

Esercitazioni di Tecniche di Programmazione

7. Settima esercitazione autoguidata: liste di punti e liste di voli

7.1. Liste di punti (LSTPUNTI.C)

Scrivere un programma che, usando un opportuno insieme di funzioni, costruisca una lista di punti colorati leggendoli da un file di testo, elimini dalla lista tutti i punti disposti più a destra degli altri, stampi la lista e poi la deallochi.

Provare a scrivere l'algoritmo che verrà programmato: già in questa fase possiamo pensare a quali funzioni già note potremo sfruttare (magari aggiungendo solo le definizioni delle funzioni accessorie, come la lettura di un elemento da file o la stampa di una info)

- per la costruzione di una lista da file useremo la funzione già nota ;
- per la ricerca della massima ascissa potremo usare una funzione oppure limitarci a includere del codice apposito nella `main()` (questa e' la soluzione proposta);
- per stampare la lista costruita e controllarne il contenuto (dopo la costruzione, dopo le eliminazioni e dopo la deallocazione) useremo una funzione di stampa, anche questa già nota;
- anche la funzione di deallocazione è nota;
- infine la funzione capace di eliminare da una lista tutti i nodi che presentano una certa caratteristica va studiata a parte: si tratta di una funzione con un algoritmo abbastanza generale e quindi cercheremo (cercherete) di implementarla nel modo più generale e riusabile possibile.

Adesso pensiamo all'algoritmo; poi penseremo ai tipi di dati con cui specializzeremo al nostro caso il tipo `TipoLista`, poi alla funzione `main()` e poi vedremo le funzioni necessarie a completare il tutto.

L'esercizio puo' essere svolto secondo due approcci: si puo' scegliere di costruire completamente il programma in un unico file; oppure si puo' decidere di costruire file header in cui siano definite/dichiarate le strutture dati e le funzioni usate nella `main`, e includere questi file in quello con il programma principale.

7.1.1. primo passo: l'algoritmo

- 1) usare una funzione `costruisciListaDaFile()`, che riceve il nome del file e costruisce una lista, restituendone il puntatore iniziale;
`lista1=costruisciListaDaFile(nomeFile);`
- 2) stampare la lista (per controllo);
- 3) scorrere la lista, cercando il punto di ascissa massima: sia `puntMaxAscissa` il puntatore al nodo (o ad uno dei nodi) avente tale ascissa massima;
- 4) eliminare dalla lista tutti gli elementi in cui il punto ha ascissa massima;
- 5) stampare la lista dopo l'epurazione;
- 6) deallocare la lista.
- 7) ristampare la lista per verificare la deallocazione

Adesso bisogna definire le strutture dati coinvolte nel programma: conosciamo già il `TipoPunto`, definito a suo tempo per i punti colorati; ora si tratta di definire `TipoLista` in modo che i campi informazione siano punti colorati e non più i soliti interi.

In altre parole, `TipoElem` deve essere definito come il tipo dei punti colorati: dopo di ciò, `TipoLista` è il tipo di una lista di punti colorati.

Scrivere le definizioni suggerite e poi verificare nella prossima sezione.

7.1.2. secondo passo: i tipi di dati

Se si segue l'approccio "file unico (vedi introduzione all'esercizio) le dichiarazioni date in questa sezione saranno contenute nel file con il programma (directory **LSTPUNTI**). Se si segue l'approccio basato su file header, queste dichiarazioni sono inserite in un file nella directory (`struct.h`) nella directory **PUNTI_H /LIB**

```
/* definizione TipoPunto come sinonimo della
struttura a tre campi (ascissa, ordinata e colore)*/
typedef
    struct {
        double ascissa;
        double ordinata;
        char colore[10];
    }
    TipoPunto;

/* tipo del dato memorizzato nei nodi della lista */
typedef TipoPunto TipoElem;

/* da qui in poi ... tipi standard gia' noti */
struct structlista {
    TipoElem info;
    struct structlista *next;
};

/* tipo dei nodi della lista */
typedef struct structlista TipoNodoLista;

/* tipo delle variabili "lista" */
typedef TipoNodoLista * TipoLista;

/* tipo dei puntatori ausiliari */
typedef TipoLista PuntNodoLista;
```

Ora si tratta di scrivere il programma principale, prevedendo più o meno quali funzioni verranno usate e come dovranno essere chiamate. Provare a scrivere lo schema di `main()` e poi verificare nella sezione successiva.

