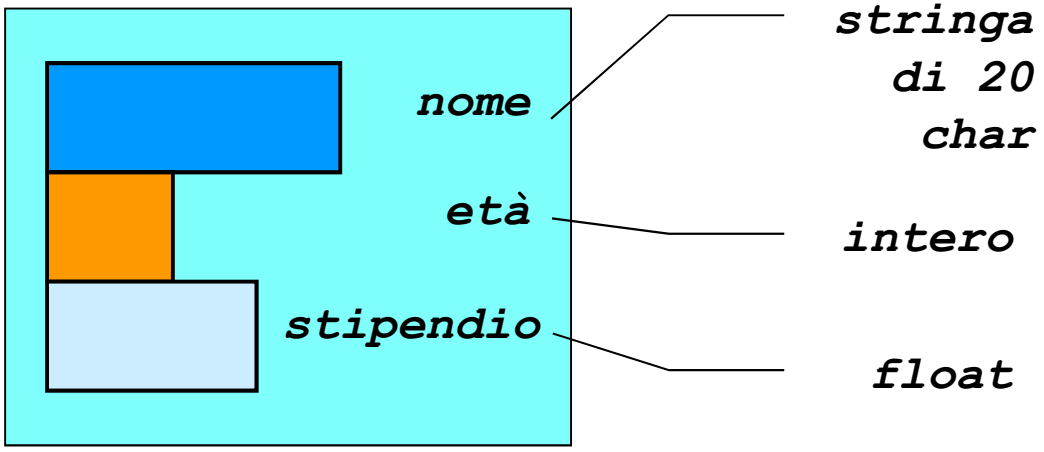
STRUTTURE

Una *struttura (record in alcuni linguaggi)* è una ***collezione finita di variabili non necessariamente dello stesso tipo***, ognuna identificata da un *nome*

`struct persona`



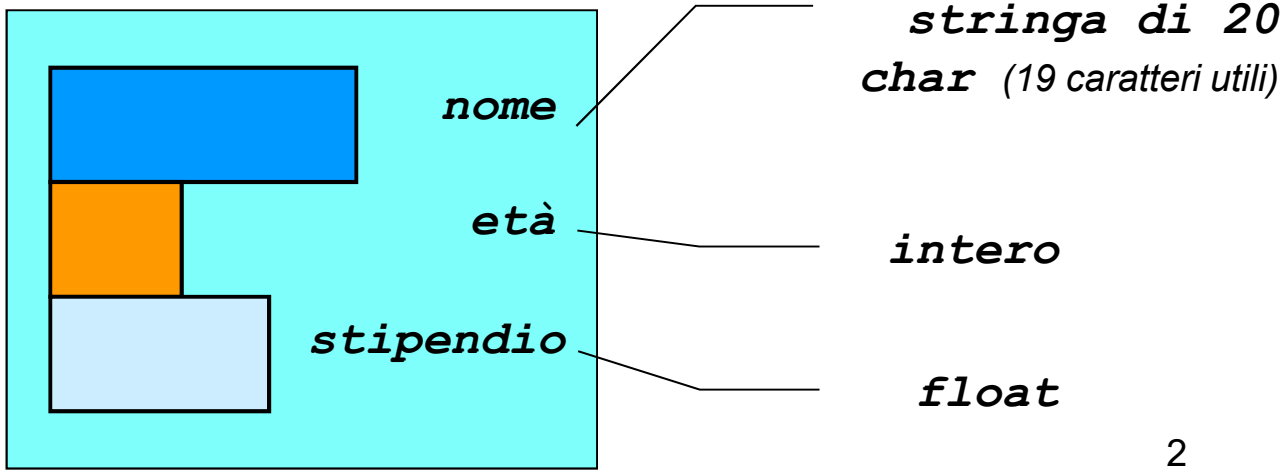

```
struct [<etichetta>] {
    { <definizione-di-variabile> }
} <nomeVariabile> ;
```

ESEMPIO

```
struct persona {
    char nome[20];
    int  eta;
    float stipendio;
} pers ;
```

Definisce una variabile *pers*

struct persona



ESEMPIO

```
struct punto {
   int  x, y;
} p1, p2 ;
```

p1 e p2 sono fatte ciascuna da due interi di nome x e y

```
struct data {
   int  giorno,mese,anno;
} d ;
```

d è fatta da tre interi di nome giorno, mese e anno

STRUTTURE

Una volta definita una variabile struttura, *si **accede ai singoli campi** mediante la **notazione puntata***

Ad esempio:

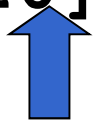
```
p1.x = 10;  p1.y = 20;
p2.x = -1;  p2.y = 12;
d.giorno = 25;
d.mese = 12;
d.anno = 1999;
```

