Indirizzi e puntatori

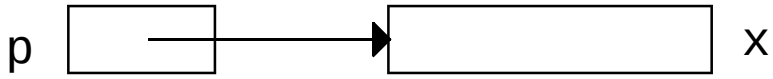
Ogni variabile occupa una regione di memoria con un certo indirizzo.

Se `x` è una variabile, `&x` rappresenta l'indirizzo di `x`.

Un puntatore è una variabile che ha per valore un indirizzo.

Dunque, se `p` è un puntatore opportuno, `p=&x` assegna a `p` l'indirizzo di `x`:

Ma in C un indirizzo non è mai noto esplicitamente, in valore; dunque una rappresentazione grafica più appropriata è:



che si descrive appunto dicendo che `p` punta a `x`.

Se `p` è un puntatore a un oggetto, `*p` rappresenta questo oggetto.

Esempio: dati un oggetto `x` e un puntatore `p`, le istruzioni

```
z = x;
x = 3;
```

equivalgono a:

```
p = &x;
z = *p;
*p = 3;
```

Ciò illustra come `*p` può comparire sia a destra che a sinistra dell'assegnazione, esattamente come una variabile.

`*p` si legge: "`p` indiretto" o meglio "oggetto puntato da `p`".

L'operatore * è detto operatore di dereferenziazione.

Puntatori

Ogni puntatore può puntare ad oggetti di un solo tipo.
Questo viene specificato nella dichiarazione della var puntatore:

`int *p` dichiara che `p` è un puntatore ad interi

Per ricordarlo: osservare che:

`*p` si legge: "l'oggetto puntato da `p`", e

`int x;` si legge: "`x` è un intero"

Con queste convenzioni, dunque:

`int *p` si legge: "l'oggetto puntato da `p` è un intero"

cioè: "`p` è un puntatore ad interi", come desiderato

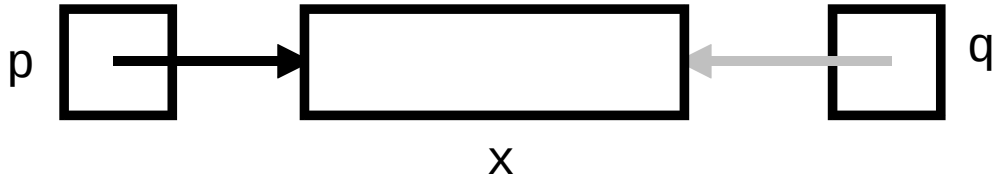
Intercambiabilità

```
p = &x;    x e *p diventano intercambiabili
y = *p+1   mette y a x+1
printf("%d\n", *p)   stampa il valore di x
d = sqrt((double) *p)   d diventa la radice di x convertito in double
*p = 0 mette x a 0
(*p)++ incrementa x
```

Assegnazione tra puntatori

```
int x=2, *p, *q;
p = &x;
q = p;
```

`q=p` pone in `q` l'indirizzo che è in `p`, cioè fa puntare `p` e `q`
allo stesso oggetto (graficamente, fa comparire la freccia grigia):

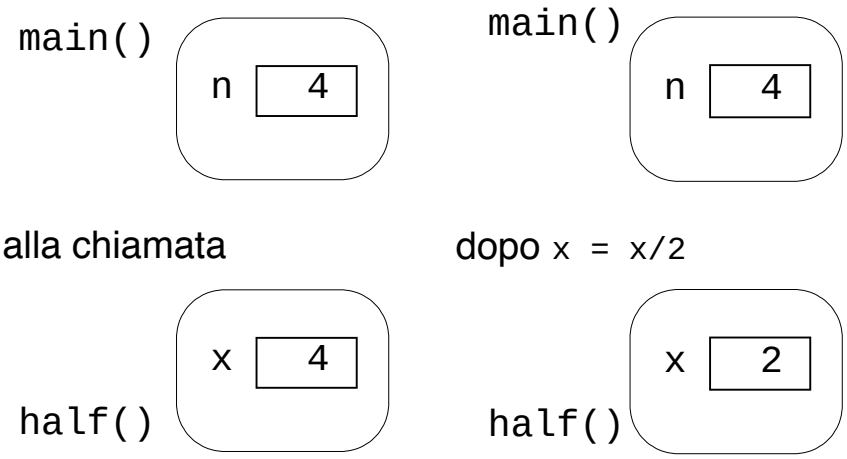


Puntatori come argomenti di funzioni

Gli argomenti vengono passati alle funzioni C per valore:
i parametri sono come variabili locali in cui viene copiato
il valore degli argomenti.

