Sistemi Operativi

Unità 3: Programmazione in C

Tipi complessi

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

Argomenti

1. Vettori
2. Le `struct`
3. Le `union`

Vettori

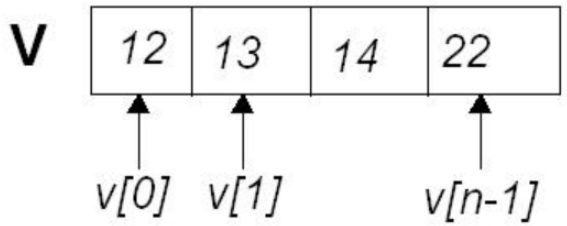
Vettori

I tipo di dato semplici (`int` o `float`) possono contenere un solo dato alla volta.
In C, si possono creare tipi di dato complessi, che contengono più valora. Noi vedremo:

- I vettori o `array`
- Le strutture o `struct`

Vettori

Definizione: Un vettore o `array` è un insieme di variabili dello stesso tipo. E' composto di `N` celle, ognuna identificata da un **indice**.



Utilizzi: vastissimi. Permettono di trattare liste di oggetti senza ripetere il codice

- Effettuare operazioni matematiche: media, varianza
- Gestire flussi di dati

Vettori

Ripetere il codice è una pratica **sbagliata.**

```
int dato1, dato2, dato3, dato4, dato5 ;
int dato6, dato7, dato8, dato9, dato10 ;

scanf("%d", &dato1);
scanf("%d", &dato2);
scanf("%d", &dato3);
...
```

Inutile e prono a errori!

Vettori

Versione corretta:

```
int dato[10]; // Definizione di array

for(i=0; i<10; i++)
    scanf("%d", &dato[i]);

for(i=9; i>=0; i--)
    printf("%d\n", dato[i]);
```

Vedremo la sintassi esatta nelle prossime slide.

Vettori

Definizione di un vettore:

```
tipo nome [N];
```

Esempio:

```
int vettore [10];
```

Definisce un array chiamato `vettore` composto da 10 interi (`int`).

Vettori

Nota: La lunghezza del vettore deve essere nota in fase di **compilazione**. Deve essere una costante.

Il seguente codice è errato:

```
int N;
scanf("%d",&N);
float data [N];
```

Questa è una grande **differenza** rispetto ad altri linguaggi di programmazione come Java o Python.
Esistono metodi per creare array di lunghezza arbitraria in C (la funzione `malloc`), che vedremo più avanti nel corso.

Vettori

Costanti: esistono due modi in C per dichiarare delle costanti.

1. Tramite una direttiva `define`:

```
#define N 10
```

2. Tramite il modificatore `const` applicato a una variabile.

```
const int N = 10; // N non è modificabile
int dato[N];
```

Vettori

Definizioni alternative: si può definire ed inizializzare allo stesso tempo un vettore.

```
int v[4] = {2, 7, 9, 10};
```

In questo caso, si può omettere la lunghezza, che viene inserita automaticamente dal compilatore.

```
int v[] = {2, 7, 9, 10};
```

Vettori

Accesso agli elementi: Si deve specificare l'indice. La sintassi è la seguente.

```
nomevettore[valoreindice]
```

Esempio:

```
int v [] = {4,5,6};
printf("%d\n", v[1]); // stampa 5
```

Vettori

Indici:

- Partono da 0 e arrivano a $N - 1$
- Devono essere `int`
- Possono essere delle variabili o i risultati di una espressione

Importante: in C, non viene controllato che l'indice sia minore di $N - 1$. Se si accede con indici oltre i limiti, si va a leggere locazioni di memoria arbitrarie, che contengono dati arbitrari.

Vettori

Esercizio: si leggano 5 interi e si stampino in ordine inverso.

```c
#include <stdio.h>
#define N 5
int main ()
{
    int v[N];
    int i;

    for (i=0; i<N; i++){
        printf ("Inserisci l'elemento %d: ", i);
        scanf("%d", &v[i]);
    }
    for (i=N-1; i>=0; i--)
        printf ("Elemento %d: %d\n", i, v[i]);

    return 0;
}
```

Vettori

Copia di un vettore:

- Bisogna compiare il contenuto elemento per elemento
- Tra vettori della stessa lunghezza e stesso tipo.
- Sbagliato tentare di copiare usando una sola istruzione.

Corretto:

```
for (i=0; i<N; i++)
    v2[i] = v1[i];
```

Sbagliato:

```
v1 = v2;
v1[] = v2[];
```

Vettori

Copia di un vettore: Spiegazione:

- La variabile vettore è un contenitore di elementi
- Di per se è immutabile
- Si possono solo modificare gli elementi contenuti
- Approfondiremo quando vedremo i puntatori

Vettori

Altri utilizzi dei vettori:

Matrici: un array di array è una **matrice**.

```
int matrice [3][2];
matrice[1][0]=12;
float m[2][2]={{1,2,},{3,4}};
```

Stringhe: un array di `char` è una **stringa**.

