Tipi di dati derivati

Nicola Bicocchi

DIEF - UNIMORE

Array multi-dimensionali

- Si tratta di una generalizzazione del concetto di vettore
- Sono permesse un numero arbitrario di dimensioni per la struttura dichiarata
- Il caso tipico di array multi-dimensionale è quello di array a due dimensioni, le cosiddette *matrici*
- La sintassi della dichiarazione di un array multi-dimensionale è la seguente:

```
1  nome-tipo identificatore [ card_1 ] [ card_2 ] ... [ card_n] ;
```

Le matrici

- La matrice è tecnicamente un array a 2 dimensioni. Può essere vista come un vettore monodimensionale i cui singoli elementi sono vettori essi stessi. La sintassi della dichiarazione di una matrice è la seguente:

```
1  nome-tipo identificatore [ card_1 ] [ card_2 ] ;
```

- *nome-tipo* è un qualsiasi tipo di dato, sia semplice che derivato
- *identificatore* è il nome che identifica la matrice
- *card_1* e *card_2* indicano la cardinalità delle due dimensioni (righe e colonne, rispettivamente)

Le matrici: esempio di dichiarazione

Esempio di dichiarazione di matrice:

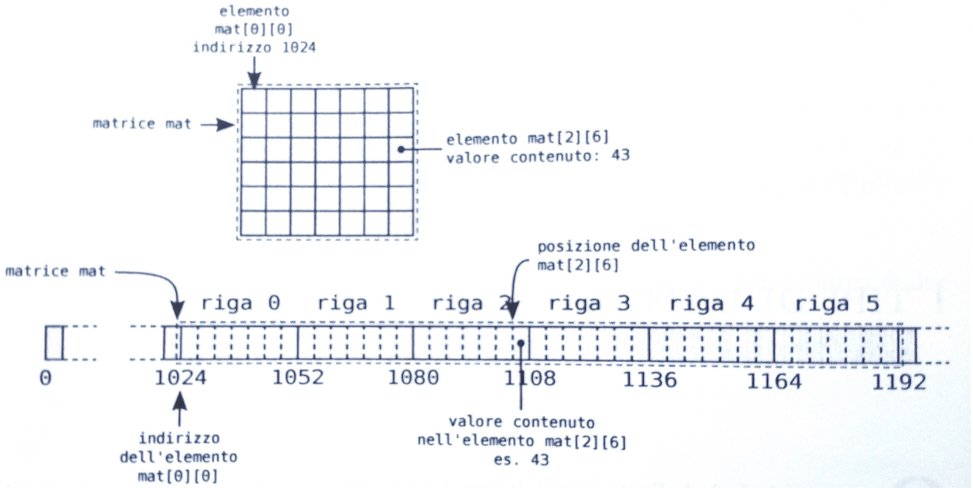
```
1  int mat[6][7];
2  mat[2][6] = 3;
3  printf("%d\n", mat[2][6]);
```

- La matrice è associata all'identificatore *mat*
- La matrice ha 6 righe e 7 colonne
- Le due componenti (righe e colonne) sono indicizzate da 0 a 5 (righe) e da 0 a 6 (colonne)
- L'elemento mat[2][6] (ultimo elemento della terza riga) è un valore di tipo intero che può essere utilizzato come un qualunque altro valore intero

Le matrici: allocazione in memoria

- La matrice è una struttura **bidimensionale**. Va definito il modo in cui mapparla all'interno della memoria RAM, che è al contrario una struttura **monodimensionale**!
- *Una matrice viene allocata in memoria per righe*. Si parte dall'indirizzo dell'elemento di indice mat[0][0] e si prosegue memorizzando in successione tutti i valori della matrice, riga dopo riga.

