Array

Tipi di dato semplici e strutturati

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

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

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

Array monodimensionali (o vettori)

Dichiarazione di variabili di tipo vettore

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

Esempio: `int vet[6];`

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

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

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

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

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

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


L'indice del vettore deve essere un intero.

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

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

Manipolazione di vettori

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

  Es.: `#define LUNG 6`

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

Esempio: Lettura e stampa di un vettore.

Implementazione: file `array/vettrw.c`

Inizializzazione di vettori

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

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

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

In C l'unica operazione possibile sugli array è l'accesso agli elementi.

$\Longrightarrow$ Non si possono effettuare direttamente delle assegnazioni tra vettori.

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

Esempio: Calcolare la somma degli elementi di un vettore.

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

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

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


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

2 aspetti da considerare

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

Implementazione: file `array/esperime.c`

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

Esempio: Leggere una sequenza di caratteri terminata da '`\n`' e stampare le frequenze delle cifre da '`0`' a '`9`'.

Implementazione: file `array/frequen1.c` (versione con `switch`)
file `array/frequen2.c` (versione migliorata)

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

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

Array multidimensionali

Dichiarazione di array multidimensionali

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

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

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

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

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

Accesso agli elementi di una matrice

Esempio:

```
int i, mat[3][4];
...
i = mat[0][0];
mat[2][3] = 28;
mat[2][1] = mat[0][0] * mat[1][3];
```

elemento di riga 0 e colonna 0 (primo elemento)

elemento di riga 2 e colonna 3 (ultimo elemento)

Come per i vettori, l'unica operazione possibile sulle matrici è l'accesso agli elementi tramite l'operatore `[]`.

Esempio: Lettura e stampa di una matrice.

Implementazione: file `array/matrici.c`

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

Implementazione: file `array/matsomma.c`

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

Inizializzazione di matrici

Esempio:

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

1	2	3
4	5	6

```
int mat[2][3] = {{1,2,3}};
int mat[2][3] = {1,2,3};
```

1	2	3
0	0	0

```
int mat[2][3] = {{1}, {2,3}};
```

1	0	0
2	3	0

Esempio: Programma che legge una matrice A $(M \times P)$ ed una matrice B $(P \times N)$ e calcola e stampa la matrice C prodotto delle due matrici.

La matrice C è di dimensione $M \times N$.

Il generico elemento C_{ij} di C è dato da:

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

Implementazione: file `array/matprod.c`

In alternativa, si può inizializzare c contestualmente alla sua dichiarazione.

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


Esercizio: Programma che legge una matrice A $(M \times N)$ e:

1. stampa l'elemento massimo con i suoi indici di riga e di colonna;
2. costruisce il vettore degli elementi massimi di ogni riga, e lo stampa;
3. costruisce il vettore degli elementi massimi di ogni colonna, e lo stampa;
4. verifica se la matrice è diagonale (in questo caso $M = N$)
   (una matrice si dice diagonale se $A_{ij} = 0$ quando $i \neq j$);
5. verifica se la matrice è simmetrica
   (una matrice si dice simmetrica se $A_{ij} = A_{ji}$ per ogni coppia di indici i e j);
6. calcola la matrice T $(N \times M)$ trasposta di A, e la stampa
   (il generico elemento T_{ij} della trasposta T di A è pari ad Aji).

Esempio: Programma che legge una matrice A $(M \times N)$ e verifica se tutte le somme degli elementi di ogni riga coincidono.

algoritmo verifica se le somme delle righe di *A*$(M \times N)$ sono tutte uguali tra loro
  prima $\leftarrow$ somma degli elementi della prima riga di *A*
  for ogni riga *i* di *A*, finché le somme delle righe sono tutte uguali
  do *somma* $\leftarrow$ somma degli elementi della riga *i* di *A*
    if *somma* $\neq$ *prima*
    then le somme delle righe sono diverse
  stampa se le somme delle righe sono diverse o meno

Implementazione: file `array/matsomri.c`

Si poteva risolvere senza usare una matrice?
E se dobbiamo verificare se le somme degli elementi di ogni colonna coincidono?

