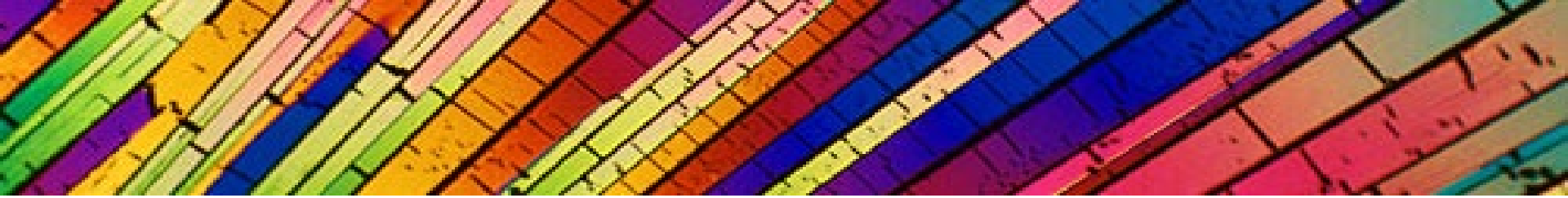


Strutture

Ver. 3

Tipi di dati aggregati

- Lo standard C definisce il termine ***tipo aggregato*** per riferirsi in generale a quei tipi di dati che contengono più elementi: i vettori e le strutture (`struct`)
  - un vettore è un raggruppamento ordinato di variabili *dello stesso tipo*
  - una struttura è un raggruppamento ordinato di variabili *anche di tipo diverso*

Tipo struct

- La `struct` *dichiara un nuovo tipo di dato* formato da uno o più elementi (scalari o aggregati)
- Essendo una dichiarazione *non* riserva memoria
- La `struct` *può* avere un nome detto ***tag*** o essere "anonima"
- Le variabili che compongono una `struct` sono denominate ***membri*** (*fields*)

Tipo struct

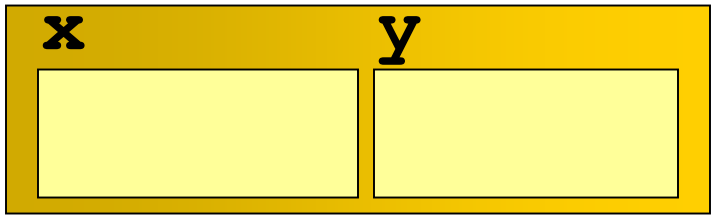
- Una `struct` viene definita come segue:

```
struct nome_tag
{
    tipo1  nome_membro1;
    tipo2  nome_membro2;
    ...
};                    ← notare il ;
```

Dichiarazione di struct

- Esempio

```
struct punto
{
   int x;
   int y;
};
```



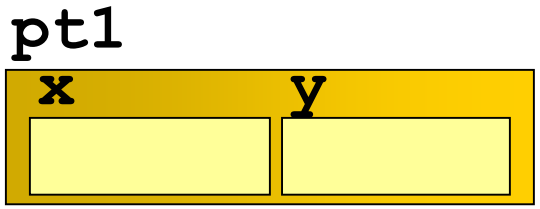
dove `punto` è il *tag* (quindi questa non è una dichiarazione anonima) e `x` e `y` sono i nomi dei due membri (entrambi variabili scalari di tipo `int`)

Dichiarazione di struct

- Il nome dei *membri* definiti in una `struct` è visibile (scope) da tutte le variabili `struct` di quel tipo
- Ogni tipo definito da una `struct` ha un proprio *namespace:* se una `struct` ha un membro `x`, possono esistere variabili (locali o esterne) o nomi di funzioni o membri di *altre* `struct` con lo stesso nome `x`
- Il **tag** di una `struct` fa parte del *namespace dei tag* (include i tag di `union` ed `enum`) e può essere usato solo con la keyword `struct`, il solo nome non ha alcun significato sintattico

