FONDAMENTI DI INFORMATICA II

ESERCITAZIONE n° 3: Linguaggio C

1. Allocazione dinamica della memoria

ESERCIZIO n° 22

Scrivere un programma che allochi dinamicamente spazio per al più 10 stringhe di al più 80 caratteri.

Soluzione

Finora, tutte le variabili viste erano o variabili globali (definite e allocate staticamente) o variabili locali a una funzione (definite staticamente e allocate/deallocate automaticamente a run-time). Tuttavia, le variabili definite staticamente hanno dei limiti, primo fra tutti quello di dover conoscere a priori la loro dimensione massima, che le rende inadatte a certi campi di applicazione.
Per ovviare a questo problema, il C mette a disposizione due primitive, `malloc` e `free`, per allocare e deallocare esplicitamente delle nuove variabili a tempo d'esecuzione. Più precisamente, la primitiva `malloc` chiede al sistema un'area ampia *n* bytes, e ne restituisce l'indirizzo sotto forma di `void*` (ossia di puntatore generico), mentre la primitiva `free` libera l'area associata a quell'indirizzo (non è necessario specificare nuovamente la dimensione in quanto provvede il sistema a tenere traccia di queste informazioni).

Per risolvere il problema dato occorre dunque predisporre un ciclo che legga dieci stringhe da tastiera, ognuna lunga al più 80 caratteri, e le ricopi ognuna in una nuova area di memoria, chiesta da `malloc` al sistema volta per volta sulla base della *lunghezza effettiva* della stringa letta. I puntatori a tali aree di memoria potranno essere memorizzati in un apposito *vettore di puntatori*. Un possibile programma che realizzi tutto questo è il seguente:

```
#include <stdio.h>
#include <stdlib.h>  /* contiene prototipi di malloc e free */
#include <string.h>

main(void)
{
     char line[81];
     char *s[10];
     int i;

     for(i=0; i<10; i++)
     {
          gets(line);
          s[i]= (char *) malloc(strlen(line)+1);
          strcpy(s[i], line);
     }

     for(i=0; i<10; i++)
     {
          printf("\n%s", s[i]);
```

```
            free(s[i]);
        }
        return 0;
}
```

E' importante notare che `malloc`, quando riserva memoria, non ha alcuna informazione sul tipo di uso che ne verrà fatto; per questo, essa si limita a restituire *l'indirizzo* di quell'area, sotto forma di `void*` che, com'è noto, proprio per questo non è dereferenziabile. Per utilizzare praticamente l'area riservata è quindi *indispensabile* usare il cast `(char*)`per convertire esplicitamente il puro indirizzo in un puntatore del tipo corrispondente all'uso che si fa di tale area di memoria.

Dal punto di vista pratico, il programma è completato da un secondo ciclo che rilegge e visualizza le stringhe precedentemente immesse, deallocando quindi l'area di memoria corrispondente tramite `free`. L'area liberata potrà essere nuovamente utilizzata in successive richieste a `malloc`, nei tempi e modi stabiliti della politica di gestione della memoria adottata dal compilatore.

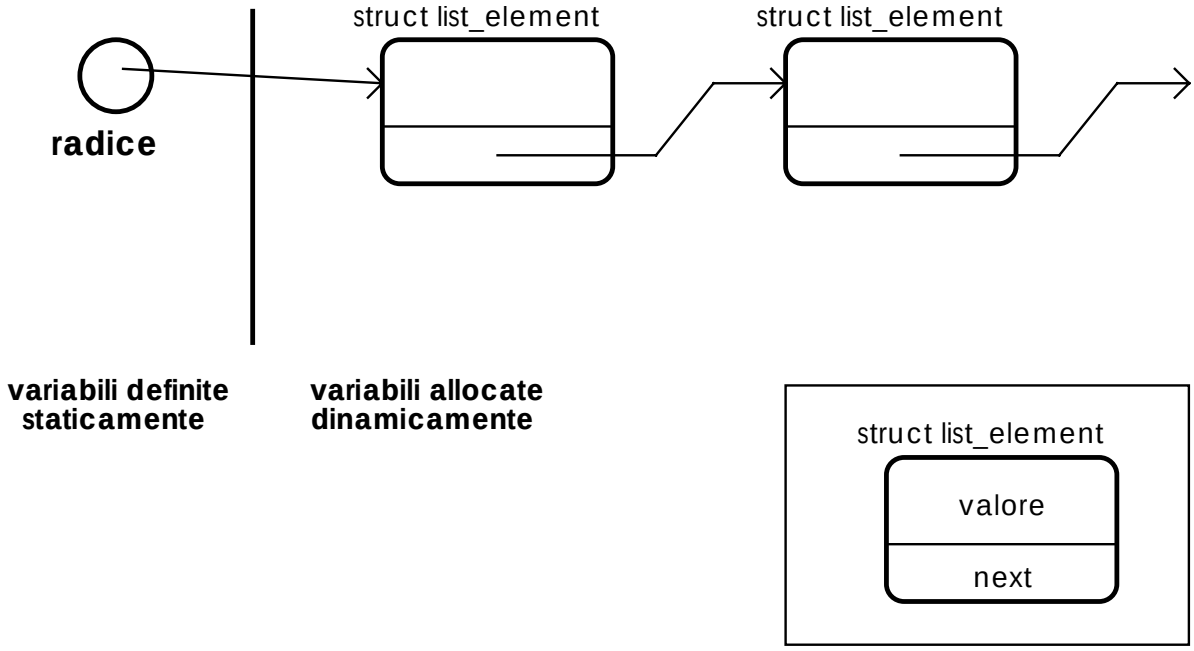
Per finire, osserviamo che, se tutto si limitasse a questo, al di là della maggior generalità ottenuta nel trattamento stringhe (e in generale di vettori) non si vedrebbe la reale utilità delle primitive di allocazione dinamica, in quanto, pur potendo allocare nuove variabili dinamicamente, e quindi in modo svincolato dalle definizioni statiche, di fatto bisognerebbe però avere definito staticamente il puntatore da utilizzare con la `malloc`, il che riporterebbe il problema al punto di partenza.