Esercizio: Programma che legge da tastiera una matrice quadrata e verifica se è magica, ovvero se le somme delle righe, delle colonne e delle due diagonali coincidono.

Soluzione: file `array/magica.c` (fa uso di funzioni con parametro di tipo matrice)

Aritmetica dei puntatori

Sui puntatori si possono effettuare diverse **operazioni**:

- di **dereferenziamento**
  Esempio:
  ```
  int *p, i;
  i = *p;
  ```
- di **assegnamento**
  Esempio:
  ```
  int *p, *q;
  p = q;
  ```
  N.B. `p` e `q` devono essere dello stesso tipo (altrimenti bisogna usare l'operatore di cast).
- di **confronto**
  Esempio: `if (p == q) ···`
  Esempio: `if (p > q) ···` **Ha senso?** Con quello che abbiamo visto finora no. Vedremo tra poco che ci sono situazioni in cui ha senso.
- **aritmetiche**, con opportune limitazioni
  - incremento (`++`) o decremento (`--`)
  - somma (`+=`) o sottrazione (`-=`) di un intero
  - sottrazione di un puntatore da un altro

Significato delle operazioni aritmetiche sui puntatori

Il **numero di byte** di cui viene modificato il puntatore **dipende dal suo tipo**.

Esempio:
```
int *pi;
*pi = 15;
pi++;
*pi = 20;
```
$\Longrightarrow$ `pi` punta al prossimo `int` (4 byte dopo)

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

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

Si può anche scrivere:
```
pi = pi + 1;
pd = pd + 3;
pc = pc - 5;
```


Relazione tra vettori e puntatori

Attenzione: in generale non sappiamo cosa contengono le celle di memoria adiacenti ad una data cella.

L'unico caso in cui sappiamo quali sono le locazioni di memoria successive e cosa contengono è quando utilizziamo dei vettori.

In C il **nome di un vettore è** in realtà **l'indirizzo dell'elemento di indice** 0.

Esempio: `int vet[10];` `vet` e `&vet[0]` hanno lo stesso valore.
$\Longrightarrow$ `printf("%p %p", vet, &vet[0]);` stampa 2 volte lo stesso indirizzo.

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

Esempio:
```
int vet[5];
int *pi;
pi = vet;
```
è equivalente a `pi = &vet[0];`

Accesso agli elementi di un vettore

Esempio:
```
int vet[5];
int *pi = vet;
*(pi + 3) = 28;
```
`pi+3` punta all'elemento di indice 3 del vettore.

3 viene detto **offset** (o scostamento) del puntatore.

N.B. Servono le "`()`" perchè "`*`" ha priorità maggiore di "`+`". Che cosa denota `*pi + 3` ?

Osservazione: `&vet[3]` equivale a `pi+3` equivale a `vet+3`
`*&vet[3]` equivale a `*(pi+3)` equivale a `*(vet+3)`

Inoltre, `*&vet[3]` equivale a `vet[3]`.

$\Longrightarrow$ In C, `vet[3]` è semplicemente un modo alternativo di scrivere `*(vet+3)`.

Notazioni per gli elementi di un vettore:
`vet[3]` .... notazione con **puntatore e indice**
`*(vet+3)` .... notazione con **puntatore e offset**

Riassumendo, si può accedere agli elementi di un vettore al seguente modo:

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

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

Esempio: Sono corrette le istruzioni nel seguente frammento di codice?
```
int vet[10];
int *pi;

vet = pi;
vet++;
vet += 2;
```

No, perché `vet` è un **puntatore costante**, e quindi non può essere modificato.

Passaggio di parametri di tipo vettore

Quando si passa un vettore come parametro ad una funzione, in realtà si sta passando l'indirizzo dell'elemento di indice 0.

Il parametro formale deve essere di tipo puntatore al tipo degli elementi del vettore.

Di solito si passa la dimensione del vettore in un ulteriore parametro.

