Programmazione 1

Lezione 11

Vincenzo Marra

vincenzo.marra@unimi.it

Dipartimento di Matematica *Federigo Enriques*
Università degli Studi di Milano

24 maggio 2017

Array multidimensionali come parametri

- Abbiamo già usato gli array multidimensionali in laboratorio. Va ricordato che tali array sono implementati come array monodimensionali i cui elementi sono a loro volta array della dimensione appropriata. Per esempio, `int v[2][3]` dichiara un array di 2 elementi, ciascuno dei quali è un array di 3 `int`.

Array multidimensionali come parametri

- Abbiamo già usato gli array multidimensionali in laboratorio. Va ricordato che tali array sono implementati come array monodimensionali i cui elementi sono a loro volta array della dimensione appropriata. Per esempio, `int v[2][3]` dichiara un array di 2 elementi, ciascuno dei quali è un array di 3 `int`.
- L'identificatore `v`, al solito, è un puntatore costante al primo elemento dell'array, ossia all'array di tre `int` che costituisce la prima riga della matrice `v[2][3]`.

Array multidimensionali come parametri

- Abbiamo già usato gli array multidimensionali in laboratorio. Va ricordato che tali array sono implementati come array monodimensionali i cui elementi sono a loro volta array della dimensione appropriata. Per esempio, `int v[2][3]` dichiara un array di 2 elementi, ciascuno dei quali è un array di 3 `int`.
- L'identificatore `v`, al solito, è un puntatore costante al primo elemento dell'array, ossia all'array di tre `int` che costituisce la prima riga della matrice `v[2][3]`.
- Ne segue che quando si passa `v` a una funzione, è necessario specificare il numero di colonne (più in generale, la dimensione dell'ultimo indice dell'array) esplicitamente, come nel prototipo:

```
void f(int v[][3]);
```

Si può rendere il valore 3 di questo esempio parametrico, aggiungendo un parametro intero, come segue.

arraypar.c

```c
#include <stdio.h>
int main(void)
{
        void visualizza(int, int c, int[][c]); /* c serve! */
        int v[2][3];

        int i, j;
        for (i=0;i<2;i++) /* riga */
                for (j=0;j<3;j++) /* colonna */
                        v[i][j]=-1;
        visualizza(2,3,v);
        return 0;
}
void visualizza(int r, int c, int v[][c]) /* int v[][c] +dopo+ c */
{
        int i,j;
        for (i=0;i<r;i++) /* riga */
        {
                for (j=0;j<c;j++) /* colonna */
                        printf("%d\t", v[i][j]);
                putchar('\n');
        }
}
```

Il preprocessore

La compilazione del sorgente C è preceduta da una fase di preparazione detta preprocessing (*pre-elaborazione*). In questa fase, il preprocessore C esegue una serie di compiti preliminari quali, per esempio, l'inclusione dei file di intestazione.

Il programmatore può richiedere al preprocessore l'esecuzione di specifici compiti tramite istruzioni dedicate note come direttive per il preprocessore. Esse non fanno parte del linguaggio C in senso proprio.

Il preprocessore

La compilazione del sorgente C è preceduta da una fase di preparazione detta preprocessing (*pre-elaborazione*). In questa fase, il preprocessore C esegue una serie di compiti preliminari quali, per esempio, l'inclusione dei file di intestazione.

Il programmatore può richiedere al preprocessore l'esecuzione di specifici compiti tramite istruzioni dedicate note come direttive per il preprocessore. Esse non fanno parte del linguaggio C in senso proprio.

Le direttive per il preprocessore cominciano tutte con il simbolo

#

variamente detto *hash*, *cancelletto*, o *diesis*. Si estendono fino alla fine della riga in cui compare #, ma non comprendono automaticamente le riga successiva.

#include

- Come abbiamo visto, include un file di intestazione. Il preprocessore sostituisce alla direttiva di inclusione il file indicato.

`#include`

- Come abbiamo visto, include un file di intestazione. Il preprocessore sostituisce alla direttiva di inclusione il file indicato.
- Nella forma `#include <...>`, il file in argomento è cercato dal sistema in una serie di directory che contengono le libreria standard del C. La semantica precisa dipende dall'implementazione.

`#include`

- Come abbiamo visto, include un file di intestazione. Il preprocessore sostituisce alla direttiva di inclusione il file indicato.
- Nella forma `#include <...>`, il file in argomento è cercato dal sistema in una serie di directory che contengono le libreria standard del C. La semantica precisa dipende dall'implementazione.
- Nella forma `#include "..."`, il file in argomento è cercato dal sistema in una serie di directory dipendente dall'implementazione, che tipicamente include la directory in cui risiede il file sorgente in cui compare la direttiva. Altrettanto tipicamente, se il file in argomento è un percorso relativo, esso è interpretato dal sistema come relativo al punto in cui risiede il file sorgente.

