Introduzione al linguaggio C

Violetta Lonati

Dipartimento di Informatica
Università degli studi di Milano

Ultimo aggiornamento

October 7, 2021

Argomenti I

Argomenti II

- Parametri, argomenti e variabili locali
- L'istruzione return
- Organizzazione dei programmi
- Alcune osservazioni sulle funzioni
- Funzioni ricorsive

Alcuni algoritmi di ordinamento ricorsivi

- Selectionsort
- Mergesort

Puntatori

- Operatori & e *
- Puntatori come argomento di funzione
- Puntatori come valori restituiti
- Puntatori a strutture
- Aritmetica dei puntatori
- Puntatori e array
- Stringhe e puntatori

Argomenti III

Array frastagliati

- Array di stringhe
- Argomenti da linea di comando

Allocazione dinamica della memoria

- Puntatore nullo
- Funzioni per l'allocazione dinamica della memoria
- Deallocare memoria
- Errori tipici
- Esempi

Uso del compilatore gcc

- Si usa da linea di comando: `gcc file_da_compilare.c`
- Per default produce il file eseguibile `a.out`
- Per eseguire il file basta digitare `./a.out`
- Opzioni utili:
  - `-o file_di_output` per salvare l'output nel file specificato,
  - `-c` per arrestare la compilazione prima della fase di link,
  - `-s` per arrestare la compilazione prima della fase di assemble,
  - `-E` per arrestare la compilazione dopo la fase di preprocessing,
  - `-W` , con parametro `all` , per la segnalazione di avvertimenti (warning),
  - `-l` , con parametro `m` , per linkare le librerie matematiche

Esempio

```
$ gcc -Wall -o cerchio cerchio.c -lm
$ ./cerchio
```

Forma generale di un programma C

Esempio di programma

```
#include <stdio.h>

int main( void ) {
  printf( "Ciao!\n" );
  return 0;
}
```

Struttura generale di un programma

```
DIRETTIVE
int main( void ) {
  ISTRUZIONI
}
```

Tipi numerici in C

In C esistono due tipi numerici built-in (di base):

- numeri interi int
- numeri decimali in virgola mobile float

La dimensione dei tipi varia a seconda della macchina. L'operatore `sizeof` consente di determinare quanta memoria occupa una variabile di un determinato tipo:

- `sizeof ( int )` rappresenta il numero di byte necessari a memorizzare una variabile di tipo int,
- `sizeof ( a )` rappresenta il numero di byte necessari a memorizzare la variabile a.

Tipo int

- Possono avere segno oppure no:
  - `signed int`: il bit più a sinistra vale 0 se il numero è ≥ 0 e 1 se è < 0
  - `unsigned int`: bisogna specificarlo nella dichiarazione
- Possono avere diverse dimensioni:
  - `short int`: nelle dichiarazioni si può abbreviare in `short`
  - `long int`: nelle dichiarazioni si può abbreviare in `long`
- Il range di variabilità non è fissato nel C standard ma cambia a seconda della macchina, gli unici vincoli sono i seguenti:

  $$\text{short int} \leq \text{int} \leq \text{long int}$$

- Il file di intestazione limits.h fornisce alcune macro che definiscono i valori limite, per l'architettura corrente, dei tipi interi.

Dichiarazioni per i tipi interi:

```
short si;          unsigned short usi;
int i;             unsigned int ui;
long li;           unsigned long uli;
```

Tipo float

- In C ci sono tre tipi di numeri in virgola mobile:
  - float: single-precision
  - double: double-precisione
  - long double: extended-precision
- Anche per i float il range di variabilità non è fissato
- Il file di intestazione float.h fornisce alcune macro che definiscono, per l'architettura corrente, la precisione dei tipi float.
- Le costanti possono essere scritte in molti modi, purchè contengano un decimale e/o un'esponente. Ad esempio:

```
57.0      57.      57.0e0      57E0
5.7e1     5.7e+1   .57e+2      570.e-1
```

  Per default le costanti sono memorizzate come double. Se basta la singola precisione, basta usare la lettera `f` o `F` alla fine della costante (es: `57.0f`).

Tipo char

L'ultimo tipo built-in del C è il tipo `char` (carattere)

```
char ch;
ch = 'a'; /* a minuscola */
ch = 'A'; /* A maiuscola */
ch = '0'; /* zero */
ch = ' '; /* spazio */
```

- Il valore di un char può cambiare a seconda della character set della macchina.
- Si può generalmente assumere che le lettere minuscole siano contigue nell'ordine alfabetico, idem per le maiuscole, idem per le cifre.
- I char sono usati dal C come tipi interi: le variabili char possono essere incrementate o confrontate come interi (l'esito del confronto dipende dall'ordinamento del character set).
- Il file di intestazione ctype.h fornisce alcune funzioni per l'elaborazione di caratteri e la conversione di lettere da maiuscole a minuscole e viceversa.

Tipo char - esempi

Assumendo che il character set sia ASCII:

```
char ch;
int i;
i = 'a';          /* ora i vale 97 */
ch = 65;          /* ora ch vale 'A' */
ch++;             /* ora ch vale 'B' */
```

Per convertire lettere minuscole in maiuscole:

```
if ( 'a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

Oppure:

```
#include <ctype.h>
  ...
if ( islower( ch ) )
  ch = toupper( ch );
```

Definizione di tipi

La parola chiave `typedef` consente di definire nuovi tipi a partire da quelli built-in o dai tipi precedentemente definiti

Esempio

Definiamo un nuovo tipo Bool. Bool potrà essere usato nelle dichiarazioni di variabili esattamente come gli altri tipi built-in.

```
#define VERO 1
#define FALSO 0

typedef int Bool;    /* Def. del tipo Bool */

int main ( void ) {

  Bool flag;         /* Dich. della var flag */
  flag = VERO;       /* Assegno a flag valore VERO */
  ...
}
```

Input e output formattati

- `printf` e `scanf` danno la possibilità di stampare output e leggere input formattati.
- Il primo argomento è dato dalla stringa di formato che può contenere
  - sequenze di escape (es `\n`)
  - specifiche di formato:
    - `%d` per gli interi;
    - `%f` per i float in notazione decimale;
    - `%e` per i float in notazione esponenziale;
    - `%c` per i char;
    - è possibile specificare anche il numero di decimali, di 0 iniziali, l'allineamento...
- Funzionamento di `scanf` in presenza di spazi bianchi:
  - gli spazi bianchi iniziali vengono ignorati nella lettura di int e float, ma non di char;
  - uno spazio bianco nella stringa di formato significa "salta uno o più caratteri bianchi"; ad esempio `scanf( " ␣%c", &ch )` salta gli spazi bianchi e poi memorizza ch.

Lettura e scrittura di caratteri

`getchar` e `putchar` permettono di leggere e stampare un carattere alla volta.

- `ch = getchar();`
  memorizza in ch il prossimo carattere da standard input;
- `putchar( ch );`
  stampa il carattere ch su standard output.

```
/* Trasforma la riga da minuscole a maiuscole */
  char ch;
  while ( ( ch = getchar() ) != '\n' ) {
    if ( islower( ch ) )
      putchar( toupper( ch ) );
    else
      putchar( ch );
  }
```

```
/* Salta gli spazi bianchi */
  while ( ( ch = getchar() ) == '␣' )
    ;
```

Array unidimensionali

Dichiarazione

```
#define N 100
int a[N + 2];
```

Indicizzazione

`a[i]` indica l'i-esimo elemento dell'array a. Al posto di `i` si può mettere una qualsiasi espressione intera.
Attenzione agli effetti collaterali in `a[i] = b[i++];`

Inizializzazione

```
int a[4] = {1, 2, 0, 0};
int b[] = {1, 2, 0, 0};
int c[4] = {1, 2};
```

Eventuali elementi mancanti sono inizializzati a 0.

Array unidimensionali - continua

Lunghezza di un array

```
sizeof( a ) / sizeof( a[0] )
```

Array costanti

```
const int MESI[12] = {31,28,31,30,31,30,31,31,30,31,30,31}
```

Il qualificatore `const` rende ogni elemento immodificabile.

Esempi

```
  for ( i  = 0; i < N; i++ )
    a[i] = 0
/* azzero l'array a */
```

```
  for ( i  = 0; i < N; i++ )
    scanf( "%d", &a[i] );
/* leggo e mem. in a */
```

```
  for ( i  = 0; i < N; i++ )
    printf( "%d␣", a[i] );  /* stampo */
```

Array unidimensionali - errori tipici

Dimensione degli array

Secondo lo standard del C90, la dimensione dell'array va fissata in fase di compilazione (non di esecuzione) usando un espressione costante. La seguente dichiarazione produce errore di compilazione in C90, ma è previsto in C99 (VLA: variable-length arrays) in cui comunque la dimensione di un array non può variare nel corso del programma. Nello standard C11 i VLA sono opzionali.

```
int n = 10;
int a[n];
```

Non c'è controllo dei limiti

```
int a[N], i;
for ( i = 0; i <= N; i++ )
  printf( "%d\n", a[i] );
/* accede ad a[N] che non e' definito!! */
```

Array unidimensionali - esercizi

- Rovescia: scrivete un programma che legga una sequenza di numeri interi terminata da 0 e li stampi dall'ultimo (0 escluso) al primo. Potete assumere che la sequenza contenga al più 100 numeri non nulli.
- Cifre ripetute: scrivete un programma che legga in input un numero intero n usando `scanf( "%d", &n )` e stabilisca se n contiene qualche cifra ripetuta e in caso affermativo quali.
- Da base 10 a base b: scrivere un programma che data una coppia di numeri interi b e n (separati da spazio e in base 10) stampi la rappresentazione di n in base b. Potete assumere che il numero di cifre in base b sia sempre minore di 100.
- Da base b a base 10: scrivere un programma che dato un numero b (in base 10) e una sequenza s di cifre in $\{0, \ldots b-1\}$ terminata da un punto, stampi il numero la cui rappresentazione in base b è data da s. Potete assumere che il numero di cifre di s sia sempre minore di 100.

Array bidimensionali

```
int mat[R][C], r, c;
/* inizializzo a 0 */
for ( r = 0; r < R; r++ )
    for ( c = 0; c < C; c++ )
      mat[r][c] = 0;
```

Mappa di memorizzazione

Un array di R righe e C colonne viene memorizzato come array di $R * C$ elementi. In altre parole `M[i][j]` si trova $i * C + j$ posizioni dopo `M[0][0]`.

Inizializzazione

```
int a[3][2] = { {1, 2}, {3, 4}, {0, 0} };
int b[][2] = { {1, 2}, {3, 4}, {0, 0} };
/* non si puo' omettere la seconda dimensione! */
int c[3][2] = {1, 2, 3, 4, 0, 0};
int d[3][2] = {1, 2, 3, 4};
/* eventuali el. mancanti sono inizializzati a 0 */
```

Array bidimensionali - esercizi

- Esami e studenti: scrivete un programma che permetta di inserire gli esiti di 5 esami per 5 studenti e calcoli la media di ciascuno studente e la media dei voti ottenuti in ciascun esame.
- Quadrato magico: scrivete un programma che legga un intero n e stampi un quadrato magico di dimensione n.

Stringhe

Le stringhe in C sono array di char, terminati dal carattere di fine stringa. Il carattere di fine stringa ha valore 0 (i char sono trattati come interi!), ma è opportuno indicarlo come carattere `'\0'`. La lunghezza di una stringa è data dal numero dei suoi caratteri + 1 (per il simbolo di fine stringa).

Inizializzazione

```
char parola1[5] = "ciao";
char parola2[] = "ciao";
char parola3[] = { 'c', 'i', 'a', 'o', '\0' };
```

Lettura e scrittura di stringhe

```
scanf( "%s", parola ); /* senza & !!! */
printf( "%s\n", parola );
```

Funzioni per elaborare stringhe

`<string.h>` contiene le dichiarazioni di alcune funzioni utili:

`strcpy`, `strcmp`, `strcat`, `strlen`,...

Stringhe - esempio

```
/* copia parola in bis*/
  char parola[] = "ciao", bis[4+1];
  int i = 0;

  while ( parola[i] != '\0' ) {
    bis[i] = parola[i];
    i++;
  }

  bis[i] = '\0';
  printf( "%s\n", bis );
```

Stringhe - esercizio

Una stringa si dice *palindroma* se è uguale quando viene letta da destra a sinistra e da sinistra a destra. Quindi `"enne"` è palindroma, ma `"papa"` non lo è. Scrivete un programma che legga una stringa terminata da `'.'` e stabilisca se è palindroma. Potete assumere che la stringa sia al più di 100 caratteri.

Strutture

- Si tratta di tipi di dati aggregati. Ogni variabile strutturata è composta da membri, ciascuno dei quali è individuato da un nome.
- Ogni struttura ha un namespace distinto, quindi i nomi dei membri possono essere usati anche per altre cose!
- In altri linguaggi, le strutture sono chiamate record e i membri campi.