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

int main(void)
{ ...
  double vet[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
  stampa(vet, 5);
  ...
}
```

Per evidenziare che il parametro formale è in realta un vettore (ovvero l'indirizzo dell'elemento di indice 0), di solito si usa la notazione *`nome-parametro[]`* invece di *`*nome-parametro`*.

Esempio: `void stampa(int v[], int dim) { ... }`

Si può anche specificare la dimensione nel parametro, ma viene ignorata.
Esempio: `void stampa(int v[5], int dim) { ... }`

Nel prototipo della funzione può anche mancare il nome del vettore.
Esempio: `void stampa(int [], int);`

Il passaggio di un vettore è in realtà un **passaggio per indirizzo**.
$\Longrightarrow$ La funzione può modificare gli elementi del vettore passato.

Esempio: Funzioni per la lettura e stampa di un vettore.
Implementazione: file `array/vettfunz.c`

Esempio: Programma che legge un vettore di dimensione N e lo inverte.
Implementazione: file `array/vettinv.c` (versioni iterativa e ricorsiva)

Utilizzo di `sizeof` con i vettori

Abbiamo visto che un vettore è un puntatore costante all'elemento di indice 0.

Esempio:
```
int vet[5];
int *pi = vet;
pi[3] = 12;
```
Possiamo usare un puntatore per accedere agli elementi del vettore.

Importante differenza tra vettori e puntatori:
- `sizeof(pi)` è equivalente a `sizeof(int*)`
- `sizeof(vet)` è equivalente a `5*sizeof(int)`
  $\Longrightarrow$ per i vettori, `sizeof` restituisce la dimensione dell'intero blocco

Il C usa la dimensione degli elementi di un vettore per calcolare l'offset di un elemento quando usiamo la notazione con indici.

Esempio: `int v[5];`

`v[i]` è equivalente a `*(v+i)`. Questo corrisponde alla cella di memoria a distanza `i*sizeof(int)` da quella a cui punta `v`.

Questo influenza il modo in cui il C gestisce le matrici.

Esempio: `int mat[2][3] = {{0, 1, 2}, {3, 4, 5}}`
corrisponde ad un vettore di 2 elementi, ognuno dei quali è un vettore di 3 interi.
Quindi `mat` è un puntatore ad un vettore di 3 `int`.

`mat[1][2]` è equivalente a
`*(mat[1] + 2)` che è equivalente a
`*(*(mat + 1) + 2)`
- nel valutare `*(mat+1)`, sul valore `1` viene effettuato uno scaling della dimensione dell'oggetto a cui punta `mat`, ovvero un array di 3 `int`
  $\Longrightarrow$ incremento di $1 \cdot 3 \cdot$ `sizeof(int)` byte
- nel valutare `*(mat[1] + 2)`, sul valore `2` viene effettuato uno scaling della dimensione dell'oggetto a cui punta `mat[1]`, ovvero un `int`
  $\Longrightarrow$ incremento di $2 \cdot$ `sizeof(int)` byte

Otteniamo: `mat` $+ 1 \cdot 3 \cdot 4 + 2 \cdot 4$ byte $=$ `mat` $+ 20$ byte
Più precisamente, `mat[1][2]` è equivalente a:
`*(int*) ((char*)mat + (1*3*4) + (2*4))`, ovvero
`*(int*) ((char*)mat + 20)`

In generale:
```
#define R ...
#define C ...
int mat[R][C];
```

`mat[i][j]` è equivalente a
`*(mat[i] + j)` che è equivalente a
`*(*(mat + i) + j)`

L'indirizzo di `mat[i][j]` è: $\texttt{mat} + (\texttt{i} \cdot \texttt{C} \cdot \texttt{sizeof(int)} + \texttt{j} \cdot \texttt{sizeof(int)})$ byte

Quindi, per calcolare l'indirizzo dell'elemento `mat[i][j]` è necessario conoscere:
- il valore di `mat`, ovvero l'indirizzo del primo elemento della matrice
