Allocazione dinamica della memoria

Il ciclo di vita della memoria allocata a una variabile è controllato dal sistema e determinato dalla classe di memorizzazione della variabile stessa.

E' possibile gestire direttamente il ciclo di vita di una zona di memoria per mezzo del meccanismo di

allocazione e *deallocazione dinamica*.

Il programma può richiedere al sistema di allocare il quantitativo di memoria specificato. Se tale richiesta ha successo, il sistema ritorna l'indirizzo del primo byte di tale zona.

La memoria così ottenuta rimane allocata fino a quando il processo termina oppure fino a quando il programma non comunica esplicitamente al sistema che essa può essere rilasciata e resa nuovamente disponibile.

La regione di memoria adibita dal sistema alla gestione delle richieste di allocazione/deallocazione dinamica — lo *heap* — solitamente è separata dalla regione di memoria usata per le variabili.

Poiché il controllo della memoria allocata dinamicamente spetta unicamente al programma, il suo uso è fonte di frequenti errori di programmazione che consistono nel tentativo di accedere una zona non ancora allocata, oppure già deallocata ed eventualmente riallocata ad un'altra risorsa.

Le funzioni `malloc`, `calloc`, `free`.

I prototipi di queste funzioni e la definizione del tipo `size_t` sono in `stdlib.h`

`size_t` coincide con `unsigned int` sulla maggior parte dei sistemi.

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

Richiede al sistema di allocare una zona di memoria di `size` byte. Se la richiesta ha successo viene restituito un puntatore di tipo `void *` che contiene l'indirizzo del primo byte della zona allocata. Se la richiesta fallisce viene restituito il valore `NULL`.

Un puntatore di tipo `void *` è considerato un puntatore di tipo "*generico*": esso può essere automaticamente convertito a puntatore al tipo T, per ogni tipo T.

L'uso corretto di `malloc` prevede l'uso contestuale dell'operatore `sizeof`.

```
int *pi = malloc(sizeof(int));
```

`malloc` richiede al sistema l'allocazione di una zona di memoria che possa contenere esattamente un valore di tipo `int`. L'indirizzo ottenuto (oppure `NULL`) viene assegnato, senza dover usare una conversione esplicita, alla variabile `pi` di tipo puntatore a `int`.

```
double *pd;

if( !( pd = malloc(5 * sizeof(double)) ) )
    error(...)
else
    use(pd);
```

`malloc` richiede l'allocazione di una zona che contenga 5 oggetti di tipo `double`. Si controlla che la richiesta abbia avuto successo.

`void *calloc(size_t n_elem, size_t elem_size);`
Richiede al sistema l'allocazione di un array di `n_elem` elementi di dimensione `elem_size`.

Se la richiesta ha successo:
– gli elementi vengono inizializzati a `0`.
– viene restituito l'indirizzo del primo byte del primo elemento.

Se la richiesta fallisce: viene restituito `NULL`.

```
double *pd;

if( !( pd = calloc(5, sizeof(double)) ) )
    error(...);
else
    use(pd);
```

`malloc` non inizializza a `0` la zona allocata.

La memoria allocata con `malloc` o `calloc` non viene deallocata all'uscita dal blocco o dalla funzione.

Per deallocarla bisogna inoltrare una richiesta esplicita al sistema:

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

– Se `p` punta al primo byte di una zona di memoria allocata dinamicamente (da `malloc`, `calloc` o `realloc`) (e non ancora rilasciata (!)), allora tutta la zona viene rilasciata.
– Se `p` è `NULL`, `free` non ha effetto.
– Se `p` non è l'indirizzo di una zona di memoria allocata oppure è l'indirizzo di una zona già rilasciata, il processo cade in una situazione inconsistente, dagli effetti imprecisati: E' un errore da evitare.

Si noti che `p` non viene posto a `NULL` da `free(p)`.

Per modificare dinamicamente la dimensione di una zona di memoria allocata precedentemente (e non ancora rilasciata):

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

Se `p` è l'indirizzo base di una zona di memoria allocata di dimensione `sizeold`, `realloc` restituisce l'indirizzo di una zona di memoria di dimensione `size`.
—— – Se `sizeold <= size` il contenuto della vecchia zona non viene modificato, e lo spazio aggiuntivo non viene inizializzato.
—— – Se `size < sizeold` il contenuto dei primi `size` byte della vecchia zona non vengono modificati.