Le matrici: allocazione in memoria



Le matrici: inizializzazione

- Quando viene dichiarata, una matrice può anche essere inizializzata specificando un elenco di valori per i suoi elementi
- Tra parentesi graffe è racchiusa una lista di elementi separata da virgola
- Ciascun elemento rappresenta una riga della matrice che, a sua volta, è una lista di valori separati da virgola e racchiusa tra graffe

```
1 int mat[2][4] = {
2     {1, 2, 3, 4},
3     {5, 6, 7, 8}
4 };
```

Le matrici: inizializzazione

- Quando nella dichiarazione della matrice si inizializzano i suoi valori, non è necessario indicare la prima dimensione (il numero di righe nel caso bidimensionale). Viene automaticamente calcolata dal compilatore in base ai valori usati per l'inizializzazione.
- Eventuali valori mancanti vengono inizializzati a 0

```
int mat[][4] = {
    {1, 2, 3, 4},
    {5, 6, 7},
    {8, 9},
    {9}
};
```

```
1 2 3 4
5 6 7 0
8 9 0 0
9 0 0 0
```

Le matrici: inizializzazione

```
int mat[][4] = {
    {1},
    {1, 9},
    {1, 7, 3, 5},
};
```

```
1 0 0 0
1 9 0 0
1 7 3 5
```

```
int mat[2][2] = { {0} };
```

```
0 0
0 0
```

Matrici come parametri di funzione

- Per comprendere come una matrice deve essere passata a una funzione è utile ricordare che essa può essere vista come un vettore, i cui elementi sono, a loro volta, vettori di cardinalità pari al numero di colonne (le righe della matrice)
- *Quando un array multi-dimensionale viene passato a una funzione, questa riceve l'indirizzo del suo primo elemento. Non una copia dell'array!* Per dichiarare il tipo del parametro corrispondente, si devono indicare tutte le cardinalità dell'array, eccetto la prima. Nel caso di una matrice, il tipo del parametro che viene passato è quello di un array in cui non viene indicato il numero di righe, ma solo quello delle colonne
- *Il numero di colonne è fondamentale per conoscere l'indirizzo degli elementi memorizzati in righe diverse dalla prima*. Come posso sapere qual'è l'indirizzo di matrice[1][0] se non conosco il numero di colonne?

```
1 void do_stuff(int rows, int cols, int v[][cols]);
```

Matrici come parametri di funzione

```c
void do_stuff(int rows, int cols, int v[][cols]) {
int i, j;

    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            v[i][j] += 1;
        }
    }
}

int main(void) {
    int v[ROWS][COLS] = { {1, 2, 3}, {4, 5, 6}, };

    do_stuff(ROWS, COLS, v);
}
```

Le strutture

- Una struttura, o **struct**, è un tipo di dato derivato che permette di aggregare un insieme di elementi, detti *campi*, all'interno di un unica entità utilizzabile in modo unitario (Si tratta di una forma di aggregazione dei dati, si raggruppano variabili che hanno una correlazione logica per il problema da risolvere)
- I campi di una struttura possono essere di tipo diverso, sia semplici che derivati, incluse altre strutture
- Dopo la dichiarazione, *struct nome* diventa il nome di un *nuovo tipo di dato* che può essere usato per dichiarare variabili e puntatori

```
1  struct nome {
2        tipo-campo nome-campo ;
3        [tipo-campo nome-campo ; ... ]
4  };
5
6  struct nome a, *p;
```

Esempio (struct punto)

- Le variabili di nome pt e pt2 sono di tipo *struct punto*
- L'identificatore pt è associato ad una porzione di memoria in grado di conservare due dati di tipo int (i campi x e y della struttura). Anche pt2 è associato ad una porzione di memoria dedicata
- La variabile pt_ptr è invece di tipo puntatore, e una volta inizializzato, contiene l'indirizzo del primo byte di una *struct punto*

```
1 struct punto {
2     int x;
3     int y;
4 };
5
6 struct punto pt, pt2; /* dichiara due variabili */
7 struct punto *pt_ptr;  /* dichiara un puntatore */
8 pt_ptr = &pt;
```

Esempio (struct punto)

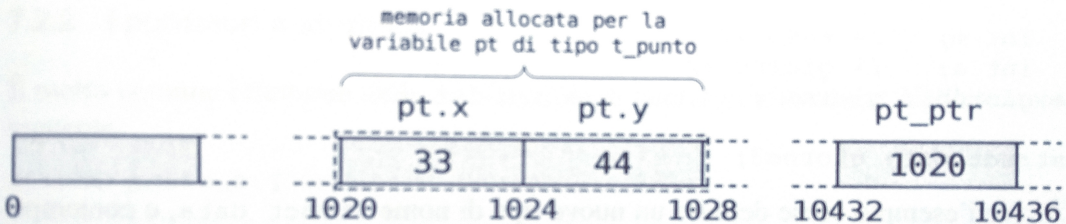


Figura 7.2: Esempio di allocazione in memoria della struttura `pt` e del puntatore `pt_ptr`. Come si vede, per la struttura viene allocato lo spazio necessario per ospitare i due campi interi di cui è composta. La dimensione del puntatore è pari a 4 byte.