In realtà, la reale potenza dell'allocazione dinamica di memoria emerge trattando strutture dati più evolute dei semplici interi, caratteri o vettori, ossia utilizzando *strutture dati definite ricorsivamente*, come *alberi* e *liste*.

2. Liste

ESERCIZIO n° 23
Scrivere un modulo base che permetta l'inserimento di numeri interi in una *lista*.

Soluzione
La lista è un esempio di struttura-dati definita ricorsivamente, la cui organizzazione è riassunta dalla figura seguente.



In sintesi, ogni elemento (detto *nodo*) della lista è concettualmente costituito di due parti: una destinata a contenere il "valore" di quel nodo (che può essere anche un'entità molto complessa, come una struttura, un vettore, o qualunque altra cosa) e un'altra, di tipo *puntatore a nodo*, in grado di referenziare l'elemento successivo. All'inizio, l'unica cosa che esiste della lista è la sua *radice*, ossia il puntatore al suo primo elemento. Questa variabile-puntatore è l'unica[1] definita staticamente nel programma, ed è inizializzata a `NULL` per ricordare che all'inizio non punta, concettualmente, da nessuna parte.
Successivamente, quando sopraggiunga la necessità di inserire un elemento, occorrerà procurarsi tramite malloc un'area ampia a sufficienza da ospitare un nodo, e porne l'indirizzo nella radice. Quindi, il campo "valore" nodo verrà riempito col valore dell'elemento da inserire, mentre il puntatore al successivo verrà posto a `NULL` per indicare che non vi è, al momento, alcun nodo successore.
Gli inserimenti successivi agiranno in modo simile: l'esatto modo di procedere dipenderà dal tipo di inserimento desiderato (in testa alla lista, in coda, in una posizione intermedia, etc).

[1] Relativamente alla lista, ovviamente.

Scegliendo come modus operandi l'inserimento dei nuovi elementi *in testa* alla lista esistente, a ogni inserimento si farà puntare il nuovo nodo al'inizio della lista esistente, ponendo quindi nella radice l'indirizzo di questo nuovo nodo. Il programma che segue realizza tutto questo.

```
#include <stdio.h>
#include <stdlib.h>  /* contiene prototipi di malloc e free */

typedef struct list_element {
           int  value;
           struct list_element  *next;
         } item;

typedef item*  list;

list insert(int e, list l)
{
      list t;

      t=(list)malloc(sizeof(item));
      t->value=e;
      t->next=l;
      return t;
}

void main(void)
{
      list root=NULL, l;
      int i;

      do {
            printf("\nImmettere valore:\t");
            scanf("%d", &i);
            root = insert(i, root);
      } while (i!=0);

      l=root;
      while (l!=NULL)
      {
            printf("\nValore estratto:\t%d", l->value);
            l=l->next;
      }
}
```

Il tipo `item` costituisce il generico elemento della lista, ed è una struttura fatta di due parti: un campo valore (qui intero, ma che potrebbe essere anche estremamente complesso, come un vettore o un'altra struttura) e un campo per puntare al successivo `item`.
Il tipo `list` introduce un'abbreviazione per `item*` assai utile sia dal punto di vista pratico sia, soprattutto, dal punto di vista concettuale, in quanto ragionare in termini di liste è senz'altro più chiaro che operare in termini di "puntatori a item".
La funzione `insert` è il cuore di tutto: alloca memoria per un nuovo elemento, ne riempie il campo `value` col valore da mettere in lista, e inizializza il campo `next` in modo da puntare alla lista

passatale come secondo argomento, realizzando con ciò, di fatto, un *inserimento in testa* alla lista del nuovo elemento.
Perciò, il `main` si limita ad acquisire una serie di interi *lunga a piacere, senza vincoli di sorta* e a introdurli uno dopo l'altro nella lista (di radice) `root`. Successivamente, utilizzando il puntatore ausiliario `l`, un secondo ciclo scorre la lista dall'inizio alla fine, stampando nell'ordine i singoli elementi (`l->value`) e spostando ogni volta il puntatore in modo da posizionarlo sull'item successivo (`l=l->next`).

ESERCIZIO n° 24
Scrivere un modulo più generale del precedente, in grado di trattare liste di numeri interi, che metta a disposizione operatori (funzioni) per:
- inizializzare la lista a vuota;

- azzerare la lista, eliminando tutti gli elementi eventualmente presenti;

- verificare se la lista è vuota;

- calcolare il numero di elementi presenti nella lista;

- inserire un nuovo elemento in testa alla lista;

- inserire un nuovo elemento in coda alla lista;

- inserire un nuovo elemento in testa alla lista solo se esso non vi compare già;

- sopprimere l'elemento in testa alla lista;

- sopprimere l'elemento in coda alla lista;

- sopprimere la prima occorrenza di un elemento nella lista.

