vettore.\n",D);
                                             }
```

Il costruttore array [ ]

- Soluzione efficiente: ricerca *dicotomica*

```
variabili:
- vettore inizializzato
- elemento da cercare
- estremi di ricerca, pos. media
- esito della ricerca
- indice per iterazioni, iniz. a 0
algoritmo:

- acquisisci elemento da cercare


- iniz. estremi

- finché Inizio <= Fine e Trovato è 0
  - calcola la pos. centrale C
- se l'elemento di pos. C è uguale a D
  - poni Trovato uguale a 1
  altrimenti
  - se l'elemento di pos. C è  < di D
    - poni Inizio uguale a C+1
    altrimenti
    - poni Fine uguale a C-1

- se Trovato è 1
  - visualizza messaggio di successo
  altrimenti
  - visualizza messaggio di fallimento
```

```
#include <stdio.h>
#define N 5
main() {
  int V[N] = {1,6,7,3,21};
  int D;
  int Inizio,Fine,C;
  int Trovato=0;
  int i;

  printf("Digitare l'elemento da cercare: ");
  scanf("%d",&D);
  /* Ricerca dicotomica */
  Inizio=0;
  Fine=N-1;
  while((Inizio<=Fine) && (Trovato==0)) {
    C=(Inizio+Fine)/2;
    if (D==V[C])
      Trovato=1;
    else
      if (D>V[C])
        Inizio=C+1;
      else
        Fine=C-1;
  }
  if (Trovato==1)
    printf("%d è nel vettore.\n",D);
  else
    printf("%d non è nel vettore.\n",D);
}
```

Stringhe di caratteri

- Non esiste in C un tipo predefinito per rappresentare stringhe; i caratteri di una stringa si memorizzano in una variabile di tipo array di `char`, utilizzata secondo appropriate regole
- Una variabile `S` è una *stringa di lunghezza massima* N se è dichiarata come vettore di caratteri di lunghezza $N+1$:

```
char S[N+1];
```

  ed è utilizzata nel rispetto delle seguenti regole:
  - Esiste una posizione L $(0 \leq L \leq N)$ in cui è memorizzato il carattere speciale `'\0'`, che ha codice ASCII pari a zero, chiamato *terminatore* della stringa
  - Le posizioni da 0 a $L-1$ sono occupate da tutti e soli gli L caratteri della stringa
  - Le posizioni da $L+1$ a N hanno contenuto indefinito

Stringhe di caratteri

- Per la stringa di lunghezza zero (stringa *vuota*) si ha `S[0]='\0'`
- Una costante di tipo stringa di N caratteri si rappresenta con la successione dei suoi caratteri racchiusa tra una coppia di ":

```
"Linguaggio C"
```

ed è un vettore di $N+1$ caratteri

L	i	n	g	u	a	g	g	i	o		C	\0

- L'inizializzazione si realizza seguendo la sintassi dell'inizializzazione di un array, oppure con una costante stringa (anche in assenza di lunghezza dell'array)

```
char S1[8]={'s','t','r','i','n','g','a','\0'};
char S2[8]="stringa";
char S3[] ="stringa";
```

Stringhe di caratteri

- Ogni frammento di codice che tratta stringhe deve essere progettato in accordo alle regole di memorizzazione
  - Calcolo della lunghezza di una stringa

```
char S[] = "stringa";
int lunghezza = 0;
while(S[lunghezza]!='\0')
  lunghezza++;
printf("La stringa \"%s\" è lunga %d\n",S,lunghezza);
```

  - Copia di una stringa in un'altra

```
char S1[] = "stringa";
char S2[10];
int i;
i=0;
while(S1[i]!='\0') {
  S2[i]=S1[i];
  i++;
}
S2[i]='\0';
```

Record

- Record
  - Corrispondenza tra un insieme di *etichette* e un insieme di valori, in modo che ad ogni etichetta e_i corrisponda un valore di uno specifico dominio DV_i
- Definizione del *tipo di dato record*
  - $etichette = \{e_1, \ldots, e_n\}$: insieme finito di simboli, di cardinalità n
  - $DV = \{DV_1, \ldots, DV_n\}$: un insieme di insiemi di valori di tipo qualunque
  - Operazioni
    - $accedi : record \times etichette \rightarrow \bigcup DV$ Data una coppia $(R, e_i) \in record \times etichette$ restituisce il valore $v \in DV_i$ corrispondente a (R, e_i)
    - $memorizza : record \times etichette \times \bigcup DV \rightarrow record$ Data una terna $(R, e_i, v) \in record \times etichette \times \bigcup DV$ produce un record R' che ha il valore v nella etichetta e_i, ed è identico a R altrove

Il costruttore `struct`

- In C variabili di tipo record si dichiarano con il costruttore `struct`
- Sintassi semplificata

$$\langle\text{var-record}\rangle \longrightarrow \langle\text{costr-struct}\rangle\ \langle\text{identif}\rangle\ \texttt{;}$$

$$\langle\text{tipo-record}\rangle \longrightarrow \texttt{typedef}\ \langle\text{costr-struct}\rangle\ \langle\text{identif}\rangle\ \texttt{;}$$

$$\langle\text{costr-struct}\rangle \longrightarrow \texttt{struct\{}\ \langle\text{seq-campi}\rangle\ \texttt{\}}$$

$$\langle\text{seq-campi}\rangle \longrightarrow \langle\text{seq-campi-omog}\rangle \mid \langle\text{seq-campi-omog}\rangle\ \texttt{;}\ \langle\text{seq-campi}\rangle$$

$$\langle\text{seq-campi-omog}\rangle \longrightarrow \langle\text{tipo}\rangle\ \langle\text{seq-nomi-campi}\rangle$$

$$\langle\text{seq-nomi-campi}\rangle \longrightarrow \langle\text{identif}\rangle \mid \langle\text{identif}\rangle\ \texttt{,}\ \langle\text{seq-nomi-campi}\rangle$$

- Esempi