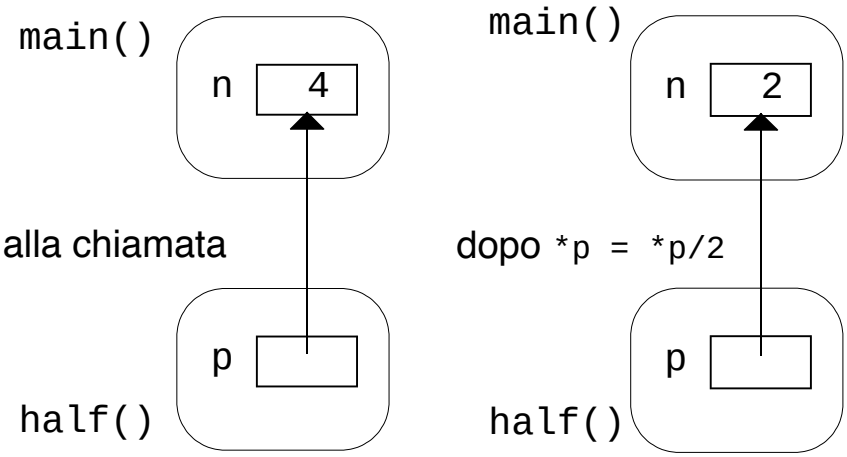
Quindi una chiamata di funzione `half(n)` non può modificare l'argomento, cioè la variabile `n`, anche se lo si desidera:

```
main()
{
    int n = 4;
    half(n);
}


half(int x)
{
    x = x / 2;
}
```



L'effetto voluto si ottiene passando a `half` un puntatore a `n`. Così, `p`, l'argomento di `half`, consente alle istruzioni di `half` (funzione chiamata) di fare riferimento a un oggetto locale al programma chiamante:

```
main()
{
    int n = 4;
    half(&n);
}


half(int *p)
{
    *p = *p / 2;
}
```



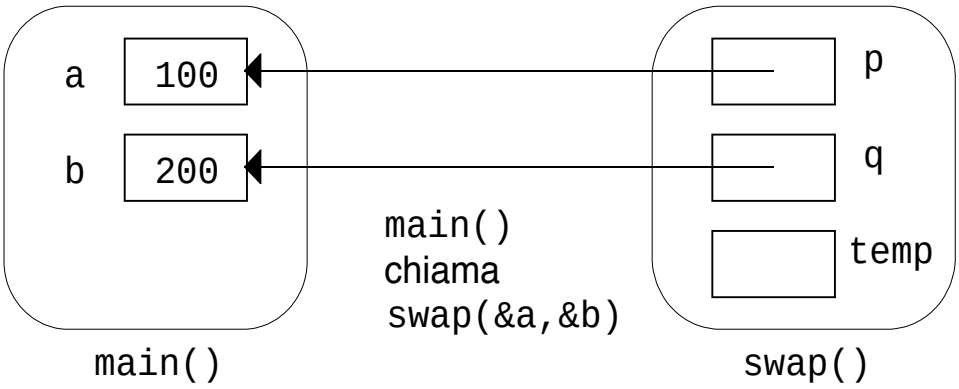
Puntatori come parametri - esempi

Usando puntatori come parametri si può scrivere una funzione `swap` tale che:

```
swap(&a,&b);
```

scambia i contenuti delle variabili `a` e `b`:

```
swap(int *p, *q)
{
    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}
```



Il ricorso a parametri puntatore è necessario quando la funzione deve calcolare più valori; di questi:

- solo uno può essere restituito,
- gli altri vanno copiati dentro variabili della funzione chiamante, il che può essere ottenuto solo attraverso puntatori a queste.

