Corso di Laboratorio di Sistemi Operativi
A.A. 2016–2017

Lezione 11

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

A.A. 2016–2017 - Primo Semestre - 9/11/2016

Allocazione dinamica della memoria

Allocazione dinamica della memoria

Nello scrivere codice che tratti array e stringhe ci si sarà resi conto di una limitazione: la dimensione va dichiarata **staticamente**.

Com'è possibile quindi scrivere programmi che manipolino una quantità arbitraria e ignota a priori di valori?

- Nei programmi dati come esercizio nella lezione 9 e 10, è necessario prevedere quanti numeri verranno letti dallo standard input.
- In tutti gli esempi fatti finora riguardo le strighe, è sempre necessario indicare la lunghezza a priori.

La soluzione è ricorrere ad uno schema di allocazione **dinamica** della memoria.

Allocazione dinamica della memoria

La funzione `malloc()`

La funzione `malloc()` è dichiarata come segue:

```
void *malloc(unsigned n);
```

- La funzione accetta come argomento il numero di **byte** di memoria di cui si ha bisogno.
- Alloca una zona di memoria **contigua** della dimensione richiesta e restituisce un puntatore all'inizio di tale zona.
- Il tipo di ritorno `void *` è un puntatore ad un tipo qualsiasi.

Allocazione dinamica della memoria

Esempio dell'uso di `malloc()`

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

int somma(int *array, int size);

int main() {
  int n = 0;

  printf("Quanti numeri verranno inseriti? ");
  scanf("%d", &n);
  if(n == 0)
    return 0;

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

  printf("Inserire i numeri: ");
  for(int i = 0; i < n; ++i) {
    scanf("%d", elementi + i);
  }
  printf("La somma dei numeri inseriti e': %d\n", somma(elementi, n));

  return 0;
}
```

Liberare la memoria allocata

La funzione `free()` serve a **liberare** la memoria allocata dinamicamente con `malloc()`.

```
int *elementi = malloc(n * sizeof(int));
// ...
free(elementi);
```

- A differenza delle variabili automatiche, la memoria allocata dinamicamente non ha uno **scope** preciso.
- Se non viene esplicitamente liberata dal programma, una zona di memoria allocata dinamicamente resterà assegnata al processo fino alla fine.

Liberare la memoria allocata

Il problema dei memory leak

In software complessi, dove le allocazioni dinamiche sono molte, è difficile tenere traccia di quando liberare la memoria:

- Ad ogni chiamata di `malloc()` dovrebbe seguire una chiamata a `free()`.
- Ma se alloco della memoria in una funzione e restituisco il puntatore al chiamante, chi dovrà poi liberare la memoria?
- Se il puntatore stesso poi andasse fuori scope, non sarebbe più possibile liberare la memoria allocata.
- Errori di questo tipo vengono detti **memory leak**.

Esempio

Creazione dinamica della concatenazione di due stringhe

```c
#include <stdlib.h>
#include <string.h>

char *concat(char *str1, char *str2)
{
  int len = strlen(str1) + strlen(str2);
  char *result = malloc(len + 1);

  strcpy(result, str1);
  strcat(result, str2);

  return result;
}
```

Riallocazione

La funzione `realloc()`

Tramite la funzione `realloc()` è possibile ridimensionare un'area di memoria allocata dinamicamente con una precedente chiamata a `malloc()`, restringendola o allargandola.

```
void *realloc(void *ptr, unsigned new_size);
```

La funzione ritorna un **nuovo** puntatore, perchè i dati potrebbero dover essere spostati se intorno all'area già allocata non c'è spazio sufficiente.

Esempio

Lettura di un numero arbitrario di valori

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

int main() {
  int size = 10;
  int *array = malloc(size * sizeof(int));

  int read = 0;
  while(scanf("%d", &array[read]) == 1) {
    if(++read == size) {
      size *= 2;
      array = realloc(array, size * sizeof(int));
    }
  }

  reverse(array, read);
  for(int i = 0; i < read; i++) {
    printf("%d\n", array[i]);
  }

  free(array);
  return 0;
}
```

Azzerare la memoria allocata

È buona norma, come al solito, inizializzare ad un valore noto la memoria.

- Nell'esempio precedente, nessun elemento veniva letto prima di essere scritto, quindi il programma era corretto.
- In codice più complesso potrebbe essere difficile esserne sicuri.
- È possibile allocare direttamente memoria già azzerata:

```
void *calloc(unsigned count, unsigned size);
```

- La funzione è equivalente a:

```
void *calloc(unsigned count, unsigned size) {
  unsigned len = count * size;
  char *mem = malloc(len);
  for(int i = 0; i < len; ++i) {
    mem[i] = 0;
  }
  return mem;
}
```

- Esempio:

```
int *array = calloc(n, sizeof(int));
```

Le strutture

Le strutture

Le strutture sono un tipo **aggregato**, che raggruppa variabili di tipo diverso in un'unica entità.

- Dichiarazione di una struttura:

```
struct point {
  float x;
  float y;
};
```

La dichiarazione di una struttura definisce un **tipo di dato**.

