PROGRAMMAZIONE 2

a.a. 2023-2024

`struct, typedef, enum, union`

Ripasso sulle variabili in C

Variabile

```
TIPO  NOME;                dichiarazione
NOME = ESPRESSIONE;        utilizzo


int x;
x = 0;
x = x+1;
```

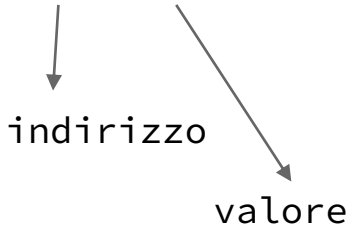
Variabile

TIPO	quanti byte occupa, come sono organizzati
NOME	identifica la variabile per il programmatore
INDIRIZZO	identifica l’area di memoria occupata dalla variabile per l’esecuzione
VALORE	dato contenuto nell’area di memoria di una variabile

Variabile

IL SIMBOLO x HA DUE SIGNIFICATI DIVERSI A SECONDA CHE OCCORRA A SINISTRA O DESTRA DEL SIMBOLO `=`:
- A SINISTRA DENOTA UN **L-VALUE**, OVVERO L'INDIRIZZO DI UNA PORZIONE DI MEMORIA IN CUI È POSSIBILE SCRIVERE;
- A DESTRA DENOTA UN **R-VALUE**, OVVERO IL VALORE CONTENUTO NELLA PORZIONE DI MEMORIA DI DIMENSIONE **sizeof(T)**, DOVE **T** È IL TIPO DI x, CHE INIZIA ALL'INDIRIZZO ASSOCIATO A x (ALL'INIZIO DELL'ESECUZIONE DEL PROGRAMMA IN CASO SI ALLOCAZIONE STATICA, AL MOMENTO IN CUI LA FUNZIONE IN CUI x È DICHIARATA IN CASO DI ALLOCAZIONE DINMAMICA)

```
int x;        DICHIARAZIONE
x = 0;        ASSEGNAZIONE, DI x SI USA L'INDIRIZZO
x = x+1;      ASSEGNAZIONE IN CUI È USATO ANCHE IL VALORE
```

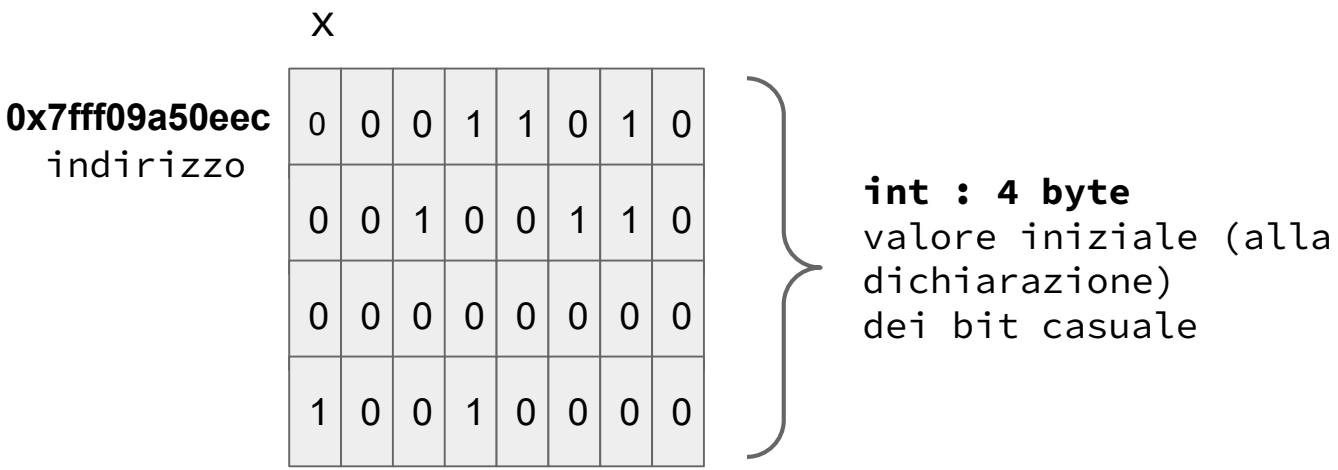


L'indirizzo di x si ottiene con l'operatore unario &:

&x : indirizzo di x

Variabile

```
int x;
x = 0;
x = x+1;
```

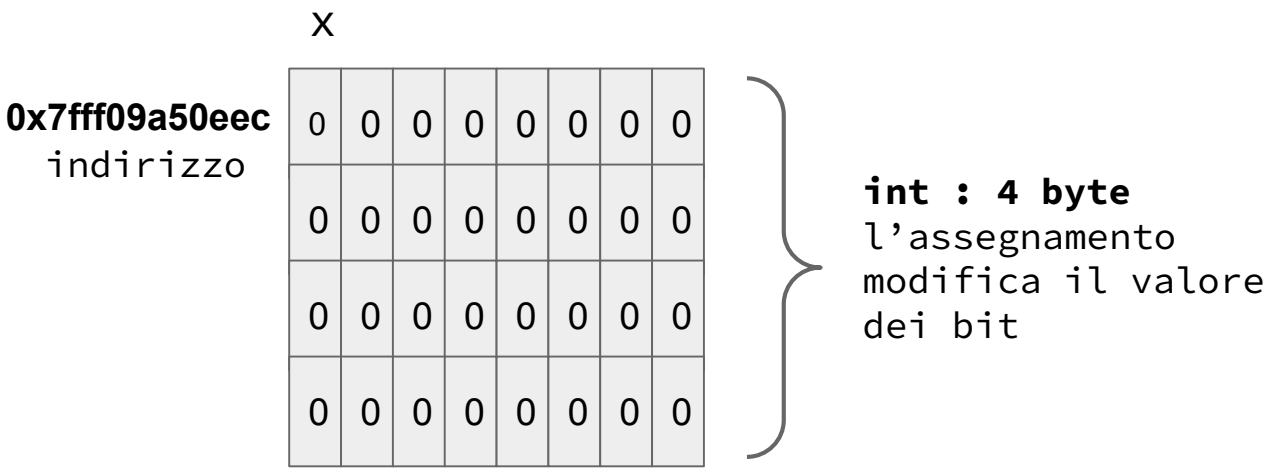


&x è 0x7fff09a50eec

NELLA MEMORIA PRINCIPALE

Variabile

```
int x;

x = 0;

x = x+1;
```

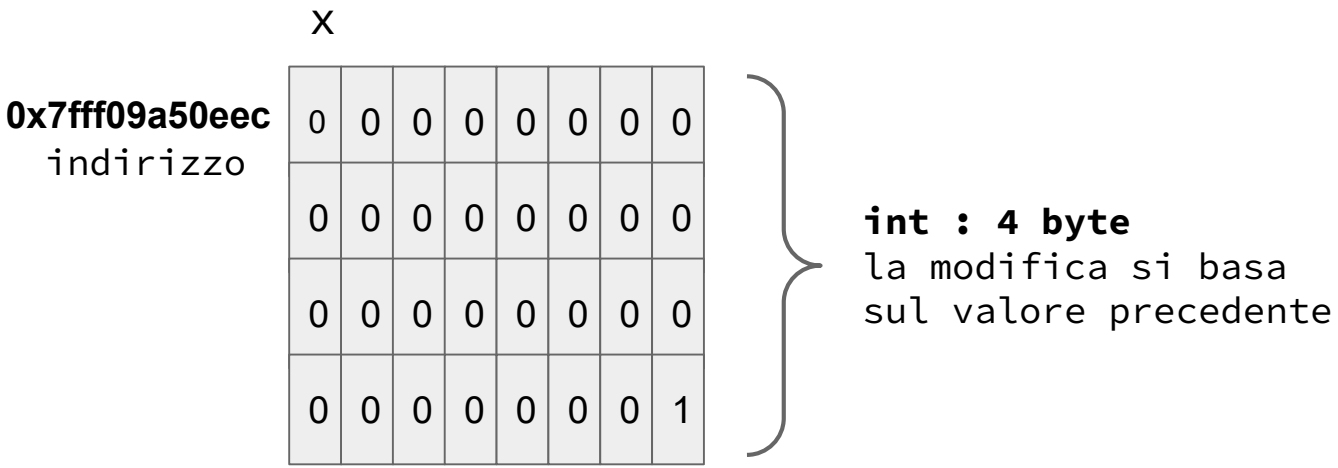


NELLA MEMORIA PRINCIPALE

Nota: **&x** è sempre 0x7fff09a50eec!

Variabile

```
int x;

x = 0;

x = x+1;
```



NELLA MEMORIA PRINCIPALE

Nota: **&x** è sempre 0x7fff09a50eec!

Operatori & e *

& vrb	indica l'indirizzo della variabile vrb
*** indexp**	l'operatore * dereferenzia un'espressione, cioè permette di accedere al contenuto della cella che ha l'indirizzo calcolato dall'espressione indexp
*** (& vrb)**	come caso particolare, indica il contenuto della variabile che ha per indirizzo &vrb, ovvero equivale a vrb

Struct

Strutture

Collezione di dati correlati che possono essere disomogenei

LIBRO:

titolo: stringa

num. pagine: intero

autore: stringa

prezzo: numero con virgola

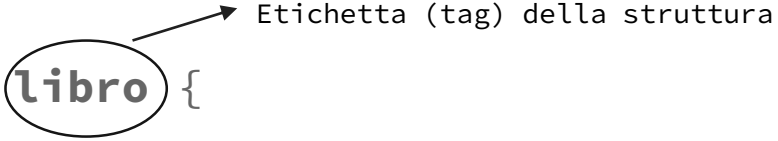
