Introduzione al C

Lezione 4

Allocazione dinamica della memoria

Rossano Venturini

rossano@di.unipi.it

Pagina web del corso

http://didawiki.cli.di.unipi.it/doku.php/informatica/all-b/start

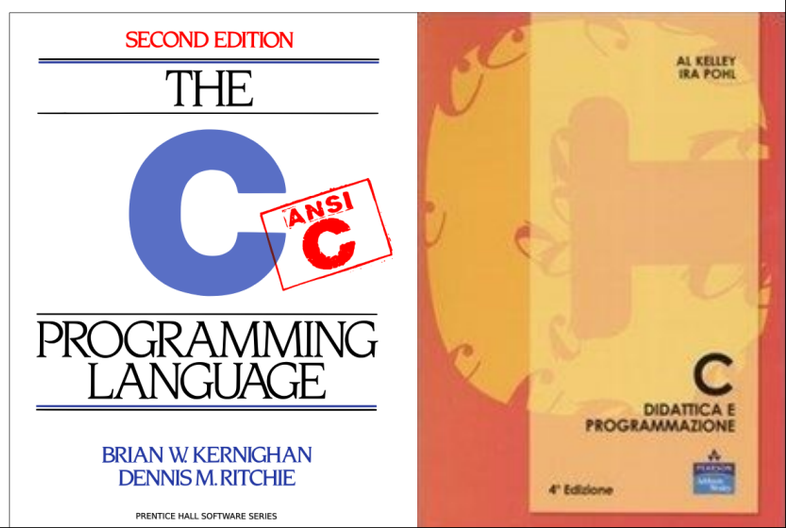
Lezioni di ripasso C

Giovedì 27 16-18 Aula A-B

Le successive lezioni di laboratorio saranno

Corso B Giovedì 14-16 Aula H-M

Corso A Giovedì 16-18 Aula H-M



Esercizio 1

FindVal

Esercizio

Scrivere una funzione

```
int* FindVal(int a[], int len, int val)
```

che, dato un array *a* e la sua lunghezza *len*, cerchi il valore *val* all'interno di *a* e restituisca un puntatore alla cella che lo contiene, o la costante predefinita `NULL` se *val* non è contenuto in *a*.

Scrivere poi un programma che legga da input un array di 10 interi e un intero val e stampi `trovato` se l'intero val si trova nell'array, `non trovato` altrimenti.

L'input è formato da dieci righe contenenti gli elementi dell'array, seguite dall'intero val da cercare.
L'unica riga dell'output contiene la stringa

```
trovato
```

se l'intero val si trova nell'array,

```
non trovato
```

altrimenti.

Esercizio 1

```
int* FindVal(int a[], int len, int val) {

  int i = 0, trovato = 0;
  int *p = NULL;
  while ((i < len) && (!trovato)) {
    if (a[i] == val) {
      trovato = 1;
      p = &a[i];
    }
    i++;
  }
  return p;
}
```

Esercizio 2

Contatore

Esercizio

Scrivere una programma che data una sequenza di interi tenga traccia delle frequenze degli interi compresi tra 0 e 9 (estremi inclusi). La sequenza termina quando viene letto il valore -1. Il programma deve stampare in output le frequenze dei valori compresi tra 0 e 9.
Le frequenze saranno mantenute in un array di contatori di lunghezza 10 che sarà inizializzato a 0.

Implementare queste due funzioni:

- `void reset(int array[], int len)`: inizializza l'array dei contatori a 0;
- `void add(int array[], int len, int val)`: incrementa il contatore `array[val]` se `val` è tra 0 e `len`-1.

L'input è formato da una sequenza di interi terminata dall'intero -1.
L'output è costituito dalle frequenze (una per riga) degli interi tra 0 e 10 nella sequenza letta in input.