– `realloc`, se è possibile, non cambia l'indirizzo base della zona: in questo caso il valore ritornato è, per l'appunto, `p`.
– Se questo non è possibile, `realloc` alloca effettivamente una zona diversa e vi copia il contenuto della zona vecchia, poi dealloca quest'ultima e restituisce l'indirizzo base della nuova.

Esempio: Giro del cavallo modificato:
La dimensione del lato (e quindi della matrice che rappresenta la scacchiera) viene specificata da input, e la matrice allocata dinamicamente:

```
#include <stdio.h>
#include <stdlib.h>
short *matrix;  /* Nota: matrix e' short *, non piu' un array bidimensionale di short */
...
int main(void)
{
    short x,y,n,N;

    printf("\nImmetti dimensione lato:\n");
    scanf("%hd",&N);
    if(!(matrix = calloc(N * N, sizeof(short)))){fprintf(stderr,"Errore di allocazione\n");exit(-1);}
    x = y = 0;
    matrix[0] = n = 1;
    if(backtrack(x,y,1,N))            /* Si comunica a backtrack anche la dimensione del lato */
        for(y = 0; y < N; y++) {
            for(x = 0; x < N; x++)
                printf("[%2d]",matrix[x * N + y]); /* Nota come si accede ora a matrix[x][y] */
            putchar('\n');
        }
    free(matrix);
}
```

Esempio: Mergesort.

```
void mergesort(int *a, int *b, int l, int r)
{
    int i,j,k,m;

    if(r > l) {
        m = (r+l)/2;
        mergesort(a,b,l,m);
        mergesort(a,b,m+1,r);
        for(i = m+1; i > l; i--)
            b[i-1] = a[i-1];
        for(j = m; j < r; j++)
            b[r+m-j] = a[j+1];
        for(k = l; k <= r; k++)
            a[k] = b[i] < b[j] ? b[i++] : b[j--];
    }
}
void ordina(int *a, int l, int r)
{
    int *b = calloc(r - l + 1, sizeof(int));

    if(!b) { fprintf(stderr,"Errore di allocazione\n"); exit(-1); }
    mergesort(a,b,l,r);
    free(b);
}
```

`void ordina(int *a, int l, int r)`
L'algoritmo di ordinamento `mergesort` è implementato ricorsivamente. Ha inoltre bisogno di un array `b[]` di appoggio per ordinare l'array `a[]`. la funzione `ordina` crea l'array ausiliario `b[]` e richiama `mergesort(a,b,l,r)`.

`int *b = calloc(r - l + 1, sizeof(int));`
Si alloca dinamicamente un array di `r - l + 1` elementi `int` e se ne assegna l'indirizzo base a `b`.

`if(!b) {...}`
Si controlla che l'allocazione dinamica abbia avuto successo.

`free(b);`
Si rilascia la zona di memoria allocata dinamicamente il cui indirizzo base è contenuto in `b`.

Lo schema generico dell'algoritmo ricorsivo `mergesort` è:

```
void mergesort(int *a, int l, int r)
{
    int m;

    if(r > l) {
        m = (r+l)/2;      /* NB: e' il Floor del razionale (r+l)/2 */
        mergesort(a,l,m);    /* sulla prima meta'   */
        mergesort(a,m+1,r);  /* sulla seconda meta' */
        merge(a,l,m,r);      /* combina i due sottoarray ordinati */
    }
}
```

Dove la funzione `merge` deve "*fondere*" in un unico array ordinato i due sottoarray di `a` già ordinati.

E' un classico esempio di tecnica *Divide et Impera*.

La corrispondente equazione di ricorrenza è:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T_{\mathtt{merge}}(n) & \text{otherwise.} \end{cases}$$

Poiché, come vedremo,

$$T_{\mathtt{merge}}(n) = \Theta(n)$$

si può riscrivere:

$$T(n) = 2T(n/2) + \Theta(n)$$

La cui soluzione è

$$T(n) = \Theta(n \log_2 n).$$

Supponendo di fondere nell'array `c` i due array ordinati `a` di `m` elementi e `b` di `n` elementi:

```
void merge(int *a, int *b, int *c, int m, int n)
{
    int i = 0, j = 0, k = 0;

    while(i < m && j < n)     /* in c[k] il piu' piccolo tra a[i] e b[j] */
        if(a[i] < b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    while(i < m)              /* esaurito b[], si accoda la parte restante di a[] */
        c[k++] = a[i++];
    while(j < n)
        c[k++] = b[j++];      /* esaurito a[], si accoda la parte restante di b[] */
}
```

Fino a quando non si è esaurito almeno uno degli array `a[]` e `b[]`, viene posto in `c[k]` il più piccolo fra `a[i]` e `b[j]`. Gli indici sono incrementati di conseguenza.
Infine, avendo esaurito uno tra `a[]` e `b[]` si accoda a `c[]` quanto rimane dell'altro.