P. es. `getint(&n)` legge un intero dall'input e lo assegna a `n`, inoltre restituisce il segno, se lo trova.

getint

La dichiarazione di `getint` specifica che il suo parametro punta a un intero (che assumerà il valore letto) ed il valore restituito è un carattere (che assumerà il segno del valore inserito).

```
char getint(int * pn)
{
    int s, c, sign;
```

Per saltare eventuali spazi bianchi iniziali:

```
    while ((c = s = getchar()) == ' ' || c == '\n' || c== '\t')
        ;
```

Per determinare il segno del numero da leggere:

```
    sign = 1;
    if ( s == '+' || s == '-') {  /* c'è un segno prima di num. */
        sign = (s=='+') ? 1 : -1;
        c = getchar();
    }
    else s = ' ';
```

Per porre in `*pn` l'intero in input, p.es. `435`:

- si legge una nuova cifra (finché ce ne sono) p. es. `'5'`
- a questo punto, `*pn` dovrebbe valere `43`:
  si moltiplica `*pn` per `10` e si somma `5`, calcolato con `'5'-'0'`:

```
    for (*pn = 0; c >='0' && c <= '9'; c = getchar())
        *pn = 10 * *pn + c-'0';
    *pn *= sign;
```

Alla fine, si ritorna il segno:

```
    return(s);
}
```

Array e puntatori - 1

In C c'è una stretta relazione tra puntatori e array.

Ogni operazione che fa riferimento a indici di un array
può essere riscritta in forma più veloce usando puntatori.

Aritmetica dei puntatori

Se `p` e `q` puntano a due elementi di un array:

- `p+n` punta all'`n`esimo elemento dopo quello puntato da `p`;
  `*(p+n)` si può anche scrivere `p[n]`
- `p-n` punta all'`n`esimo elemento prima di quello puntato da `p`
- `p-q` è il numero di elementi tra quelli puntati da `p` e `q`
- `p<q` è vero se l'elemento puntato da `p` viene prima di quello puntato da `q` nell'array (il test si può fare anche con `<=` `>` `>=`)
- `p` e `q` non vanno combinati se puntano a elementi di array diversi.

P. es., dati

```
int a[10];
p = &a[0];
```

`*(p+1)`, `p[1]` e `a[1]` sono lo stesso oggetto.

Inoltre in C, un nome di array `a` fa da puntatore ad `a[0]`, quindi:

- `a`, `p` e `&a[0]` sono lo stesso puntatore;
- `a+1` e `&a[1]` sono lo stesso puntatore (ad `a[1]`);
- `*(a+1)` e `a[1]` sono lo stesso elemento

Array e puntatori - 2

Le proprietà viste si spiegano secondo principi più generali.

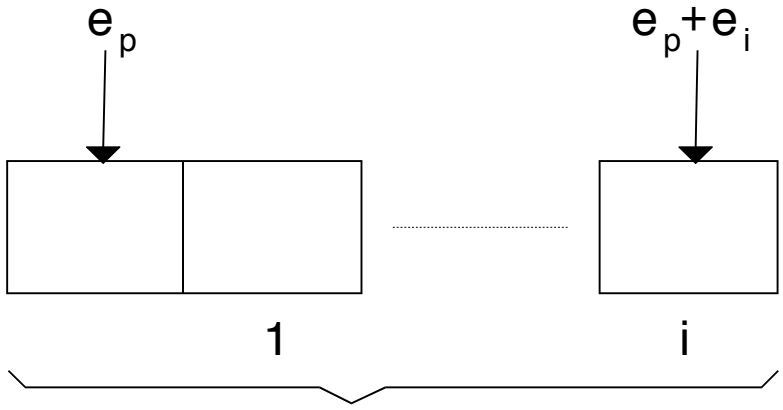
1 Siano:

e_i un'espressione `int` di valore i

e_p un'espressione di tipo puntatore al tipo T e di valore p

S la dimensione in byte di un oggetto di tipo T

allora e_p+e_i e e_i+e_p hanno per valore l'indirizzo p incrementato (in quanto dato binario) di i*S. Quindi:



2 Ogni riferimento a un array `a` (non ai suoi elementi) è tradotto nell'indirizzo di `a[0]`.

Un nome di array è cioè una costante,non una variabile puntatore:

```
int a[10], x, *p;
a++;      /* illegale! */
a = &x;   /* illegale! */
p = &a;   /* illegale!, & si può applicare a un oggetto, non a un indirizzo */
```

3 e_p`[`e_i`]` viene convertito in `*(`e_p+e_i`)`, ed e_i`[`e_p`]` in `*(`e_i+e_p`)`

Quindi, malgrado le apparenze, l'operazione indice è commutativa.

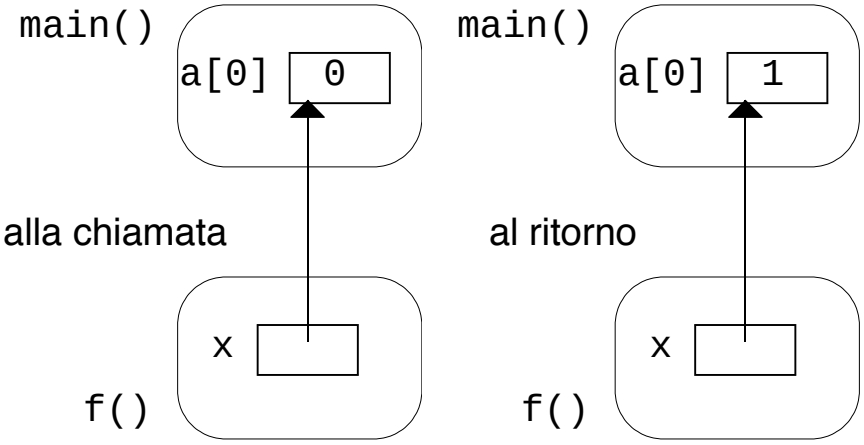
Array e puntatori come parametri

Dato un array `a` di elementi di tipo T, una chiamata `f(a)` passa a `f` il valore dell'argomento `a`, cioè l'indirizzo di `a[0]`, che ha tipo T.

Quindi il parametro `x` di `f` sarà un puntatore a T di valore iniziale `a`.

Così, attraverso `x`, `f` può modificare gli elementi di `a`:

```
main()
{
    int a[1]={0};
    f(a);
}

f(int *x)
{
    x[0] = 1;
}
```



Anche se dichiarato come array, `x` resta un puntatore variabile

Esempio

```
int lung(char x[])     /* restituisce la lunghezza della stringa x */
{
    int n;
    for (n = 0; *x != '\0'; x++)   /* x++ varia l'indirizzo in x */
        n++;
    return(n);
}
```

Altra possibilità: usare la differenza tra puntatori

```
{   char *p = x;
    while (*p != '\0')
        p++;
    return(p-x);
}
```

Pregio del meccanismo: si presta a operare su sottoarray.
P.es. `lung(a+2)` = lunghezza del sottoarray `a[2]`...

Strutture dati

La definizione di una struttura dati (ovvero di un record) si fa attraverso la parola chiave `struct`

La definizione di un record e la dichiarazione di una variabile che conterrà dati per quel record possono essere fatte insieme:

```
struct {
   int x, y;
   char nome[10];
} p;  /* variabile del tipo record appena definito */
```

Un record è anche un nuovo tipo, a cui si può dare un nome attraverso:

```
struct Punto {
   int x, y;
   char nome[10];
}; /* definizione record */
```

In questo caso la dichiarazione viene fatta con:

```
struct Punto p;
```

Per accedere ai campi di una variabile di un tipo record si usa il “.”:

```
p.x = 1;
```

Un record è utile per definire una lista concatenata. Ogni elemento della lista contiene dei dati ed un puntatore al successivo elemento della lista.

```
struct elem {
   int info;
   struct elem *succ;
};
```

Allocazione di memoria

La funzione `malloc(n)` disponibile con la libreria `stdlib.h` alloca n byte contigui di memoria e restituisce il puntatore al primo degli n byte allocati.

