Puntatori

Maria Rita Di Berardini

Dipartimento di Matematica e Informatica
Universitá di Camerino
mariarita.diberardini@unicam.it

Definizione di puntatore

- Ad ogni variabile corrisponde un nome ed una locazione di memoria
- Il nome di una variabile è il simbolo attraverso il quale facciamo riferimento al contenuto della corrispondente locazione di memoria

```
int a;
a = 5;   scrittura
printf(``%d'', a); lettura
```

- Un indirizzo di memoria può essere assegnato solo ad una categoria speciale di variabili, dette **puntatori**
- Una variabile di tipo puntatore può contenere l'indirizzo di memoria memoria di un'altra variabile
- Si possono manipolare sia il puntatore che la variabile puntata (cioè la variabile memorizzata a quell'indirizzo di memoria).

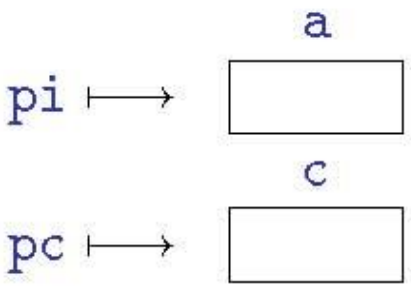
Dichiarazioni di puntatori

- Sintassi della dichiarazione di un puntatore:

```
tipo* nome_variabile;
tipo *nome_variabile;
```

- Introduciamo una nuova variabile in grado di contenere indirizzi di memoria di variabili di tipo `tipo`
- Possiamo inizializzare un variabile puntatore utilizzando l'operatore `&` che restituisce l'indirizzo di una data variabile

```
int a;
char c;
int* pi = &a;
char* pc = &c;
```



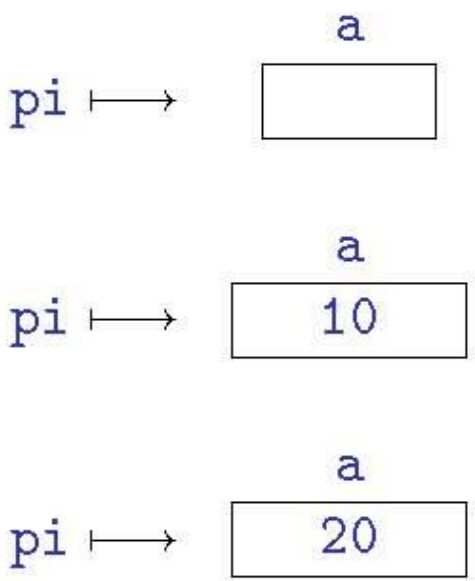
Operatore di indirezione

- Un operatore fondamentale la gestione dei puntatori è il cosiddetto operatore di **indirezione** o **deferenziazione**, l'operatore `*`
- Se `pi` è un puntatore, `*pi` restituisce il contenuto della locazione di memoria puntata da un `pi`

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

a = 10;
printf(``%d\n'', a);

*pi =20;
printf(``%d\n'', a);
```

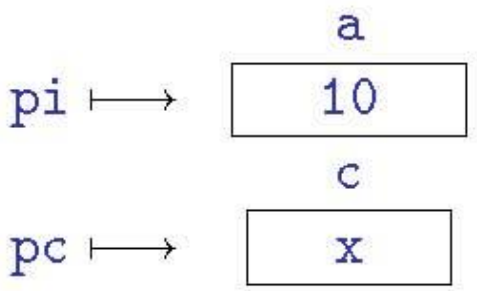


Operatore di indirezione

- Dopo l'assegnamento `pi = &a`, `a` e `*pi` sono degli alias, ossia sono perfettamente equivalenti; emtrambi consentono di accedere alla variabile introdotta dalla dichiarazione `int a;`
- Le seguenti `printf` producono lo stesso risultato: visualizzano la stringa `a = 10 c = x`