Nel nostro caso si realizza la fusione (`merge`) nell'array `a[]` delle due parti dell'array ausiliario `b[]`.
Per prima cosa bisogna istanziare opportunamente l'array `b[]`:

```
for(i = m+1; i > l; i--)
    b[i-1] = a[i-1];
for(j = m; j < r; j++)
    b[r+m-j] = a[j+1];
```

Gli elementi da `a[l]` ad `a[m]` sono copiati nei corrispondenti elementi di `b[]`. Alla fine `i == l`.
Gli elementi da `a[m+1]` ad `a[r]` sono copiati negli elementi `b[m]`,. . . ,`b[r]`, ma *in ordine inverso*. Alla fine `j = r`.

```
for(k = l; k <= r; k++)
    a[k] = b[i] < b[j] ? b[i++] : b[j--];
```

Queste righe realizzano la fusione. Il trucco di aver copiato la seconda parte in ordine inverso e il fatto che la scandiamo da destra a sinistra (`b[j--]`), ci permettono di non doverci occupare delle "code" degli array. La preparazione di `b[]` costa $\Theta(n)$. Ciò non incide sul comportamento asintotico.

Stringhe

Il tipo stringa in C è rappresentato da array di `char`, il cui ultimo carattere è il carattere nullo `'\0'` (cioè il valore intero `0`).

`char s[] = "pippo";` equivale a `char s[] = {'p','i','p','p','o',0};`

Una costante stringa in un'espressione è un puntatore al primo carattere della zona di memoria dove la stringa stessa è memorizzata.

```
char s[] = "ban";
printf("%s%s%c", s, s+1, *(s+1)); /* stampa banana */
```

Nota: `char *s = "stringa";` differisce da `char s[] = "stringa";`
Nel primo caso, `s` è una variabile puntatore a `char` inizializzata con l'indirizzo del primo byte della zona di memoria allocata alla costante `"stringa"`.
Nel secondo caso, `s` è un valore puntatore costante: `s` non può essere modificato.

Manipolare stringhe

La manipolazione di stringhe si effettua attraverso la manipolazione dei puntatori associati.

Per copiare (duplicare) una stringa `s` in una stringa `t`:
– allocare (dinamicamente o no), spazio sufficiente per la stringa `t` che deve ricevere la copia.
– copiare `s` in `t` carattere per carattere:

```
char *s = "stringa";
char t[10];
char *t1 = t, *s1 = s;

while(*t1++ = *s1++);
```

Cosa avrebbe comportato scrivere invece: `t = s;` ? E `s = t;` ?

Alcuni esempi:

```
/* copia s in t e restituisce t */
char *strcpy(char *t, const char *s)
{
    register char *r = t;

    while(*r++ = *s++);
    return t;
}
```

`const char *s` : il qualificatore `const` informa che il carattere puntato da `s` è costante e non può essere modificato.

`while(*r++ = *s++);` :
Viene scandita `s`. Si termina quando il carattere puntato è `0`.
Contestualmente il carattere viene copiato nel carattere puntato dalla scansione di `t`.
`;` :Il ciclo non ha corpo poiché tutto il lavoro è svolto nell'argomento di `while`.

```
/* restituisce la lunghezza della stringa s (non conta lo 0 finale) */
unsigned strlen(const char *s)
{
    register int i;

    for(i = 0; *s; s++, i++);
    return i;
}
```

`const char *s` : anche qui si indica che il carattere puntato da `s` è costante. Invece `s` stesso può essere modificato.

`for(i = 0; *s; s++, i++);` :
Si scandisce `s`. Ci si ferma quando il carattere puntato è `0`. Simultaneameante a `s` si incremente il contatore `i` (nota l'uso dell'operatore virgola).
`;` :Anche in questo caso non c'è bisogno di fare altro e il corpo del ciclo è vuoto.

Versione alternativa di `strlen` che usa l'aritmetica dei puntatori:

```
unsigned strlen(const char *s)
{
    register const char *t = s;

    while(*t++);
    return t-s-1;
}
```

`register const char *t = s;` :
Dichiariamo un puntatore a carattere costante `t` dello stesso tipo di `s`. Non ci sono problemi a dichiararlo `register`. Non modifichiamo mai caratteri puntati da `t`.