Soluzione
Per comodità, struttureremo il programma su più file. Cominciamo perciò col definire il contenuto del file di header, che chiameremo "`list.h`"e che sarà destinato a contenere le definizioni delle costanti usate (`NULL`, `TRUE`, `FALSE`) e delle funzioni esterne visibili a tutti (nel seguito denominate funzioni "pubbliche").
Tale header potrà presentarsi così:

```
#define  NULL   0
#define  TRUE   1
#define  FALSE  0

void  Init( ),                Reset( );
void  Insert_First( int i ),  Insert_Last( int i ),  Insert( int i );
int   Delete_First( ),        Delete_Last( ),        Delete( int i );
int   IsIn( int i ),          IsEmpty( ),            Length( );
```

Il secondo file sarà quello che implementerà tutte le funzioni di lista: lo chiameremo "`list.c`".
La struttura dati di base è sempre il *nodo*, costituito dai soliti due membri. Sebbene a rigore sia necessario allocare staticamente soltanto la radice, per comodità allocheremo anche un secondo puntatore all'ultimo elemento della lista, onde facilitare le operazioni che avvengono in coda alla

lista stessa (senza doverla ripercorrere ogni volta). Entrambi questi due puntatori devono essere *invisibili* esternamente al file "`list.c`", perché si tratta di dettagli implementativi, e `static` in quanto devono essere allocati in modo permanente.
Una possibile implementazione è quindi la seguente:

```
#include  <stdio.h>
#include  <alloc.h>
#define   NULL  0

struct node                     /* elemento (nodo) della lista */
{
    int item;                   /* valore memorizzato nel nodo */
    struct node *next;          /* puntatore al nodo successivo */
};

static struct node *first, *last;
                                /* puntatori a inizio e fine lista */

void Init( )                    /* inizializza la lista */
{
    first = last = NULL;
}

void Reset( )                   /* riazzera e svuota la lista */
{
    int i;
    while(!IsEmpty( ))  i = Delete_Last( );
}

int IsIn( int i )         /* controlla se un elemento è nella lista */
{
    struct node *t = first;
    while( t != NULL  &&  t->item != i )  t = t -> next;
    return( t != NULL );
}

int IsEmpty( )                  /* verifica se la lista è vuota */
{
    return( first == NULL );
}

int Length( )                   /* conta gli elementi della lista    */
{
    int count = 0;
    struct node *t = first;
    while ( t != NULL )  { count++; t = t -> next; }
    return count;
}

void Insert_First( int i )         /*  aggiunge  un  item  in  testa  alla
lista */
{
    struct node *newnode;
    newnode = (struct node *) malloc( sizeof( struct node ) );
    newnode -> next = first;
    newnode -> item = i;
```

```
    if (first == NULL )  last = newnode;
    first = newnode;
}

void Insert_Last( int i )    /* aggiunge un item in coda alla lista */
{
    struct node *newnode;
    newnode = (struct node *) malloc( sizeof( struct node ) );
    newnode -> next = NULL;
    newnode -> item = i;
    if ( first == NULL )  { first = newnode;  last = newnode; }
     else { last -> next = newnode;  last = newnode; }
}

void Insert( int i )      /* aggiunge un item in testa alla lista */
{                         /* solo se esso non vi compare già       */
    struct node *t = first;
    while ( t != NULL  &&  t -> item != i )   t = t -> next;
    if ( t == NULL )  Insert_First( i );
}

static int Destroy( struct node *t ) /* funzione PRIVATA (static) */
{                                    /* usata dalle varie Delete     */
    int value = t -> item;
    free( t );
    return value;
}

int Delete_First( )           /* elimina il primo elemento della    */
{                             /* lista, restituendone il valore     */
    struct node *t = first;
    if ( first == NULL )  return( NULL );
     else{
               if ( first == last )  first = last = NULL;
                else  first = t -> next;
               return Destroy( t );
          }
}

int Delete_Last( )            /* elimina l'ultimo elemento della */
{                             /* lista, restituendone il valore     */
    struct node *old,
               *new, *t = last;
    if ( first == NULL )  return( NULL );
     else{
               if ( first == last )  first = last = NULL;
                else
               {
                     old = first;
                     new = old -> next;
                     while ( new != last )
                           { old = new;  new = new -> next; }
                     last = old;  old -> next = NULL;
               }
               return Destroy( t );
          }
}
```

```
int Delete( int i )
{
    if ( first == NULL )  return( NULL );
     else {
               if ( first -> item == i )  return Delete_First( );
                else {
                          struct node *old, *new;
                          old = first; new = first -> next;
                          while ( new != NULL  &&  new -> item != i )
                               { old = new; new = new -> next; }
                          if ( new != NULL )
                                    {
                                      if (new == last ) last = old;
                                      old -> next = new -> next;
                                      return Destroy( new );
                                    }
                           else return NULL;
                     }
          }
}
```