```
int a = 10;
char c = 'x';
int *pi = &a;
char *pc = &c;

printf("a=%d c=%c\n", a, c);
printf("a=%d c=%c\n", *pi, *pc);
```

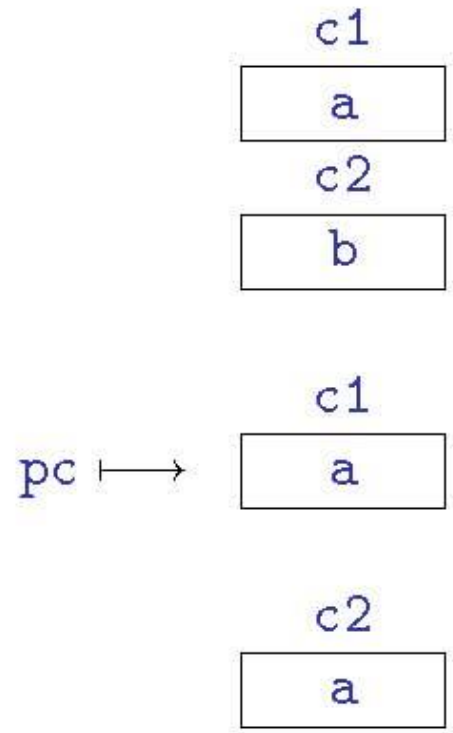


Uso degli operatori & e *

- Quando un puntatore viene dichiarato non punta a nulla; assume un valore speciale NULL
- Prima di usare un puntatore bisogna assicurarsi che punti ad una qualche locazione di memoria

```
char c1, c2;
char *pc;
c1 = 'a';
c2 = 'b';

// *pc = 20;    errore
pc = &c1;
c2 = *pc;
```



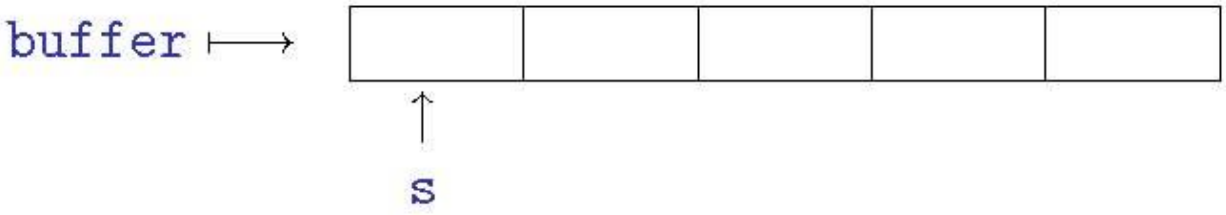
Array e puntatori

- Il **nome** di un array è il **puntatore** alla locazione di memoria che contiene il suo primo elemento
- Supponiamo di aver dichiarato le seguenti variabili:

```
char buffer[5];
char *s;
```

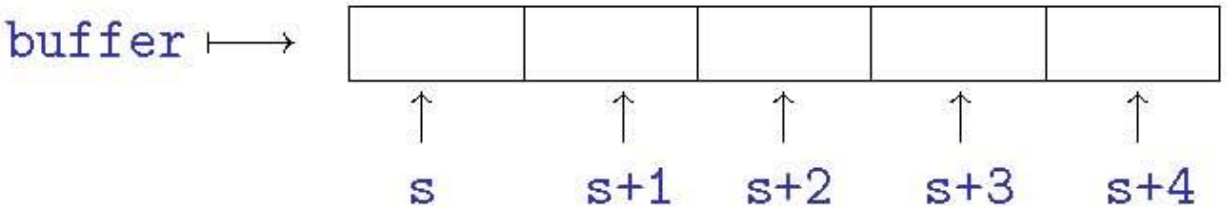
i due seguenti assegnamenti sono perfettamente equivalenti

```
s = &buffer[0];
s = buffer;
```



Array e puntatori

- Abbiamo visto come gli elementi di un array possono essere scanditi tramite un indice
- Si può avere accesso agli stessi elementi tramite un puntatore
- Se `s` è il puntatore alla locazione di memoria contenente il primo elemento elemento di un array, allora `(s + j)` è il puntatore alla locazione contenente il suo `(j+1)`-esimo elemento




```
buffer[2] = 'a';   è equivalente a
*(s+2) = 'a';
```

Array e puntatori

- Per queste ragioni, dopo l'assegnamento `s = buffer;` il seguente frammento di programma