Figure 2: Allocazione Matrici

Accesso ai campi

- Per far riferimento ai valori memorizzati nei singoli campi di una struttura si usa la *notazione punto*

```
1 <nome variabile>.<nome campo>
2 pt.x = 5;
3 pt.y = -7
```

- Le strutture si possono anche assegnare direttamente (i valori vengono **copiati** fra le due aree di memoria come nel caso delle normali variabili)

```
1 pt1 = pt;
```

Assegnamento diretto

```c
struct punto {
    int x;
    int y;
};

int main(void) {
    struct punto pt, pt2;

    pt.x = 5; pt.y = -7;

    pt2 = pt;

    pt2.x = -5; pt2.y = 7;

    /* Output: pt=[5, -7] pt2=[-5, 7] */
    printf("pt=[%d, %d] pt2=[%d, %d]\n", pt.x, pt.y, pt2.x, pt2.y);
}
```

Puntatori a struttura

- *pt_ptr* è un puntatore a struttura e memorizza l'indirizzo di una struttura (*struct punto*)
- La sua dichiarazione, non alloca memoria per una struttura ma soltanto per un puntatore ad essa
- La due istruzioni seguenti, ottengono il medesimo scopo, ed assegnano a *pt_ptr* l'indirizzo del primo byte della struttura *pt*

```
1  pt_ptr = &pt;
2  *pt_ptr = pt;
```

Inizializzazione dei campi di strutture

- La prima forma è poco leggibile e legata all'ordine dei parametri (scomodo)
- La seconda forma non è standard (warning)
- *La terza forma è quella consigliata*
- Tutti i campi non specificati vengono impostati al valore 0

```
1  struct info {
2      int id;
3      char *nome;
4      int valore;
5      int privato;
6  }
7
8  struct info el1 = {3, "aldo", 45};
9  struct info el2 = {id: 3, nome: "aldo", valore: 45};
10 struct info el3 = {.id=3, .nome="aldo", .valore=45};
```

Strutture come parametri di funzioni

- Il passaggio dei parametri per valore richiede l'allocazione di una copia locale delle variabili dichiarate nella lista dei parametri. Oltre all'allocazione, tali variabili devono anche essere inizializzate per riflettere il valore della espressione del chiamante. Questo comporta la copia esplicita di una porzione di memoria dalla variabile utilizzata per la chiamata alla variabile locale (perdita di efficienza proporzionale alla dimensione della variabile)
- *Il passaggio per riferimento elimina il tempo necessario per effettuare la copia*. Viene copiato soltanto l'indirizzo della variabile che ha dimensione limitata e fissa (32/64bit). Questo approccio migliora in modo sensibile l'efficienza dei programmi

```
1 double distanza_v1(struct punto p1, struct punto p2);
2 double distanza_v2(struct punto *p1, struct punto *p2);
```

Strutture come parametri di funzioni

- Per ragioni di efficienza, anche se è consentito, le strutture non vengono normalmente passate né come argomenti né vengono utilizzate come valori di ritorno
- Si utilizzano invece spesso *puntatori a struttura*, adeguando la notazione

```c
double distanza(struct punto p1, struct punto p2) {
     return hypot(p1.x - p2.x, p1.y - p2.y);
}
```

```c
double distanza(struct punto *p1, struct punto *p2) {
    return hypot((*p1).x - (*p2).x, (*p1).y - (*p2).y);
}
```

```c
double distanza(struct punto *p1, struct punto *p2) {
    return hypot(p1->x - p2->x, p1->y - p2->y);
}
```

