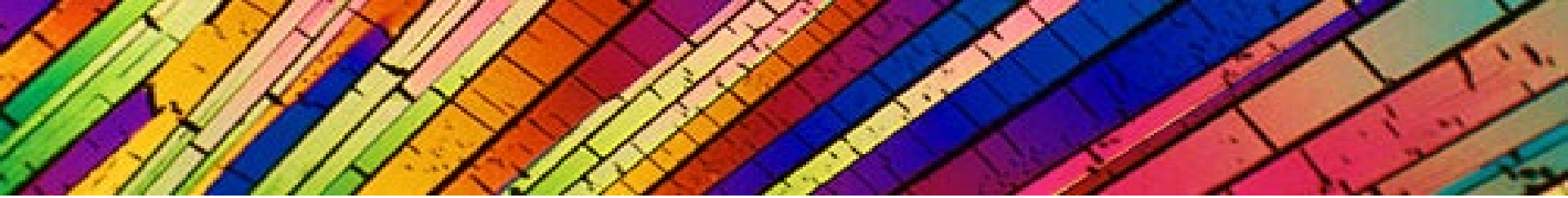
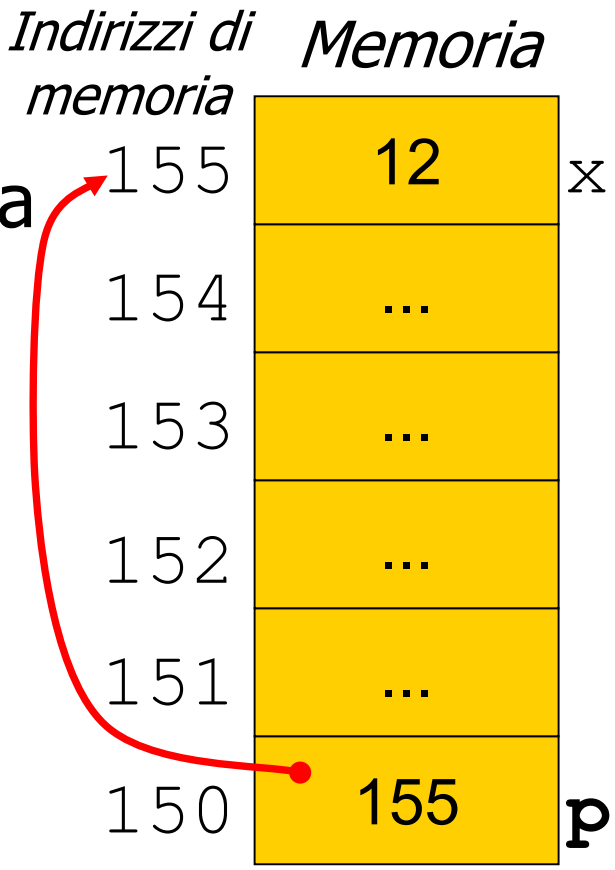


I puntatori

Ver. 3

Puntatore

- Una *variabile puntatore* contiene l'*indirizzo di memoria* di un oggetto (variabile o costante)
- Esempio
  `p` è una *variabile puntatore* e "punta" alla variabile intera `x` che in questo esempio ha *indirizzo di memoria* 155 e valore 12
  quindi `p` contiene 155



Puntatore

- È comune utilizzare il termine *puntatore* non solo per riferirsi a una *variabile puntatore*, ma (impropriamente) anche a un generico indirizzo di memoria (ad esempio si dice che "`&a` dà il *puntatore* ad `a`" invece che il più corretto "`&a` dà l'indirizzo di memoria di `a`")
- Quando è importante distinguere i due casi, si usano esplicitamente i termini *"variabile puntatore"* e *"indirizzo di memoria"*

Definizione

- Sintassi
  tipo **nomeVariabile*;
- La definizione di *ogni* puntatore richiede un '`*`'
- Esempi
  `char x, *p, *q, y;`
  `x` e `y` sono variabili di tipo `char`
  `p` e `q` sono variabili di tipo *puntatore-a-*`char`
- La definizione di una variabile puntatore riserva memoria solo per contenere l'indirizzo di un oggetto, ma *non riserva memoria per l'oggetto puntato:* il puntatore deve essere inizializzato o assegnato successivamente

Indirizzi di memoria

- L'*operatore di indirizzo* '&' ("ampersand") determina l'indirizzo di memoria di un oggetto (variabile, costante, array, ecc.):

```
int x, vet[10], mx[5][10];
```

`&x` → *determina l'indirizzo dell'intero* `x`

`&vet[5]` → *det. l'indirizzo dell'intero* `vet[5]`

`&mx[3][2]` → *det. l'indirizzo dell'intero* `mx[3][2]`

`&mx[3]` → *determina l'indirizzo del vettore-di-10-interi* `mx[3]`

- Si ricordi che il nome di un vettore/matrice è già il sinonimo dell'indirizzo di memoria del suo *primo elemento*, quindi `&vet` di `&mx` non hanno alcun significato

Assegnazione

- Essendo un puntatore una variabile di tipo numerico, l'assegnazione avviene con l'operatore '='
- Il valore da assegnare a una variabile puntatore deve essere un indirizzo di memoria:

```
int *p, *q;
int x, vet[10];
p = &x;   → assegna a p l'indirizzo di x
q = vet;  → assegna a q l'indirizzo di vet[0]
```

- Si noti che la variabile `p` non è preceduta da '`*`' quando viene assegnata

Inizializzazione

- Per inizializzare un puntatore con un indirizzo di memoria si usa la consueta sintassi:

```
int x=6, y = 21;
int *p = &x;
```

→ *definisce* `p` *e lo inizializza con l'indirizzo di* `x`

```
int *q = &y;
int z = 12, *t = &z;
```

- Queste sono invece assegnazioni:

```
q = &x;
```

→ *assegna* `q` *con l'indirizzo di* `x`

```
p = &y;
```

→ *assegna* `p` *con l'indirizzo di* `y`

- Qui `p` e `q` diventano *alias* uno dell'altro:

```
p = q;
```

→ *i due puntatori puntano alla stessa var.*

Utilizzo

- Per accedere all'oggetto puntato da un puntatore, nelle espressioni si usa l'operatore '*****' detto ***operatore di deriferimento*** ("dereference") o di ***indirezione*** ("indirection")
- Se `p` è un *puntatore*, `*p` è l'*oggetto puntato*

```
int x = 24;
int *p = &x;
```

dopo quest'ultima inizializzazione, `x` e `*p` identificano la *stessa variabile*, `x` direttamente e `*p` indirettamente, quindi:

`*p` `= 12;` → *ora* `x` *vale 12*

`*p` `+= 6;` → *ora* `x` *vale 18*

Utilizzo

- Si noti che la sintassi per indicare l'oggetto puntato e quella della definizione del puntatore sono identiche: `*p`
- Attenzione a non confondere il significato degli asterischi '`*`' 1) nella definizione e 2) nell'uso:

  1) `int *p = &x;`

     nella definizione l'asterisco serve per la *definizione del tipo*, quindi l'indirizzo prodotto da `&x` viene messo in `p` e non in `*p`, come avviene in `p = &x;`

  2) `*p = 12;`

     nell'utilizzo l'asterisco è l'*operatore di deriferimento* e quindi `*p` identifica l'oggetto puntato da `p` (cioè `x`) ed è quindi `x` che viene modificato

Utilizzo

- Sia `p` sia `*p` sono *L-value modificabili*
- Un *L-value modificabile* è un "qualcosa" (variabile, elemento di vettore, …) a cui si può assegnare un valore (ved. slide su espressioni)
- Un asterisco '`*`' è considerato operatore di indirezione solo se precede un puntatore, altrimenti viene valutato come moltiplicazione
- L'operatore '`*`' ha priorità superiore a quella degli operatori matematici
  `x = 6 * *p;` equivale a: `x = 6 * (*p);`
- Per visualizzare il valore di un puntatore in una `printf` si può utilizzare la specifica `%p`

NULL

- `NULL` è un valore di tipo `void*` definito in `stdio.h`, `stdlib.h`, `string.h`, `stddef.h`, `time.h`, `locale.h`
- Per indicare che un puntatore non punta a nulla si utilizza il valore `NULL`

```
int *p = NULL;   → inizializzazione
p = NULL;        → assegnazione
```

- In un contesto dove è previsto un puntatore, la costante 0 è convertita in `NULL` dal compilatore, in un contesto aritmetico `NULL` non è convertito in 0 mentre in un contesto logico equivale a falso

Tipi e puntatori

- L'informazione relativa al tipo è necessaria per permettere ai puntatori di conoscere la dimensione dell'oggetto puntato
- L'assegnazione e il confronto tra puntatori di tipo diverso (escluso `void`) è errata e il compilatore genera un Warning

```
int *p, x=12;
long *q, y=26;
p = &x;   → OK!
q = &y;   → OK!
q = p;    → NO! Warning
q = &x;   → NO! Warning
```

Cast di puntatori non void

- Si può usare un cast per modificare il tipo del puntatore

```
int *p = (int *)q;
```

- Il cast può contenere le keyword `const` e `volatile` (descritta in altre slide)

```
int *pc = (const int *)p;
```

Puntatori a void

- Sono puntatori generici e *non possono essere dereferenziati* (non si può scrivere `*p`), possono essere utilizzati solo come *contenitori* temporanei di puntatori a qualsiasi tipo
  `void *h;`
- Per *dereferenziare* un puntatore (accedere al valore puntato) a `void` è necessario prima convertirlo a un puntatore al tipo appropriato per poter far conoscere al compilatore la dimensione dell'oggetto puntato

Puntatori a void

- Non è necessario il cast `(void *)` per copiare un puntatore non-`void` in un puntatore `void`
  `void *h = p;` *(con ad es.* `int *p;`*)*
- Non è necessario il cast `(`*tipo*` *)` per copiare un puntatore `void` in un puntatore non-`void`, ma per chiarezza è bene utilizzarlo:

```
int *p;
p = h;          → OK
p = (int *)h;   → OK, più chiaro
*p = 23;        → ora x contiene 23
```

- Qualsiasi tipo di puntatore può essere confrontato con un puntatore a `void`

Puntatori e vettori

- Il **nome** di un *vettore-di-T* è un valore *costante* di tipo *puntatore-a-T* e corrisponde all'indirizzo di memoria del primo elemento di *vettore*, ossia `vett` equivale a `&vett[0]`
- Se al nome di un vettore viene *sommato un valore intero* `i`, il risultato è l'indirizzo di memoria dell'elemento di indice `i`

  Più in generale, le equivalenze sono:
  - `vett+i` equivale a `&vett[i]`, in entrambi i casi il valore è l'*indirizzo dell'elemento* di indice `i`
  - `*(vett+i)` equivale a `vett[i]`, in entrambi i casi il valore è quello dell'*elemento* di indice `i`

Puntatori e vettori

- Il nome di un *vettore-di-T* può quindi essere assegnato a una variabile di tipo *puntatore-a-T*, così quest'ultimo contiene l'indirizzo di memoria di `vett` e ora punta a `vett[0]`
- Esempio

```
int vett[100];
int *p;
p = vett;
```

l'indirizzo di memoria di `vett` viene messo in `p`, equivale a scrivere: `p = &vett[0]` (le parentesi hanno priorità maggiore di `&`)

Puntatori e vettori

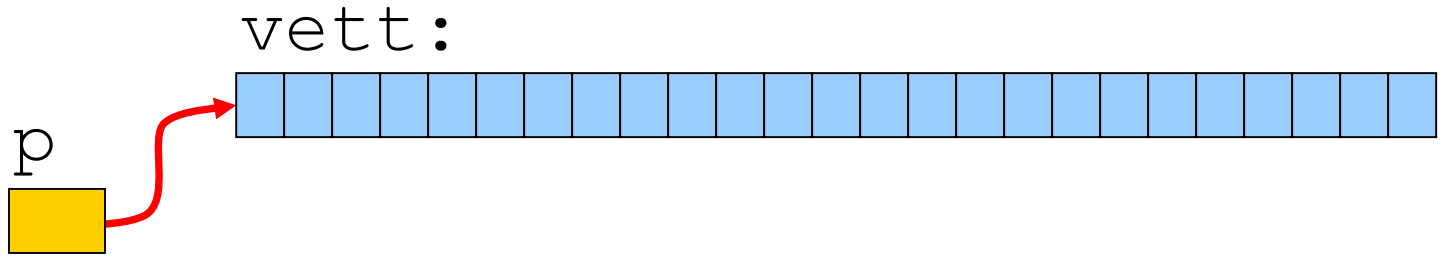
- Attenzione, è ERRATO scrivere:
  `vett = p;`
- Non si può assegnare un valore a `vett` in quanto NON è una *variabile puntatore*, ma un "sinonimo" di un *indirizzo di memoria*

  Gli indirizzi di memoria sono valori *costanti* stabiliti dal compilatore, non sono variabili e quindi non hanno uno spazio in memoria modificabile per contenere un valore (ossia non sono L-value)