La funzione `Init` inizializza unicamente i puntatori `first` e `last` a `NULL`, a indicare che la lista è vuota. La funzione `Reset` invece esegue ripetutamente l'operazione di `Delete_Last` sulla lista, finché essa non diviene vuota.
La funzione `IsIn` scandisce la lista finché o non trova l'elemento richiesto, o la lista stessa termina: il valore restituito è il risultato dell'espressione logica (`t!=NULL`), in quanto, effettivamente, l'elemento non è presente se si è giunti al termine della lista (`t==NULL`), mentre è presente in caso contrario.
Tralasciando la funzione `IsEmpty`, che è assolutamente ovvia, e la funzione `Length`, che si limita a scandire da cima a fondo la lista incrementando a ogni passo un contatore, passiamo a esaminare la funzione `Insert_First`, che agisce nel modo seguente.
In primo luogo, si crea un nuovo nodo, utilizzando per referenziarlo il puntatore `newnode`; quindi, si provvede a porre nel suo campo `item` il valore da inserire, inizializzando il campo `next` a `first`: ciò equivale a porre, logicamente, il nuovo nodo in testa alla lista attualmente esistente. Infine, si aggiorna `first` rendendolo uguale a `newnode`, in modo che punti al nuovo (primo) elemento. Trattandosi di un inserimento in testa, il puntatore `last` di norma non viene alterato, salvo il caso che la lista sia vuota: in tal caso, infatti, primo e ultimo elemento vengono a coincidere entrambi col nuovo nodo, e perciò anche `last` (che valeva in tal caso `NULL`) dev'essere fatto puntare al nuovo (e unico) elemento.
La logica della funzione `Insert_Last` è sostanzialmente duale: qui il nuovo nodo viene inizializzato in modo da puntare a `NULL`, perché sarà l'ultimo della lista, e il solo puntatore toccato è, di norma, `last`: solo nel caso che la lista sia vuota occorre, analogamente a sopra, inizializzare anche `first`, facendolo puntare al nuovo elemento. Da notare l'aggiornamento del puntatore `last`, che è un poco più complesso del precedente: per agganciare correttamente il nuovo nodo in coda alla lista, infatti, è necessario *prima* inserire nell'ex-ultimo elemento il puntatore al nuovo, e *solo dopo* spostare `last`; diversamente, aggiornando `last` immediatamente, diverrebbe impossibile agganciare il nuovo nodo all'ex-ultimo della lista, in quanto il relativo puntatore sarebbe andato perso.
La funzione `Insert` non è altro che un caso particolare di `Insert_First`, che prima di agire verifica se l'elemento da inserire è già presente, nel qual caso non fa nulla. In pratica, evita di

inserire dei doppioni, al prezzo di dover ogni volta scandire l'intera lista per sincerarsi della presenza o assenza del valore da inserire.
La funzione `Destroy` è una funzione locale al file "`list.c`", invisibile al suo esterno: per questo è dichiarata `static`. Si tratta di una funzione di utilità, usata dalle varie `Delete` per effettuare la deallocazione fisica della memoria: in effetti, `Destroy` è il solo posto in cui compare una chiamata alla primitiva di sistema `free` (duale della `malloc`). Prima di distruggere il nodo indicato dal puntatore passatole come argomento, `Destroy` ne recupera il valore, e lo restituisce come valore di ritorno: questo viene fatto per generalità, nell'ipotesi che tale valore possa servire in certi casi come riscontro (vedere `Delete` più oltre).
La funzione `Delete_First`, che restituisce `NULL` senza fare nulla se la lista è vuota, aggiorna `first` (ed eventualmente `last`) in modo da escludere dalla lista il primo elemento, il cui puntatore viene tuttavia mantenuto nella variabile temporanea `t`: successivamente, si chiama la funzione `Destroy` per la distruzione fisica del nodo così isolato.
La funzione `Delete_Last` agisce in modo concettualmente simile: tuttavia, la situazione è complicata dalla necessità di scandire la lista con due puntatori (denominati `old` e `new`), per recuperare il puntatore al penultimo nodo, che è quello da modificare per isolare ed escludere dalla lista l'ultimo elemento. Tale puntatore al penultimo nodo viene ottenuto scandendo la lista con due puntatori "affiancati" (uno dietro l'altro), fermandosi quando il più avanzato diventa uguale a `last`: in quel momento, infatti, il secondo puntatore punta evidentemente proprio al (penultimo) nodo richiesto. Infine, una chiamata alla `Destroy` sopprime fisicamente il nodo ormai isolato.
Di nuovo, la `Delete` è un caso particolare della `Delete_First`: se l'elemento da distruggere è il primo si richiama direttamente quest'ultima funzione, altrimenti si attiva una scansione della lista (ciclo `while`) che termina o quando termina la lista, o quando l'elemento richiesto viene trovato. Nel primo caso, il puntatore "avanzato" (`new`) vale `NULL`, non si fa nulla e si termina restituendo `NULL` al chiamante; altrimenti, si procede all'isolamento del nodo da sopprimere e quindi alla sua effettiva soppressione, tramite `Destroy`: il valore distrutto costituisce altresì il valore di ritorno di `Delete`, che offre così un modo semplice per verificare effettivamente l'avvenuta distruzione. La fase di isolamento deve distinguere fra due casi, quello in cui l'elemento sia stato trovato (ovvero, `new` punti) all'ultimo posto della lista (nel qual caso basta aggiornare `last`, facendolo puntare a `old`, che rappresenta il penultimo elemento della lista), e quello in cui invece esso sia stato trovato in una posizione intermedia nella lista: se così è, occorre "saltare" logicamente il nodo da sopprimere, facendo puntare il precedente al successivo di questo (ad esempio, se l'elemento è stato trovato al 14° posto, il 13° nodo deve essere fatto puntare al 15°, onde "saltare" il 14°, che subito dopo sarà soppresso), con l'istruzione `old -> next = new -> next;`.

