Lezione 9: Strutture e allocazione dinamica della memoria

Laboratorio di Elementi di Architettura e Sistemi Operativi

9 Maggio 2012

Allocazione dinamica della memoria

Memoria dinamica

- È possibile creare strutture dati allocate nella memoria dinamica del processo (*heap*)
- Allocazione al tempo di esecuzione tramite la funzione

```
void* malloc(int numero_di_byte)
```

  - Ritorna `NULL` in caso di errore
  - Per allocare il tipo desiderato si usa l'operatore di cast `(tipo)`
- Richiede `#include <stdlib.h>`
- Due usi:
  - Allocazione dinamica di vettori e matrici
  - Allocazione dinamica di strutture
- È buona regola *liberare* la memoria allocata:
  - `free(<puntatore allocato con malloc>);`

Allocazione dinamica di vettori

- Bisogna dichiarare la dimensione del vettore
- Bisogna sapere quanto spazio (byte) occupa un elemento del vettore:
  - `sizeof(tipo)` ritorna il numero di byte occupati da `tipo`
- Esempi:
  - per allocare un vettore di 10 caratteri:

```
char* s;
s = (char*) malloc(10);
```

  - per allocare un vettore di n interi:

```
int* v;
int n = atoi(argv[1]);
v = (int*) malloc(n*sizeof(int));
```

Allocazione dinamica di matrici

- Allocazione statica

```
float matrice[15]   // 1 dimensione
float matrice[10][10]   // 2 dim
float matrice[12][16][19]    // 3 dim
```

- Allocazione dinamica

```
float *matrice;   // indip. dalle dimensioni
matrice=(float*)malloc(dim1*...*dimN*sizeof(float));
if (matrice == NULL)
{
  fprintf(stderr, "Out of memory\n");
   return -1;  //oppure exit(-1);
}
...
free(matrice);
```

- Accesso ad una matrice bidimensionale allocata staticamente

```
float matrice[10][10]   // 2 dim
...
matrice[i][j] = 3.14;
```

- Accesso ad una matrice bidimensionale allocata dinamicamente

```
float *matrice;
...
matrice = (float*) malloc(10*10*sizeof(float));
matrice[i*num_colonne+j] = 3.14;
```

I tipi di dati strutturati

Tipi strutturati

- In C è possibile definire dati composti da elementi eterogenei (*record*), aggregandoli in una singola variabile
- Individuata dalla keyword `struct`
- Simile alla classi (ma no hanno metodi!)
- Sintassi (definizione di tipo):

```
struct nome {
  campi
};
```

- I campi sono nel formato `tipo nomecampo;`
- Esempi:

```
struct complex {
  double re;
  double im;
};

struct identity {
  char nome[30];
  char cognome[30];
  char codicefiscale[15];
  int altezza;
  char stato_civile;
};
```

- Un definizione di `struct` equivale ad una definizione di tipo
- Successivamente una struttura può essere usata come un tipo per definire variabili e argomenti di funzione:

```
struct complex {
  double re;
  double im;
};

struct complex somma(struct complex num1,
                     struct complex num2)
{
  struct complex num3;
  ...
}
```

- Una struttura permette di accedere ai singoli campi tramite l'operatore '`.`', applicato a variabili del corrispondente tipo struct:

```
struct complex somma(struct complex num1,
                     struct complex num2)
{
  struct complex num3;
  num3.re = num1.re + num2.re;
  num3.im = num1.im + num2.im;
  return num3;
}
```

- È possibile definire un nuovo tipo a partire da una `struct` tramite la direttiva `typedef`
- Passabili come parametri
- Indicizzabili in vettori
- Sintassi: `typedef` *`vecchio_tipo`* `nuovo_tipo;`
- Esempio:

```
typedef struct complex {
  double re;
  double im;
} complex_t;

complex_t num1, num2;
```

Allocazione dinamica di strutture

- Supponiamo di dichiarare una struttura per rappresentare punti nel piano cartesiano:

```
typedef struct pt {
    float x,y;
  } Point;
```

- La dichiarazione `struct pt *pp;` afferma che `pp` è un puntatore ad una `struct pt`;
- `*pp` è la struttura, `(*pp).x` e `(*pp).y` sono i membri
- *Attenzione:* l'espressione `*pp.x` è scorretta!
- Una notazione alternativa (e consigliata) usa l'operatore `->`:

```
pp->x;       pp->y;
```