Strutture

Tipo che identifica una collezione di dati correlati che possono essere disomogenei

```
struct {
            char titolo[MAXT];
            int pagine;
            char autore[MAXN];
            float prezzo;
       };
```

Strutture

Tipo che identifica una collezione di dati correlati che possono essere disomogenei




```
struct libro {
        char titolo[MAXT];
        int pagine;
        char autore[MAXN];
        float prezzo;
};
```

QUESTO È UN PROTOTIPO DI STRUTTURA. NON HA ALLOCATA MEMORIA.

IN SEGUITO E’ POSSIBILE DICHIARARE VARIABILI DI TIPO “struct libro”

Strutture

```
struct libro {
        char titolo[MAXT];
        int pagine;
        char autore[MAXN];
        float prezzo;
    };

struct libro L;
```

QUESTO È UN PROTOTIPO DI STRUTTURA

NON HA ALLOCATA MEMORIA

DICHIARAZIONE DI VARIABILI DI TIPO “struct libro”

Strutture – accesso ai campi

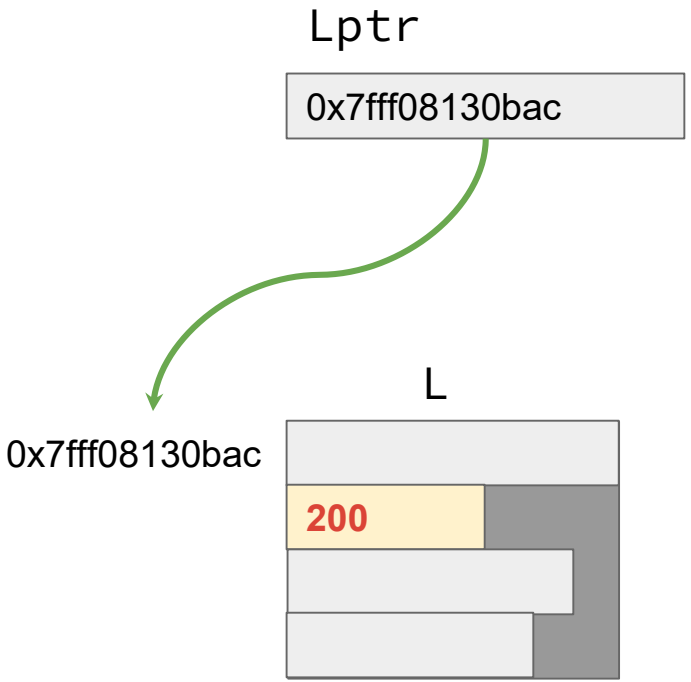
```
struct libro L;
struct libro* Lptr = &L;
```