```
struct {
          int Giorno;
          int Mese;
          int Anno;
       } Data;
```

```
typedef struct {
                   float X,Y;
                } PuntoNelPiano;
```

Il costruttore `struct`

- Operazioni sul tipo di dato record

$memorizza(R, e, V)$	`R.e=V`
$accedi(R, e)$ e memorizza in VV	`VV=R.e`

- L'inizializzazione può essere effettuata nella definizione, specificando i valori tra graffe, nell'ordine in cui le corrispondenti etichette sono dichiarate

```
struct {
         int Giorno, Mese, Anno;
} Data = {29,6,1942};
```

- È consentito l'accesso alla variabile nel suo insieme, oltre che etichetta per etichetta

```
typedef struct {
  float X,Y;
} PtoNelPiano;
```

```
PtoNelPiano A,B;
PtoNelPiano PtoMax = {3.14,3.141};
A.X=3.14;
B.X=3.141;
A=PtoMax;
```

Il costruttore `struct`

▶

Esempio. Calcolo del punto medio di un segmento

```
#include <stdio.h>
typedef struct {
  float X,Y;
} PtoNelPiano;

PtoNelPiano PtoMedio(PtoNelPiano P1, PtoNelPiano P2) {
  PtoNelPiano P;
  P.X=(P1.X+P2.X)/2;
  P.Y=(P1.Y+P2.Y)/2;
  return P;
}
main(){
  PtoNelPiano A,B,PtoMedioAB;
  printf("Coordinate del primo punto: ");
  scanf("%f%f",&A.X,&A.Y);
  printf("Coordinate del secondo punto: ");
  scanf("%f%f",&B.X,&B.Y);
  PtoMedioAB=PtoMedio(A,B);
  printf("Punto medio: %f,%f\n", PtoMedioAB.X,PtoMedioAB.Y);
}
```

Il tipo puntatore

- I tipi finora visti danno luogo a dichiarazioni di variabili cosiddette *statiche* per le quali cioè
  - Il nome della variabile viene fatto corrispondere in fase di compilazione con un indirizzo di memoria
  - Il codice oggetto compilato contiene un riferimento a tale indirizzo dove il sorgente contiene un riferimento al nome
  - L'indirizzo è quello della prima di una serie di celle di memoria *allocate*, cioè riservate a quella variabile
- È possibile creare variabili *dinamiche*, chiamando in fase di *esecuzione* una opportuna procedura che
  - alloca lo spazio in memoria per la variabile
  - restituisce l'indirizzo di quello spazio
- Per utilizzare la variabile dinamica piú volte, l'indirizzo restituito deve essere memorizzato in una variabile di tipo particolare, il tipo *puntatore*

Il costruttore *

- Variabili di tipo puntatore si dichiarano con il costruttore * ;
- Sintassi semplificata

$$\langle \text{var-puntat} \rangle \longrightarrow \langle \text{costr-puntat} \rangle \, \langle \text{identif} \rangle \; \texttt{;}$$

$$\langle \text{tipo-puntat} \rangle \longrightarrow \texttt{typedef} \, \langle \text{costr-puntat} \rangle \, \langle \text{identif} \rangle \; \texttt{;}$$

$$\langle \text{costr-puntat} \rangle \longrightarrow \langle \text{tipo} \rangle \, \texttt{*}$$

$$\langle \text{oper-dereferenziazione} \rangle \longrightarrow \texttt{*} \, \langle \text{nome-variabile} \rangle$$

- Esempi

```
int *P;
struct {
  int Giorno,
      Mese,
      Anno; } *PData;
```

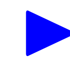
```
typedef int *TipoPI;
TipoPI P1,P2;
```

Il costruttore $*$

- Con lo stesso simbolo si denota l'operatore di *dereferenziazione*, che
  - applicato a una variabile di tipo puntatore rappresenta la variabile dinamica il cui indirizzo è memorizzato nel puntatore
- Sintassi semplificata

$$\langle \text{oper-dereferenziazione} \rangle \longrightarrow \texttt{*} \, \langle \text{nome-variabile} \rangle$$

Allocazione/deallocazione dinamica



- La funzione di allocazione dichiarata in `stdlib.h` con intestazione

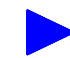
```
void *malloc(int NumByte)
```

  - alloca `NumByte` byte di memoria
  - restituisce un puntatore alla memoria allocata

  - Il calcolo del numero di byte sufficiente a contenere un valore di un tipo assegnato non è sempre agevole; perciò nella pratica si fa uso della funzione `sizeof` per calcolare il numero di byte
  - Il tipo della funzione è un puntatore a `void`; per assegnare il valore della funzione a un puntatore a un tipo qualunque, si fa uso di un type cast

```
int *P;
P = (int *) malloc(sizeof(int));
```

Allocazione/deallocazione dinamica



- Quando una variabile dinamica non è piú necessaria, occorre **sempre** *deallocarla*, cioè liberare la memoria ad essa riservata (da `malloc`)
- La funzione di deallocazione dichiarata in `stdlib.h` con intestazione