```
for(i=0; i<5; i++)
    buffer[i] = i;
```

è equivalente a

```
for(i=0; i<5; i++){
    *s = i;          oppure s[i] = i;
    s = s + 1;
}
```

- In entrambi i casi

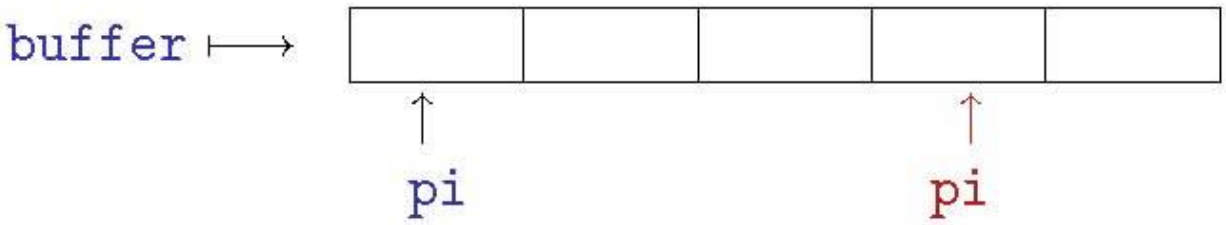
Aritmetica dei puntatori

- Un puntatore contiene un indirizzo di memoria e le operazioni che possiamo eseguire su di essi sono quelle che hanno un senso per un indirizzo:
- Le possibili operazioni sono l'incremento (per passare da un indirizzo più basso ad uno più alto) ed il decremento (per passare da un indirizzo più alto ad uno più basso)
- Gli unici operatori ammessi per i puntatori sono, di conseguenza, `+` e `++`, `-` e `--`
- Qual'è il risultato di `p++` ?? Dipende, perchè il valore di un puntatore non viene icrementato o decrementato come una qualunque costante numerica
- Se per esempio, `pc` vale 10 e `pc` è stato dichiarato come un puntatore a carattere non è affatto detto che `pc++` valga 11, quello che conta è il tipo del puntatore

Incremento

- Incrementare di uno un il valore di un puntatore significa aggiungere al valore del puntatore tanti byte quanti sono quelli usati per rappresentare il tipo del puntatore (1 per i caratteri, 4 per un intero o un float, ...)
- Ad esempio, l'assegnamento `(1)` sposta il puntatore `pi` in avanti di tre posizioni

```
int a[5], *pi;
pi = a;
pi = pi + 3;    (1)
```



Decremento

- In maniera analoga decrementare di uno un il valore di un puntatore significa sottrarre al valore del puntatore tanti byte quanti sono quelli usati per rappresentare il tipo del puntatore (1 per i caratteri, 4 per un intero o un float, ...)
- In generale, se `p` è un puntatore che punta ad un dato elemento di un vettore, `p + 1` significa "prossimo elemento", mentre `p - 1` significa "elemento successivo"
- L'assegnamento `(1)` sposta il puntatore `pi` indietro di tre posizioni

```
int a[5], *pi;
pi = &a[4];
pi = pi - 3;     (1)
```

Usate questi operatori con attenzione

- Assumiamo di aver dichiarato le seguenti variabili