7.1.3. programma principale

```
int main() {
    TipoLista lista1;           /* la lista di punti */
    char nomeFile1[20];        /* file di input */
    PuntNodoLista aux,        /* per scandire una lista */
                    puntoMaxAscissa; /* punterà a un nodo avente
                                    ascissa massima */

    printf("nome del file: "); /* file di dati */
    scanf("%s", nomeFile1);

    /* 1) */

    lista1 = costruisciListaDaFile(nomeFile1);

    if (lista1 == NULL) {
        printf("\nLISTA VUOTA\n");
        return 0;
    }

    /** se siamo qui, il programma non e' finito per lista
    vuota ... **/

    /* 2) */

    /* stampe di controllo ... */
    printf("----- lista di partenza -----\n");
    stampaLista(lista1);

    /* 3) */

    /** ricerca del punto di massima ascissa nella lista:
    si usa una funzione minoreAscissa() che riceve due
    puntatori a nodi di lista e dice se il punto del primo
    nodo ha ascissa minore del punto del secondo nodo **/

    puntoMaxAscissa = lista1; /* init massimo parziale */

    /* ciclo di scansione della lista, alla ricerca del massimo */

    aux=lista1->next; /* init (il primo è già stato controllato */

    while (aux!=NULL) {
        /* se aux punta ad un punto di massima
        ascissa parziale
        aggiornamento massimo parziale
        e poi comunque avanti lungo la lista */
        if ( minoreAscissa(puntoMaxAscissa->info, aux->info) )
            puntoMaxAscissa = aux;

        aux = aux->next;
    }

    /* abbiamo usato una funzione minoreAscissa() che, dati due elementi - punti
    - dice se il primo è minore del secondo o no */
    /* ora puntoMaxAscissa punta sul nodo di massima ascissa nella lista (nella
    lista, comunque, ci potrebbero essere piu' nodi con quell'ascissa) */

    /* 4) */
}
```

```

        /* eliminazione di tutti i nodi di lista in
        cui l'ascissa e' uguale a quella del massimo
        appena calcolato */
    eliminaTutti(&lista1, puntoMaxAscissa->info) ;

/* 5) */
        /* stampe di controllo ... */
    printf("----- lista dopo le eliminazioni ----- \n");
    stampaLista(lista1);

/* 6) */
        /* deallocazione delle due liste */
    deallocaLista (&lista1);

/* 7) */
        /* stampe di controllo ... */
    printf("----- lista dopo la dealloc ----- \n");
    stampaLista(lista1);

printf("\nFINE\n");
return 0;
}

```

Ora è utile provare ad elencare i prototipi delle funzioni che servono: ci sono funzioni che hanno un'utilità diretta (come `costruisciListaDaFile()`, o `eliminaTutti()`, perché vengono chiamate direttamente per eseguire parti dell'algoritmo; ci sono poi funzioni di supporto, che vengono chiamate da quelle suddette (come `stampaElem()`, o `leggiElemDaFile()`)

Per inserire un elemento in lista abbiamo riusato `insTestaLista()`, che implementa un algoritmo di inserimento in testa; ovviamente va bene anche qualunque altro tipo di inserimento, infatti il problema non specifica niente riguardo all'ordine dei dati in lista e quindi, da questo punto di vista, siamo liberi di scegliere la strada che preferiamo (in particolare, per indolente pigrizia, scegliamo inserimento in testa).

Provare a elencare i prototipi, almeno per le funzioni di uso diretto. E poi verificare nella sezione successiva.

7.1.4. funzioni (dichiarazione)

Se si segue l'approccio "file unico (vedi introduzione all'esercizio) le dichiarazioni date in questa sezione saranno contenute nel file con il programma (directory **LSTPUNTI**). Se si segue l'approccio basato su file header, queste dichiarazioni sono inserite in un file nella directory (funzioni.h, funzioni.c) nella directory **PUNTI_H /LIB**

```
        /* funzioni che verranno usate direttamente */

TipoLista costruisciListaDaFile (char *);
void stampaLista(TipoLista);
void eliminaTutti (TipoLista *, TipoElem);
void deallocaLista (TipoLista *);

        /* funzioni di supporto */

void insTestaLista (TipoLista *, TipoElem);
/* usata da costruisciLista() */

void leggiElemDaFile (FILE *, TipoElem *);
/* usata da costruisciLista() */

void stampaElem (TipoElem);
/* usata da stampaLista() */
void stampaPunto(TipoPunto );
/* usata da stampaElem() */

int minoreAscissa (TipoElem, TipoElem);
/* usata da eliminaTutti() */
```

Adesso provare a definire le funzioni suddette: se servono suggerimenti, stanno nella sezione seguente; poi c'è la sezione con le definizioni della soluzione proposta.