```
void free(void *P)
```

  - dealloca tutti i byte di memoria allocati alla variabile dinamica puntata dall'argomento `P`
  - lascia indefinito il valore di `P`
  - Se l'indirizzo di una variabile dinamica viene perso, **non è piú possibile deallocare** la memoria

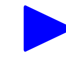
Allocazione/deallocazione dinamica

▶

Esempio. Allocazione e deallocazione corretta di una variabile dinamica di tipo `int`

```
#include <stdio.h>
#include <stdlib.h>
main() {
  typedef int TI;  /* dichiara tipo per la var. dinamica */
  typedef TI *PTI; /* dichiara il tipo puntatore */
  PTI P;           /* dichiara la var. statica puntatore */
  P=(int *)malloc(sizeof(int)); /* crea la var. dinamica */
  *P=3;                         /* assegna alla var. dinamica */
  printf("%d\n",*P);            /* accede  alla var. dinamica */
  free(P);                      /* rimuove la var.  dinamica */
}
```

Operatore &



- L'operatore di *estrazione di indirizzo* `&`, applicato a una varibile statica, ne restituisce l'indirizzo
- Sintassi

$$\langle\text{oper-estrazione-indirizzo}\rangle \longrightarrow \texttt{\&}\,\langle\text{nome-variabile}\rangle$$

- L'indirizzo può essere memorizzato in una variabile puntatore

```
int *P
int X=0, Y;
P = &X;
Y = *P;
*P = Y+1;
```

- L'applicazione piú significativa è però il passaggio dei parametri **per riferimento** nella chiamata di funzione

Operatore &

- Il passaggio dei parametri in C per riferimento si realizza utilizzando parametri formali di tipo puntatore e passando, come parametri attuali, gli **indirizzi** delle variabili, estratti con l'operatore &

- **Esempio.** Programmare una funzione Scambia che scambia il contenuto di due variabili in tipo intero **passate come parametri**

```
void Scambia(int *X, int *Y){
  int Temp;

  Temp = *X;
  *X = *Y;
  *Y = Temp;
}
```

```
main() {
  void Scambia(int *X, int *Y);
  int A=39,B=88;
  Scambia(&A,&B);
  printf("%d %d\n",A,B);
}
```

Operazioni su puntatori

- In generale, somma, sottrazione, incremento, decremento e operatori di confronto sono applicabili alle variabili di tipo puntatore, purché dello stesso tipo
  - Le effettive operazioni permesse dipendono dal compilatore
- Le operazioni aritmetiche considerano il *valore logico* del puntatore, non fisico
  - ad esempio il frammento di codice a lato incrementa `P` di 4 unità e incrementa `Q` di 8 unità (GNU C i386)

```
int *P;
double *Q;
P++;
Q++;
```

  - generalmente tali operazioni sono significative solo quando applicate a puntatori a elementi di un array

Operazioni su puntatori

- Ogni variabile `V` di tipo `array` ha un valore pari a `&V[0]`, l'indirizzo del suo primo elemento — cioè `V` è un **puntatore costante**
- Quindi si possono effettuare operazioni aritmetiche
  - `*(V+i)` equivale a `V[i]`
  - `&V[i]` equivale a `(V+i)`
  - `*(p+i)` equivale a `p[i]`

Operazioni su puntatori

- È essenziale, nell'utilizzo dell'aritmetica dei puntatori, prestare la **massima attenzione** all'intervallo di variazione degli indirizzi calcolati, che devono ricadere sempre nello spazio allocato a una variabile
  - risultati errati
  - interruzione forzata del programma da parte del sistema operativo con un errore a tempo di esecuzione

file sorgente erresec.c — compilazione/esecuzione a terminale

```
main() {
  char *p, V[]="stringa";
  p = V;
  printf("%s\n",p);
  printf("%s\n",p+200);
  printf("%s\n",p+5000);
}
```