```
/* Dichiarazione della var. strutturata studente */
/* avente 3 membri: cognome, nome, anno. */
  struct {
    char cognome[10];
    char nome[10];
    int anno;
} studente;
```

Strutture - continua

Inizializzazione

```
struct {
  char cognome[10];
  char nome[10];
  int anno;
} studente1 = { "Ferrari", "Mario", 1988 },
  studente2 = { "Rossi", "Maria", 1988 };
```

Accesso ai membri

```
studente.anno = 1987;
printf( "%s\n", studente.nome );
```

Assegnamento

E' possibile farlo tra due variabili strutturate dello stesso tipo: tutti i membri vengono copiati (con gli array questo non si può fare!!).

```
studente1 = studente2;
```

Definizione di tipi strutturati

Ci sono due modi per definire tipi strutturati:

```
/* usando un tag */
  struct studente {
       char nome[10];
       char cognome[10];
       int anno;
  };
  ...
  struct studente stud1, stud2;
```

```
/* usando typedef */
  typedef struct {
       char nome[10];
       char cognome[10];
       int anno;
  } Studente;
  ...
  Studente stud1, stud2;
```

Strutture nidificate

```
typedef struct {
  float x, y;
} Punto;

typedef struct {
  Punto p1, p2;
} Rettangolo;

...

Rettangolo r;
r.p1.x = 0;
r.p1.y = 1;
r.p2.x = 5;
r.p2.y = 3;
```

Esercizio: Scrivete un programma che calcoli l'area e il perimetro di rettangoli e cerchi.

Array di strutture

```
#define LUNG 20 /* lungh. massima per le stringhe */
#define N 100 /* numero massimo di voci */

typedef char Stringa[LUNG + 1];

typedef struct {
  Stringa nominativo;
  Stringa tel;
} Voce;

...

/* dichiaro rubrica come array di N voci */
Voce rubrica[N];
```

Esercizio: Scrivete un programma per la gestione di una semplice rubrica. Attraverso un menu l'utente deve poter visualizzare la rubrica e inserire nuovi numeri.

Enumerazioni

- Si tratta di tipi di dati i cui valori sono *enumerati* dal programmatore.

```
enum { CUORI, QUADRI, FIORI, PICCHE }
        seme1, seme2;
```

- Ad ogni possibile valore si associa una costante di enumerazione intera. Per default i valori associati alle costanti sono 0, 1, 2, ..., in questo ordine. E' possibile scegliere altri valori.

```
enum { VENDITE = 10,
       RICERCA = 21,
       PRODOTTI = 17 } settore;
```

- Si possono definire tipi enumerativi usando sia typedef che tag.

```
typedef enum { FALSO, VERO } Bool;
```

Funzioni

- Le funzioni sono costituite da una sequenze di istruzioni raccolte sotto un solo nome.
- Le funzioni possono avere argomenti e possono calcolare e restituire dei valori, ma anche no! (a differenza delle funzioni matematiche)
- I programmi visti sin qui erano costituiti da una sola funzione, il main.
- A cosa servono le funzioni?
  - permettono di evitare la duplicazioni di codice;
  - sono riutilizzabili.

Esempio: calcolare la media

```
#include <stdio.h>

float media (float x, float y ) {
  return (x + y) / 2;
}

int main( void ) {
  float a, b, c, m;
  scanf( "%f%f%f", &a, &b, &c );

  m = media(a, b);
  printf( "La media tra %f e %f e' %f\n",
                        a, b, m );

  printf( "La media tra %f e %f e' %f\n",
                        a, c, media(a, c) );
  return 0;
}
```

Esempio: conto alla rovescia

```
#include <stdio.h>

void stampa_conteggio ( int n ) {
  printf( "meno␣%d...\n", n );
}

int main( void ) {
  int i;

  for ( i = 10; i > 0; --i )
    stampa_conteggio( i );
  return 0;
}
```

Definizione di funzioni

La forma generale di una funzione è la seguente:

```
TIPO_RESTITUITO NOME_FUNZIONE ( PARAMETRI ) {
  DICHIARAZIONI
  ISTRUZIONI
}
```

TIPO_RESTITUITO

E' il tipo del valore che la funzione restituisce mediante il return:

- le funzioni non possono restituire array, ma non ci sono restrizioni sugli altri tipi;
- se `TIPO_RESTITUITO` è `void`, la funzione non restituisce alcun valore;
- se il `TIPO_RESTITUITO` è omesso, per default è `int`; è comunque meglio indicarlo sempre!

NOME_FUNZIONE

E' il nome della funzione, si usa per chiamare la funzione stessa.

Definizione di funzioni - continua

PARAMETRI

E' una lista di parametri separati da virgole.

- ciascun parametro deve essere preceduto dal suo tipo (il tipo va ripetuto anche se ci sono più parametri dello stesso tipo)

```
float media( float x, y ) /* SBAGLIATO! */
```

Corpo della funzione

Il corpo della funzione può contenere dichiarazioni e istruzioni; le variabili dichiarate nel corpo di una funzione si chiamano locali.

```
float media (float x, float y ) {
  float sum;          /* dichiarazione */

  sum = x + y;        /* istruzione */
  return sum / 2;     /* istruzione */
}
```

Chiamata di funzioni

- Una chiamata di funzione consiste nel nome della funzione seguito da una lista di argomenti tra parentesi.

```
media( 3, 6 )
media( a, b * b )
stampa_conteggio( 9 )
getchar( )
```

- La chiamata di una funzione coinvolge sempre due funzioni (che possono anche coincidere, nella ricorsione):
  - una funzione chiamante
  - e una funzione chiamata.

  Nei primi esempi, la funzione chiamante è il main.
- Quando una funzione viene chiamata, il controllo passa dalla funzione chiamante alla funzione chiamata. Quando questa termina (alla fine delle istruzioni oppure in presenza dell'istruzione return), restituisce il controllo alla funzione chiamante.

Chiamata di funzioni - continua

- Se mancano le parentesi, la funzione non verrà chiamata. Se una funzione è di tipo void, bisognerà chiamarla così:

```
getchar( );
```

- Una chiamata di funzione di tipo void è una istruzione, quindi deve concludersi con punto-e-virgola;

```
stampa_conteggio( 9 );
```

- Una chiamata di funzione di tipo non void è un'espressione e produce un valore che può essere assegnato ad una variabile, testato, stampato, ecc

```
m = media( 3, 6 );
if ( media( a, b * b ) > c ) printf( "..." );
```

- Il valore restituito da una funzione può essere scartato:

```
printf( "Ciao␣mondo!\n" );
chars = printf( "Ciao␣mondo!\n" );
```

Dichiarazione di funzioni

- La definizione di una funzione non deve necessariamente precedere il punto in cui viene chiamata. Ad esempio, le definizioni di funzioni possono seguire il main.
- Se il compilatore trova una chiamata prima della definizione, non sa quanti e di che tipo siano i parametri, quindi fa delle assunzioni e effettua dei casting. Non è detto che le queste assunzioni siano corrette!
- Per evitare questo problema, è possibile dichiarare le funzioni prima che vengano chiamate.

```
TIPO_RESTITUITO NOME_FUNZIONE ( PARAMETRI );
```

  Una dichiarazione di questo tipo si chiama prototipo o segnatura.
- Attenzione: ci deve essere coerenza tra dichiarazione e definizione!!

Esempio: media rivisitata

```
#include <stdio.h>

/* prototipo della funzione media */
float media (float x, float y );

/* main */
int main( void ) {
  float a, b, c, m;
  scanf( "%f%f%f", &a, &b, &c );

  printf( "La media tra %f e %f e' %f\n",
                        a, b, media( a, b ) );
  return 0;
}

/* definizione della funzione media */
float media (float x, float y ) {
  return (x + y) / 2;
}
```

Parametri e argomenti

- I parametri di una funzione compaiono nella sua definizione: sono nomi che rappresentano i valori da fornire alla funzione al momento della chiamata.
- Gli argomenti sono invece le espressioni che compaiono nella chiamata di funzione tra parentesi.
- A volte i parametri sono chiamati parametri formali mentre gli argomenti sono chiamati parametri attuali.
- Se il tipo di un argomento non corrisponde al tipo del parametro, viene eseguita una conversione coerente con il prototipo della funzione oppure una conversione di default.
- Gli argomenti sono passati per valore: al momento della chiamata, ogni argomento è valutato e il valore è assegnato al corrispondente parametro; eventuali cambiamenti di valore dei parametri durante l'esecuzione della funzione non influenzano il valore dell'argomento!

Parametri e variabili locali

Le variabili locali hanno le seguenti proprietà:

- sono utilizzabili solo dal punto in cui sono dichiarate fino alla fine della stessa funzione (block scope);
- non possono essere usate o modificate da altre funzioni;
- sono automaticaticamente allocate al momento della chiamata della funzione e deallocate al return (automatic storage duration);
- alla fine dell'esecuzione della funzione non conservano il loro valore.

I parametri hanno le stesse proprietà (block scope e automatico storage duration) delle variabili locali. Di fatto, la sola differenza tra variabili locali e parametri è che i parametri vengono inizializzati al momento della chiamata (con il valore degli argomenti della chiamata).

Conversione di tipi

- In C sono consentiti assegnamenti ed espressioni che mescolano i tipi, ma alcuni operandi verranno convertiti automaticamente (implicitamente) in modo da rendere possibile la valutazione dell'espressione.
- Nelle chiamate di funzione è consentito passare argomenti di tipo diverso da quelli dei parametri; anche in questo caso si avranno delle conversioni implicite.
- Le regole di conversione implicita sono complicate! In sintesi:
  - nelle espressioni aritmetiche gli operandi vengono promossi;
  - negli assegnamenti il lato destro viene convertito al tipo del lato sinistro;
  - se la funzione è dichiarata/definita prima della chiamata, ogni argomento è convertito implicitamente al tipo del parametro corrispondente;
  - se la funzione non è dichiarata/definita prima della chiamata, il compliatore esegue delle promozioni di default
- E' possibile anche effettuare conversioni esplicite con il cast.

Array come argomenti

- Se il vettore è unidimensionale, la dimensione può essere omessa.

```
int f( int a[] ) { ... }
```

- Se il vettore è multidimensionale, solo la prima dimensione può essere omessa.

```
int f( int a[][LUN] ) { ... }
```

- la funzione non ha modo di sapere quanto è lungo il vettore (l'operatore `sizeof` non può essere usato...) quindi può essere utile passare la lunghezza come parametro aggiuntivo:

```
int somma_array( int a[], int n ) {
  int i, sum = 0;
  for ( i = 0; i < n; i++ )
    sum += a[i];
  return sum;
}
```

L'istruzione return

- Ogni funzione non void contiene l'istruzione return per specificare quale valore deve essere restituito alla funzione chiamante.

```
return ESPRESSIONE;
```

- L'espressione può essere una semplice costante o variabile, oppure un'espressione più complicata (del tipo giusto!)