`#include`

- Come abbiamo visto, include un file di intestazione. Il preprocessore sostituisce alla direttiva di inclusione il file indicato.
- Nella forma `#include <...>`, il file in argomento è cercato dal sistema in una serie di directory che contengono le libreria standard del C. La semantica precisa dipende dall'implementazione.
- Nella forma `#include "..."`, il file in argomento è cercato dal sistema in una serie di directory dipendente dall'implementazione, che tipicamente include la directory in cui risiede il file sorgente in cui compare la direttiva. Altrettanto tipicamente, se il file in argomento è un percorso relativo, esso è interpretato dal sistema come relativo al punto in cui risiede il file sorgente.
- I file inclusi possono a loro volta contenere direttive `#include`.

`#define`

- La direttiva

  `#define` *identificatore testo*

  chiede al preprocessore di sostituire nel file sorgente, a partire dal punto in cui essa compare, ciascuna occorrenza di *identificatore* con *testo*. Si dice nel gergo del C che `identificatore` è una macro.

`#define`

- La direttiva

  `#define` *identificatore testo*

  chiede al preprocessore di sostituire nel file sorgente, a partire dal punto in cui essa compare, ciascuna occorrenza di *identificatore* con *testo*. Si dice nel gergo del C che `identificatore` è una macro.

- Una successiva direttiva `#define` con il medesimo *identificatore* — cioè, un tentativo di redefinizione — causa un errore un compilazione, a meno che *testo* non sia identico nei due casi. (Gli spazi prima e dopo *testo* sono ignorati; in *testo*, più spazi equivalgono a uno solo.)

`#define`

- La direttiva

  `#define` *identificatore testo*

  chiede al preprocessore di sostituire nel file sorgente, a partire dal punto in cui essa compare, ciascuna occorrenza di *identificatore* con *testo*. Si dice nel gergo del C che `identificatore` è una macro.
- Una successiva direttiva `#define` con il medesimo *identificatore* — cioè, un tentativo di redefinizione — causa un errore un compilazione, a meno che *testo* non sia identico nei due casi. (Gli spazi prima e dopo *testo* sono ignorati; in *testo*, più spazi equivalgono a uno solo.)
- Si usa di frequente la convenzione di di definire le macro in maiuscola, per distinguerle da altri identificatori quali i nomi di variabili. Si possono anche definire macro con parametri, ma non approfondiremo l'argomento.

macro.c

```c
#include "definizioni.h"
#include "funzioniaux.h"
int main(void)
{
        int v[MAXL];
        inizializza(v,INT_IN);
        visualizza(v);
        return 0;
}
```

funzioniaux.h

```c
#include "definizioni.h"
#include <stdio.h>

void inizializza(int *, int); /* Prototipi */
void visualizza(int *);

void inizializza(int *v, int d) /* Implementazioni */
{
        int i=0;
        for (;i<MAXL;i++)
                v[i]=d;
}
void visualizza(int *v)
{
        int i=0;
        for (;i<MAXL;i++)
                printf("%d\n",v[i]);
}
```

definizioni.h

```c
#define MAXL 20 /* Lunghezza max array */
#define INT_IN 1 /* Valore di default elementi array int */
```

Conversioni di tipo implicite ed esplicite

- In molte situazioni il compilatore C si trova a dover convertire dati di un certo tipo in dati di un altro tipo. Si parla di conversione di tipo. La conversione può avvenire implicitamente, o può essere esplicitamente richiesta dal programmatore. Nel secondo caso si parla di casting, o conversione forzata dei tipi.

Conversioni di tipo implicite ed esplicite

- In molte situazioni il compilatore C si trova a dover convertire dati di un certo tipo in dati di un altro tipo. Si parla di conversione di tipo. La conversione può avvenire implicitamente, o può essere esplicitamente richiesta dal programmatore. Nel secondo caso si parla di casting, o conversione forzata dei tipi.
- Esiste un complesso sistema di regole, formalmente definite, che normano la conversione dei tipi in C. Qui daremo solo qualche cenno. Per un approfondimento, si veda K&R, Sezione 6 dell'Appendice A.

Conversioni di tipo implicite ed esplicite

- In molte situazioni il compilatore C si trova a dover convertire dati di un certo tipo in dati di un altro tipo. Si parla di conversione di tipo. La conversione può avvenire implicitamente, o può essere esplicitamente richiesta dal programmatore. Nel secondo caso si parla di casting, o conversione forzata dei tipi.
- Esiste un complesso sistema di regole, formalmente definite, che normano la conversione dei tipi in C. Qui daremo solo qualche cenno. Per un approfondimento, si veda K&R, Sezione 6 dell'Appendice A.
- Un caso semplice si ha quando si combinano tipi diversi in una singola espressione. Per esempio, se `i` è `int` e `d` è `double`, l'espressione

  ```
  d/i
  ```

  ha tipo `double`, e la divisione è eseguita in virgola mobile.

