Stringhe

- In ANSI C una *stringa* è una successione di caratteri terminata dal carattere '\0' (carattere nullo, il cui valore è 0, da non confondere con il carattere '0' il cui valore è 48).
- I caratteri che formano la stringa (compreso il carattere '\0' di fine stringa) *devono* essere memorizzati in un array di `char`.

Una stringa è quindi rappresentata in memoria come una sequenza di byte (dimensione del tipo `char`) che termina con 0.

Esempio

```
char t[5] = {'l', 'u', 'n', 'a','\0'};
```

definisce l'array

t[0]	'l'
t[1]	'u'
t[2]	'n'
t[3]	'a'
t[4]	'\0'

dove `'l'` è la costante intera 108, `'u'` è la costante 117, ecc.; il valore di `t[4]` è 0.
L'array `t[]` rappresenta la stringa `"luna"`.

Definizioni equivalenti:

```
char t[5] = "luna";

char t[10] = "luna";
```

- Nella prima definizione l'array `t[]` contiene 5 elementi, che è la minima dimensione che un array deve avere per contenere la parola `"luna"` (4 caratteri + carattere fine stringa).
- Nel secondo caso, `t[]` contiene 5 elementi in più del necessario, il cui valore iniziale non è definito. Tuttavia, anche in questo caso `t[]` rappresenta la stringa `"luna"`.

Nota

Le stringhe non hanno una rappresentazione astratta, ma sono array di caratteri su cui sono ammesse *tutte* le operazioni per gli array.
È compito del programmatore assicurarsi che le convenzioni sulla rappresentazione di stringhe siano rispettate.

Stringhe costanti

Una stringa costante è scritta tra doppi apici.

- Il *tipo* di una stringa costante è `char*`.
- Il *valore* di una stringa costante è l'indirizzo al primo elemento dell'array che rappresenta la stringa stessa.

Esempio
La stringa costante `"sole"` è rappresentata dall'array

's'	'o'	'l'	'e'	'\0'

Si possono usare sulla stringa le notazioni usate per array e puntatori. Quindi, le espressioni

```
"sole"          "sole" + 1          "sole" + 4
```

equivalenti a

```
&"sole"[0]      &"sole"[1]          &"sole"[4]
```

hanno tipo `char*` e il loro valore è dato dall'indirizzo delle locazioni di memoria contenenti rispettivamente i caratteri `'s'`, `'o'` e `'\0'`.

Le espressioni

```
*"sole"          *("sole" + 1)          *("sole" + 4)
```

equivalenti a

```
"sole"[0]          "sole"[1]              "sole"[4]
```

hanno tipo `char` e valore rispettivamente `'s'` (=115), `'o'` (=111) e `'\0'` (=0).

Osservazioni

- Non confondere `1` (costante di tipo `int` e valore 1), `'1'` (costante di tipo `int` e valore 49) e `"1"` (stringa rappresentata da un array contenente i caratteri `'1'` e `'\0'`).
- Non è possibile modificare le stringhe costanti (anche se alcuni compilatori lo permettono!); è possibile invece modificare le stringhe non costanti (come avviene per gli array).

- La dichiarazione

  ```
  char t[N]= "luna";
  ```

  dove si assume `N` ≥ 5, definisce un array di `char` contenente la stringa `"luna"` (se `N` ≥ 6, il valore nelle locazioni di memoria `t[5]`, `t[6]`, ... `t[N-1]` non è definito e dipende dal sistema).

  È invece sbagliato fare

  ```
  char t[N];
  t = "luna";
  ```

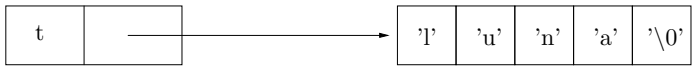
  Infatti, l'assegnamento è interpretato come:

  > Assegna a `t` il valore di `"luna"` (= indirizzo dell'array che rappresenta tale stringa).

  Non è possibile fare ciò in quanto il nome di un array è una *costante* (il cui valore è l'indirizzo al primo elemento dell'array), quindi non può essere modificato. Il codice non è compilato.
  È invece possibile fare:

  ```
  char  *t;
  t = "luna";
  ```


Il valore di `t` è ora l'indirizzo dell'array contente la stringa `"luna"`.