```
return i > j ? i : j;
```

- L'esecuzione dell'istruzione return ha per effetto la restituzione del controllo alla funzione chiamante. La restituzione del controllo alla funzione chiamante avviene comunque al termine dell'esecuzione della funzione.
- Il valore restituito dal main è un codice di stato: il codice 0 indica la terminazione senza errori.

Organizzazione dei programmi

Quando un programma è costituito da più di una funzione, è importante organizzarlo bene!

- Ricordare che variabili locali e parametri hanno block scope e automatic storage duration;
- Le funzioni comunicano fra loro attraverso i parametri, oppure attraverso variabili esterne o globali, che hanno queste proprietà:
  - sono definite fuori dal corpo di ogni funzione;
  - hanno static storage duration, cioè permangono e mantengono il loro valore indefinitamente;
  - sono visibili dal punto in cui sono definite per tutto il file.

Organizzazione dei programmi - continua

In generale, un programma può essere organizzato come segue:

- direttive `#include`
- direttive `#define`
- definizione di tipi con `typedef`
- dichiarazione di variabili esterne
- prototipi delle funzioni diverse dal `main`
- definizione del `main`
- definizione delle altre funzioni

Esercizi

- Scrivete una funzione avente due parametri interi b ed e che calcoli la potenza b^e.
- Scrivete una funzione con parametro un intero n che stabilisca se n è un numero primo.
- Scrivete un programma che generi a caso un array di interi e calcoli la somma dei suoi elementi: strutturate il programma usando una funzione per generare l'array e una per sommare i suoi elementi.
- Scrivete una variante dell'esercizio precedente per un array bidimensionale.

Esercizi - continua

- Scrivete un programma che legga un carattere, uno spazio e quindi una sequenza di caratteri minuscoli terminati da `'.'` e che stampi quanto ha letto dopo il primo spazio, ma sostituendo tutte le vocali con il primo carattere letto. Per farlo, usate una funzione che, dati due caratteri, restituisca il primo carattere se il secondo è una vocale minuscola, altrimenti restituisca il secondo carattere.
- Rivedete gli esercizi svolti nelle esercitazioni precedenti e provate a strutturarli usando delle funzioni. Ad esempio, modificate l'esercizio sulla rubrica usando una funzione per stampare la rubrica, una per leggere una nuova voce, una per inserire la nuova voce nell'ordine corretto.

Overloading

A differenza di Java, in C non è possibile fare overloading, ovvero, non si possono definire due funzioni con lo stesso nome e propotipo diverso!

```
int media_int( int x, int y ) {
  return ( x + y ) / 2
}
```

```
float media_float( float x, float y ) {
  return ( x + y ) / 2.0
}
```

Lista di parametri a lunghezza variabile

- Esistono funzioni con numero di parametri variabile (es: `printf`);
- Il file di intestazione stdarg.h fornisce gli strumenti per poterle gestire (qui non le vediamo, si veda il capitolo 26 del libro).
- Attenzione: chiamare una funzione di questo tipo è un'operazione sempre rischiosa!

Funzioni ricorsive

Una funzione si dice ricorsiva quando chiama se stessa.

```
int fact( int n ) {
  if ( n <= 1 )
    return 1;
  else
    return n * fact( n - 1 );
}
```

Le chiamate ricorsive si impilano.

Ad esempio, tracciamo l'esecuzione della chiamata `fact(3)`:

`fact(3)` chiama
- `fact(2)` che chiama
  - `fact(1)` che restituisce 1, quindi
- `fact(2)` restituisce $2 \times 1 = 2$, quindi

`fact(3)` restituisce $3 \times 2 = 6$.

Esempio

La potenza può essere definita ricorsivamente, osservando che $b^0 = 1$ e $b^e = b * b^{e-1}$ per ogni $e > 0$.

```
int power( int b, int e ) {
  if ( e = = 0 )
    return 1;
  else
    return b * power( b, e - 1 );
}
```

La chiamata `power(5, 3)` sarà eseguita come segue:

`power(5, 3)` chiama
  `power(5, 2)` che chiama
    `power(5, 1)` che chiama
      `power(5, 0)` che restituisce 1, quindi
    `power(5, 1)` restituisce $5 \times 1 = 5$, quindi
  `power(5, 2)` restituisce $5 \times 5 = 25$, quindi
`power(5, 3)` restituisce $5 \times 25 = 125$.

Algoritmi di ordinamento

- Abbiamo già implementato l'ordinamento per inserimento, utile per riempire un array mantenendolo ordinato ad ogni passo.
- Ora vediamo alcuni algoritmi di ordinamento classici, usati per ordinare un array già riempito ma in disordine.
  - selection sort (in italiano ordinamento per selezione);
  - mergesort (in italiano ordinamento per immersione);
- Tutti e due gli algoritmi sono basati sui confronti tra elementi e sono ricorsivi: la base della ricorsione è data dagli array di lunghezza 0 o 1, che sono sempre ordinati.
- Per semplicità faremo riferimento soltanto ad array di interi.

selectionsort (int a[ ], int n)

Ordina i primi n elementi del vettore a, procedendo come segue:

1. cerca in a l'elemento massimo e lo scambia con l'elemento nell'ultima posizione dell'array;
2. richiama ricorsivamente se stessa per ordinare i primi $n - 1$ elementi dell'array.

selectionsort (int a[ ], int n)

Ordina i primi n elementi del vettore a, procedendo come segue:

1. cerca in a l'elemento massimo e lo scambia con l'elemento nell'ultima posizione dell'array;
2. richiama ricorsivamente se stessa per ordinare i primi $n-1$ elementi dell'array.

Esempio su input a = 15, 29, 11, 7

selectionsort (int a[ ], int n)

Ordina i primi n elementi del vettore a, procedendo come segue:

1. cerca in a l'elemento massimo e lo scambia con l'elemento nell'ultima posizione dell'array;
2. richiama ricorsivamente se stessa per ordinare i primi $n-1$ elementi dell'array.

Esempio su input a = 15, 29, 11, 7

```
selectionsort( a, 4 )
```

15 29 11 7

selectionsort (int a[ ], int n)

Ordina i primi n elementi del vettore a, procedendo come segue:

1. cerca in a l'elemento massimo e lo scambia con l'elemento nell'ultima posizione dell'array;
2. richiama ricorsivamente se stessa per ordinare i primi $n-1$ elementi dell'array.

Esempio su input a = 15, 29, 11, 7

```
selectionsort( a, 4 )
```

15	29	11	7
15	7	11	29

selectionsort (int a[ ], int n)

Ordina i primi n elementi del vettore a, procedendo come segue:

1. cerca in a l'elemento massimo e lo scambia con l'elemento nell'ultima posizione dell'array;
2. richiama ricorsivamente se stessa per ordinare i primi $n-1$ elementi dell'array.

Esempio su input a = 15, 29, 11, 7

```
selectionsort( a, 4 )
  selectionsort( a, 3 )
```

15	29	11	7
15	7	11	29
15	7	11	29

selectionsort (int a[ ], int n)

Ordina i primi n elementi del vettore a, procedendo come segue:

1. cerca in a l'elemento massimo e lo scambia con l'elemento nell'ultima posizione dell'array;
2. richiama ricorsivamente se stessa per ordinare i primi $n-1$ elementi dell'array.

Esempio su input a = 15, 29, 11, 7

```
selectionsort( a, 4 )
  selectionsort( a, 3 )
```

15	29	11	7
15	7	11	29
15	7	11	29
11	7	15	29

selectionsort (int a[ ], int n)

Ordina i primi n elementi del vettore a, procedendo come segue:

1. cerca in a l'elemento massimo e lo scambia con l'elemento nell'ultima posizione dell'array;
2. richiama ricorsivamente se stessa per ordinare i primi $n-1$ elementi dell'array.

Esempio su input a = 15, 29, 11, 7

```
selectionsort( a, 4 )
  selectionsort( a, 3 )
    selectionsort( a, 2 )
```

15	29	11	7
15	7	11	29
15	7	11	29
11	7	15	29
11	7	15	29

selectionsort (int a[ ], int n)

Ordina i primi n elementi del vettore a, procedendo come segue:

1. cerca in a l'elemento massimo e lo scambia con l'elemento nell'ultima posizione dell'array;
2. richiama ricorsivamente se stessa per ordinare i primi $n-1$ elementi dell'array.

Esempio su input a = 15, 29, 11, 7

```
selectionsort( a, 4 )
  selectionsort( a, 3 )
    selectionsort( a, 2 )
```

15	29	11	7
15	7	11	29
15	7	11	29
11	7	15	29
11	7	15	29
7	11	15	29

selectionsort (int a[ ], int n)

Ordina i primi n elementi del vettore a, procedendo come segue:

1. cerca in a l'elemento massimo e lo scambia con l'elemento nell'ultima posizione dell'array;
2. richiama ricorsivamente se stessa per ordinare i primi $n-1$ elementi dell'array.

Esempio su input a = 15, 29, 11, 7

```
selectionsort( a, 4 )
  selectionsort( a, 3 )
    selectionsort( a, 2 )
      selectionsort( a, 1 )
```

15	29	11	7
15	7	11	29
15	7	11	29
11	7	15	29
11	7	15	29
7	11	15	29
7	11	15	29

selectionsort (int a[ ], int n)

Ordina i primi n elementi del vettore a, procedendo come segue:

1. cerca in a l'elemento massimo e lo scambia con l'elemento nell'ultima posizione dell'array;
2. richiama ricorsivamente se stessa per ordinare i primi $n-1$ elementi dell'array.

Esempio su input a = 15, 29, 11, 7

