



U.D. 5

I tipi strutturati

1. **Array**
2. **Stringhe**
3. **Struct/Union**

Fonti:

Antola Dispense del Corso

Il Record

Esempio:

Si vuole realizzare l'astrazione "contribuente", caratterizzata dai seguenti attributi

- Nome,
- Cognome,
- Reddito,
- Aliquota

Con gli strumenti visti finora, per ogni contribuente e' necessario introdurre 4 variabili:

```
charNome[20], Cognome[20];  
int Reddito, Aliquota;
```

soluzione scomoda e non "astratta": le quattro variabili sono indipendenti tra di loro.

- E' necessario un costrutto che consenta l'aggregazione dei 4 attributi nell'astrazione "contribuente": il **record**.

Tipi strutturati: il Record

Un record e' un **insieme finito** di elementi **non omogeneo**:

- il numero degli elementi e' rigidamente fissato a priori.
- gli elementi possono essere di tipo diverso.
- il tipo di ciascun elemento componente (*campo*) e' prefissato.

Ad esempio:

OME	COGNOME	REDDITO	ALIQUOTA

Formalmente:

Il record e' un tipo strutturato il cui dominio si ottiene mediante **prodotto cartesiano**:

dati n insiemi, $A_{c1}, A_{c2}, \dots, A_{cn}$, il prodotto cartesiano tra essi:

$$A_{c1} \times A_{c2} \times \dots \times A_{cn}$$

consente di definire un tipo di dato strutturato (il record) i cui elementi sono n-ple ordinate:

$$(a_{c1}, a_{c2}, \dots, a_{cn})$$

dove $a_{ci} \in A_{ci}$.

Ad esempio: Il numero complesso e' definito attraverso il prodotto cartesiano: $R \times R$

Il Record in C: struct

Collezioni con un numero finito di campi (anche disomogenei) sono realizzabili in C mediante il costruttore di tipo strutturato **struct**.

Definizione di variabile di tipo record:

```
struct { <lista definizioni campi> } <id-variabile>;
```

dove:

- **<lista definizioni campi>** e' l'insieme delle definizioni dei campi componenti, costruita usando stesse regole sintattiche della definizione di variabili:

```
<tipo1> <campo1>;
```

```
<tipo2> <campo2>;
```

```
...
```

```
<tipoN> <campoN>;
```

- **<id-variabile>** e' l'identificatore della variabile di tipo record cosi' definita.

Esempio:

```
struct { char Nome[20];  
        char Cognome[20];  
        int Reddito;  
        int Aliquota;  
        }contribuente;
```

Nome, Cognome, Reddito ed Aliquota sono i campi della variabile contribuente (di tipo record, o struct).

Il tipo struct

Operatori:

Gli unici operatori previsti per dati di tipo struct sono:

- l'operatore di **assegnamento** (=): e' possibile l'**assegnamento** diretto tra record di tipo equivalente.
- operatori di **uguaglianza** e **disuguaglianza** relazionale (==, !=)

Accesso ai campi:

E' possibile accedere (e manipolare) i singoli campi di un record.

1. Per accedere ai campi di un record, in C si usa la notazione *postfissa* :

<id-variabile>.<componente>

indica il valore del campo *<componente>* della variabile *<id-variabile>* .

- I singoli campi possono essere manipolati con gli operatori previsti per il tipo ad essi associato.

Ad esempio:

```
struct { char Nome[20];  
        char Cognome[20];  
        int   Reddito;  
        int   Aliquota;  
}contribuente;
```

```
contribuente.Reddito=2000+1500;  
strcpy(contribuente.Nome,"Mario");  
strcpy(contribuente.Cognome,"Rossi");  
contribuente.Aliquota=40;
```

Inizializzazione di record:

E' possibile inizializzare i record in fase di definizione.