L'istruzione

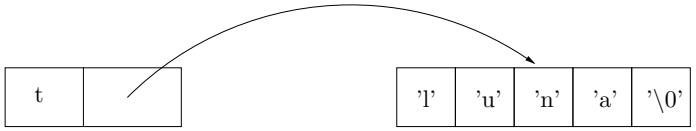
```
printf("%c", *t);
```

stampa su standard output il carattere `l`.
Poiché `t` è una variabile di tipo puntatore (e non il nome di un array!), il suo valore può essere modificato.

Dopo l'assegnamento

```
t = t + 2;
```

si ha:



Questa volta l'istruzione `printf("%c", *t)` stampa su standard output il carattere `n`.
Non è invece possibile fare

```
*t = 'c';
```

per ottenere la stringa `"luca"`, in quanto `"luna"` è una stringa costante e non può essere modificata (il programma è compilato ugualmente, ma si hanno poi problemi in esecuzione).

Funzioni su stringhe

Lettura di una stringa

Scrivere una funzione `read()` che legge una stringa da standard input in un array di `char` passato come parametro, saltando gli eventuali caratteri di spaziatura iniziali (spazi, "a capo", ecc.). La funzione deve restituire 1 se la lettura è terminata con successo, `EOF` se è stato letto il carattere "end-of-file".
Usare esclusivamente `getchar()` per leggere i caratteri presenti su standard input e `isspace()` per riconoscere i caratteri di spaziatura: `isspace()` restituisce un valore diverso da 0 (il cui valore non è stabilito dall'ANSI C) se il carattere passato come argomento è un carattere di spaziatura, 0 altrimenti.

Assumere che l'array passato come parametro contenga sufficiente spazio per contenere la stringa letta.

Nota

Quando una funzione deve memorizzare una stringa in un array `a[]` passato come parametro, è consuetudine assumere che `a[]` contenga spazio sufficiente per memorizzare la stringa (questa assunzione viene fatta anche nelle funzioni su stringhe delle librerie ANSI C).
È compito del chiamante assicurarsi che l'assunzione sia rispettata.
In generale questo tipo di assunzione va fatta quando una funzione memorizza dei dati in un array passato come parametro.
Ad esempio, in

```
void copy(int  to[], int from[], int n)
```

(copia `n` elementi dall'array `from[]` all'array `to[]`) è necessario assumere che `to[]` abbia almeno `n` elementi.

Codice di `read()`

L'intestazione della funzione può essere scritta in una delle due forme equivalenti:

```
int read(char word[])          int read(char *word)
```

La funzione deve compiere le seguenti operazioni:

1. *Salta gli spazi iniziali*
   Legge un carattere alla volta da standard input e termina quando il carattere letto è `EOF` oppure è diverso da un carattere di spaziatura. Se l'ultimo carattere letto è `EOF`, la funzione termina restituendo `EOF`.

```
 while( (c=getchar())!= EOF  &&  isspace(c) );
/*   all'uscita dal ciclo, c vale EOF  oppure
     c e' il primo carattere della stringa  */
 if(c == EOF)
    return EOF;
```

Quando viene valutata l'espressione di assegnamento

```
 c = getchar()
```

viene chiamata `getchar()`, il valore restituito è assegnato alla variabile c e il valore dell'intera espressione è dato dal valore di c. Quindi, la condizione in `while` è verificata se e solo se il carattere letto da `getchar()` è diverso da `EOF` ed è un carattere di spaziatura.

2. *Lettura della parola*
   Vengono letti da standard input un carattere per volta, terminando quando si incontra "end-of-file" oppure un carattere di spaziatura. I caratteri letti i sono inseriti in `word[]`.

```
/* c contiene il primo carattere della stringa in input */
  k = 0;
  while(c!=EOF && !isspace(c)){
      word[k++] = c;
    /* equivale a: word[k] = c;  k++;  */
      c = getchar();
    }
/* all'uscita dal ciclo, c vale EOF oppure c e' un carattere
   di spaziatura */
```


3. *Inserimento marcatore di fine stringa*
   All'uscita del ciclo `while`, sono stati letti tutti i caratteri della parola e l'ultimo carattere della parola è in `w[k-1]`.
   Occorre inserire il carattere `'\0'` di fine stringa in `word[k]`

```
  word[k] = '\0';
  return 1;
```

Si noti che l'ultimo assegnamento è equivalente a

```
 word[k] = 0;
```

ma è preferibile usare la prima forma per rendere il codice più leggibile.

