Strutture Dati Dinamiche

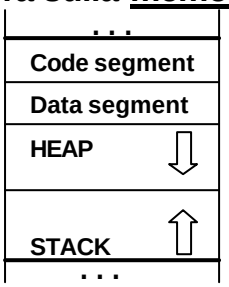
Motivazioni

- Le variabili considerate fino a questo punto devono essere **dichiarate staticamente**, ossia la loro esistenza, il loro nome e la loro dimensione **devono essere previsti a tempo di programmazione (non a tempo di esecuzione!)**
- Mentre per le variabili di tipo scalare o primitivo ciò non costituisce un problema, può essere limitativo per variabili di tipo strutturato (es., struct, stringhe, vettori) per le quali talvolta non è possibile sapere a priori la quantità di dati in ingresso da dover memorizzare
- Per superare la rigidità della dichiarazione statica, occorre un modo per **"chiedere al sistema nuova memoria quando c'è bisogno"**
- Questo è possibile grazie al concetto di **allocazione dinamica**

Allocazione dinamica in C

- **Molti linguaggi, ma non tutti, consentono di "chiedere nuova memoria quando c'è bisogno"**
- **A tale scopo, il linguaggio C utilizza una *primitiva di sistema* che "invia" la richiesta al sistema operativo:**

  ➔*funzione di libreria malloc()*
- **Questa funzione opera sulla memoria HEAP**



Funzione *malloc()*

- **La funzione *malloc()*:**
  - **Chiede al sistema di allocare un'area di memoria grande quanto richiesto dal programmatore**
  - **Se l'allocazione è stata possibile, restituisce l'indirizzo dell'area di memoria allocata**
  - **Se l'allocazione non è stata possibile, restituisce un puntatore con valore NULL (cioè 0)**
  - **Appartiene alla *standard library*, per cui si può utilizzare solo dopo aver indicato nel programma:**

    #include <stdlib.h>

Funzione *malloc()*

- **In pratica, per usare la funzione *malloc()* bisogna:**
  - **specificare la dimensione dell'area di memoria desiderata (*in byte*)**
  - **utilizzare un puntatore per memorizzare l'indirizzo restituito da *malloc()***
- **L'area allocata può essere utilizzata mediante il puntatore**
- **Poiché *malloc()* restituisce un indirizzo puro, è indispensabile un casting per "etichettare" l'area di memoria come contenente dati di un certo tipo**

ESEMPIO (*allocare 12 byte*)

```
int *p;
p  =  (int *) malloc(12);
```

Problema

- **Quante variabili *int* alloca l'istruzione:**

  p = (int *) malloc(12); **?**

  Dipende! Da cosa?

  ➔ Dall'architettura e compilatore utilizzato

- **Infatti, la dimensione dei tipi non è standard. Quindi, 12 byte potrebbero contenere:**
  - ***6 int* in un compilatore che utilizzi *int* da 2 byte**
  - ***3 int* nel caso di un compilatore che usi *int* da 4 byte**

- **MORALE: Per la *portabilità* dei programmi, come parametro di *malloc()* conviene non utilizzare mai valori numerici assoluti!**

Soluzione: Uso della funzione *sizeof()*

Due tipi di applicazione:

sizeof(<espressione>) **Restituisce la quantità di memoria (in *char* o *byte*) necessaria per memorizzare <espressione>**

Es., char str[8]; sizeof(str[8]);

sizeof(<tipo di dato *T*>) **Restituisce la quantità di memoria richiesta (in *char* o *byte*) per memorizzare valori di tipo *T***

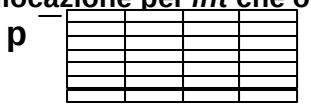
Es., sizeof(int);

Uso della funzione *sizeof()* in combinazione con *malloc()*

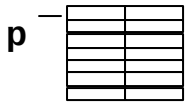
- **Per allocare 7 variabili di tipo *int***

```
int *p;
p = (int *) malloc(7*sizeof(int));
```

- **Risultato dell'allocazione per *int* che occupano 4 byte: 28 byte**



- **Risultato dell'allocazione per *int* che occupano 2 byte: 14 byte**



Attenzione

- **A differenza della memoria occupata dal record di attivazione nella memoria stack, l'area di memoria allocata dinamicamente in una funzione "sopravvive" alla funzione che l'ha allocata fino a quando non viene deallocata esplicitamente**

