Esercitazione di Laboratorio - 7

Pagina del corso :
http://didawiki.cli.di.unipi.it/doku.php/fisica/inf/start

Oggi facciamo esercitazione su

- Tipi definiti dall'utente
- Liste
- Allocazione dinamica della memoria
- Ricorsione

Obiettivo: nessun warning!

E' molto importante avere programmi che compilino senza nemmeno uno warning.
Da ora in poi, quindi:

1) useremo `int main()` invece di
   `int main(int argc, char* argv[])`
   quando gli argomenti non vengono usati;
2) assegneremo sempre il valore di ritorno di `scanf` ad una variabile intera;
3) non useremo più la funzione `trunc` ed al suo posto useremo `floor` (o `ceil`).

Il makefile da usare

- Anche oggi useremo lo stesso makefile della scorsa volta

```
# makefile
CC=gcc
CFLAGS=-Wall -g -O -pedantic -Wformat=2 -Wextra -lm
```

ossia quello modificato con l'aggiunta dell'opzione **-lm** (meno elle emme) per la libreria matematica.

- Copiatelo dalla cartella della scorsa volta in una nuova cartella **es07** della vostra home directory, dove metteremo i file di oggi.

Richieste implicite degli esercizi

- Come le volte precedenti, se un esercizio richiede di scrivere una funzione, *implicitamente* vi richiede di scrivere anche un main minimo per preparare dei dati di test e verificare il corretto funzionamento delle procedure scritte.
- Inoltre, dove c'è bisogno di allocazione dinamica della memoria, dovete assicurarvi di deallocare SEMPRE *tutta* la memoria allocata: un esercizio funzionante ma che non deallochi tutta la memoria allocata sarà considerato scorretto.
- Ogni volta che trattate dinamicamente con i puntatori, state molto attenti alla produzione di garbage e ai riferimenti pendenti!

Strutture dati

- In alcuni degli esercizi che seguono (quando richiesto) si utilizzi la seguente definizione di tipo:

```
typedef struct nodo {
  int info;
  struct nodo *next;
} Nodo;
typedef Nodo *ListaInt;
```

- Ricordate che dovete includere la libreria `stdlib.h` per la gestione della memoria dinamica.
- Usate la seguente funzione per stampare le vostre liste:

```
void StampaLista(ListaInt lista){
    while (lista != NULL)  {
        printf("%d -->", lista->info);
        lista = lista->next;
    }
    printf("//\n");
}
```

Algoritmi polimorfi e facili soluzioni

- Con le liste spesso ci troviamo a dover eseguire lo swap di due nodi (o altre operazioni complesse che coinvolgono un ampio uso dei puntatori)
- Poichè ne conosciamo la struttura, potremmo essere tentati di scambiare gli elementi (il contenuto informativo) invece dei nodi stessi (strutture dati)
- Questo è sbagliato: i nostri algoritmi devono essere il più possibile **polimorfi**, cioè astrarre dalla sottostante struttura, per quanto possibile
- Il che significa che i vostri algortimi (a meno di cambiamenti ovvii) devono essere riusabili, per fare un esempio, anche su liste di array o di stringhe

Esercizi proposti

1) Si consideri la definizione del tipo di dati "lista di interi" data nella slide 5
   Ispirandosi a questa definizione, definire un nuovo tipo di dati che rappresenta una lista di interi bidirezionale (*ListaBidir*).
   In questa lista ogni nodo dovrà avere un puntatore al nodo precedente (prev) e un puntatore al nodo successivo della lista (next), oltre al campo informativo (info, un intero).
   Naturalmente se il nodo precedente o il nodo successivo non esistesse, il corrispondente puntatore sarebbe NULL.
   Si noti che a differenza delle liste semplici, per manipolare una lista bidirezionale basta avere un puntatore a uno qualunque dei suoi nodi, non necessariamente alla testa.
   Modificare la funzione **StampaLista** della slide 5 perchè accetti e stampi una *ListaBidir* in questo modo: \\ ← 1 ↔ 2 ↔ 3 ↔ 4 → \\

2) Scrivere una procedura **insInListaBidir** che ha come parametri una *ListaBidir* e un intero, e inserisce l'intero nella lista.
   Come quale modalita' deve essere passata la *ListaBidir*?

Esercizi proposti

3) Definire una funzione **creaListaBidir** (eventualmente sfruttando **insInListaBidir** dell'esercizio 2) che chieda all'utente di fornire una sequenza di numeri interi terminati da un numero negativo e restituisca una *ListaBidir* che contenga i numeri inseriti.

4) Scrivere una funzione **lungListaBidir** che abbia come parametro un puntatore a un nodo di una *ListaBidir* (non necessariamente il primo) e restituisca la lunghezza della lista.

5) Scrivere una procedura **deallocaListaBidir** che riceva come parametro l'indirizzo di un puntatore a un nodo di una *ListaBidir (non necessariamente il primo)* e deallochi tutta la lista.

6) Scrivere una funzione **makeBidir** che prende come parametro una *ListaInt* (lista semplice di interi, vedi slide 5) e ritorna una nuova *ListaBidir* contenente gli stessi elementi.

Esercizi proposti

7) Si consideri la dichiarazione del tipo *ListInt* (lista semplice di interi, vedi slide 5). Scrivere una funzione ricorsiva **bigsRic** che restituisce il numero di elementi della lista il cui valore sia strettamente maggiore della somma di tutti quelli che lo precedono. Esempi:
   - 1 → // restituisce 1
   - 0 → // restituisce 0
   - 1 → 5 → 3 → 10 → // restituisce 3
   - 1 → 2 → 3 → 4 → // restituisce 2
8) Scrivere una procedura **raddoppiaLista** che dopo ogni elemento della *ListaBidir* data aggiunga un nuovo elemento con lo stesso campo info del precedente. Esempio:
   - // ← 4 ↔ 5 ↔ 6 ↔ 7 → //

     deve diventare
   - // ← 4 ↔ 4 ↔ 5 ↔ 5 ↔ 6 ↔ 6 ↔ 7 ↔ 7 → //

Esercizi proposti

9) Facendo riferimento alle slide sugli ordinamenti dell'ultima lezione, scrivere una procedura **ordinaIns** che ordini (con l'algoritmo **Insertion Sort**) una *ListaInt* ricevuta ritornando il puntatore alla nuova testa della lista ordinata.

10) Facendo riferimento all'esercizio 13 della scorsa esercitazione (Laboratorio 6), si definisca una procedura ricorsiva **mergeRic** che date due *ListaInt* ordinate, restituisca una *ListaInt* ordinata contenente tutti gli elementi delle due liste.
A differenza dell'esercizio originale, comunque, questa volta le due liste originali possono essere modificate.