Puntatori e vettori

- Se si assegna l'indirizzo di un oggetto di tipo *vettore-di-T* a una variabile di tipo *puntatore-a-T*, questa può essere utilizzata come un *vettore-di-T*

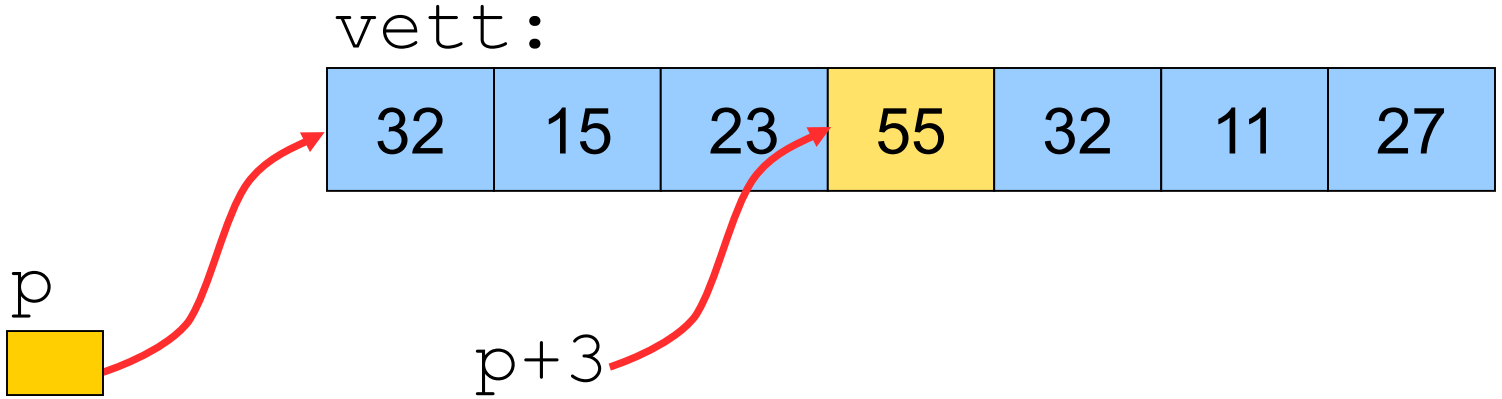
```
int vett[25];
int *p = vett;
```



Ad esempio, qui `p[3]` equivale a `vett[3]`

Aritmetica dei puntatori

- Quando `p` punta a un vettore, gli può essere *sommato un valore intero* `i`, il risultato è l'indirizzo di memoria dell'elemento che si trova `i` *posizioni* (non byte) dopo `p`



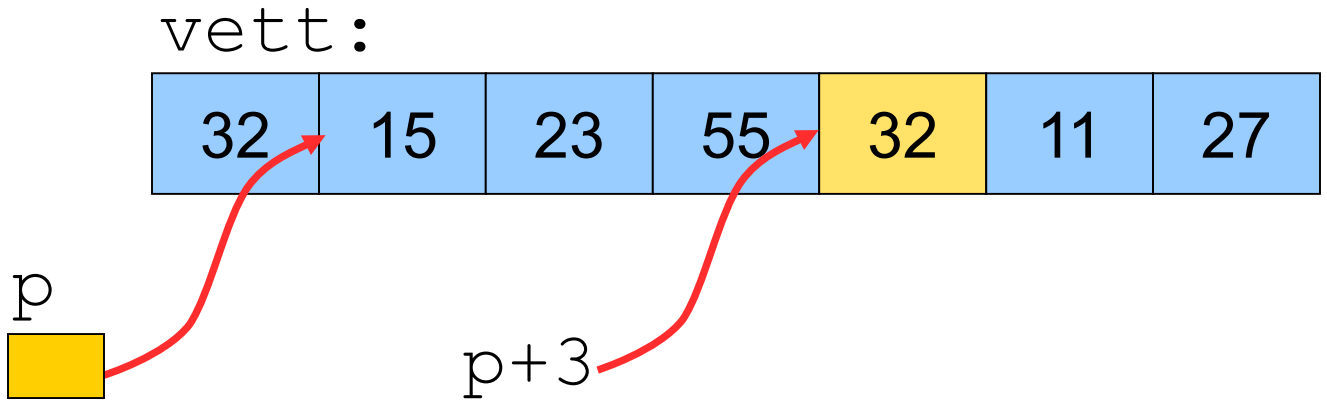
`p` punta a (contiene l'indirizzo di) `vett`

`p+3` punta a (produce l'indir. di) `vett[3]`

`*(p+3)` equivale a `vett[3]`

Aritmetica dei puntatori

- L'istruzione `p++` porta `p` a puntare a `vett[1]` (ne contiene l'indirizzo) quindi ora `p` punta a `vett[1]` e `p[3]` corrisponde a `vett[4]`



- È lecito *sottrarre un valore* `i` a un puntatore se l'elemento puntato risultante fa ancora parte del vettore (nell'esempio `p-1` punta a `vett[0]`)

Aritmetica dei puntatori

- In C i vettori iniziano con indice 0, ma usando i puntatori è possibile simulare un vettore con indice iniziale qualsiasi, ad esempio:
  `int v[13], *voti=v-18;`
  `voti` *simula* un vettore con indice da 18 a 30
  E quindi è possibile scrivere:
  `voti[18]=10;  voti[30]=5;`
  ma in realtà è `v[0]=10` e `v[30]=5`
- L'alternativa classica è usare il solo vettore `v` e sottrarre ogni volta uno scostamento (qui 18): `v[x-18]=10;` con `x` che varia da 18 a 30, ma essendo calcolato ogni volta, è meno efficiente
- Attenzione a non sforare il vettore `v`

Equivalenza puntatori e vettori

- L'operatore `sizeof` (discusso in altre slide) dà la dimensione in byte di un oggetto, se viene applicato a:
  - **un vettore**: dà la dimensione dell'*intero vettore* (numero di elementi * dimensione elemento)
  - **un puntatore**: dà la dimensione della *variabile puntatore* e non del vettore eventualmente puntato (alcuni byte, es. 4)

Conversione dei puntatori

- È il compilatore che per facilità d'uso converte il tipo del nome di un vettore da *vettore-di-T* a *puntatore-a-T*
- Questa conversione è detta "decadimento" o "decay" e non avviene quando il nome del vettore:
  - è preceduto da '`&`': in questo caso il valore costante prodotto è di tipo *puntatore-a-vettore-di-T* (nella sintassi C il tipo è: `T (*)[]`)
  - è passato a `sizeof` (che dà la dim. del vettore)
  - è un letterale stringa usato per inizializzare una variabile stringa (array di `char`)