Definizione di variabili struct

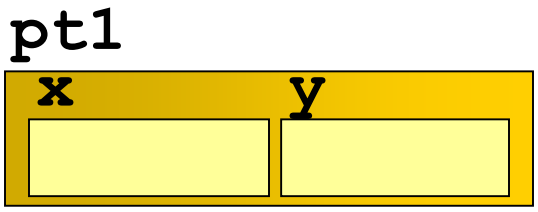
- La definizione riserva memoria per la variabile
- Ha la consueta forma:
  tipo *var1*, *var2*, *var3*, ...
  dove *tipo* è `struct` *tag* e il *tag* deve essere presente (no `struct` anonima)
- Esempio
  `struct` *punto* **`pt1, pt2, pt3;`**



Definizione di variabili struct

- La definizione di variabili può essere contestuale alla dichiarazione del tipo (la `struct` può essere anonima se non serve definire in seguito altre variabili di questo tipo)

```
struct punto {
    int x;
    int y;
} pt1, pt2, pt3;
```



Inizializzazione di struct

- Una variabile `struct` può essere inizializzata solo con espressioni costanti (tra parentesi graffe, se in numero minore dei membri, questi vengono inizializzati a `0`/`NULL`)

```
struct punto pt7 = {12, 14};
```

Operazioni su struct

- Per accedere ai singoli membri di una variabile di tipo `struct` si usa la forma:

  nomeVar . *nomeMembro*

  Si noti che *nomeVar* è il nome della *variabile*, NON quello del *tag*, ad es. `pt1.x`
- I membri di una `struct` sono espressioni *L-value* quindi assegnabili:

```
pt1.x = 24;
```

Operazioni su struct

- Una variabile `struct` può essere assegnata a un'altra (di tipo uguale o *compatibile*) con il simbolo '=', l'assegnazione avviene mediante *copia del contenuto* (non del puntatore)

```
pt1 = pt2;
```

- I cast sono applicabili solo a oggetti scalari:

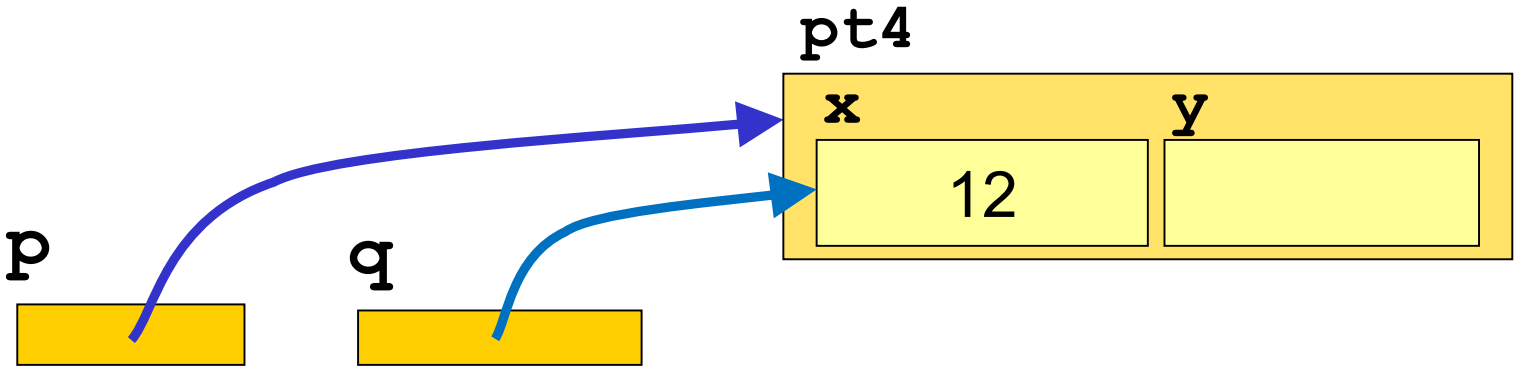
```
struct linea p;
pt1 = (struct punto)p;
```

→ *ERRORE!*

- Può essere utile notare che se una struttura contiene una stringa, la stringa intera viene copiata, non il suo puntatore

Operazioni su struct

- Definizione di un puntatore a `struct`:
  `struct punto *``p;`
- Indirizzo di una variabile di tipo `struct`:
  `p = &pt4;`
- Indirizzo di un membro di una variabile di tipo `struct`:
  `int *q = &pt4.x; /* &(pt4.x) */`
  (l'operatore '`.`' ha priorità maggiore di '`&`')