```
int v1[10], v2[10];
int* p;
int i;
```

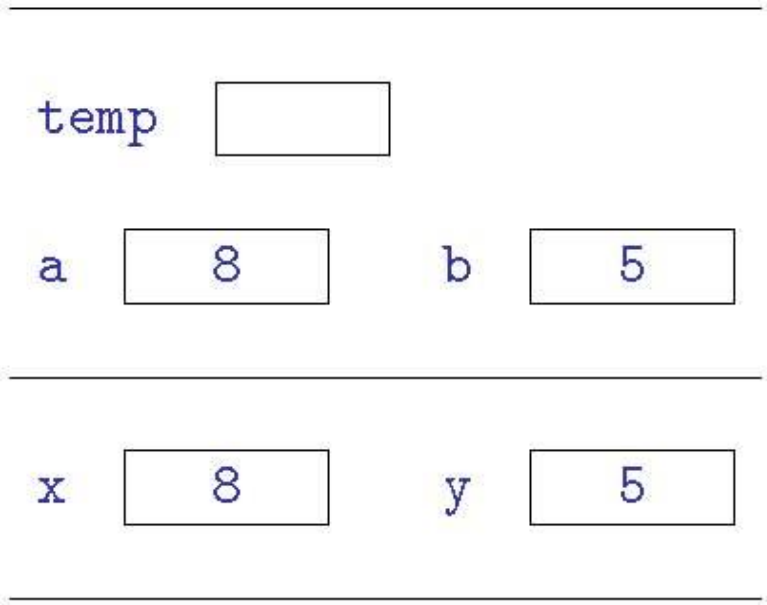
- Dopo l'assegnato `i = &v1[5] - &v1[3]` la variabile `i` vale 2
- Il risultato dell'assegnato `i = &v2[5] - &v1[3]` non è prevedibile
- Discorso simile per l'assegnamento `p = v2 - 2`

La funzione "scambia" con passaggio per valore

```
int x = 8, y = 5;
scambia(x, y);

void scambia(int a, int b)
{
    int temp;

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

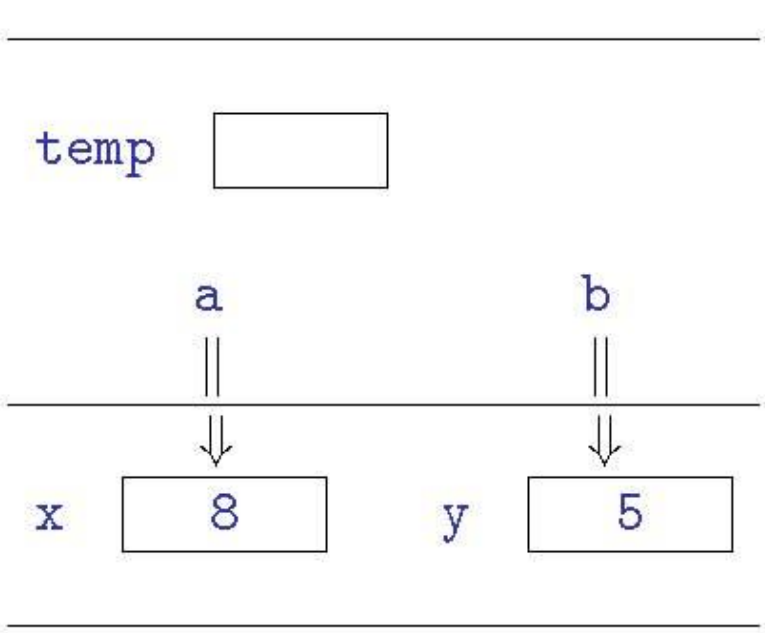


La funzione "scambia" con parametri di tipo puntatore

```
int x = 8, y = 5;
scambia(&x, &y);

void scambia(int* a, int* b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```



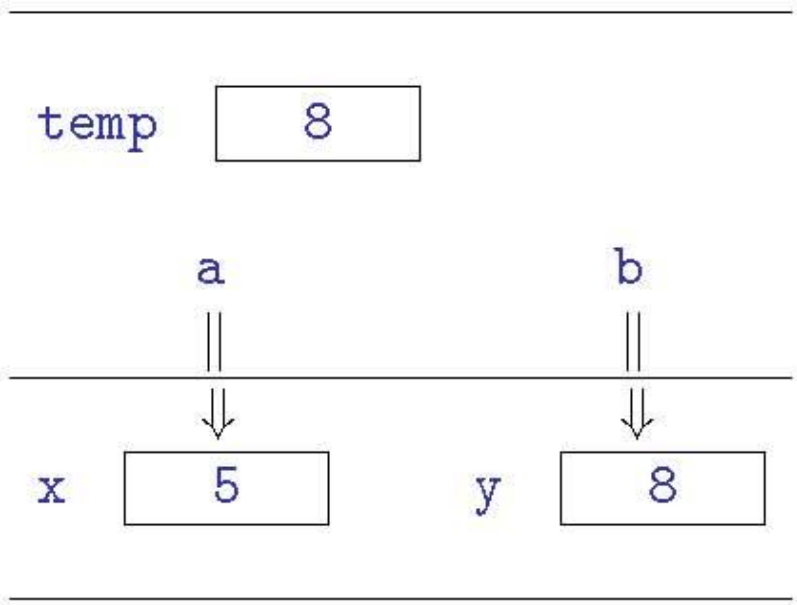
- Effuttuare il passaggio dei parametri equivale ad effettuare gli assegnamenti: `a = &x` e `b = &y`
- Modificare il contenuto delle locazioni puntate dai puntatori `a` e `b` significa chiaramente modificare i valori associati alle variabili `x` e `y`

La funzione "scambia" con parametri di tipo puntatore