Equivalenza puntatori e vettori

- Una variabile di tipo *puntatore-a-T* non "sa" se il valore a cui punta è singolo o è l'elemento di un vettore, lo sa solo il programmatore e sta a questi utilizzarlo in modo coerente

```
int x = 10, vett[10], *p, *q;
p = &x;
p++;        NO! Non esiste l'oggetto puntato da p+1
q = p+1;    NO! Non esiste l'oggetto puntato da p+1
p = vett;
p++;        OK! Ora p punta a vett[1]
q = p+1;    OK! Ora q punta a vett[2]
```

Equivalenza puntatori e vettori

- Il compilatore *in genere* trasforma le espressioni con notazione vettoriale `[]` in espressioni con i puntatori
- Non è detto, con gli attuali compilatori, che un ciclo che scandisca i vettori mediante puntatori sia *sempre* più veloce di uno che utilizzi la notazione vettoriale `[]`
- La scelta tra la notazione con puntatori e quella vettoriale deve essere suggerita dalla chiarezza del codice e non da questioni di efficienza, questione rimandata al compilatore

Aritmetica dei puntatori

- Il puntatore che "sfora" i limiti del vettore è un problema frequente e importante, lo standard non richiede che il compilatore faccia controlli, molti compilatori lo offrono opzionalmente, ma ciò riduce le prestazioni
- È lecito che un puntatore *punti* a quello che sarebbe l'elemento del vettore *successivo all'ultimo* (ma non esiste), ma questo puntatore può essere utilizzato **solo** per calcolare la differenza o fare confronti tra puntatori (vedere prossime slide)

Aritmetica dei puntatori

- Due puntatori a elementi dello *stesso vettore* possono essere ***sottratti***, il *risultato+1* è il numero di elementi del vettore compresi tra quelli puntati dai due puntatori (inclusi):

```
p = &vett[4];
q = &vett[10];
d = q-p+1;
```

→ 7: num degli elementi dalla posizione 4 alla 10 incluse

- Se i due puntatori non appartengono allo stesso vettore il risultato è indefinito

Aritmetica dei puntatori

- Due puntatori possono essere ***confrontati*** con i normali operatori relazionali
- Per gli operatori `<`, `<=`, `>` e `>=` questo è lecito solo se fanno parte dello *stesso vettore* oppure uno dei due è `NULL` (o `0`)
- Per elaborare tutti gli elementi di un vettore è anche possibile usare le forme:
  - `for (p=&vett[0]; p<&vett[N]; p++)…`
  - `for (p=vett; p<vett+N; p++)…`

  ricordando che `&vett[N]` è lecito se non viene utilizzato il valore contenuto

Priorità dell'operatore *

- Dalla tabella delle priorità si vede che l'operatore di *deriferimento* '`*`' ha priorità quasi massima, inferiore solo alle parentesi, a '`->`' e a '`.`', e ha associatività da destra a sinistra
- Quindi, considerando che gli operatori `*` e `++` hanno stessa priorità e associatività D → S:
  - `*p++` equivale a `*(p++)` → `++` incrementa `p`
  - `*++p` equivale a `*(++p)` → `++` incrementa `p`
  - `++*p` equivale a `++(*p)` → `++` incrementa `*p`

  inoltre:
  - `(*p)++` → incrementa `*p`
  - `*p+1` equivale a `(*p)+1` e non a `*(p+1)`

Copia di stringhe - 1ª versione

- La stringa `y` viene copiata in `x`

```
char x[30], y[30], *t=x, *s=y;
int i=0;
gets(y);
while (s[i] != '\0')
{
   t[i] = s[i];
   i++;
}
t[i] = '\0';
```

- Il `'\0'` viene copiato fuori dal ciclo
- `s` e `t` vengono usati senza vantaggio come semplici sinonimi di `x` e `y`, non come puntatori

Copia di stringhe - 2ª versione

- La stringa `y` viene copiata in `x`

```
char x[30], y[30], *t=x, *s=y;
int i=0;
gets(y);
while ((t[i] = s[i]) != '\0')
   i++;
```

- Il `'\0'` viene copiato nel ciclo stesso
- `s` e `t` vengono usati senza vantaggio come semplici sinonimi di `x` e `y`, non come puntatori
- *Nota: non si può scrivere* `t[i] = s[i++]` *nella condizione del* `while` *(side effect)*

Copia di stringhe - 3ª versione

- La stringa `y` viene copiata in `x`

```
char x[30], y[30], *t=x, *s=y;
gets(y);
while ( (*t = *s) != '\0')
{
   t++;
   s++;
}
```

- Il `'\0'` viene copiato nel ciclo stesso
- *Nota:* `!= '\0'` *può essere omesso*

Copia di stringhe - 4ª versione

- La stringa `y` viene copiata in `x`

```
char x[30], y[30], *t=x, *s=y;
gets(y);
while ((*t++ = *s++))
    ;
```

- Il `'\0'` viene copiato nel ciclo stesso
- È sintatticamente corretto lasciare solo una coppia di parentesi, ma molti compilatori in questo caso segnalerebbero un Warning: un `while` contiene una condizione e un singolo '=' è in genere una svista, il compilatore lo segnala come *possibile* problema

Puntatori e stringhe

- Si noti la differenza tra le seguenti definizioni:
  - `char str[100];`

    RISERVA spazio per contenere i caratteri, è una variabile e il suo contenuto può essere modificato
  - `char *s;`

    NON RISERVA spazio per contenere i caratteri, quindi per essere utilizzata come stringa le si deve assegnare una stringa "vera":
    - *Assegnazione di stringa <u>variabile</u> al puntatore:*
      `s = str;` ← *assegna l'indirizzo di* `str`
      `scanf("%s", s);` → *SÌ*
    - *Assegnazione di stringa <u>costante</u> al puntatore:*
      `s = "ciao";` ← *assegna l'indirizzo di* `"ciao"`
      `scanf("%s", s);` → *NO*

Puntatori e stringhe

- Si considerino i seguenti esempi:
  - `char str[] = "ciao";`
    È l'**inizializzazione** di una **variabile stringa**: il compilatore riserva memoria per `str` e vi copia i caratteri di `"ciao"`; la *stringa costante* `"ciao"` *non esiste in memoria:* è stata usata dal compilatore per inizializzare la stringa `str`, ma esiste in memoria la *stringa variabile* `str`
    `str[0]='m';` → *SÌ*
  - `char *s = "hello";`
    È l'**inizializzazione** di una **variabile puntatore**: il compilatore determina l'indirizzo della *stringa costante* `"hello"` (che *esiste in memoria*) e lo assegna alla variabile puntatore `s`
    `s[0]='b';` → *NO!* `"hello"` *è costante!*