Operazioni su struct

- L'operatore '`.`' ha priorità maggiore dell'operatore di deriferimento *****, per indicare il membro `x` della variabile di tipo `struct` puntata da `p` si indica:
  `(*p).x = 12;`
  le parentesi sono necessarie in quanto `*p.x` equivale a `*(p.x)` ossia l'oggetto puntato da `x` (che dovrebbe essere un puntatore)
- Al posto di `(*p).x` è preferibile la forma equivalente con l'*operatore freccia* (*right arrow selection*):
  `p->x = 12;`

Operazioni su struct

- La priorità dell'operatore `->` è la più alta in assoluto tra gli operatori del C, la stessa delle parentesi tonde e quadre, quindi `++p->x` equivale a
  `++(p->x)` → *incrementa* `x`*, non* `p`
- L'associatività di `->` è sinistra→destra, quindi `q->r->x` equivale a `(q->r)->x`, ma questo equivale a: `((*q).r)->x` e quindi a `*((*q).r).x` (decisamente meno leggibile di `q->r->x`) dove il membro `r` della struttura puntata da `q` è un puntatore a una struttura con un membro `x`

Tipo dei membri

- Il tipo dei membri può essere qualsiasi:
  - scalare (`int`, `float`, ecc.)
  - aggregato (vettori, stringhe, altre `struct`, `union`)

```
struct persona
{
   char nome[20];
   int eta;
};
struct rettangolo {
   struct punto basso_sinistra;
   struct punto alto_destra;
};
```

Tipo dei membri

- L'inizializzazione di una `struct` contenente un'altra `struct` avviene come nel seguente esempio (le parentesi graffe interne possono essere tralasciate, possibile Warning)

```
struct rettangolo rett={{2,3},{12,9}};
```

- L'accesso ai membri interni si ha tramite l' indicazione del "percorso" da seguire:

```
rett.alto_destra.x = 14;
```

variabile di tipo `struct rettangolo`

membro di `struct punto`

membro di `struct rettangolo`

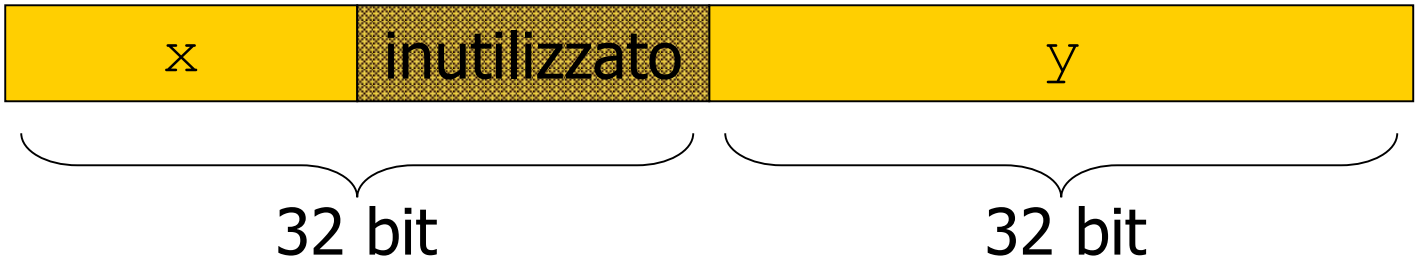
Allocazione dei membri

- In memoria i membri sono allocati contiguamente e nello stesso ordine di dichiarazione (il primo elemento ha indirizzo di memoria più basso)
- Tra un membro e il successivo (non prima del primo) *possono* esserci spazi intermedi di *allineamento della memoria* non indirizzabili (quindi inutilizzabili) e dal contenuto indefinito
- La presenza o no di spazi intermedi dipende dal tipo di microprocessore: molti richiedono che le variabili siano allocate a indirizzi multipli di un certo valore (es. 4 byte, 8 byte, …)

Allocazione dei membri

- Esempio
  (supponendo `short` su 16 bit e `long` su 32 in una macchina con allineamento a 32 bit):

```
struct mix {
   short x;
   long y;
};
```