Esercizio 2

```
void reset(int array[], int len) {
  int i;
  for (i = 0; i < len; i++) {
    array[i] = 0;
  }
}
```

```
void add(int array[], int len, int val) {
  if ( (val < 0) || (val >= len) ) return;
  array[val] += 1;
}
```

Esercizio 4 Anagramma

Esercizio

Scrivere la funzione

`int anagramma(unsigned char *s1, unsigned char *s2)`

che restituisca 1 se le stringhe puntate da $s1$ e $s2$ sono una l'anagramma dell'altro e 0 altrimenti.
Esempio: `anagramma(''pizza'', ''pazzi'') == 1`

Strivere quindi un programma che legga da input due stringhe $s1$ e $s2$ e utilizzi questa funzione per stabilire se una è l'anagramma dell'altra. Nota: utilizzare il tipo `unsigned char *` per le stringhe.

Hint. Data una stringa S, costruire un array $aS[256]$ tale che $aS[i]$ memorizzi il numero di occorrenze del carattere i in S. Come sono gli array aS e aZ di due stringhe S e Z che sono una l'anagramma dell'altra?

L'input è formato da due stringhe $s1$ e $s2$.
L'output è 1 se $s1$ è l'anagramma di $s2$, 0 altrimenti.

Esempi

Input	Output
`aeiou` `uoaei`	`1`

Esercizio 4

```
int anagramma(unsigned char *x, unsigned char *y) {
  int i;
  int xc[256], yc[256];
  for (i = 0; i < 256; i++) { xc[i] = yc[i] = 0; }
  int lenx = strlen(x);
  int leny = strlen(y);
  if(lenx != leny) return 0;

  for (i = 0; i < lenx ; i++) {
    xc[x[i]] += 1;
    yc[y[i]] += 1;
  }

  for (i = 0; i < 256; i++) {
    if (xc[i] != yc[i]) return 0;
  }
  return 1;
}
```

Allocazione dinamica della memoria

Allocazione dinamica della memoria

In C la memoria può essere gestita in modo dinamico, attraverso l'allocazione e deallocazione di blocchi di memoria.

Allocazione dinamica della memoria

In C la memoria può essere gestita in modo dinamico, attraverso l'allocazione e deallocazione di blocchi di memoria.

A cosa serve?

Allocazione dinamica della memoria

In C la memoria può essere gestita in modo dinamico, attraverso l'allocazione e deallocazione di blocchi di memoria.

A cosa serve?

- Ad allocare array la cui dimensione non è nota a tempo di compilazione ma decisa tempo di esecuzione;

Allocazione dinamica della memoria

In C la memoria può essere gestita in modo dinamico, attraverso l'allocazione e deallocazione di blocchi di memoria.

A cosa serve?

- Ad allocare array la cui dimensione non è nota a tempo di compilazione ma decisa tempo di esecuzione;
- Per gestire strutture dati che crescono e decrescono durante l'esecuzione del programma (ad esempio liste o alberi);

Allocazione dinamica della memoria

In C la memoria può essere gestita in modo dinamico, attraverso l'allocazione e deallocazione di blocchi di memoria.

A cosa serve?

- Ad allocare array la cui dimensione non è nota a tempo di compilazione ma decisa tempo di esecuzione;

- Per gestire strutture dati che crescono e decrescono durante l'esecuzione del programma (ad esempio liste o alberi);

- Per avere maggiore flessibilità sulla durata della memoria allocata. Altrimenti la memoria verrebbe deallocata automaticamente all'uscita del blocco (es. funzione) nella quale è stata allocata.

Due esempi

Esempio

```
int main() {
    int n;
    scanf("%d", %n);
    int a[n]; // NO! Questo non si può fare in ANSI C
    ...
```

Due esempi

Esempio

```
int main() {
    int n;
    scanf("%d", %n);
    int a[n]; // NO! Questo non si può fare in ANSI C
    ...
```

Esempio

```
int *crea_array(int n) {
    int a[n]; // NO! Questo non si può fare in ANSI C
    int i;
    for( i = 0; i < n; i++) a[i] = 0;

    return a; // NO! Lo spazio allocato per a viene deallocato all'uscita
}
```

Allocazione dinamica della memoria

Allocazione dinamica della memoria

- I blocchi sono allocati tipicamente in una parte della memoria chiamata *heap*;

Allocazione dinamica della memoria

- I blocchi sono allocati tipicamente in una parte della memoria chiamata *heap*;
- La memoria allocata è accessibile attraverso l'uso di *puntatori;*

Allocazione dinamica della memoria

- I blocchi sono allocati tipicamente in una parte della memoria chiamata *heap*;
- La memoria allocata è accessibile attraverso l'uso di *puntatori;*
- Lo spazio allocato dinamicamente NON viene deallocato all'uscita delle funzioni;

Allocazione dinamica della memoria

- I blocchi sono allocati tipicamente in una parte della memoria chiamata *heap*;
- La memoria allocata è accessibile attraverso l'uso di *puntatori;*
- Lo spazio allocato dinamicamente NON viene deallocato all'uscita delle funzioni;
- La memoria che non serve più va deallocata in modo da renderla nuovamente disponibile.