E' utile per immagazzinare dati non conosciuti al momento della creazione del programma.

Se non è possibile allocare gli n byte richiesti, `malloc()` restituisce un puntatore a NULL.

I byte allocati NON sono normalmente inizializzati.

```
#include <stdlib.h>
```

Elementi di un record possono essere allocati dinamicamente in memoria tramite `malloc()`:

```
struct elem *p;                          //puntatore ad elem
p = (struct elem *) malloc(elemsize); //alloca memoria per elem
p->info = 10;                           //inserisce 10 in elem.info
```

Per accedere al campo info nella struttura puntata da `p` dovrei usare `(*p).a`, tuttavia per aumentare la leggibilità, al posto dell'operatore "`.`" si usa l'operatore freccia "`->`", quindi:

`(*p).a` è sostituito da `p->a`

La funzione `free( ptr )` fa sì che la memoria referenziata da `ptr` sia disponibile per future allocazioni.

Vantaggio di usare i puntatori:

- Posso allocare dinamicamente la memoria che mi serve

Dichiarazioni composte

In C una dichiarazione di un identificatore ha la forma:

tipo dichiaratore

e dichiaratore ha una delle 5 forme:

identificatore
`(`dichiaratore`)`
dichiaratore`()`
`*`dichiaratore
dichiaratore`[`espressione-costante`]`

Per esempio: `int *a[5]; char (*b)[5]; char x[3][5]`

`()` `*` `[e]` si dicono costruttori e si leggono così:

`()` si legge “funzione che restituisce”

`*` si legge “puntatore a”

`[e]` si legge “array di e”

In un dichiaratore composto, `[]` lega più forte di `()` e `()` di `*`.

Per interpretare un dichiaratore composto, si leggono, nell’ordine:

- l’identificatore
- “è un”
- i costruttori dal più interno al più esterno
- il tipo

Esempi: `int *a[5]` dà: `a` è un array di 5 puntatori a `int`,
`char (*b)[5]` dà: `b` è un puntatore a un array di 5 `char`
`char x[3][5]` dà: `x` è un array di 3 array di 5 `char`.

Puntatori, array e stringhe

Una costante stringa è rappresentata con una sequenza di byte (uno per carattere), terminata da un byte contenente `\0`.

Una costante stringa è dunque un array di caratteri, ma ogni riferimento a una stringa è un puntatore al suo primo byte.

Alcune conseguenze:

- una stringa si può assegnare a un puntatore a un `char`: ciò fa puntare il puntatore al primo byte della stringa; p. es.

```
char *message;
message = "ciao caro\n"
```

  non copia in `message` `"ciao caro\n"`, ma il suo indirizzo.

- peraltro, una stringa non si può assegnare a un array di `char`, in quanto un nome di array è un puntatore costante. p. es.

```
char a[10];
a = "ciao";
```

  è errato. Tuttavia, come per ogni array si può inizializzare:

```
char a[10] = "ciao";
char b[] = "caro"; /* dimensione di b determinata da "caro"*/
char c[] = {'c', 'a', 'r', 'o', '\0'} /* c[] è come b[] */
```

- se una funzione ha un argomento stringa, il valore passato al parametro corrispondente è un puntatore alla stringa: p.es. in

```
printf("ciao caro\n");
```

  `printf` riceve un puntatore all'argomento `"ciao caro\n"`

Copia e confronto tra stringhe

Per copiare una stringa `t` in una `s`, `s = t` non va bene; occorre una funzione:

```
strcpy(char s[], char t[])
{
    int i = 0;
    while ((s[i]=t[i]) != '\0')
        i++;
}
```

Una versione che usa puntatori anziché indici di array:

```
strcpy(char *s, char *t)
{
    while ((*s=*t) != '\0') {
        s++; t++
    }
}
```

Il `while` si può semplificare in:

```
    while (*s++=*t++)  ;
```

Per decidere quale di due stringhe viene prima in ordine alfabetico, le si confronta carattere per carattere, e si decide appena se ne trova uno diverso:

```
strcmp(char s[], char t[])   //restituisce un numero <0 se s<t, >0 se s>t
{
    int i = 0;
    while (s[i] == t[i])
        if (s[i++] == '\0')     /* anche t[i] sarà '\0'*/
            return(0);
    return(s[i] - t[i]);     /* differenza tra codici ASCII */
}
```

