FONDAMENTI DI INFORMATICA II

ESERCITAZIONE n° 2

IL LINGUAGGIO C

ESERCIZIO n° 1
Definire una macro `ODD(i)` che restituisca `TRUE` o `FALSE` a seconda che il valore intero passato sia rispettivamente dispari o pari.

Soluzione
Poiché un intero è dispari se ha il bit meno significativo a 1, ed è pari in caso contrario, una possibile definizione può essere la seguente:

```
#define ODD(i)   ((i) & 1)
```

ESERCIZIO n° 2
Definire una macro `CPL2(i)` che restituisca il complemento a 2 del valore passatole come argomento.

Soluzione
Poiché il complemento a 2 si ottiene sommando 1 al complemento a 1 del numero dato, si ha semplicemente:

```
#define CPL2(a)     (1 + (~a))
```

ESERCIZIO n° 3
Definire una macro `AND(a,b)` che restituisca il risultato dell'operazione di AND logico fra i due parametri `a` e `b`.

Soluzione
Ricordando che nell'operazione di AND il risultato è certamente zero se uno dei due operandi è falso, mentre è uguale all'altro operando in caso contrario, si può scrivere:

```
#define AND(a,b)    ((a) ? (b) : 0)
```

ESERCIZIO n° 4
Definire una macro `DEMORGAN_AND(a,b)` che restituisca il risultato dell'operazione di AND logico fra i due parametri effettuando il calcolo sulla base delle identità del Teorema di De Morgan.

Soluzione
Poiché è noto che:

$$A \wedge B = \neg ((\neg A) \vee (\neg B))$$

si ha immediatamente:

```
#define DEMORGAN_AND( a, b ) ( ~( (~a) | (~b) ) )
```

ESERCIZIO n° 5

Definire una macro `GENERIC_SWAP( NAME, ELEM_TYPE )` che definisca una funzione di nome `NAME` che effettui lo scambio (swap) fra due elementi di tipo `ELEM_TYPE`: `NAME` deve ricevere come parametri i puntatori ai due elementi da scambiare. In altri termini:

`GENERIC_SWAP(int_swap, int)`	dovrebbe definire una funzione `int_swap` capace di scambiare due interi
`GENERIC_SWAP(float_swap, float)`	dovrebbe definire una funzione `float_swap` capace di scambiare due valori di tipo float

Soluzione

Poiché questa macro è decisamente più complessa di quelle fin qui viste, è necessario fare una precisazione. Come regola generale, le macro devono essere scritte su un'unica riga: qualora sia indispensabile andare a capo, è necessario, per indicare che il testo continua nella riga successiva, porre al termine della riga stessa un backslash '\'. Chiarito ciò, una macro che effettua quanto richiesto può essere la seguente:

```
#define GENERIC_SWAP( NAME, ELEM_TYPE ) \
    void NAME( a, b ) \
    ELEM_TYPE  *a,  *b; \
    { ELEM_TYPE  t; \
      t = *a;  *a = *b; *b = t; }
```

Nel primo dei due casi indicati nel testo dell'esercizio (`GENERIC_SWAP(int_swap, int)`) il risultato sarebbe quindi una funzione `int_swap` così definita:

```
void int_swap ( a, b )
int *a,  *b;
{ int  t;
  t = *a;  *a = *b; *b = t; }
```

ESERCIZIO n° 6

Scrivere una funzione che, a ogni invocazione, restituisca il successivo numero primo.

Soluzione

Poiché il tempo di vita di una variabile locale di una funzione (variabile *automatica*) è quello della funzione stessa, a ogni nuova invocazione delle funzione, tutte le variabili locali sono concettualmente vuote: quelle per le quali è specificato un valore iniziale sono inizializzate a tale valore, le altre hanno un contenuto indefinito. Sebbene questo comportamento sia fortemente desiderabile e opportuno nella stragrande maggioranza dei casi, perché garantisce l'indipendenza di una invocazione da tutte quelle passate e future, esistono casi (come quello in esame) in cui, per il particolare compito svolto da una funzione, è utile che una o più variabili (a tutti gli effetti, *variabili*

di stato) mantengano il loro valore fra un'invocazione e l'altra della funzione stessa. Questa necessità insorge particolarmente quando sono richieste *funzioni di generazione* di sequenze di valori che non devono ripetersi e in cui, eventualmente, ogni valore può dipendere dai precedenti.
In questi casi, è utile la possibilità, offerta dal C, di *dichiarare* `static` *delle variabili interne a una funzione*. Il significato di questa dichiarazione, del tutto diverso da quello di un'analoga dichiarazione applicata a un oggetto esterno, è che *una variabile locale dichiarata* `static` *mantiene il suo valore fra una chiamata e l'altra* della stessa funzione, il che è esattamente quanto ci serve per tenere traccia dell'ultimo numero primo generato.

Costruiremo quindi una funzione `nextPrime`, senza parametri, che a sua volta si avvarrà di una funzione `isPrime` per controllare che un dato valore sia primo. Quest'ultimo controllo, esclusi i casi banali dei numeri 1 e 2, verrà attuato col metodo di Eratostene, verificando che il numero (dispari) dato non sia divisibile per alcun numero minore della sua radice quadrata. Il programma potrà allora presentarsi come segue:

```
#include <stdio.h>
#include <math.h>    /* Include la libreria matematica (sqrt) */

#define NO     0
#define YES    1

int isPrime(int n)     /* Ritorna 0 o 1 secondo se n è primo    */
{
     int i, max=sqrt( (double)n );

     if (n>0 && n<4) return YES;
     else if (!(n%2)) return NO;

     for(i=3; i<=max; i+=2)
      if (!(n%i)) return NO;

     return YES;
}

int nextPrime(void)
{
     static n=0;    /* variabile statica con inizializzazione */

     if (n>=0 && n<=2) return ++n;
      else {
               do
                n+=2;
               while(!isPrime(n));
               return n;
           }
}

void main()
{
     int p,max;
```

```
     do {
       printf("\nGenerare i numeri primi fino a:\t");
       scanf("%d",&max);
     } while(max<1);

     do {
      printf("\nNumero primo:\t%d", p=nextPrime() );
     } while(p<max);
}
```

L'unico elemento realmente innovativo è costituito dalla funzione `nextPrime`, che contiene al suo interno una dichiarazione di variabile statica il cui valore viene mantenuto fra una chiamata e l'altra. Una variabile statica è sempre inizializzata automaticamente dal sistema a zero, a meno che non sia espressamente indicato un diverso valore. Tale inizializzazione avviene una sola volta, concettualmente *prima* dell'inizio dell'esecuzione del programma. Perciò la variabile `n` viene azzerata (da codice generato ad hoc dal compilatore) ogni volta che il programma viene *rieseguito*, e *non a ogni invocazione* della funzione `nextPrime`, come invece avverrebbe per una normale variabile locale.

ESERCIZIO n° 7

Scrivere una funzione *ricorsiva* che sommi i primi *N* numeri naturali.

