

TIPI DI DATO

Un **tipo di dato** T è definito come:

- un **dominio di valori**, D
- un **insieme di funzioni** F_1, \dots, F_n sul dominio D
- un **insieme di predicati** P_1, \dots, P_m sul dominio D

$$T = \{ D, \{F_1, \dots, F_n\}, \{P_1, \dots, P_m\} \}$$

1

TIPI DI DATO: ESEMPIO

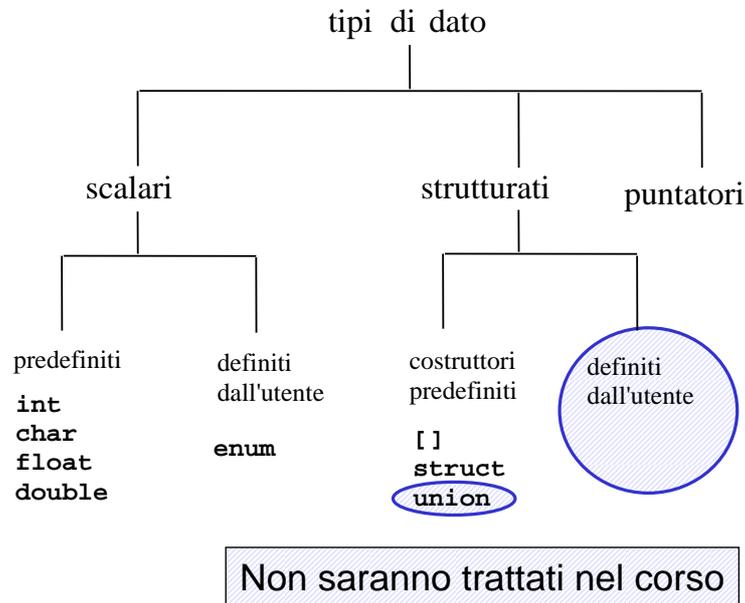
Il **tipo di dato** **INTERO** è definito come:

- un **dominio di valori**, Z
- un **insieme di funzioni** F_1, \dots, F_n sul dominio D
ad esempio, SOMMA, SOTTRAZIONE, PRODOTTO
- un **insieme di predicati** P_1, \dots, P_m sul dominio D
ad esempio, MAGGIORE, MINORE, UGUALE...

2

TIPI DI DATO

I tipi di dato si differenziano in *scalari* e *strutturati*



3

TIPI DI DATO

In C si possono *definire tipi strutturati*

Vi sono due *costruttori* fondamentali:

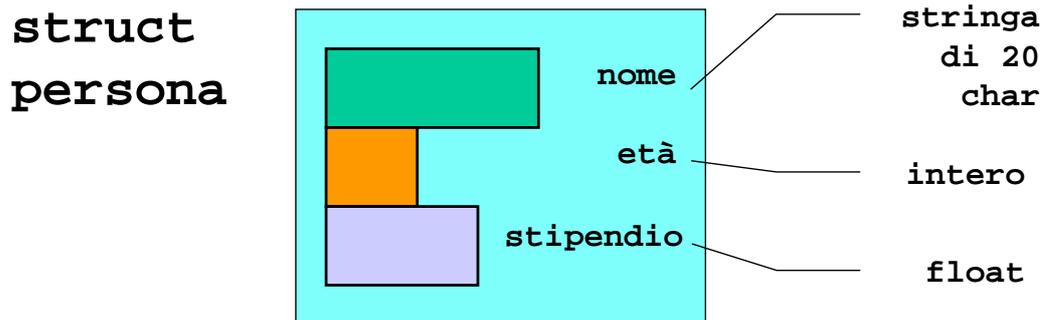
[] (array)

struct (strutture)

4

STRUTTURE

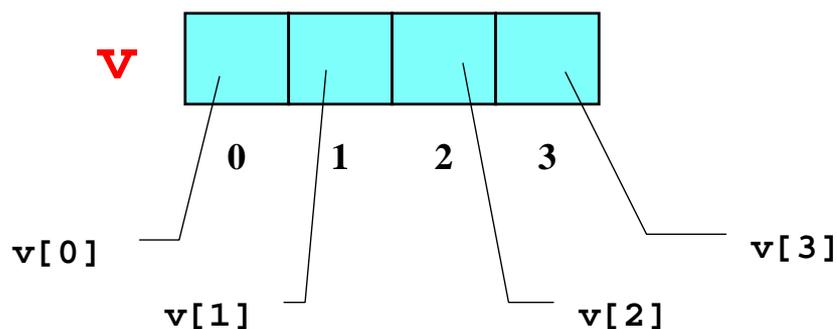
Una *struttura* è una collezione finita di variabili non necessariamente dello stesso tipo, ognuna identificata da un *nome*