```
> cc erresec.c -o erresec
> erresec
stringa
ÿÿ¿_ÿÿ¿®ÿÿ¿Âÿÿ¿Îÿÿ¿
Segmentation fault
>
```

La memoria secondaria ▶

- Caratteristiche della *memoria secondaria*

  Persistenza I dati sono conservati per lungo tempo anche in assenza di energia

  Tempo di accesso dell'ordine dei millisecondi

  Costo per byte molto inferiore alla memoria interna o primaria: oggi ≈ 0.5 Euro/GB contro ≈ 100 Euro/GB

- Realizzazioni della memoria secondaria

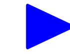
  Hard disk Supporto magnetico di grande capacità: $\sim$ centinaia di GB, scrivibile, fragile

  CD/DVD Supporto ottico di bassa capacità: CD $\approx 700 \div 800$ MB, DVD ≈ 8 GB robusto, scritture lente

La memoria secondaria

- Utilità della memoria secondaria
  - La quasi totalità dei dati oggi deve essere memorizzata in modo persistente
  - La memoria primaria non ha dimensioni sufficienti per molte applicazioni
    - Collezioni di programmi di uso comune (fino a varie decine di GB)
    - Basi di dati di grandi dimensioni (fino a vari Tera Byte) (1 Tera Byte $= 10^3$ GB)

Il file



File strutturato sequenza, di lunghezza non prefissata, di valori dello stesso tipo

- Accesso a un componente di un file
  - Si individua la posizione nella memoria secondaria del componente cercato

    Sequenziale Per accedere a un componente, tutti i componenti precedenti devono essere letti

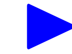
    Diretto Specificando l'indirizzo del componente nel file
  - Si effettua una operazione di ingresso
    - Il componente viene copiato in memoria primaria

Il file

- Il file in C è definito come una sequenza di lunghezza non prefissata di byte
- Si parla di file testo quando i byte del file sono considerati come codici di caratteri; altrimenti si parla genericamente di file binario
- In `stdio.h` è definita la struttura `FILE`; contiene
  - nome del file
  - modalità di accesso al file
  - posizione corrente
- Per utilizzare i file è necessaria la direttiva

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

Utilizzare un file in C



1. Dichiarare un puntatore a FILE

   `FILE *FP;`

2. Aprire il file con la funzione `fopen`, la cui intestazione è

   `FILE *fopen(const char *NomeFile, const char *Modo);`

   dove

   - `NomeFile` è un puntatore a una stringa che contiene un nome di file valido,
   - `Modo` può essere solo "r", "w", "a", "r+", "w+", "a+"
   - `fopen` restituisce un puntatore a una struttura di tipo `FILE` allocata dal sistema operativo in caso di successo, `NULL` altrimenti.

3. Eseguire letture e/o scritture

4. Chiudere il file con la funzione

   `int fclose(FILE *FP);`

Utilizzare un file in C ▶

`"r"`	lettura; la posizione corrente è l'inizio del file
`"w"`	scrittura; tronca il file a lunghezza zero, se esiste; altrimenti crea il file; la posizione corrente è l'inizio del file
`"a"`	scrittura alla fine (*append*); il file è creato se non esiste; la posizione corrente è la fine del file
`"r+"`	lettura e scrittura; la posizione corrente è l'inizio del file
`"w+"`	lettura e scrittura; tronca il file a lunghezza zero, se esiste; altrimenti crea il file; la posizione corrente è l'inizio del file
`"a+"`	scrittura alla fine (*append*) e lettura; il file è creato se non esiste; la posizione corrente è la fine del file

Utilizzare un file in C ▶

Posizione corrente numero (di byte) che misura la posizione attuale sul file

- una operazione di lettura o scrittura viene effettuata alla posizione successiva a quella corrente

- Nota:
  - Immediatamente dopo l'apertura in modo `"r"`, `"r"`, `"w+"`, `"w+"` la posizione vale 0
  - Immediatamente dopo l'apertura in modo `"a"`, `"a+"`, la posizione vale la lunghezza del file in byte

Utilizzare un file in C

Esempio.

```
#include <stdio.h>
#define NOME_FILE "prova.txt"
main() {
  FILE *FP;
  if ( (FP = fopen(NOME_FILE, "r")) == NULL )
    printf ("Impossibile aprire %s\n", NOME_FILE);
  else {
    /* elabora il file */
    fclose(FP);
  }
}
```

- Funzione utile:

`int feof(FILE *FP);`	restituisce non zero se non è stata raggiunta la fine del file, 0 se è stata raggiunta

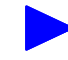
File testo

- Una sequenza di caratteri, organizzata in linee
- i dispositivi di ingresso e uscita sono disponibil come file testo

  `stdin` variabile che punta al file che rappresenta la tastiera

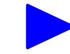
  `stdout` variabile che punta al file che rappresenta il video
- Tre famiglie di funzioni, per la lettura/scrittura
  - a caratteri
  - a stringhe
  - formattata

File testo



- Lettura/scrittura a caratteri

`int getc(FILE *FP);`	Restituisce il prossimo carattere in `FP`, o `EOF`, o un errore
`int putc(int Ch, FILE *FP);`	Scrive `Ch` restituendo come intero il carattere scritto
`int getchar(void);`	Legge da `stdin` il prossimo carattere e lo restituisce
`int putchar(int Ch);`	Scrive `Ch` su `stdout` e lo restituisce come intero

File testo 

Esempio. Lettura e visualizzazione di un file testo carattere per carattere

```
                            #include <stdio.h>
                            #define NOME_FILE "prova.txt"
                            main() {
                              FILE *FP;
                              char Ch;
- apri il file in lettura     FP=fopen(NOME_FILE, "r");
```

File testo

- se il file esiste e si può aprire
  - leggi carattere dal file, assegnalo a Ch
  - finché non è stata raggiunta la fine file
    - visualizza il carattere Ch
    - leggi carattere, assegnalo a Ch
  - chiudi il file

  altrimenti
  - visualizza messaggio di errore

```
if (FP != NULL) {
  Ch=getc(FP);

  while (!feof(FP)) {

    putchar(Ch);
    Ch=getc(FP);
  }
  fclose(FP);
}
else
  printf("Impossibile aprire %s\r
              NOME_FILE);
}
```

File testo

- Versione piú sintetica

```
#include <stdio.h>
#define NOME_FILE "prova.txt"
main() {
  FILE *FP;
  char Ch;
  if ((FP=fopen(NOME_FILE, "r")) != NULL) {
    while ((Ch=getc(FP)) != EOF)
      putchar(Ch);
    fclose(FP);
  }
  else
    printf("Impossibile aprire %s\n",
            NOME_FILE);
}
```

File testo

- I/O formattato su file avviene con modalità simili a quelle dell'I/O da tastiera e video
- Le funzioni `fprintf`, `fscanf` si comportano in modo simile a `printf`, `scanf`
  - `int fprintf(FILE *FP, const char *formato, ...);`
  - `int fscanf(FILE *FP, const char *formato, ...);`
- È possibile scrivere programmi utilizzabili sia con con file testo persistenti che con `stdin` e `stdout`
  - La logica di controllo della ripetizione può sfruttare in entrambi i casi `feof(FP)`, o la costante `EOF`
  - Nel caso di tastiera e video, l'uso di `feof(FP)` o `EOF` è un'alternativa alla richiesta all'utente di un carattere di scelta

File testo ▶

Esempio. Il file testo “reali.txt” contiene solo numeri reali in notazione decimale, separati da spazio bianco. Visualizzare la media di tutti i reali del file.

