FONDAMENTI DI INFORMATICA II

ESERCITAZIONE n° 2: Linguaggio C

1. Filtri su file + Stringhe e Vettori (parte 2)

ESERCIZIO n° 13

Realizzare un 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 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 un 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.