`while(*t++);` :
Cicliamo fino a incontrare il valore `0` di terminazione della stringa. All'uscita dal ciclo `t` punta al carattere successivo allo `0`.

`return t-s-1;` :
Utilizziamo la differenza tra puntatori per determinare quanti caratteri abbiamo scandito.

La libreria standard contiene l'implementazione di versioni di `strcpy` e di `strlen`.

Per usare le funzioni per la manipolazione di stringhe, bisogna includere il file d'intestazione `string.h`.

Altre funzioni nella libreria standard (richiedono `string.h`):

```
char *strcat(char *s1, const char *s2);
```
concatena `s2` a `s1`. Restituisce `s1`.
`s1` deve puntare a una zona di dimensione sufficiente a contenere il risultato.

```
int strcmp(const char *s1, const char *s2);
```
restituisce un intero < 0 se `s1` precede `s2`, 0 se `s1 == s2`, > 0 se `s1` segue `s2` nell'ordine lessicografico.

Array multidimensionali

Un array k-dimensionale è un array di array $(k-1)$-dimensionali.

`int matrix[10][10];` : definisce una matrice di 10 x 10 interi.

Per accedere all'elemento di posizione (j, k): `matrix[j][k]`.

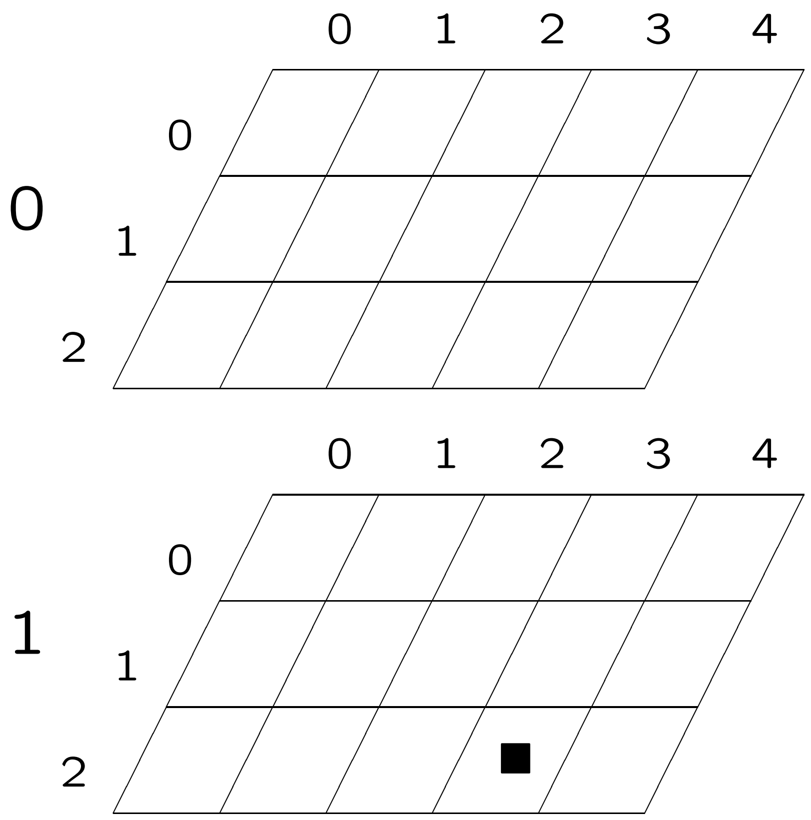
L'indirizzo dell'elemento base è `&matrix[0][0]`,
mentre `matrix == &matrix[0]`.

In genere, per l'array `int a[`k_1`][`k_2`]`$\cdots$`[`k_n`];` si ha: `a == &a[0]` e

`a[`i_1`][`i_2`]`$\cdots$`[`i_n`] ==`

`(&a[0][0]`$\cdots$`[0] + `k_n`*`k_{n-1}`*`$\cdots$`*`k_2`*`i_1` + `k_n`*`k_{n-1}`*`$\cdots$`*`k_3`*`i_2` + `$\cdots$` + `k_n`*`i_{n-1}` + `i_n`)`

Si dichiari `int a[2][3][5]`.
La mappa di memorizzazione ci dice che l'elemento `a[1][2][3]` è l'elemento in posizione $5*3*1+5*2+3=28$ sulle $5*3*2=30$ posizioni $0,\ldots,29$ che costituiscono l'array `a`.



