```
/*
 * Corso di Fondamenti di Informatica
 * Esercizio:
 * uso dei puntatori.
 */

/*-------------------------------------------------------------------------+
 |            RIEPILOGO DELLE NOTAZIONI RELATIVE AI PUNTATORI              |
 +-------------------------------------------------------------------------+
 |     se:                                                                 |
 |     T è un tipo di variabile, predefinito o definito dal programmatore)|
 |     V è una variabile di tipo T                                         |
 |     P è una variabile puntatore ad un dato di tipo T                    |
 |                                                                         |
 | La notazione *P indica il dato puntato da P                             |
 |                                                                         |
 | La notazione &V indica l'indirizzo di V (cioè un valore di tipo         |
 | puntatore a V, in genere descritto come "valore di tipo T*", dove "T*" |
 | viene spesso pronunciato "T star")                                      |
 |                                                                         |
 | La dichiarazione di una variabile P di tipo puntatore a T ha la forma   |
 |                                                                         |
 |     T *P;                                                               |
 |                                                                         |
 | L'asterisco può trovarsi indifferentemente vicino al nome del tipo o    |
 | della variabile: una dichiarazione equivalente alla precedente e'       |
 |                                                                         |
 |     T* P;                                                               |
 +-------------------------------------------------------------------------*/

#include <stdio.h>
#include <string.h>

int main()
{
  /* dichiarazioni di variabile */
  int Intero;
  int ArrInt[5];
  char ArrChar[16];
  int *PuntInt1;
  /* dove sia l'asterisco è indifferente, ma la notazione
```

```
       * della riga successiva (asterisco vicino al tipo) è
       * altrettanto valida */
int* PuntInt2;
char* PuntChar, FintoPuntatore;
/* ATTENZIONE: dichiarato così FintoPuntatore è un char! */
int** PuntPuntInt;
/* notare che questo è un puntatore a puntatore a
       * int, ovverosia un "puntatore doppio" */
int*** PuntPuntPuntInt;
/* si può proseguire all'infinito... */

typedef struct
{
  int CampoInt;
  char CampoChar;
} TipoStruttura;

TipoStruttura Struttura1;
TipoStruttura *PuntStruttura;

/* --------- uso elementare dei puntatori --------- */

PuntInt1 = NULL;
/* fa in modo che pPuntInt non punti a nulla:
       * NULL è il valore "neutro" per i puntatori */

      /* NOTA: il valore della costante NULL è definito nella libreria stdlib.h: per
       * usare NULL è dunque necessario mettere in testa al programma la direttiva
       * #include <stdlib>
       * a meno che si sia già inclusa una libreria, come stdio.h, che a sua volta
       * include stdlib. */

Intero = 7;

PuntInt1 = &Intero;
/* ora pPuntInt punta a Intero */
printf("\n%d", Intero);
/* stampa "7" */

printf("\n%d", *PuntInt1);
/* stampa ancora "7" */
```

```
*PuntInt1 = 2;
/* cambiamo il valore di Intero usando pPuntInt per raggiungere la
 * variabile */
printf("\n%d", Intero);
/* stampa "2" */

PuntInt2 = PuntInt1;
/* adesso anche PuntInt2 punta dove punta attualmente PuntInt1, ovvero ad
 * Intero */
*PuntInt2 = 4;
printf("\n%d", Intero);
/* stampa "4" */

PuntChar = &Intero;
/* ATTENZIONE: il compilatore C può non segnalare l'errore che qui,
 * chiaramente, compare. Infatti tutti i puntatori sono indirizzi, anche se
       * vengono usati per puntare a dati aventi tipi differenti: nulla impedisce,
       * ad esempio, di assegnare l'indirizzo di un dato intero ad un puntatore a
       * carattere, come abbiamo appena fatto. Nonostante ciò, è una pessima
       * idea far puntare un dato da un puntatore inadatto a quel tipo di
       * dato, perché rende poco comprensibili (e poco correggibili) i programmi */


/* ------------- array e puntatori ------------- */

/* NOTA IMPORTANTE. il simbolo ArrInt, che identifica un array di interi,
 * "fisicamente" non è altro che il puntatore (di valore COSTANTE, ovvero
 * non modificabile durante l'esecuzione del programma) alla prima cella
 * di memoria occupata dall'array. Dunque il simbolo ArrInt può essere
 * trattato a tutti gli effetti come una costante di tipo int* (puntatore ad
 * int). Considerazioni simili valgono per qualsiasi array.