7.1.5. funzioni (definizione): suggerimenti

Se si segue l'approccio "file unico (vedi introduzione all'esercizio) le definizioni suggerite e date qui e nelle sezioni successive, saranno contenute nel file con il programma (directory **LSTPUNTI**). Se si segue l'approccio basato su file header, tutto sarà inserito in un file (`funzioni.h`, `funzioni.c`) nella directory **PUNTI_H /LIB**

- `costruisciListaDaFile()` dovrebbe essere semplicemente importata da esercizi precedenti; idem per `insTestaLista()`, che realizza l'inserimento in testa;
- e per farla funzionare sarà necessario scrivere ad hoc la funzione `leggiElemDaFile()`, che deve essere capace di leggere da un file un punto (cioè il particolare `TipoElem` con cui lavoriamo) e assegnarlo ad una variabile in modo opportuno.
- per `stampaLista()` valgono discorsi analoghi (`stampaLista()` e `stampaElem()` sono da prendere in esercizi già fatti; `stampaElem()` va modificata in modo che richiami `stampaPunto()`, che deve essere definita prendendola, anche lei, da esercizi già fatti.
- `eliminaTutti()` realizza l'algoritmo con cui tutti gli elementi che verificano una certa condizione vengono eliminati dalla lista. La condizione di eliminazione è di che il punto abbia ascissa uguale a quella massima; per calcolarla usiamo la funzione `minoreAscissa()`, che era stata definita per la fase di ricerca dell'ascissa massima e che qui può essere riutilizzata (evitando così di definire una ulteriore funzione).
- la funzione `deallocaLista()` è un'altra funzione da importare da esercizi precedenti. Si tratta di una funzione applicabile anche nel caso in cui `TipoElem` è un punto, senza modifiche o altre funzioni da aggiungere. La funzione `deallocaLista()` è in generale largamente riutilizzabile, a patto che l'informazione associata ad un nodo di lista sia deallocabile con una sola applicazione di `free()` (cioè, in poche parole, non ci siano in info campi puntatore a locazioni allocate dinamicamente).

7.1.6. funzioni (definizione): costruzione lista da file

```
/* *****  
/* legge dati dal file di nome nF e costruisce la loro lista; i dati  
si immaginano memorizzati in un file di testo, un dato per riga */  
TipoLista costruisciListaDaFile (char * nF) {  
FILE * fin;  
TipoElem el;  
TipoLista res = NULL; /* sara' la lista che viene costruita */  
char ch;  
  
    printf("\n --- COSTRUISCO LISTA DA  %s ---\n", nF);  
  
    fin = fopen(nF, "r");  
    if (fin==NULL) {  
        printf("AARGH!");  
        return NULL;  
    }  
  
    /* se siamo qui, vuol dire che fopen era andata bene */  
  
    while (!feof(fin)) {  
        leggiElemDaFile(fin, &el);  
        fscanf(fin, "%c", &ch); /*per andare a capo (che succede senza?)*/  
        insTestaLista(&res, el);  
    }  
  
return res;  
}
```

```
/* *****  
/* legge un dato dal file f, e lo memorizza in *pelem) Assumiamo che  
il dato sia un punto e si disposto come detto sopra e che la funzione  
non venga mai chiamata su un file terminato */  
  
void leggiElemDaFile (FILE * f, TipoElem *pelem) {  
    TipoPunto v;  
    /** LETTURA DI UN PUNTO, DISPOSTO SU UNA LINEA DEL FILE **/  
    fscanf (f, "%lf", &(v.ascissa));  
    fscanf (f, "%lf", &(v.ordinata));  
    fscanf (f, "%s", v.colore);  
    /* E ASSEGNAZIONE A *pelem*/  
    *pelem = v;  
return;  
}
```

```

/*****/
/* inserisce elem nella lista; plis e' l'indirizzo della variabile
puntatore che punta all'inizio della lista */
void insTestaLista(TipoLista * plis, TipoElem elem) {

    PuntNodoLista nuovo;
                                /*allocazione nuovo nodo */
    nuovo=malloc(sizeof(struct structlista));

/*se l'allocazione ha successo, inseriamo il nuovo nodo in testa alla lista */
    if (nuovo==NULL)
        printf("EEEEKK!!! manca memoria per un nodo!!!\n\n");
    else {
        nuovo->info=elem;        /* riempimento nuovo nodo con l'info */
        nuovo->next=*plis;      /* nuovo nodo sopra al primo della lista */
        *plis=nuovo;           /* ora il nuovo nodo e' il primo della lista */
    }
    return;
}