Ad esempio:

```
struct { char Nome[20];
        char Cognome[20];
        int   Reddito;
        int   Aliquota;
    }p= ("Mario", "Rossi", 17000, 10);
```

Il costruttore di tipo struct

Il costruttore struct puo` essere utilizzato per dichiarare tipi non primitivi basati sul record:

Dichiarazione di tipo strutturato record:

```
typedef struct{<lista dichiarazioni campi>} <id-tipo>;
```

dove:

- <lista definizioni campi> e` l'insieme delle definizioni dei campi componenti;
- <id_tipo> e` l'identificatore del nuovo tipo.

Ad esempio:

```
typedef struct { int anno;
                int mese;
                int giorno;
            }tipodata;
```

```
tipodata data;
unsigned int anno=1999;
```

```
data.anno=anno;
data.mese=1;
data.giorno=6;
```

- ☞ Gli identificatori di campo di un record devono essere distinti tra loro, ma non necessariamente diversi da altri identificatori (ad es., anno).

I Record in C

Riassumendo:

Sintassi:

```
[typedef] struct {  
    <tipo_1> <nome_campo_1>;  
    <tipo_2> <nome_campo_2>;  
    ...  
    <tipo_N> <nome_campo_N>;  
} <nome>;
```

Vincoli:

- <nome_campo_i> e' un identificatore stabilito che individua il campo i-esimo;
- <tipo_i> e' un **qualsiasi** tipo, semplice o strutturato.
- <nome> e' l'identificatore della struttura (o del tipo, se si usa **typedef**)

Uso:

- la struttura e' una collezione di un numero fissato di elementi di vario tipo (<tipo_campo_i>);
- il singolo campo <nome_campo_i> di un record R e' individuato mediante la notazione: R.<nome_campo_i>;
- se due strutture di dati di tipo **struct** hanno lo stesso tipo, allora e' possibile l'assegnamento diretto.

TIPI STRUTTURATI DEL C - COSTRUTTORE STRUCT - 1

Consente di dichiarare variabili che tramite un unico identificatore rappresentano un aggregato di informazioni non omogenee (record).

Costruttore struct {}

- una struttura (record) è definita come un insieme prefissato di **campi** (elementi) **non omogenei**
- ogni campo è dotato di **nome** e di **tipo**

Dichiarazione di una variabile di tipo record

SINTASSI

```
struct {  
    tipo_campo1 nome_campo1;  
    tipo_campo2 nome_campo2;  
    tipo_campo3 nome_campo3;  
} nome_var;
```

SEMANTICA

- **nome_var** rappresenta il valore della variabile (valore visto come aggregato di valori di tutti i suoi campi)
- **&nome_var** rappresenta l'indirizzo del primo byte allocato per la variabile

COSTRUTTORE STRUCT - 2

Accesso ai campi della struttura: sintassi C (operazione legata al costruttore di tipo): *notazione puntata*

```
nome_var. nome_campo1
```

```
struct {
    char    cognome[30];
    char    nome[30];
    int     matricola;
    int     votiesa[29];
}studente;
```

```
studente.matricola
studente.votiesa[3]
studente.cognome[0]
```

Operatori utilizzabili per i singoli campi: operatori «leciti» per il tipo del campo

Operazioni «globali»: assegnamento

```
typedef struct {
    char    cognome[30];
    char    nome[30];
    int     matricola;
    int     votiesa[29];
}STUDENTE;
```

```
STUDENTE    studente1,studente2;
```

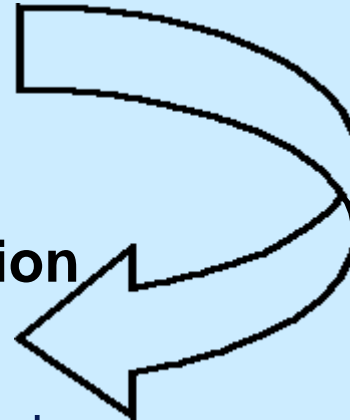
```
studente2=studente1;
```


Definition

- **Keyword struct**
 - introduces the definition
 - followed by optional tag
 - members within braces
 - optional instance(s)