```
int x = 8, y = 5;
scambia(&x, &y);

void scambia(int* a, int* b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```



Passaggio di parametri per variabile

- L'utilizzo di puntatori e dell'operatore di indirezione consente di simulare il passaggio di parametri per riferimento (passaggio per variabile)
- Non passiamo più il valore, ma l'indirizzo del parametro (tipicamente una variabile) in modo che tale le istruzioni all'interno della funzione possano modificare il suo contenuto

Passaggio di array ad una funzione

```
#include <stdio.h>

int my_strlen(char*);

main()
{
    printf(``La stringa %s ha lunghezza '', str);
    printf(``%d\n'', my_strlen(str));
}

int my_strlen(char* s)
{
    int i = 0;
    while (s[i] != `\0')      i++;
    return i;
}
```

Problemi con i puntatori

- I puntatori forniscono immense potenzilità e sono indispensabili per molti programmi (ad esempio, quelli che richiedono gestione dinamica della memoria);
- Allo stesso tempo, quando un puntatore contiene un indirizzo sbagliato, può originare dei bug difficili da individuare
- Il problema non risiede tanto nel puntatore quanto nel uso che ne facciamo: lettura/scrittura di una locazione di memoria
- Se l'operazione è di lettura, la cosa peggiore che possa capitare è leggere valori senza senso
- Se l'operazione è di scrittura, si potrebbe tentare di scrivere porzioni di memoria non accessibili, oppure locazioni di memoria riservati ad altri dati con risultati del tutto imprevedibili e, soprattuto, difficili da scovare

Errori più comuni

- Uso di puntatori non inizializzati o inizializzati "male": tentativo di accedere a porzioni di memoria sconosciute

```
int x, *pt;
x = 10;
*pt = 20;
```

```
int x, *pt;
x = 10;
pt = x;
printf(``%d\n'', *pt);
```

- Assunzioni errate riguardo posizioni di variabili in memoria: non sappiamo dove i dati si troveranno in memoria, se successivamente verrano posizionati nello stesso luogo, o se ogni compilatore li tratterà allo stesso modo

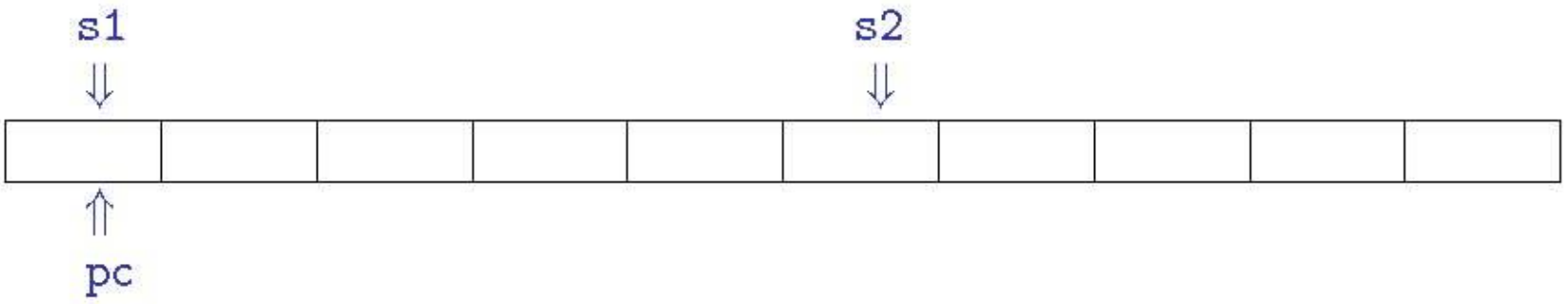
```
char s1[5], s2[5];
char *p1 = s1, *p2 = s2;
if (p1 < p2)  printf(``p1 < p2\n'');
else  printf(``p1 >= p2\n'');
```