- l'indice di riga `i` dell'elemento
- l'indice di colonna `j` dell'elemento
- il numero `C` di colonne della matrice

Non è invece necessario conoscere il numero `R` di righe della matrice.

Nota: Se si specifica un indice in meno delle dimensioni, si ottiene un puntatore al tipo base dell'array.

Esempio: `mat[1]` coincide con `&mat[1][0]`, che fornisce un puntatore ad `int`.

Passaggio di matrici come parametri

Quando passiamo un vettore ad una funzione stiamo passando il puntatore (costante) all'elemento di indice 0. $\Longrightarrow$ **Non** serve specificare la dimensione del vettore nel parametro formale (è un puntatore al tipo degli elementi del vettore).

Quando passiamo una matrice ad una funzione, per poter calcolare l'offset corretto, la funzione deve **conoscere il numero di colonne** della matrice. $\Longrightarrow$
Non possiamo specificare il parametro nella forma `mat[][]`, come per i vettori, ma dobbiamo specificare il numero di colonne.

Esempio: `void stampa(int mat[][5], int righe) { ... }`

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

Esempi ed **esercizi** sulle matrici: Usando opportune funzioni realizzare per ciascuno dei punti seguenti un programma che legge una matrice A $(M \times N)$ e

- stampa l'elemento massimo con i suoi indici di riga e colonna
  Soluzione: file `array/matmax.c`
- costruisce il vettore degli elementi massimi di ogni riga, e lo stampa
  Soluzione: file `array/matvetma.c`
- verifica se è diagonale (in questo caso $M = N$) (ovvero, $A_{ij} = 0$ quando $i \neq j$)
  Soluzione: file `array/matdiago.c`
- verifica se è simmetrica (ovvero, $A_{ij} = A_{ji}$)
  Soluzione: file `array/matsimm.c`
- calcola la matrice $N \times M$ trasposta
  Soluzione: file `array/mattrasp.c`

Esercizio: Fornire versioni di `array/matdiago.c` e `array/matsimm.c` in cui l'uscita dai cicli viene anticipata usando una variabile booleana e/o l'istruzione `break`.

Array dinamici

Un vettore è un puntatore costante e possiamo assegnarlo ad una variabile di tipo puntatore.

Esempio:
```
int vet[5];
int *pi = vet;
pi[3] = 18;
```

Invece di far puntare `pi` ad un vettore allocato attraverso una dichiarazione (quindi statico o sullo stack), possiamo farlo puntare ad una zona di memoria allocata dinamicamente:

Esempio:
```
int *pi, dim;
scanf("%d", &dim);
pi = malloc(dim * sizeof(int));
pi[dim-1] = 20;
```

Esempio: Progettare un programma che legge un intero N, alloca dinamicamente un vettore di N interi, legge N interi da tastiera e stampa gli N interi in ordine inverso. Implementazione: file `array/allocdin.c`

Esercizio: È necessario gestire l'archivio di un deposito di autobus. Ciascun autobus è identificato da un codice numerico (di tipo `int`).

- Quando un autobus arriva al deposito, il suo codice deve essere inserito nell'archivio nella prossima posizione libera.
- Quando un autobus parte, il suo codice deve essere cancellato dall'archivio, e i codici dei rimanenti autobus devono essere scalati di una posizione, in modo da mantenere l'ordine di arrivo degli autobus.

Progettare un programma per la gestione dell'archivio, in cui l'interfaccia permetta di gestire:

- l'arrivo di un nuovo autobus (evitando duplicati nell'archivio);