- **Example of struct definition**

```
struct <tag> {  
    datatype membername:  
    datatype membername;  
    datatype membername;  
} instance1, instance2;
```




Examples

```
struct employee {  
    char name[MAXNAME];  
    int age;  
} emp;
```

```
struct date {  
    int month, day, year;  
} HIRE_DATE, TERM_DATE, CURRENT_DATE;
```

```
struct pay_period {  
    struct date start;  
    struct date end;  
} Emp_Worked, Emp_Schedule;
```



Typedef

- **Creates synonym for defined data type**
any data type, not just structs
`typedef int integer; /* now integer means int */`
- **Often used with structs to make shorter type names**
`typedef struct family_type family_rec;`
`/* now family_rec means struct family_type */`
- **Does NOT create new data type**
- **Does NOT allocate storage**

Typedef Examples

```
typedef int WORDSIZENUM;
```

```
struct date  
{  
    int month, day, year;  
};
```

```
typedef struct pay_period  
{  
    struct date start;  
    struct date end;  
} PAY;
```

```
PAY week1;
```

Usage

- **Cannot be compared**
 - members not always in consecutive bytes
 - machine-dependent memory boundaries
- **Can be initialized**
 - using initializer list like arrays
 - remaining members set to 0 or NULL
 - *extern structs initialized by default*
 - by assignment
 - of same type struct
 - of each member

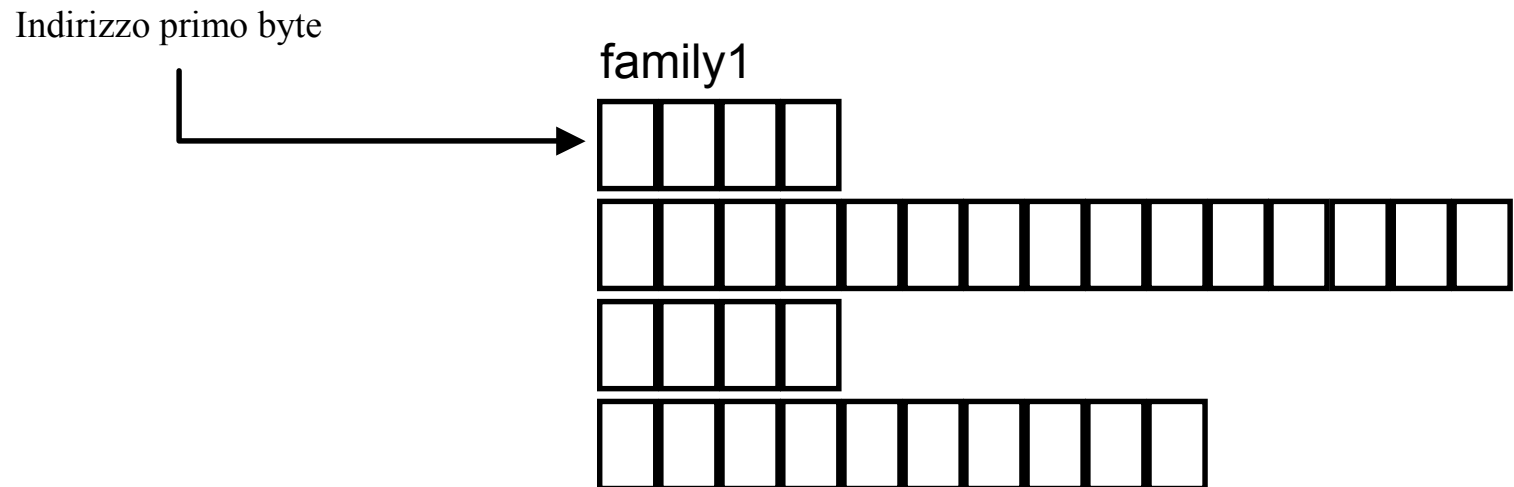
Spazio di memoria allocato per una variabile di tipo struct