Soluzione

In C, una funzione può, se ciò è utile e necessario, richiamare se stessa, direttamente o indirettamente (cioè attraverso altre funzioni): può, cioè, essere *ricorsiva*. Dal punto di vista concettuale, non c'è alcuna differenza fra chiamare un'altra funzione o la stessa funzione: in entrambi i casi, infatti, vi è un *cliente* (main o altra funzione) che si rivolge a un *servitore* per l'espletamento di un particolare compito. Come caso particolare, una funzione può utilizzare se stessa come suo servitore, qualora ciò sia utile. Conseguentemente, anche il linguaggio tratta i due casi esattamente nello stesso modo. Tecnicamente, ciò è possibile perché ogni attivazione di una nuova funzione produce immediatamente la creazione (sul momento) del relativo *ambiente*, ossia di una zona con tutti i parametri e tutte le variabili locali della funzione stessa: ergo, ogni nuova attivazione di funzione (anche della stessa funzione) è indipendente da tutte le attivazioni precedenti.

Per affrontare un problema con approccio ricorsivo, bisogna innanzitutto modificare il proprio modo di pensare, cambiando metodo rispetto al "classico" approccio iterativo. Infatti, mentre quest'ultimo richiede di cogliere l'essenza del problema ed essere in grado di esprimere il passo generico che porta alla soluzione, l'approccio ricorsivo si limita a tentare di *abbassare il grado di difficoltà* del problema, esprimendone la soluzione in termini di alcuni passi elementari *e inoltre* della soluzione dello stesso problema, in un caso più semplice.
Nel caso in esame, anziché vedere il problema come accumulazione progressiva di valori con relativi risultati parziali, per evidenziare un *passo elementare di decomposizione* si può osservare che se $N=0$ la somma è banalmente zero, mentre se $N>0$ la somma vale N più la somma dei valori fino a $N-1$. Quest'ultima somma non è altro che la soluzione del medesimo problema (sommare un certo insieme di numeri), applicato però al caso (più semplice) di $N-1$ valori anziché a N valori.

Questo porta a scrivere direttamente il codice della soluzione che si presenta come segue.

```
#include <stdio.h>

int sum_to(unsigned int n)
{
     if (n==0) return 0;
      else return n+sum_to(n-1);
}

main()
{
     int n;

     printf("\nIntrodurre N:\t");
     scanf("%d",&n);

     printf("\nSomma fino a %d:\t%d\n", n, sum_to(n));
}
```

La funzione `sum_to` opera in modo concettualmente assai semplice: se *n* è zero, il risultato della somma è zero; altrimenti, esso è pari alla somma di tutti gli *n-1* numeri precedenti (il cui calcolo è delegato alla stessa `sum_to`), più *n* stesso. Ad esempio, per *n*=4 si ha:

```
sum_to(4)  = 4 + sum_to(3) =
           = 4 + ( 3 + sum_to(2) ) =
           = 4 + ( 3 + ( 2 + sum_to(1) ) ) =
           = 4 + ( 3 + ( 2 + (1 + sum_to(0) ) ) ) =
           = 4 + ( 3 + ( 2 + (1 + 0 ) ) ) =                    /* § */
           = 4 + ( 3 + ( 2 + 1 ) ) =
           = 4 + ( 3 + 3 ) =
           = 4 + 6 =
           = 10.
```

Concretamente, ogni "espansione" di `sum_to` corrisponde alla creazione di un ambiente per una nuova attivazione di questa funzione. Quindi, in corrispondenza del punto §, vi sono cinque attivazioni di `sum_to` contemporaneamente attive, ognuna in attesa che la successiva produca il valore necessario per concludere la rispettiva operazione. Da § in poi, avendo l'ultima chiamata di `sum_to` fornito direttamente un risultato (0), le varie invocazioni iniziano a chiudersi, e inizia a formarsi il risultato finale. Questa successione di chiamate ricorsive può essere seguita col debugger del Turbo C, tramite l'opzione *call stack* del menù *debug*.

ESERCIZIO n° 8

Scrivere una funzione ricorsiva che stampi una parola rovesciata..

Soluzione

Per stampare a rovescio (cioè dall'ultima lettera alla prima) una parola, si può partire dall'idea di stampare innnanzitutto l'ultimo carattere, e poi stampare rovesciati (se esistono) tutti gli altri

caratteri. Con questa semplice osservazione, si è espresso il problema di stampare rovesciata una parola lunga *n* tramite un'operazione elementare (la stampa del singolo carattere) e dello stesso problema in un caso più semplice (stampare rovesciati i precedenti *n*-1 caratteri).
La funzione richiesta può quindi agire esattamente in questo modo. Più precisamente, se la stringa passatale è lunga zero non fa nulla, se è lunga uno la stampa e termina, mentre se è lunga più di uno stampa l'ultimo carattere e richiama se stessa sulla sottostringa iniziale (privata, cioè, dell'ultimo carattere). Una possibile codifica può essere la seguente:

```
#include <stdio.h>
#include <string.h>
#define MAXLEN 30

void print_rev(char word[])
{
     int len=strlen(word);
     char w[MAXLEN];

     if (len<1) return;
     else if (len==1) putchar(word[0]);
      else {
               strncpy(w, word, len-1);
               w[len-1]='\0';
               putchar(word[len-1]);
               print_rev(w);
           }
     return;
}

main()
{
     int n;
     char parola[MAXLEN];

     printf("\nIntrodurre una parola:\t");
     scanf("%s",&parola);

     print_rev(parola);
}
```

ESERCIZIO n° 9

Realizzare una funzione che risolva ricorsivamente il problema della *Torre di Hanoi*.

La Torre di Hanoi

Si tratta di un antico gioco, in cui sono date tre torri (sinistra, centrale, e destra) e un certo numero di dischi forati. I dischi hanno diametro diverso gli uni dagli altri, e inizialmente sono infilati uno sull'altro (dal basso in alto) dal più grande al più piccolo sulla torre di sinistra. Scopo del gioco è portarli tutti sulla torre di destra, usando quella centrale come appoggio (torre di transito), e rispettando due regole:

a) si può muovere un solo disco alla volta;

b) un disco più grande non può mai stare sopra a un disco più piccolo.

Soluzione

Una soluzione generale, non ricorsiva, è molto difficile da dare, soprattutto quando il numero dei dischi supera poche unità (già 4 non è banale). La soluzione ricorsiva, invece, è immediata: per spostare *n* dischi dalla torre di sinistra alla torre di destra basta supporre di saper spostare gli *n-1* dischi superiori in quella centrale, muovere quello rimasto dalla torre di sinistra a quella di destra, e infine rispostare gli *n-1* superiori da quella centrale a quella di destra.

