LEZIONE 20-10

CONTINUAZIONE CONVERSIONE IMPLICITA:

il problema è sempre la codifica di oggetti, come ad esempio int o float, che sono anche piuttosto simili, ma quando si inizia a dover comparare un int con un char inizia a essere più complicato.

Per fare questo si seguono delle regole che il C segue e a cui noi quindi ci atteniamo.

```
int i=10,j=3; float f=3.0, f1;
f1 = i + f;//conversione implicita
f = j/i;   //conversione implicita
```

Facendo una conversione implicita, il compilatore si occupa di convertire le variabili in modo che siano dello stesso tipo.

La conversione implicita ha come lato negativo che ogni tanto si possono perdere risorse, ad es. float => int, presuppone una perdita di numeri dopo la virgola.

formalizziamo

- Regole di **conversione implicita**:
  - Si converte temporaneamente l'operando di tipo inferiore facendolo divenire di tipo superiore;
  - ogni variabile di tipo **char** o **short** (incluse le rispettive versioni **signed** o **unsigned**) viene convertita in variabile di tipo **int**;
  - se dopo l'esecuzione del passo 1 l'espressione risulta ancora eterogenea rispetto al tipo degli operandi coinvolti, rispetto alla gerarchia:

int < long < unsigned < unsigned long <
float < double < long double

Il risultato dell'espressione avrà tipo uguale a quello di più alto livello gerarchico.

si eseguono tutte le azioni in ordine in modo che le variabili di tipo inferiori diventino superiori, di conseguenza se devo fare:

```
int a == float b;
    int a => float a;
    float a == float b;
```

Inoltre il fatto che sia implicita dichiara che non si vede, ma comunque bisogna saperla leggere.

Per rendere meglio comprensibile/visibile o per rendere il codice più ordinato/leggibile, si usa la keyword **(cast)**

```
float media;
int num1=5, num2=3;
everage = (num1+num2)/2; // in questo caso ci affidiamo alla converione
implicita, il valore mancherà di cifre decimali

// se invece
everage = (float) (num1+num2)/2; // risultato corretto, ma anche più leggibile.
```

Questo operatore è uno di quelli che viene tradotto quasi per primo, nell'ordine in cui si eseguono le operazioni. (per ordine di operazioni intendo quello che definiva che:

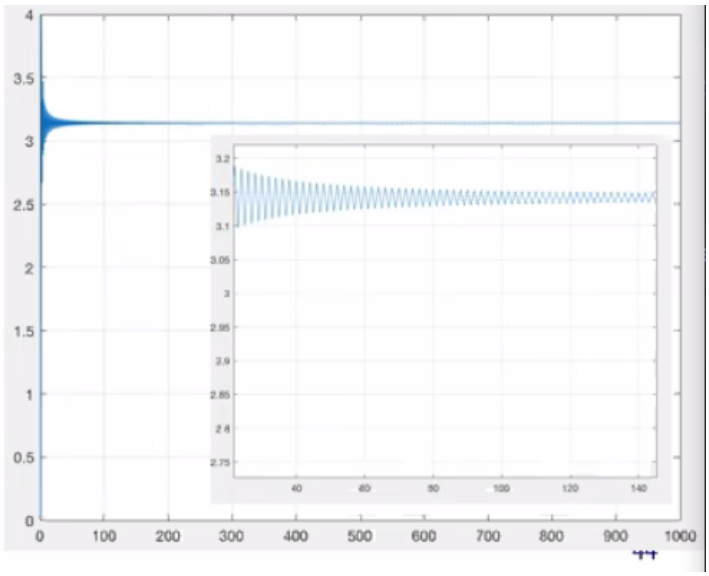
```
(a + c) / b
è diverso da
a + c/b
```

).

esempio:

calcolare pigreco, con precisione a piacere usando l'approssimazione di Leibniz:

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$



Questo metodo (soprattutto si deduce dal grafico di questo metodo), permette di calcolare il pigreco con un certo grado di precisione. Per avere 4 cifre piuttosto convergenti bisogna, come si vede appunto dal grafico, usare almeno un centino di cifre.

nel main

```c
int i;
int precisione;
int segno = -1;
float pigreco = 1;

scanf("%d", &precisione);

for (i=1; i<=precisione; i++){
    pigreco += segno * 1/(2*i+1);

    printf("%f\n", pigreco*4);
    segno = (-1) * segno
}
```

In questo programma, in cui si usa la conversione implicita, si porta a un errore di approssimazione, perché la conversione implicita nel caso di divisone tra due variabili **int** causa un numero intero, ma che logicamente manca del resto dalla divisione stessa.

Con il casting in modo esplicito, il programma funziona meglio, e soprattutto non lascia posto al caso che il compilatore possa ogni tanto fare correttamente e in altri casi sbagliando nella conversione implicita.

TIPI USER-DEFINED IN C:

Utilizzo di **struct** cioè delle variabili strutturate.

Questo si usa per descrivere un oggetto, che ha molte variabili al suo interno, che possono prendere dei tipi di valori diversi. Al loro interno si possono anche avere array.

