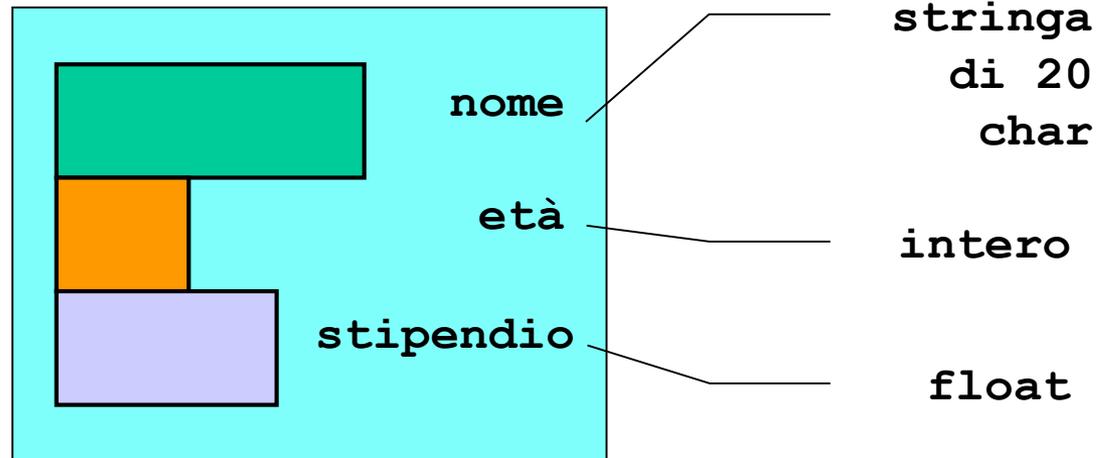


# STRUTTURE

- Una *struttura* è una collezione finita di variabili non necessariamente dello stesso tipo, ognuna identificata da un *nome*.

**struct  
persona**



# STRUTTURE

---

Definizione di una *variabile* di tipo *struttura*:

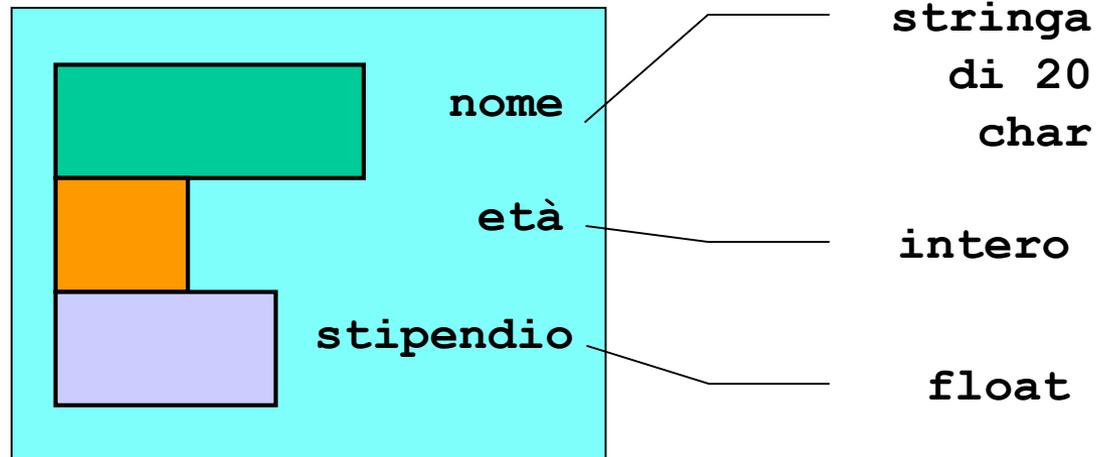
```
struct [<etichetta>] {  
    { <definizione-di-variabile> }  
} <nomeStruttura> ;
```

# ESEMPIO

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

Definisce una variabile **pers** strutturata nel modo illustrato.

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 al tipo del campo.