Allocazione dinamica della memoria

- I blocchi sono allocati tipicamente in una parte della memoria chiamata *heap*;
- La memoria allocata è accessibile attraverso l'uso di *puntatori;*
- Lo spazio allocato dinamicamente NON viene deallocato all'uscita delle funzioni;
- La memoria che non serve più va deallocata in modo da renderla nuovamente disponibile.

Come?

Si utilizzano due funzioni della libreria standard (stdlib.h) per allocare (funzione malloc) e deallocare (funzione free) quando necessario.

Allocazione della memoria: malloc

Definita nella libreria stdlib.h che deve quindi essere inclusa.

Allocazione della memoria: malloc

Definita nella libreria stdlib.h che deve quindi essere inclusa.

```
void * malloc(size_t size_in_byte)
```

Allocazione della memoria: malloc

Definita nella libreria stdlib.h che deve quindi essere inclusa.

void * malloc(size_t size_in_byte)

dove

- size_in_byte specifica la dimensione in byte del blocco di memoria che vogliamo allocare. size_t è un tipo definito in stdlib.h, generalmente un unsigned int.

Allocazione della memoria: malloc

Definita nella libreria stdlib.h che deve quindi essere inclusa.

void * malloc(size_t size_in_byte)

dove

- size_in_byte specifica la dimensione in byte del blocco di memoria che vogliamo allocare. size_t è un tipo definito in stdlib.h, generalmente un unsigned int.
- la chiamata restituisce un void * (da convertire al tipo desiderato), puntatore alla prima cella della memoria appena allocata. se non è possibile allocare memoria, la chiamata restituisce NULL.

Allocazione della memoria: malloc

Definita nella libreria stdlib.h che deve quindi essere inclusa.

void * malloc(size_t size_in_byte)

dove

- size_in_byte specifica la dimensione in byte del blocco di memoria che vogliamo allocare. size_t è un tipo definito in stdlib.h, generalmente un unsigned int.
- la chiamata restituisce un void * (da convertire al tipo desiderato), puntatore alla prima cella della memoria appena allocata. se non è possibile allocare memoria, la chiamata restituisce NULL.

Esempio

```
int *p = (int *) malloc( 5 * sizeof(int));

char *s = (char *) malloc( 5 * sizeof(char));

float *f = (float *) malloc( 5 * sizeof(float));
```

Allocazione della memoria: malloc

Definita nella libreria stdlib.h che deve quindi essere inclusa.

void * malloc(size_t size_in_byte)

dove

- size_in_byte specifica la dimensione in byte del blocco di memoria che vogliamo allocare. size_t è un tipo definito in stdlib.h, generalmente un unsigned int.
- la chiamata restituisce un void * (da convertire al tipo desiderato), puntatore alla prima cella della memoria appena allocata. se non è [illegible] la chiamata restituisce NULL.

Conversione dal tipo void * al tipo int *

Esempio

```
int *p = (int *) malloc( 5 * sizeof(int));

char *s = (char *) malloc( 5 * sizeof(char));

float *f = (float *) malloc( 5 * sizeof(float));
```

Allocazione della memoria: malloc

Definita nella libreria stdlib.h che deve quindi essere inclusa.

void * malloc(size_t size_in_byte)

dove

- size_in_byte specifica la dimensione in byte del blocco di memoria che vogliamo allocare. size_t è un tipo definito in stdlib.h, generalmente un unsigned int.
- la chiamata restituisce un void * (da convertire al tipo desiderato), puntatore alla prima cella della memoria appena allocata. se non è [illegible] L.