Errori più comuni

- Assunzioni riguardo posizioni in memoria di array "adiacenti" (simile al precedente)

```
int s1[5], s2[5];
char *pc = s1;

for (i=0; i < 10; i++)    *pc++=i;
```



Un programma con un errore

```
#include <stdio.h>

main()
{
    char str[80], *pt;

    pt = str;
    do {
        printf(''Inserisci stringa:  '');
        scanf(''%s'', str);
        while(*pt)   printf(''%d'', *pt++);
        //stampa l'equivalente decimale di ogni carattere
    } while (strcmp(s, ''fine''));
}
```

La sua versione corretta

```
#include <stdio.h>

main()
{
    char str[80], *pt;

    do {
        printf(''Inserisci stringa:  '');
        scanf(''%s'', str);
        pt = str;
        while(*pt)   printf(''%d'', *pt++);
        //stampa l'equivalente decimale di ogni carattere
    } while (strcmp(s, ''fine''));
}
```

Funzioni che restituiscono puntatori

```
char* ch_first(char* str, char ch)
//restituisce il puntatore alla prima occorrenza
//di ch in str
{
    int i;

    for (i=0; str[i] !='\0'; i++)
        if(str[i] == ch)
            return (&str[i]);

    return NULL;
}
```

Funzioni che restituiscono puntatori

```
char* ch_first(char str[] *, char ch)
//restituisce il puntatore alla prima occorrenza
//di ch in str
{
    int i;

    for (i=0; !str[i] **; i++)
        if(str[i] == ch)
            return (&str[i]);

    return NULL;
}
```

* Un modo diverso per indicare che il parametro `str` è un puntatore a caratteri

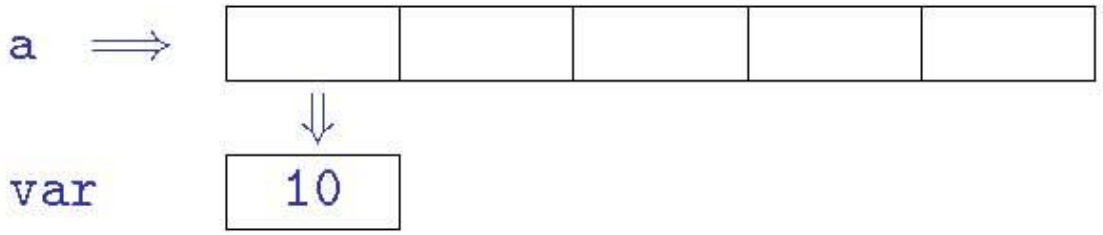
** Un modo diverso, equivalente a `str[i] !='\0'`, per indicare la condizione di uscita del for (il valore di `\0'` è zero)

Array di puntatori

- I puntatori possono essere disposti su un array come qualsiasi altri tipo di dato

```
int* a[5], x = 10;
a[0] = &x
printf(``%d\n'', *a[0]);   *a[0] ≡ **a valore di x
```

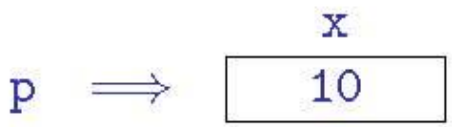
- il nome dell'array è il puntatore alla locazione contenente il primo elemento dell'array, e quindi è **un puntatore a puntatori** di interi



Indirettezza singola e multipla

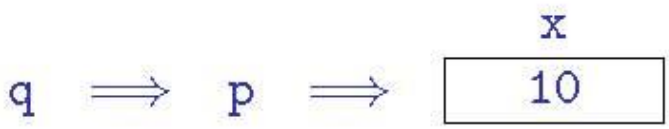
- Indirettezza singola: il puntatore contiene l'indirizzo dell'oggetto (variabile) a cui vogliamo accedere

```
int x = 10; *p = &x;
accediamo al valore di x tramite *p
```