- adeguato a contenere i valori dei suoi campi
- lo spazio di memoria viene allocato ad indirizzi fisici contigui

Rappresentazione in memoria di una struttura

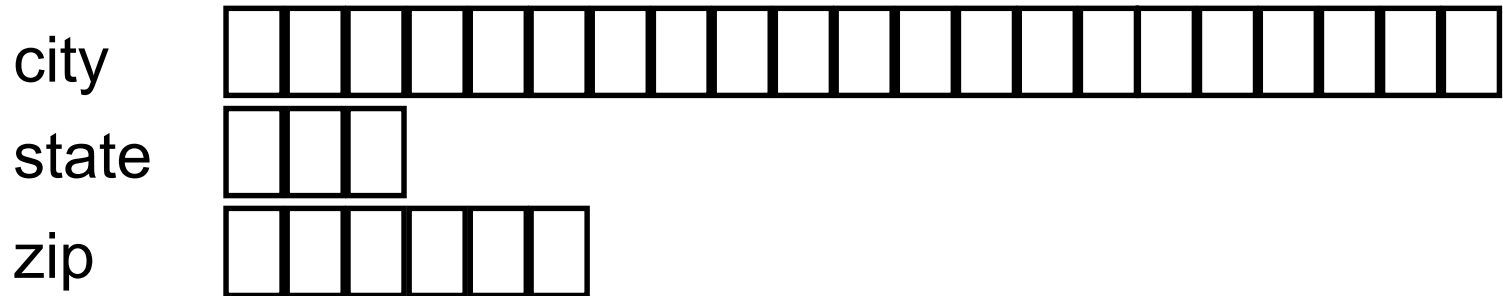
Memory and Struct

```
Given: struct family_rec {  
        int id_num;  
        char name[15];  
        int num_members;  
        double income;  
    } family1;
```



Nested Structs

```
struct address {  
    char city[21];  
    char state[3];  
    char zip[6];  
};
```



```
struct address default;  
strcpy(default.city, "Hammond");
```