Molte altre funzioni sarebbero possibili: inserimento ordinato, inserimento di un valore in una specifica posizione, cancellazione di tutte le occorrenze di un dato elemento (`Delete` si ferma alla prima che trova), eccetera. Si consiglia di riflettere su come tali nuove funzionalità potrebbero essere incluse.

ESERCIZIO n° 25

Scrivere un modulo C che implementi l'astrazione lista attraverso operatori (funzioni) ricorsive. In particolare devono essere disponibili le funzioni:
- `CONS` per costruire, data una lista e un elemento, una nuova lista avente l'elemento dato in testa;
- `HEAD` per selezionare, data una lista, l'elemento in testa;

- TAIL per selezionare, data una lista, la cosiddetta "coda" della stessa, ovvero la lista stessa privata del primo elemento (operazione duale delle precedente);
- WRITEL per stampare gli elementi di una lista data;
- LENGTH per calcolare il numero di elementi di una lista data;
- SUM per sommare i valori di tutti gli elementi di una lista data;
- MEMBER per determinare se un certo elemento appartiene o meno a una lista data;
- APPEND per costruire, dati un elemento e una lista, una nuova lista uguale alla precedente ma avente il nuovo elemento in coda;
- APPENDA per aggiungere in coda a una prima lista un elemento (alterando la prima);
- INSERT per inserire un elemento in una lista, sempre che lo stesso non sia già presente, mantenendo l'ordinamento crescente degli elementi della lista stessa.

Soluzione

A differenza dell'esercizio precedente, presenteremo stavolta il codice in un unico file: la divisione in parte header e parte implementativa è lasciata al lettore.

Per comodità definiamo il tipo enumerativo BOOLEAN come {FALSE, TRUE}, la costante NULL, e il tipo strutturato ELEMENT: quest'ultima definizione di tipo serve per semplificare la scrittura (a vantaggio della leggibilità) nelle funzioni successive. Definiamo infine il tipo LIST come puntatore a ELEMENT, e la costante EMPTYLIST come sinonimo di NULL. Rimandando i commenti, presentiamo direttamente il codice delle funzioni (primitive) di base.

```
#include <stdio.h>
#include <alloc.h>
#define  NULL      0
#define  EMPTYLIST NULL

typedef enum { FALSE, TRUE } BOOLEAN;

typedef struct ELEMENT
{     int item;
      struct ELEMENT *next;
} ELEMENT;

typedef (ELEMENT *) LIST;

LIST CONS(int value, LIST root)
{
      LIST punt;
      Punt = (LIST) malloc( sizeof(ELEMENT) );
      Punt -> item = value;
      Punt -> next = root;
      return Punt;
};

int  HEAD (LIST root)
{
      if ( root == EMPTYLIST ) return 0 ;
       else return (root -> item);
};

LIST TAIL (LIST root)
{
```

```
     if ( root == EMPTYLIST ) return EMPTYLIST ;
      else return (root -> next);
};

void WRITEL (LIST root)
{
     printf("Lista: ");
     while ( root != EMPTYLIST )
     {
          printf( "%d", root -> item );
          root = root -> item;
     };
     printf( "\n" );
};
```

Tutte le altre funzioni possono essere scritte in termini di queste, che perciò incapsulano completamente i dettagli relativi alla rappresentazione interna. Ogni modifica all'implementazione di basso livello (ad esempio, un tipo `LIST` basato su array) rimarrà perciò confinata entro queste funzioni. In verità, la stessa `WRITEL` potrebbe essere scritta in funzione delle precedenti, limitando ancora di più l'insieme delle primitive di base alle sole `CONS`, `HEAD`, `TAIL`. Si è preferito in questo caso fornire la funzione come primitiva (realizzandola quindi sfruttando la conoscenza della rappresentazione interna) in considerazione della frequenza con cui sarà utilizzata, che fa preferire l'efficienza alla indipendenza e alla leggibilità.

```
int LENGTH (LIST root)
{
     if ( root == EMPTYLIST ) return 0;
      else return ( 1 + LENGTH (TAIL(root)) );
};

int SUM (LIST root)
{
     if ( root == EMPTYLIST ) return 0;
      else return ( HEAD(root) + SUM (TAIL(root)) );
};

enum BOOLEAN MEMBER (int value, LIST root)
{
     if ( root != EMPTYLIST )
     {
          if (value == HEAD(root) return TRUE;
           else return MEMBER( value, TAIL(root) );
     };
     return FALSE;
};

LIST APPEND( int value, LIST root)
{
     if ( root == EMPTYLIST )  return CONS( value, EMPTYLIST);
      else return CONS( HEAD(root), APPEND(value, TAIL(root)) );
};

void APPENDA( int value, LIST *rootPtr)
{
     if ( *rootPtr== EMPTYLIST )   *rootPtr = CONS( value, EMPTYLIST);
```

```
      else APPENDA( value, &(TAIL( *rootPtr )) );
};

LIST INSERT( int value, LIST root)
{
     if ( root == EMPTYLIST )  return CONS( value, EMPTYLIST);
      else if ( ! MEMBER(el, root) )
           if (HEAD(root) > value)  return CONS( el, root );
            else return CONS( HEAD(root), INSERT(value, TAIL(root)) );

};
```

Analizziamo la struttura di queste funzioni, molto interessanti in quanto mostrano come esprimere le soluzioni ai problemi in termini di se stesse. La funzione `LENGTH` è un primo esempio: se la lista è vuota la lunghezza è zero, altrimenti vale 1 più la lunghezza della restante parte della lista, quella che abbiamo chiamato coda (`TAIL`). Analogamente, nel caso della funzione `SUM`, la somma degli elementi vale zero se la lista è vuota, altrimenti è pari alla somma del valore del primo elemento (la testa, `HEAD`) e di tutti i successivi, che può essere calcolata mediante una chiamata ricorsiva alla funzione `SUM` stessa.