- Allocazione e deallocazione si gestiscono come i tipi base:

```
Point* segment;
segment = (Point*) malloc(2*sizeof(Point));
/* vettore di due variabili point */
/* equivalente a Point segment[2]  */
if (segment == NULL)
{ fprintf(stderr, "Out of memory\n");
  return -1;
}
...
free(segment);
```

I tipi di dati dinamici

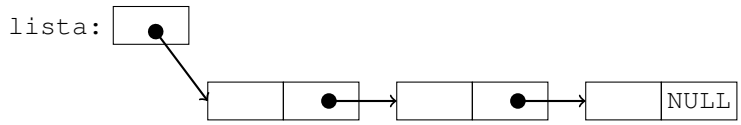
Strutture ricorsive

- Supponiamo di dover implementare una *lista* di numeri interi, di lunghezza non nota a priori
- Ogni elemento della lista è rappresentato da una struttura con due componenti:

```
typedef struct elem {
  int key;
  struct elem *next;
} elemen_t;
```

- `key` contiene il numero
- `next` è un puntatore al prossimo elemento della lista
- Una lista vuota è rappresentata da un puntatore `NULL`, una lista non vuota è rappresentata da un puntatore al primo elemento:

```
typedef elem_t* lista_t;
```



- Test di lista vuota:

```
int is_empty(lista_t lista) {
  return(lista == NULL);
}
```

- Lunghezza di una lista:

```
int length(lista_t lista) {
  int n = 0;
  while(lista != NULL) {
    n++;
    lista = lista->next;
  }
  return n;
}
```

- Inserimento di un elemento in testa alla lista:

```
lista_t insert(lista_t lista, int key) {
  elemen_t *paux;
  paux = (elemen_t *)malloc(sizeof(elemen_t));
  paux->key = key;
  paux->next = lista;
  return paux;
}
```

- Valore del primo elemento:

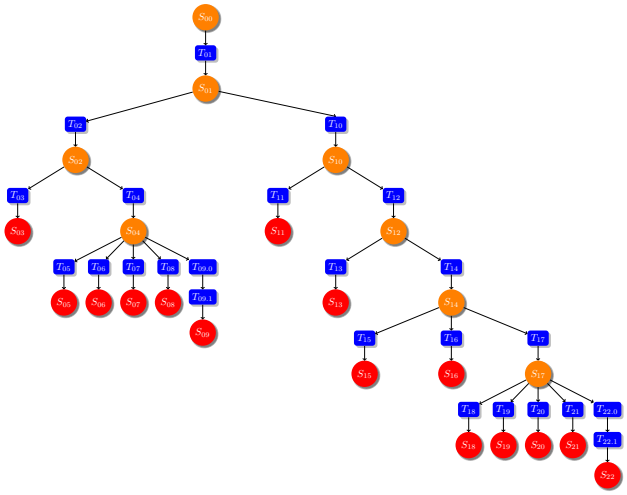
```
int head(lista_t lista) {
   if(lista != NULL) {
     return lista->key;
   }
   return 0;
}
```

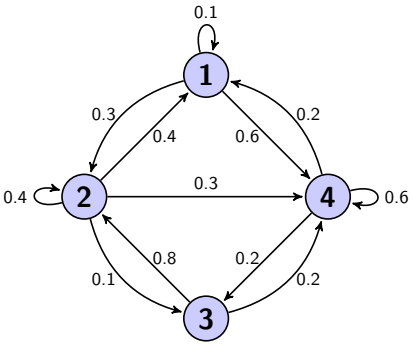
- Rimozione del primo elemento di una lista:

```
void delete(lista_t *plista) {
  elemen_t *paux;
  if(*plista != NULL) {
    paux = *plista;
    *plista = (*plista)->next;
    free(paux);
  }
}
```

Altri esempi di strutture dati dinamiche

- Utilizzando le strutture ricorsive e l'allocazione dinamica della memoria, è possibile rappresentare anche altri tipi di dati dinamici, come *pile*, *alberi* e *grafi*.




0.1
1
0.3
0.4
0.6
0.2
0.4
2
0.3
4
0.6
0.8
0.2
0.1
0.2
3