```
#include <stdio.h>
#define NOME_FILE "reali.txt"
main() {
  FILE *FP;
  float X, Somma;
  int N;
  if ( (FP = fopen(NOME_FILE, "r")) == NULL )
    printf("Impossibile aprire %s\n", NOME_FILE);
  else {
    Somma = 0;
    N = 0;
    while  ( fscanf(FP, "%f", &X) != EOF ) {
      Somma += X;
      N++;
    }
    printf("%f\n", Somma/N);
    fclose(FP);
  }
}
```

File testo

- Versione con lettura da tastiera

```
#include <stdio.h>
main() {
  FILE *FP;
  float X, Somma;
  int N;
  FP = stdin;
  Somma = 0;
  N = 0;
  while ( fscanf(FP, "%f", &X) != EOF ) {
    Somma += X;
    N++;
  }
  printf("%f\n", Somma/N);
}
```

Tipi di dati astratti

- Un *tipo di dato astratto* ha cinque componenti
  - un insieme di *atomi*, secondo il tipo di dato
  - un insieme di *posizioni* occupate dagli atomi
  - una *relazione strutturale* tra le posizioni
  - una funzione di valutazione associa atomi a posizioni
  - operazioni (procedure e funzioni) che specificano le manipolazioni ammesse

Tipi di dati astratti

- **Esempio.** La rubrica telefonica di un telefono mobile è un tipo di dato astratto

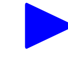
  atomi Tutte le stringhe di caratteri alfabetici di lunghezza massima 16 (S_{16})

  posizioni i numeri da 1 a 99

  relazione strutturale l'ordine totale dei numeri ristretta a $\{1, 2, \ldots, 99\}$

  funzione di valutazione una qualunque successione $s\colon \{1, 2, \ldots, 99\} \to S_{16}$

  operazioni $\{chiama, aggiungi, modifica\}$

Lista 

- *Lista semplice*

  atomi insieme omogeneo rispetto al tipo: caratteri, interi, reali, record, vettori, stringhe, . . .

  posizioni un iniziale dei naturali: $\{1, 2, \ldots, n\}$

  relazione strutturale Ordine lineare con primo e ultimo elemento

  operazioni inserimento, cancellazione, modifica, operazioni ausiliarie

- Implementazioni
  - Array: le posizioni sono gli indici di un array statico o dinamico
  - Puntatori: le posizioni sono puntatori a variabili dinamiche strutturate

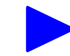
Lista

Operazione	Intestazione della funzione
crea una lista vuota	`void ListaCrea(Lista *L);`
vero se la lista è vuota	`boolean ListaVuota(Lista L)`
restituisce la prima posizione	`Posiz Primo(Lista L);`
restituisce l'ultima posizione	`Posiz  Ultimo(Lista L);`
restituisce la posizione successiva a P	`Posiz SuccL(Lista L);`
restituisce la posizione precedente a P	`Posiz PrecL(Lista L);`
restituiscel'atomo nella posizione P	`int Recupera(Posiz P,Lista L,Atomo *A);`
sostituisci l'atomo nella posizione P con A	`int Aggiorna(Atomo A, Posiz  P,Lista L);`
cancella l'atomo nella posizione P	`int Cancella(Posiz P,Lista *L);`
inserisce una nuovo atomo prima della posizione P	`int InserPrima(Atomo A,Posiz P,Lista *L);`
inserisce un nuovo atomo dopo la posizione P	`int InserDopo(Atomo A,Posiz P,Lista *L);`
restituisce la lungezza della lista	`int Lungh(Lista L);`

Lista implementata con array

- Gli elementi sono memorizzati in un array
- Il tipo dell'elemento dell'array è `Atomo`
- Le posizioni sono gli indici dell'array compresi tra 1 e il numero di elementi della lista
- Si memorizza la lunghezza L della lista per maggiore efficienza
- Per trattare i casi in cui la lista è vuota, si introduce la pseudo-posizione 0
- La list a ha una capienza pari al dimensionamento dell'array
- L'operazione di inserimento dopo una posizione P comporta
  - liberare la posizione successiva a P, copiando ciascun elemento seguente P nella posizione successiva alla propria; l'ordine di visita degli elementi è critico
  - copia dell'elemento da inserire nella posizione liberata

Lista implementata ad array



- In C, conviene per ragioni di efficienza passare la lista come puntatore anche quando la funzione non intende modificarne il contenuto
- Ad esempio si preferisce

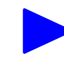
```
Posiz SuccL(Posiz P, Lista *L);
```

  a