Ogni campo si usa come una normale variabile del tipo corrispondente a quello del campo

STRUTTURE

Possiamo anche semplicemente dichiarare il tipo di dato e utilizzarlo poi successivamente

```
int main(){
  struct frutto {
     char nome[20]; int  peso;
  };
  struct frutto f2 ;

  ...
}
```



Non occorre ripetere l'elenco dei campi perché è implicito ***nell'etichetta*** `frutto`*, che è già comparsa sopra*

ESEMPIO

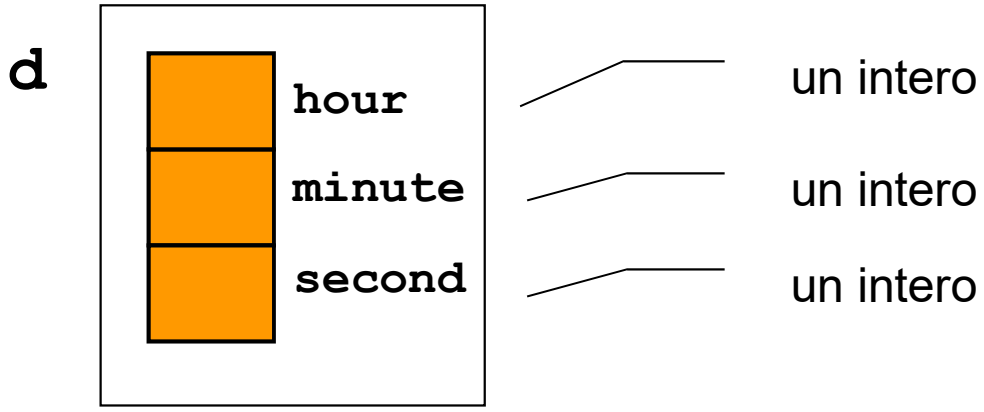
```
int main(){
  struct frutto {
     char nome[20]; int  peso;
  } f1 = {"mela", 70};
  struct frutto f2 = {"arancio", 50};

  int peso = f1.peso + f2.peso;

}
```

Non c'è alcuna ambiguità fra la variabile ***peso*** *definita nel* ***main*** *e quella definita nella* ***struct***

STRUTTURE

```
struct time
{
  int hour, minute, second;
} t ;
```

La variabile **t** è composta da tre interi di nome **hour**, **minute** e **second**



STRUTTURE

- **`time`** è solo un'etichetta, è opzionale e serve per dichiarare altre variabili dello stesso tipo
- **`struct time t1, t2;`**

  → dichiara due variabili **`t1`** e **`t2`** di tipo struttura **`time`**
- L'accesso ai campi delle strutture avviene tramite la notazione puntata:

```
t1.hour = 12;
t1.minute = 55;
t1.second = 23;
printf("It's %d:%d:%d; it's time for LUNCH!",
    t1.hour, t1.minute, t1.second);
```

STRUTTURE

- Si possono dichiarare strutture all'interno di altre strutture

```
struct punto {
      int  x, y;
  } p1, p2 ;


  struct linea {
      struct punto coord1;

      struct punto coord2;

  } l1, l2 ;
```

STRUTTURE

- In memoria i campi di una struttura sono allocati in modo contiguo e nello stesso ordine in cui sono stati dichiarati.
- L'indirizzo di memoria di una variabile di tipo struct coincide con l'indirizzo di memoria del suo primo membro.
- Possiamo verificare la dimensione in byte di una struttura utilizzando l'operator `sizeof`
- A differenza di quanto accade con gli array, *il nome della struttura rappresenta la struttura nel suo complesso*

STRUTTURE

È <u>possibile</u>:

- ***assegnare una struttura a un'altra (copia!)***
  - `f2 = f1;`
- ***che una funzione restituisca una struttura (restituzione di una copia!)***
  - struct time getNoon(void)
    {
    struct time t;
    t.hour = 12; t.minute = 0; t.second = 0;
    return t;
    }
- ***passare una struttura come parametro a una funzione (passaggio di una copia!)***

ASSEGNAMENTO TRA STRUTTURE

```
int main(){
  struct frutto {
     char nome[20]; int  peso;
  } f1 = {"mela", 70};
  struct frutto f2 = {"arancio", 50};

  f1 = f2;

}
```

Equivale a copiare ***f2.peso*** *in* ***f1.peso****,*
e ***f2.nome*** *in* ***f1.nome***

```
f1.peso=f2.peso;
strcpy(f1.nome, f2.nome);
```

STRUTTURE passate COME PARAMETRI

- Il nome della struttura rappresenta ***la struttura nel suo complesso***

- quindi, non ci sono problemi nel passare strutture come parametro a una funzione: *avviene il classico* ***passaggio per valore***
  - ***tutti i campi vengono copiati, uno per uno***