- Indirettezza multipla: il puntatore contiene l'indirizzo di un altro puntatore che a sua volta contiene l'indirizzo dell'oggetto a cui vogliamo accedere

```
int x = 10; *p = &x, **q =&p;
printf(''%d\n'', **q);
accediamo al valore di x tramite **q
```



Array di stringhe

- Gli array di puntatori vengono spesso usati per rappresentare array di stringhe

```
void syntax_error(int num)
{
    char* err[] = {
            ``Impossibile aprire il file'',
            ``Errore in lettura'',
            ``Errore in scrittura'',
            ``Guasto al dispositivo''
            };

    if (num < 4) printf(``%s\n'', err[num]);
    else printf(``%d non valido'', num);
}
```

Allocazione dinamica della memoria

- I puntatori forniscono il supporto necessario per il potente sistema di allocazione dinamica del C
- Con il termine "allocazione dinamica" ci si riferisce al modo in cui un programma può ottenere della memoria durante la sua esecuzione
- Il C fornisce due funzioni di allocazione dinamica della memoria (la `malloc` e la `calloc`) ed una funzione di deallocazione (`free`)
- La memoria necessaria per gestire dati globali è allocata staticamente dal compilatore
- La memoria necessaria per variabili locali e gestione delle chiamate di funzione viene allocata dinamicamente e gestita come uno stack
- Una terza regione di memoria (l'`heap`) è riservata per la gestione della memoria allocata (dinamicamente) tramite le funzioni malloc e calloc

Funzioni di allocazione dinamica

- Le funzioni `malloc` e la `calloc` sono contenute nella libreria standard `stdlib.h`

```
void* malloc(int num_bytes);
num_bytes:  quantità di memoria da allocare

void* calloc(int num_elementi, int num_bytes)
num_elementi:  numero di elementi
num_bytes:  memoria per ogni elemento
```

- Entrambe restituiscono un puntatore di tipo `void*` che può essere assegnato ad ogni tipo di puntatore
- Se la memoria disponibile è sufficiente per soddisfare le richieste di memoria, la chiamata termina restituendo il puntatore alla prima locazione di memoria allocata
- In caso contrario (l'heap non è infinito), si verifica un errore di allocazione e viene restituito il puntatore nullo (NULL)

Funzioni di allocazione dinamica

- Il seguente esempio consente di allocare 1000 bytes contigui

```
char* p;
p = malloc(1000);
```

- Il seguente esempio alloca spazio per 50 interi; l'uso dell'operatore `sizeof` garantisce la trasportabilità

```
int* p;
p = malloc(50*sizeof(int));
p = calloc(50, sizeof(int));
```

- Bisogna controllare che il puntatore restituito da una malloc/calloc non sia nullo, questo perchè il riferimento ad un puntatore nullo provoca quasi certamente il blocco del programma

```
int* p;
p = malloc(50*sizeof(int));
if (p) { uso di p };
```

Funzioni di allocazione dinamica

- Le funzioni `malloc` e `calloc` ritornano un puntatore all'oggetto dinamico creato
- In realtà, queste funzioni ritornano dei puntatori a `void`, cioè ad un tipo generico
- Il tipo del puntatore può essere esplicitamente convertito mediante un operazione di casting

```
int* p;
p = (int*) malloc(50*sizeof(int));
```

- In questo modo forziamo la restituzione di un puntatore ad intero

La funzione `free`

- La funzione `free` consente di deallocare memoria precedentemente allocata tramite una malloc/calloc

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

- È necessario richiamare **sempre** la funzione free con un argomento valido (un puntatore restituito da malloc/calloc)
- Bisogna fare molte attenzione ad evitare errori del tipo "riferire un puntatore ad una locazione già deallocata"
- Alcuni linguaggi (Lisp, Perl, Java, ...) hanno dei meccanismi di recupero della memoria detti garbage collector (lett. spazzini)
- In C un programmatore deve raccogliere la propria spazzatura da solo