Quanto alla `MEMBER`, premesso che il risultato è certamente falso se la lista è vuota, l'algoritmo si può esprimere dicendo che l'elemento appartiene alla lista se coincide con la testa della lista stessa, oppure se appartiene alla sottolista costituita dalla coda; di nuovo, questo secondo caso implica una chiamata ricorsiva a `MEMBER` medesima per analizzare la restante parte della lista.

Due parole in più le merita la funzione `APPEND`, che, nonostante l'apparenza semplice, è piuttosto sofisticata. La logica su cui si basa è questa: se la lista d'ingresso è vuota, il risultato deve essere una nuova lista costituita dal solo nuovo elemento, che si può costruire partendo dalla primitiva `CONS` con secondo parametro `EMPTYLIST`; altrimenti, l'elemento dato deve essere appeso alla restante parte della lista, cioè alla coda. Questo significa che occorre costruire una nuova lista avente lo stesso primo elemento (`HEAD`) di quella data, e comprendente il nuovo elemento nella propria coda. Quest'ultima operazione richiede, di fatto, l'esecuzione della medesima funzione `APPEND` sulla sottolista costituita dalla coda della lista data.
E' opportuno sottolineare che la funzione `CONS` crea sempre nuovi elementi: quindi, la lista risultante è una copia di quella data (con in più, appeso in fondo, il nuovo elemento), che, viceversa, resta immodificata.

Dunque, `APPEND` effettua il proprio compito senza provocare *effetti collaterali* (*side effects*) sulla lista originale, e in questo sta proprio la differenza con la successiva funzione `APPENDA`, che, pur operando concettualmente nello stesso modo, agisce invece direttamente sulla lista passatale (per indirizzo!) come secondo argomento, modificando quindi in modo definitivo e irrevocabile la lista originale.

Per finire, due parole sulle `INSERT`. Anche qui, se la lista è vuota basta crearne una nuova costituita di un solo elemento (quello da inserire), mentre se così non è occorre distinguere il caso in cui l'elemento sia già presente nella lista (nel quale non si fa nulla, dato che per ipotesi non si vogliono doppioni) dal caso opposto, che è l'unico realmente interessante.
In questo caso, poiché si desidera effettuare un inserimento ordinato, si confronta in primo luogo l'elemento da inserire col primo della lista: se il nuovo elemento è minore, basta anteporlo alla lista

preesistente tramite una `CONS`, altrimenti occorre inserire l'elemento nella coda, mediante una chiamata ricorsiva alla `INSERT` stessa.
Anche qui, in modo del tutto analogo alla già vista `APPEND`, l'uso della funzione `CONS` nella chiamata ricorsiva garantisce la replica della parte iniziale di lista e la conseguente assenza di side effects sulla lista originale.
Ad esempio, se la lista originale è costituita dai valori [3, 5, 8, 10], l'inserimento dell'elemento 7 dà luogo a una nuova lista, fatta dai valori [3, 5, 7, 8, 10], in cui gli ultimi due elementi sono condivisi con l'originale, mentre i primi due (3 e 5) sono replicati.
Data l'importanza della ricorsione in moltissimi settori applicativi, si consiglia di riflettere attentamente su questo meccanismo, simulando il funzionamento della `INSERT` e della `APPEND`, fino a comprenderlo perfettamente.

3. Stack

ESERCIZIO n° 26
Scrivere un modulo C che implementi l'astrazione stack.

Soluzione
Per operare in un caso concreto, supporremo di dover realizzare uno stack di numeri, di tipo *double*: l'adattamento ad altri tipi è, peraltro, immediato. Per implementare lo stack, utilizziamo un vettore, che chiameremo `val`, con un indice (referenziante la prima posizione libera) di nome `sp` (Stack Pointer). Naturalmente, con queste scelte la nostra realizzazione è soltanto un'approssimazione del concetto di stack, in quanto, mentre teoricamente non vi sono limiti al numero di elementi inseribili, nel nostro caso non potranno essere introdotti più di `MAXDIM` elementi, corrispondentemente alla dimensione massima del vettore usato come supporto.
Per questo, oltre alle classiche funzioni di inserimento ed estrazione di un elemento, decidiamo di fornire anche due funzione di utilità, denominate `isFull()` e `isEmpty()`, che restituiscano un valore (intero) di tipo vero/falso a seconda che lo stack sia, rispettivamente, pieno o vuoto.
Al fine di garantire la consistenza della struttura dati, provvediamo a dichiarare `static` le due variabili globali, in modo da renderle invisibili fuori dal file corrente: l'accesso alla struttura dovrà così obbligatoriamente avvenire attraverso le due funzioni predisposte, `push()` e `pop()`, mentre lo stato della struttura sarà accessibile tramite le due funzioni di utilità sopra citate.
A proposito di `pop()`, occorre stabilire cosa accade nel caso che venga invocata quando lo stack è vuoto: mentre nel caso della `push()`, infatti, può bastare un messaggio d'errore, perché tale funzione è di tipo void e non restituisce nulla, nel caso della `pop()` occorre decidere se restituire qualcosa e, se sì, cosa. Per semplicità, in questa sede si è scelto di restituire un valore qualunque, -333.333: in una realizzazione reale, viceversa, sarebbe opportuno predisporre una soluzione più sofisticata, quale, ad esempio, l'introduzione di un'ulteriore variabile globale con funzione di flag, da settare quando qualcosa va storto, e leggibile tramite un'apposita funzione `stack_result()`. Questa modifica è lasciata per esercizio.

```
     /*    File STACK.C:    implementa l'astrazione stack        */

#include <stdio.h>
#define MAXDIM 100
```

```
static int      sp=0;           /* variabili esterne, permanenti e   */
static double   val[MAXDIM];    /* invisibili, che costituiscono le  */
                                /* strutture dati                    */

void push(double f)
{
     if (sp<MAXDIM)
          val[sp++]=f;
     else
          printf("\nErrore: stack pieno; %g non inseribile\n",f);
}

double pop(void)
{
     if (sp>0)
          return val[--sp];
     else {
          printf("\nErrore: stack vuoto\n");
          return -333.333;
          }
}

int isFull(void)
{
     return (sp==MAXDIM);
}

int isEmpty(void)
{
     return (sp==0);
}
```