```
selectionsort( a, 4 )
  selectionsort( a, 3 )
    selectionsort( a, 2 )
      selectionsort( a, 1 )
```

15	29	11	7
15	7	11	29
15	7	11	29
11	7	15	29
11	7	15	29
7	11	15	29
7	11	15	29
7	11	15	29

mergesort (int a[ ], int sx, int dx)

Ordina la parte dell'array *a* compresa tra gli indici *sx* e *dx*, come segue:

1. divide l'array in due sotto-array di dimensione circa uguale;
2. ordina il sotto-array di sinistra richiamando se stessa;
3. ordina il sotto-array di destra richiamando se stessa;
4. integra (merge) i due sotto-array in un unico array ordinato.

La base della ricorsione è data dagli array di lunghezza 0 o 1, che sono sempre ordinati.

merging

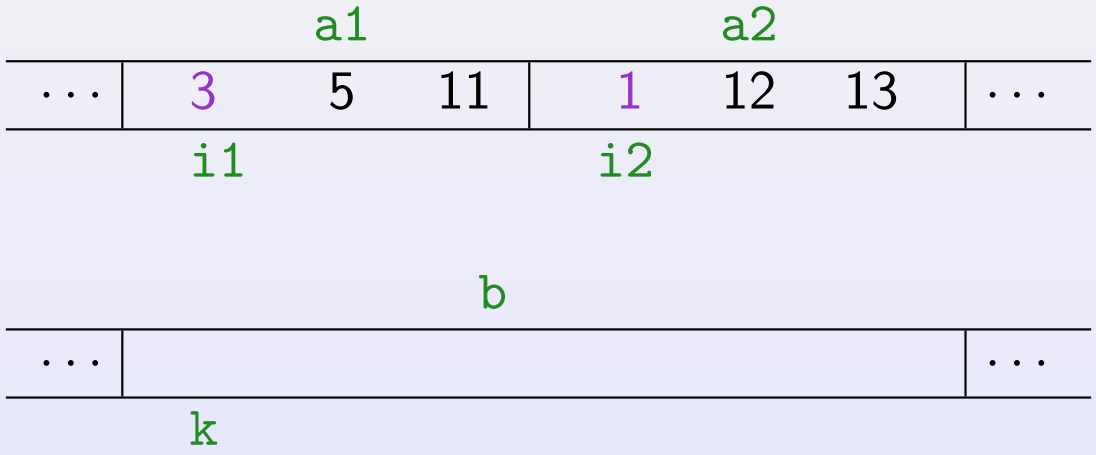
La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.

merging

La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

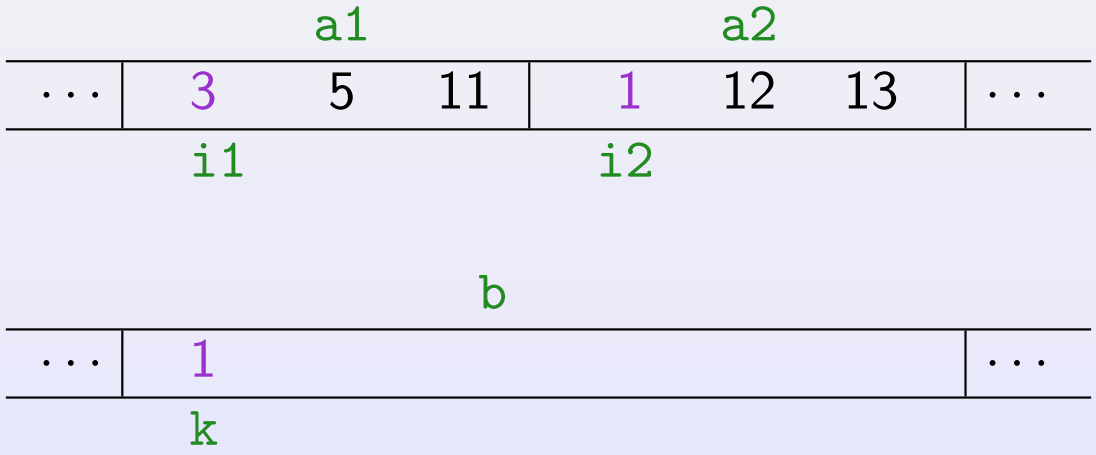
- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.



merging

La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

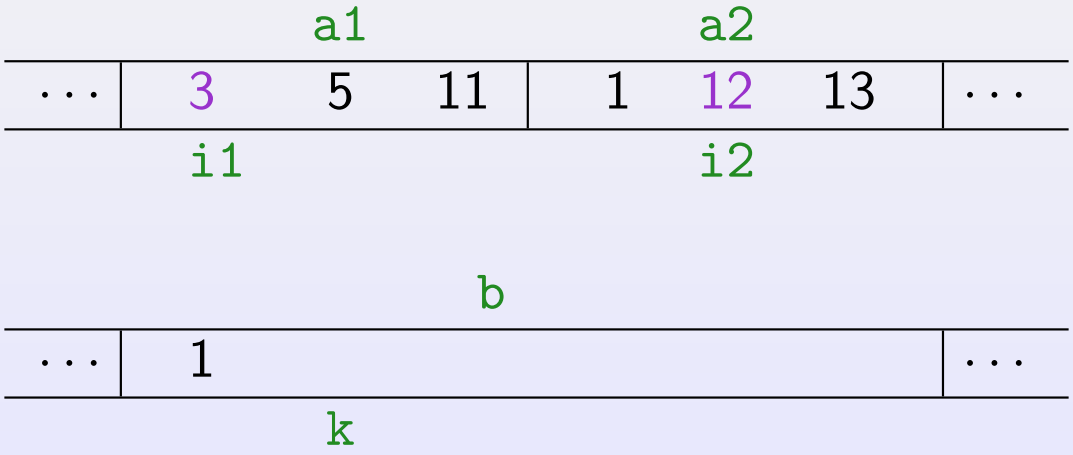
- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.



merging

La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

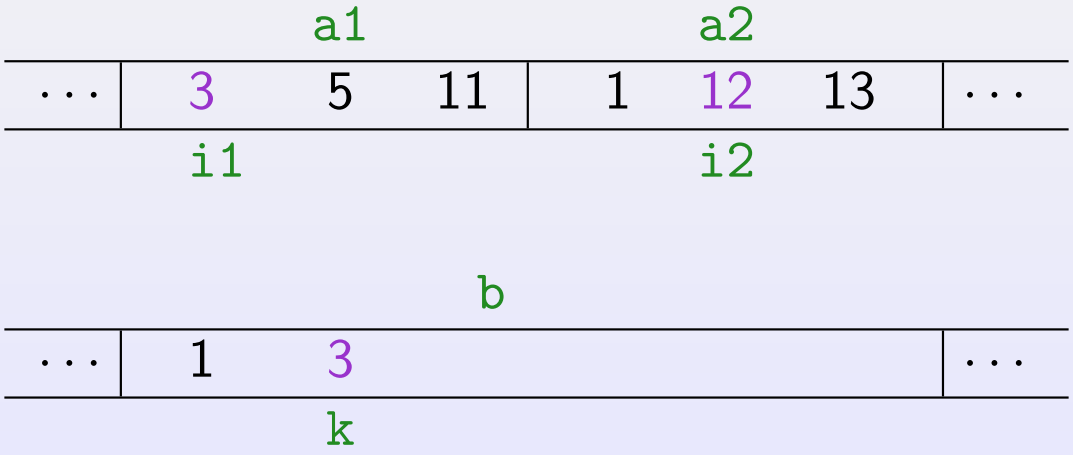
- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.



merging

La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

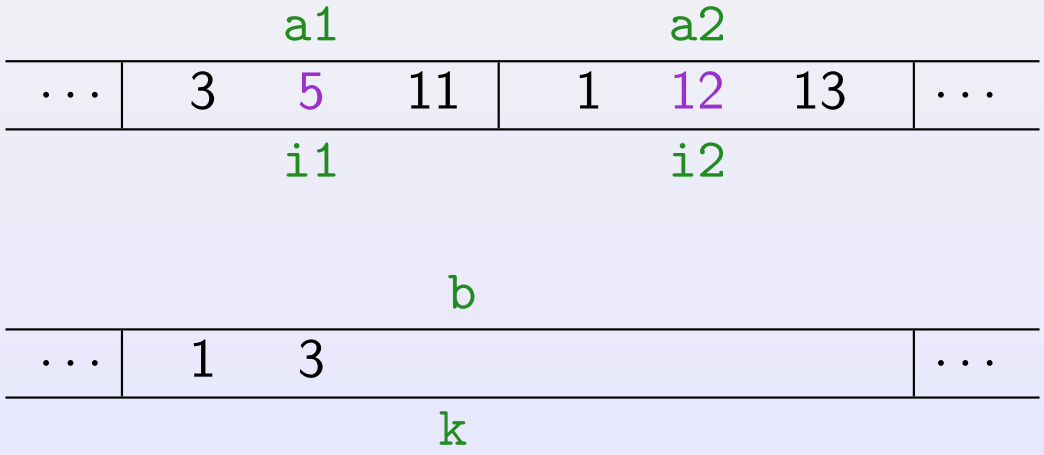
- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.



merging

La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

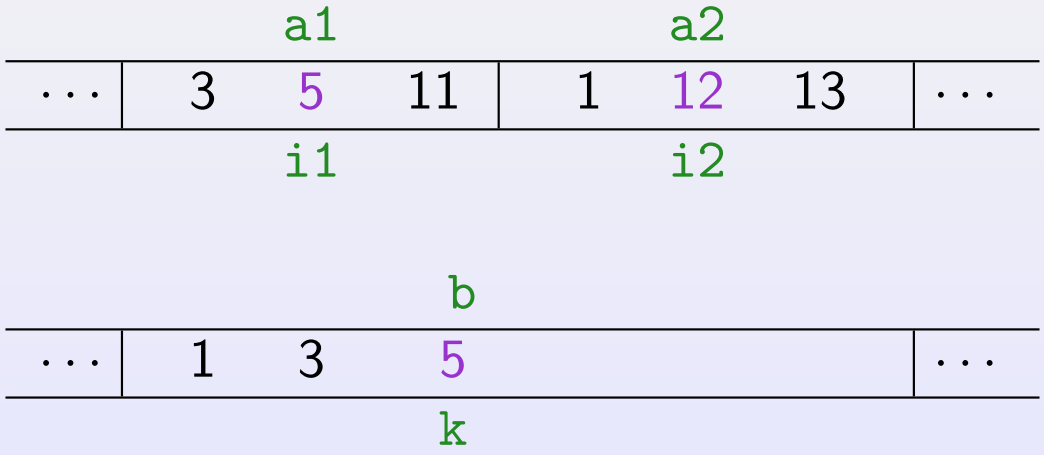
- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.



merging

La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

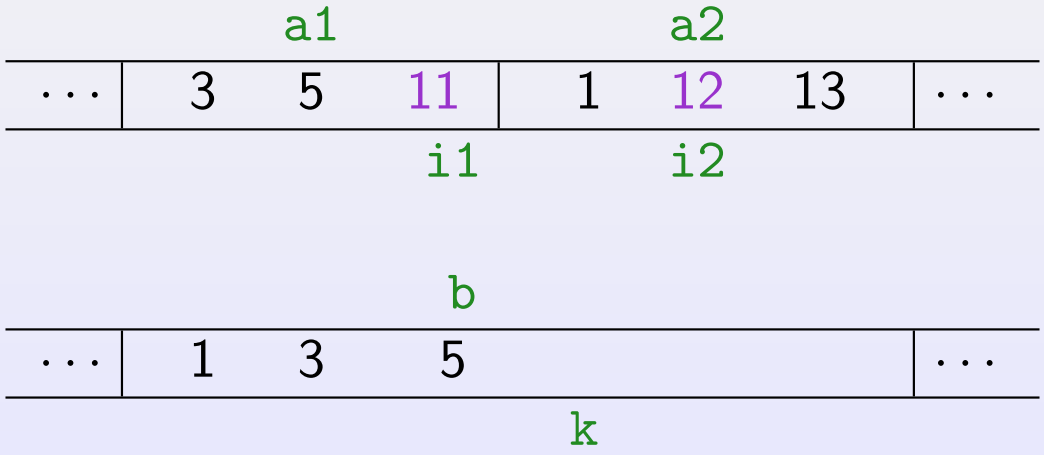
- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.



merging

La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

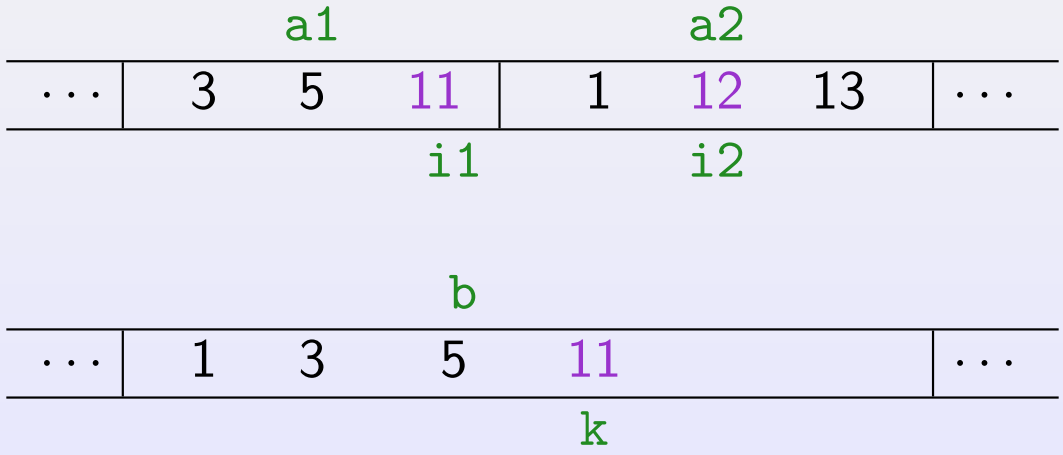
- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.



merging

La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

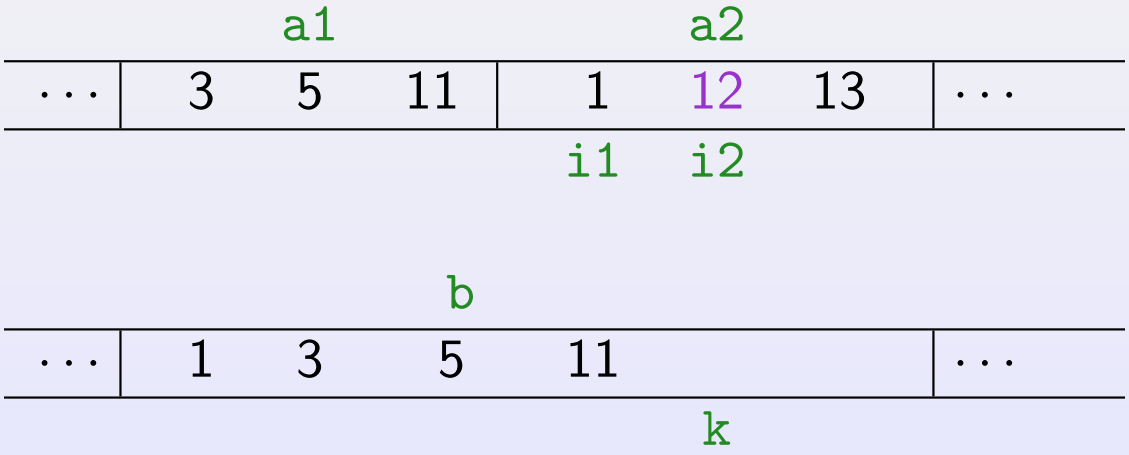
- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.



merging

La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

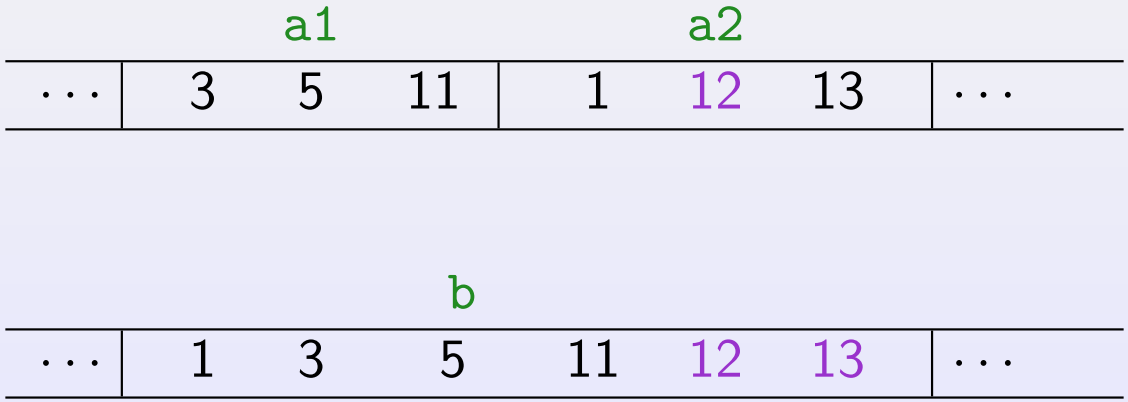
- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.



merging

La parte di integrazione (merge) di due array ordinati `a1` e `a2` funziona con un vettore di supporto `b`:

- si scorrono entrambi gli array da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente;
- ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nell'array di supporto `b` (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso;
- quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro array viene copiata in `b`;
- alla fine si copia il contenuto dell'array `b` nell'array originale.



mergesort (int a[ ], int sx, int dx)

Ordina la parte dell'array *a* compresa tra gli indici *sx* e *dx*, come segue:

1. divide l'array in due sotto-array di dimensione circa uguale;
2. ordina il sotto-array di sinistra richiamando se stessa;
3. ordina il sotto-array di destra richiamando se stessa;
4. integra (merge) i due sotto-array in un unico array ordinato.

La base della ricorsione è data dagli array di lunghezza 0 o 1, che sono sempre ordinati.

mergesort - esempio