- Il compilatore può avere modo (es. `#pragma`) di definire `struct` senza spazi di allineamento

Tipi definiti dalle struct

- ***Ogni*** dichiarazione `struct` crea un *tipo* diverso (due dichiarazioni `struct` sono di tipo diverso anche se hanno stesso tag e struttura)
- Quindi non si può scrivere:

```
struct a {
   int x;
} var1 = {10};
struct a {
   int x;
} var2;
var2=var1;
```

perché duplica la dichiarazione di `struct a` e `var2` è considerata di tipo diverso da `var1`

Tipi definiti dalle struct

- Per dichiarare variabili dello stesso tipo `struct` in funzioni diverse la dichiarazione deve essere esterna (in realtà sono considerati tipi diversi, ma compatibili)

Funzioni e struct

- Nelle funzioni, una variabile di tipo `struct` viene passata *per valore:*
  - chiamata:
    ```
    funz(pt1);
    ```
  - definizione del parametro formale:
    ```
    int funz(struct punto pt) {...}
    ```
- Una funzione può restituire una `struct`
  - chiamata:
    ```
    pt1 = funz(...);
    ```
  - dichiarazione del tipo della funzione:
    ```
    struct punto funz(...)
    { ...
      return ptx;}
    ```

Funzioni e struct

- Se i parametri e/o il valore restituito sono variabili `struct`, ci può essere un overhead significativo al run-time dovuto alla copia dei byte, in caso di necessità si passino i puntatori

Funzioni e struct

- Perché chiamante e chiamato riconoscano la stessa `struct`, questa deve avere un'*unica dichiarazione esterna* alle funzioni
- *Nel caso di progetti multi-file*, la definizione della struct deve essere ripetuta in tutti i file, tipicamente questa viene messa in un file di header (con protezione dalla ri-inclusione)
- Più correttamente le dichiarazioni delle `struct` nei due file devono essere *compatibili*, il che significa: avere lo stesso numero di membri con lo stesso nome, nello stesso ordine e con tipi tra loro compatibili

Campi di bit

- Sono insiemi di bit che costituiscono un valore di tipo intero (`signed` o `unsigned`)
-
```
struct cartaDaGioco {
   unsigned valore : 4;
   unsigned seme   : 2;
   unsigned colore : 1;
};
```
- Il numero a destra di ciascun membro indica da quanti bit è costituito il campo
- I singoli campi si comportano come valori interi e quindi possono comparire in espressioni, essere assegnati, confrontati, etc. ma prima dell'uso sono soggetti alle promozioni integrali

Campi di bit

- I campi vengono accorpati a costituire gruppi di byte delle dimensioni di una *Storage Unit* senza spazi vuoti
- La *Storage Unit* ha dimensioni dipendenti dall'implementazione (8, 16, 32 bit) ma in genere è pari alla dimensione di un `int`
- Anche l'ordine di memorizzazione dei campi nelle Storage Units (se da sinistra a destra o viceversa) è implementation dependent
- Non si può determinare la posizione di un campo di bit (può essere a metà a un byte) e non è possibile definire un puntatore ad esso

Campi di bit

- Un campo *anonimo* (senza nome della variabile) può servire come riempitivo (padding) con quel numero di bit, ma non può essere utilizzato per contenere valori
- Un campo anonimo con dimensione 0 forza l'*allineamento* di memoria del membro seguente alla successiva Storage Unit (`int`)
-
```
struct cartaDaGioco {
    unsigned valore : 4;
    unsigned        : 5;
    unsigned seme   : 2;
    unsigned        : 0;
    unsigned colore : 1;
};
```