```

7.1.7. funzioni (definizione): stampa della lista

```

/*****/
/* stampa sul video una lista */

void stampaLista (TipoLista l) {
    while (l) {
        stampaElem(l->info); /* si assume sia stampato un elemento su una riga */
        putchar('\n');
        l=l->next;
    }
    return;
}

/*****/
/* stampa sul video un elemento di lista facendo uso della funzione
stampaPunto() che gia' conosciamo */

void stampaElem (TipoElem elem) {

    stampaPunto(elem);
    return;
}

/*****/
/* ecco la funzione stampaPunto di antica e cara memoria: funzione
che riceve un punto e lo stampa */

void stampaPunto(TipoPunto p) {
    printf("(%g, %g, %s)", p.ascissa, p.ordinata, p.colore);
    return;
}

```

7.1.8. funzioni (definizione): minoreAscissa()

```
/* ***** */
/* restituisce 1 se il primo punto ha ascissa minore del secondo */
int minoreAscissa (TipoPunto e1, TipoPunto e2) {

return ( (e1.ascissa < e2.ascissa) );
}

```

7.1.9. funzioni (definizione): eliminaTutti()

```
/* ***** */
/* elimina dalla lista tutti gli elementi che hanno ascissa
maggiore o uguale a quella del punto elem;
plis e' l'indirizzo della variabile puntatore che punta all'inizio della
lista
Si scandisce la lista, con due puntatori, in modo che
- se l'elemento puntato da corr e' da eliminare,
lo si elimina e si porta corr sul successivo
- altrimenti si fanno avanzare prec e corr */
void eliminaTutti(TipoLista * plis, TipoElem elem) {
PuntNodoLista prec, corr, pgen;

/*creazione record generatore */
pgen=malloc(sizeof(struct structlista));

if (pgen==NULL) { /* problemi: usciamo subito */
printf("EEEEK!!! manca memoria per il rec.gen!!!\n\n");
return;
}
else { /* preparazione della scansione: pgen viene posto prima del
primo elemento in lista, prec viene fatto puntare al record
generatore e corr viene fatto puntare al primo in lista */
pgen->next = *plis;
prec=pgen;
corr = *plis;
}

/* posizionamento di prec e corr in modo che corr punti
all'elemento da eliminare e prec punti al nodo immediatamente
precedente */
while ( (corr!=NULL) )
if ( corr->info.ascissa == elem.ascissa ) {
prec->next = corr->next; /* eliminare e riposizionare corr */
free(corr);
corr=prec->next;
} else {
prec=corr; /* solo riposizionare corr e prec */
corr=corr->next;
}

/* adesso bisogna eliminare il record generatore e fare in modo che list
punti al primo nodo effettivo della lista */
*plis=pgen->next;
free(pgen);
return;
}

```

7.1.10. funzioni (definizione): deallocaLista()

```
/* **** */
/* deallocazione lista
   dato che possiamo liberarci di un nodo con un semplice free,
   questa e' la funzione generale di deallocazione
   (non c'e' bisogno di trattamenti speciali per l'eliminazione
   di una info
*/
void deallocaLista (TipoLista *plis) {
    PuntNodoLista aux;

    while (*plis) {
        aux = *plis;
        *plis = (*plis)->next;
        free(aux);
    }
    return;
}
```

7.2. Liste di voli (directory LSTVOLI1 e VOLI_H1)

Questa esercitazione è stata ideata seguendo la falsariga di quella precedente: un programma esegue un certo insieme di funzionalità, gestendo liste di particolari `TipoElem`, che in questo caso sono oggetti di tipo `TipoVolo`. Rispetto all'esercizio precedente ci sono più liste con cui avere a che fare e bisogna operare su liste in modo meno elementare; inoltre è previsto il salvataggio su file di testo dei dati rimasti nelle liste.