Puntatori e stringhe

- Si considerino i seguenti esempi (*cont.*):
  - `s = "ciao";`
    È l'**assegnazione** a una **variabile puntatore**: il compilatore determina l'indirizzo della *stringa costante* `"salve"` (che esiste in memoria) e lo assegna alla variabile puntatore `s`
    `s[3]='b';` → *NO!* `"ciao"` *è costante!*
  - `s = str;`
    È l'**assegnazione** a una **variabile puntatore**: il compilatore determina l'indirizzo della *stringa variabile* `str` (che esiste in memoria) e lo assegna alla variabile puntatore `s`
    `s[0]='m';` → *SÌ*

Puntatori e stringhe

- Noti i puntatori, si possono completare le informazioni già date sulle funzioni relative alle stringhe aggiungendo quanto segue:
  - le funzioni di copia di stringhe `strcpy`, `strncpy`, `strcat` e `strncat` restituiscono il puntatore alla stringa di destinazione
  - le funzioni di parsing `strchr`, `strrchr`, `strstr`, `strpbrk` e `strtok` restituiscono il puntatore all'oggetto cercato o `NULL` se non lo trovano

Funzioni sui blocchi di byte

- *Blocchi di byte:* sequenze di byte (caratteri) qualsiasi, il carattere di codice ASCII 0 (`'\0'`) è un carattere normale e non viene utilizzato come terminatore (non c'è alcun terminatore)
- Una porzione di memoria (allocata in qualsiasi modo) viene trattata come generico blocco di byte da alcune funzioni contenute in `<string.h>`
- Per riservare una porzione di memoria si può definire una variabile o una stringa o utilizzare la funzione `malloc` (trattata in altre slide)
- Nelle seguenti funzioni, s (*source*) e t (*target*) sono puntatori a `void`

Funzioni sui blocchi di byte

- `memcpy(`t`,`s`,`n`)`
  copia n byte da s a t
- `memmove(`t`,`s`,`n`)`
  copia n byte da s a t
  (i blocchi possono anche essere sovrapposti)
- `memcmp(`s`,`t`,`n`)`
  confronta *byte per byte* secondo il codice ASCII i primi n byte di s e di t, il valore restituito è un `int` come quello della `strcmp`
- `memchr(`s`,`c`,`n`)`
  cerca il byte c tra i primi n byte di s, dà `NULL` se non lo trova o il puntatore ad esso se trova
- `memset(`s`,`c`,`n`)`
  copia il byte c in tutti i primi n byte di s

Funzioni sui blocchi di byte

- Queste funzioni vengono talvolta usate con vettori (anche multidimensionali) e `struct` (argomento trattato in altra sezione) per:
  - copiare velocemente un vettore in un altro
  - azzerare velocemente un vettore
  - confrontare due vettori per verificare se sono uguali o no (ma attenzione alle parti non inizializzate dei vettori e agli *spazi di allineamento* e ai *campi anonimi* delle `struct`)

Puntatori const
Puntatore a oggetto costante

- La keyword `const` usata *prima dell'asterisco* impedisce che possa essere modificato l'*oggetto puntato,* due sintassi equivalenti:
  - `int const *p;`
  - `const int *p;`
- `p` è una variabile e può essere riassegnata:

```
const int x=2, y=3;   → entrambe const
const int *p;         → puntatore-a-costante
p = &x;               → Corretto
p = &y;               → Corretto
*p = 13;              → Errore, *p è costante
```

- `p` è una *variabile* di tipo *puntatore-a-costante*

Puntatori const
Puntatore a oggetto costante

- L'assegnazione
  puntatore-a-variabile = *puntatore-a-costante*
  genera un Errore perché permetterebbe di by-passare la restrizione (`const`)

```
const int x = 12;   → costante
const int *p;       → puntatore-a-costante
int *q;             → puntatore-a-variabile
p = &x;             → Corretto
*p = 5;             → Errore, bypass const su *p
q = &x;             → Errore, bypass const su *p
q = p;              → Errore, bypass const su *p
*q = 5;             → Corretto!
q = &x;             → Errore, bypass const su x
```

Puntatori const
Puntatore a oggetto costante

- L'assegnazione
  puntatore-a-costante = *puntatore-a-variabile*
  è corretta perché è l'oggetto puntato a essere costante, non la variabile puntatore, l'oggetto puntato dal *puntatore-a-costante* non è modif.

```
int z = 12;          → variabile
const int *p;        → puntatore-a-costante
int *q;              → puntatore-a-variabile
q = &z;              → Corretto
p = &z;              → Corretto
p = q;               → Corretto
*p = 24 ;            → Errore, bypass const su *p
```

Puntatori const
Puntatore costante a variabile

- La keyword `const` usata *dopo l'asterisco* impedisce che possa essere modificato il *puntatore* (non può puntare ad altro):
  - `int * const p = &x;`
- I puntatori costanti devono essere inizializzati
- La variabile puntata può essere modificata
-
```
int x, y;
int * const p = &x;
*p = 13;
```
→ *Corretto,* `*p` *è una variabile*

```
p = &y;
```
→ *Errore,* `p` *è una costante*

`p` è una *costante* di tipo *puntatore-a-variabile*

Puntatori const
Puntatore costante a costante

- La keyword `const` usata *prima e dopo l'asterisco* impedisce che possa essere modificato sia il puntatore sia l'oggetto puntato:
  - `int const * const p = &x;`
  - `const int * const p = &x;`
- ```
  int x, y;
  int const * const p = &x;
  *p = 13;
  ```
  → *Errore,* `*p` *è una costante*
  ```
  p = &y;
  ```
  → *Errore,* `p` *è una costante*