5

ARRAY (VETTORI)

Un *array* è una collezione finita di N variabili dello stesso tipo, ognuna identificata da un *indice compreso fra 0 e N-1*



6

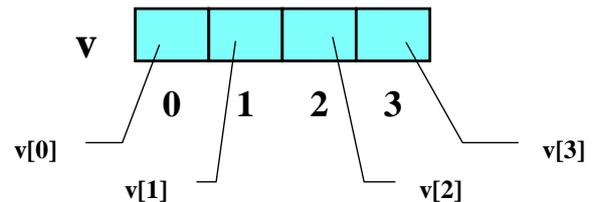
ARRAY (VETTORI)

Definizione di una *variabile* di tipo *array*:

`<tipo> <nomeArray> [<costante>] ;`

Esempi:

```
int v[4];  
char nome[20];
```



ATTENZIONE: Sbagliato

```
int N;  
char nome[N];
```

Il compilatore non sa quanta memoria allocare per l'array

7

ESEMPIO

Problema: leggere da tastiera gli elementi di un vettore

```
#include <stdio.h>  
#define N 3  
  
main()  
{ int k;  
  int A[N];  
  
  for(k=0; k < N; k++)  
  {printf("Dammi elemento %d: ", k);  
   scanf("%d", &A[k]);  
  }  
}
```

8

ESEMPIO

Problema: inizializzare un vettore con il prodotto degli indici posizionali dei suoi elementi

```
#include <stdio.h>
#define N 3

main()
{ int i=0;
  int A[N];

  while (i<N)
  {
    A[i]=i*i; /*gli elementi del vettore sono 0,1,4*/
    i++;
  }
}
```

9

ESEMPIO

Problema: scrivere un programma che, dato un vettore di N interi, ne determini il valore massimo

Specifica di I livello:

Inizialmente, si assuma come **massimo di tentativo** il **primo** elemento $m_0 = v[0] \rightarrow m_0 \geq v[0]$

Poi, si **confronti il massimo di tentativo con gli elementi** del vettore: nel caso se ne trovi uno **maggiore** del massimo di tentativo attuale, si **aggiorni il valore del massimo di tentativo**

$m_i = \max(m_{i-1}, v[i]) \rightarrow m_i \geq v[0], v[1], \dots, v[i]$

Al termine, il valore del massimo di tentativo coincide col valore massimo ospitato nel vettore $m_{n-1} \geq v[0], v[1] \dots v[n-1]$

10

ESEMPIO

Codifica:

```
#define DIM 4
main() {
    int v[DIM] = {43,12,7,86};
    int i, max=v[0];
    for (i=1; i<DIM; i++)
        if (v[i]>max) max = v[i];
    /* ora max contiene il massimo */
}
```

Espressione di
inizializzazione
di un array

11

ESEMPIO

Codifica:

```
#define DIM 4
main() {
    int v[] = {43,12,7,86};
    int i, max=v[0];
    for (i=1; i<DIM; i++)
        if (v[i]>max) max = v[i];
    /* ora max contiene il massimo */
}
```

Se vi è una *inizializzazione esplicita*, la dimensione dell'array *può essere omessa*

12

ESEMPIO

Anziché inizializzare l'array a priori, calcoliamo i valori iniziali come parte dell'algoritmo

```
#define DIM 4
main() {
    int i, max, v[DIM];
    for (i=0; i<DIM; i++) v[i]=i+1;
    max=v[0];
    for (i=1; i<DIM; i++)
        if (v[i]>max) max=v[i];
}
```

Nessuna inizializzazione

Calcolo valori iniziali

13

DIMENSIONE FISICA VS. LOGICA

Un array è una **collezione finita di N celle dello stesso tipo**

- Questo non significa che si debbano per forza *usare sempre tutte*. La **dimensione logica** di un array può essere **inferiore** alla sua **dimensione fisica**
- Spesso, la *porzione di array* realmente utilizzata *dipende dai dati d'ingresso*

14

DIMENSIONE FISICA VS. LOGICA

Esempio

È data una serie di rilevazioni di temperature espresse in gradi Kelvin

Ogni serie è composta al più da 10 valori, *ma può essere più corta*. Il valore “-1” indica che la serie delle temperature è finita