L'esercizio può essere svolto secondo due approcci: si può scegliere di costruire completamente il programma in un unico file; oppure si può decidere di costruire file header in cui siano definite/dichiarate le strutture dati e le funzioni usate nella main, e includere questi file in quello con il programma principale. La proposta di soluzione, nel primo approccio, è data nella directory `LSTVOLI1`. La proposta di soluzione, nel secondo approccio, è data nella directory `VOLI_H1`. In entrambe le directory (ci sono opportuni file `README.TXT` da leggere, se interessa)

7.2.1. problema e algoritmo

`nomeFile1` e `nomeFile2` sono due file contenenti dati (non specifichiamo di che tipo). Scrivere un programma che stampa ed elimina tutti i dati comuni ai due file.

Una soluzione didatticamente interessante è la seguente:

- 1) costruire una lista `lista1`, con i dati di `nomeFile1`;
- 2) costruire una lista `lista2`, con i dati di `nomeFile2`;
- 3) scansione di `lista1`, con un puntatore `aux`, in modo che, se `aux->info` è anche in `lista2`, venga stampato ed eliminato da entrambe le liste;

```
while (aux!=NULL) {
    if ( esisteInLista(aux->info, lista2) ) {
        stampElem(aux->info);
        elimLista(&lista1, aux->info);
        elimLista(&lista2, aux->info);
    }
    aux = aux->next;
}
```

- 4) scaricamento della lista `lista1` nel file `nomeFile1`;
- 5) scaricamento della lista `lista2` nel file `nomeFile2`;
- 6) deallocazione `lista1`;
- 7) deallocazione `lista2`

7.2.2. secondo passo: i tipi di dati

valgono considerazioni analoghe al caso delle liste di punti: bisogna definire opportunamente `TipoElem` come sinonimo di `TipoVolo`: dopodiché i tipi standard delle liste assumeranno la forma che ci serve, senza ulteriori modifiche.

7.2.3. terzo passo: main()

Ecco il programma principale, nel quale sono usate variabili di tipo `TipoLista` (per le liste) e stringhe di al più 19 caratteri per i nomi dei file:

```
int main() {
    TipoLista lista1, lista2;
    char[20] nomeFile1, nomeFile2;
    PuntNodoLista aux; /* per scandire una lista */
}
```

```

printf("nome del primo file: ");
scanf("%s", nomeFile1);
printf("nome del secondo file: ");
scanf("%s", nomeFile2);

lista1 = costruisciListaDaFile(nomeFile1);
lista2 = costruisciListaDaFile(nomeFile2);

aux = lista1;

while (aux!=NULL) {
    /* se aux->info e' (anche) in lista2,
                                   stampalo ed eleiminalo */

    if ( esisteInLista(aux->info, lista2) ) {
        stampElem(aux->info);
        elimLista(&lista1, aux->info);
        elimLista(&lista2, aux->info);
    }
    /* e poi comunque vai avanti */
    aux = aux->next;
}

/* registrazione delle due liste nei file originali;
cosi' gli elementi comuni (che non sono piu'
nelle liste) non saranno piu' nemmeno nei file */

stampaListaSuFile(lista1, nomeFile1);
stampaListaSuFile(lista2, nomeFile2);

/* deallocazione delle due liste */

deallocaLista (lista1);
deallocaLista (lista2);

printf("\n\nFINE\n");
return 0;
}

```

7.2.4. funzioni (dichiarazione e definizione)

rimandiamo al contenuto del file LSTVOLI.C

7.3. Secondo esercizio sulle liste di voli (directory LSTVOLI2 e VOLI_H2)

In questa esercitazione si riprende l'esercitazione autoguidata sulle tabelle di voli. Stavolta le strutture dati utilizzate sono intrinsecamente dinamiche: si tratta di liste concatenate, rappresentate mediante struct e puntatori. (Una interessante ricaduta e' che non e' piu' necessario conservare l'informazione relativa al massimo numero di voli inseribili).