Nella definizione

`int a[7][5];`

il nome `a` è un puntatore ad un array di interi, con valore dato da `&a[0]`, mentre l'indirizzo del primo elemento dell'array bidimensionale è `&a[0][0]`: il valore è lo stesso, ma il tipo è diverso.

```
int a[7][5];
int *p,*q;
int (*pp)[5];

printf("&a[0]:%p\t&a[0][0]:%p\n",&a[0],&a[0][0]);
p = &a[0][0];
pp = &a[0];
q = *a;
q = *(&a[0]);
printf("p:%p\tpp:%p\tq:%p\n",p,pp,q);
```

Nelle dichiarazioni di parametri formali si può omettere la lunghezza della prima dimensione dell'array multidimensionale:
la formula vista prima per l'accesso agli elementi (la *mappa di memorizzazione*) non abbisogna del valore di $\mathtt{k}_1$.

`int a[][5]` equivale a `int (*a)[5]` equivale a `int a[7][5]`

Le altre lunghezze non possono essere omesse.

Anche nell'inizializzazione esplicita la prima lunghezza può essere omessa:

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

ESEMPIO: Ordinamento di parole in un testo

```c
/* sw.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define BUFFERSIZE 512
#define BLOCKSIZE 10

char *leggiparola(FILE *fp)
{
    static char buf[BUFFERSIZE];
    char *s = buf;
    char *t;

    while((*s = fgetc(fp)) != EOF && !isalnum(*s));
    if(*s != EOF)
        while(isalnum(*++s = fgetc(fp)));
    *s = 0;
    t = malloc(strlen(buf)+1);
    return strlen(buf) ? strcpy(t,buf) : NULL;
}
```

```
char **leggitesto(FILE *fp,int *pn)
{
    char **words = malloc(BLOCKSIZE * sizeof(char *));
    char *s;

    *pn = 0;
    while((s = leggiparola(fp)) && strcmp("END",s) != 0) {
        words[(*pn)++] = s;
        if(!(*pn % BLOCKSIZE))
            words = realloc(words,(*pn/BLOCKSIZE + 1) * BLOCKSIZE * sizeof(char *));
    }
    return words;
}

void mergesort(char **a, char **b, int l, int r)
{
    int i,j,k,m;

    if(r > l) {
        m = (r+l)/2;
        mergesort(a,b,l,m);
        mergesort(a,b,m+1,r);
        for(i = m+1; i > l; i--)      b[i-1] = a[i-1];
        for(j = m; j < r; j++)        b[r+m-j] = a[j+1];
        for(k = l; k <= r; k++)       a[k] = strcmp(b[i],b[j]) < 0 ? b[i++] : b[j--];
    }
}
```

```
int main(void)
{
    int i,n;
    char **w = leggitesto(stdin,&n);
    char **b = calloc(n, sizeof(char *));

    mergesort(w,b,0,n-1);
    for(i = 0;i < n;i++)
        printf("%s\n",w[i]);
}
```

Commenti: `sw.c` legge un testo e ne estrae le parole fino ad incontrare la fine del file. Poi elenca le parole estratte in ordine lessicografico.

L'elenco delle parole viene memorizzato in un array di stringhe allocato dinamicamente.

```c
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

Per usare `malloc`,`calloc` e `realloc` includiamo `stdlib.h`.
Per usare `strcpy`,`strlen` e `strcmp` includiamo `string.h`.
Il file `ctype.h` contiene numerose macro per effettuare test su caratteri.
Qui usiamo `isalnum(c)` che è vera quando `c` è un carattere alfanumerico.

```c
#define BUFFERSIZE 512
#define BLOCKSIZE 10
```

Consideriamo parole lunghe al più `BUFFERSIZE` caratteri.
Allochiamo l'array di puntatori a `char` `BLOCKSIZE` elementi per volta.

```
char *leggiparola(FILE *fp)
```

legge una parola dallo stream associato a `fp`.

```
    static char buf[BUFFERSIZE];
    char *s = buf;
    char *t;
```

`buf` è la memoria di lavoro dove si costruisce la parola letta. `s` scandisce `buf` carattere per carattere. `t` punterà alla parola letta allocata in memoria dinamica.