  `p` è una *costante* di tipo *puntatore-a-costante*
  ```

Esercizi

1. Si scriva un programma che chieda una stringa all'utente e conti quanti siano i caratteri che la costituiscono, NON si usi la funzione `strlen` della libreria standard.
2. Scrivere un programma che verifichi se la stringa data in input è palindroma o no
3. Scrivere un programma che chieda *n* valori interi (massimo 100), li collochi in un vettore e inverta il vettore (scambiando il primo elemento con l'ultimo, il secondo con il penultimo, etc.). Si usi la notazione vettoriale.
4. Come il precedente, ma si usino i puntatori.

Esercizi

5. Scrivere un programma che data una stringa in input dica se la stessa contiene almeno una 'A' tra i primi 10 caratteri.
6. Si scriva un programma che chieda all'utente 2 stringhe e concateni la seconda alla fine della prima, NON si usi la funzione `strcat` della libreria standard, si usino i puntatori e non la notazione vettoriale. Attenzione a terminarla con il carattere '`\0`'.
7. Si scriva un programma che chieda all'utente 2 stringhe e indichi se la seconda è uguale alla parte terminale della prima.

Esercizi

8. Scrivere un programma che chieda *N* valori (massimo 100), li collochi in un vettore e li ordini in senso crescente (senza usare vettori ausiliari).
9. Scrivere un programma che verifichi se la stringa data è composta di due parti uguali, trascurando il carattere centrale se la lunghezza è dispari (es. “CiaoCiao”, “CiaoXCiao”).
10. Scrivere un programma che date 2 stringhe in input indichi quante volte la più corta è contenuta nella più lunga.

Esercizi

11. Scrivere un programma che legga da un file un testo e spezzi su più righe quelle più lunghe di N caratteri (N chiesto all'utente). Le righe si possono spezzare solo dove c'è uno spazio (che non va riportato nella riga successiva). L'output deve essere salvato in un altro file.

Vettori di puntatori

- La riga seguente definisce un vettore di 10 puntatori a `int` (le `[]` hanno priorità maggiore dell'operatore `*`):
  `int *vett[10];`
  Il tipo di `vett` è: `int *[10]`
- Mentre un puntatore a un vettore di 10 `int` si definisce come:
  `int (*vett)[10];`
  ed è quindi assegnabile con un valore dello stesso tipo:
  `int mx[5][10];`
  `vett = mx;` → *vedere "puntatori e matrici"*
  Il tipo di `vett` è: `int (*)[10]`

Vettori di puntatori

- Gli inizializzatori dei vettori devono essere quantità costanti, note prima che il programma inizi l'esecuzione (*load-time*), tra queste ci sono i NULL, i letterali stringa (stringhe tra doppi apici), gli indirizzi di variabili *statiche* (NON quelle automatiche)

Vettori di puntatori

- Esempio di inizializzazione con assegnazione:

```
int a, b, c;
int *vett[10]={NULL};
vett[0]=&a;
vett[1]=&b;
vett[2]=&c;
```

I valori da `vett[3]` a `vett[9]` sono tutti `NULL` in quanto è stato inizializzato il primo elemento (i successivi sono automaticamente a 0 che viene convertito in `NULL`)

Vettori di puntatori

- Esempio di inizializzazione errata

```
int a, b, c;
int *vett[10]={&a, &b, &c};
```

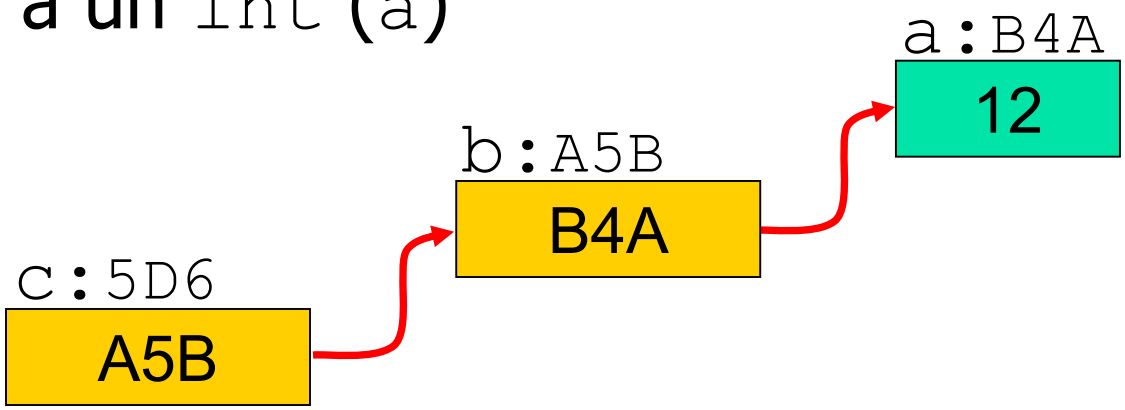
È errato perché in questo esempio gli inizializzatori sono indirizzi di *variabili non statiche* (concetto descritto in altre slide)

Puntatori a puntatori

- Variabili che contengono l'indirizzo di memoria di una variabile puntatore
- Esempio

```
int a, *b, **c;
a = 12;
b = &a;
c = &b;
```

Il puntatore `c` punta a una variabile (`b`) che punta a un `int` (`a`)



Puntatori a puntatori

- Esempio

```
int a=10, b=20, c=30;
int *v[3], int **w;
v[0]=&a; v[1]=&b; v[2]=&c;
w = v; /* oppure w = &v[0]; */
```

- `v` è un *vettore-di-*`(`*puntatore-a-*`int)`, che decade a *puntatore-a-puntatore-a-*`int`
- `w` è invece *una variabile puntatore* (che punta a un altro puntatore), quindi il suo tipo è *puntatore-a-puntatore-a-*`int`

Puntatori e matrici

- Una matrice è un *vettore di vettori* e quindi, considerando che l'associatività di `[]` è da sinistra a destra, si ha che:
  `int Mx[7][5];`
  definisce `Mx` come vettore di 7 elementi ciascuno delle quali è un vettore di 5 `int`
- Gli elementi del *vettore* `Mx` sono i 7 vettori identificati da `Mx[0]`… `Mx[6]`, questi sono indirizzi di memoria, in particolare `Mx[`*i*`]` equivale a `&Mx[`*i*`][0]`
- Gli elementi di ciascun `Mx[`*i*`]` sono i 5 `int` identificati da `Mx[`*i*`][0]` … `Mx[`*i*`][4]`

Puntatori e matrici

- **Digressione**
- Si può spiegare matematicamente perché `Mx[i]` equivale a `&Mx[i][0]`
- Partendo dall'equivalenza:

  `v``[i] ≡ *(`**`v`**`+i)`

  considerando una matrice `mx` si può dedurre:

  `mx[i][j] ≡ *`**`(mx+i)`**`[j] ≡ *(*`**`(mx+i)`**`+j)`

  quindi:

  `&mx[i][j] ≡ &*(*(mx+i)+j) ≡ *(mx+i)+j`

  In particolare:

  `&mx[i][0] ≡ *(mx+i)+0 ≡ mx[i]`

Puntatori e matrici

- Poiché in seguito al decadimento in un'espress. il nome di un *vettore-di-T* diventa di tipo *puntatore-a-T*, anche gli `Mx[`*i*`]` non sono di tipo *vettore-di-*`int` ma decadono al tipo *puntatore-a-*`int` (inoltre, essendo indirizzi, non si può assegnare loro un valore)
- Il "decadimento" a puntatore per il nome di un vettore può avvenire una sola volta, quindi il tipo di una matrice decade a *puntatore-a-vettore-di-T* e non a *puntatore-a-T* , in questo esempio quindi incrementando `Mx` si punta al successivo vettore di 5 `int`

Puntatori e matrici

- `Mx` non è di tipo *puntatore-a-*`int` (`int *`), quindi `int *p = Mx;` è sbagliato
- `Mx` non è di tipo *puntatore-a-puntatore-a-*`int` (`int **`), quindi `int **p = Mx;` è sbagliato
- `Mx` è di tipo *puntatore-a-vettore-di-5-*`int` (`int (*)[5]`), quindi `int (*p)[5] = Mx;` è corretto e `p` può essere usato al posto di `Mx`
- Le parentesi `()` sono necessarie perché le `[]` hanno priorità maggiore di `*` e `int *p[5]` è invece un vettore di 5 puntatori a `int`

Puntatori e matrici

- `int (*p)[5];`
  definisce `p` come *puntatore-a-vettore-di-5-*`int`, è necessario che la dimensione delle colonne (5) sia specificata perché altrimenti la dimensione del vettore puntato non è nota
- Poiché `Mx` è un *puntatore-a-vettore-di-5-*`int` (e non un puntatore a un intero), allora: `Mx+1` punta all'elemento successivo, ma l'elemento qui è un vettore di 5 interi (la seconda riga della matrice): `Mx+1` aggiunge all'indirizzo di `Mx` il numero corretto di byte per puntare al secondo vettore di 5 `int`

Puntatori e matrici

- Ricapitolando