Tutto ciò permette di caratterizzare variabili molto complesse, con una serie di caratteristiche diverse fra di loro.

```c
struct Impiegato{
    char Nome[20];
    char Cognome[20];
    float Stipendio;
    char CodiceFiscale[16];
}
```

Alla fine di questa definizione di struttura, che descrive come sarà questa variabile, non viene alloccata nessuna memoria, ma è solo un passo formale del C.

Però com'è che si fa ad accedere a queste variabili? Perchè negli array sapevo come erano i campi e l'unica cosa a cui dovevo stare attento era di non sorpassare il numero di spazi assegnati nel richiamare gli oggetti. Se non più seguire una dichiarazione questa funzione struct non ha valore (vuol dire che la dichiaro e poi posso iniziare a usarla).

```
struct Impiegato{
    char Nome[20];              // DEFINIZIONE
    char Cognome[20];           // DEFINIZIONE
    float Stipendio;            // DEFINIZIONE
    char CodiceFiscale[16];     // DEFINIZIONE
} amministratore, segretario;   //DICHIARAZIONE

struct Impiegato staff1,staff2; //DICHIARAZIONE
//Da qui in poi posso iniziare a usarla, siccome la ho dichiarata
```

All'interno dello struct non importa l'ordine (a meno che una variabile dipenda da un altra, in questo caso la variabile indipendente va dichiarata prima). Questo avviene perché al compilatore non cambia se metti prima il nome o il cognome, perché poi la variabile viene richiamata in ugual modo. (Nessun cambio di ordine della definizione porta a miglioramento delle prestazioni, il programma ci mette uguale a definire prima il nome o poi il cognome).

Per indicizzare i campi all'interno dello **struct** viene usata la dot notation:

```
impiegato.Stipendio = 0.0;      // assegno al campo stipendio dello struct impiegato uguale a zero
// dopo aver inizializzato queste cose, poi posso stilare l'albero sintattico per usare lo struct come variabile normale:

if (impiegato.Stipendio < 2000){
    impiegato.Stipendio = 1.5 * impiegato.Stipendio;
}
```

Il **dot operator** è in cima alla tabella delle precedenze di traduzione, in linea con l'ordine di precedenza delle parentesi.

Se all'interno dello struct è dichiarato un campo che a sua volta è uno struct, allora il dot operator sarà riutilizzato fino ad ottenere il campo ricercato.

```
struct DAta{                // definizione
    int giorno;
    int mese;
    int anno;
};

struct Data mia-data;       // dichiarazione

mia-data.mese=12;  // e uguale agli altri campi
```

Il construtto **struct** occupa, come memoria, esattamente come la somma della memoria occupata dagli elementi che lo compongono.

problema della retta

- voglio definire una retta come y=mx+q

- meglio usare un array o uno struct?
    - possibile usare un array
    - lo struct permette un risultato molto più leggibile e comprensibile dagli altri.
    - esempio sul ppt

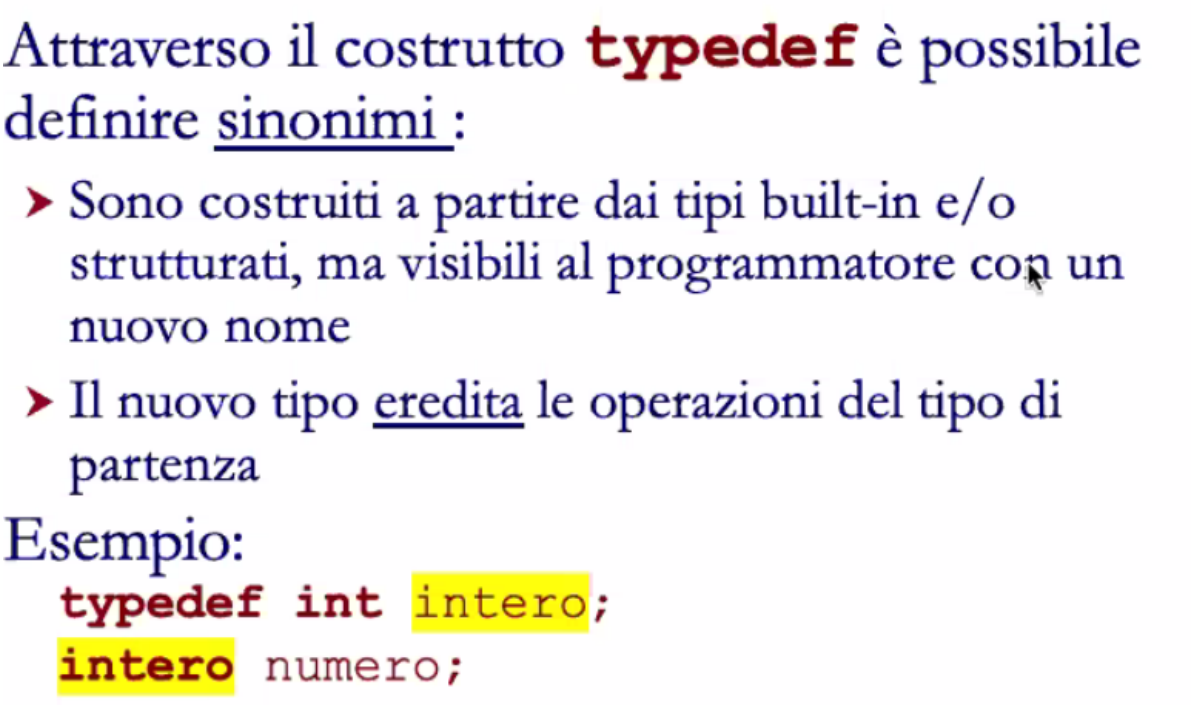
STRUTTURE COMPLESSE:

- uso delle strutture per definirne altre
- ```
  struct lato{
      int lunghezza;
      int x;
      int y;
  }
  struct rettangolo{
      lato lato-lungo;
      lato lato-corto;
  }
  ```

utilizzo del typedef:

L'idea è quello di creare un meccanismo sintattico e formale in modo che io possa usare degli altri nomi per definire e chiamare le variabili, che possono essere variabili semplici o strutturate a loro volta. Di solito il typedef è posto tra la definizione di costanti e il main().

Il typedef crea una tabella di sinonimi:

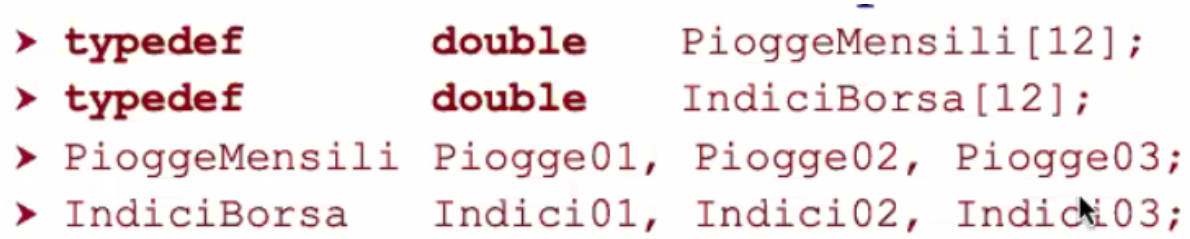

  ```

la variabile numero adesso è di tipo int.

Il Typedef non si basa sulla ereditarietà, cioè non funziona che gli elementi prendono il valore di quelli di partenza perchè lo "ereditano", ma proprio perchè il compilatore crea una tabella di sinonimi e poi li sostituisce direttamente al posto del codice.

Dove si usa

- variabile che ricorre spesso e che necessita di avere una serie di variabili che necessitano solo di sinonimi appunto
- evitare di definire più volte uno stesso tipo, e semplicemente usare il typedef per entrambi.



A questo punto il programmare diventa sempre più a pensare a che cosa scegliere, in modo che il programma sia il più comprensibile e utilizzabile possibile.

```
typedef int Vettore[20];
typedef Vettore Vettore20per20;
typedef vettore Nome;
// osso dichiarare una serie di cose usando sempre il tipo vettore

// a questo punto posso io a priori dichiare delle costanti a un tipo di dato in
modo che mi venga più comodo il suo uso.
```

Viene usata la keyword **enum** per indicare una variabile strutturata che può prendere solo un valore finito e predeterminato di valori. Questo molto utile ad esempio se si vuole usare una lista di cose di cui si conosce tutto, e che deve essere presa come variabile comprensibile al compilatore all'interno del programma. Ad esempio un enum potrebbero essere i giorni della settimana a cui ad ogni giorni si associa un valore int, in modo da poter fare cicli for e if facilmente con i numeri.

```
enum Settimana {
    lun, mar, merc, giove, sab, dom
}
```

INTRODUZIONE AI PUNTATORI:

- sono un elemento necessario per la programmazione in C
- necessario per:
    - costruzione di funzioni
    - allocazione dinamica
    - efficenza dei programmi in memoria

- i puntatori sono positivi nel senso che permettono un grandissimo uso della memoria e quindi a rendere il programma più efficente, ma al contrario potrebbe causare errori di alloccamento di memoria o sovrascrivere altra memoria. Di conseguenza punto di forza del C, ma anche punto di debolezza
- il puntatore è una variabile che memorizza un indirizzo di memoria
    - prima la variabile stessa conteneva al suo interno il suo indirizzo e gestiva tutto il compilatore.
    - se adesso la memoria ha un indirizzo, adesso posso usare il right-value del puntatore per accedere ad altri luoghi di memoria.
- il puntatore è una variabile che punta a una località dell'indirizzo di un altra variabile.
- Il puntatore non da garanzie che all'interno di quello spot di memoria ci sia qualcosa o se ci sia, sia comprensibile.
- ```
  // sintassi:
  TipoDato *Puntatore;
  ```
    - la sintassi del C fa si che non ci sia un tipo del puntatore, anche se, essendo una variabile dovrebbe possedere un tipo.
    - uso un * simbolo che indica il fatto che è un puntatore, però quindi il tipo di dato deve essere quella a cui l'indirizzo referenzia
    - il tipo di variabile diventa quindi **puntatore**
    - l'operatore * indica la deferenziazione.
  ```