Conversione dal tipo al tipo int *

sizeof(int) restituisce il numero di byte occupati da un int

Esempio

```
int *p = (int *) malloc( 5 * sizeof(int));

char *s = (char *) malloc( 5 * sizeof(char));

float *f = (float *) malloc( 5 * sizeof(float));
```

Allocazione della memoria: malloc

Definita nella libreria stdlib.h che deve quindi essere inclusa.

void * malloc(size_t size_in_byte)

dove

- size_in_byte specifica la dimensione in byte del b
  che vogliamo allocare. size_t è un tipo definito in
  generalmente un unsigned int.
- la chiamata restituisce un void * (da convertire al
  puntatore alla prima cella della memoria appena

Conversione dal tipo
al tipo int *

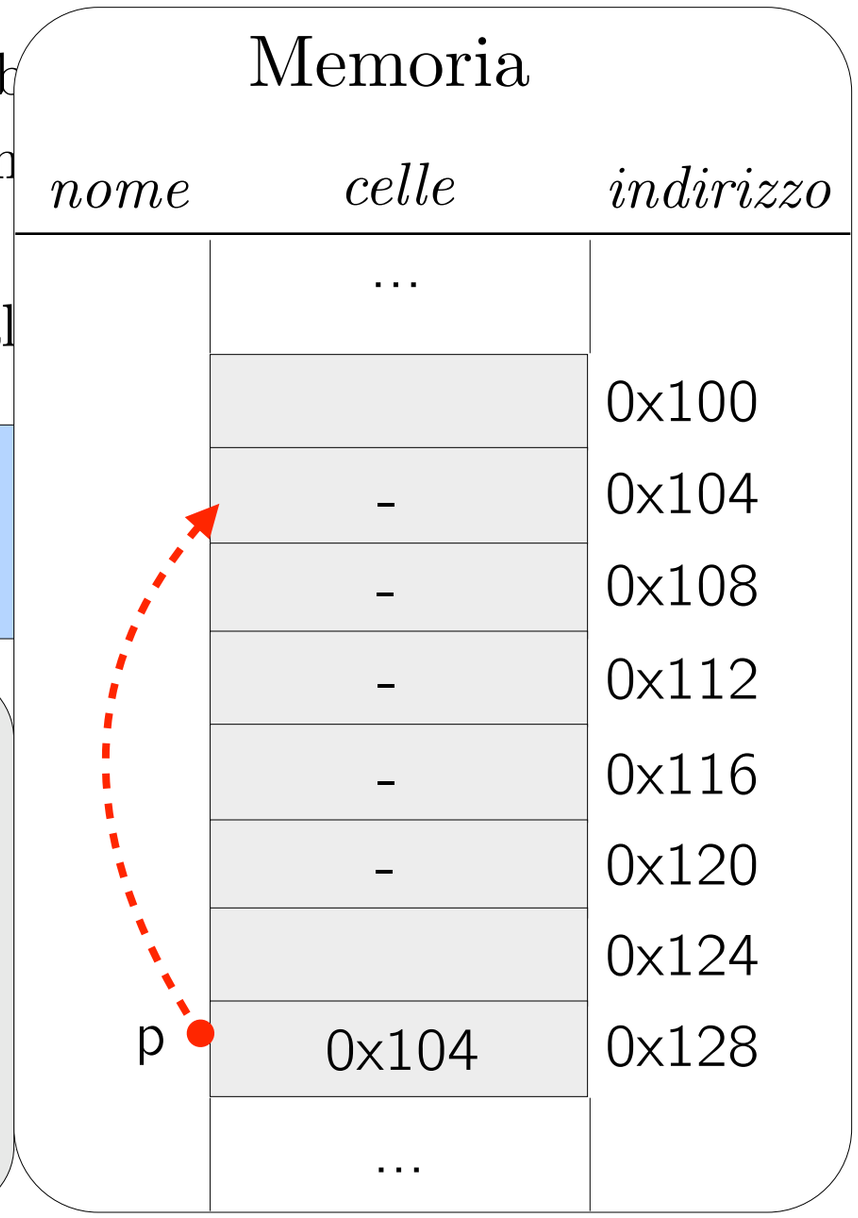
sizeof(int) restituisce il numero
byte occupati da un int