Ciò equivale a dire che il problema di grado *n* può essere ricondotto a una mossa elementare e a due problemi di grado *n-1*, che possono essere a loro volta risolti procedendo nello stesso modo. Ovviamente, a forza di decomporre il problema si arriva al punto di spostare un solo disco, che è una mossa elementare fattibile direttamente. Questa strategia, ancorché semplice, non è comoda da gestire a mano: viceversa è adattissima a un calcolatore. Il programma che la implementa è riportato qui di seguito.

```
#include <stdio.h>

typedef enum { SINISTRA, CENTRALE, DESTRA } torre;

void hanoi(int d, torre start, torre end, torre transit);

main()
{
 int d;

 printf("\nTORRE DI HANOI\n\n");
 printf("\nQuanti dischi?\t");
 scanf("%d", &d);
 printf("\n\nSOLUZIONE:\n");
 hanoi(d, SINISTRA, DESTRA, CENTRALE);
 return 0;
}

void hanoi(int d, torre iniziale, torre finale, torre ausiliaria)
{
 if (d==1)
 {
  printf("Muovi un disco dalla torre %d alla torre %d\n",
       iniziale, finale);
  return;
```

```
 }
 else
 {
  hanoi(d-1,iniziale, ausiliaria, finale);
  printf("Muovi un disco dalla torre %d alla torre %d\n",
       iniziale, finale);
  hanoi(d-1, ausiliaria, finale, iniziale);
  return;
 }
}
```

Osserviamo che per modellare la tre torri, si è introdotto, tramite la direttiva `typedef`, un nuovo tipo `torre`, che da questo momento è liberamente usabile al pari dei tipi predefiniti del C (int, char, etc). Quindi, sarà possibile, in particolare, definire e usare variabili di tipo `torre`.[1] Concretamente, essendo questo nuovo tipo definito come un'enumerazione, i tre identificatori `SINISTRA`, `CENTRALE` e `DESTRA` sono associati ai tre valori interi 0, 1 e 2: perciò l'output del programma si esprime in termini di "torre 0, torre 1, torre 2". Naturalmente sarebbe semplice prevedere forme di output più chiare e leggibili (ad esempio facendo stampare "sinistra" in luogo di "torre 0", etc.): questo miglioramento è lasciato al lettore. Piuttosto, interessa sottolineare che la funzione `hanoi` realizza esattamente la logica sopra riportata: infatti, nel caso di un solo disco lo muove direttamente, altrimenti prima ne sposta *n-1*, poi sposta quello rimasto, indi risposta nuovamente i precedenti *n-1*.
Le tre torri iniziale, finale e ausiliaria naturalmente variano da una chiamata all'altra, perché i vari sottoproblemi devono in generale spostare dei dischi fra due generiche torri (p. es. dalla torre centrale a quella di destra), per la qual operazione sfruttano come transito la terza torre (ad es. quella di sinistra).

ESERCIZIO n° 10
Realizzare una funzione che copi una stringa in un'altra tramite puntatori.

Soluzione
Il problema della copiatura di una stringa in un'altra è interessante perché evidenzia i limiti della "analogia" fra puntatori e vettori. Per questo, nel seguito sono presentate e commentate tre versioni: una errata, le altre due entrambe corrette ma con diversa semantica.

Una prima idea può essere quella di assegnare direttamente un vettore di caratteri a un'altro, come nell'esempio che segue:

```
#include <stdio.h>

main()
{
 char s1[80],s2[80];

 printf("\nStringa 1:\t");
 scanf("%s",s1);
```

[1] Maggiori dettagli sulla direttiva typedef si possono trovare nel commento all'esercizio n° 20.

```
 s2=s1;                  /* ATTENZIONE:  ERRATO!!      */

 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);

 printf("\nModifica stringa 1:\t");
 scanf("%s",s1);
 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);
}
```

Tuttavia, assegnare direttamente una stringa a un'altra (qui, s1 a s2) *non significa copiarla,* ma soltanto (tentare di) assegnare all'una l'indirizzo dell'altra (qui, l'indirizzo di s1 all'indirizzo di s2). Ciò è però errato, non essendo ovviamente possibile cambiare l'indirizzo a cui si trova un vettore. Una tale operazione, *ancorché limitata ai soli puntatori*, è invece lecita se alla sinistra dell'assegnamento compare un *puntatore a char*, come nell'esempio che segue.

```
#include <stdio.h>

main()
{
 char s1[80], *s2;

 printf("\nStringa 1:\t");
 scanf("%s",s1);

 s2=s1;             /* LECITO, ma NON DUPLICA LA STRINGA! */

 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);

 printf("\nModifica stringa 1:\t");
 scanf("%s",s1);
 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);
}
```

Ora l'assegnamento è sintatticamente lecito, tuttavia esso *produce solo una copia di puntatori*, non delle variabili da essi puntate! Perciò, s1 e s2 referenziano ora entrambi *la stessa stringa*, e dunque qualunque modifica fatta sull'una si ritrova immediatamente sull'altra. Per copiare davvero una stringa in un'altra è invece necessaria una copia fisica, uno per volta, di tutti gli elementi, che si può attuare ad esempio tramite un ciclo:

```
#include <stdio.h>

void strcpy(char *s, char *t)
{
     while((*s=*t)!='\0') {s++; t++;}
}
```

```
main()
{
 char s1[80], s2[80];    /* stavolta, s2 deve avere lo spazio */

 printf("\nStringa 1:\t");
 scanf("%s",s1);

 strcpy(s2,s1);

 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);

 printf("\nModifica stringa 1:\t");
 scanf("%s",s1);
 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);
}
```

Osserviamo che la funzione strcpy, scritta in questo modo, è ridondante: un programmatore esperto l'avrebbe probabilmente sintetizzata così:

```
void strcpy(char *s, char *t)
{
     while(*s++=*t++);
}
```

Infatti, gli aggiornamenti dei due puntatori possono essere fatti efficacemente con due post-incrementi, e il confronto col carattere '\0' è, al solito, sovrabbondante.

Osserviamo anche che, per quanto sia stretta la correlazione fra i due, *vettori e puntatori non sono due concetti identici*. Ciò è confermato dal seguente esempio:

```
char msg[] = "testo del messaggio";    /* msg è un VETTORE   */
char *msg  = "testo del messaggio";    /* msg è un PUNTATORE */
```

Nel primo caso, `msg` è il nome di un vettore, la cui lunghezza è fissata staticamente: perciò, i singoli caratteri entro il "testo del messaggio" possono cambiare, ma `msg` si riferisce sempre alla stessa area di memoria. Non gli si può quindi assegnare, come nel primo degli esercizi precedenti, un altro indirizzo, perché il nome msg è univocamente associato a questa zona di memoria, e non ad altre.
Nel secondo caso, invece, essendo `msg` un puntatore, può essere fatto puntare altrove, ma il contenuto della stringa è una costante che non dovrebbe essere modificata. Tentando di farlo, il risultato è, in generale, indefinito[2].

ESERCIZIO n° 11

[2] La ragione di questo è abbastanza evidente: si pensi a cosa potrebbe accadere se più stringhe condividono, nella rappresentazione dei dati organizzata dal sistema. delle sottostringhe...

Scrivere una funzione C in grado di riconoscere identificatori, ovvero sequenze di caratteri che rispettino la seguente grammatica:

```
<identificatore> ::=<lettera> | <lettera> <caratteri>
<caratteri> ::=     <lettera> <caratteri> | <cifra> <caratteri>
<lettera> ::=       _ | a | b | c | ..... | z | A | B | C | ..... | Z
<cifra> ::=         0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Soluzione

Tradotto dal formalismo, si tratta evidentemente di riconoscere sequenze di caratteri che cominciano con una lettera o un underscore, e proseguono con lettere, undescore o cifre. Quindi, ci è richiesta una funziona che riceva in ingresso una stringa e, sostanzialmente, risponda SI o NO (oppure TRUE o FALSE, o 1 o 0, eccetera) a seconda che essa rispetti tale sintassi e costituisca, perciò, un identificatore, o meno. Da notare che è opportuno, per generalità, prevedere l'eventualità che esistano spazi iniziali, che ovviamente devono essere saltati. Una funzione che fa questo può essere la seguente.