Per definizione, in C le stringhe sono array di char, il cui ultimo elemento è 0 (o `'\0'`)

- Perchè così se ne può derivare la lunghezza
  Molti usi e funzioni sulle stringhe. Vedremo più avanti.

```
char s[4] = {'a', 'p', 'e', '\0'};
```

Vettori

Esercizio: si un numero N da tastiera. Si leggano poi N interi e si stampi se essi includono duplicati.

```c
#include <stdio.h>
#define MAXN 50 // Limite massimo del vettore
int main ()
{
    int v[MAXN]; // Vettore sovradimensionato
    int N, i, j;

    printf ("Si inserisca N: ");
    scanf("%d", &N); // Lunghezza effettiva del vettore

    if (N>MAXN){
        printf("N deve essere minore o uguale a %d\n", MAXN);
        return 1; // Ritorna un errore
    }

    for (i=0; i<N; i++){
        printf ("Inserisci l'elemento %d: ", i);
        scanf("%d", &v[i]);
    }

    for (i=0; i<N; i++)
        for (j=0; j<i; j++)
            if (v[i]==v[j])
                printf("L'elemento %d è duplicato dell'elemento %d\n", i,j);
    return 0;
}
```

Osservazione: si è scelto di sovradimensionare il vettore `v` rendendolo lungo `MAXN`, ma utilizzandolo fino all'elemento `N-1`.

Con la memoria dinamica che vedremo più avanti, questo work-around non sarà più necessario.

Le `struct`

Le `struct`

Le **strutture** o `struct` sono collezioni che contengono variabili non necessariamente dello stesso tipo.

Funzionamento:

1. Si definisce la `struct`, un nuovo tipo di dato complesso formato da più record.
2. Si creano e si usano variabili del tipo appena creato.

Le `struct`

Definizione di una `struct`:

```
struct nome {
    campi
};
```

Esempio:

```
struct punto{
    float x;
    float y;
};
```

Le `struct`

Creazione di variabili `struct`:

Per creare nuove variabili di un tipo `struct` definito in precedenza.

```
struct nome;
```

Esempio:

```
struct p1, p2;
```

Le `struct`

Acesso ai campi `struct`:

```
variabile.campo
```

Esempio:

```
p1.x = 2.5;
p1.y = 3.0;
```

Le `struct`

Utilizzo di `typedef`: per evitare di dover premettere `struct` ogniqualvolta si crea una variabile, si può usare la keyword `typedef`, con la seguente sintassi.

```
typedef struct {
    campi
} nome;
```

Esempio:

```
typedef struct{
    float x;
    float y;
} punto;
punto p1, p2; // Si può omettere struct
```

Le `struct`

Esercizio: si crei un programma che effettua la somma vettoriale tra due vettori bi-dimensionali.

```c
#include <stdio.h>

// La dichiarazione di una struct è solitamente fuori da ogni funzione
typedef struct{
    float x;
    float y;
} punto;

int main ()
{
    punto p1, p2, p3; // Tre variabili di tipo 'punto'

    // Lettura
    printf ("P1 -> x: ");
    scanf("%f", &p1.x);
    printf ("P1 -> y: ");
    scanf("%f", &p1.y);
    printf ("P2 -> x: ");
    scanf("%f", &p2.x);
    printf ("P2 -> y: ");
    scanf("%f", &p2.y);

    // Somma
    p3.x = p1.x + p2.x;
    p3.y = p1.y + p2.y;

    printf("La somma vettoriale è il punto: (%f, %f)\n", p3.x, p3.y);

}
```

Le `struct`

Osservazioni:

Non è possibile confrontare due `struct` con gli operatori `==` o `!=`.

E' necessario confrontare tutti i campi

```
punto p1, p2;
if (p1==p2) // Sbagliato!
    ...
if (p1.x==p2.x && p1.y == p2.y) // Corretto
    ...
```

Inizializzazione: si può fare come coi vettori

```
punto p1 = {1.1, 2.4} // x=1.1 e y = 2.4
```

Le `struct`

Le `struct` sono molto usate per creare nuovi tipo di dato complesso:

- Sono come record di un database
- Esempi: numero complesso, indirizzo stradale, ecc...

Sono molto usate nelle librerie del C:

- Permettono di creare tipi di dato arbitrari
- Per rappresentare strutture del sistema operativo.
- Esempi: variabili di sincronizzazione, pacchetti di rete, ecc...