Confronto fra strutture

```c
struct data {
    int g; int m; int a;
};

int datecmp(const struct data *d1, const struct data *d2) {
    int ret;
    if (!(ret=(d1->a - d2->a)))
        if (!(ret=(d1->m - d2->m)))
            ret=(d1->g - d2->g);
    return ret;
}

int main(void) {
    struct data d1 = {.g=10, .m=10, .a=2010};
    struct data d2 = {.g=10, .m=10, .a=2020};
    printf("%d\n", datecmp(&d1, &d2));
}
```

typedef

- In C è possibile assegnare dei nomi simbolici ai tipi di dati esistenti
- Migliora la chiarezza di programmi lunghi e complessi
- La definizione di un nuovo tipo si realizza per mezzo della parola chiave **typedef** utilizzando la seguente sintassi:

```
typedef tipo nuovo-tipo;
```

```
typedef long time_t;
typedef unsigned long long size_t;
typedef struct {
    int x, y;
    int raggio;
} cerchio_t;
```

typedef

- In UNIX, per tenere traccia del trascorrere del tempo in unità discrete si usa la seguente definizione:

```
1 typedef long time_t;
```

- Questo permette di individuare facilmente nel programma le variabili che sono collegate alla gestione del tempo. Esse sono dichiarate di tipo *time_t*, distinguendole da generiche variabili di tipo long utilizzate per altri scopi
- Il fatto di affermare che le variabili sono *dichiarate di tipo time_t è improprio*. L'assegnazione del nome time_t al tipo long non crea un nuovo tipo di dato. *Dal punto di vista semantico una variabile dichiarata di tipo long è perfettamente equivalente ad una di tipo time_t*

typedef

- E’ anche possibile assegnare un nome sintetico a tipi complessi
- Dopo l’utilizzo di **typedef**, si possono definire e utilizzare variabili di tipo cerchio_t
- I puntatori c1 e c2 possono essere utilizzati come puntatori a struttura (equivalenza semantica)

```
1  typedef struct {
2      int x, y;
3      int raggio;
4  } cerchio_t;
```

```
1  int equals(const cerchio_t *c1, const cerchio_t *c2) {
2      return ((c1->x == c2->x) &&
3          (c1->y == c2->y) &&
4          (c1->raggio == c2->raggio));
5  }
```

Le enumerazioni (enum)

- Le enumerazioni sono usate per definire degli insiemi omogenei di costanti intere. Il loro scopo è quello di rendere più comprensibile il codice, permettendo di dichiarare insiemi di costanti dal significato logico coerente
- A ciascuna costante viene associato un nome univoco. Una variabile di tipo enum può essere usata in tutti i contesti nei quali è possibile usare variabili intere (l'indicizzazione di vettori, espressioni)
- Le enumerazioni rappresentano una alternativa alle macro del preprocessore per la definizione di costanti. Hanno il vantaggio che i valori numerici vengono assegati automaticamente dal compilatore.
- Al contrario delle macro, si tratta di tipi veri e propri su cui vengono fatti tutti i controlli di coerenza d'uso

Le enumerazioni (enum)

La sintassi è la seguente:

```
enum identificatore { lista-di-elementi }
```

- *lista-di-elementi* è un elenco di identificatori separati dalla virgola
- Al primo elemento viene assegnato il valore 0
- Ogni elemento successivo viene incrementato di 1

```
enum direzioni { nord, sud, ovest, est };
enum direzioni dir;
dir = est;
...
dir = nord;
```

Le enumerazioni (enum)

- E' anche possibile effettuare degli assegnamenti espliciti
- Il seguente codice usa una enumerazione per dichiarare delle costanti associate ai punti cardinali. A nord viene assegnato il valore 0, sud = 1, ovest = 10, est = 11. Successivamente, dichiara una variabile (dir) e la inizializza al valore est (11)
- Spesso il valore numerico non ha importanza, i nomi sono semplici etichette (non è definito un ordinamento)

```
1 enum direzioni { nord, sud, ovest = 10, est };
2 enum direzioni dir;
3 dir = est;
4 ...
5 dir = nord;
```

Le enumerazioni (enum)

```c
typedef enum { falso, vero } booleano;

booleano flags[10] = { vero };
booleano flag = vero;

/* Output: 1 */
printf("%d\n", flag);
/* Output: vero */
printf("%s\n", flag != falso ? "vero" : "falso");
/* Non produce errori di compilazione! */
flag = 5;

/* in alternativa... */
#define booleano int
#define falso 0
#define vero 1
```