```
int Mx[7][5];
```

  - `Mx[i][j]`
    - è un valore scalare
    - è di **tipo** `int`
    - è modificabile
    - `Mx[i][j]+1` somma 1 al contenuto di `Mx[i][j]`
  - `Mx[i]`
    - è l'indirizzo di un *vettore-di-5-*`int`
    - è di **tipo** *puntatore-a-*`int` ("decade"): `int *`
    - non è modificabile
    - `Mx[i]+1` punta a `Mx[i][1]` (`*(Mx[i]+1)` corrisponde a `Mx[i][1]`)

Puntatori e matrici

- Ricapitolando (*continuazione*):
  - `Mx`
    - è l'indirizzo di un *vettore-di-7-vettori-di-5-*`int`
    - è di **tipo** *puntatore-a-vettore-di-*`int` (ossia `int (*)[]`) in quanto decade una volta sola, più precisam. è un *puntatore-a-vettore-di-5-*`int` (`int (*)[5]`, senza il 5 nel tipo il compilatore non saprebbe di quanti byte spostarsi per puntare al *vettore-di-5-*`int` successivo quando si scrive `Mx+1`)
    - non è modificabile
    - `Mx+i` corrisponde all'indirizzo di memoria di `Mx[i]` ossia il vettore di 5 `int` di indice `i`

Esempio di uso di matrici

- Azzerare la colonna `k` di una matrice:
  - Con i vettori:

```
int riga;
for (riga=0; riga<RIGHE; riga++)
  vett[riga][k] = 0;
```

  - Con i puntatori:

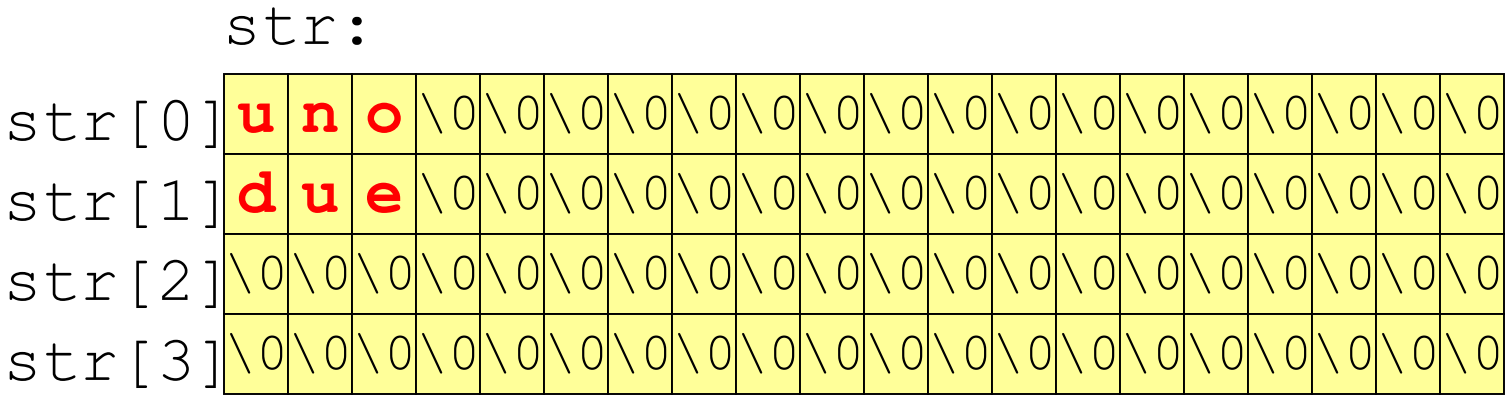
```
int (*p)[COLONNE];
for (p=Mx; p<Mx+RIGHE; p++)
  (*p)[k]=0;
```

Vettori di stringhe

- Essendo una stringa un vettore di caratteri, un vettore di stringhe è in realtà un vettore di vettori di caratteri, cioè una matrice di `char`

  `char str[4][20]={"uno", "due"};`
  definisce un vettore di 4 stringhe di 20 `char`

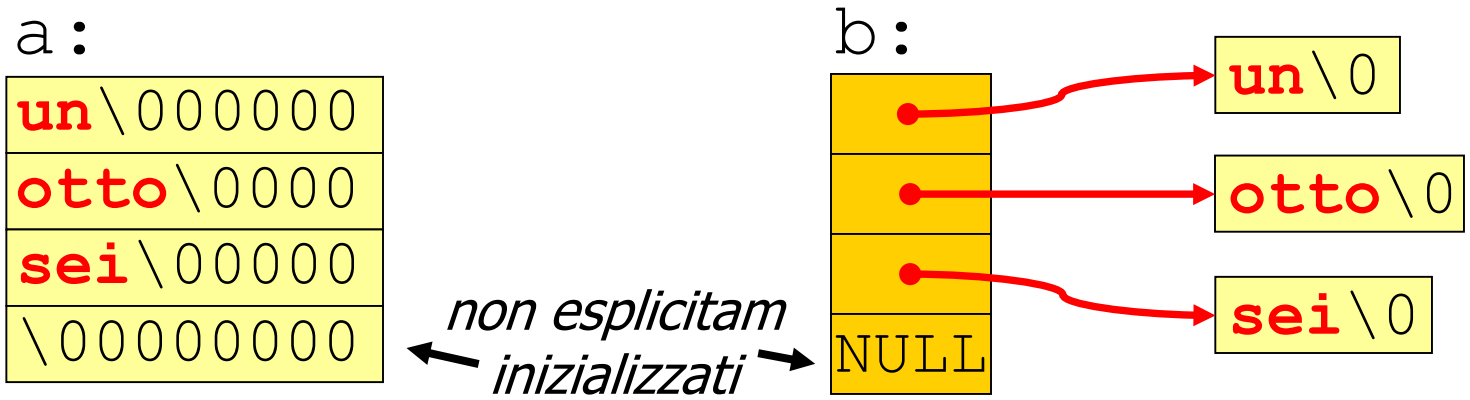
  Le 4 stringhe sono identificate da `str[`*i*`]`



Vettori di puntatori e matrici

- Si notino le differenze tra:

```
char a[4][8]={"un","otto","sei"};
char *b[4]={"un","otto","sei"};
```



- `a` è un vettore di 4 vettori di 8 `char`
- `b` è un vettore di 4 puntatori a stringhe costanti di lunghezza diversa (*jagged array* – "frastagliato" o *ragged array* – "irregolare")

Vettori di puntatori e matrici

- `a[`i`]` è l'*indirizzo di memoria* (costante) della riga i di `a`, tale riga è una stringa *variabile* di 8 `char`
- `b[`i`]` è una *variabile puntatore* contenente l'indirizzo costante della riga i, che qui è un letterale stringa.
  È possibile assegnare a `b[`i`]` un qualsiasi altro puntatore a carattere:

```
char riga[N] = "Ciao";
b[0] = riga;
b[1] = &riga[2];
b[2] = "Buongiorno a tutti";
```

Vettori di puntatori e matrici

- `a[2] = "hello";`
  ERRORE: `a[2]` non è un puntatore
- `strcpy(a[2], "hello");`
  CORRETTO
- `b[2] = "hello";`
  CORRETTO: `b[2]` è una *variabile* puntatore a cui viene assegnato l'indirizzo di memoria di una stringa *costante*
- `strcpy(b[2], "hello");`
  ERRORE: `b[2]` punta a una stringa costante
- Entrambi `a[1][0]` e `b[1][0]` sono il carattere `'c'`
  - `a[1][0]='m';` → *SÌ! L'oggetto puntato da* `a[1]` *è una stringa variabile*
  - `b[1][0]='m';` → *NO! L'oggetto puntato da* `b[1]` *è una stringa costante*

Const per puntatori a puntatori

- Si può considerare come regola che la keyword `const` si riferisca al solo primo elemento alla sua destra (a inizio definizione però `const` e il nome del tipo possono essere invertiti, quindi la si può considerare a destra)

1. `int **x;`
   `x` è una *variabile* di tipo puntatore a…
   … un puntatore *variabile* a…
   … una *variabile* di tipo `int`
2. **`const int`** `**x;`
   `int const` `**x;`
   `x` è una *variabile* di tipo puntatore a…
   … un puntatore *variabile* a…
   … una *costante* di tipo `int`

Const per puntatori a puntatori

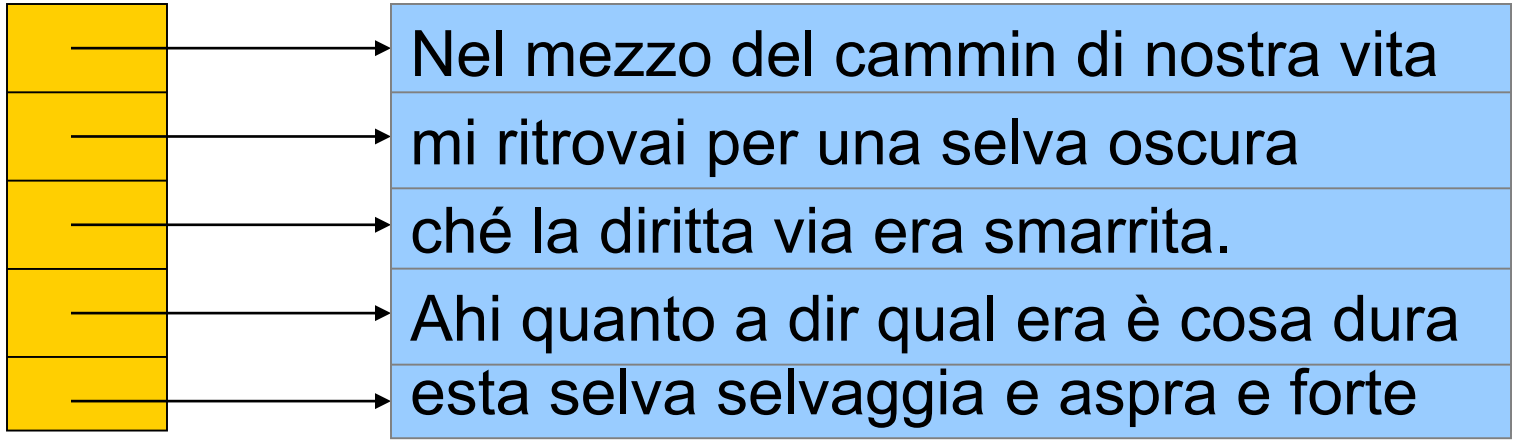
3. `int * * const x;`
   `x` è una *costante* di tipo puntatore a…
   … un puntatore *variabile* a…
   … una *variabile* di tipo `int`
4. `int * const * x;`
   `x` è una *variabile* di tipo puntatore a…
   … un puntatore *costante* a…
   … una *variabile* di tipo `int`
5. `const int * const * x;`
   `x` è una *variabile* di tipo puntatore a…
   … un puntatore *costante* a…
   … una *costante* di tipo `int`
6. `const int * const * const x;`
   `int const * const * const x;`
   `x` è una *costante* di tipo puntatore a…
   … un puntatore *costante* a
   … una *costante* di tipo `int`

Esercizi

12. Scrivere un programma che legga da un file al massimo un certo numero MAXRIGHE di righe di testo e le memorizzi in una matrice di caratteri (MAXRIGHE x 100), una riga del file per ciascuna riga della matrice.
    Si definisca un vettore di puntatori a carattere e lo si inizializzi in modo che il primo puntatore punti alla prima riga, il secondo alla seconda, ecc. Si ordinino le stringhe scambiando tra di loro solo i puntatori e le si visualizzino ordinate.

Esercizi (seguito)

prima



dopo