Le `union`

Le `union`

Definizione

Una `union` o **unione** è una variabile che può contenere **in momenti diversi** oggetti di tipo (e dimensione) diversi, con, in comune, il ruolo all'interno del programma.

- Le `union` servono per **risparmiare memoria**
- Usate particolarmente in sistemi *embedded* con stretti vincoli di risorse

Le `union` servono anche per avere un **tipo di dato generico** che ha tipo diverso a seconda della circostanza

Le `union`

Implementazione in C

Si alloca la memoria per la **più grande delle variabili**, visto che esse non possono mai essere utilizzate contemporaneamente (la scelta di una esclude automaticamente le altre)

- I campi **condividono** il medesimo spazio di memoria.
- Se si cambia il valore a un campo, il valore di tutti gli altri campi viene sovrascritto

Le `union`

Definizione in C

In C una unione viene definita tramite la parola chiave `union`

- La definizione di un'unione è molto simile a quella di una `struct`, ed è medesimo il modo di accedervi
- Si può usare `typedef` per evitare di dover premettere `union` ogniqualvolta si crea una variabile

```
union student
{
  uint64_t tessera_sanitaria;
  uint32_t matricola;
};
```

Le `union`

Utilizzo

Si dichiarano e utilizzano come le `struct`

```c
#include <stdio.h>
#include <stdint.h> /* Per uint32_t e uint64_t */
union student
{
  uint64_t tessera_sanitaria;
  uint32_t matricola;
};

int main( int argc, char *argv[] ){
    union student luca;
    luca.matricola = 1234;
    printf("Matricola: %d\n", luca.matricola);

    luca.tessera_sanitaria = 2897189786;
    /* Usare il formato %ld essendo in intero lungo */
    printf("Tessera Sanitaria: %ld\n", luca.tessera_sanitaria);
    return 0;
}
```

Le `union`

Errori di utilizzo

L'assegnazione di un campo **sovrascrive** il valore gli altri campi.

```
union student luca;
luca.matricola = 1234;

// Sovrascrivo luca.matricola
luca.tessera_sanitaria = 2897189786;

// Leggo correttamente tessera_sanitaria
printf("Tessera Sanitaria: %d\n", luca.tessera_sanitaria);

// Leggo matricola, che è stata sovrascritta!
printf("Matricola di Luca: %d\n", luca.matricola); // Errore!
```

Le `union`

Casi d'uso

Si usano quando serve dichiarare variabili che possono assumere tipo diverso a seconda delle circostanze.

Esempio: Uno studente può essere identificato col *numero di tessera sanitaria* o con la *matricola* a seconda della situazione

```
union student
{
  uint64_t tessera_sanitaria;
  uint32_t matricola;
};
```

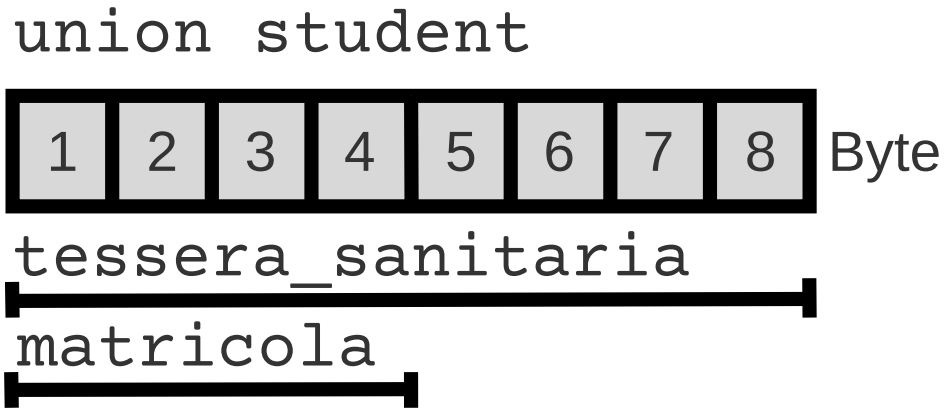
La `union student` **non** può contenere **contemporaneamente** tessera sanitaria e matricola

- Il programma deve essere scritto di conseguenza

Le `union`

Memoria e `union`

Come detto, una `union` occcupa lo spazio necessario al campo più grande.