```
#define NULL 0

char *identifier( char *s )
{
     while ( isspace(*s) ) s++;
     if ( !(isalpha(*s)) && *s != '_' ) return NULL;
      else s++ ;
     while ( isalpha(*s) || isdigit(*s) || *s=='_' )  s++ ;
     if ( isspace(*s) || *s=='\0' ) return s;
      else return NULL;
};
```

Questa funzione non soltanto verifica la sintassi degli identificatori: restituisce anche un puntatore al primo carattere nella stringa che non corrisponde alla sintassi (ad esempio uno spazio che separa l'identificatore stesso dalla parola successiva). Qualora la verifica dia esito negativo, per convenzione il valore ritornato è NULL.

ESERCIZIO n° 12

Scrivere una funzione C in grado di riconoscere espressioni matematiche che rispettino la seguente grammatica (che costituisce una versione semplificata di quella illustrata nella lezione n. 34):

```
<espressione> ::=   <termine> |
                    <termine> + <termine> |
                    <termine> - <termine>
<termine> ::=       <fattore> |
                    <fattore> * <fattore> |
                    <fattore> / <fattore>
<fattore> ::=       <identificatore> | <numero> | ( <espressione> )
```

Soluzione

In primo luogo, osserviamo che occorrerà disporre di idonee funzioni atte a riconoscere identificatori e numeri, che sono le entità di più basso livello trattate dalla grammatica. Rimandando per gli identificatori all'esercizio precedente, tralasceremo in questa sede di presentare il codice della

funzione che riconosce numeri, in quanto facilmente realizzabile sulla scorta degli esercizi precedenti; ci concentreremo invece sulla parte, più innovativa, di riconoscimento delle espressioni.
Sebbene l'operazione di riconoscimento in sé sia non banale, a causa soprattutto della definizione ricorsiva dei fattori in termini di espressioni, la possibilità offerta dal C di specificare funzioni ricorsive consente di ottenere lo scopo con un codice soprendentemente breve e compatto, come mostra l'esempio sotto riportato.

```
#define NULL 0

char *term(char *);       /* dichiarazioni delle funzioni successive  */
char *factor(char *);
char *identifier(char *);       /* vedere esercizio precedente       */
char *number(char *);           /* da fare a cura del lettore        */

char *expression (char *s)
{
     char *t;
     printf("\nEspressione...");
     t = term(s);
     if ( t == NULL ) return NULL;
      else {
          s = t;
          while ( isspace(*s) ) s++ ;
          if (*s != '-' && s* != '+' ) return s;
                                                 /* è un puro termine */
           else s++;
          while ( isspace(*s) ) s++ ;
          t = term(s);
          if ( t == NULL ) return NULL;
           else return t;
      }
}

char *term (char *s)
{
     char *f;
     printf("...termine...");
     f = factor(s);
     if ( f == NULL ) return NULL;
      else {
          s = f;
          while ( isspace(*s) ) s++ ;
          if (*s != '*' && *s != '/' ) return s;
                                          /* è un puro fattore */
           else s++;
          while ( isspace(*s) ) s++ ;
          f = factor(s);
          if ( f == NULL ) return NULL;
           else return f;
      }
}

char *factor (char *s)
{
     char *t;
```

```
     printf("...fattore...");
     t = identifier(s);
     if ( t != NULL ) return t;
      else  t = number(s);
     if ( t != NULL ) return t;
      else  {                              /* espress. fra parentesi */
           printf("...(espressione)...");
           while ( isspace(*s) ) s++ ;
           if (*s != '(' ) return NULL;
            else s++;
           while ( isspace(*s) ) s++ ;
           t = expression(s);              /* chiamata ricorsiva !! */
           if ( t == NULL ) return NULL;
            else s=t;
           while ( isspace(*s) ) s++ ;
           if (*s != ')' ) return NULL;
            else return ++s;
      }
}

     /* ------ un esempio di main ------- */

main()
{
 char s[250];
 char *t;
 printf("\nIntrodurre l'espressione:\t");
 scanf("%s", &s);
 t = expression(s);
 if ( t==NULL ) printf("\nEspressione errata\n");
  else
     printf("\nEspressione OK\nParte residua non riconosciuta:%s\n",t);
}
```

Come si può vedere, la funzione che analizza i termini è del tutto analoga a quella che analizza le espressioni, in ossequio al fatto che la struttura sintattica delle due entità è assolutamente analoga: in effetti, tutta la differenza sta nel nome dell'entità analizzata e negli operatori (additivi in un caso, moltiplicativi nell'altro) accettati e riconosciuti. Per questo, descriveremo solo la prima.
Nella funzione expression, in primo luogo si verifica la presenza di un termine: se esso non esiste, la funzione termina e fallisce restituendo NULL. Altrimenti, si cerca il primo carattere non-spazio e si controlla se si tratta di un operatore di somma (+ o -): se così non è si termina, restituendo il puntatore a tale elemento non identificato. Se invece si riscontra la presenza di "+" o "-", si parte, tolti gli eventuali spazi, alla ricerca di un nuovo termine. Di nuovo, se esso non viene trovato la funzione fallisce restituendo NULL, diversamente ritorna il puntatore (fornito dalla funzione term) al primo carattere non identificato.
La funzione factor è sostanzialmente simile, anche se formalmente diversa. In questo caso, si cerca in primo luogo un identificatore: se lo si trova si termina restituendo, al solito, il puntatore (fornito da identifier) al primo elemento non identificato; altrimenti, e qui è la sola reale differenza, si cambia strada (ovvero si tenta di applicare una diversa regola di produzione), controllando se non si tratti per caso di un numero. Di nuovo, se così è, si restituisce il puntatore al primo carattere non identificato, altrimenti si passa a tentare la terza e ultima via: quella corrispondente a un fattore realizzato racchiudendo una espressione fra parentesi.

Tolti gli spazi, si procede allora alla ricerca di una parentesi tonda aperta (fallendo se non la si trova), quindi si richiama ricorsivamente la funzione expression originaria per l'analisi del contenuto delle parentesi (fallendo se da essa risulta che non si tratta di una espressione corretta), infine si cerca la parentesi chiusa, fallendo se manca.
Da notare che, per come è definita la grammatica, vi sono alcune differenze rispetto al modo usuale di scrivere le espressioni: in particolare, ogni operatore di un dato livello deve agire esclusivamente su due entità ben definite, eventualmente delimitate da parentesi. Ne segue che

A + 3 6 - 8 * 5 4 / 3 - k2 sono legali,

mentre

a + 3 + 5 4 * 5 / 8 5 - k + 8 sono errate;

la versione corretta sarebbe:

a + (3+5) 4 * (5/8) (5-k) + 8

Questa apparente complicazione è alla base della semplicità con cui è stato possibile scrivere il codice: la grammatica completa della lezione 34 sarebbe stata decisamente più complessa, essenzialmente a causa del fatto che gli operatori devono risultare associativi a sinistra (ovvero, 7 - 4 - 6 deve essere letto come (7-4)-6, e non come 7-(4-6)), il che richiede un'analisi della stringa più sofisticata, con capacità di "guardare avanti". Non approfondiamo oltre questo aspetto ora: chi desidera può comunque provare a riflettere su come differenze grammaticali anche piccole possano comportare notevoli complicazioni nell'analizzatore corrispondente.

ESERCIZIO n° 13

Realizzare una programma che ricopi il file dato come primo argomento nel file dato come secondo argomento, assicurandosi di non avere mai linee più lunghe di 40 caratteri.

Soluzione

Si tratta evidentemente di operare in due fasi: in primo luogo analizzare i parametri di ingresso recuperando i nomi dei file su cui agire, aprendo tali file rispettivamente in lettura e in scrittura (ed emettendo, in mancanza, opportuni messaggi d'errore), e in secondo luogo attivare un ciclo di copiatura che, ogni 40 caratteri letti, inserisca nel nuovo file il carattere di *newline*. Una possibile codifica può essere la seguente:

```
#include <stdio.h>
#define MAXCHAR 40

main(int argc, char **argv)
{
     FILE *fin, *fout;
     void stop(char *);
     int count=0, ch;

     if (argc==3)
     {
          fin=fopen(argv[1],"r");
          fout=fopen(argv[2], "w");
          if (fin==NULL)
               stop("Impossibile aprire file d'ingresso\n");
```

```
            if (fout==NULL)
                  stop("Impossibile aprire file d'uscita\n");
      }
      else stop("Manca qualche parametro\n");

      while(!feof(fin))
      {
            ch=fgetc(fin);
            fputc(ch, fout);
            if (++count>MAXCHAR)
            {
                  fputc('\n', fout);
                  count=0;
            }
      }

      fclose(fin);
      fclose(fout);
      exit(0);
}

void stop(char *msg)
{
      fprintf(stderr,msg);
      exit(1);
}
```

La funzione `feof` garantisce l'uscita dal ciclo al termine del file d'ingresso; la funzione `stop`, da noi definita, stampa invece un messaggio d'errore e fa abortire il programma. Da notare il controllo sui parametri d'ingresso, che tenta di fornire una messaggistica il più possibile vicina al tipo di errore verificatosi. Osserviamo che questa versione del programma non resetta il contatore qualora il testo d'ingresso contenga già al suo interno dei newline: volendolo fare, basta cambiare la parte centrale del ciclo di copiatura come segue.

```
      while(!feof(fin))
      {
            if ((ch=fgetc(fin))=='\n') count=0;
            fputc(ch, fout);
            if (++count>MAXCHAR)
            {
                  fputc('\n', fout);
                  count=0;
            }
      }
```

ESERCIZIO n° 14

Scrivere un programma che appenda al file dato come primo argomento il file dato come secondo argomento.

Soluzione

Anche qui, in funzione dei parametri di ingresso occorre aprire un file in lettura, e l'altro in modo append, ovviamente intercettando e segnalando opportunamente le situazioni anomale; quindi, si attiverà un ciclo di copiatura carattere per carattere. Il programma risultante è quello sotto riportato.

```
#include <stdio.h>

main(int argc, char **argv)
{
     FILE *fin, *fapp;
     void stop(char *);

     if (argc==3)
     {
          fapp=fopen(argv[1],"a");
          fin=fopen(argv[2], "r");
          if (fin==NULL)
               stop("Impossibile aprire file d'ingresso\n");
          if (fapp==NULL)
               stop("Impossibile aprire file d'uscita\n");
     }
     else stop("Manca qualche parametro\n");

     while(!feof(fin))
          fputc(fgetc(fin), fapp);

     fclose(fin);
     fclose(fapp);
     exit(0);
}

void stop(char *msg)
{
     fprintf(stderr,msg);
     exit(1);
}
```

Osserviamo il ciclo di copiatura, che, non essendo più necessario salvare in una variabile il carattere letto, è ridotto all'osso, al punto da scriversi sostanzialmente in un'unica istruzione `while`.

ESERCIZIO n° 15

Scrivere un programma che sostituisca tutte le minuscole in maiuscole nel file dato come (unico) argomento.

Soluzione

Per risolvere il problema, occorre evidentemente essere in grado, in buona sostanza, di aprire un file effettuando sia delle letture sia, ove necessario (in particolare: ove il carattere letto sia una minuscola), delle scritture (qui per sostituire quel carattere con la corrispondente maiuscola). Inoltre, è necessario potersi collocare in una ben precisa posizione sul file, onde sovrascrivere esattamente ciò che si vuole sostituire, e non un carattere qualsiasi. A questo scopo, servono alcune caratteristiche della gestione file del C che fanno parte delle cosiddette *funzioni di aggiornamento (file update functions)*.

In primo luogo, osserviamo che le modalità di lettura, o di append, o di scrittura fin qui utilizzate per aprire i file sono insufficienti, in quanto le prime due *presuppongono che il file esista*, provocando il fallimento della `fopen` in caso contrario, mentre la modalità di scrittura se il file non esiste lo crea, altrimenti lo apre ma *cancellando nel contempo ogni precedente contenuto*. In tal caso, la `fopen` può fallire solo se non è fisicamente possibile creare il file.

Per aprire un file esistente senza distruggerlo, ma con la possibilità di modificarne il contenuto, serve una nuova modalità di apertura, detta *modalità di aggiornamento* e indicata dall'aggiunta, in coda alla normale specifica di apertura, dello specificatore `+` .
Si possono quindi avere due modalità di base `"r+"` e `"w+"`. Ognuna mantiene le sue caratteristiche di base: quindi, `"r+"` presuppone che il file esista, lo apre in lettura ma consente anche (alternativamente) di scriverci sopra, mentre `"w+"`, pur consentendo delle letture, mantiene la propria caratteristica base, creando il file se non esiste o cancellandolo completamente se già esiste. La modalità `"a+"` è analoga a `"r+"`, con la differenza di iniziare ad agire in coda al file.
Nel nostro caso, dato che si tratta di agire su un file che non si vuole distruggere o riscrivere, ma soltanto modificare *selettivamente* in alcuni punti (precisamente, in presenza delle minuscole), la modalità necessaria è chiaramente la `"r+"`.

Aperto quindi il file fornito dal parametro di ingresso, e intercettate le eventuali situazioni anomale, occorrerà avviare un ciclo di letture carattere per carattere, controllando a ogni carattere letto se si tratti di una minuscola. In caso positivo, occorrerà retrocedere di una posizione (perché la testina virtuale di lettura/scrittura di un file si sposta concettualmente in avanti di una posizione subito dopo aver letto o scritto un carattere), e sovrascrivere tale carattere con la maiuscola corrispondente. Questi posizionamenti sul file possono essere effettuati con la funzione di libreria `fseek`. Il programma risultante è quello sotto riportato.

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

main(int argc, char **argv)
{
      FILE *file;
      void stop(char *);
      int ch;

      if (argc==2)
      {     if ((file=fopen(argv[1], "r+"))==NULL)
                  stop("Impossibile aprire file d'ingresso\n");
      }
      else stop("Manca il nome del file da elaborare\n");

      while((ch=fgetc(file))!=EOF)
       if(islower(ch))
       {
            fseek(file, ftell(file)-1, SEEK_SET);
            fputc(toupper(ch), file);
            fseek(file, 0, SEEK_CUR);
       }
      fclose(file);
      exit(0);
}
```

```
void stop(char *msg)
{
     fprintf(stderr,msg);
     exit(1);
}
```

La funzione `fseek` è doppiamente essenziale: serve infatti sia per posizionarsi su una data posizione nel file, *sia a consentire l'alternanza di letture e scritture*. In effetti, la semantica del modo di aggiornamento "+" (abbinato a uno qualunque dei modi `"r"`, `"w"`, e `"a"`) *richiede esplicitamente* che, dopo una sequenza di letture, e prima di iniziare delle scritture, venga usata una delle funzione di riposizionamento su file. Lo stesso vale dopo una sequenza di scritture, prima di passare a eseguire nuovamente delle letture. La sintassi completa di `fseek` (e della funzione ausiliaria `ftell`) è riassunta nel riquadro sottostante.