Esempio

```
int *p = (int *) malloc( 5 * sizeof(int));

char *s = (char *) malloc( 5 * sizeof(char));

float *f = (float *) malloc( 5 * sizeof(float));
```



Allocazione della memoria: malloc

Esempio

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

int main () {
    int i, n, *p;
    scanf("%d", &n);

    p = (int *) malloc(n * sizeof(int));

if(p == NULL) {  // controllo il buon esito della allocazione
    printf("Allocazione fallita\n");
    return 1;
}

for( i = 0; i < n; i++) {
    scanf("%d", p+i);
    }
...
```

Allocazione della memoria: malloc

Esempio

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

int main () {
    int i, n, *p;
    scanf("%d", &n);

    p = (int *) malloc(n * sizeof(int));

if(p == NULL) {  // controllo il buon esito della allocazione
    printf("Allocazione fallita\n");
    return
}

for( i = 0; i < n; i++) {
    scanf("%d", p+i);
    }
...
```

Attenzione: non si può accedere fuori dallo spazio allocato, ad esempio p[n].

Allocazione della memoria: malloc

Esempio

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

int main () {
    int i, n, *p;
    scanf("%d", &n);

    p = (int *) malloc(n * sizeof(int));

    if(p == NULL) {  // controllo il buon esito della allocazione
        printf("Allocazione fallita\n");
        return
}

for( i = 0; i < n; i++) {
    scanf("%d", p+i);
    }
...
```

Attenzione: non si può accedere fuori dallo spazio allocato, ad esempio p[n].

Esistono altre due funzioni, calloc e realloc, per allocare memoria. man calloc; man realloc per info

Deallocazione della memoria: free

Quando un blocco di memoria non serve più è importante deallocarlo e renderlo nuovamente disponibile utilizzando la funzione

Deallocazione della memoria: free

Quando un blocco di memoria non serve più è importante deallocarlo e renderlo nuovamente disponibile utilizzando la funzione

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

dove p è l'indirizzo di memoria restituito dalla malloc.

Deallocazione della memoria: free

Quando un blocco di memoria non serve più è importante deallocarlo e renderlo nuovamente disponibile utilizzando la funzione

void free(void * p)

dove p è l'indirizzo di memoria restituito dalla malloc.

Esempio

```
int *p = (int *) malloc( 5 * sizeof(int));
free(p);

char *s = (char *) malloc( 5 * sizeof(char));
free(s);