L'esercizio puo' essere svolto secondo due approcci: si puo' scegliere di costruire completamente il programma in un unico file; oppure si puo' decidere di costruire file header in cui siano definite/dichiarate le strutture dati e le funzioni usate nella main, e includere questi file in quello con il programma principale. La proposta di soluzione, nel primo approccio, e' data nella directory LSTVOLI2. La proposta di soluzione, nel secondo approccio, e' data nella directory VOLI_H2. In entrambe le directory (ci sono opportuni file README.TXT da leggere, se interessa)

In un primo stadio di sviluppo del programma, pensiamo solo le funzioni utili per una gestione di base, cioe':

- inserimento di un volo nella lista dei voli (inserimento in testa), supponendo di utilizzare una funzione che, data una lista e un volo, inserisce il volo nella lista e un'altra funzione che, data una lista e un intero k, inserisce k voli nella lista;
- lettura della lista;

7.4. strutture dati (LSTVOLI2/VOLI1.C oppure directory VOLI_H2/VOLI1)

Anche qui e' necessario definire i tipi dei dati che utilizzeremo. In sostanza si tratta di modellare delle struct apposite. Ovviamente tali strutture devono essere definite "top level", in modo che siano visibili a tutte le funzioni definite successivamente (la main() e le altre funzioni).

(e' chiaro che definire quelle strutture direttamente nella main() e' un errore: si tratterebbe di definizioni la cui visibilita' non va oltre il codice della funzione main(), cioe' di cose invisibili alle altre funzioni ...).

Forza, definite questi tipi, tanto per cominciare.

Suggerimenti di seguito ...

Suggerimenti:

Un'ipotesi per i tipi da usare potrebbe essere la seguente (ma ce ne sono sicuramente di migliori ...)

```
struct volo{
    char codice[6];
    char *destinazione;
    int ora;
    int minuti;
    int postiLiberi;
};
typedef struct volo tVolo;

struct nodoLista{
    tVolo unVolo;
    struct nodoLista *nextVolo;
};
typedef struct nodoLista tNodo;

typedef tNodo *tLista;
```

7.5. inserimento ordinato in lista (LSTVOLI2/VOLII.C oppure directory VOLI_H2/VOLII)

L'approccio iniziale prevede l'inserimento in testa di k elementi, con k letto da input, e la stampa degli elementi nell'ordine con il quale sono stati inseriti.

Usiamo una funzione di stampa di una lista di voli e una funzione che riceve k e una lista e aggiunge nella lista k voli.

La funzione di stampa, potrebbe usare una funzione che, dato un nodo della lista ne stampa l'informazione (il volo). Quest'ultima funzione, a sua volta, potrebbe usare una funzione StampaVolo(), progettata ad hoc per il particolare tipo di informazione contenuto nelle liste di voli (questa funzione non e' implementata nella soluzione proposta)

La funzione di aggiunta probabilmente usera' una funzione che, data una lista e un volo, aggiunge il volo in lista.

Suggerimenti nella prossima pagina ...

Suggerimenti:

I prototipi delle funzioni necessarie in questa prima fase, potrebbero essere:

```
void insTestaLista(tLista *pInizioLista, tVolo *pNuovoVolo);  
    /*inserisce un nuovo volo all'inizio della lista*/  
void inserisciVoli(int numeroVoli, tLista *pLista);  
    /*inserisce un numero di voli definito dall'utente nella lista puntata da pLista*/  
void stampaLista (tLista laLista);  
    /*stampa la lista il cui primo nodo si trova alla locazione puntata da laLista  
    (nel typedef si evidenzia il carattere di puntatore del tipo tLista*/
```

Suggerimenti (2):

```
void stampaNodoLista(tNodo *pNodo);  
/* stampa i contenuti informativi puntati da pNodo, che sono di tipo tVolo*/  
void leggiVolo(tVolo *pVolo);  
    /* legge un oggetto di tipo Volo (una struttura) e lo memorizza nella  
    locazione puntata da pVolo */
```

Questo frazionamento in tante funzioni e' utile, in generale, per concentrare i propri sforzi su compiti non troppo grandi; in questo modo si ottiene anche il vantaggio di mantenere certe funzioni il piu' possibile indipendenti dal tipo di informazione contenuta in lista. Ad esempio, se dovessimo implementare una lista di televisori, la struttura del tipo di informazione cambierebbe, la struttura della funzione stampaTelevisore() sarebbe ovviamente diversa da quella della stampaVolo(), ma la struttura della funzione stampaLista() rimarrebbe uguale ... cosi' questa funzione sarebbe altamente riusabile (o comunque piu' riusabile di una sua versione in cui i particolari di stampa legati al tipo di informazione sono programmati esplicitamente nel suo codice (e cambiano molto da voli a televisori). La prossima sezione va vista come un esempio di cosa succede quando si deve modificare il programma in conseguenza di qualche modifica nei dati gestiti (l'aggiunta di qualche informazione in un volo).