- è perciò possibile anche ***restituire come risultato*** una struttura

ESEMPIO

Tipo del valore di ritorno della funzione.

```
struct frutto macedonia(
   struct frutto f1, struct frutto f2){
      struct frutto f;
      f.peso = f1.peso + f2.peso;
      strcpy(f.nome, "macedonia");
      return f;
}
```

La funzione di libreria ***strcpy()*** *copia la costante stringa* ***"macedonia"*** *in* ***f.nome***

ESEMPIO

***PROBLEMA**: leggere le coordinate di un punto in un piano e modificarle a seconda dell'operazione richiesta*

1. *proiezione sull'asse X*
2. *proiezione sull'asse Y*
3. *traslazione di DX e DY*

Specifica:

- *leggere le coordinate di input e memorizzarle in una struttura*
- *leggere l'operazione richiesta*
- *effettuare l'operazione*
- *stampare il risultato*

ESEMPIO

```
#include <stdio.h>
int main()
{ struct punto{float  x,y;} P;
  unsigned int op;
  float  Dx, Dy;
  printf("ascissa? ");    scanf("%f",&P.x);
  printf("ordinata? ");  scanf("%f",&P.y);
  printf("%s\n","operazione(0,1,2,3)?");
  scanf("%d",&op);
  switch (op)
  {case 1: P.y=0;break;
   case 2: P.x=0; break;
   case 3: printf("%s","Traslazione?”);
           scanf("%f%f",&Dx,&Dy);
                        P.x=P.x + Dx;
                        P.y=P.y + Dy;
                 break;
   default: ;
  }
  printf("%s\n",“Le nuove coordinate sono");
  printf("%f%s%f\n",P.x,"  ",P.y);
}
```

Strutture & Array: piccolo trucco

Se una struttura, anche molto voluminosa,
viene copiata elemento per elemento...

.. perché non usare una struttura per incapsulare un array?

In effetti:

- il C non rifiuta di manipolare gli array come un tutt' uno "per principio": è solo la conseguenza del modo in cui si interpreta il loro nome
- quindi, *"chiudendoli in una struttura"…*

Strutture & Array

```
int main(){
   struct string20
   {
      char s[20];
   }

   s1 = {"Paolino Paperino" },
   s2 = {"Gastone Paperone" };

   s1 = s2;    /* FUNZIONA! */
}
```

- È fondamentale ricordare che si stanno assegnando strutture che contengono array e non array direttamente

Strutture & Array

Analogamente, adottando lo stesso “trucco”, una funzione può essere forzata a restituire un array come valore di ritorno (nota, toupper è una funzione di libreria che trasforma lettere minuscole in maiuscole).

```
struct string20 { char s[20]; } ;

struct string20 maiusc(struct string20 x)

{
   int k;
   for (k = 0; k < strlen(x.s); k++)
       x.s[k] = toupper(x.s[k]);
   return x;
}

int main()

{
 struct string20 m = {"Che bello!"}, mm;
 mm = maiusc(m);
 printf("%s", mm.s);
}
```

ESEMPIO 4

Leggere (e poi riscrivere) nome, cognome, e data di nascita di una persona

```
#include <stdio.h>
int main() {
   struct { char cognome[20], nome[20];
            int g, m, a;
   } p;
   printf("Cognome, nome e data di nascita: ");
   scanf("%s%s%d/%d/%d", p.cognome, p.nome,
                 &p.g, &p.m, &p.a);
   printf("%s %s è nato il %d/%d/%d\n",
      p.cognome, p.nome, p.g, p.m, p.a);
}
```

Gli spazi di separazione sono eliminati automaticamente

TIPI DI DATO

- Tipicamente un elaboratore è ***capace*** di trattare **domini di dati di *tipi primitivi***
  - *numeri naturali, interi, reali*
  - *caratteri* e *stringhe di caratteri*

  e quasi sempre anche collezioni di oggetti, mediante la definizione di ***tipi strutturati***
  - *array, strutture*
- Spesso un linguaggio di programmazione permette di **introdurre altri tipi *definiti dall'utente***

TIPI DEFINITI DALL'UTENTE

- In C, l'utente può introdurre ***nuovi tipi*** tramite una ***definizione di tipo***
- **La definizione associa a un identificatore (*nome del tipo*) un tipo di dato**
  - aumenta la leggibilità del programma
  - consente di ragionare per astrazioni
- Linguaggio C consente in particolare di:
  - **ridefinire tipi già esistenti**
  - **definire dei nuovi *tipi strutturati***
  - **definire dei nuovi *tipi enumerativi***

TIPI RIDEFINITI

Un nuovo identificatore di tipo viene dichiarato identico a un tipo già esistente