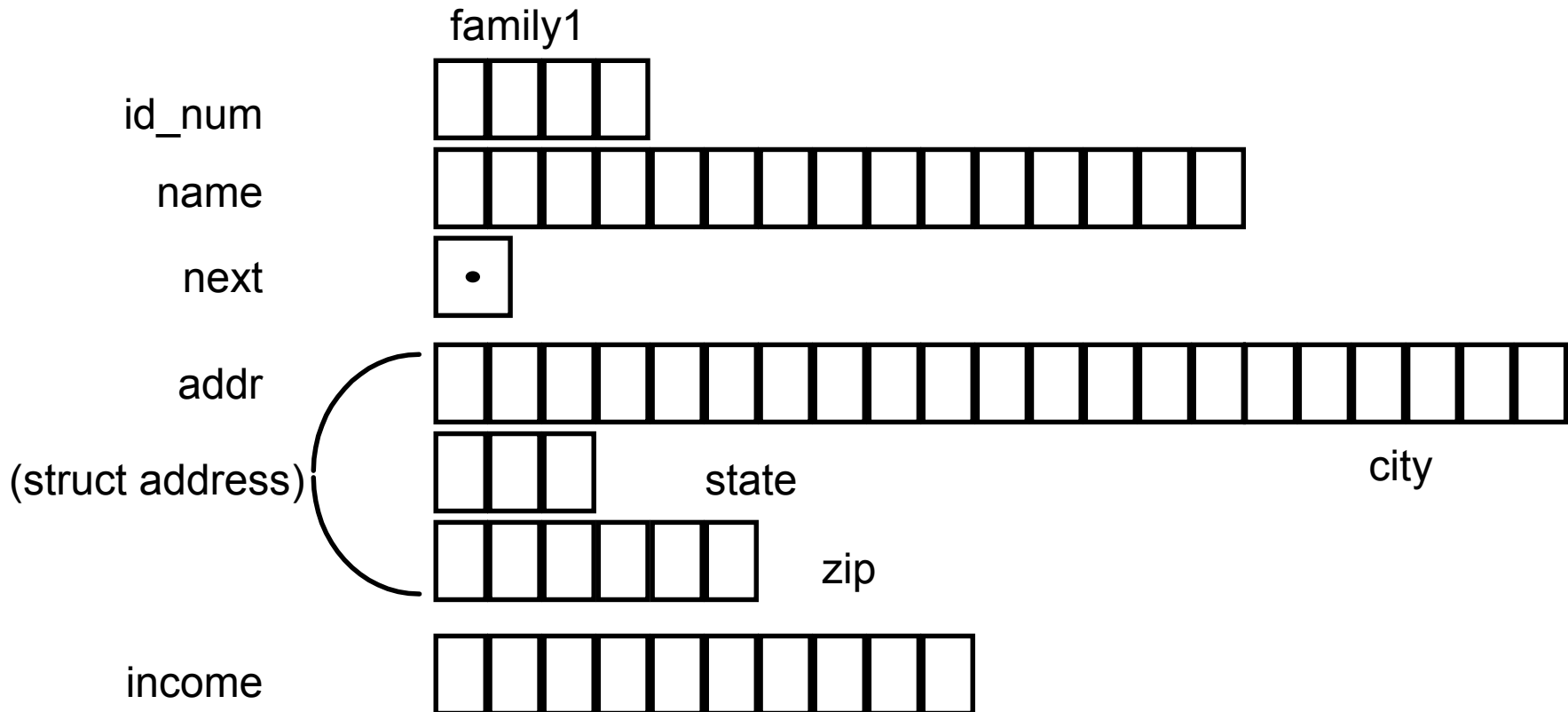

Nested Structs

Given:

```
struct address {  
    char city[21];  
    char state[3];  
    char zip[6];  
};  
  
struct family_rec {  
    int id_num;  
    char name[15];  
    struct family_rec *next;  
    struct address addr;  
    double income;  
} family1, family2, *fptr;
```



Nested Structs



Nested Structs - Accessing Members

```
family1.id_num = 5;  
strcpy(family1.name, "Jones");  
family1.next = &family2;  
strcpy(family1.addr.city, "Gary");  
strcpy(family1.addr.state, "IN");  
strcpy(family1.addr.zip, "46403");  
family1.income = 600.00;
```

Examples

```
struct employee_days_worked {  
    struct employee person  
    struct pay_period work_schedule;  
} EMPLOYEE;
```

```
strcpy(EMPLOYEE.person.name, "Sherlock Holmes");  
EMPLOYEE.person.age = 27;  
EMPLOYEE.work_schedule.start.month = 4;  
EMPLOYEE.work_schedule.start.day = 5;  
EMPLOYEE.work_schedule.start.year = 95;
```

What membername(s) composes the struct date?

Vettori e record

Non ci sono vincoli riguardo al tipo degli elementi di un vettore: si possono realizzare anche **vettori di record (tabelle)**.

Ad esempio:

```
typedef struct {   char Nome[20];  
                  char Cognome[20];  
                  int  Reddito;  
                  int  Aliquota;  
                  } Contribuente;  
contribuente archivio[1000];
```

☞ archivio è un vettore di 1000 elementi, ciascuno dei quali è di tipo contribuente ➤ vettore di record (struttura *tabellare*).