Sintassi di fseek e ftell:

```
int fseek(FILE *f, long offset, int origin);

long ftell(FILE *f);
```

dove:

`offset` dà la posizione, rispetto a `origin`, a cui portarsi sul file

`origin` dà la posizione, rispetto a cui misurare lo scostamento, e può valere:

- `SEEK_SET` per indicare l'inizio del file
- `SEEK_CUR` per indicare la posizione corrente nel file
- `SEEK_END` per indicare la fine del file

PER UN FILE DI TESTO:

- `offset` deve essere o zero, o un valore restituito da `ftell`. In quest'ultimo caso, `origin` deve obbligatoriamente valere `SEEK_SET`.

PER UN FILE BINARIO:

- la nuova posizione si trova a `offset` caratteri da `origin`.

Nel nostro caso, la funzione `fseek` è necessaria una prima volta (con scostamento `ftell(file)-1` rispetto all'inizio del file) per retrocedere di una posizione, tornando in corrispondenza del carattere appena letto (una minuscola da sostituire), e una seconda volta (con offset zero rispetto alla posizione corrente) semplicemente per consentire, alla successiva iterazione del ciclo, una lettura: in effetti, in questo secondo caso non si effettua alcun reale riposizionamento.
Il ciclo che costituisce il cuore del programma avrebbe anche potuto essere riformulato utilizzando, per esprimere la condizione di controllo, la funzione `feof`:

```
     while(!feof(file)) {
      ch=fgetc(file);
      if(islower(ch)) {
          fseek(file, ftell(file)-1, SEEK_SET);
          fputc(toupper(ch), file);
          fseek(file, 0, SEEK_CUR);
```

```
    }
  }
```

o anche:

```
  while(!feof(file))
   if(islower(ch=fgetc(file)))
   {
        fseek(file, ftell(file)-1, SEEK_SET);
        fputc(toupper(ch), file);
        fseek(file, 0, SEEK_CUR);
   }
```

ESERCIZIO n° 16
Scrivere un programma che salvi in un file, dato come argomento, un vettore di interi.

Soluzione
Finora l'unico tipo di file considerato è stato il file di caratteri, o *file di testo*. Tuttavia, è spesso necessario salvare su file dati che *non* sono caratteri. In questi casi, è assai utile disporre di un supporto più generale di quello costituito dai file di testo: a questo scopo, il linguaggio C offre il concetto di *File Binario*.

Un file binario serve a supportare qualunque tipo di file, in particolare file che *non* contengono caratteri. Poiché un file binario non è un file di caratteri, *la fine del file* deve essere necessariamente testata tramite la primitiva `feof`, non potendosi più applicare il confronto fra un carattere letto e il carattere `EOF`. Per leggere e scrivere su un file binario il linguaggio mette a disposizione le due primitive `fread` e `fwrite`, la cui sintassi è riassunta nel riquadro sottostante.

Sintassi di fread e fwrite:

```
size_t fread(void *vet, size_t size, size_t n, FILE *f);

size_t fwrite(const void *vet, size_t size, size_t n,FILE *f);
```

dove:

`vet`	è un (puntatore a un) vettore di elementi, ognuno di ampiezza `size`
`n`	è il numero di elementi da leggere o scrivere
`f`	è il file (binario)

La prima legge dal file, collocandoli nel vettore `vet`, al più `n` oggetti, ciascuno di ampiezza pari a `size`, e restituisce il numero di oggetti effettivamente letti, che può essere minore di quanto richiesto (se il file conteneva meno elementi). Per determinare lo stato di terminazione si devono usare le funzioni `feof` e `ferror`.
La funzione `fwrite` viceversa scrive sul file, prelevandoli dal vettore `vet`, `n` oggetti, ciascuno di ampiezza pari a `size`, e restituisce il numero di oggetti scritti, che può essere inferiore al richiesto solo in caso di errore (come ad esempio l'esaurimento di spazio su disco).

Usando queste primitive, il problema dato si risolve semplicemente passando a `fwrite` gli opportuni parametri: il programma si potrà presentare allora come sotto illustrato.

```
#include <stdio.h>

main(int argc, char **argv)
{
     FILE *file;
     void stop(char *);
     int i, n;
     int tab[]={3, 6, -12, 5, -76, 3, 32, 12, 65, 1, 0, -9} ;

     if (argc==2)
     {    if ((file=fopen(argv[1], "wb"))==NULL)
```

```
                stop("Impossibile aprire file d'uscita\n");
     }
     else stop("Manca il nome del file di uscita\n");

     n = sizeof(tab)/sizeof(tab[0]);

     fwrite(tab, sizeof(tab[0]), n, file);
     fclose(file);
     exit(0);
}

void stop(char *msg)
{
     fprintf(stderr,msg);
     exit(1);
}
```

Si noti il metodo, del tutto generale, usato per determinare il numero di elementi nel vettore, come rapporto fra la dimensione del vettore e quella di un suo generico elemento.
Da notare che questo programma non verifica che siano stati effettivamente scritti tutti gli elementi: in una applicazione reale, il valore di ritorno di `fwrite` dovrebbe essere controllato, gestendo il caso in cui l'operazione non abbia potuto essere completata.

ESERCIZIO n° 17

Scrivere un programma che che legga da un file, dato come argomento, una lista di interi, ponendola in un vettore.

Soluzione

Si tratta evidentemente del problema duale del precedente, risolvibile semplicemente usando opportunamente la primitiva `fread`. Una possibile codifica può quindi essere la seguente:

```
#include <stdio.h>
#define MAX 40

main(int argc, char **argv)
{
     FILE *file;
     void stop(char *);
     int i, n, tab[MAX];

     if (argc==2)
     {    if ((file=fopen(argv[1], "rb"))==NULL)
               stop("Impossibile aprire file d'ingresso\n");
     } else stop("Manca il nome del file d'ingresso\n");

     n=fread(tab, sizeof(tab[0]), MAX, file);
     fclose(file);
     for(i=0;i<n;i++)
          printf("%d%c", tab[i], (i==n-1) ? '\n' : '\t');
     exit(0);
}

void stop(char *msg)
{    fprintf(stderr,msg);
     exit(1);
}
```

Si noti lo sfruttamento della semantica di `fread` al fine di ottenere "gratis" il numero effettivo di elementi letti come valore di ritorno (salvato in `n`), fornendo nel contempo `MAX` come numero massimo di elementi da leggere in modo da evitare di acquisire più elementi di quanti il vettore ne possa contenere.