Schema generale:

```
typedef  TipoEsistente NuovoTipo;
```

Esempio

```
typedef     int MioIntero;
MioIntero   X,Y,Z;
int         W;
```

DEFINIZIONE DI TIPI STRUTTURATI

Abbiamo visto a suo tempo come introdurre *variabili* di tipo array e struttura:

```
char msg1[20], msg2[20];
struct persona {...} p, q;
```

Non potendo però ***dare un nome*** al nuovo tipo, dovevamo ***ripetere la definizione*** per ogni nuova variabile

- per le strutture potevamo evitare di ripetere la parte fra {..}, ma **`struct persona`** andava ripetuto comunque

DEFINIZIONE DI TIPI STRUTTURATI

Ora siamo in grado di ***definire nuovi tipi*** **array e struttura**:

```
typedef char string[20];
typedef struct {... } persona;
```

Ciò consente di *non dover più ripetere la definizione per esteso* ogni volta che si definisce una nuova variabile:

```
string s1, s2;  /* due stringhe di 20 caratteri */
persona p1, p2;  /* due strutture "persona" */
```

- per le strutture, ciò rende ***quasi sempre inutile specificare etichetta*** dopo parola chiave **`struct`**

Typdef

- Nota che **`typedef`** cambia la semantica nelle dichiarazioni/definizioni
- **`time`** risulta una variabile se

```
struct
{
  int hour, minute, second;
} time ;
```

- **`time`** risulta un tipo di dato se

```
typedef
{
  int hour, minute, second;
} time ;
```

Esercizio

- Sia data la struttura

```
struct time
{
  int hour, minute, second;
};
```

- Si progetti una funzione in grado di calcolare la differenza fra due strutture **`time`** e che restituisca il risultato in termini di una nuova struttura **`time`**

Esercizio

- Per semplicità si può definire il tipo Time

```
typedef struct time Time;
```

- L'interfaccia della funzione è facilmente desumibile dalle specifiche:

```
Time subtract(Time t1, Time t2);
```

- Due possibili approcci:
  1. Trasformare in secondi, eseguire la differenza, trasformare in ore, minuti, secondi
  2. Eseguire la sottrazione direttamente tenendo conto dei riporti

Esercizio

```
Time subtract1(Time t1, Time t2)
{
    int s1, s2, sResult;
    Time result;

    s1 = t1.hour * 3600 + t1.minute * 60 + t1.second;
    s2 = t2.hour * 3600 + t2.minute * 60 + t2.second;
    sResult = s1 - s2;

    result.hour = sResult / 3600;
    sResult = sResult % 3600;
    result.minute = sResult / 60;
    sResult = sResult % 60;
    result.second = sResult;

    return result;
}
```

Esercizio

```
Time subtract2(Time t1, Time t2)
{
    Time result;
    int carry;
    result.second = t1.second - t2.second;
    carry = 0;
    if (result.second < 0)
    {
        result.second = 60 + result.second;
        carry = -1;
    }
    result.minute = t1.minute - t2.minute + carry;
    carry = 0;
    if (result.minute < 0)
    {
        result.minute = 60 + result.minute;
        carry = -1;
    }
    result.hour = t1.hour - t2.hour + carry;
    return result;
}
```

Strutture innestate

- Ovviamente (?) non ci sono problemi ad ***innestare strutture in altre strutture***
- Ad esempio si può pensare di avere una struttura *address* contenuta nella struttura *person*
- Come esercizio si può pensare di fornire alcune funzioni (servizi) che consentano di operare in modo agevole con le strutture di cui sopra
- Per cominciare:
  - Operazioni di lettura da console
  - Operazioni di formattazione su stringa

Person & Address – Definizioni

```
typedef struct addressStruct
{
    char street[80];
    char postalCode[8];
    char city[30];
    char state[20];
} Address;
```

```
typedef struct personStruct
{
    char firstName[50];
    char secondName[50];
    char phone[18];
    char cell[18];
    Address address;
} Person;

#define PERSONARRAYDIM 100

typedef Person
  PersonArray[PERSONARRAYDIM
  ];
```

Person & Address – Ricerca Esatta

Ricerca di un contatto per cognome (first name)

- Problema facile e già visto
- Se i contatti sono:
  - ordinati → ricerca binaria
  - non ordinati → ricerca lineare
- Per semplicità si implementa la ricerca lineare...
- Si può utilizzare **`strcmp()`** come funzione di confronto fra stringhe...

Person & Address – Ricerca Esatta

In ingresso:

- cognome da cercare
- array in cui cercare
- numero di strutture effettivamente presenti nell'array

```
int findExactByFirstName(char firstName[50],
      PersonArray persons, int dim)
{
    int i;
    for (i = 0; i < dim; i++)
        if (strcmp(persons[i].firstName, firstName) ==
  0)
            return i;
    return -1;
}
```