  "Sopravvive" = "Tempo di vita" ^{1} "Visibilità"

- **E' un errore frequente far riferimento ad una struttura dati che è stata già deallocata**

- **Se c'è necessità di muoversi all'interno delle locazioni di memoria allocate nell'HEAP, usare un'altra variabile e NON MODIFICARE quella restituita dalla malloc()**

Come si rilascia la memoria dinamica

- **La memoria dinamica allocata nella memoria HEAP:**
  - **rimane allocata fino a quando non viene deallocata**
  - **quindi va rilasciata esplicitamente**

  Infatti, nel linguaggio C non vi sono *garbage collector* ("spazzini") come in altri linguaggi, tipo Java e Perl

- **L'area di memoria allocata dinamicamente deve essere rilasciata mediante la funzione**

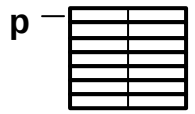
  ***free*(p)**

Utilizzo della memoria dinamica

```
int *p;
p  =  (int *) malloc(7*sizeof(int));
```

- L'area di memoria è usabile tramite il puntatore:
  - o tramite la notazione *p
  - o tramite la notazione p[i]

  che infatti sono sempre equivalenti
- Quindi, in pratica si è creato nell'HEAP un *vettore di 7 int nella memoria dinamica HEAP*:



Memoria dinamica e Memoria statica

- Che differenza c'è tra le seguenti dichiarazioni?

```
int *pdin;
pdin  =  (int *) malloc(7*sizeof(int));
```

?

```
int pstat[7];
```

- Nell'utilizzo?
  ➔ NESSUNA DIFFERENZA
- Nella dichiarazione?
  ➔ DIFFERENZA SOSTANZIALE: Nel caso di memoria dinamica, la dimensione 7 può essere sostituita da una variabile

Esempio

```
#include <stdio.h>
#include <stdlib.h>
main()
 { int vector[15];    /* spazio per 15 interi */
   int *dynVect;    /* spazio per il puntatore al vettore dinamico */
   int k, i;
   printf("Inserire la dimensione desiderata del vettore\n");
   scanf("%d", &k);
   dynVect = (int *) malloc(k * sizeof(int));

   /* ora è possibile usare liberamente sia vector sia dynVect come vettori,
      lunghi 15 e k, rispettivamente */

  for (i=0;i<k;i++)   dynVect[i] = i*i;          /* i primi k quadrati */
  for (i=0;i<15;i++) vector[i] = 2*i;           /* i primi 15 numeri pari */
  free(dynVect);
 }
```


Allocazione dinamica

- **E' possibile allocare memoria dinamica per qualsiasi struttura dati**
  - ***Vettore*** (già visto)
  - ***Singola variabile*** (non si usa, ma è possibile):

    p = (int *) malloc(sizeof(int));
  - ***Stringa***
  - ***Struct***

Allocazione stringa dinamica

- ***Stringa di 10 caratteri***:

```
char *str;
str  =  (char *) malloc(11*sizeof(char));
```

- ***Stringa di dimensione definita da input***

```
char *str;
int dim;
scanf("%d", &dim);
str  =  (char *) malloc(dim*sizeof(char));
```

NOTA: Come nel caso del vettore, non vi è differenza nell'utilizzo della stringa

Allocazione struct dinamica

- ***Struct***:

```
struct persona
  {     char nome_cognome[41];
        char codice_fiscale[17];
        float stipendio;
  } ;
typedef struct persona Persona;
Persona *t, *t10;
t =  (Persona *) malloc(sizeof(Persona));
t10 =  (Persona *) malloc(10*sizeof(Persona));
```

Utilizzare i campi delle *struct* dinamiche

- Vi sono 3 modi per riferirsi agli elementi di una *struct* dinamica:
  - Operatori di dereferenziamento: * e []
  - Operatore di puntatore a struct: ->
- Nel caso dell'esempio precedente, le seguenti modalità sono equivalenti:

  (*t).stipendio = 2354.38;
  t -> stipendio = 2354.38; *Modalità più corretta*

Esercizio 1 (*Specifica*)

- **Scrivere un programma che utilizzi una funzione per leggere da input un certo numero di valori *int* e li inserisca in un vettore allocato dinamicamente dalla funzione stessa.**

  La funzione deve restituire al *main*() il puntatore al vettore dinamico creato

Esercizio 1 (*Algoritmo della funzione*)

- E' necessario chiedere in input il numero di valori che si vogliono inserire
- Si dichiara un vettore dinamico della dimensione richiesta
- Si scandisce tutto il vettore, inserendo elemento per elemento

Esercizio 1 (*Rappresentazione informazioni*)

- Serve un puntatore a int sia nella funzione sia nel main()
- Serve una variabile int per memorizzare la dimensione presa da input
- Serve un int come indice per scandire il vettore

Esercizio 1 (*Programma*)

```
#include <stdio.h>
#include <stdlib.h>
int* creaVett(void);

main()
 { int *pv;
   pv = creaVett();
   free(pv);
 }
int* creaVett(void)
 {  int *v, num, i;
    printf("Quanti valori? "); scanf("%d", &num);
    v = (int *) malloc(num*sizeof(int));
    for (i=0; i<num; i++)
      { printf("v[%d]=", i); scanf("%d", &v[i]); }   /* o anche: scanf("%d", v+i); */
    return(v);
 }
```

Non si sa quanti elementi abbia il vettore.
Il main() e altre eventuali funzioni non possono utilizzare il vettore senza sapere la dimensione.

Esercizio 2 (*Programma*)

```
#include <stdio.h>
#include <stdlib.h>
int* creaVett(int *);
main()
 { int *pv;  int dim, i, neg=0;
   pv = creaVett(&dim);
   for (i=0; i<dim; i++)  { if (pv[i]<0)  neg++; }
   free(pv);
 }
int* creaVett(int *num)
 {  int *v, i;
    printf("Quanti valori? ");  scanf("%d", num);
    v = (int *) malloc(*num * sizeof(int));
    for (i=0; i<*num; i++)
       { printf("v[%d]=", i); scanf("%d", &v[i]); }
    return(v);
 }
```

Versione che riporti la dimensione del vettore mediante un parametro (passato per riferimento). In questo modo, il main() può usare il vettore per calcolare, per esempio, il numero degli elementi negativi.

Esercizio 2bis (*Programma*)

```
#include <stdio.h>
#include <stdlib.h>
void creaVett(int **, int *);
main()
 { int *pv;  int dim, neg=0;
   creaVett(&pv, &dim);
   for (i=0; i<dim; i++)  { if pv[i]<0  neg++; }
   free(pv);
 }
void creaVett(int **pvett, int *num)
 {  int i;
    printf("Quanti valori? ");  scanf("%d", num);
    *pvett = (int *) malloc(*num * sizeof(int));
    for (i=0; i<*num; i++)
       { printf("v[%d]=", i); scanf("%d", *pvett
```

Versione con due parametri che riportino sia il vettore sia la sua dimensione.
La funzione deve restituire il vettore attraverso un parametro passato per riferimento. Poiché il tipo del vettore è un puntatore a int (cioè, int *), il tipo del parametro è un puntatore a ~~puntatore a int~~

Esempio 3 (per esercizio)

- **Scrivere una funzione vett_copy() che, dato un vettore di interi, ne crei un altro uguale, copiando in quest'ultimo il contenuto del primo.**

Esercizio 3

```
void vett_copy(int* v1, int num, int** pv2)
 { int i;
  *pv2 = (int*) malloc(num * sizeof(int));
   for (i=0; i<num; i++)
        (*pv2)[i] = v1[i];
 }
 main()
  { int vettore[] = {20,10,40,50,30};
    int* newVet = NULL;
    vcopy(vettore, 5, &newVet);
    free(newVet);
  }
```

Riepilogo

- Ingredienti fondamentali per la gestione della memoria dinamica:
  - **Funzione malloc()**
  - **Funzione free()**
  - **Variabile puntatore**
- Ingredienti necessari per la gestione della memoria dinamica:
  - **Funzione sizeof()**
  - **Operatore di casting**