Esempio
Supponiamo di eseguire le linee di codice

```
char w[10];
read(w);
read(w);
```

e che lo standard input contenga

```
    elefante                    cane
```

Dopo la prima chiamata a `read()`, l'array `w[]` contiene:

w[0]	'e'
w[1]	'l'
w[2]	'e'
w[3]	'f'
w[4]	'a'
w[5]	'n'
w[6]	't'
w[7]	'e'
w[8]	'\0'
w[9]	?

L'ultimo carattere della parola è in `w[7]` e `w[8]` contiene il carattere di fine stringa.

Dopo la seconda chiamata a `read()`, l'array `w[]` contiene:

w[0]	'c'
w[1]	'a'
w[2]	'n'
w[3]	'e'
w[4]	'\0'
w[5]	'n'
w[6]	't'
w[7]	'e'
w[8]	'\0'
w[9]	?

Si noti che la parola in `w[]` termina nella locazione di indirizzo `w+3` in quanto `w[4]` contiene il carattere `'\0'`. L'array `w[]` rappresenta la stringa `"cane"`.

Se si esegue l'istruzione

```
w[0] = '\0';
```

l'array `w[]` rappresenta invece la stringa nulla, ossia la stringa non contenente alcun carattere.

La stringa nulla è denotata da `""`.

Stampa di una stringa

Scrivere una funzione `print()` che stampa su standard output la stringa passata come parametro (cioè, la stringa rappresentata dall'array passato come parametro) e restituisce il numero di caratteri stampati. Usare solo `putchar()`.

Codice di `print()`

A differenza delle funzioni sugli array viste prima, non è necessario passare come parametro la lunghezza della stringa. Infatti, per le convenzioni sulla rappresentazione di stringhe, la parola termina quando si incontra il carattere `'\0'`.

```
int print(char word[]){
  int k;
  for(k=0 ; word[k]!='\0' ; k++)
    putchar(word[k]); //   putchar(*(word+k));
  return k;
}
```

In forma più compatta:

```
int print(char word[]){
  int k;
  for(k=0 ; word[k]!='\0' ; putchar(word[k++]) );
  return k;
}
```

Si ricordi che l'ultima espressione in un ciclo `for` viene sempre valutata al termine di ogni iterazione. La valutazione di

```
putchar(word[k++])
```

avviene eseguendo la chiamata `putchar()` con argomento `word[k]`, successivamente `k` viene incrementato di uno.

Esempio

Le linee di codice

```
char w[10];
print("Esempio\n");
read(w);
print(w);
print("\n");
read(w);
print(w);
w[0] = '\0'; /*  #  */
print(w);
print(w+2);
print(w+6);
```

avendo su standard input

```
    elefante                    cane
```

producono come output

```
Esempio
elefante
cane
ne
te
```


Per capire l'effetto delle ultime operazioni di stampa, occorre tenere presente che, dopo l'istruzione #, l'array `w[]` contiene

w[0]	'\0'
w[1]	'a'
w[2]	'n'
w[3]	'e'
w[4]	'\0'
w[5]	'n'
w[6]	't'
w[7]	'e'
w[8]	'\0'
w[9]	?

Pertanto, l'istruzione

```
print(w);
```

non produce alcun effetto in quanto `w[]` rappresenta la stringa nulla.

L'istruzione

```
print(w+2);
```

richiede la stampa della stringa contenuta nel sottoarray che inizia in `w+2`. Tale stringa è data dalla sequenza di caratteri da `w[2]` fino al prossimo carattere `'\0'`, che è in `w[4]`.

Viene quindi stampato

```
ne
```

Analogamente, l'ultima operazione stampa la stringa contenuta nell'array `w+6` che è

```
te
```

Osservazioni

1. Se prima dell'ultima operazione di stampa si esegue l'istruzione

   ```
   w[8] = 'a';
   ```

   cosa succede?

In questo caso l'array `w+6` non rappresenta più una stringa e l'esecuzione di `read()` ha un comportamento impredicibile. Verrano stampati i valori nelle locazioni indirizzo `w+6`, `w+7`,... fintanto che tali valori sono diversi da 0.
Un possibile output è:

```
tea?????B Y@?????VB
```

Il compilatore non può rilevare errori di questo tipo perché in C una stringa è un array, su cui sono ammesse tutte le operazioni per gli array.