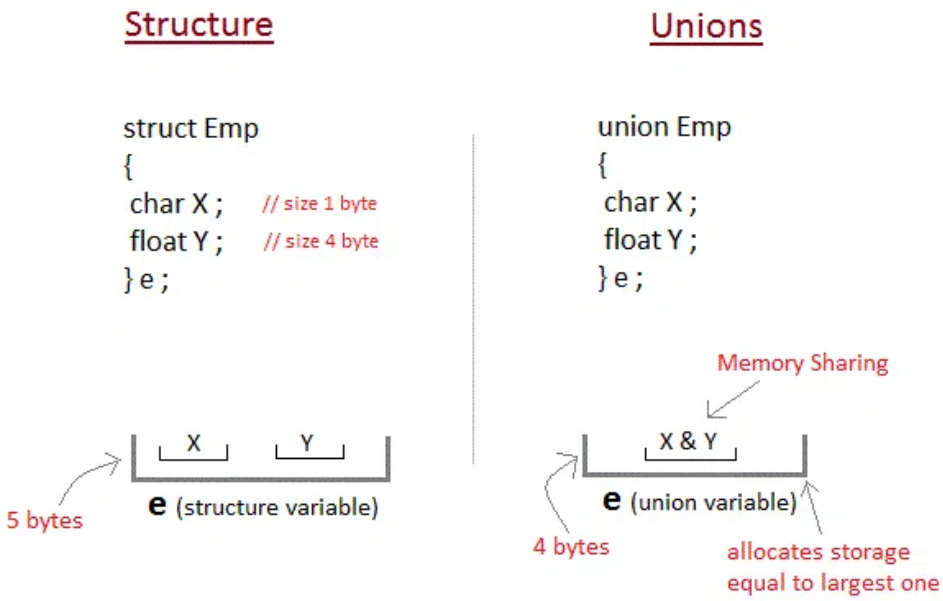


Corollario: non ha senso una `union` in cui ho più campi dello stesso tipo

Le `union`

`struct` vs `union`

- La `struct` contiene abbastanza spazio per contenere **tutti** i campi
- La `union` ha spazio per contenere **un campo alla volta**, occupando in memoria lo spazio del campo più grande



Le `union`

Quale campo viene usato?

Una `union` **non** contiene indicazione sul campo che è in uso

- Semplicemente, accedere a campi non valorizzati, genera un comportamento imprevedibile

Il programmatore deve **tenere traccia** del tipo della variabile di unione usata in un determinato momento:

- Determinato dal flusso del programma
- Si usa una variable esterna
- Si combinano una `struct` e una `union`
  - Una `struct` contiene una `union` e una variabile che indica il tipo dell'unione

Le `union`

Combinare `struct` e `union` (1/2)

Esempio: si usano `struct` e `union` per definire un tipo di dato che può rappresentare velivoli diversi

I tipi di velivolo, con caratteristiche diverse. Ognuno è una una `struct` con campi diversi.

```
typedef struct {
    int maxpassengers;
} jet;

typedef struct{
    int liftcapacity;
} helicopter;

typedef struct{
    int maxpayload;
} cargoplane;
```

Le `union`

Combinare `struct` e `union` (2/2)

A questo punto si definisce una `union` che contiene campi di tipo `jet`, `helicopter` e `cargoplane`.

- Ovvero, può contenere in istanti diversi un `jet`, `helicopter` e `cargoplane`

```
typedef union{
    jet jetu;
    helicopter helicopteru;
    cargoplane cargoplaneu;
} aircraft;
```

Il tipo di dato finale è `an_aircraft`, una `struct` che contiene solamente una `union` `aircraft` e un `int` che indica di che tipo è il velivolo

```
typedef struct {
    int kind;
    aircraft description;
} an_aircraft;
```

Le `union`

Accedere in diversi modi allo stesso dato

Una `union` si può usare per accedere in maniera diversa alla stessa variabile

```
typedef union
{
    struct {
        unsigned char byte1;
        unsigned char byte2;
        unsigned char byte3;
        unsigned char byte4;
    } bytes;
    uint32_t dword;
} cpu_register;
```

Posso accedere a una variabile di tipo `cpu_register` in due modi:

```
cpu_register reg;
reg.dword = 0x12345678;
reg.bytes.byte3 = 4;
```

Le `union`

Utilizzo per `bit mask` in Sistemi *embedded*

Similmente all'esempio precedente, le `union` sono un modo pratico per manipolare variabili a cui spesso si fa accesso *bit a bit*.

```
typedef union {
    unsigned char control_byte;
    struct {
        unsigned int nibble  : 4;
        unsigned int nmi     : 1;
        unsigned int enabled : 1;
        unsigned int fired   : 1;
        unsigned int control : 1;
    };
} ControlRegister;
```

Le `union`

Utilizzo in Linux

Alcune **System Call** in Linux accettano `union` per essere più flessibili.

Esempio: I **segnali** permettono di mandare **messaggi** a **processi**. A seconda delle circostanze il messaggio può essere un `int` o un *puntatore* (ovvero un indirizzo di memoria)

```
union sigval {
  int     sigval_int; /* Integer value */
  void   *sigval_ptr; /* Pointer value */
};
```