Scrivere un programma che, data una serie di temperature memorizzata in un vettore, ***calcoli la media delle temperature*** fornite

15

ESEMPIO

Il vettore deve essere ***dimensionato per 10 celle*** (caso peggiore) ma la porzione realmente usata ***può essere minore***

Specifiche di I livello:

- calcolare la somma di tutti gli elementi del vettore, e nel frattempo contare quanti sono
- il risultato è il rapporto fra la somma degli elementi così calcolata e il numero degli elementi

16

ESEMPIO

Specifica di II livello:

Inizialmente, poni uguale a 0 una variabile S che rappresenti la somma corrente, e poni uguale a 0 un indice K che rappresenti l'elemento corrente

$$s_0 = 0, k_0 = 0$$

A ogni passo, aggiungi l'elemento corrente a una variabile S che funga da somma

$$s_k = s_{k-1} + v[k],$$
$$k_{k+1} = k_k + 1, \quad k < N$$

Al termine (quando un elemento vale -1 oppure hai esaminato N elementi), l'indice K rappresenta il numero totale di elementi: il risultato è il rapporto S/K

$$s_{N-1} = s_{N-2} + v[N-1],$$
$$k_N = N$$

17

ESEMPIO

Codifica:

```
#define DIM 10
main() {
    int k, v[DIM] = {273, 340, 467, -1};
    int media, s=0;
    for (k=0; (k<DIM) && (v[k]>=0); k++)
        s += v[k];
    media = s / k;
}
```

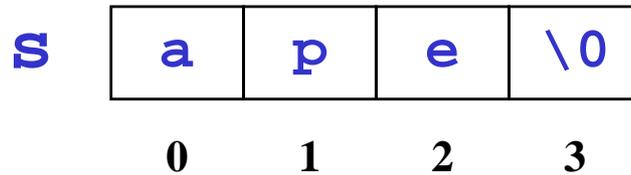
Dimensione fisica = 10

Dimensione logica = 4

Condizione di prosecuzione del ciclo: la serie di dati non è finita ($v[k] \geq 0$) e ci sono ancora altre celle nell'array ($k < DIM$)

STRINGHE: ARRAY DI CARATTERI

Una **stringa** di caratteri in C è un **array di caratteri terminato dal carattere ' \0 '**

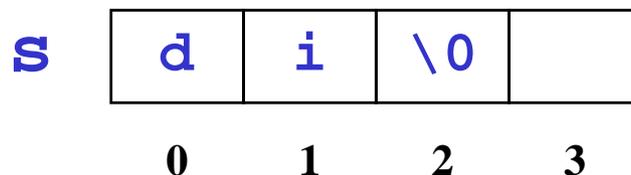


Un vettore di N caratteri può dunque ospitare stringhe **lunghe al più N-1 caratteri**, perché una cella è destinata al **terminatore ' \0 '**

19

STRINGHE: ARRAY DI CARATTERI

Un array di N caratteri può essere usato per memorizzare **anche stringhe più corte di N-1**



In questo caso, **le celle oltre la k-esima** (k essendo la lunghezza della stringa) sono **logicamente vuote**: sono inutilizzate e contengono un valore casuale

20

STRINGHE

Una stringa si può *inizializzare*, come ogni altro array, elencando le singole componenti:

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

oppure anche, più brevemente, *con la forma compatta* seguente:

```
char s[4] = "ape" ;
```

Il carattere di terminazione `'\0'` è *automaticamente incluso* in fondo
Attenzione alla lunghezza

21

STRINGHE: LETTURA E SCRITTURA

Una stringa si può *leggere da tastiera e stampare*, come ogni altro array, elencando le singole componenti:

```
... char str[4]; int i;  
    for (i=0; i < 3; i++)  
        scanf("%c", &str[i]); str[3] = '\0'  
...
```

oppure anche, più brevemente, *con la forma compatta* seguente:

```
...char str[4]; scanf("%s", str);
```

22

ESEMPIO 1

Problema:

Date due stringhe di caratteri, decidere quale precede l'altra in ordine alfabetico

Rappresentazione dell'informazione:

- poiché vi possono essere *tre* risultati ($s1 < s2$, $s1 == s2$, $s2 < s1$), *un boolean non basta*
- Si può decidere di utilizzare:
 - *un intero (negativo, zero, positivo)*