	Nome	Cognome	Reddito	Aliquota
0				
1				
2				
...				
999				

Vettori e Record

Allo stesso modo, si possono fare record di record e record di vettori; ad esempio:

```
typedef struct{ int giorno;
                int mese;
                int anno;
            }data

typedef struct{ char nome[20];
                char cognome[40];
                data data_nasc;
            } persona;

persona P;

...
P.data_nasc.giorno=25;
P.data_nasc.mese=3;
P.data_nasc.anno=1992;

...
```

ARRAY DI STRUTTURE

```
#define N    1000
```

```
typedef struct {  
    char    cognome[30];  
    char    nome[30];  
    int     matricola;  
    int     votiesa[29];  
}STUDENTE;
```

```
STUDENTE    iscrittiaer[N], iscrittiges[N],  
iscritttimec[N];
```

```
iscrittiaer[20].matricola  
iscritttimec[300].votiesa[27]
```

Unions

Definition

- **Derived data type**
 - as is a struct
- **members share SAME storage space**
 - # of bytes \geq largest member
 - only 1 member can be referenced at a time
- **same format as struct; keyword is union**

```
union <tag> {  
    datatype membername;  
    data type membername;  
};
```


Usage

- **Initialization**
 - only with same type as first member
- **Operations**
 - assign to another union of same type
 - address operator
 - accessing members
 - via member operator
 - via pointer operator
 - CANNOT compare unions, only members

Examples

```
union number {  
    char short_num;  
    int inum;  
    double dnum;  
};
```

```
/*declares a_num as type union number */
```

```
union number a_num = {'A'}; /* valid */
```

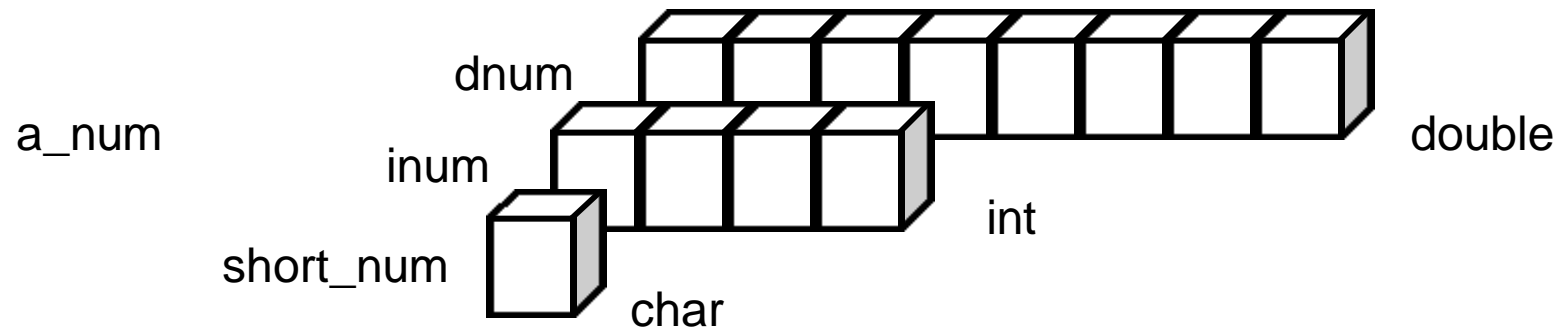
```
union number b_num = {1.55}; /* invalid */
```

```
b_num = a_num;
```

```
a_num.dnum = 88.99;
```

Unions in C

Union



Examples

```
typedef struct lawyer {
    char    summer_home_addr[30];
    char    winter_home_addr[30];
    char    vacation_condo_addr[30];
    double  annual_income;
} LAWYER;
```

```
typedef struct doctor {
    char    practice_addr[30];
    char    malpr_law_firm[20];
    int     number_specialties;
    float   annual_income;
} DOCTOR;
```

Examples

```
typedef struct college_prof {  
    char   school[30];  
    int    tenure_flag;  
    short  annual_income;  
} PROF;
```

```
typedef union career {  
    DOCTOR   career1;  
    LAWYER   career2;  
    PROF     career3;  
} CAREER;
```

```
CAREER person;
```

Il costruttore di tipo union

Mediante il costruttore union si possono dichiarare tipi non primitivi.