Osserviamo che i test all'interno di `push()` e `pop()` corrispondono esattamente a chiedersi se lo stack sia, rispettivamente, pieno (nel qual caso è impossibile la push) o vuoto (nel qual caso è impossibile la pop): poiché abbiamo predisposto due funzioni che realizzano esattamente tali test, il codice di push e pop potrebbe anche essere riscritto, forse in nodo più leggibile, come segue.

```
          /*       variante di push() e pop()       */

void push(double f)
{
     if (!isFull())
          val[sp++]=f;
     else
          printf("\nErrore: stack pieno; %g non inseribile\n",f);
}

double pop(void)
{
     if (!isEmpty())
          return val[--sp];
     else {
          printf("\nErrore: stack vuoto\n");
          return -333.333;
```

```
        }
}
```

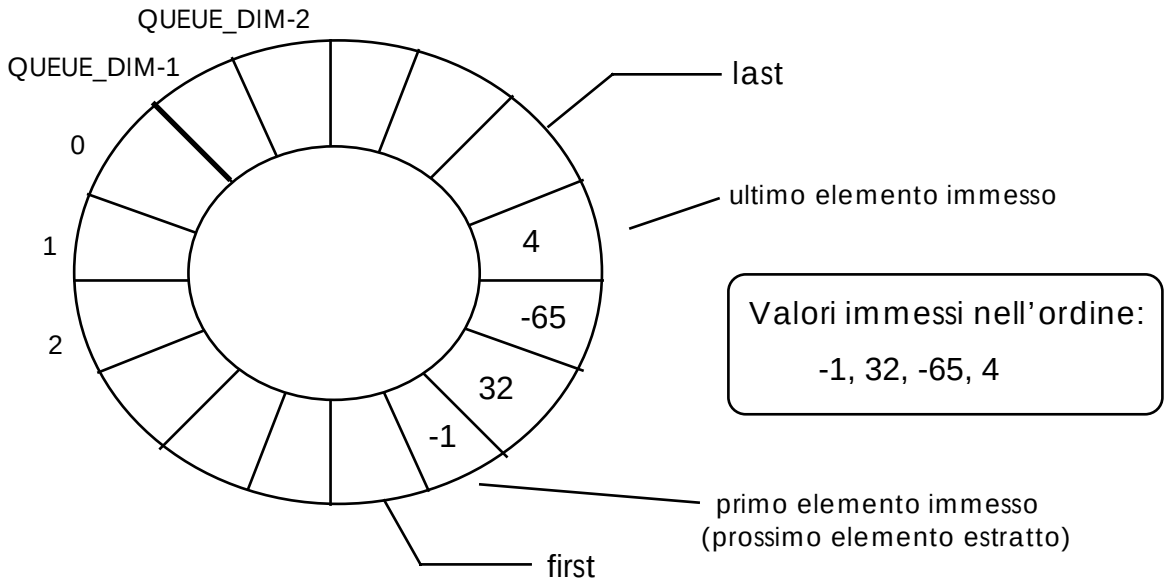
Per provare questo modulo, approntiamo un piccolo programma di prova, che consente di immettere valori (introducendo 0.0 per terminare) e successivamente li estrae, visualizzandoli. Il file "stack.h" riunisce le *dichiarazioni* delle quattro funzioni esportate (cioè rese visibili all'esterno) da "stack.c".

```
                /*    UN PROGRAMMA DI PROVA */

#include <stdio.h>
#include "stack.h"      /* include le dichiarazioni delle funzioni */

void main(void)
{
 double f;

 do {
  printf("\nValore:\t");
  scanf("%lf",&f);
  push(f);
 } while(f!=0.0);

 while (!isEmpty())
  printf("\nValore:\t%f", pop() );

 printf("\nLa struttura dati è vuota.\n");
}
```

4. Coda circolare

ESERCIZIO n° 27
Scrivere un modulo C che implementi l'astrazione coda circolare.