 * Considerando quanto appena detto, è evidente che le due espressioni
 *
 *                 ArrInt[k]     e     *(ArrInt+k)
 *
 * hanno lo stesso valore, ovvero rappresentano lo stesso dato: il contenuto
 * della cella di indice k dell'array ArrInt. Infatti, per l'aritmetica dei
 * puntatori, ArrInt+k è un puntatore pari all'indirizzo della cella posta k
 * elementi (di tipo int, visto che ArrInt è un puntatore di tipo int*) dopo
 * quella di indirizzo ArrInt, e quest'ultimo indirizzo è quello del primo
```

```
 * elemento dell'array ArrInt. */

PuntInt1 = ArrInt;
/* perfettamente accettabile */

/* NOTA: al contrario, scrivere
 *  ArrInt = PuntInt1;
 * darebbe luogo ad un errore in compilazione poiché non è possibile cambiare
 * valore alla costante ArrInt */

ArrInt[0] = 11;
printf("\n%d", ArrInt[0]);
/* ovviamente stampa "11" */

*ArrInt = 22;
/* assegna il valore 22 alla prima cella dell'array */
printf("\n%d", ArrInt[0]);
/* stampa "22" */

*(ArrInt + 2) = 33;
/* assegna il valore 33 alla terza cella dell'array. Notare che (ArrInt + 2)
 * è una notazione SIMBOLICA: ArrInt NON viene incrementato di 2, ma di un
 * valore tale da ottenere un puntatore a 2 celle più avanti (in questo caso
 * tale valore è 8, poiché le celle occupano 4 byte ciascuna. Al
 * programmatore, tuttavia, non serve saperlo: è il compilatore ad occuparsi
 * di questi dettagli sulla gestione della memoria). */

printf("\n%d", ArrInt[2]);
/* stampa "33": infatti ArrInt[2] e (ArrInt+2) sono equivalenti per il
 * compilatore C */

PuntInt1 = &ArrInt[2];
/* così pPuntInt punta al 3° elemento dell'array */

printf("\n%d", *PuntInt1);
/* stampa ancora "33" */

PuntInt2 = &ArrInt[10];
/* Attenzione: sebbene l'array ArrInt NON possegga un elemento di indice 10
 * il compilatore NON dà un errore, e nemmeno un warning! */
```

```
ArrInt[25] = 1000;
/* Ancora peggio: neanche qui! Infatti il compilatore interpreta questa
 * istruzione come la richiesta di scrivere il valore 1000 nella cella di
 * indirizzo &ArrInt[25] = ArrInt+25.
 * Se tale cella sia interna o esterna all'array non viene verificato. */

printf("\n%d", *PuntInt2);
/* stampa un valore imprevedibile, visto che PuntInt2 punta ad una cella di
 * memoria che è fuori dall'array e non sappiamo cosa contenga. */

/* Come si vede STA AL PROGETTISTA DI UN PROGRAMMA ASSICURARSI CHE ESSO NON
 * VADA MAI A LEGGERE O SCRIVERE IN ZONE DI MEMORIA "SBAGLIATE"!!
 * Non è possibile affidarsi al compilatore per scoprire questo tipo di
 * errori di programmazione. */


/* --------- array di caratteri e stringhe --------- */

/* In C una stringa è un array di caratteri, il cui contenuto "utile" va
 * dalla prima cella dell'array fino alla cella che precede quella che
 * contiene il carattere speciale di terminazione '\0' (esempio: la funzione
 * printf("%s", ...) non stampa il '\0' e i caratteri che lo seguono).
 * Se il carattere di terminazione è nella prima cella la stringa è
 * considerata "vuota". Dopo il primo possono trovarsi altri '\0', che -se
 * presenti- non hanno alcun significato particolare.
 */

strcpy(ArrChar, "123456789012345");
/* la funzione strcpy inserisce automaticamente un '\0' in coda
       * ai caratteri inseriti: infatti... */
printf("\n%s", ArrChar);
/* stampa "123456789012345" */

printf ("\n%hd", ArrChar[15]);
/* stampa 0, cioè il 16° elemento della stringa letto come unsigned short int
       * (indicando '%d' nella stringa di formattazione); ed il codice numerico
       * corrispondente a '\0' è proprio 0 */

ArrChar[9] = '\0';
/* fa terminare la stringa dopo il nono carattere; si noti che i caratteri
 * successivi ad ArrChar[9] rimangono inalterati, ma trovandosi dopo il '\0'
```

```
 * sono considerati -ad esempio dalla funzione printf- non significativi */

printf("\n%s", ArrChar);
/* stampa solo "123456789" */

PuntChar = &ArrChar[3];
/* in questo modo PuntChar punta alla quarta cella di ArrChar */

printf("\n%s", PuntChar);
/* stampa "456789". Questo perché printf, usata con il carattere di
 * formattazione %s, si aspetta come secondo argomento un puntatore a
 * che interpreta come identificatore di stringa (ovvero come puntatore al
 * primo carattere di un array di caratteri), e poi procede a stampare il
 * contenuto della stringa fermandosi, secondo la convenzione, al
 * raggiungimento di un carattere '\0' */

PuntChar = ArrChar + 3;
/* del tutto equivalente all'assegnamento precedente: infatti... */

printf("\n%s", PuntChar);
/* ...stampa "456789" */

++PuntChar;
/* questo è un modo valido per far avanzare "di una cella" un puntatore */

printf("\n%s", PuntChar);
/* ...stampa "56789" */

printf("\n%c", *(ArrChar + 6));
/* stampa il settimo carattere di ArrChar, ovverosia '7' */


/* ------------- puntatori a puntatori ------------- */

PuntInt1 = &Intero;

PuntPuntInt = &PuntInt1;

*(*PuntPuntInt) = 100;

PuntPuntPuntInt = &PuntPuntInt;
```

```
  printf("\n%d", *(*(*PuntPuntPuntInt)));
  /* stampa "100" */

  printf("\n%d", ***PuntPuntPuntInt);
  /* equivalente alla precedente istruzione */


  /* ------------- puntatori e strutture ------------- */

  PuntStruttura = &Struttura1;

  PuntInt1 = &(Struttura1.CampoInt);
  /* le parentesi non sono necessarie ma chiariscono */

  PuntChar = &Struttura1.CampoChar;
  /* va bene anche così, senza parentesi */

  *PuntInt1 = 111;

  *PuntChar = 'z';

  printf("\n%d", (*PuntStruttura).CampoInt);
  /* questo è un modo per accedere ai campi della struttura; stampa "111" */

  printf("\n%c", PuntStruttura->CampoChar);
  /* notazione semplificata usando l'operatore ->, che rappresenta le due
   * operazioni (1) accesso alla struttura puntata da un puntatore e (2) accesso
   * ad un campo di tale struttura. L'istruzione stampa "z". */

  printf("\n\n");

  return(0);
}
```