float *f = (float *) malloc( 5 * sizeof(float));
free(f);
```

Deallocazione della memoria: free

Quando un blocco di memoria non serve più è importante deallocarlo e renderlo nuovamente disponibile utilizzando la funzione

void free(void * p)

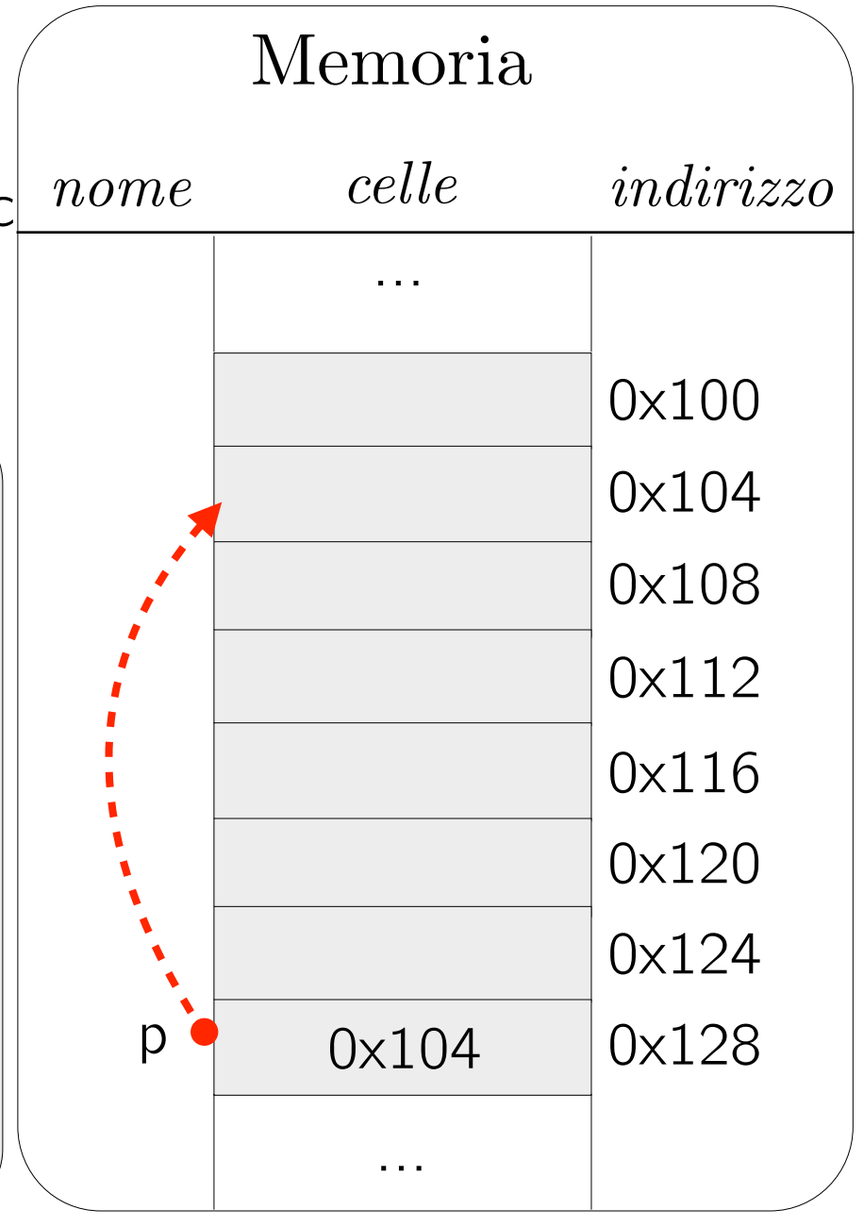
dove p è l'indirizzo di memoria restituito dalla malloc

Esempio

```
int *p = (int *) malloc( 5 * sizeof(int));
free(p);

char *s = (char *) malloc( 5 * sizeof(char));
free(s);

float *f = (float *) malloc( 5 * sizeof(float));
free(f);
```



Deallocazione della memoria: free

Quando un blocco di memoria non serve più è importante deallocarlo e renderlo nuovamente disponibile utilizzando la funzione

void free(void * p)

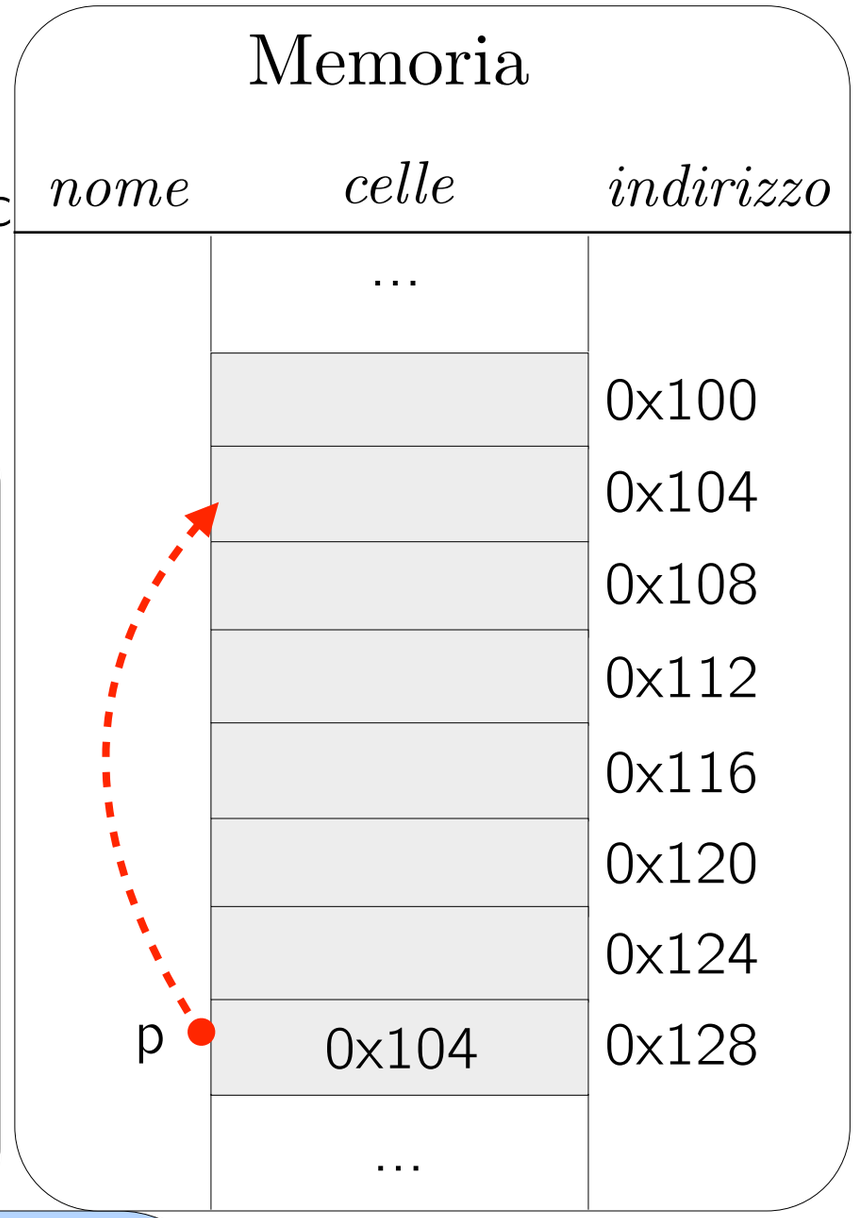
dove p è l'indirizzo di memoria restituito dalla malloc

Esempio