Soluzione
Come nel caso precedente, assumiamo di operare con numeri, di tipo *double*. Per implementare la coda, utilizziamo ancora un vettore, che chiameremo `val`, gestito però, stavolta, con *due indici*, di nome `last` e `first`, referenzianti rispettivamente la fine dalla coda (più esattamente, la prima posizione libera in fondo alla coda, dove verrà inserito il prossimo elemento) e l'inizio della stessa (più precisamente, la prima posizione davanti all'elemento da estrarre). La figura sottostante illustra la situazione su un esempio concreto, unitamente alla numerazione delle celle del vettore (da 0 a `QUEUE_DIM`-1)



La realizzazione proposta è limitata a un massimo di `QUEUE_LENGTH` elementi, corrispondenti alla dimensione massima del vettore usato come supporto (`QUEUE_DIM`) *diminuita di uno*: questo fatto è una diretta conseguenza della scelta di denotare con `last` e `first` le due celle limitrofe alle coda vera e propria. Se da un lato, infatti, ciò implica che almeno una cella resterà sempre vuota, dall'altro consente di scrivere le funzioni di inserimento ed estrazione in modo più semplice, ed è stato perciò preferito in questa sede. Inoltre, forniremo le usuali funzioni di utilità `isFull()` e `isEmpty()`, con il solito significato. Sempre per garantire la consistenza della struttura dati, dichiariamo `static` le variabili globali, in modo da renderle invisibili fuori dal file corrente e assicurare che l'accesso alla struttura avvenga soltanto attraverso le funzioni predisposte, che per ragioni di uniformità continuiamo a chiamare `push()` e `pop()`, sebbene non si tratti più di uno stack.

Dal momento che la coda va gestita circolarmente, tutti gli incrementi e i decrementi degli indici devono avvenire *modulo `QUEUE_DIM`*: ciò implica controllare, a ogni operazione, se si è "a cavallo" della fine del vettore (denotata in figura da una riga più marcata), provvedendo in tal caso a correggere i calcoli appunto di una quantità `QUEUE_DIM`.
Per questa ragione, le espressioni di verifica se la coda è piena (in `isFull`) o vuota (in `isEmpty`) contengono delle espressioni condizionali, che effettuano il calcolo della differenza fra gli indici o in modo "normale" o con la correzione suddetta a seconda della situazione attuale. In ogni caso, al di là dei tecnicismi usati nelle formule, concettualmente `isFull` è vera se la differenza fra gli indici (eventualmente corretta come sopra) raggiunge `QUEUE_DIM`, mentre `isEmpty` è vera se tale differenza si riduce a 1, come si può constatare da una lettura attenta del codice.
In conseguenza dell'organizzazione adottata, l'inserzione di un nuovo elemento (push) richiede che *prima* si riempia la cella corrente (indicata da `last`), e *poi* si incrementi l'indice, mentre, dualmente, l'estrazione (pop) richiede che *prima* si incrementi l'indice `first` e *poi* si estragga l'elemento da esso indicato. Ciò si riflette immediatamente nel tipo di incremento adottato, che è un *post-incremento* (`last++`) nel primo caso, e un *pre-incremento* (`++first`) nel secondo. Per comprendere bene il codice di `pop`, tenere presente che la condizione dell'`if` viene valutata (e quindi `first` incrementato) sia che poi la stessa risulti vera, sia che risulti falsa.

```
/* File QUEUE.C:   implementa l'astrazione coda circolare  */
```

```
#include <stdio.h>
#define QUEUE_DIM 100
#define QUEUE_LENGTH QUEUE_DIM-1

static int last=0;              /* variabili globali che gestiscono  */
static int first=QUEUE_LENGTH;  /* la struttura dati (invisibili)    */
static double val[QUEUE_DIM];

int isFull(void)
{
   return (last>first ? last-first : QUEUE_DIM+last-first)>=QUEUE_DIM;
}

int isEmpty(void)
{
   return (last>first ? last-first : QUEUE_DIM+last-first) <= 1;
}

void push(double el)
{
     if (!isFull())
       {
          val[last++]=el;
          if(last>=QUEUE_DIM) last-=QUEUE_DIM;
       }
     else
          printf("\nErrore: coda piena. %lg non inseribile\n",el);
}

double pop(void)
{
     if (!isEmpty())
       {
          if (++first>=QUEUE_DIM) first-=QUEUE_DIM;
          return val[first];
       }
     else
       {
          printf("\nErrore: coda vuota.\n");
          return -555.555;
       }
}
```

Avendo mantenuto identici i nomi delle funzioni, come programma di prova si può riusare, *senza nessuna modifica*, quello dell'esercizio precedente: solo, in luogo dello header "stack.h" andrà posto il nuovo header "queue.h", che, peraltro, risulta identico all'altro.
Per usare l'una o l'altra struttura dati basterà quindi sostituire (in Turbo C) il file "stack.c" con "queue.c" nella descrizione del progetto, e rifare il *make* del tutto. Lo studente attento potrà notare che il programma in sé *non* sarà ricompilato, ma soltanto collegato (dal linker) alle nuove funzioni definite in "queue.c", questo, sì, compilato sul momento.

Questo esercizio, unitamente al precedente, mostra in tutta evidenza l'estrema utilità di *modularizzare i problemi*, *incapsulando* i dettagli delle soluzioni e delle realizzazioni dei singoli

sottoproblemi, in modo da ridurre al minimo i punti di contatto fra essi, ottenendo in cambio grande *flessibilità* e *facilità di adottare nuovi comportamenti* al mutare delle situazioni.
Come risultato, immettendo gli stessi elementi rispettivamente nello stack o nella coda, con lo stesso programma principale, essi saranno restituiti e visualizzati in un diverso ordine, corrispondente alla diversa *politica di accesso* realizzata dai due tipi di dato astratto (LIFO nel caso dello stack, FIFO nel caso della coda).