2. Cosa succede se si esegue

   ```
   char w[10];
   read(w);
   ```

   e la parola letta da standard input è più lunga di 9 caratteri (l'array `w[]` non contiene sufficiente spazio per contenerla)?

I primi 10 caratteri della parola vengono memorizzati nelle locazioni di indirizzo `w`, `w+1`, ..., `w+9` che corrispondono a locazioni dell'array `w[]`. I caratteri successivi, vengono posti nelle locazioni aventi indirizzo `w+10`, `w+11`, ... e il comportamento non è prevedibile.

3. Se si esegue

   ```
   char w[10];
   read(w);
   ```

   avendo su standard input i caratteri

   ```
   a0B'\0'c
   ```

   cosa contiene l'array `w[]`?

La stringa su standard input, saltando gli spazi iniziali, è composta da 8 caratteri.
Si noti che il secondo e il sesto carattere corrispondono al carattere `'0'` (valore 48, da non confondere con l'intero 0), il quarto e settimo carattere corrispondono al carattere "apice" (valore 39).

I valori in `w[]` sono

w[0]	'a'
w[1]	'0'
w[2]	'B'
w[3]	' '
w[4]	'\'
w[5]	'0'
w[6]	' '
w[7]	'c'
w[8]	'\0'
w[9]	?

Se si esegue ora

`print(w);`

viene stampato

`a0B'\0'c`

4. Cosa succede se per rappresentare una stringa si usa una array di `int` anziché di `char`?

Non è possibile dichiarare un array in questo modo

```
int w[10] = "abc";
```

in quanto l'inizializzazione con stringhe è corretta solo per array di `char`.
In questo caso il codice non è compilato.
È invece possibile dichiarare un array di `int` in questo modo

```
int w[10] = {'a', 'b', 'c', '\0'};
```

dove `w[0]` vale `'a'`, `w[1]` vale `'b'`, ecc..

Tuttavia, l'array non rappresenta la stringa `"abc"`.

Infatti, assumendo che il tipo `int` occupi 4 byte, in memoria si ha la sequenza di byte

```
          ........
w[0] -->  01100001    /* = 97 = 'a' */
          00000000
          00000000
          00000000
w[1] -->  01100010    /* = 98 = 'b' */
          00000000
          00000000
          00000000
w[2] -->  01100011    /* = 99 = 'c' */
          00000000
          00000000
          00000000
w[3] -->  00000000
          ........
```


Per verificarlo, eseguire le seguenti linee di codice che stampano il contenuto dell'array `w[]` byte per byte:

```
char *p;
...
p = (char*) w; // p contiene  &w[0];
int dim = sizeof(int) * 10; // numero di  byte in w[]
for(k=0; k<dim; k++)
  printf("%o\n", p[k]);
     // stampa in ottale il valore  *(p+k)
```

Si ricordi che `p+k`, secondo l'aritmetica dei puntatori, corrisponde all'indirizzo "`&w[0]` + k byte".
Poiché in C una stringa è una sequenza di byte terminante con 0, la stringa rappresentata da `w[]` è `"a"`.
Infatti, eseguendo

```
print(w);
```

viene stampato

```
a
```


Esercizio
Dire cosa succede eseguendo ora le istruzioni

```
w[0] = w[0] + 'z' * pow(2,8);
print(w);
```

e verificare la risposta scrivendo un opportuno programma. Si tenga presente che `pow(2,8)` calcola 2^8 e che moltiplicare un numero in notazione binari per 2^k equivale ad aggiungere al numero k zeri (analogamente a quanto avviene moltiplicando un numero in notazione decimale per 10^k).

Lettura e stampa con printf() e scanf()

Le operazioni di lettura e scrittura descritte prima possono essere effetuate usando `scanf()` e `printf()`.

Supponiamo che `w[]` sia un array di `char`.

L'istruzione

```
scanf("%s",w);
```

legge una stringa presente da standard input nell'array `w`, saltando gli eventuali caratteri di spaziatura che la precedono.

Restituisce:

- 1 se l'operazione di lettura ha successo,
- `EOF` se, dopo i caratteri di spaziatura, viene letto il carattere di "end-of-file".

Si *assume* che `w[]` contenga spazio sufficiente per contenere la parola letta (lunghezza della parola più uno per il carattere di fine stringa).

È analoga a:

```
read(w);
```


L'istruzione

```
printf("%s",w);
```

stampa su standard output la stringa rappresentata da `w[]` e restituisce il numero di caratteri stampati.