- Dichiarazioni di variabili di tipo `struct point`:

```
struct point p = { 3, 4 };
printf("%f, %f\n", p.x, p.y);
```

Uso di strutture

Data una struttura è possibile accedere liberamente alle sue componenti oppure operare sull'intera struttura.
Esempi:

- Uso individuale:

```
struct point p = { };
scanf("{ %f , %f }", &p.x, &p.y);
```

- Passaggio di un'intera struttura ad una funzione:

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

float abs(struct point p) {
  return sqrt(p.x * p.x + p.y * p.y);
}
...
struct point p = { 0.707, 0.707 };
printf("%f\n", abs(p)); // Stampa ~ 1
```

Puntatori e array di strutture

Le strutture sono da considerarsi allo stesso livello dei tipi base:

- Possono essere contenuti in **array**:

```
struct point points[] = { { 3, 4 }, { 12, 15 }, { 0, -1 } };
printf("%d %d", points[0].x, points[0].y);
```

- Si possono dichiarare **puntatori** a strutture:

```
struct point   p = { 0, 0 };
struct point *pp = &p;
```

- L'accesso alle componenti di una struttura passando per un puntatore è un'operazione così comune da meritare un **operatore** apposta (`s->var`):

```
printf("%f %f\n", pp->x, pp->y); // Equivalente alla seguente
printf("%f %f\n", (*pp).x, (*pp).y);
```

Uso delle strutture

Le strutture sono uno strumento per organizzare il codice, aggregando dati correlati tra loro.

- Si possono vedere come un antenato del concetto di **classe** della programmazione orientata agli oggetti.
- Attenzione però a non trasferire troppi concetti: il C non è orientato agli oggetti.
- In congiunzione con l'allocazione dinamica, sono i mattoni di base per l'implementazione di qualsiasi **struttura dati** complessa.

Esempio di struttura dati

Lista concatenata

Una lista concatenata può essere descritta come una catena di nodi, ognuno dei quali punta al successivo nella catena.

In C, questa idea si può implementare in modo molto diretto:

```
struct node {
  int data;
  struct node *next;
};
```

Esempio di struttura dati (2)

Lista concatenata

I nodi della lista andranno aggiunti e rimossi in modo non prevedibile a priori, quindi vanno **allocati dinamicamente**.

- Ogni nuovo nodo deve essere inizializzato con il valore del campo `data` e il puntatore `next` impostato a `NULL`.
- Meglio raggruppare queste operazioni in una funzione:

```
struct node *create(int data) {
  struct node *ptr = malloc(sizeof(struct node));
  ptr->data = data;
  ptr->next = NULL;

  return ptr;
}
```

Esempio di struttura dati (3)

Lista concatenata

Esempio di navigazione della lista, la funzione `length()`:

```
int length(struct node *head) {
  int len = 0;
  for(struct node *n = head; n; n = n->next) {
    ++len;
  }

  return len;
}
```

Esempio di struttura dati (4)

Lista concatenata

Ricerca di un elemento in una lista:

```
struct node *find(struct node *head, int data) {
  for(struct node *n = head; n; n = n->next) {
    if(n->data == data)
      return n;
  }
  return NULL;
}
```

Esempio di struttura dati (5)

Lista concatenata

Concatenazione di due liste:

```
struct node *last(struct node *head) {
  for(struct node *n = head; n; n = n->next) {
    if(n->next == NULL)
      return n;
  }
  return NULL;
}

struct node *append(struct node *head1, struct node *head2) {
  struct node *last1 = last(head1);
  last1->next = head2;

  return head1;
}
```

Esempio di struttura dati (6)

Lista concatenata

La memoria occupata da tutti i nodi allocati dinamicamente, alla fine del proprio utilizzo, deve essere rilasciata al sistema operativo.

È opportuno raggruppare in una funzione il rilascio di tutta la lista in un colpo solo:

```
void destroy(struct node *head) {
  struct node *next = head;

  while(next) {
    struct node *n = next;
    next = n->next;
    free(n);
  }
}
```

Esercizi

Esercizi

Per mercoledì 16 novembre

1. Dichiarare una struttura `struct complex` per rappresentare numeri complessi a partire dalla parte reale e immaginaria:
   - Utilizzare il tipo di dato `float` per le componenti.
   - Scrivere le funzioni `cabs()` e `angle()` per il calcolo del modulo e dell'argomento di un numero complesso.
   - Scrivere una funzione `from_polar()` che restituisca un numero complesso a partire da modulo e argomento.

   Nota: Le funzioni trigonometriche si trovano in `<math.h>`

   Nota: In questo esercizio non servono allocazioni dinamiche.
2. Riscrivere le funzioni `length()`, `find()`, `last()`, `append()`, e `destroy()` per le liste concatenate utilizzando un approccio ricorsivo.

Esercizi (2)

Per mercoledì 16 novembre

3. Dichiarare una struttura per la rappresentazione di un **albero binario di ricerca** (non necessariamente bilanciato), ed implementare le relative operazioni:
   - Una funzione `create()` per creare un albero con solo la radice.
   - Una funzione `insert()` per inserire un numero intero all'interno dell'albero.
   - Una funzione `find()` per trovare un valore nell'albero.
   - Una funzione `remove()` per rimuovere un valore dall'albero.
   - Una funzione `destroy()` per liberare la memoria occupata da tutti i nodi dell'albero.
   - Una funzione `to_list()` che restituisca una lista concatenata (implementata a lezione o nell'esercizio 2), contenente i valori dell'albero in ordine di visita.

   Nota: La funzione `remove()` deve occuparsi di liberare la memoria del nodo contenente il valore che viene rimosso.