La versione con puntatori è:

```
strcmp(char *s, char *t)    //restituisce un numero <0 se s<t, >0 se s>t
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')     /* anche *t sarà '\0'*/
            return(0);
    return(*s - *t);
}
```

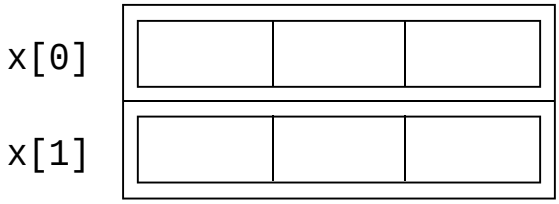
Array a più dimensioni

Un array a 2 dimensioni, p. es. 2 righe e 3 colonne, si può dichiarare con:

```
static int x[2][3] = {      /* nel C originale, l'inizializzazione è */
        {1, 2, 3},          /* ammessa solo se l'array è static */
        {3, 5, 6}
};
```

Leggendo questa dichiarazione secondo le note regole, si ha:

`x` è un array di 2 array di 3 `int`, cioè:



Convenzionalmente, in `x[2][3]`, il primo indice dà il n. di righe, dunque un array a due dimensioni è in realtà un array di array riga.

Vediamo come viene tradotto un riferimento a, p.es., `x[1][2]`, secondo le regole già viste:

- `x+1` è l'indirizzo del 2° elemento di `x`;
- `x[1]` è `*(x+1)`, cioè il 2° elemento di `x`; si tratta di un array di 3 `int`;
- in quanto array, `x[1]` viene tradotto nell'indirizzo del 1° dei suoi 3 elementi;
- `x[1]+2` è l'indirizzo del 3° elemento dell'array `x[1]`;
- `x[1][2]` è `*(x[1]+2)`, cioè il 3° elemento dell'array `x[1]`; si tratta dunque di un `int`.

Parametro array a più dimensioni

Se la funzione `f` ha un parametro `x` corrispondente all'argomento

```
int a[2][3]
```

ci sono tre possibilità: o si dichiara `f` così:

```
f(int x[2][3])
```

o, dato che `a` è un array di 2 array riga, ma il numero di elementi di un parametro array è irrilevante:

```
f(int x[][3])
```

o, dato che `x` è un puntatore a un array di 3 elementi (la prima riga):

```
f(int (*x)[3])
```

NB: ciò è diverso da:

```
int *x[3]
```

che va letto: `int *(x[3])` e significa che `x` è un array di 3 puntatori ad interi.