- Questo è un esempio di conversione implicita. Si noti che questa conversione implicita di `d/i` non comporta perdita di informazione: il tipo `int` è sottotipo del tipo `double`.

- Questo è un esempio di conversione implicita. Si noti che questa conversione implicita di `d/i` non comporta perdita di informazione: il tipo `int` è sottotipo del tipo `double`.
- D'altro canto, sappiamo già che se `j` è anch'esso `int`, l'espressione

  `j/i`

  ha ancora tipo `int`, la divisione eseguita è quella intera: il risultato è il quoziente; il resto è scartato. Non si ha conversione.

- Questo è un esempio di conversione implicita. Si noti che questa conversione implicita di `d/i` non comporta perdita di informazione: il tipo `int` è sottotipo del tipo `double`.
- D'altro canto, sappiamo già che se `j` è anch'esso `int`, l'espressione

  ```
  j/i
  ```

  ha ancora tipo `int`, la divisione eseguita è quella intera: il risultato è il quoziente; il resto è scartato. Non si ha conversione.
- Analogamente, `char` è sottotipo di `int`. Così, se `c` e `e` sono `char` e `i` è `int`, l'assegnazione

  ```
  i=c-e;
  ```

  è legittima, e comporta una conversione automatica del tipo dell'espressione a membro destro nel tipo dell'espressione a membro sinistro. Tuttavia, il valore del risultato è qui dipendente dall'implementazione, perché lo standard lascia libertà alle implementazioni di rappresentare `char` in aritmetica con segno o senza segno.

conv.c

```c
#include <stdio.h>
int main(void)
{
        int i=2, j=5;
        double d=5;
        char c='a', e='A'; /* 'a'==97 e 'A'==65 */

        printf("d/i = %g\n", d/i);
        printf("j/i = %d\n", j/i);
        printf("c-e = %d\n", c-e);
        printf("e-c = %d\n", e-c);

        return 0;
}
```

Cosa stampa il programma?

- Per richiedere al compilatore una conversione forzata, o cast, di una data espressione *espr* al tipo `T`, si usa la sintassi:

`(T)` *espr*

- Per richiedere al compilatore una conversione forzata, o cast, di una data espressione *espr* al tipo `T`, si usa la sintassi:

  `(T)` *espr*

- Ovviamente in alcuni casi tale conversione può portare a perdita di informazione: per esempio, quando si chiede di convertire da un tipo in virgola mobile a un tipo intero.

- Per richiedere al compilatore una conversione forzata, o cast, di una data espressione *espr* al tipo `T`, si usa la sintassi:

  `(T)` *espr*

- Ovviamente in alcuni casi tale conversione può portare a perdita di informazione: per esempio, quando si chiede di convertire da un tipo in virgola mobile a un tipo intero.
- Più in generale, si ha sempre potenziale perdita di informazione quando il tipo di *espr* non è un sottotipo di `T`.

- Per richiedere al compilatore una conversione forzata, o cast, di una data espressione *espr* al tipo `T`, si usa la sintassi:

  `(T)` *espr*

- Ovviamente in alcuni casi tale conversione può portare a perdita di informazione: per esempio, quando si chiede di convertire da un tipo in virgola mobile a un tipo intero.
- Più in generale, si ha sempre potenziale perdita di informazione quando il tipo di *espr* non è un sottotipo di `T`.
- Vi sono casi in cui il casting è utile. Tuttavia, esso va usato con cautela. Se ti trovi a scrivere codice che coinvolge casting che possono comportare perdita d'informazione, chiediti innanzitutto se la struttura del tuo programma e la scelta dei tipi di dati siano corrette.

cast.c

```c
#include <stdio.h>
int main(void)
{
        int i=2, j=5; double d=5;
        char c='a', e='A'; /* 'a'==97 e 'A'==65 */

        printf("d/i = %d (no cast)\n", d/i); //perdita di inf
        printf("d/i = %d (cast)\n", (int) d/i); //perdita di inf

        printf("j/i = %g (no cast)\n", j/i); //no perdita di inf,
                                             //ma tipicamente warning
        printf("j/i = %g (cast)\n", (double) j/i); //no perdita di inf,
                                                   //tipicamente no warning

        printf("c-e = %d\n (no cast)", c-e); //tipicamente no warning
        printf("e-c = %c\n (cast)", (char) e-c); //tipicamente no warning

        return 0;
}
```

Cosa stampa il programma?

Buona prosecuzione degli studi!