```
int *p = (int *) malloc( 5 * sizeof(int));
free(p);

char *s = (char *) malloc( 5 * sizeof(char));
free(s);

float *f = (float *) malloc( 5 * sizeof(float));
free(f);
```



La memoria è sempre deallocata al termine del programma.

Esercizio 1

My strcat 1

Esercizio

Implementare la funzione

```
char* my_strcat1(char *s1, char *s2)
```

che restituisce un puntatore alla *nuova* stringa ottenuta concatenando le stringhe puntate da s1 e s2.

Scrivere un programma che legga due stringhe da tastiera e stampi la stringa ottenuta concatenandole. Si può assumere che le stringhe in input contengano non più di 1000 caratteri.

Notare che il comportamento di `my_strcat1()` è diverso da quello della funzione `strcat()` presente nella libreria **string**.

L'input è formato da due stringhe di lunghezza non maggiore di 1000 caratteri.
L'unica riga dell'output contiene la stringa ottenuta concatenando nell'ordine le due stringhe inserite.

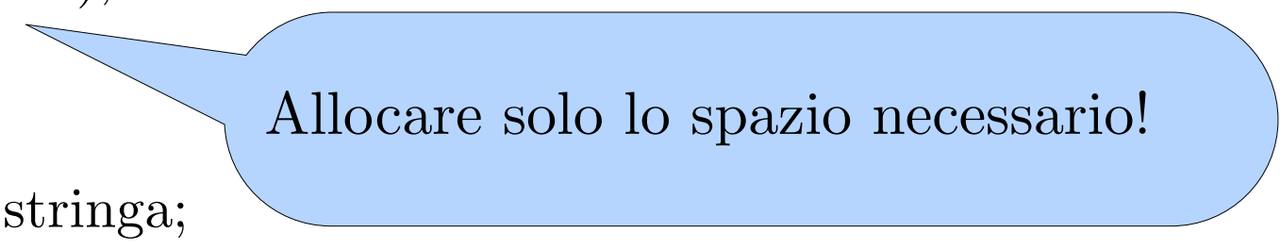
Esercizio 2

My strcat 2

Esercizio

Modificare il codice del precedente esercizio “My Strcat 1” che restituisce una nuova stringa ottenuta concatenando due stringhe passate input.
Questa volta il programma prende in input:

- la lunghezza della prima stringa (e alloca esattamente quanto necessario, ricordarsi il terminatore);
- la prima stringa;
- la lunghezza della seconda stringa;
- la seconda stringa.



L’input è formato, nell’ordine, da: la lunghezza della prima stringa, la prima stringa, la lunghezza della seconda stringa, la seconda stringa.
L’unica riga dell’output contiene la stringa ottenuta concatenando nell’ordine le due stringhe inserite.

Esercizio 3

My strcat

Esercizio

Implementare la funzione