- la partenza di un autobus;
- la stampa della lista di autobus nel deposito (in ordine di arrivo).

Si realizzi l'archivio tramite un vettore dinamico.

- Inizialmente l'archivio può contenere i codici di al più 5 autobus.
- Se è necessario inserire un nuovo autobus e l'archivio è pieno, allocare dinamicamente un nuovo vettore di dimensione doppia, ricopiare il contenuto del vecchio vettore nel nuovo, e deallocare il vecchio vettore.
- Se è necessario cancellare un autobus dall'archivio e l'archivio è pieno per meno di un terzo della sua capacità, allocare dinamicamente un nuovo vettore di metà dimensione, ricopiare il contenuto del vecchio vettore nel nuovo, e deallocare il vecchio vettore.

Vettori di puntatori

Gli elementi di un vettore possono essere puntatori.

Esempio: `char * vet1[20];` ... vettore di 20 puntatori a `char`
è equivalente a `char * (vet2[20]);` ... vettore di 20 puntatori a `char`
che è diverso da `char (* vet3)[20];` ... puntatore a vettore di 20 `char`

Esempio:
```
printf("%d %d\n", sizeof(vet1), sizeof(*vet1));   stampa 80 4
printf("%d %d\n", sizeof(vet2), sizeof(*vet2));   stampa 80 4
printf("%d %d\n", sizeof(vet3), sizeof(*vet3));   stampa 4 20
```

Nota: Ogni elemento di `vet1` (o `vet2`) è un puntatore a `char`.

Tipicamente i puntatori a `char` si usano per le stringhe.

Una **stringa** è un vettore di caratteri, terminato dal carattere `'\0'` (ovvero, in cui l'ultimo elemento da considerare per una generica operazione è il primo che si incontra contenente il carattere `'\0'`).

Qualificatore `const`

Il qualificatore `const` permette di specificare che una variabile deve essere **costante**, ovvero non può essere modificata dal programma.

N.B. Questo è diverso dalle costanti simboliche definite con `#define`.

Esempio:
```
const double pi = 3.1415;
pi = 5.2;
```
$\Longrightarrow$ dà errore di compilazione

Bisogna fare attenzione a dove si mette `const`:

`int * const pi;` ... (puntatore costante) ad `int`

`const int *pi;` ... puntatore ad (`int` costante)

Ha importanza soprattutto nel passaggio dei parametri alle funzioni.

Esempio: `void stampaValore(const int x) { ... }`

$\Longrightarrow$ la funzione non può modificare il parametro `x`.

Usare `const` permette al compilatore di rilevare errori di programmazione (dovuti alla modifica errata di un parametro).

Modi di utilizzare `const` quando si hanno parametri

- passaggio per valore: due modi
  - modificabile (senza `const`): la funzione può alterare la sua copia locale del valore passato come parametro
  - con `const`: la funzione non può alterare la sua copia locale del valore passato come parametro
- passaggio per indirizzo: quattro modi

  puntatore modificabile/costante a dati modificabili/costanti

Cosa dobbiamo usare?

Si applica il **criterio del minimo privilegio**: Si permette alla funzione di fare solo quello che deve, e niente di più.

1. Puntatore modificabile a dati modificabili

Dopo aver risolto il riferimento i dati possono essere modificati ed il puntatore può essere modificato per puntare ad altri dati.

Esempio: Stringa che deve essere modificata, ed in cui faccio l'incremento del puntatore per scorrere gli elementi.