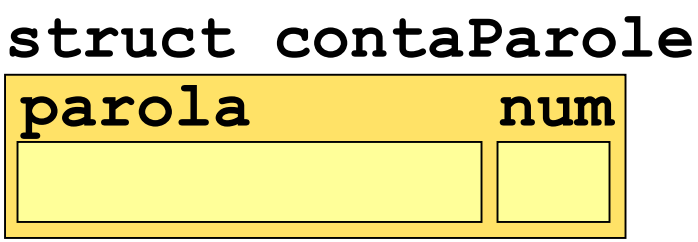
allocati nel primo `int`

allocato nel secondo `int`

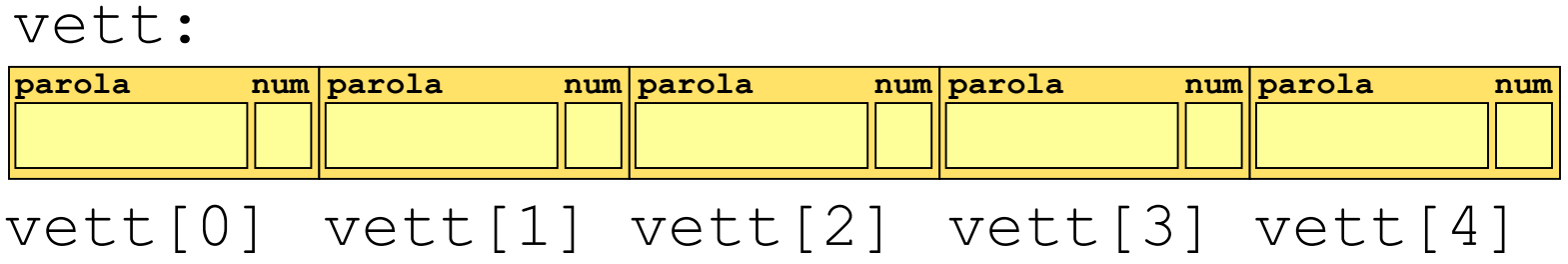
Vettori di struct

- È possibile definire un vettore i cui elementi siano oggetti di tipo `struct`:

```
struct contaParole {
    char parola[20];
    int num;
} vett[5];
```



- Il vettore `vett` ha 5 elementi, ciascuno è una `struct contaParole`:



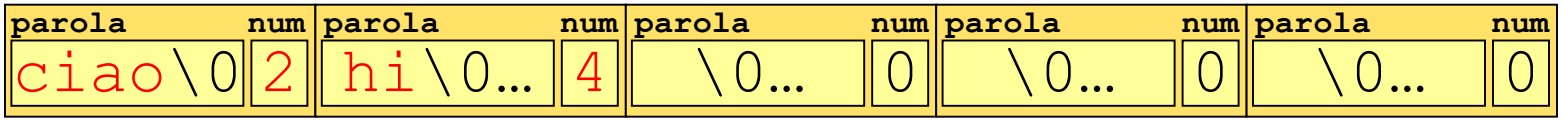
Vettori di struct

- Per inizializzare un vettore di `struct` si elencano i valori tra graffe:

```
struct contaParole {
    char parola[20];
    int num;
} vett[5] = {{"ciao",2},{"hi",4}};
```

- I primi 2 elementi sono inizializzati, i successivi sono "" e 0 (le graffe interne possono essere omesse)

vett:



Confronto di variabili struct

- Per verificare se due variabili dello stesso tipo `struct` sono uguali (stesso contenuto), si deve confrontare ciascun membro
  `if (pt1.x==pt2.x && pt1.y==pt2.y)`
- NON si possono confrontare direttamente:
  `if (pt1 == pt2)` ← *ERRORE*
- NON si possono confrontare con `memcmp` perché gli spazi di allineamento hanno contenuto indefinito e quindi eventualm diverso (ma se le variabili sono stati precedentemente azzerate, ad esempio con `memset` o `calloc`, allora è possibile usare `memcmp`)