```
Posiz SuccL(Posiz P, Lista L);
```

- La dimensione del parametro `L` nel record di attivazione sarebbe $O(n)$ utilizzando quest'ultima intestazione, solo $O(1)$ utilizzando la prima
  - esattamente `L` occuperebbe `sizeof(int)+MaxDim*sizeof(Atomo)` byte, dove `MaxDim` è la capienza della lista

Lista implementata ad array



- Utilizziamo una unità costituita dai file `Infobase.h` e `Infobase.c` per definire `Atomo` e alcune costanti e tipi indipendenti dall'implementazione

```
/* InfoBase.h */
#include <stdio.h>
#define MaxDim 10
/* elemento particolare */
#define Null 0

typedef int boolean;

typedef int Atomo;


extern void Acquisisci(Atomo *A);
extern void Visualizza(Atomo A);
extern int Minore(Atomo A,Atomo B);
extern boolean AtomoNullo(Atomo A);
```

```
/* InfoBase.c */
#include <stdio.h>
#include "InfoBase.h"

void Acquisisci(Atomo *A){
        char linea[80];
        printf("inserire un intero ");
        gets(linea);
        sscanf(linea,"%d", A);
}

void Visualizza(Atomo A){
        printf("%d\n",A);
}

int Minore(Atomo A,Atomo B){
        return (A<B);
}

boolean AtomoNullo(Atomo A){
        return A==0;
}
```

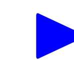
Lista implementata ad array ▶

```
#define ListaNoSucc 1 /* Codici di stato */
#define ListaNoPrec 2 /* Sono assegnati a ListaStato come */
#define ListaPosErr 3 /* risultato delle operazioni */
#define ListaPiena 4  /* soggette ad errore */
#define ListaOK 0     /* Terminazione senza errori */
#define NoPosiz 0     /* Posizione non valida */

typedef int Posiz;  /* 0, pseudo-posizione per lista vuota */
typedef struct {
      Posiz Lungh;
      Atomo Dati[MaxDim];
} Lista;

extern void ListaCrea(Lista *L);
extern boolean ListaVuota(Lista *L);
extern Posiz Primo(Lista *L);                        /* prima posizione */
extern Posiz Ultimo(Lista *L);                       /* ultima posizione */
extern Posiz SuccL(Posiz P, Lista *L);               /* posizione successiva a P */
extern Posiz PrecL(Posiz P, Lista *L);               /* posizione precedente a P */
extern int Recupera(Posiz P, Lista *L,  Atomo *A); /*atomo della posizione P => A*/
extern int Aggiorna (Atomo A, Posiz P, Lista *L);    /*A => atomo della pos. P*/
extern int Cancella(Posiz P, Lista *L);      /* cancella l'atomo della pos. P */
extern int InserDopo(Atomo A, Posiz P, Lista *L);  /* inserisce A dopo la pos. P */
extern int InserPrima(Atomo A, Posiz P, Lista *L); /*inserisce A prima della pos. P*/
extern int Lungh(Lista *L);          /* restituisce la lunghezza della lista */
extern char *ListaErrore ();
extern int InserOrdinato(Atomo A, Lista *L);
extern int ListaStato;
```

Lista implementata ad array



```
/* ListaArr.c */
#include "InfoBase.h"
#include "ListaArr.h"

int ListaStato=0;

void ListaCrea(Lista *L){
  L->Lungh=0;
}                /* end ListaCrea */

boolean ListaVuota(Lista *L){ /* *L per economia */
  return (L->Lungh==0);
}                /* end ListaVuota */

Posiz Primo(Lista *L){ /* *L per economia */
  if (L->Lungh==0)
    return NoPosiz;
  else
    return 1;
}              /* end Primo */

Posiz Ultimo(Lista *L){ /* *L per economia */
  if (L->Lungh==0)
    return NoPosiz;
  else
    return L->Lungh;
}              /* end Ultimo */
```

Lista implementata ad array ▶

```
Posiz SuccL(Posiz P, Lista *L){ /* *L per economia  */
  if ( (P<1) || (P>=L->Lungh))  /* P<1 non è valida  */
  {     /* l'ultimo non ha successore     */
    ListaStato = ListaNoSucc;
    return NoPosiz;
  }
  else{
    ListaStato = ListaOK;
    return (++P); /* !! (P++) NON VA BENE PERCHÉ.. */
  }
}     /* end SuccL */

Posiz PrecL(Posiz P, Lista *L){
  if ( (P<=1) || (P>L->Lungh)) /* P=1 non è valida  */
  {         /* il primo non ha precedenti  */
    ListaStato = ListaNoPrec;
    return NoPosiz;
  }
  else{
    ListaStato = ListaOK;
    return (--P);
  }
}       /* end SuccL */
```

Lista implementata ad array

```
int Recupera(Posiz P, Lista *L, Atomo *A){ /* *L per econ. */
  if ( (P<1) || (P>(L->Lungh))) /* pos. non valida */
      ListaStato = ListaPosErr;
  else{
    ListaStato = ListaOK;
    *A=L->Dati[P-1];
  }
  return ListaStato;
}       /* end Recupera */

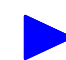
int Aggiorna(Atomo A, Posiz P, Lista *L){
  if ((P<1) || (P>L->Lungh)) /* pos. non valida */
      ListaStato = ListaPosErr;
  else{
      ListaStato = ListaOK;
      L->Dati[P-1]=A;
  }
  return ListaStato;
}         /* end Aggiorna */
```

Lista implementataad array

- Inserimento dopo una posizione assegnata P