```
mergesort( a, 0, 4 )
```

29 15 11 7 13

mergesort - esempio

```
mergesort( a, 0, 4 )
  mergesort( a, 0, 2 )
```

29 15 11 7 13

29 15 11 7 13

mergesort - esempio

```
mergesort( a, 0, 4 )
  mergesort( a, 0, 2 )
    mergesort( a, 0, 1 )
```

29	15	11	7	13
29	15	11	7	13
29	15	11	7	13

mergesort - esempio

`mergesort( a, 0, 4 )`	29	15	11	7	13
`  mergesort( a, 0, 2 )`	29	15	11	7	13
`    mergesort( a, 0, 1 )`	29	15	11	7	13
`      mergesort( a, 0, 0 )`	29	15	11	7	13

mergesort - esempio

`mergesort( a, 0, 4 )`	29	15	11	7	13
`  mergesort( a, 0, 2 )`	29	15	11	7	13
`    mergesort( a, 0, 1 )`	29	15	11	7	13
`      mergesort( a, 0, 0 )`	29	15	11	7	13
`      mergesort( a, 1, 1 )`	29	15	11	7	13

mergesort - esempio

`mergesort( a, 0, 4 )`	29	15	11	7	13
`  mergesort( a, 0, 2 )`	29	15	11	7	13
`    mergesort( a, 0, 1 )`	29	15	11	7	13
`      mergesort( a, 0, 0 )`	29	15	11	7	13
`      mergesort( a, 1, 1 )`	29	15	11	7	13
`      merge( a, 0, 1, 0 )`	15	29	11	7	13

mergesort - esempio

`mergesort( a, 0, 4 )`	29	15	11	7	13
`  mergesort( a, 0, 2 )`	29	15	11	7	13
`    mergesort( a, 0, 1 )`	29	15	11	7	13
`      mergesort( a, 0, 0 )`	29	15	11	7	13
`      mergesort( a, 1, 1 )`	29	15	11	7	13
`      merge( a, 0, 1, 0 )`	15	29	11	7	13
`    mergesort( a, 2, 2 )`	15	29	11	7	13

mergesort - esempio

`mergesort( a, 0, 4 )`	29	15	11	7	13
`  mergesort( a, 0, 2 )`	29	15	11	7	13
`    mergesort( a, 0, 1 )`	29	15	11	7	13
`      mergesort( a, 0, 0 )`	29	15	11	7	13
`      mergesort( a, 1, 1 )`	29	15	11	7	13
`      merge( a, 0, 1, 0 )`	15	29	11	7	13
`    mergesort( a, 2, 2 )`	15	29	11	7	13
`    merge( a, 0, 2, 1 )`	11	15	29	7	13

mergesort - esempio

`mergesort( a, 0, 4 )`	29	15	11	7	13
`  mergesort( a, 0, 2 )`	29	15	11	7	13
`    mergesort( a, 0, 1 )`	29	15	11	7	13
`      mergesort( a, 0, 0 )`	29	15	11	7	13
`      mergesort( a, 1, 1 )`	29	15	11	7	13
`      merge( a, 0, 1, 0 )`	15	29	11	7	13
`    mergesort( a, 2, 2 )`	15	29	11	7	13
`    merge( a, 0, 2, 1 )`	11	15	29	7	13
`  mergesort( a, 3, 4 )`	11	15	29	7	13

mergesort - esempio

`mergesort( a, 0, 4 )`	29	15	11	7	13
`  mergesort( a, 0, 2 )`	29	15	11	7	13
`    mergesort( a, 0, 1 )`	29	15	11	7	13
`      mergesort( a, 0, 0 )`	29	15	11	7	13
`      mergesort( a, 1, 1 )`	29	15	11	7	13
`      merge( a, 0, 1, 0 )`	15	29	11	7	13
`    mergesort( a, 2, 2 )`	15	29	11	7	13
`    merge( a, 0, 2, 1 )`	11	15	29	7	13
`  mergesort( a, 3, 4 )`	11	15	29	7	13
`    mergesort( a, 3, 3 )`	11	15	29	7	13

mergesort - esempio

`mergesort( a, 0, 4 )`	29	15	11	7	13
`  mergesort( a, 0, 2 )`	29	15	11	7	13
`    mergesort( a, 0, 1 )`	29	15	11	7	13
`      mergesort( a, 0, 0 )`	29	15	11	7	13
`      mergesort( a, 1, 1 )`	29	15	11	7	13
`      merge( a, 0, 1, 0 )`	15	29	11	7	13
`    mergesort( a, 2, 2 )`	15	29	11	7	13
`    merge( a, 0, 2, 1 )`	11	15	29	7	13
`  mergesort( a, 3, 4 )`	11	15	29	7	13
`    mergesort( a, 3, 3 )`	11	15	29	7	13
`    mergesort( a, 4, 4 )`	11	15	29	7	13

mergesort - esempio

`mergesort( a, 0, 4 )`	29	15	11	7	13
`  mergesort( a, 0, 2 )`	29	15	11	7	13
`    mergesort( a, 0, 1 )`	29	15	11	7	13
`      mergesort( a, 0, 0 )`	29	15	11	7	13
`      mergesort( a, 1, 1 )`	29	15	11	7	13
`      merge( a, 0, 1, 0 )`	15	29	11	7	13
`    mergesort( a, 2, 2 )`	15	29	11	7	13
`    merge( a, 0, 2, 1 )`	11	15	29	7	13
`  mergesort( a, 3, 4 )`	11	15	29	7	13
`    mergesort( a, 3, 3 )`	11	15	29	7	13
`    mergesort( a, 4, 4 )`	11	15	29	7	13
`    merge( a, 3, 4, 3 )`	11	15	29	7	13

mergesort - esempio

`mergesort( a, 0, 4 )`	29	15	11	7	13
`  mergesort( a, 0, 2 )`	29	15	11	7	13
`  mergesort( a, 0, 1 )`	29	15	11	7	13
`  mergesort( a, 0, 0 )`	29	15	11	7	13
`  mergesort( a, 1, 1 )`	29	15	11	7	13
`  merge( a, 0, 1, 0 )`	15	29	11	7	13
`  mergesort( a, 2, 2 )`	15	29	11	7	13
`  merge( a, 0, 2, 1 )`	11	15	29	7	13
`  mergesort( a, 3, 4 )`	11	15	29	7	13
`  mergesort( a, 3, 3 )`	11	15	29	7	13
`  mergesort( a, 4, 4 )`	11	15	29	7	13
`  merge( a, 3, 4, 3 )`	11	15	29	7	13
`  merge( a, 0, 4, 2 )`	7	11	13	15	29

Variabili di tipo puntatore

Indirizzi di memoria

La memoria è divisa in byte, ognuno dei quali ha un indirizzo. Ogni variabile occupa uno o più byte a seconda del suo tipo.

Variabili puntatore

- Sono variabili che hanno come valore un indirizzo di memoria.
- Se `p` contiene l'indirizzo di memoria in cui si trova la variabile `i`, diciamo che `p` punta a `i`.

Dichiarazione di variabili puntatore

Dato che due variabili di tipo diverso occupano quantità di memoria diversa, è importante specificare che tipo di variabile può puntare un puntatore:

```
int *p;    /* p punta a variabili di tipo int */
int a, b, *p, n[10];  /* dichiarazione composta*/
```

Operatori di indirizzo (&) e indirezione (*)

- L'operatore & consente di ottenere l'indirizzo di memoria di una variabile:

```
int i, *p; /* dichiaro il puntatore a intero p *
p = &i; /* assegno a p l'indirizzo di i */
```

- E' possibile inizializzare una variabile puntatore in fase di dichiarazione:

```
int i;
int *p = &i;
```

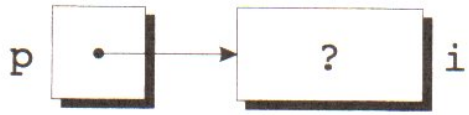
- L'operatore * consente di accedere alla variabile puntata dal puntatore:

```
int i = 3, *p = &i;
printf( "%d", *p ); /* stampa 3 */
```

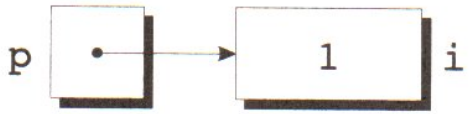
- Cambiando il valore di *p si cambia il valore della variabile puntata!

Esempio

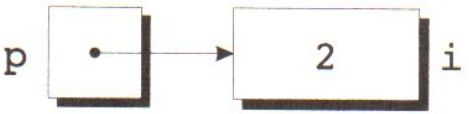
```
p = &i;
```




```
i = 1;
```




```
printf("%d\n", i);        /* stampa 1 */
printf("%d\n", *p);       /* stampa 1 */
*p = 2;
```




```
printf("%d\n", i);        /* stampa 2 */
printf("%d\n", *p);       /* stampa 2 */
```

Assegnamento di puntatori

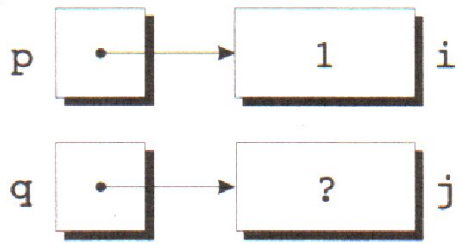
- Attenzione! Se un puntatore `p` non è inizializzato, il suo valore non è definito, quindi non è definito nemmeno `*p`; `p` potrebbe puntare ad uno spazio di memoria qualsiasi, anche riservato al sistema operativo
  $\rightarrow$ segmentation fault!
- La seguente porzione di programma copia il valore di `p` (cioè l'indirizzo di `i`) dentro `q`, ovvero dopo la copia anche `q` punterà alla variabile `i`.