OPERATORE .

L.pagine = 200;

OPERATORE ->

ptrL->pagine = 200;



Nota: L'operatore -> permette di accedere più semplicemente alla struct noto il suo puntatore, risparmiando la scrittura (equivalente): (*ptrL).pagine = 200

Strutture – utilizzo con le funzioni

- Le variabili `struct` vengono passate per valore! (come se passassi una variabile di tipo primitivo, es: `int`)
- I membri della struct sono passati nel modo in cui verrebbero passate variabili dello stesso tipo
  - Pertanto la funzione chiamata non modifica la struct nel chiamante!
  - Per modificare la struct bisogna passarla per riferimento (&)
- Passare le strutture per riferimento è più efficiente (non occorre la copiatura dell'intera struttura)
- Si può sfruttare questa peculiarità delle strutture per passare ad una funzione un array per valore (se ciò è desiderato), "inpacchettandolo" in una struttura!

Strutture – utilizzo con le funzioni

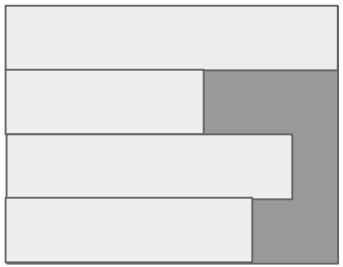
```
struct libro {
    char titolo[MAXT];
    int pagine;
    char autore[MAXN];
    float prezzo;
};
```

Esempio:

```
void stampalibro(struct libro L){
      // richiede la copia di sizeof(struct libro) bytes!
}
void stampalibro(struct libro* Lptr){
      // molto più efficiente
}
```

Strutture in memoria – padding

```
struct libro {
                char titolo[MAXT];
                int pagine;
                char autore[MAXN];
                float prezzo;
        };

struct libro L;
```



I campi hanno dimensione diversa

Sono allocati in ordine di specifica

I sistemi operativi non allocano la memoria a byte ma a gruppi di 4, 8, 16 … byte

Una struct può avere allocata un po' più memoria dello stretto necessario (**padding**)

Conseguenze del padding

```
struct prova {
    char x;
    char y;
    int  z;
} P1, P2;
```

```
sizeof(struct prova) = 8

sizeof(char)*2 + sizeof(int) = 6
```

```
P1 = P2; // OK
```

```
if (P1 == P2) …   // NON SI PUÒ FARE !   Perché?
```

Strutture autoreferenziali

Una struct non può contenere un membro del proprio tipo di struct, ma può contenere un puntatore a quel tipo di struct.

Ad esempio:

```
struct employee {
        char firstName[20];
        char lastName[20];
        struct employee manager;
};
```

// NON SI PUÒ FARE ! Perché?

```
struct employee {
        char firstName[20];
        char lastName[20];
        struct employee* managerPtr;
};
```

// OK !

struct employee è detta struct autoreferenziale (self-referential struct)

Esempio più complesso

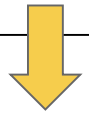
```
/* dichiarazione nutrizionale di un alimento */

struct dich_nutriz {
   int        energia;
   float      grassi;
   float      carboidrati;
   float      fibre:
   float      proteine;
   float      sale;
}
```

Esempio più complesso

Posso costruire tipi di dati/variabili complessi/e, es.

```
struct cibo {
    char nome[MAX];
    struct dich_nutriz tabella;
} dispensa[MAXD];
```