```
int InserDopo(Atomo A, Posiz P, Lista *L){
  Posiz I;
  if ( (P< 0) || (P>L->Lungh)|| ((L->Lungh)==MaxDim))
    if ((L->Lungh)==MaxDim)
        ListaStato = ListaPiena;
    else
        ListaStato = ListaPosErr;
  else{
    ListaStato = ListaOK;
    for (I=L->Lungh;I>P;I--) /* crea spazio */
                L->Dati[I]=L->Dati[I-1];
    L->Dati[I] = A;
    L->Lungh++; /* incremento di lunghezza */
  }
  return ListaStato;
}         /* end InserDopo */
```

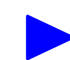
Lista implementata ad array



- Inserimento prima di una posizione assegnata P

```
int InserPrima (Atomo A, Posiz P, Lista *L){
  Atomo Temp;
  if ( (P< 0) || (P>L->Lungh)|| ((L->Lungh)==MaxDim))
    if ((L->Lungh)==MaxDim)
          ListaStato = ListaPiena;
    else
      ListaStato = ListaPosErr;
  else{     /* la posizione è accettabile */
    ListaStato = ListaOK;
    if (ListaVuota(L))
          InserDopo(A,P,L);
    else{ /* inserisce dopo e scambia i due atomi */
      InserDopo(A,P,L);
      Recupera(P,L,&Temp);
      Aggiorna(A,P,L);
      Aggiorna(Temp,SuccL(P,L),L);
    }
  } /* end if la posizione è accettabile */
  return ListaStato;
} /* end InserPrima */
```

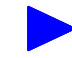
Lista implementata ad array



- Cancellazione

```
int Cancella(Posiz P, Lista *L){
  Posiz I;
  if ( (P<1) || (P>L->Lungh)) /* pos. non valida */
      ListaStato = ListaPosErr;
  else{
      ListaStato = ListaOK;
      for (I=P; I<L->Lungh;I++) /* compattamento */
          L->Dati[I-1]=L->Dati[I];
      L->Lungh--; /* decremento di lunghezza */
  }
  return ListaStato;
}         /* end Cancella */
```

Lista implementata ad array



- Gestione degli errori
  - Funzione per di visualizzazione dell'errore

```
char *ListaErrore (){
  switch(ListaStato){
    case  ListaNoSucc : return "Posizione errata per SuccL";
    break;
    case  ListaNoPrec : return "Posizione errata per PrecL";
    break;
    case  ListaPosErr : return "Posizione errata per lista";
    break;
    case  ListaPiena  : return "Lista Piena";
  }
  return "Stato errato";
}     /* end ListaErrore */
```

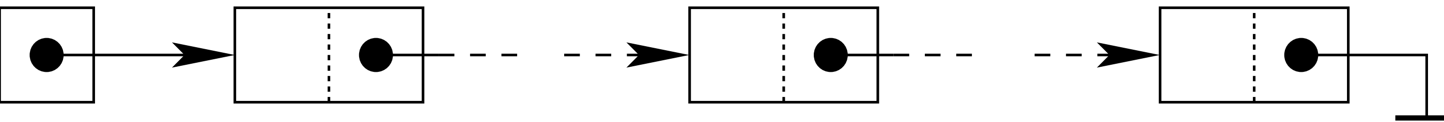
  - Le funzioni che restituiscono una posizione, in caso di errore restituiscono la posizione nulla `NoPosiz` e aggiornano la variabile di stato
  - le altre funzioni restituiscono immediatamente il valore della variabile di stato, che può essere esaminato dal programma chiamante

Lista implementata ad array ▶

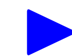
- Complessità computazionale delle operazioni
  - `InserDopo` e `Cancella` hanno complessità $O(n)$
    - contengono istruzioni ripetitive in cui il numero di ripetizioni è minore o uguale al numero di elementi della lista, a meno di costanti additive
  - Le rimanenti operazioni hanno complessità $O(1)$
    - Non contengono istruzioni ripetitive o ricorsioni

Lista implementata con puntatori

- Le posizioni possono essere cositituite da puntatori
- I puntatori sono parte di record in memoria dinamica
- La relazione strutturale lineare è memorizzata assicurando le seguenti proprietà
  - Esiste un unico record il cui puntatore ha valore `NULL`; in utti gli altri il puntatore contiene sempre un indirizzo valido di un record della lista
  - L'indirizzo del primo record è memorizzato in un puntatore statico
  - Non esistono due record con puntatori uguali
- In pratica, i puntatori implementano la relazione successore intercorrente tra gli elementi in posizione i e $i + 1$



Lista implementata con puntatori



- Nelle implementazione a puntatori è necessario utilizzare un tipo ricorsivo
  - è un tipo record contenente un puntatore al proprio tipo
- in C è realizzabile utilizzando i *tag* di struttura (*structure tag*)

```
typedef struct  TCella
{
        struct TCella  *Prox;
        Atomo          Dato;
}Cella;
```

- Memorizziamo puntatori al primo elemento, la *testa della lista*, e all'ultimo elemento, la *coda della lista*, e la lunghezza della lista