Tipo union

- Definisce un tipo che è adatto a contenere un solo elemento di uno dei tipi indicati
- La dichiarazione è simile alle strutture, ma *un solo membro* per volta è valido
- ```
  union tris {
      int x;
      double y;
      char nome[10];
  } var;
  ```
- In questo es. la variabile `var` è considerata di tipo `int` se viene usata come `var.x`, `double` se usata come `var.y`, stringa di 10 caratteri se usata come `var.nome`
  ```

Tipo union

- Sta al programmatore mantenere memoria del tipo attuale di `var` e usarla coerentemente, l'uso incoerente non viene rilevato come errore
- L'assegnazione del valore cambia il tipo contenuto, rispetto all'esempio precedente:
  - `alfa.x = 12;`
    da questo punto `alfa` è di tipo `int`
  - `alfa.y = 23.23;`
    da questo punto `alfa` è di tipo `double`
  - `strcpy(alfa.nome, "ciao");`
    da questo punto `alfa` è di tipo *vettore-di-*`char`

Tipo union

- Lo spazio dei vari membri è sovrapposto, quindi quando si assegna il valore a un membro, gli altri sono indefiniti

Tipo union

- Per tener traccia del tipo attuale di una variabile `union`, si può includere questa in una `struct` con un campo etichetta (di solito una `enum`) che ne indica il tipo:

```
struct {
   enum {INT, DOUBLE, STR} tipo;
   union {
      int x;
      double y;
      char nome[10];
   }
} var;
```

Operatore typedef

- Dichiara il nome di un *nuovo tipo di dato* a partire da altri tipi di dati già definiti (scalari, aggregati, ecc.)

  `typedef` *tipoEsistente nuovoTipo*`;`
- Viene spesso collocata dopo le `#define`
- La dichiarazione di tipo appare identica alla definizione di una *variabile*, ma è preceduta dalla keyword `typedef`
- Per comprendere una dichiarazione `typedef`, si supponga che non ci sia la keyword `typedef` ottenendo quindi la definizione di una *variabile*, il tipo che avrebbe quella variabile è quello definito dalla `typedef`

Operatore typedef

- Esempi
  - `typedef char string[80];`
    Se non ci fosse `typedef`, `string` sarebbe una *variabile* di tipo *vettore-di-80-*`char`;
    con `typedef`, `string` è il tipo *vettore-di-80-*`char`
    quindi:
    `string parola;`
    definisce la *variabile* `parola` di tipo `string`, cioè di tipo *vettore-di-80-*`char`
  - `typedef char *strp;`
    dichiara *il tipo* `strp` come puntatore a `char`
    quindi:
    `strp par;`
    definisce *la variabile* `par` di tipo `strp`, cioè `char*`

Operatore typedef

- Esempio
  - ```
    typedef struct rett
    {
      struct punto basso_sinistra;
      struct punto alto_destra;
    } rettangolo;
    ```
    dichiara il *tipo* `rettangolo` come `struct rett`
    quindi:
    ```
    rettangolo r = {{2,3},{12,9}};
    ```
    definisce la *variabile* `r` di tipo `struct rett`
- Quando si usa una `typedef` su una `struct` o una `union`, il tag può essere omesso, tranne quando un membro della `struct` è un puntat. a una `struct` dello stesso tipo (vedi liste)
    ```

Operatore typedef

- Esempio
  - `typedef struct`
    `{`
    `  int x;`
    `  int y;`
    `} vpunti[10];`

    dichiara *il tipo* `vpunti` come vettore di 10 elementi di quel tipo `struct` *anonimo* lì dichiarato

    `vpunti vett;`

    definisce *la variabile* `vett` di tipo `vpunti`, ossia vettore-di-10-`struct` (la `struct` anonima dichiarata sopra); utilizzabile ad esempio così:

    `vett[0].x = 12;`

    `mettere esempio alla 18`

Operatore typedef

- I nomi dei nuovi tipi non devono essere nomi utilizzati da altri *identificatori ordinari*
- Poiché i nomi dei tag appartengono a *namespace* indipendenti, è possibile dichiarare un nome di tipo con lo stesso nome di un tag

```
typedef struct rett {
   int x;
   int y;
} rett;
```

Questa pratica è frequente, altre volte si utilizza un nome di tipo con iniziale maiuscola (`Rett`) o che termini con `_t` come fa la libreria standard (`rett_t`)

