Tipi derivati

Strutture
Matrici
typedef
enum

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 da gestire in modo unitario
- Si raggruppano variabili che hanno una correlazione logica per il problema da risolvere
- I campi di una struttura possono essere di tipo diverso, sia tipi semplici che derivati, incluse altre strutture

Le strutture: sintassi della dichiarazione

- Una struttura viene dichiarata come segue:

```
struct nome {
        tipo-campo nome-campo ;
        [tipo-campo nome-campo ; ... ]
};
```

- Dopo la dichiarazione, “struct nome” è il nome di un nuovo tipo che può essere usato per dichiarare variabili e puntatori

Esempio la struttura punto

```
struct punto {
    int x;
    int y;
};
struct punto pt, pt1; /* dichiara due variabili */
struct punto *pt_ptr; /* dichiara un puntatore */
```

- Le variabili di nome pt e pt1 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 della struttura
- I campi si chiamano x e y

Le strutture: accesso ai campi

- Per far riferimento ai valori memorizzati nei singoli campi si usa la notazione

```
<nome variabile>.<nome campo>
pt.x = 5;
```

- Le strutture si possono anche assegnare direttamente

```
pt1 = pt;
```

Puntatori a struttura

- pt_ptr è un puntatore a struttura
- Memorizza l'indirizzo di una struttura
  - non è stata allocata memoria per la struttura ma soltanto per un puntatore ad essa
- La seguente istruzione assegna a pt_ptr l'indirizzo della struttura pt

```
pt_ptr = &pt;
```

- È ovviamente lecita anche l'istruzione:

```
*pt_ptr = pt;
```

Strutture come parametri di funzioni

- Le strutture possono essere utilizzate come parametri di funzioni:

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

Strutture come parametri di funzioni

- Spesso si preferisce passare puntatori a struttura per questioni di efficienza:

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

Strutture come parametri di funzioni

- Con i puntatori per accedere ai campi si preferisce la notazione ->:

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

Strutture come argomenti di funzione

- Anche se è consentito, le strutture non vengono normalmente passate né come argomenti né vengono utilizzate come valori di ritorno
- Si preferisce allocare le strutture dati separatamente e passare solo i puntatori ad esse, effettuando un passaggio per riferimento
- Questo approccio migliora l'efficienza dei programmi

Strutture come argomenti di funzione

- 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

Strutture come argomenti di funzione

- C'è una perdita di efficienza nel passaggio dei parametri per valore proporzionale alla dimensione della variabile
- Il passaggio per riferimento elimina il tempo necessario per effettuare la copia
- Viene copiato soltanto l'indirizzo della variabile
- Esso ha dimensione limitata e fissa (la dimensione di un puntatore)
- Questo rende più veloce la chiamata alla funzione

Programmi di esempio

`data.c, retangolo.c`

Matrici

Array multidimensionali
Matrici: definizione
Matrici: inizializzazione
Matrici come parametri di funzioni

Gli 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

Le matrici

- La matrice è tecnicamente un array a 2 dimensioni
- La sintassi della dichiarazione di una matrice è la seguente:

```
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)
- una matrice può essere vista come un vettore i cui singoli elementi sono vettori essi stessi

Le matrici

- Esempio di dichiarazione di matrice:

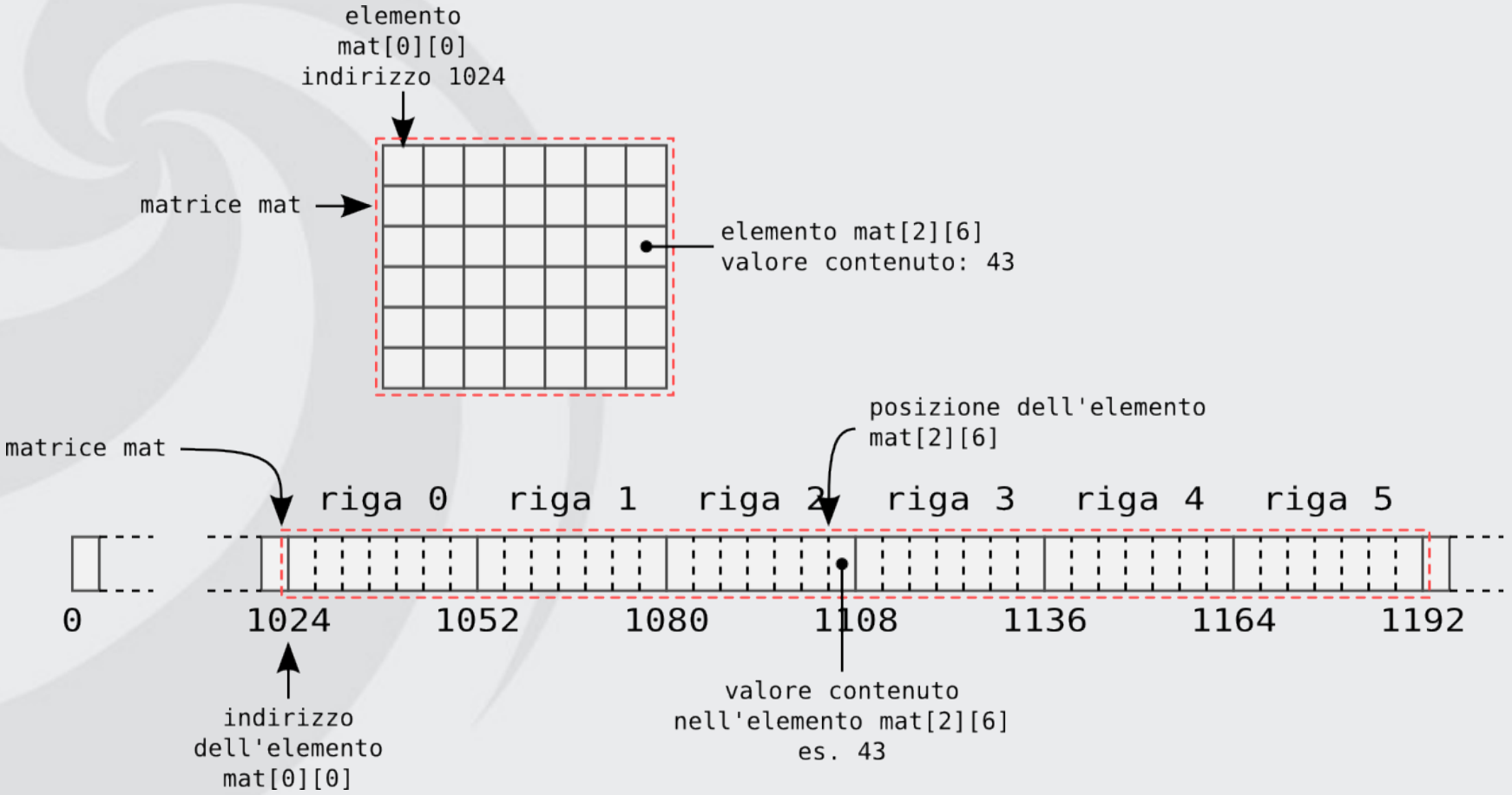
```
int mat[6][7];
```

  - la matrice si chiama mat
  - ha 6 righe e 7 colonne
  - le due componenti sono indicizzate da 0 a 5 e da 0 a 6
  - per esempio mat[2][6] è un valore intero che può essere utilizzato come un qualunque altro valore intero

Le matrici: allocazione

- La matrice è una struttura bidimensionale
- Va definito il modo in cui mapparla all'interno della memoria RAM, che è una struttura monodimensionale
- Una matrice viene allocata in memoria per righe
  - si parte dall'indirizzo dell'elemento di indice mat[0][0]
- vengono memorizzati in successione tutti i valori della matrice
  - sono collocati tutti gli elementi della prima riga
  - si prosegue per tutte le righe che compongono la matrice

Le matrici: allocazione



Le matrici: inizializzazione

- Tra parentesi graffe è racchiusa una lista di elementi separata da virgola
- Ciascun elemento rappresenta una riga della matrice
  - A sua volta è una lista di valori separati da virgola e racchiusa tra graffe

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

Le matrici: inizializzazione

- Non è necessario indicare la prima dimensione. Viene automaticamente calcolata
- Eventuali valori mancanti vengono inizializzati a 0