Dichiarazione di tipi non primitivi:

```
typedef union { <definizione 1>;  
               <definizione 2>;  
               ...  
               <definizione N>;  
            }<nometipo>;
```

dove:

- <nometipo> è l'identificatore del nuovo tipo;
- <definizione 1>, <definizione 2>, .. <definizione N> sono le definizioni alternative (*varianti*) per la variabile <nome>.

Ad esempio:

```
typedef union{ float raggio_circ;  
              float lati Rettangolo[2];  
              float lati triangolo[3];  
              float lato_quadrato;  
            }figura;  
figura O1, O2;  
...  
O1.raggio_circ=12.5;  
O2.lato_quadrato=5.25;  
...
```

Record e Union

Non ci sono vincoli riguardo al tipo dei campi di un record: è possibile realizzare uno o più campi di un record mediante union.

☞ In questo modo si possono realizzare strutture che hanno:

- una **parte fissa**
- una **parte variante**

Esempio:

Si vuole realizzare un tipo di dato per rappresentare i libri gestiti da una biblioteca. In particolare, per ogni libro si vogliono memorizzare le seguenti informazioni:

- Codice
- Titolo
- Autore
- Editore
- Anno

Inoltre per ogni libro:

- può essere necessario registrare la sua **collocazione** all'interno della biblioteca (se è disponibile);
- oppure può essere necessario memorizzare i **dati del cliente** al quale è prestato. (se è in prestito).

- ☞ Notiamo che il libro è caratterizzato da una parte fissa (codice, autore, titolo, etc.) e da una parte variante (cliente o collocazione).

```
typedef struct { char Cognome[30];
                char Indirizzo[40];
                char N_tel[15];
                int giorno;
                int mese;
                int anno;
            }cliente;
```

```
typedef struct { int stanza;
                int scaffale;
            }collocazione;
```

```
typedef struct
{ char Codice[6];
  char Titolo[30];
  char Autore[30];
  char Editore[20];
  int Anno;
  union { cliente Cli;
         collocazione Col;
        }V; /* p. variante */
} libro;
```

```
libro L1, L2;
/* L1 è disponibile: */
L1.V.Col.stanza=3;
L1.V.Col.scaffale=7;
/* L2 è in prestito: */
scanf("%s", &L2.V.Cli.Cognome);
gets(L2.V.Cli.Indirizzo);
...
```

Union

Esempio:

```
...
libro L;
...
/* L è disponibile: */
L.V.Col.stanza=3;
L.V.Col.scaffale=7;
printf("%s", L.V.Cli.Cognome); /* ?? */
...
```

- ☞ Non c'è alcun controllo sulla rappresentazione adottata: anche se L rappresenta un libro disponibile, si può accedere ad L.V.Cli.Cognome !!

Problema:

Come riconoscere il tipo di rappresentazione valida (ad un certo istante) per un dato di tipo union?

- ☞ Per favorire una gestione corretta delle union, può essere utile introdurre un campo aggiuntivo (**tag**) in base al cui valore si può dedurre il tipo di rappresentazione adottata.

Esempio:

```
typedef struct
{ char Codice[6];
  char Titolo[30];
  char Autore[30];
  char Editore[20];
  int Anno;
  int disponibile; /* tag */
  union { cliente Cli;
        collocazione Col;
        }V; /* p. variante */
} libro;

libro L;
...
if (L.disponibile) /* L è disponibile: */
  {L.V.Col.stanza=3;
  L.V.Col.scaffale=7;
  }
else /* L è in prestito: */
  {printf("%s", L.V.Cli.Cognome);
  printf("%s", L.V.Cli.Indirizzo);
  }
...

```

☞ In questo modo si mantiene **esplicitamente** la **consistenza** della struttura dati.