```
int i, j, *p, *q;
p = &i;
q = p; /* copia di puntatori */
```

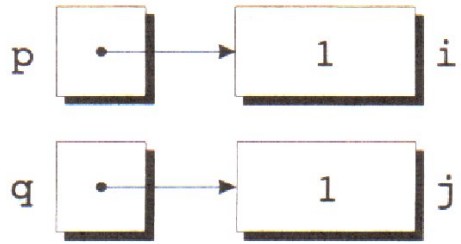
- Cambiando il valore di `*p` si cambia automaticamente anche il valore di `*q`.
- Fate attenzione a non confondere `q = p;` con `*q = *p`: nel secondo caso, il valore della variabile puntata da `p` viene assegnato alla variabile puntata da `q`.

Esempio

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



Puntatori come argomento di funzione

- Passando ad una funzione un puntatore ad una variabile, si può fare in modo che la funzione modifichi il valore della variabile stessa. Invece di passare la variabile `x` dovremo passare come argomento il suo indirizzo, ovvero `&x`.
- Al momento della chiamata il parametro `p` (di tipo puntatore) corrispondente a `&x` verrà inizializzato col valore di `&x` ovvero con l'indirizzo di `x`.

Esempio: l'uso di scanf

Passo alla funzione `scanf` l'indirizzo della variabile `i` di cui voglio cambiare il valore con la chiamata della funzione `scanf` stessa:

```
int i;
scanf( "%d", &i );
```

L'operatore `&` nella `scanf` non è richiesto se come argomento uso un puntatore:

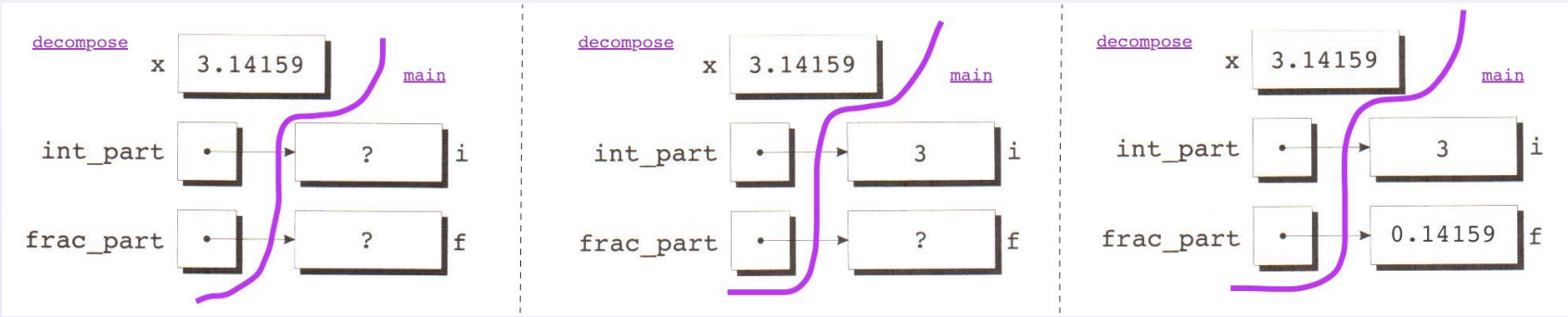
```
int i, *p = &i;
scanf( "%d", p );
```

Esempio

```
void decompose( float x, int *int_part,
                         float *frac_part) {
  *int_part = (int) x;
  *frac_part = x - *int_part;
}
```

Se `i` e `f` sono rispettivamente di tipo `int` e `float`, possiamo effettuare la chiamata

```
decompose( 3.14159, &i, &f);
```



Esempio

```
void split_time( long int tot_sec, int *h,
                 int *m, int *s ) {
  *h = tot_sec / 3600;          tot_sec %= 3600;
  *m = tot_sec / 60;            *s = tot_sec % 60;
}

int main( void ) {
  long int time = 1800;
  int h=0, m=0, s=0;
  split_time( time, &h, &m, &s );
  printf( "h = %d, m = %d, s = %d\n", h, m, s );
  return 0;
}
```

Puntatori come valori restituiti

E' possibile scrivere funzioni che restituiscono puntatori. Ad esempio:

```
int *max( int *a, int *b) {
  if ( *a > *b )
    return a;
  else
    return b;
}
```

Per invocare la funzione `max`, le passiamo due puntatori a variabili `int` e salviamo il risultato in una variabile puntatore:

```
int *p, i, j;
...
p = max( &i, &j);
```

Attenzione: non restituite mai un puntatore ad una variabile locale (a meno di averla dichiarata static)!

Puntatori a strutture

Accedere ai membri di una struttura usando un puntatore è un'operazione molto frequente, tanto che il linguaggio C fornisce l'operatore `->` specifico per questo scopo:

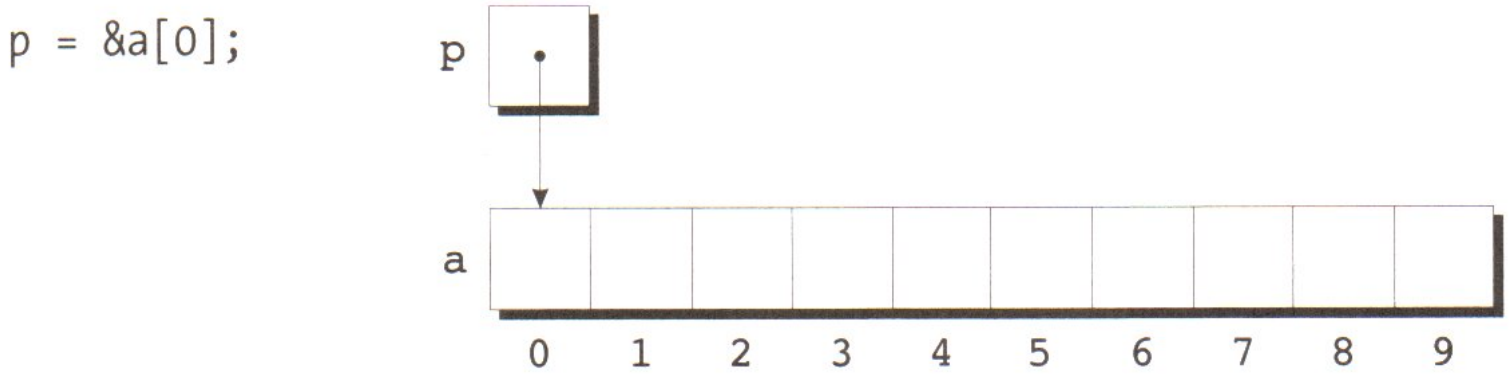
```
typedef struct {
  float x, y;
} punto;

typedef struct {
  punto p1, p2;
} rettangolo;

/* stampa i vertici che def. il rett. puntato da r*/
void stampa( rettangolo *r ) {
  printf( "Rett. di vertici (%f, %f) e (%f, %f).\n",
            r -> p1.x, r -> p1.y,
            r -> p2.x, r -> p2.y );
}
```

Aritmetica dei puntatori

Dichiarati

```
int a[10], *p;
```

possiamo fare in modo che un puntatore `p` punti ad `a[0]`:

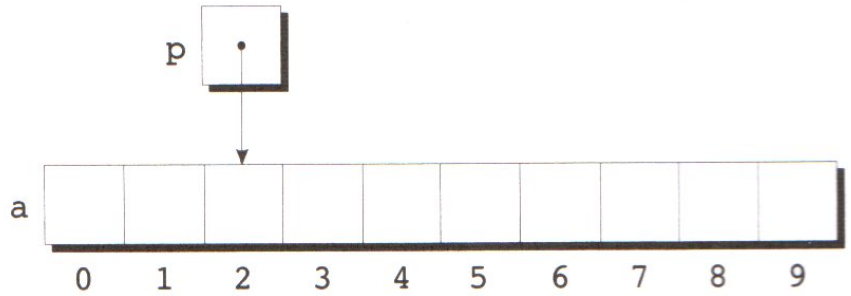


Usando i puntatori, possiamo anche accedere agli altri elementi di `a` usando l'aritmetica dei puntatori che prevede 3 operazioni:

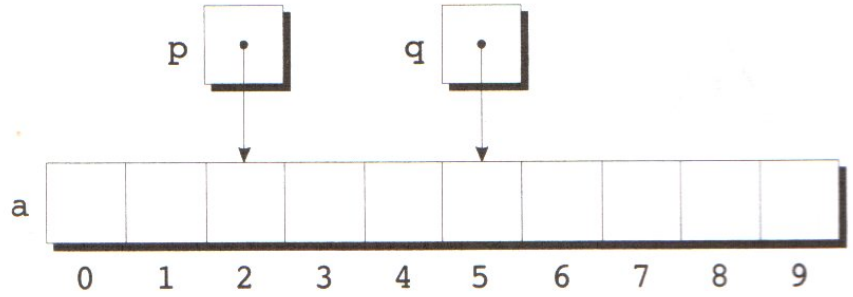
- sommare un intero a un puntatore;
- sottrarre un intero a un puntatore;
- sottrarre da un puntatore un altro puntatore.

Sommare un intero a un puntatore

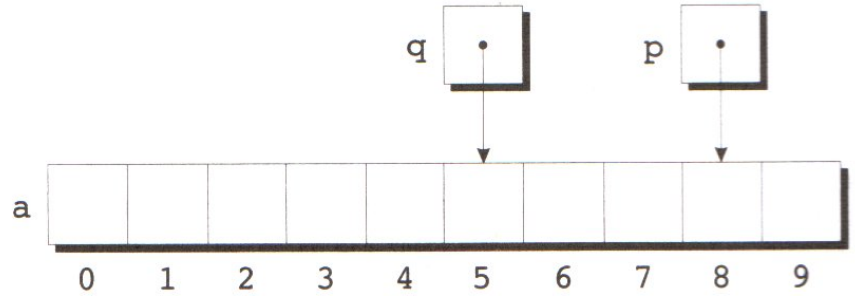
```
p = &a[2];
```