ESERCIZIO n° 18

Scrivere un programma che stampi una lista di interi su video, stampante o file a seconda del parametro passato come argomento

Soluzione

Per ottenere lo scopo, la soluzione più semplice è adottare la primitiva `fprintf`, passandole come `FILE*` una variabile opportunamente settata, in precedenza, o uno dei file standard (`stdout` o `stdprn` rispettivamente per video e stampante) o il risultato della `fopen` nel caso di un file su disco. Per comodità, si può stabilire che il default sia lo schermo (e quindi la variabile `FILE*` valga inizialmente `stdout`). Il programma assume quindi un aspetto come il seguente:

```
#include <stdio.h>
main(int argc, char **argv)
{
      FILE *out=stdout;
      char lastch='\n';
      int diskfile=0, i, n;
      int tab[]={ 12, 54, -34, 65, 978, 19, 32 };

      if (--argc>0)
        if (strcmp(argv[argc], "prn:")==0)
             {
                out=stdprn;
                lastch='\f';
             }
        else {
                out=fopen(argv[argc], "a");
                diskfile=1;
             }

      n=sizeof(tab)/sizeof(tab[0]);
      for(i=0;i<n;i++)
           fprintf(out, "%d%c", tab[i], (i==n-1) ? lastch : '\t');

      if (diskfile) fclose(out);
      return 0;
}
```

Dal punto di vista pratico, il programma emette tutto il proprio output sul file `out`, indipendentemente da *cosa* esso realmente sia. A sua volta, il file `out`, inizializzato per default a `stdout`, viene settato su `stdprn` se l'argomento passato vale `"prn:"`, o al file su disco corrispondente al nome fornito, se è presente un qualunque altro argomento. Come si è già accennato, `stdprn` è lo *standard printing device*, aperto anch'esso automaticamente dal sistema all'inizio dell'esecuzione (come `stdin`, `stdout` e `stderr`). Ogni output diretto su `stdprn` viene, di fatto, inviato alla stampante. Analogamente, esiste anche lo stream standard `stdaux`, che corrisponde allo *standard auxiliary device*, la cui funzione dipende da quale dispositivo fisico (porte seriali, o, altro) è realmente connesso al dispositivo logico `"aux:"`.
Il carattere `lastch`, settato per default a un *newline*, viene modificato nel caso della stampante al carattere `'\f'` (form feed) per provocare un salto di pagina.
Da notare, infine, *la cura posta nell'evitare di chiudere per sempre stdout*, dato che non c'è modo di riaprire esplicitamente uno dei file standard, una volta chiusi (le funzioni `fopen` e `freopen`, mostrata nell'esempio successivo, richiedono sempre esplicitamente, infatti, un nome di file).
A questo scopo, il flag `diskfile` condiziona l'operazione di chiusura all'essere `out` un file su disco precedentemente aperto con `fopen`, impedendola se `out` risulta inizializzato a `stdout` (come accade ad esempio in assenza di parametri).

ESERCIZIO n° 19

Scrivere un programma che stampi una lista di interi su video o su file a seconda del parametro passato come argomento, *redirigendo in modo permanente stdout* sul file.

Soluzione

In questo caso, è necessaria una primitiva che riutilizzi un descrittore già attivo (predisposto in precedenza da `fopen`) per "agganciargli" un diverso file fisico. Questo servizio viene svolto in C dalla primitiva di libreria `freopen`. Il programma può quindi presentarsi come segue:

```
#include <stdio.h>
main(int argc, char **argv)
{
     int i, n, tab[]={ 12, 54, -34, 65, 978, 19, 32 };

     /*   ridirige permanentemente stdout:
          il vecchio stdout è perso       */

     if (--argc>0)
          if(freopen(argv[argc], "a",stdout)==NULL)
          {    perror("error redirecting stdout\n");
               exit(1);
          }

     /* tutte le prossime printf andranno sul file */

     n=sizeof(tab)/sizeof(tab[0]);
     for(i=0;i<n;i++)
          printf("%d%c", tab[i], (i==n-1) ? '\n' : '\t');

     fclose(stdout);
     return 0;
}
```

In effetti, la funzione `freopen` è simile alla `fopen`, da cui si distingue perché *riapre* un file anziché aprirlo ex novo. Per questo, richiede come parametro il nome di un `FILE*` già esistente, che viene chiuso e riassociato al nuovo file.

Da notare anche la funzione `perror`, che stampa un messaggio d'errore sullo standard error. Più esattamente:

```
perror(msg)
```

equivale a:

```
fprintf(stderr, "%s: %s\n", msg, errmsg)
```

dove `errmsg` è il messaggio d'errore standard, definito dall'implementazione, relativo all'errore attualmente indicato dalla variabile di sistema `errno` (error number).

ESERCIZIO n° 20

Scrivere una prima versione semplificata del preprocessore C che effettui l'intercettazione delle `#define` e costruisca la tabella delle sostituzioni.

Soluzione

In pratica, si richiede di analizzare il testo di un programma C, intercettando le righe che iniziano con `#define`, e catturando le due stringhe successive. Per semplicità, in questa versione non si considereranno le `#define` relative alle macro: quindi non saranno trattati testi da sostituire contenenti spazi o altri separatori, né le definizioni su più righe (che richiederebbero di gestire il carattere `'\'`).
Sotto queste ipotesi, la soluzione può consistere nel leggere *riga per riga*, tramite `fgets`, il testo del programma, analizzando poi ogni singola riga con `sscanf` al fine di intercettare le `#define`. In questo caso, la `sscanf` stessa può estrarre le due stringhe rappresentanti rispettivamente il testo da cercare e quello da sostituire, riempiendo una riga dell'apposita tabella. Al termine, la stampa della suddetta tabella (organizzata in forma di vettore di strutture) darà il quadro della situazione.
Un esempio di programma che realizza tutto questo è riportato qui di seguito.

```
#include <stdio.h>
#include <string.h>
#define MAXNAME_LEN 30
#define MAXVALUE_LEN 50

typedef struct {
        char name[MAXNAME_LEN];
        char value[MAXVALUE_LEN];
} tab_entry;

#define MAXDEFS 100
#define MAXLINE_LEN 132

static tab_entry tab[MAXDEFS];
static int tabptr = 0;

main(int argc, char **argv)
{
 FILE *source;
 char line[MAXLINE_LEN];
 void get_def(char *line);
 void print_tab(char *);
 if(argc<2)
 {   fprintf(stderr,"Sintassi: DEFS <nomefile.c>\n");
     exit(1);
 }else if ((source=fopen(argv[1],"r"))==NULL)
 {   fprintf(stderr,"File sorgente non trovato\n");
     exit(2);
 }

 while (!feof(source))
 {
  fgets(line, MAXLINE_LEN, source);
  if (strncmp(line, "#define", 7)==0) get_def(line);
 }

 fclose(source);
 print_tab(argv[1]);
 return 0;
}

void get_def(char *line)
```

```
{
 if (tabptr<MAXDEFS)
 {
   sscanf(line, "#define %s%s",
            tab[tabptr].name, tab[tabptr].value);
   tabptr++;
 }
 else
 {
   fprintf(stderr, "\nMassimo numero di definizioni trattabili"
                     " superato\n");
   exit(3);
 }
}

void print_tab(char *file)
{
 int i;

 printf("\n\n File: %s", file);
 printf("\n ----------------------------------------"
          "----------------------------------------");
 printf("\n|     COSTANTE   SIMBOLICA    "
         " |                DEFINIZIONE                 ");
 printf("\n ----------------------------------------"
          "----------------------------------------");
 for(i=0; i<tabptr; i++)
   printf("\n| %*s | %*s |", MAXNAME_LEN-2, tab[i].name,
                             MAXVALUE_LEN-10, tab[i].value );
 printf("\n ----------------------------------------"
          "----------------------------------------");
 printf("\n\n");
 return;
}
```