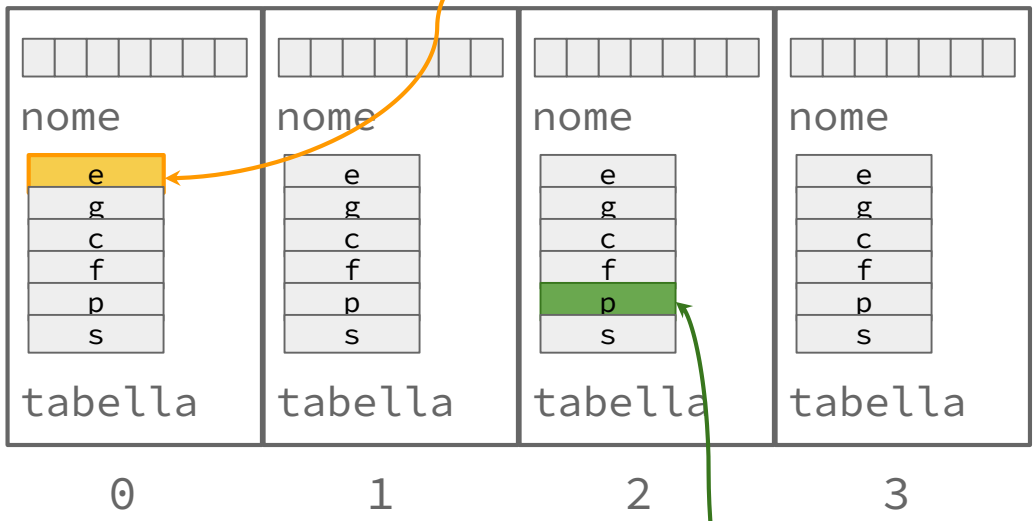
in dispensa ci sono tanti cibi, mantengo associata a ciascuno la sua dichiarazione nutrizionale (es. per applicazione dietologica)



array di struct, ognuna comprende un nome e una dichiarazione nutrizionale (struct annidata)

Esempio più complesso

dispensa[0].tabella.energia



uso la conoscenza della struttura per accedere all'informazione che mi serve

dispensa[2].tabella.proteine

TYPEDEF

TYPEDEF

- Consente di creare sinonimi (o alias) per tipi precedentemente definiti
- Spesso (ma non solo) usato con le struct, esempio:

Nota: la struct tag non è più necessaria

```
typedef struct {
        char titolo[MAXT];
        int pagine;
        char autore[MAXN];
        float prezzo;
} Libro; // nuovo tipo “Libro”

Libro L1, L2;    // variabili di tipo “Libro”
```

TYPEDEF

- Per convenzione, si scrive in Maiuscolo la prima lettera dei nomi dei tipi che sono sinonimi di altri tipi
- Altro es:

```
typedef char* Stringa;
```

- Dopodichè è permesso scrivere ad esempio:

```
Stringa s;

Stringa doc[MAXLINES];

int strcmp(Stringa, Stringa);
```

TYPEDEF

- Anticipazione sul Lab 2: useremo la struct:

```
/** Un tipo di dato per i contatti telefonici e
cyberspaziali */

typedef struct {

        char* name;
        char* surname;
        char* mobile;
        char* url;
} Contact, *ContactPtr
```

- non vi spaventate quando vedrete:

```
/** dealloca un contatto */
void dsContact(ContactPtr* p);   /* cosa è p?
```

ENUM

enum

- Permette di definire una lista di valori interi costanti convenientemente rinominati con etichette (univoche)
- Ad esempio:

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL,
AUG, SEP, OCT, NOV, DEC };

/* JAN è 1, FEB è 2, e così via */
```

- Convenzionalmente si scrivono le etichette in maiuscolo
- Sono un mezzo efficiente (preferibile alle #define) per associare valori interi a dei nomi

ENUM

- Posso anche combinare typedef e enum:

```
typedef enum outcome {OUTOFMEMORY = -1,
ALREADYPRESENT, INSERTED } Outcome;
```

- Outcome ora è un intero che può valere -1,0,1

```
Outcome operazione(…) {
        …
        return OUTOFMEMORY;
}
```

UNION

Unioni: cosa sono?

- Variabili simili alle strutture, ma che in un dato momento contengono uno solo dei membri specificati entro le parentesi graffe {}
- Sintassi identica a quella per le strutture, sostituendo la keyword struct con la keyword union:

```
union myunion {
        char cval;
        int ival;
        float fval;
} u;
```

```
LA VARIABILE u PUO' CONTENERE:
UN CHAR DI NOME cval
          OPPURE
UN INTERO DI NOME ival
          OPPURE
UN FLOAT DI NOME fval
```

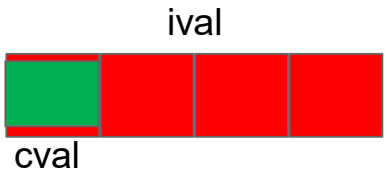
Unioni: cosa sono?

- Attenzione alla differenza tra strutture e unioni:

```
struct mystruct {
        char cval;
        int ival;
} s;
```

```
union myunion {
        char cval;
        int ival;
} u;
```

- In memoria:



Unioni: sintassi

- Stessa sintassi e operazioni delle strutture:

```
union myunion {
        char cval;
        int ival;
        float fval;
} u, *ptru;

u.cval = 'h';

u.ival = 0xDE4D7E91;      /* sostituisce cval! */

ptru = &u;

ptru->fval = 3.14;      /* sostituisce ival! */
```

Unioni e strutture annidate

- Nota: come membro di una unione potrei avere una struttura!

```
union name1 {

    struct name2 {
       int     i;
       float   f;
    } svar;

    double d;

} uvar;


uvar.svar.i = 1;
```

Strutture e unioni annidate

- Similmente, una unione può apparire in una struttura:

```
struct {

    int flags;
    char *name;
    int utype;

    union {
        int ival;
        float fval;
        char *sval;
    } u;

} symtab[NSYM];
```

Questo è un esempio realistico di uso delle unioni per realizzare una "tabella di simboli"

In questo esempio, secondo voi cosa è

```
*symtab[i].u.sval?
```

Unioni: cosa contengono in un dato istante?

- E’ responsabilità del programmatore tenere traccia di quale sia il tipo attualmente registrato in una unione
- Un approccio comune è quello di memorizzare in una variabile il tipo correntemente memorizzato nella unione, tramite ad esempio un piccolo intero con valori definiti tramite enumerazione

```
enum Union_Tag { IS_INT, IS_CHAR };
struct TaggedUnion {
   enum Union_Tag  tag;
   union {
      int   i;
      char  c;
   } data;
};
```

SI PARLA IN QUESTO CASO DI UNA "UNIONE ETICHETTATA" (TAGGED UNION)

Unioni: esperimento pratico

- Scrivere un semplice programma di test, unione.c, copiando le dichiarazioni qui sotto
- Scrivere una funzione che, dato un puntatore ad una unione etichettata, stampa opportunamente il valore in essa contenuto
- Scrivere un semplice main per provarne il funzionamento
- Cosa succede se si prova a estrarre da una unione un tipo sbagliato?

```
enum Union_Tag { IS_INT, IS_CHAR };

struct TaggedUnion {
   enum Union_Tag  tag;
   union {
      int   i;
      char  c;
   } data;
};
```

Unioni: per riassumere

- Una unione è in pratica una struttura nella quale tutti i membri hanno spiazzamento nullo rispetto alla base, e la struttura è sufficientemente grande da contenere il membro più ampio, garantendo il corretto allineamento in memoria per tutti i tipi presenti nella union

- Sono utili in **Sistemi Operativi,** ad esempio nelle primitive di allocazione dinamica di memoria, oppure nella costruzione di tabelle di simboli di tipo diverso