```
char* my_strcat(char *s1, char *s2)
```

che aggiunge la stringa $s2$ al termine di $s1$, sovrascrivendo il terminatore `'\0'` al termine di $s1$ ed aggiungendolo al termine della nuova stringa presente in $s1$ dopo la concatenazione. La funzione restituisce un puntatore ad $s1$.
Si noti che, a differenza dei due esercizi precedenti ("My strcat 1" e "My strcat 2"), in questo caso nessuna nuova stringa viene creata. La funzione assume che in $s1$ vi sia spazio sufficiente per contenere $s2$ (è compito del chiamante assicurarsi che ciò sia vero). Tale comportamento di `my_strcat()` è uguale a quello della funzione `strcat()` presente nella libreria **string**.

Scrivere poi un programma che legga due stringhe da tastiera e stampi la stringa ottenuta concatenandole tramite `my_strcat()`. Si può assumere che le stringhe in input contengano non più di 1000 caratteri.

L'input è formato da due stringhe di lunghezza non maggiore di 1000 caratteri.
L'unica riga dell'output contiene la stringa ottenuta concatenando nell'ordine le due stringhe inserite.

Esercizio 4

My strcmp

Esercizio

Scrivere una funzione

`int my_strcmp(char* s1, char* s2)`

che confronti lessicograficamente $s1$ e $s2$. Il valore restituito è: < 0 se $s1 < s2$; 0 se $s1 == s2$; > 0 se $s1 > s2$.
Si noti che il comportamento di `my_strcmp()` è uguale a quello della funzione `strcmp()` presente nella libreria **string**.

Scrivere poi un programma che legga due stringhe da tastiera e stampi `-1`, `0` o `+1` se la prima stringa è rispettivamente minore, uguale o maggiore della seconda. Si può assumere che le stringhe in input contengano non più di 1000 caratteri.

L'input è formato da due stringhe di lunghezza non maggiore di 1000 caratteri.
L'unica riga dell'output contiene `-1`, `0` o `+1` se la prima stringa è rispettivamente minore, uguale o maggiore della seconda.

Esercizio 5

My strcpy

Esercizio

Scrivere una funzione

`char* my_strcpy(char* dest, char* src)`

che copi *src* in *dest* (incluso il terminatore `'\0'`) e restituisca un puntatore a *dest*. La funzione assume che in *dest* vi sia spazio sufficiente per contenere *src* (è compito del chiamante assicurarsi che ciò sia vero).
Si noti che il comportamento di `my_strcpy()` è uguale a quello della funzione `strcpy()` presente nella libreria **string**.

Scrivere poi un programma che: legga una stringa da tastiera (di lunghezza non maggiore di 1000 caratteri); allochi spazio sufficiente per una seconda stringa destinata a contenere la prima; copi la prima stringa nella seconda; stampi la seconda stringa.

L'input è formato da una sola riga contenente una stringa di lunghezza non maggiore di 1000 caratteri.
L'unica riga dell'output contiene la stampa della seconda stringa.

Esercizio 6 Moltiplicazione di stringhe

Esercizio

Si scriva una funzione

`char* product(char *str, int k)`

che data una stringa *str* e un intero *k* restituisca una stringa ottenuta concatenando *k* volte la stringa *str*.

Si scriva un programma che legga in input:

- una stringa (assumendo che la stringa sia non più lunga di 1000 caratteri);
- un intero, che indica quante volte ripetere la stringa.

e infine stampi l'output di `product()`.

L'input è costituito, nell'ordine, da: una stringa di lunghezza non superiore a 1000 caratteri; un intero k che indica quante volte ripetere la stringa inserita.
L'unica riga dell'output è formata da una stringa contenente k concatenazioni della stringa data in input.