```
    while((*s = fgetc(fp)) != EOF && !isalnum(*s));
    if(*s != EOF)
        while(isalnum(*++s = fgetc(fp)));
    *s = 0;
```

Si "saltano" tutti i caratteri letti fino al prossimo alfanumerico. Se non si è incontrato `EOF`, si accumulano caratteri in `buf`, grazie alla scansione `++s`, fintanto che si leggono alfanumerici.
`*s = 0` pone il carattere di terminazione della stringa letta in `buf`.

```
    t = malloc(strlen(buf)+1);
    return strlen(buf) ? strcpy(t,buf) : 0;
```

Viene allocata la memoria dinamica necessaria a contenere la stringa appena letta, e se ne assegna l'indizirizzo base a `t`.
Se `buf` è vuota si restituisce il puntatore nullo, altrimenti si restituisce `t`, dopo avervi copiato la stringa contenuta in `buf`.
N.B. per brevità non si è controllato l'esito di `malloc`: nelle applicazioni vere bisogna farlo sempre.

```
char **leggitesto(FILE *fp,int *pn)
```

`leggitesto` crea l'elenco delle parole estratte dallo stream associato a `fp`.
In `*pn`, passata per indirizzo, si memorizza il numero di parole lette.
La funzione restituisce un puntatore a puntatore a `char`: restituisce l'indirizzo base di un array di `char *` allocato dinamicamente.

```
char **words = malloc(BLOCKSIZE * sizeof(char *));
```

`words` è un puntatore a puntatore a `char`.
Gli si assegna l'indirizzo base di una zona di `BLOCKSIZE` elementi di tipo `char *`.

```
while((s = leggiparola(fp)) && strcmp("END",s) != 0) {
    words[(*pn)++] = s;
```

Si esce dal ciclo quando viene letta la parola vuota (da `leggiparola(fp)`) o quando la parola letta è `"END"`.
Si pone la parola letta nell'array `words`, nella posizione scandita da `*pn`.
NOTA gli assegnamenti:
`s = leggiparola(fp)` e `words[...] = s`.
Qui si maneggiano direttamente i puntatori, non si creano invece copie delle stringhe puntate.

```
    if(!(*pn % BLOCKSIZE))
        words = realloc(words,(*pn/BLOCKSIZE + 1) * BLOCKSIZE * sizeof(char *));
```

Se si è esaurita la capacità corrente dell'array `words`, lo si realloca con `BLOCKSIZE` elementi nuovi.
Anche in questo caso, bisognerebbe controllare che `realloc` abbia successo.

```
    return words;
```

Viene restituito l'array di parole così costruito.

```
void mergesort(char **a, char **b, int l, int r) {
```

Per ordinare l'array `words` si adatta a questo tipo (`char *`) l'algoritmo `mergesort`.

```
            a[k] = strcmp(b[i],b[j]) < 0 ? b[i++] : b[j--];
```

Per il confronto tra elementi si utilizza `strcmp`.

Nel `main` ci si deve preoccupare di creare un array di appoggio `b` di tipo e dimensioni adeguate.

```
int main(void)
{
    int i,n;
    char **w = leggitesto(stdin,&n);
    char **b = calloc(n, sizeof(char *));

    mergesort(w,b,0,n-1);
    for(i = 0;i < n;i++)
        printf("%s\n",w[i]);
}
```

Non ci si preoccupa di liberare con `free` la memoria allocata dinamicamente, poiché potremmo liberarla solo poco prima della fine del processo stesso, allorché viene rilasciata automaticamente.

Controllo dell'input orientato ai caratteri: `ctype.h`

Il file d'intestazione `ctype.h` contiene numerosi prototipi e macro relativi alla gestione dei caratteri:

```
int isalpha(int c); /* >0 se c e' una lettera, 0 altrimenti*/
int isupper(int c); /* >0 se c e' una lettera maiuscola, 0 altrimenti*/
int islower(int c); /* >0 se c e' una lettera minuscola, 0 altrimenti*/
int isdigit(int c); /* >0 se c e' una cifra, 0 altrimenti */
int isalnum(int c); /* >0 se c e' una lettera o una cifra, 0 altrimenti*/
int isspace(int c); /* >0 se c e' un carattere di spaziatura, 0 altrimenti */
int ispunct(int c); /* >0 se c e' un carattere di punteggiatura, 0 altrimenti*/
...

int tolower(int c); /* se c e' maiuscola restituisce
                       la corrispondente minuscola, altrimenti c */
int toupper(int c); /* se c e' minuscola restituisce
                       la corrispondente maiuscola, altrimenti c */
```

dove `>0` è un valore positivo ma non fissato dallo standard.