Le istruzioni

```
print(w);          print(w+2);          print(w+6);
```

sono equivalenti a

```
printf("%s",w);    printf("%s",w+2);     printf("%s",w+6);
```


Osservazioni

1. Il primo argomento di `printf()` è una stringa; `printf()` stampa la stringa passata come primo argomento sostituendo gli eventuali caratteri di conversione presenti in essa con il valore dei corrispondenti argomenti, secondo il formato richiesto.

Ad esempio,

```
int a = 48;
 printf("a vale %d, corrispondente al carattere %c.", a , a);
```

stampa

```
a vale 48, corrispondente al carattere 0.
```


2. Il preprocessore non effettua sostituzioni all'interno di una stringa.

Ad esempio:

```
#define ESEMPIO  sole
....
  printf("ESEMPIO \n");
```

stampa su standard output

```
ESEMPIO
```

Se si vuole stampare `sole` occorre fare

```
#define ESEMPIO "sole"
...
 printf("%s \n", ESEMPIO);
 /* il preprocessore sostituisce ESEMPIO con "sole" */
```

3. La stringa

```
""
```

in cui non ci sono caratteri denota la stringa nulla (rappresentata da un array contenente solamente il carattere '\0').

È diversa dalla stringa

```
" "
```

che è una stringa composta dal carattere "spazio" (ed è quindi rappresentata da un array di due caratteri).

Esempio di utilizzo della stringa nulla:

```
int n;
char *p;
p  =  n > 1   ?  "s" :  "";
printf("%d element%s.", n, p);
```

Se `n` vale 5, a `p` viene assegnata la stringa `"s"` e viene stampato

```
5 elements.
```

Se `n` vale 1, a `p` viene assegnata la stringa nulla e viene stampato

```
1 element.
```

Se a `p` viene assegnata la stringa `" "` anziché la stringa nulla, viene stampato

```
1 element .
```


Copia di stringhe

Definire una funzione `strCopy()` che copia una stringa `s` (source) passata come secondo parametro in un array di `char` `t` (target) passato come primo parametro. Si *assume* che `t` contenga sufficiente spazio per contenere la stringa (ossia, `t` è l'indirizzo di un array di `char` la cui dimensione è maggiore del numero dei caratteri della stringa in `s` più uno).

Il problema è analogo a quello già visto della copia di vettori. L'unica differenza è che in questo caso non è necessario passare come parametro il numero degli elementi da copiare in quanto la funzione è in grado di riconoscere dove termina la stringa in `s`.


```
void strCopy1 (char t[], char s[]){
/* oppure: void strCopy1 (char *t, char *s) */
    int i=0;
    while ( (t[i] = s[i]) != '\0')
            i++;
}
```

dove `t[i]` può essere sostituito con `*(t+i)`, `s[i]` può essere sostituito con `*(s+i)`.

Per valutare

```
(t[i] = s[i]) != '\0'
```

vengono confrontati i valori delle espressioni a sinistra e a destra di `!=`. Per l'espressione a sinistra, viene prima eseguito l'assegnamento `t[i]=s[i]`; il valore dell'espressione di assegnamento è il valore assegnato a `t[i]`. Si noti che l'assegnamento viene effettuato anche quando `s[i]` contiene `'\0'`, dopo di che l'espressione associata a `while` non è verificata e il ciclo termina.

Altra versione:

```
void strCcopy2 (char *t, char *s){
    while ((*t = *s) != '\0'){
            s++;
            t++;
        }
}
```

In questo caso i puntatori `s` e `t` a ogni iterazione vengono incrementati di uno (nel senso dell'aritmetica dei puntatori) e quindi puntano ai successivi elementi degli array passati come parametri.

La funzione `strCopy2()` può essere riscritta come

```
void strCopy2 (char *t, char *s){
    while ( (*t++ = *s++) != '\0');
}
```

oppure (ma questa forma è meno leggibile)

```
void strCopy2 (char *t, char *s){
    while ( *t++ = *s++ );
}
```

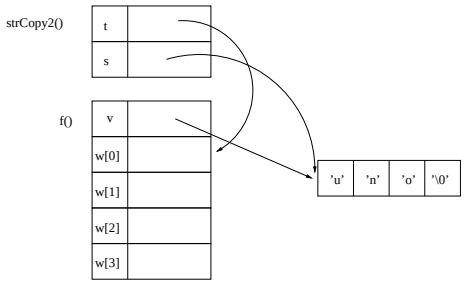
in quanto l'uscita dal `while` si ha quando l'espressione fra parentesi diventa uguale a 0 (ossia, quando a `*t` è assegnato `'\0'`).