```
q = p + 3;
```

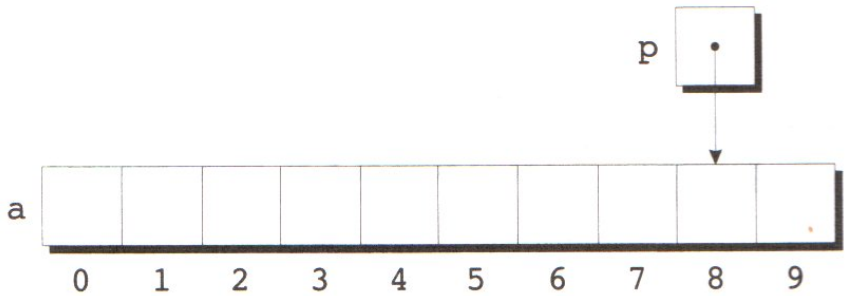



```
p += 6;
```

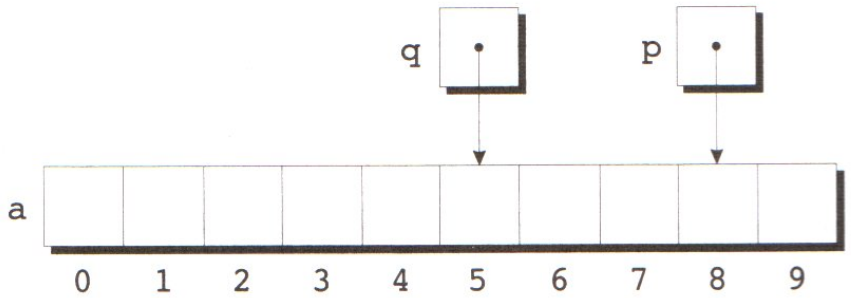


Sottrarre un intero a un puntatore

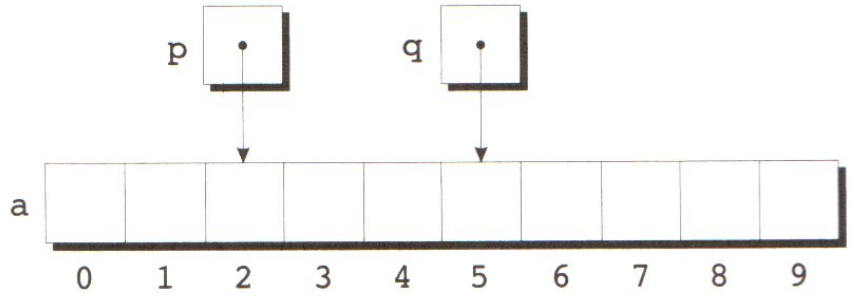
```
p = &a[8];
```




```
q = p - 3;
```

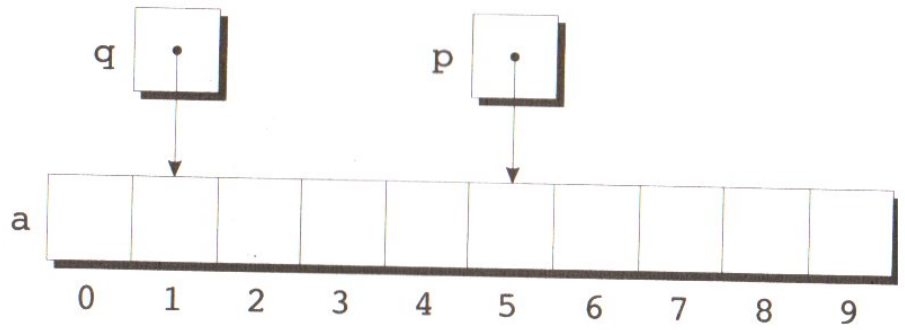



```
p -= 6;
```



Sottrarre da un puntatore un altro puntatore

```
p = &a[5];
q = &a[1];


i = p - q;/* i è uguale a 4 */
i = q - p;/* i è uguale a -4 */
```



Puntatori e array

Uso di puntatori per scorrere array

```
int a[N], *p, sum = 0;

for ( p = &a[0]; p < &a[N]; p++ )
   sum += *p;
```

Combinazione tra * e ++

- `*p++` equivale a `*(p++)`: prende il valore dell'oggetto puntato da p, poi incrementa il puntatore;
- `(*p)++`: prende il valore dell'oggetto puntato da p, poi incrementa tale valore;
- `*++p`: incrementa p, poi prende il valore dell'oggetto puntato;
- `++*p`: incrementa il valore dell'oggetto puntato e prende il valore incrementato

Nomi di array come puntatori costanti

Il nome di un array può essere usato come puntatore costante al primo elemento dell'array. `a[i]` corrisponde a `*(a + i)`.

```
int a[N];
*a = 7;             /* salva 7 in a[0] */
*(a + 1) = 12       /* salva 12 in a[1] */
```

Uso di puntatori per scorrere array (rivisitato)

```
int a[N], *p, sum = 0;
for ( p = a; p < a + N; p++ )
  sum += *p;
```

Attenzione: non posso cambiare il valore di un array!

```
a++;            /* SBAGLIATO! */
a = p;          /* SBAGLIATO! */
```

Array come argomenti di funzioni

Il nome di un array argomento di funzione è sempre considerato come un puntatore.

- l'array non viene copiato (maggiore efficienza);
- l'array non è protetto da cambiamento (usare `const`);
- non c'è modo di sapere quanto è lungo l'array;
- è possibile passare porzioni di array:

```
somma( a, n );
somma( &a[5], n );
```

- il parametro può essere definito indifferentemente
  - come array `int a[]`
  - o come puntatore `int *a`

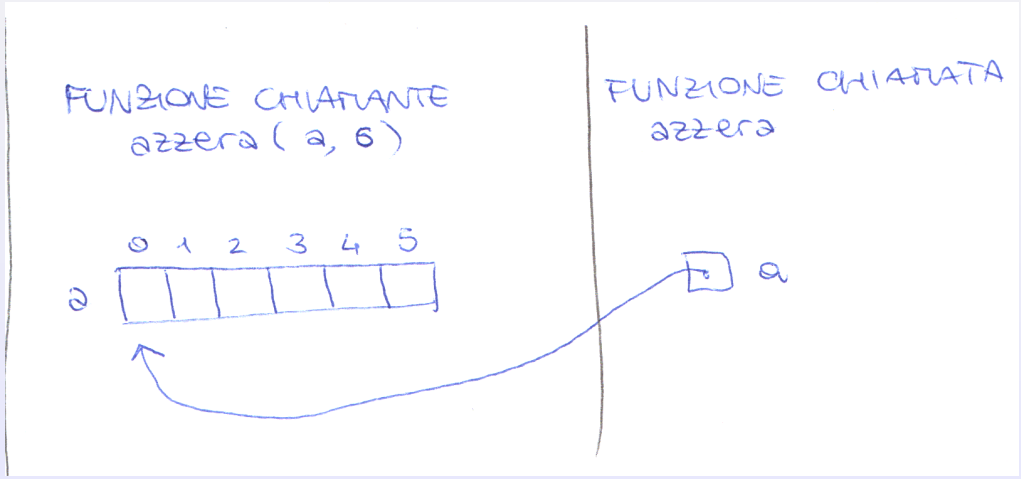
Attenzione: nelle dichiarazioni di variabili invece c'è differenza tra queste due dichiarazioni:

```
int a[N];    /* alloca lo spazio per N interi */
int *a;      /* alloca lo spazio solo per
                un puntatore a intero */
```

Esempio

```
void azzera( int a[], int n) {
  int i;
  for ( i = 0; i < n; i++ )
    a[i] = 0;
}
```

Il vettore `a` viene trattato come puntatore, quindi non viene copiato, ma viene passato alla funzione l'indirizzo del suo primo elemento `&a[0]`. La funzione azzera quindi direttamente gli elementi di `a` e non di una copia locale!



Array come argomenti di funzione

- Se il vettore è unidimensionale, la dimensione può essere omessa.

```
int f( int a[] ) { ... }
```

- Se il vettore è multidimensionale, solo la prima dimensione può essere omessa.

```
int f( int a[][LUN] ) { ... }
```

- la funzione non ha modo di sapere quanto è lungo il vettore (l'operatore `sizeof` non può essere usato...) quindi può essere utile passare la lunghezza come parametro aggiuntivo:

```
int somma_array( int a[], int n ) {
  int i, sum = 0;
  for ( i = 0; i < n; i++ )
    sum += a[i];
  return sum;
}
```

Stringhe e puntatori

- Ogni letterale stringa (es: `"ciao"`) è un array di caratteri, quindi è trattato come puntatore a carattere.
- Ad esempio il primo parametro nel prototipo della funzione printf è un puntatore a carattere:

```
int printf( const char *format, ...);
```

- Se una stringa è argomento di una funzione, alla chiamata essa non viene copiata, ma viene passato l'indirizzo della sua prima lettera.

Stringhe e puntatori - attenzione alle dichiarazioni

Dichiarazione come array

```
char data[] = "13␣maggio";
```

- data è un vettore che contiene i caratteri `'1'`, `'3'`, ...
- i singoli caratteri possono essere modificati (Es: `data[1] = '4'`);

Dichiarazione come puntatore

```
char *data = "13␣maggio";
```

- il letterale costante `"13␣maggio"` è memorizzato in un array;
- l'inizializzazione fa si che `data` punti al letterale costante;
- il puntatore `data` può essere modificato in modo che punti altrove.

File di intestazione `string.h`

Copia di stringhe

Non si possono usare assegnamenti tipo `str = "abcd"`; usiamo la funzione

```
char *strcpy(char *dest, const char *src);
```

che copia `src` in `dest` e ne restituisce l'indirizzo.
Esempio: `sctrcpy(str, "abcd")` copia `"abcd"` in `str`.

Concatenazione di stringhe

```
char *strcat(char *dest, const char *src);
```

aggiunge il contenuto di `src` alla fine di `dest` e restituisce `dest` (ovvero il puntatore alla stringa risultante).

Confronto tra stringhe

```
int strcmp(const char *s1, const char *s2);
```

restituisce un valore maggiore, uguale o minore di 0 a seconda che `s1` sia maggiore, uguale o minore di `s2`.

Esempio: calcolare la lunghezza di una stringa

```c
/* Prima versione */
int lun_stringa( const char *s ) {
  int n = 0;
  while ( *s != '\0' ) {
    n++; s++;
  }
  return n;
}
```

```c
/* Seconda versione */
int lun_stringa( const char *s ) {
  int n = 0;
  while ( *s++ != '\0' )
    n++;
  return n;
}
```

Esempio: calcolare la lunghezza di una stringa - continua

```c
/* Terza versione */
int lun_stringa( const char *s ) {
  int n = 0;
  while ( *s++ )
    n++;
  return n;
}
```

```c
/* Quarta versione */
int lun_stringa( const char *s ) {
  const char *p = s;
  while ( *s++ )
    ;
  return s - p - 1;
}
```

Array di stringhe

Un vettore bidimensionale di caratteri può essere pensato come un array di stringhe di lunghezza costante: ogni riga contiene una stringa della stessa lunghezza.

```
char colori1[6][10] = {"rosso", "blu", "oltremare"
                       "verde", "nero", "giallo"};
```

- Il numero di colonne è pari almeno alla lunghezza massima delle stringhe (incluso il fine stringa): in questo esempio occupo lo spazio per $6 \times 10 = 60$ caratteri.
- Buona parte di questo spazio è occupata dal carattere nullo `'\0'`.
- NB: se si usa un vettore bidimensionale come parametro di funzione, è necessario specificare la seconda dimensione, ad esempio:

```
void stampa( char c[][10] );
```

Array di stringhe - 2

Il modo più corretto (ed efficiente) di gestire un insieme di stringhe di lunghezza variabile è attraverso l'uso di un array frastagliato, ovvero di un array di puntatori a char:

```
char *colori2[6] = {"rosso", "blu", "oltremare"
                    "verde", "nero", "giallo"};
```

- In questo caso occupo lo spazio di 6 puntatori a char, più lo spazio strettamente necessario a contenere le 6 stringhe, pari a $(5+1)+(3+1)+(9+1)+(5+1)+(4+1)+(6+1) = 38$ caratteri.
- Il nome di un array di stringhe può essere considerato come un puntatore a stringa. Ad esempio, come parametro di funzione si può usare indifferentemente: `char *c[]` oppure `char **c`.
- NB: nei parametri di funzione devo usare dichiarazioni diverse se uso array frastagliati o array bidimensionali: `char c[][10]` non equivale a `char **c`!!

Argomenti da linea di comando

Ogni programma C può avere degli argomenti passati da linea di comando. Per poterli usare è necessario che la funzione `main` sia definita con due parametri, chiamati solitamente `argc` e `argv`.