```
typedef  Cella *Posiz;

typedef struct{
        int Lungh;
        Posiz Coda,Testa;
} Lista;
```

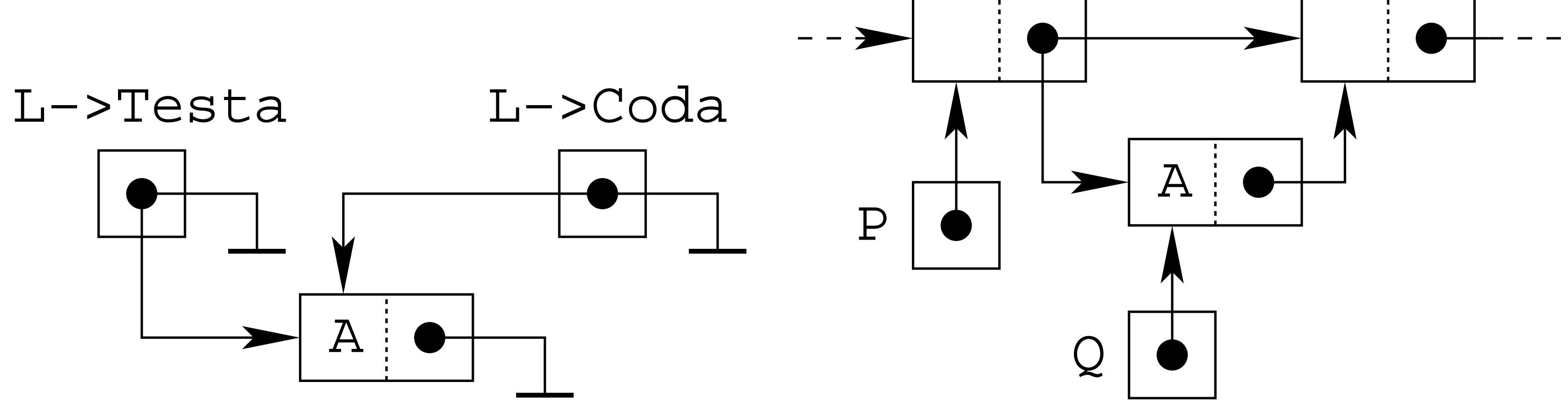
Lista implementata con puntatori

- Inserimento e cancellazione non richiedono,diversamente dal caso delle liste ad array, di copiare $O(n)$ elementi della lista
- È sufficiente allocare o deallocare una cella e modificare un numero limitato di puntatori

Lista implementata con puntatori

- Inserimento dopo una posizione assegnata

```
1  - se la lista è vuota
2    - alloca cella assegnando l'indirizzo al puntatore alla testa
3    - assegna A all'atomo di testa
4    - poni il successore della testa uguale a NULL
5    - poni la coda uguale alla testa
6    altrimenti
7    - alloca nuova cella e memorizza il suo indirizzo in Q
8    - assegna A all'atomo della cella puntata da Q
9    - poni il successore di Q uguale al successore di P
10   - poni il successore di P uguale a Q
11   - se P è uguale all'indirizzo della coda
12     - la nuova coda è Q
13 - incrementa la lunghezza della lista
```



Lista implementata con puntatori

```
int InserDopo(Atomo A, Posiz P, Lista *L){
  Posiz Q;
  if (L->Testa==NULL)
      {
          L->Testa=malloc(sizeof(Cella));
          L->Testa->Dato = A;
          L->Testa->Prox = NULL;
          L->Coda=L->Testa;
      }
  else
      {
          Q=malloc(sizeof(Cella));
          Q->Dato=A;
          Q->Prox=P->Prox;
          P->Prox=Q;
          if (P==L->Coda)
                L->Coda=Q;
      }
  L->Lungh++;
  ListaStato = ListaOK;
  return ListaStato;
}
```

Lista implementata con puntatori ▶

- Inserimento prima di una posizione assegnata

```
int InserPrima (Atomo A, Posiz P, Lista *L){
  Atomo Temp;
  if (ListaVuota(L))
      InserDopo(A,P,L);
  else
      { /* inserisce dopo e scambia i due atomi */
        InserDopo(A,P,L);
        Recupera(P,L,&Temp);
        Aggiorna(A,P,L);
        Aggiorna(Temp,SuccL(P,L),L);
      }
  ListaStato = ListaOK;
  return ListaStato;
}     /* end InserPrima */
```

Lista implementata con puntatori

- Cancellazione dell'elemento in posizione assegnata

```
- se la lista non è vuota e la posizione P è valida
  - se l'elemento da cancellare è quello di testa
    - se la lista aveva lunghezza unitaria
      - vuota la lista aggiornando testa e coda
    altrimenti
    - il secondo elemento diventa quello di testa
    altrimenti
    - cerca la posizione Q precedente all'elemento da cancellare
    - aggiorna il campo prossimo di Q con il campo prossimo di P
    - se P era la coda, Q diventa la nuova coda
  - rilascia l'elemento in posizione P
    e decrementa la lunghezza della lista
```

Lista implementata con puntatori

```
int Cancella(Posiz P, Lista *L){
  Posiz Q;
  if ((L->Lungh==0) || (P==NULL))
      ListaStato = ListaPosErr;
  else
    {
      ListaStato = ListaOK;
      if (P==L->Testa)
          if (L->Lungh==1)
              {
                  L->Testa=NULL;
                  L->Coda=NULL;
              }
          else
              L->Testa=L->Testa->Prox;
      else
          {
              Q=PrecL(P,L);
              Q->Prox=P->Prox;
              if (L->Coda==P)
                    L->Coda=Q;
          }
      free(P);
      L->Lungh--;
    }
    return ListaStato;
}
```