.

Gestione della memoria

Stefano Ferrari

Università degli Studi di Milano
stefano.ferrari@unimi.it

Programmazione

anno accademico 2017–2018

Allocazione di memoria

A una variabile puntatore si può assegnare un'area di memoria (intervallo di celle consecutive) nuova, cioè allocata dinamicamente

`p = (`*tipo* `*) malloc(`*dimensione*`);`

La funzione `malloc`, dichiarata nella libreria `stdlib.h`, richiede

- il numero totale di celle assegnate (anche espressione variabile)

Rende un puntatore `void *` da convertire nel tipo giusto con l'operatore di cast

Conviene calcolare implicitamente la dimensione con `sizeof`

Esempio: allochiamo spazio per un numero intero e 10 reali consecutivi

```
int *pi;
double *pd;
pi = (int *) malloc(sizeof(int));
pd = (double *) malloc(10*sizeof(double));
```

.

Strutture dinamiche

L'allocazione consente di decidere durante l'esecuzione quanto spazio riservare, usando un'espressione variabile per la dimensione

La dichiarazione statica, che indica la dimensione massima, soffre di

- consumo eccessivo di memoria nel caso di esigenze inferiori
- sforamenti (e quindi errori) nel caso di esigenze superiori

Le strutture dinamiche combinano elementi semplici in vari modi

- strutture sequenziali ("vettori" dinamici)
- liste concatenate
- alberi
- grafi
- ...

Vettori dinamici

Allocando spazio per più oggetti uguali, si ha l'equivalente di un vettore col vantaggio che la dimensione può variare da un'esecuzione all'altra

```
int n;
n = ...;
int *p1 = (int *) malloc((n+1)*sizeof(int));
int *p2 = (int *) calloc(n+1,sizeof(int));
```

La funzione `calloc` azzera l'intero blocco, oltre ad allocarlo

Si può accedere agli "elementi" del vettore dinamico

- con gli indici numerici: `p[4]`
- (volendo) con l'aritmetica dei puntatori: `*(p + 4)`

.

La funzione `realloc`

```
void *realloc(void *puntatore, size_t dimensione);
```

Ridimensiona l'area di memoria puntata dal *puntatore*

- se *dimensione* cala, le celle in eccesso tornano disponibili
- se *dimensione* cresce, occupa la celle successive (se disponibili) o alloca un'area della nuova dimensione e vi copia quella originale

La funzione `realloc`

- non controlla che vi fosse un'area già allocata a *puntatore*
- non azzera la nuova area allocata né la vecchia
- non controlla se vi fossero altri puntatori all'area riallocata (alias)

Il puntatore `NULL`

Puntatore nullo (`NULL`) è un indirizzo speciale definito in `stddef.h` reso dalle funzioni di allocazione quando non trovano un'area adatta

In genere `#define NULL ((void *)0)`, ma è meglio non fidarsi

Occorre verificare che il puntatore non sia nullo prima di usare l'area

```
int *p = (int *) calloc(n+1,sizeof(int));
if (p == NULL) printf("Errore di allocazione!\n");
```

Si può usare un puntatore come espressione logica:
`NULL` vale `FALSE`, qualsiasi altro puntatore vale `TRUE`

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

.

Deallocazione

L'area allocata è marcata così che successive allocazioni non la usino

Quando non occorre più, l'area allocata va deallocata (resa disponibile) per evitare il memory leakage (consumo progressivo della memoria)

`free(`*puntatore*`);`

dove *puntatore* è l'indirizzo della prima cella dell'area

- un'area va deallocata una volta sola, anche se ha più puntatori
- non si deallocano puntatori a oggetti statici
- dopo la `free`, il puntatore punta ancora l'area: usandolo si accede a celle non allocate (errore); meglio fissarlo a `NULL`
- il puntatore deve indicare la prima cella dell'area da liberare (potrebbe essere stato scalato per ottenere vettori da `S` a `D`)

Matrici dinamiche

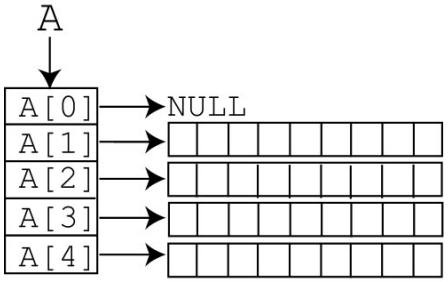
Matrice dinamica è un vettore dinamico di puntatori a vettori dinamici

Allocazione

```
int **A;
int m, n, i, j;
A = (int **) calloc(m+1,sizeof(int *));
for (i = 1; i <= m; i++)
  A[i] = (int *) calloc(n+1,sizeof(int));
```

Deallocazione

```
for (i = 1; i <= m; i++)
  free(A[i]);
free(A);
```

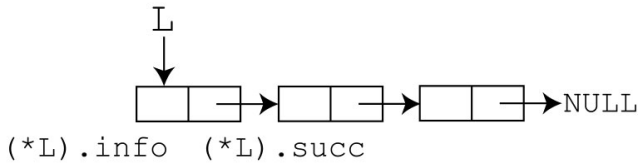


Liste concatenate

Lista concatenata è una catena di strutture (elementi)
ciascuna contenente informazioni e un puntatore alla successiva

```
typedef struct {
  ...  info;
  struct elem_lista *succ;
} elem_lista;

elem_lista *L;
```



Liste concatenate (2)

Rispetto al vettore dinamico, la lista concatenata è

- più flessibile: può allungarsi e accorciarsi a piacere
- meno efficiente: si accede a un elemento scorrendo tutti i precedenti

Per quanto riguarda la memoria occupata

- occupa più spazio per ogni singolo elemento
  (il dato più il puntatore, anziché solo il dato)
- richiede spazio solo per gli elementi effettivamente presenti

.

L'operatore ->

Dato un puntatore pn a una struct

```
typedef struct elem_lista elemento;
elem_lista *pn = (elem_lista *)
malloc(sizeof(elem_lista));
```

per accedere a un suo campo si devono combinare gli operatori * e .

Occorrono le parentesi poiché l'operatore . prevale sull'operatore *

```
(*pn).info = 10;
```

Per evitare una scrittura poco leggibile, si usa la scrittura equivalente

```
pn->info = 10;
```

Gestione delle liste

Date le definizioni:

```
typedef struct elem_lista elemento;
elem_lista *L, *pn, *prev;
```

- per creare un nuovo elemento

  ```
  pn = (elem_lista *) malloc(sizeof(elem_lista));
  ```

- inserire un nuovo elemento in una lista

  ```
  pn->succ = L;
  L = pn;
  ```

- scorrere una lista (l'elemento corrente è *pn)

  ```
  for (pn = L; pn != NULL; pn = pn->succ)
    ...
  ```

- eliminare un elemento da una lista
  (occorre un puntatore prev all'elemento che precede *pn)

  ```
  prev->succ = pn->succ;
  ```