Esempio 1

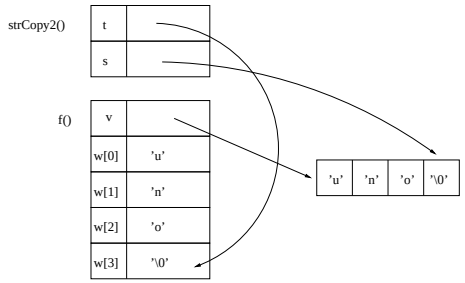
```
void f(){
 char *v = "uno";
 char w[4];
 strCopy2(w,v);
}
```


Supponiamo di eseguire `f()`.

Quando `strCopy2()` viene eseguita, prima del ciclo `while` la configurazione della memoria è



Dopo il ciclo si ha:



Esempio 2

```
char w1[30] = "cavallo";
char w2[50];
strCopy(w2,w1);
printf("%s %s", w1, w2);
strCopy(w2+1 , w1+3);
printf("%s %s", w1, w2);
strCopy(w1 , w1+2);
printf("%s %s", w1, w2);
```

dove `strCopy()` è una delle funzioni per la copia di stringhe viste prima. Eseguendo il codice viene stampato:

```
cavallo cavallo
cavallo callo
vallo callo
```


Errori da evitare

Supponiamo di aver definito

```
char *v = "luna";
char *w;
```

e di voler copiare la stringa `"luna"` in `w`. Un tipico errore è quello di fare

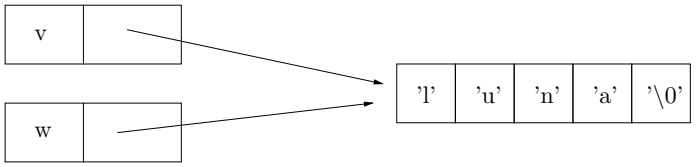
```
strCopy(w,v); /* una delle due versioni di prima */
```

pensando che la funzione `strCopy()` si preoccupi di "creare" una nuova stringa e di assegnarla a `w`. In realtà la funzione si limita a copiare il carattere `'l'` in `w[0]`, il carattere `'u'` in `w[1]`, ecc. Poiché il valore di `w` non è definito, il comportamento del codice non è prevedibile. Per funzionare correttamente, occorre assegnare a `w` (che è una variabile di tipo puntatore) l'indirizzo di un array di dimensione opportuna.

È anche sbagliato fare

```
w = v;
```

Infatti, dopo questa istruzione `w` e `v` contengono l'indirizzo alla *stessa* stringa, quindi in `w` non si ha una copia della stringa `"luna"`, ma si ha un'unica stringa condivisa da due variabili.



Questo tipo di errori non possono essere segnalati dal compilatore (che si limita solo a controlli di tipo sintattico).

Se occorre verificare se due stringhe `w1` e `w2` sono uguali è sbagliato fare

```
if(w1 == w2)
  ...
```

in quanto in questo modo vengono confrontati gli *indirizzi* delle due stringhe, non i carattari che le compongono.
Occorre invece definire una funzione `strUguale()` che confronta le due stringhe carattere per carattere e restituisce 1 se sono uguali, 0 altrimenti e fare

```
if(strUguale(w1, w2) == 1)
  ...
```

o più semplicemente

```
if(strUguale(w1, w2))
  ...
```

%

Analogamente, per controllare se `w` è uguale alla stringa nulla, è sbagliato fare

```
if(w == "")
   ...
```

Infatti, in questo modo vengono confrontati i *valori* di `w` e di `""` (ossia, gli *indirizzi* dei vettori che contengono la stringa `w` e la stringa nulla) non il contenuto delle stringhe. Si può fare ad esempio:

```
if(strUguale(w, ""))
  ...
```


La libreria string.h

Per gestire le stringhe si possono utilizzare le funzioni di libreria i cui prototipi sono definiti in `<string.h>`.
Ad esempio, la funzione `strCopy()` prima definita è analoga alla funzione `strcpy()`. Sono definite funzioni per determinare la lunghezza di una stringa, concatenare stringhe, confrontarle, ecc.
Per usarle occorre leggere con attenzione la documentazione riportata sui manuali.