```
void inMaiuscole(char *s)
{
  while (*s != '\0') {
    if (*s >= 'a' && *s <= 'z')
      *s = *s + 'A' - 'a';
    s++;
  }
}
```

2. Puntatore modificabile a dati costanti

Gli elementi del vettore non possono essere modificati, ma il puntatore può essere incrementato per scandire gli elementi.

Esempio: Stampa di una stringa.

```
void stampaStringa(const char *s)
{
  while (*s != '0') {
    putchar(*s);
    s++;
  }
}
```

Esercizio: Conteggio del numero di occorrenze di un carattere in una stringa.

3. Puntatore costante a dati modificabili

Questo è il caso dei vettori dichiarati con `[]` passati alle funzioni (un vettore è un puntatore constante), nel caso in cui **vogliamo** permettere la modifica degli elementi del vettore.

Esempio: Lettura di un vettore da tastiera.

```
void leggiVettore(int * const vet, int dim)
{
  int i;
  for (i = 0; i < dim; i++)
    scanf("%d\n", &vet[i]);
}
```


4. Puntatore costante a dati costanti

Questo è il caso dei vettori dichiarati con `[]` passati alle funzioni (un vettore è un puntatore constante), nel caso in cui **non vogliamo** permettere la modifica degli elementi del vettore.

Esempio: Stampa degli elementi di un vettore.

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

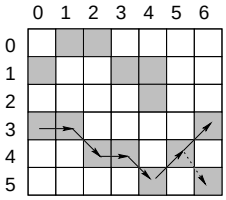
Esempio: Attraversamento di una palude.

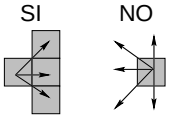
Una palude è rappresentata come una zona di $R \times C$ aree quadrate, ciascuna delle quali è – un'area di terra, o
– un'area di acqua.

Vogliamo attraversare la palude dal bordo sinistro a quello destro.

Sviluppare un programma che, usando una mappa della palude, trova un cammino attraverso la palude (**attraversamento**):

- solo su zone di terra
- assunzione semplificativa: ogni passo deve fare avanzare verso il bordo destro





Un attraversamento è un percorso

- da una zona nella colonna 0
- ad una zona nella colonna $C-1$
- spostandosi ad ogni passo di una colonna in avanti.

Es.: colonna: 0 1 2 3 4 5 6
riga: 3 3 4 4 5 4 3

Strutture di dati

- mappa della palude: matrice $R \times C$ di 0 (acqua) o 1 (terra).
- cammino: vettore di C elementi, ognuno dei quali è un indice di riga (tra 0 e $R-1$)
  Passaggio per la zona (i,j) è rappresentato memorizzando il valore i nella componente di indice j del vettore.

Es.: colonna: 0 1 2 3 4 5 6

riga:	3	3	4	4	5	4	3

Algoritmo di ricerca di un attraversamento

- Utilizziamo una funzione ricorsiva che cerca un attraversamento a partire da una generica posizione (i,j).
- Per cercare un attraversamento della palude si deve cercarlo a partire dalle posizioni della prima colonna, ovvero $(0,0),\ldots,(R-1,0)$.

algoritmo cerca un attraversamento a partire dalla posizione (i,j)
  if (i,j) non è di terra
  then non esiste un attraversamento a partire da (i,j)
  else aggiungi (i,j) al cammino corrente
    if $j = C-1$
    then l'attraversamento è stato trovato (passo base)
    else cerca un attraversamento a partire da una delle posizioni raggiungibili da (i,j), ovvero $(i-1,j+1)$ (solo se $i > 0$)
      $(i,j+1)$
      $(i+1,j+1)$ (solo se $i < R-1$)

algoritmo cerca un attraversamento della palude
  inizializza i a 0
  while l'attraversamento non è stato trovato e i è minore di R
  do cerca l'attraversamento a partire da $(i,0)$
    $i \leftarrow i+1$
  if l'attraversamento è stato trovato **then** restituisci il cammino trovato
  else restituisci che non esiste un attraversamento

Implementazione: file `ricorsio/palude.c`

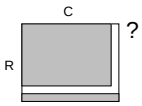
Visualizziamo l'evoluzione della pila dei record di attivazione per la palude di esempio.

- gli unici parametri di `cercaCammino` che cambiano sono `i` e `j`
- `palude` e `cammino` sono array, e quindi passati implicitamente per indirizzo

Osservazione: ogni zona di terra può venire esplorata più volte $\Longrightarrow$ inefficiente

Es.: Come si comporta il precedente programma sulla palude ?

Molto male: $\sim R \cdot 3^C$ attivazioni ricorsive



Come si può ottimizzare, evitando di esplorare più volte la stessa posizione?
È sufficiente aggiungere prima delle chiamate ricorsive: `palude[i][j] = 0;`
Corrisponde a marcare una posizione già esplorata (come se fosse acqua) in modo che successivamente non venga più considerata (N.B. viene modificata la matrice originale).

Esercizio: Permettere anche movimenti nelle altre direzioni:

- è indispensabile marcare le posizioni già visitate in modo da evitare cicli infiniti
- serve un altro modo per rappresentare un cammino (può essere più lungo di C)