7.6. *cambiamenti (LSTVOLI2/VOLI2.C oppure directory VOLI_H2/VOLI2)*

Che succede se nel tipo volo si desidera aggiungere anche il nome del capitano del volo?

Quali modifiche sono da apportare al precedente programma?

Prova a fare le modifiche e poi guarda il file `voli2.c` o la directory `voli2`

7.7. *estensione dell'applicazione, cioe' del programma (LSTVOLI2/VOLI3.C oppure directory VOLI_H2/VOLI3)*

E' corretto pensare che una piccola applicazione permetta all'utente di scegliere tra piu' funzioni singole. Questo si realizza facilmente con un menu ed una iterazione dell'acquisizione di scelta. In questa fase si puo' pensare di inserire anche una funzione fondamentale per l'utente: la modifica del volo. E' necessario a questo punto fermarsi a pensare seriamente agli elementi che si possono modificare, considerando la struttura dati definita:

- il codice del volo rappresenta l'elemento per l'identificazione del volo stesso, quindi non sara' modificabile;
- la destinazione del volo, puo' variare, ma solo in casi eccezionali;
- il numero dei posti puo' variare (in positivo corrisponde ad una prenotazione, in negativo ad un annullamento); in considerazione di cio' per mantenere un rapporto con la realta' e' necessario effettuare delle verifiche sulla 'consistenza' dei dati inseriti (il numero dei posti da prenotare non puo' essere superiore al numero dei posti totali, il numero dei posti liberati non puo' essere superiore al numero dei posti occupati);

Suggerimento (+/-)

In ragione di tutte queste possibili modifiche puo' risultare opportuno creare un sottomenu' di modifica del volo, come in `voli3.c`. (Ma prima scrivi la tua soluzione e solo poi guarda il file)

7.8. *file & c. (LSTVOLI2/VOLI4.C oppure directory VOLI_H2/VOLI4)*

E' possibile utilizzare un file per conservare tutte le modifiche effettuate:

Si tratta di sviluppare elementi del menu e relative funzioni che permettano di leggere dati da un file, memorizzarli e poi, dopo averli usati a piacere, liberare la memoria occupata dalla lista.

A parte le funzionalita' che ormai abbiamo gia' realizzato, i prototipi delle funzioni necessarie, potrebbero essere:

```
void leggiVoli(tLista *pLista, char *nomeFile);

    /* legge i voli dal file e, utilizzando un ciclo di lettura da
    file, inserisce i dati nella lista*/

void memorizzaVoli(tLista pNodo, char *nomefile);

    /* scorre la lista dall'inizio e la memorizza nel file, liberando,
    tramite la struttura di appoggio temporanea creata ad hoc, la
    memoria relativa alla destinazione ed al nome del comandante*/
```

Il meccanismo utilizzato per liberare la memoria consiste nello scorrere la lista, mantenendo un puntatore di appoggio che punta all'elemento precedente: una volta memorizzati i valori contenuti nel precedente si puo' deallocare la memoria. Un approccio e' presentato in [voli4.c](#).

Suggerimenti:

La funzione di cancellazione dovrebbe prevedere lo scorrimento della lista fino a trovare il volo contenente il codice da cancellare. Per poter effettuare il collegamento e' necessario che ci sia una verifica che non sia proprio il primo nodo della lista (nel qual caso si dovra' modificare l'inizio lista, da cui la necessita' di passare per indirizzo l'inizio della lista) e, nel caso il nodo non sia in testa alla lista, e' invece necessario, nello scorrere la lista, portare anche il puntatore 'precedente' per operare gli 'scollegamenti' ed il relativo collegamento. Un possibile prototipo della funzione potrebbe essere:

```
void cancellaVolo(tLista *pLista)
```