23

ESEMPIO 1

Specifica:

- scandire uno a uno gli elementi **di equal posizione** delle due stringhe, **o fino alla fine delle stringhe, o fino a che se ne trovano due diversi**
 - *nel primo caso, le stringhe sono uguali*
 - *nel secondo, sono diverse*
- nel secondo caso, confrontare i due caratteri così trovati, e determinare qual è il **minore**
 - la stringa a cui appartiene tale carattere **precede l'altra**

24

ESEMPIO 1

Codifica:

```
main() {  
    char s1[] = "Maria";  
    char s2[] = "Marta";  
    int i=0, stato;  
    while (s1[i]!='\0' && s2[i]!='\0' &&  
           s1[i]==s2[i]) i++;  
    stato = s1[i]-s2[i];  
    .....  
}
```

negativo \leftrightarrow s1 precede s2
positivo \leftrightarrow s2 precede s1
zero \leftrightarrow s1 è uguale a s2

ESEMPIO 2

Problema:

Data una stringa di caratteri, copiarla in un altro array di caratteri (di lunghezza non inferiore)

Ipotesi:

La stringa è "ben formata", ossia correttamente terminata dal carattere '\0'

Specifica:

- scandire la stringa, elemento per elemento, fino a trovare il terminatore '\0' (che esiste certamente)
- *nel fare ciò, copiare l'elemento corrente nella posizione corrispondente dell'altro array*

ESEMPIO 2

Codifica: copia della stringa carattere per carattere

```
main() {  
    char s[] = "Nel mezzo del cammin di";  
    char s2[40];  
    int i=0;  
    for (i=0; s[i]!='\0'; i++)  
        s2[i] = s[i];  
    s2[i] = '\0';  
}
```

La dimensione deve essere tale da garantire che la stringa non ecceda

Al termine, occorre garantire che anche la nuova stringa sia "ben formata", inserendo esplicitamente il terminatore

27

ESEMPIO 2

Perché non fare così?

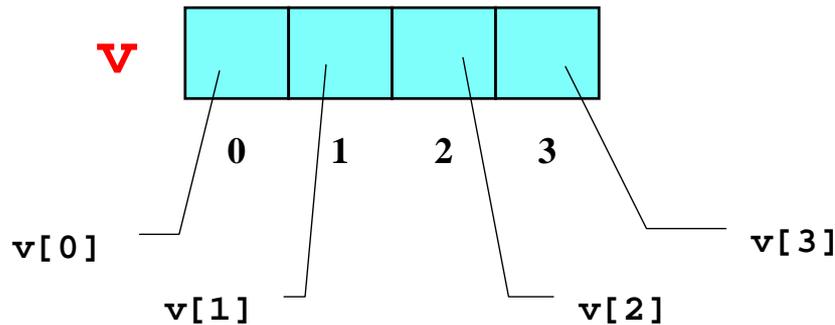
```
main() {  
    char s[] = "Nel mezzo del cammin di";  
    char s2[40];  
    s2 = s;  
}
```

ERRORE DI COMPILAZIONE:
incompatible types in assignment

**GLI ARRAY NON POSSONO
ESSERE MANIPOLATI NELLA LORO INTEREZZA**

ARRAY: IMPLEMENTAZIONE in C

Un *array* è una collezione finita di N variabili dello stesso tipo, ognuna identificata da un *indice* compreso fra 0 e N-1

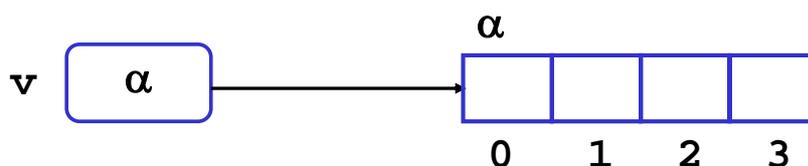


Le cose non stanno proprio così...