```
int main ( int argc, char *argv[] ) { ... }
```

oppure

```
int main ( int argc, char **argv ) { ... }
```

- `argc` è pari al numero degli argomenti (incluso il nome del comando);
- `argv` è un array frastagliato di stringhe, di lunghezza `argc`+1:
  - `argv[0]` è il nome del comando;
  - `argv[1]` è il primo argomento;
  - `argv[argc-1]` è l'ultimo argomento;
  - `argv[argc]` è il puntatore `NULL` (...);

Argomenti da linea di comando - esempio

```c
/* somma.c - programma che somma i suoi argomenti */
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] ) {
  int i, somma = 0;
  for ( i = 1; i < argc; i++ )
    somma += atoi( argv[i] );

  printf( "Somma = %d\n", somma );
  return 0;
}
```

```
$ gcc -o somma somma.c
$ ./somma 1 3 10
Somma = 14
```

Automatic vs static storage duration

Attraverso le variabili, il C gestisce la memoria staticamente o automaticamente:

- le variabili con storage duration static sono allocate in memoria principale all'inizio dell'esecuzione del programma e persistono per tutta l'esecuzione del programma
  - es: variabili globali
- le variabili con storage duration automatic sono allocate sullo stack, all'interno dei record di attivazione delle chiamate di funzione; queste variabili *vanno e vengono*, cioè perdono il loro valore quando termina l'esecuzione del blocco in cui sono dichiarate e non persistono tra una chiamata e l'altra.
  - es: variabili locali definite all'interno di un blocco

NB: per blocco si intende:

- il corpo di una funzione, oppure
- una sequenza di istruzioni e dichiarazioni raccolte tra graffe

Secondo lo standard ANSI le dichiarazioni vanno all'inizio del blocco.

Allocazione dinamica della memoria

In C la memoria può essere anche gestita in modo dinamico, attraverso l'allocazione esplicita di blocchi di memoria di data dimensione:

- tali blocchi sono allocati tipicamente in una parte della memoria chiamata heap;
- è possibile accedere a tali blocchi di memoria attraverso l'uso di puntatori;
- lo spazio allocato dinamicamente non viene liberato all'uscita delle funzioni;
- sempre con l'uso di puntatori la memoria che non serve più va deallocata in modo da renderla nuovamente disponibile.

Allocazione dinamica della memoria

A cosa serve?

- Per allocare di vettori e/o stringhe con lunghezza non nota in fase di compilazione, ma calcolata durante l'esecuzione. ($\rightarrow$ C99)
- Per gestire strutture dati che crescono e si restringono durante l'esecuzione del programma (es: liste).
- Per avere maggiore flessibilità nel gestire la durata delle variabili.

Quattro funzioni fondamentali:

```
void *malloc( size_t size );
void *calloc( size_t nmemb, size_t size );
void *realloc( void *p, size_t size );
void free( void *p);
```

I prototipi sono contenuti nel file di intestazione stdlib.h.

Puntatore NULL

Quando viene chiamata una funzione per l'allocazione dinamica della memoria , c'è sempre la possibilità che non ci sia spazio sufficiente per soddisfare la richiesta. In questo caso, la funzione restituisce un puntatore nullo, ovvero un puntatore che "non punta a nulla".

Nota bene

Avere un puntatore nullo è diverso da avere un puntatore non inizializzato, o un puntatore di cui non si conosce il valore!!

- Il puntatore nullo è rappresentato da una macro chiamata NULL, di valore 0, dichiarata in stdlib.h, stdio.h, string.h e altri.
- I puntatori possono essere usati nei testi: NULL ha valore falso (vale 0!), mentre ogni puntatore non nullo ha valore vero (è diverso da NULL, cioè da 0!)

```
if ( p == NULL) ...
if( !p ) ...
```

Malloc

La funzione

```
void *malloc( size_t size );
```

alloca un blocco di memoria di `size` byte e restituisce un puntatore all'inizio di tale blocco.

- `size_t` è un tipo definito nella libreria standard (di solito corrisponde ad `unsigned int`);
- il blocco di memoria allocato può contenere valori di tipo diverso, il puntatore di tipo generico `void *` permette di gestire tutti i casi;
- in caso di assegnamento il puntatore restituito dalla malloc viene convertito implicitamente (alcuni esplicitano il cast);
- sul blocco di memoria allocato è possibile usare i puntatori con l'usuale aritmetica.

Malloc - esempi

```
p = malloc( 10000 );
if ( p == NULL ) {
  /* allocazione fallita;
     provvedimenti opportuni */
  ...
}
```

Stringhe allocate dinamicamente

```
/* alloca lo spazio per una stringa di n caratteri
un char occupa sempre un byte! */
  char *p;
  int n;
  ...
  p = malloc( n + 1 );
```

Esempio - restituire un puntatore ad una "nuova" stringa

Il seguente programma concatena le due stringhe `s1` e `s2` in una nuova stringa di cui restituisce l'indirizzo (ovvero un puntatore che punta ad essa).

```
char *concat( const char *s1, const char *s2)  {
  char *result;

  result = malloc( strlen(s1) + strlen(s2) + 1 );
  if ( result == NULL ) {
    print( "malloc␣failure\n" );
    exit(EXIT_FAILURE);
  }

  strcpy( result, s1 );
  strcat( result, s2 );
  return result;
}
```

```
p = concat( "abc", "def");
```

Vettori allocati dinamicamente

Si può usare `malloc` anche per allocare spazio per un vettore (come per le stringhe). La differenza è che gli elementi dell'array possono occupare più di un byte (a differenza dei `char`).

```
int *a, i, n;

/* alloca lo spazio per un array di n interi */
a = malloc( n * sizeof(int) );

/* inizializza l'array a 0 */
for ( i = 0; i < n; i++ )
  a[i] = 0;
```

Calloc

```
void *calloc( size_t nmemb, size_t size );
```

alloca spazio per un array di `nmemb` elementi, ciascuno di dimensione `size`, li inizializza a 0 e restituisce il puntatore al primo elemento (oppure `NULL`).

Esempio

A volte può essere comodo usare calloc con primo argomento pari a 1, in questo modo è possibile allocare e inizializzare anche oggetti diversi da un array.

```
struct point{ float x, y } *p;
p = calloc( 1, sizeof( struct point) );
```

Alla fine dell'esecuzione di queste istruzioni, `p` punterà ad una struttura di tipo `point` i cui membri `x` e `y` sono inizializzati a 0.

Realloc

```
void *realloc( void *p, size_t size );
```

ridimensiona lo spazio puntato da `p` alla nuova dimensione `size` e restituisce il puntatore al primo elemento (oppure `NULL`):

- il puntatore `p` deve puntare ad un blocco di memoria già allocato dinamicamente, altrimenti il comportamento è indefinito;
- tendenzialmente `realloc` cerca di ridimensionare il vettore in loco, ma se non ha spazio può allocare nuovo spazio altrove, copiare il contenuto del vecchio blocco nel nuovo e restituire l'indirizzo del nuovo blocco;
- attenzione ad aggiornare eventuali altri puntatori dopo la chiamata di `realloc` perchè il blocco potrebbe essere stato spostato!

Free

Quando un blocco di memoria allocato dinamicamente non serve più, è importante deallocarlo e renderlo nuovamente disponibile usando la funzione

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

L'argomento di free deve essere stato allocato dinamicamente, altrimenti il comportamento è indefinito.

Errori tipici: fallimento nell'allocazione

E' importante sempre verificare che l'allocazione abbia avuto successo e il puntatore restituito non sia `NULL`.
In caso contrario si rischia di usare il puntatore `NULL` come se puntasse a memoria allocata, e si provocherebbero errori.

```
char *ptr;
ptr = malloc(10);
*ptr = 'a';
/* RISCHIOSO: se malloc restituisce NULL... */
```

Errori tipici: dangling pointer

Dopo la chiamata `free(p)`, il blocco di memoria puntato da `p` viene deallocato, ma il valore del puntatore `p` non cambia; eventuali usi successivi di `p` possono causare danni!

```
char *p = malloc(4);
...
free(p);
...
strcpy(p, "abc"); /* SBAGLIATO! */
```

Si dice in questo caso che `p` è un dangling pointer (letteralmente: puntatore ciondolante).

Errori tipici: memory leak - esempio 1

```
p = malloc(...);
q = malloc(...);
p = q;
```

L'oggetto puntato da `p` prima dell'ultimo assegnamento non è più raggiungibile! Quel blocco di memoria resterà allocato ma non utilizzabile. Si parla in questo caso di memory leak.
Prima di effettuare l'assegnamento `p = q;` bisogna deallocare il blocco puntato da `p`:

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

Errori tipici: memory leak - esempio 2

E' importante usare sempre un puntatore temporaneo per il valore di ritorno di realloc. In caso contrario può succedere che il puntatore originario venga trasformato in `NULL`. Ad esempio:

```
int *ptr, *tmp, size = N;
ptr = malloc(size);
...
/* vogliamo raddoppiare l'area allocata */
size *= 2;
tmp = realloc(ptr, size);
if ( tmp != NULL )
  ptr = tmp;
```

Errori tipici: memory leak - esempio 3

```
void f(void) {
  void* s = malloc( 50 );
}

int main(void) {
     while (1) f();
}
```

Ad ogni chiamata di `f`, la memoria viene allocata e puntata da `s`.
Quando la funzione restituisce il controllo al main, lo spazio rimane allocato, ma `s` viene distrutta quindi la memoria allocata diventa irrangiungibile.
Prima o poi la memoria verrà esaurita!
Il codice va corretto in uno dei seguenti modi:

- aggiungere l'istruzione `free(s)` alla fine di `f`
- far si che `f` restituisca `s` alla funzione chiamante, la quale si dovrà preoccupare di deallocare lo spazio.

Lettura di una riga con allocazione di memoria

La seguente funzione legge da standard input una sequenza di caratteri terminata da `\n` e la memorizza in una stringa di dimensione opportuna allocata dinamicamente.

- La dimensione della stringa viene incrementata man mano che vengono letti i caratteri: all'inizio viene allocato lo spazio per 2 caratteri; quando lo spazio è tutto occupato, la dimensione viene raddoppiata.
- Uso due variabili intere: `size` rappresenta la dimensione allocata; `n` rappresenta il numero di caratteri letti.
- Se `n >= size`, bisogna allocare nuovo spazio!

```
char *read_line( void ) {

  char *p, c;
  int n = 0, size = 2;

  p = my_malloc( size );

  while ( ( c = getchar() ) != EOF ) {

    if ( n >= size ) { /* spazio terminato, lo raddoppio */
      size *=2;
      p = my_realloc( p, size );
    }

    if ( c == '\n' ) { /* fine stringa, interrompo */
      p[n] = '\0';
      break;
    }

    p[n++] = c;
  }

  return p;
}
```

Lettura di una parola con allocazione di memoria

La funzione precedente può essere modificata in modo da leggere una sola parola (fino al primo carattere non alfabetico) memorizzandola in una stringa di dimensione opportuna allocata dinamicamente.
Al posto di:

```
if ( c == '\n' )
```

devo scrivere:

```
if ( !isalpha( c  ) )
```

Allocazione dinamica di una matrice bidimensionale I

La seguente funzione alloca lo spazio per una matrice bidimensionale di caratteri e la inizializza con il carattere '.'

```
char **creaMatrice( int n ){
  char **m;
  int r, c;

  m = malloc( n * sizeof( char * ) );
  for ( r = 0; r < n; r++ ) {
    *(m+r) = malloc( n * sizeof( char ) );
  }

  for ( r = 0; r < n; r++ )
    for ( c = 0; c < n; c++ )
      m[r][c] = '.';
  return m;
}
```

Allocazione dinamica di una matrice bidimensionale II