Il vettore `tab` è un vettore (`static`, e perciò invisibile fuori da questo file) di strutture, ognuna fatta da due stringhe di caratteri denominate `name` e `value`. Esso è gestito dall'indice `tabptr`, pure `static` e inizializzato a zero. Come previsto in fase di specifica, il ciclo principale del programma legge una a una le righe del file d'ingresso, e verifica se iniziano con la sequenza "`#define`": se sì, chiama `get_def` per estrarre la relativa definizione. Alla fine di tutto, `print_tab` stampa (per ora sullo standard output) la tabella delle sostituzioni.
Il cuore della funzione di analisi, `get_def`, è la funzione standard di libreria `sscanf`, che è del tutto identica a `scanf`, tranne per il fatto di prendere il proprio input non dallo standard input (o da un file come `fscanf`), ma dalla stringa passatale (tipicamente, come in questo caso, precedentemente letta con `gets` o funzioni similari). Questo approccio risulta utile ogni volta che è necessario sottoporre una linea a diverse analisi (anche consecutive), spesso senza sapere in partenza quale di queste analisi avrà esito positivo.
La specifica di conversione `%*s` usata nella `printf` della funzione `print_tab` presenta il massimo grado di parametrizzazione del formato: significa che *l'ampiezza del campo* disponibile per stampare `s` non è costante e nota a priori, ma viene fornita volta per volta nella lista degli argomenti. In questo caso, essa vale, rispettivamente per le due stringhe, `MAXNAME_LEN-2` e `MAXVALUE_LEN-10`.

La parola chiave `typedef` definisce un nuovo tipo, in questo caso chiamato `tab_entry`, sulla base del prototipo fornitole. Dal momento in cui essa compare, il nuovo tipo può essere usato in ogni luogo in cui possono essere usati i tipi predefiniti del linguaggio (come `int`, `float`, `char`...). La `typedef` differisce da una `#define`, in quanto la prima è una keyword del linguaggio C, mentre la seconda è soltanto una direttiva al preprocessore, i cui effetti sono certamente già completati *prima* che la compilazione vera e propria abbia inizio.

Da notare che non è indifferente che il nome tab_entry sia posto prima o dopo le parentesi graffe che delimitano la struttura: nel primo caso infatti esso diviene il nome di un tipo, mentre nel secondo è soltanto un'abbreviazione per la successiva elencazione dei membri (tra parentesi graffe), ma non assurge al livello di tipo. Il riquadro sottostante illustra queste due possibilità.

```
typedef struct {
        char name[MAXNAME_LEN];
        char value[MAXVALUE_LEN];
} tab_entry;



struct tab_entry {
        char name[MAXNAME_LEN];
        char value[MAXVALUE_LEN];
};
```

Pertanto, mentre nel primo caso `tab_entry` può essere usato come nome del tipo di una variabile in fase di dichiarazione/definizione della stessa, nel secondo il nome del tipo è `struct tab_entry`.

ESERCIZIO n° 21

Scrivere una programma che consenta di vedere la stessa area di memoria o come un numero intero o come sequenza di bytes, utilizzando il costrutto `union`.

Soluzione

Si tratta semplicemente di definire una `union` avente come campi un `int` e un vettore di bytes (cioè di `unsigned char`) lungo quanto un `int`: la selezione dell'uno o dell'altro consentirà quindi in modo naturale di "commutare", per così dire, fra le due rappresentazioni. Un programma che fa ciò è presentato qui di seguito.

```
#include <stdio.h>

typedef union {
      int val;
      unsigned char rep[sizeof(int)];
} integer;

main(void)
{
      integer i;
      void print_internal_representation(integer);

      i.val=34;

      printf("\nValore: %d\n", i.val);
      print_internal_representation(i);

      printf("\nIndirizzo di i.val:\t%ld\n", &i.val);
      printf("\nIndirizzo di i.rep:\t%ld\n", &i.rep);

      return 0;
}

void print_internal_representation(integer i)
{
      int k;
      printf("\nRappresentazione interna:\n\n");
      for(k=0; k<sizeof(integer); k++)
            printf("%4d%c", i.rep[k],
                  (k==sizeof(integer)-1 ? '\n' : '\t') );
}
```

Sebbene `printf` e `print_internal_representation` stampino i valori di due variabili all'apparenza diverse, ciò che viene visualizzato da quest'ultima dipende dall'assegnamento precedente a `i.val`. In effetti, una `union` non è altro che una collezione di variabili aventi tutte il medesimo indirizzo di partenza, come, peraltro, si può facilmente verificare.

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.

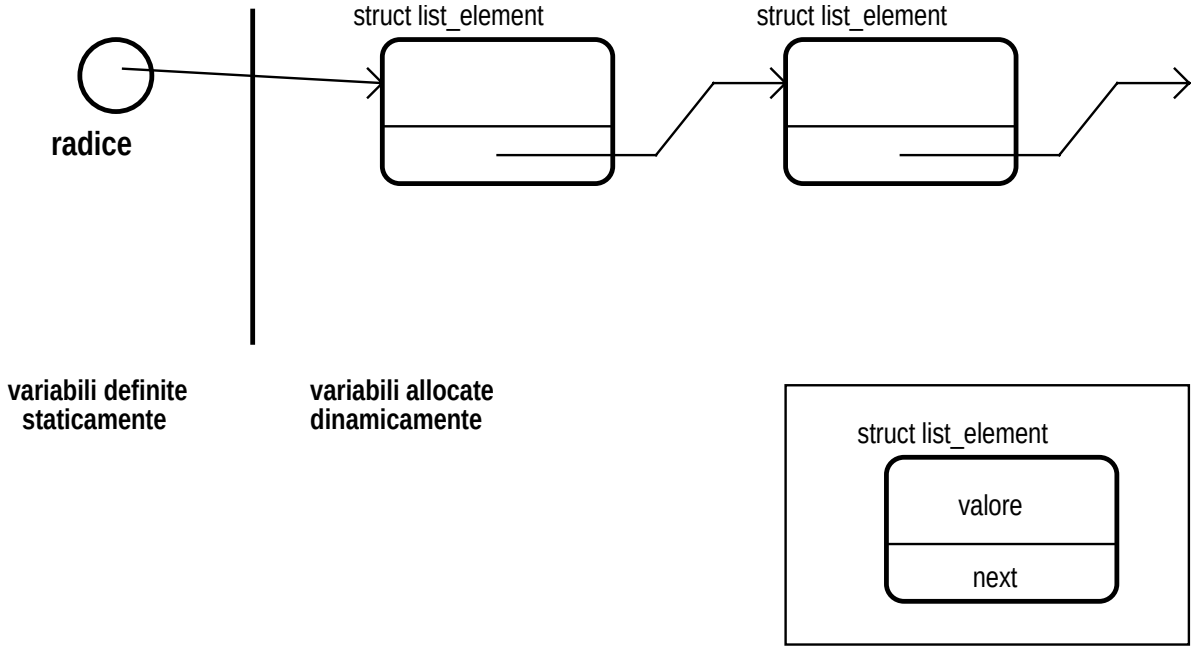
In realtà, la reale potenza dell'allocazione dinamica di memoria emerge trattando strutture dati più evolute dei semplici interi, caratteri o vettori, ossia utilizzando *strutture dati definite ricorsivamente*, come *alberi* e *liste*.

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[3] 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

[3] Relativamente alla lista, ovviamente.

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).

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.