Typedef e portabilità

- L'operatore `typedef` viene spesso utilizzato per "nascondere" come il compilatore realizza internamente una certa funzionalità, fornendo al programmatore un comportamento standard
- Questo si traduce in una migliore *portabilità* del codice (piattaforma hardware, sistema operativo, compilatore, etc.)
- Ad esempio, il tipo `size_t` è definito con `typedef` in modo appropriato da ciascun compilatore: non importa come internamente sia stato realizzato, usando `size_t` non serve preoccuparsi di questi dettagli

Typedef e funzioni

- Nel caso di progetto multi-file è utile riportare le `typedef` in un file di header e includerlo
- La compatibilità di tipo di due variabili viene determinata "smontando" la dichiarazione `typedef` nella sua struttura basata sui tipi primitivi, ad esempio:

```
typedef int * intp;
intp p;
int *q;
q=p;
```

← *lecito perché sono dello stesso tipo*

Typedef e funzioni

- Quando un tipo è dichiarato con `typedef` su strutture aggregate anonime, nell'operazione di "smontaggio" idealmente le strutture aggregate anonime ricevono lo stesso tag fittizio, diverso per ogni `typedef`, ma compatibile e quindi le var di quel tipo sono tra loro compatibili e assegnabili

```
typedef struct {
  int a;
  int b;} Miastruct;
Miastruct a={0,0};
Miastruct b;
b=a;
```
← *lecito perché sono dello stesso tipo*

Typedef e const

- Nella definizione di una variabile di tipo `T` dichiarato con una `typedef`, la posizione della `const` non è significativa e si applica alla variabile
- Quindi le due definizioni seguenti sono equivalenti

```
const T var;
T const var;
```

Se ad esempio `T` è dichiarato come:

```
typedef int* T;
```

entrambe le definizioni sono equivalenti a:

```
int * const var;
```

Typedef e const

- Attenzione: se invece `T` viene definito con una `#define`, `T` non è un vero nuovo tipo e la posizione della `const` è significativa
- Se ad esempio `T` è definito come:
  `#define int* T`
  la definizione `const T var` equivale a
  `const int* var;`
  e la definizione `T const var` equivale a
  `int* const var;`
  (non è equivalente alla precedente)

Sizeof su struct

- Definendo la variabile `vett` come:

```
struct x {
   int b;
} vett[10], s;
```

`sizeof` produce i seguenti valori:

- `n = sizeof vett;`
  dà il numero di byte richiesti da un vettore di 10 elemento di tipo `struct x`
- `n = sizeof vett / sizeof(struct x);`
  dà il numero di elementi del vettore `vett`
- `n = sizeof vett / sizeof s;`
  dà il numero di elementi del vettore `vett`

Esercizi

1. Si scriva un programma che chieda all'utente le coordinate x e y di 4 punti nel piano, li memorizzi in un vettore di `struct`, quindi calcoli la lunghezza del perimetro del quadrilatero e la distanza minima tra i punti

Esercizi

2. Si scriva un programma che dichiari il seguente tipo di dati che descrive un generico poligono regolare

```
struct poligono
{
    int nlati;
    double lato;
};
```

Si scrivano le seguenti funzioni e le si chiamino da un `main` di prova:

1. `struct poligono creapoli(void);`
   chiede all'utente il numero lati e la lunghezza lato, quindi restituisce una `struct poligono`

Esercizi

Seguito

- `double areapoli(struct poligono p);`
  calcola l'area del poligono passato

$$A = \frac{n \cdot l^2}{4 \cdot \tan \frac{\pi}{n}}$$

- `double perimpoli(struct poligono p);`
  calcola il perimetro del poligono passato
- `void doppiopoli(struct poligono *pp);`
  raddoppia il lato *dello stesso poligono passato*

3. Come il precedente, ma si usi la `typedef`

Esercizi

4. Un file contiene, su ciascuna riga, separati da spazi, 4 campi relativi a: nome, cognome, età, e salario (in Euro) di un certo numero di persone (max 100). Scrivere un programma che crei un vettore di 100 `struct`, lo riempia con i dati letti dal file, lo passi ad una funzione che lo riordini in base al cognome e quindi salvi il risultato ordinato in un secondo file mantenendo lo stesso formato del file originale (campi separati da uno spazio).
   N.B. Il campo in base al quale viene riordinato un vettore viene detto *chiave*.