29

ARRAY: IMPLEMENTAZIONE in C

In C un *array* è in realtà un **puntatore che punta a un'area di memoria pre-allocata**, di ***dimensione prefissata***



Pertanto, ***il nome dell'array è un sinonimo per il suo indirizzo iniziale***: $v \equiv \&v[0] \equiv \alpha$

30

ARRAY: IMPLEMENTAZIONE in C

Il fatto che il nome dell'array non indichi l'array, ma l'indirizzo iniziale dell'area di memoria ad esso associata ha una conseguenza:

È impossibile denotare un array nella sua globalità, in qualunque contesto

Quindi NON è possibile:

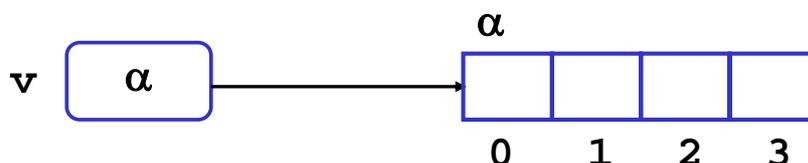
- assegnare un array a un altro ($s2 = s$)
- che una funzione restituisca un array
- passare un array come parametro a una funzione non significa affatto passare l'intero array

31

ARRAY PASSATI COME PARAMETRI

Poiché un *array* in C è un puntatore che punta a un'area di memoria pre-allocata, di dimensione prefissata, il nome dell'array:

- non rappresenta l'intero array
- è un alias per il suo indirizzo iniziale
($v \equiv \&v[0] \equiv \alpha$)



32

ARRAY PASSATI COME PARAMETRI

Quindi, *passando un array a una funzione*:

- non si passa l'intero array
- si passa solo (**per valore, come sempre**) il suo indirizzo iniziale

$(v \equiv \&v[0] \equiv \alpha)$

L'effetto finale apparente è che *l'array sia passato per riferimento*

33

RIASSUMENDO...

A livello fisico:

- il C passa i parametri **sempre e solo per valore**
- nel caso di un array, si passa il suo indirizzo iniziale $(v \equiv \&v[0] \equiv \alpha)$ perché tale è il significato del nome dell'array

A livello concettuale:

- il C passa **per valore** tutto tranne gli array, che appaiono trasferiti **per riferimento**

34

NOTAZIONE A PUNTATORI

Ma se quello che passa è solo *l'indirizzo iniziale* dell'array, che è un puntatore...

...allora ***si può adottare direttamente la notazione a puntatori nella intestazione della funzione***

In effetti, *l'una o l'altra notazione sono, a livello di linguaggio, assolutamente equivalenti*

- non cambia niente nel funzionamento
- si rende solo più evidente ciò che accade comunque

35

ESEMPIO

Problema:

Data una stringa di caratteri, ***scrivere una funzione che ne calcoli la lunghezza***

Nel caso delle stringhe, la dimensione non serve
perché può essere dedotta dalla posizione dello '\0'
Si può anche usare `lunghezza(char *s)`

Codifica:

```
int lunghezza(char s[]) {  
    int lung=0;  
    for (lung=0; s[lung]!='\0'; lung++);  
    return lung;  
}
```

36

ESEMPIO

Da così...

```
int lunghezza(char s[]) {
    int lung=0;
    for (lung=0; s[lung]!='\0'; lung++);
    return lung;
}
```

... a così:

```
int lunghezza(char *s) {
    int lung=0;
    for (lung=0; s[lung]!='\0'; lung++);
    return lung;
}
```

Oppure: *(s+lung)

37

OPERATORI DI DEREFERENCING

- L'operatore *****, applicato a un puntatore, accede alla variabile da esso puntata
- L'operatore **[]**, applicato a un nome di array e a un intero i, accede alla i-esima variabile dell'array

Sono entrambi operatori di dereferencing

***v ≡ v[0]**

38

ARITMETICA DEI PUNTATORI (1)

Oltre a $*v \equiv v[0]$, vale anche:

$*(v+1) \equiv v[1]$

...

$*(v+i) \equiv v[i]$

Gli operatori $*$ e
 $[]$ sono
intercambiabili

Espressioni della forma $p+i$ vanno sotto il nome di aritmetica dei puntatori, e denotano l'indirizzo posto i celle dopo l'indirizzo denotato da p (**celle, non byte**)

39

ARITMETICA DEI PUNTATORI (2)

Non solo sono consentite operazioni di somma fra puntatori e costanti intere ma anche:

- **Assegnamento e differenza** fra puntatori

`int *p, *q; p=q; p-q; p=p-q;`

La differenza però produce un warning

- Altre operazioni aritmetiche fra puntatori non sono consentite:

`int *p, *q; p=p*2; q=q+p;`

Le operazioni sono **corrette** se i puntatori riferiscono lo **STESSO TIPO (non tipi compatibili)**. Attenzione: comunque solo **warning** dal compilatore negli altri casi

40

ESEMPIO

Problema:

Scrivere una funzione che, dato un array di N interi, ne calcoli il massimo

Si tratta di riprendere l'esercizio già svolto, e impostare la soluzione come funzione anziché codificarla direttamente nel *main()*

Dichiarazione della funzione:

```
int findMax(int v[], int dim);
```

41

ESEMPIO

Il cliente:

```
main() {  
    int max, v[] = {43,12,7,86};  
    max = findMax(v, 4);  
}
```



Trasferire esplicitamente la dimensione dell'array è **NECESSARIO**, in quanto la funzione, ricevendo solo l'indirizzo iniziale, non avrebbe modo di sapere quando l'array termina (possibile indirizzamento scorretto)

42

ESEMPIO

La funzione:

```
int findMax(int v[], int dim) {
    int i, max;
    for (max=v[0], i=1; i<dim; i++)
        if (v[i]>max) max=v[i];
    return max;
}
```

43

ESEMPIO

La funzione:

Per assicurarsi che la funzione NON modifichi l'array (passato per riferimento), si può imporre la qualifica `const`. Se si tentano modifiche:
cannot modify a const object



```
int findMax(const int v[], int dim) {
    int i, max;
    for (max=v[0], i=1; i<dim; i++)
        if (v[i]>max) max=v[i];
    return max;
}
```

44

ESEMPIO

Problema:

Scrivere *una procedura* che copi una stringa in un'altra

Codifica:

```
void strcpy(char dest[], char source[]) {
    while (*source) {
        *(dest++) = *(source++);
    }
    *dest = '\0';
}
```

45

LIBRERIA SULLE STRINGHE

Il C fornisce una ricca libreria di funzioni per operare sulle stringhe:

```
#include <string.h>
```

Include funzioni per:

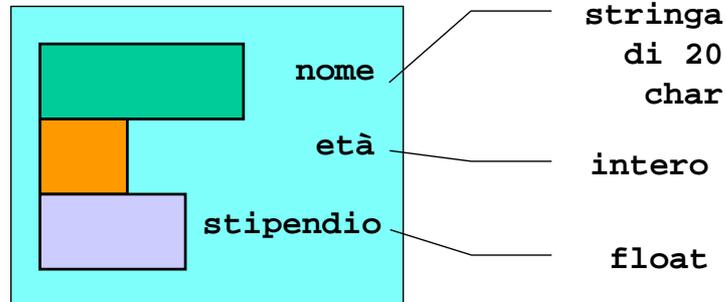
- copiare una stringa in un'altra (**strcpy**)
- concatenare due stringhe (**strcat**)
- confrontare due stringhe (**strcmp**)
- cercare un carattere in una stringa (**strchr**)
- cercare una stringa in un'altra (**strstr**)
- ...

46

STRUTTURE

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

`struct
persona`



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

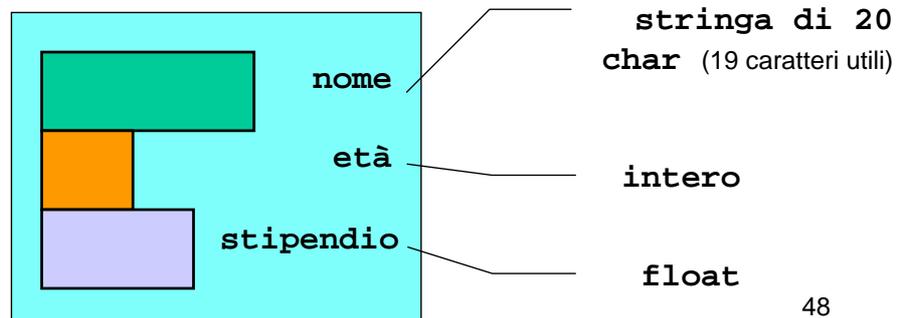
47

ESEMPIO

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

Definisce una variabile
pers

`struct
persona`



48

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**

49

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

50

STRUTTURE

```
main(){  
    struct frutto {  
        char nome[20]; int peso;  
    } f1;  
    struct frutto f2 ;  
    ...  
}
```



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

51

ESEMPIO

```
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 perché ogni variabile di nome *peso* è definita nella propria struct

52

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 (`f2 = f1`)
- che una funzione restituisca una struttura

E soprattutto:

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

53

ASSEGNAZIONE TRA STRUTTURE

```
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

55

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`

56

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

57

ESEMPIO

```
#include <stdio.h>
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);
}
```

58