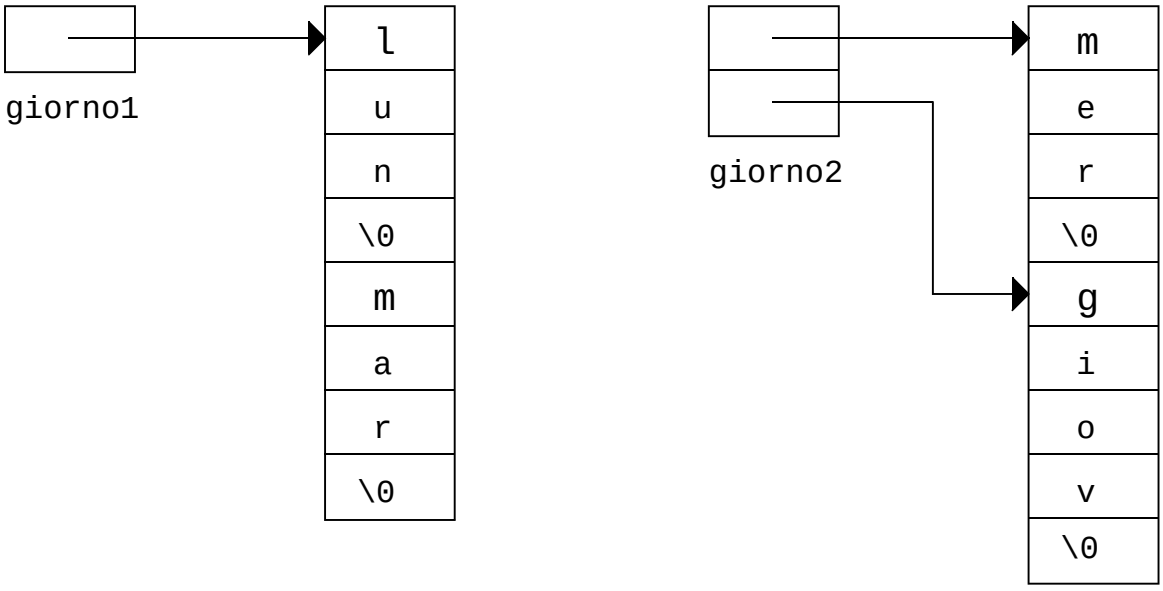
Array di puntatori

```
main()
{
    static char giorno1[2][4] = {    /* static consente di */
        "lun",                       /* inizializzare */
        "mar"
    };
    static char *giorno2[2] = {
        "mer",
        "giov"
    };
    int i;

    for (i=0; i<2; i++)
        printf("%s\t%s\n",giorno1[i],giorno2[i]);
}
```

Che differenza c'è tra `giorno1 e giorno2`?

`giorno1` è un array di 2 array di 4 caratteri

`giorno2` è un array di 2 puntatori a caratteri:
riserva spazio in più per i puntatori, ma, se questi puntano a stringhe, consente di avere stringhe di lunghezza variabile (come `mer` e `giov`).



Esempio: array di puntatori a stringhe

La funzione `readlines(lineptr,maxlines)`:

- legge righe di input, finché non ne trova una vuota;
- la riga letta `n` viene memorizzata nella memoria statica e un puntatore ad essa viene posto in `lineptr[n-1]`;

  NB: il parametro `lineptr` è un array (di puntatori a stringhe), per cui l'effetto di `readlines` sarà visibile alla funzione che la chiama

restituisce il numero di righe lette, o -1 se:
le righe lette superano maxlines o
la memoria disponibile per le stringhe lette si è esaurita.

```
#define MAXLEN 1000
readlines(char *lineptr[], int maxlines)
              /* lineptr[n] punterà alla riga letta n (da 0) */
{
    int len, nlines=0;
    char *p, line[MAXLEN];

    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines)
            return(-1);
        else if ((p = malloc(len))        /* fai puntare p a ...    */
                  == NULL)    /* len byte allocati in mem. statica    */
            return(-1);       /* memoria esaurita */
        else {
            line[len-1] = '\0'; /* elimina il '\n' da riga letta      */
            strcpy(p, line);   /*copia riga in mem. non locale       */
            lineptr[nlines++] = p;
        }
    return(nlines);
}
```

NB: la memoria statica puntata da `lineptr` va richiesta dinamicamente, con `malloc`, da `readlines` che legge le righe e sa quanta ne serve.

L'alternativa è dichiarare `char lineptr[MAXLIN][MAXLEN]`

Puntatori: assegnazione across-scope

```
readlines(char *lineptr[], int maxlines)
              /* lineptr[n] punterà alla riga letta n (da 0) */
{
    int len, nlines=0;
    char *p, line[MAXLEN];

    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines)
            return(-1);
        else if ((p = malloc(len))        /* fai puntare p a ...    */
                  == NULL)    /* len byte allocati in mem. statica    */
            return(-1);       /* memoria esaurita */
        else {
            line[len-1] = '\0'; /* elimina il '\n' da riga letta      */
            strcpy(p, line);   /*copia riga in mem. non locale       */
            lineptr[nlines++] = p;
        }
    return(nlines);
}
```

Perché introdurre `p` e non scrivere `lineptr[nlines++]=line`?

Perché `p` punta a memoria non locale,
mentre la memoria puntata da `line` ha esistenza locale.