# STRUTTURE

---

```
void main() {  
    struct persona{  
        char nome[20];  
        int  eta;  
        float stipendio;  
    } P1;  
    struct persona P2 ;  
    ...  
}
```



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

# ESEMPIO

---

```
void main() {
    struct persona {
        char nome[20];
        int  eta;
        float stipendio;
    } P1 ;
    struct persona P2;
    float sommastip;
    printf("Inserire nome, eta' e stipendio di P1);
    scanf("%s %d %f", P1.nome, &P1.eta, &P1.stipendio);

    printf("Inserire nome, eta' e stipendio di P2);
    scanf("%s %d %f", P2.nome, &P2.eta, &P2.stipendio);

    sommastip = P1.stipendio + P2.stipendio;

    printf("Lo stipendio totale e' %f", sommastip);
}
```

Non c'è alcuna ambiguità perché ogni variabile di nome **stipendio** è definita nella propria struct.

# STRUTTURE

---

- A differenza di quanto accade con gli array, il nome della struttura rappresenta la struttura nel suo complesso.

Quindi, è possibile:

- assegnare una struttura a un'altra ( $P2 = P1$ )
- che una funzione restituisca una struttura

E soprattutto:

- passare una struttura come parametro a una funzione significa passare una copia

# ASSEGNAZIONE TRA STRUTTURE

---

```
void main() {  
    struct persona {  
        char nome[20];  
        int  eta;  
        float stipendio;  
    } P1 ;  
    struct persona P2;  
    ...  
    P2 = P1;  
}
```

Equivale a

```
strcpy (P2.nome, P1.nome) ;  
P2.eta= P1.eta;  
P2.stipendio= P1.stipendio;
```

# STRUTTURE PASSATE COME PARAMETRI

---

- Il nome della struttura rappresenta, come è naturale, *la struttura nel suo complesso*
- quindi, non ci sono problemi nel passarle a 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

---

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

Tipo del valore di ritorno della funzione.



```
struct punto mezzo (  
    struct punto P1, struct punto P2) {  
    struct punto P;  
    P.x = (P1.x + P2.x) / 2;  
    P.y = (P1.y + P2.y) / 2;  
    return P;  
}
```

# ESEMPIO

---

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

proiezione sull'asse X

proiezione sull'asse Y

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

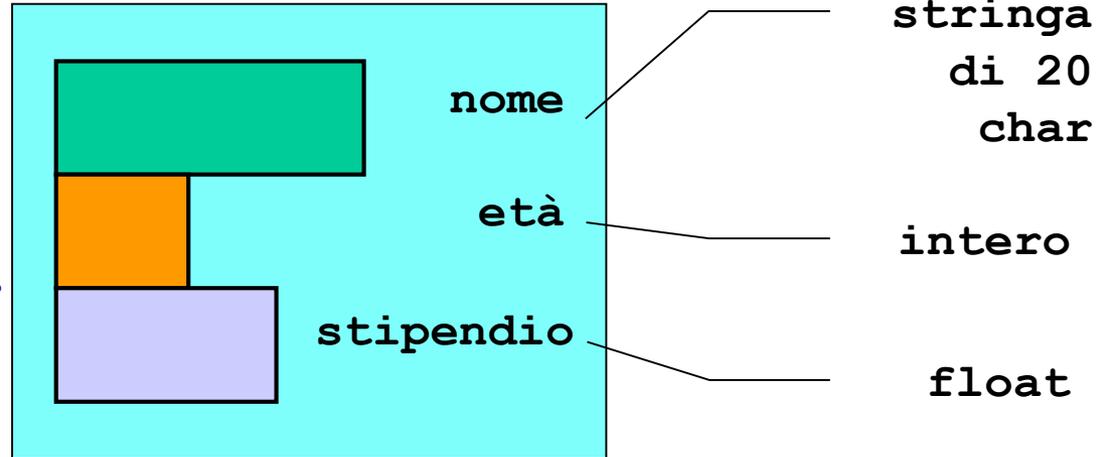
# ESEMPIO

---

```
#include <stdio.h>
void 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","nuove coordinate sono");
printf("%f%s%f\n",P.x,"  ",P.y);
}
```