```
int mat[][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9},
    {0}
};
```

```
int mat[5][6] = {{0}};
```

- Corrisponde alla matrice

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

```
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

Array con più di 2 dimensioni

- è possibile definire array con un numero arbitrario di dimensioni
- la sintassi è la seguente:

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

- Esempio:

```
double var[3][6][9][12];
```

  - viene dichiarato un array a 4 dimensioni
  - un elemento qualsiasi di questo array, per esempio var[0][5][8][1], è un valore double

Matrici come parametri di funzione

- A volte capita di dover elaborare delle matrici di cardinalità prefissata per mezzo di funzioni
  - 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
- 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 puntatore a vettore della dimensione di una riga
  - la sua dichiarazione deve fare riferimento al numero di colonne della matrice

Programmi di esempio

`matrice.c`

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
- La sintassi è la seguente:

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

- L'istruzione associa il nome nuovo-tipo al tipo tipo

typedef

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

```
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 un po' 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

- è possibile assegnare un nome sintetico a tipi complessi, questo aumenta la chiarezza del codice

```
typedef struct {
  int x, y;
  int raggio;
} cerchio_t;
```

- si possono definire e utilizzare variabili di tipo cerchio_t

```
int uguale(cerchio_t c1, cerchio_t c2)
{
  return ((c1.x == c2.x) && (c1.y == c2.y) &&
     (c1.raggio == c2.raggio));
}
```

Le enumerazioni

- Le enumerazioni sono usate per definire degli insiemi omogenei di costanti intere
- A ciascuna costante viene associato un nome univoco
- Il loro scopo è quello di rendere più comprensibile il codice, permettendo di dichiarare insiemi di costanti dal significato logico coerente
- 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: sintassi

- 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
  - è possibile effettuare degli assegnamenti espliciti

Le enumerazioni: sintassi

- il seguente codice usa una enumerazione per dichiarare delle costanti associate ai punti cardinali:

```
enum direzioni { est, sud, ovest = 10, nord };
```

  - a est viene assegnato il valore 0, sud = 1, ovest = 10, nord = 11
  - esempio di uso (dichiaro una variabile e la inizializzo a sud):

```
enum direzioni dir = sud;
```

  - spesso il valore numerico non ha importanza, i nomi sono semplici etichette (non è definito un ordinamento)

Esempio: valori booleani

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

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

printf("%d", flag);
printf("%s", flag != falso ? "vero " : "falso");

flag = 5;  // non dà errori in compilazione

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

```
1
vero
```

in stdbool.h tramite #define sono definiti bool, true e false

Union

- La definizione di una union è molto simile ad una struct:

```
union nome {
        tipo-campo nome-campo ;
        [tipo-campo nome-campo ; ... ]
};
```

- viene usata la parola chiave union invece di struct
- con le union tutti i campi sono sovrapposti, ovvero associati allo stesso indirizzo di memoria
  - il termine union deriva da "uniti", riferito ai campi che sono memorizzati a partire dallo stesso indirizzo
  - si riserva spazio in memoria sufficiente solo per il dato più grande fra i vari campi descritti

Esempio di uso di union

test-endian.c