Per la stessa ragione non può essere

```
strcpy(s,t) char *s; char *t; { s = t; }
```

perché `p` e `lineptr[nlines++]` punterebbero a memoria locale.

Un altro problema per questa versione di `strcpy`:

```
char w[] = "ok";

main() {
    char *z = malloc(8);
    strcpy(z,w); /* z non punta più alla memoria allocata da malloc */
    free(z); /* errore: z non punta più a memoria allocata da malloc */
}
```

Insomma, `strcpy(s,t)` non deve cambiare la natura della memoria puntata da `s`.

Sorting dell'input

```
#define NL 100
main()    /* sorting delle righe di input */
{
    char *lineptr[NL]    /* la stringa lineptr[i] è la riga i */
    int nlines;

    if ((nlines = readlines(lineptr,NL)) >= 0) {
        sort(lineptr,nlines);
        writelines(lineptr, nlines);
    }
    else
        printf("input too big\n");
}
```

La funzione `sort(lineptr,nlines)` ordina alfabeticamente `lineptr` in modo che `lineptr[i] < lineptr[i+1]`.

L'output è affidato a:

```
writelines(char *lp[], int nl)
{
    int i;
    for (i=0; i<nl; i++)
        printf("%s\n", lp[i]);
}
```

oppure:

```
writelines(char *lp[], int nl)
{
    int i;
    while (--nl >= 0)
        printf("%s\n", *lp++);
}
```

Argomenti della riga di comando

Finora, `main` è stato mostrato senza parametri.

In realtà ne può prendere 2, che convenzionalmente si chiamano:

- `argc` , numero degli argomenti che il programma compilato prende sulla riga di comando;
- `argv` , array di puntatori a stringhe:

  `argv[0]` è il nome con cui il programma è stato chiamato sulla riga di comando;

  `argv[1]` è il nome del 1° argomento sulla riga di comando ....

  `argv[2]` ....

Il programma seguente si comporta come `echo` in UNIX:

```
main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

Altra versione del blocco, incrementa `argv`:

```
    while (--argc > 0)
        printf("%s%c", *++argv, (argc>1) ? ' ' : '\n');
```

O, valutando il primo argomento di `printf`:

```
    while (--argc > 0)
        printf((argc>1) ? "%s " : "%s\n", *++argv);
```

Funzioni come argomenti

Si desidera una funzione `sum(x,n)` che, dato un array `x` di `n` `double`, calcoli:

$$x[1]^2+x[2]^2+...x[n]^2$$

Eccone un uso:

```
main()
{
    double a[] = {1,2,3};

    printf("%f\n", sum(a,3));
}
```

e una realizzazione:

```
double sum(double x[], int n)
{
    double f();
    int i;
    double tot=0;

    for (i=0; i<n; i++)
        tot += f(x[i]);
    return(tot);
}
double f(double x)
{
    return(x*x);
}
```

Sarebbe utile che `sum` potesse calcolare `f(x[1])+...+f(x[n])` per `f` arbitraria, ma si può avere solo una `f` per volta.

Parametri funzione

La soluzione sta nel passare a `sum` la funzione `f` come argomento. Il parametro corrispondente `pf` è in effetti un puntatore (al codice del)la funzione `f`, in quanto non si vuole passare una copia di `f`:

```
double sum(double (*pf)(double), double x[], int n)
{
    int i;
    double tot=0;

    for (i=0; i<n; i++)
        tot += (*pf)(x[i]);
    return(tot);
}

double sqr(double x)
{
    return(x*x);
}

double cub(double x)
{
    return(x*x*x);
}
```

Nello stesso programma si può ora invocare `sum` per diverse `f`:

```
main()
{
    double a[] = {1,2,3};
    printf("%f\n%f\n", sum(sqr,a,3), sum(cub,a,3));
}
```

Come per gli array, in C ogni riferimento a una funzione viene convertito nell'indirizzo della funzione.

Per questo gli argomenti corrispondenti a parametri puntatori a funzione sono nomi di funzione e non richiedono `&`.

P. es. si scrive `sum(sqr,a,3)`, anziché `sum(&sqr,a,3)`.