# ALLINEAMENTO IN MEMORIA

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



**In memoria il compilatore le organizza in ordine di dichiarazione!**

# ALLINEAMENTO IN MEMORIA

---

```
struct prova1{  
    char ch1;  
    char ch2;  
    short s;  
    double d;  
    int i;  
};
```

ch1	ch2	s	s	x	x	x	x
d	d	d	d	d	d	d	d
i	i	i	i	x	x	x	x

**3 \* 8 BYTE**

**In memoria il compilatore le organizza in ordine di dichiarazione!**

# ALLINEAMENTO IN MEMORIA

---

```
struct prova2{  
    double d;  
    int i;  
    short s;  
    char ch1;  
    char ch2;  
};
```

d	d	d	d	d	d	d	d
i	i	i	i	s	s	ch1	ch2

**2 \* 8 BYTE**

**In memoria il compilatore le organizza in ordine di dichiarazione!**

# ALLINEAMENTO IN MEMORIA

---

```
struct prova1{  
    char ch1;  
    short s;  
    char ch2;  
    double d;  
    int i;  
};
```

ch1	x	s	s	ch2	x	x	x
d	d	d	d	d	d	d	d
i	i	i	i	x	x	x	x

**3 \* 8 BYTE**

**L'allineamento dipende dal contesto (S.O. ,  
compilatore, Hardware..)**

# STRUTTURE - PUNTATORI

---

- Una volta definito un puntatore ad una struttura, *si accede ai singoli campi mediante la notazione a freccia.*

Ad esempio:

```
struct punto *p;
```

```
p1->x = 10;   p1->y = 20;
```

```
p2->x = -1;   p2->y = 12;
```

Ogni campo si usa come una normale variabile del tipo corrispondente al tipo del campo.

# ESEMPIO

---

In C è possibile definire NUOVI TIPI.

```
typedef <descrizione_nuovo_tipo> NEWT;
```

- Si usa la parola chiave **typedef**
- La dichiarazione associa a un nuovo tipo di dato non primitivo un identificatore **NEWT**
- Le caratteristiche di nuovo tipo sono indicate in **<descrizione\_nuovo\_tipo>**

# ESEMPIO

---

```
typedef int NewInt; /* NewInt è un tipo  
                    non primitivo che  
                    ridefinisce int */
```

```
NewInt X; /* X è di tipo NewInt */  
int Y;    /* Y è di tipo int */
```

Molto comodo per le strutture

# ESEMPIO

---

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

Struct persona è un tipo.

```
void main() {

    struct persona P1, P2;
    float sommastip;
    printf("Inserire nome, eta' e stipendio di P1);
    scanf("%s %d %f", P1.nome, &P1.eta, &P1.stipendio);

    printf("Inserire nome, eta' e stipendio di P2);
    scanf("%s %d %f", P2.nome, &P2.eta, &P2.stipendio);

    sommastip = P1.stipendio + P2.stipendio;

    printf("Lo stipendio totale e' %f", sommastip);
}
```

# ESEMPIO

---

```
typedef struct {  
    char nome[20];  
    int  eta;  
    float stipendio;  
} impiegato;
```

Da ora in poi, impiegato è un nuovo TIPO definito dall'utente.

```
void main() {  
    impiegato P1, P2;  
    float sommastip;  
    printf("Inserire nome, eta' e stipendio di P1);  
    scanf("%s %d %f", P1.nome, &P1.eta, &P1.stipendio);  
  
    printf("Inserire nome, eta' e stipendio di P2);  
    scanf("%s %d %f", P2.nome, &P2.eta, &P2.stipendio);  
  
    sommastip = P1.stipendio + P2.stipendio;  
  
    printf("Lo stipendio totale e' %f", sommastip);  
}
```