# LABORATORIO DI PROGRAMMAZIONE 1
## CORSO DI LAUREA IN MATEMATICA
## UNIVERSITÀ DEGLI STUDI DI MILANO
### 2019–2020

Indice



Ultima revisione: 15 maggio 2020.

### La parola chiave `typedef`

Abbiamo visto come definire nuovi tipi in C tramite l'uso delle strutture. È possibile abbreviare il nome dei nuovi tipi definiti in questo modo usando la parola chiave `typedef`. La possibilità di rinominare tipi esistenti con `typedef` non è limitata alle sole strutture, anche se è in questo contesto che il suo uso è più frequente. In generale, assumiamo che il tipo `T` sia già stato definito (o sia primitivo). Allora l'istruzione:

```
typedef T mio-nome;
```
$(*)$

stabilisce che `mio-nome` è sinomimo di `T`.[a] Il nome di tipo `T` continua a essere disponibile anche dopo l'esecuzione di $(*)$. Applicando questi fatti generali all'Esercizio 1, le righe di codice:

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

definiscono un tipo strutturato (con etichetta opzionale `punto`) atto a rappresentare un punto nel piano. Questo nuovo tipo, ossia:

```
struct punto
```

può ora essere rinominato `Vett`, come chiede l'esercizio, in questo modo, cfr. $(*)$:

```
typedef struct punto Vett;
```

Se però volessimo usare nel resto del programma solo il sinonimo `Vett`, e mai il nome originario `struct punto`, allora potremmo più concisamente definire `Vett` in questo modo, senza bisogno dell'etichetta `punto`, e sempre in accordo con $(*)$:

```
typedef struct
        {
            double x;
            double y;
        } Vett;
```

In ogni caso, dopo il `typedef` è possibile dichiarare una variabile di tipo `Vett` al solito modo:

```
Vett v;
```

[a] Si noti quindi bene che, a rigore, `typedef` *non* definisce un nuovo tipo, ma più semplicemente introduce un sinonimo per un tipo già esistente.

Negli Esercizi 1 e 5 definirete i tipi `Vett` e `Tri` i cui valori rappresentano punti e triangoli nel piano euclideo $\mathbb{R}^2$, rispettivamente. Nei rimanenti esercizi i due tipi sono usati senza altre spiegazioni.

## Esercizio 1

*Distanza euclidea nel piano.*

Tempo: 15 min.

Si scriva un programma che definisca il tipo `Vett` come una struttura di due campi, ciascuno di tipo `double`, che rappresentano le coordinate di un punto nel piano. Chiamate i due campi `x` e `y`, rispettivamente.

Il programma chiede all'utente di inserire le coordinate di due punti del piano, ne calcola la distanza invocando una funzione di prototipo `double dist(Vett, Vett)`, visualizza il risultato e termina.

## Esercizio 2

*Operazioni sui vettori.*

Tempo: 25 min.

Si scrivano funzioni di prototipo:

```
Vett somma(Vett,Vett)
Vett invadd(Vett)
Vett sott(Vett,Vett)
Vett pscal(Vett, double)
double pint(Vett,Vett)
```

che implementino, rispettivamente, le operazioni di somma, inverso additivo ($\vec{v} \mapsto -\vec{v}$), sottrazione, prodotto per uno scalare e prodotto interno standard. Scrivete una funzione `main` che permetta di testare le vostre implementazioni.

## Esercizio 3

*Indipendenza lineare.*

Tempo: 15 min.

Si scriva una funzione di nome `linind` che accetti in ingresso due puntatori a `Vett` e restituisca in uscita l'intero 1 se i due vettori puntati dai parametri in ingresso sono linearmente indipendenti, e 0 altrimenti. Si scriva poi una funzione `main` che accetti dalla riga di comando le coordinate di due vettori nel piano, e produca in uscita sulla console uno dei due messaggi `Linearmente indipendenti` o `Linearmente dipendenti`, a seconda di quale caso si verifichi. Si gestiscano gli errori legati alla lettura degli argomenti dalla riga di comando. Per convertire una stringa sulla riga di comando in `double` si usi la funzione `double atof(char *s)` della libreria standard dichiarata nel file di intestazione `stdlib.h`.

## Esercizio 4

*Indipendenza affine.*

Tempo: 15 min.

Si scriva una funzione di nome `affind` che accetti in ingresso tre puntatori a `Vett` e restituisca in uscita l'intero 1 se i tre vettori puntati dai parametri in ingresso sono affinemente indipendenti, e 0 altrimenti. Si scriva poi una funzione `main` che accetti dalla riga di comando le coordinate di tre vettori nel piano, e produca in uscita sulla console uno dei due messaggi `Affinemente indipendenti` o `Affinemente dipendenti`, a seconda di quale caso si verifichi. Si gestiscano gli errori legati alla lettura degli argomenti dalla riga di comando.

Si ricordi che, per definizione, i tre vettori $v_0, v_1, v_2 \in \mathbb{R}^2$ sono affinemente indipendenti se i due vettori $v_1 - v_0, v_2 - v_0 \in \mathbb{R}^2$ sono linearmente indipendenti.

## Esercizio 5

*Triangoli.*

Tempo: 10 min.

Definite il tipo `Tri`, i cui valori rappresentano triangoli nel piano euclideo. Il tipo è definito come una struttura di tre campi, ciascuno di tipo `Vett`, che rappresentano le coordinate dei tre vertici del triangolo. Chiamate i tre campi `v1`, `v2` e `v3`, rispettivamente.

Si scriva una funzione di prototipo:

```
int degenere(Tri *)
```

che restituisca 1 se il triangolo puntato dall'argomento è degenere — ossia, se i suoi vertici sono affinemente dipendenti — e 0 altrimenti. Scrivete una funzione `main` che permetta di testare la vostra implementazione.

## Esercizio 6

*Area di un triangolo.*

Tempo: 25 min.

Si scriva una funzione di prototipo:

```
double area(Tri *)
```

che restituisca l'area del triangolo puntato dall'argomento. Per calcolare l'area del triangolo potete usare la *formula di Erone*: detta $\mu(T)$ l'area del triangolo $T$, si ha

$$\mu(T) = \sqrt{s(s-a)(s-b)(s-c)},$$

dove $a$, $b$ e $c$ sono le lunghezze dei tre lati di $T$, ed $s := \frac{1}{2}(a+b+c)$ ne è il semiperimetro. Scrivete una funzione `main` che permetta di testare la vostra implementazione.

## Esercizio 7

*L'area è invariante per rotazioni.*

Tempo: 30 min.

Si ricorda che, dato un angolo $\theta$, la matrice

$$R(\theta) := \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

è detta *matrice di rotazione* (di angolo $\theta$). Dato un vettore $\vec{v} \in \mathbb{R}^2$, il prodotto $R(\theta)\vec{v}$ è l'immagine del vettore $\vec{v}$ dopo la rotazione in senso antiorario del piano di un angolo $\theta$ attorno all'origine.

Scrivete una funzione di prototipo:

```
void rot(Vett *, double a)
```

che applichi al vettore passato come primo parametro la rotazione attorno all'origine di un angolo `a`. (*Suggerimento.* Per calcolare seni e coseni, usate le funzioni `double sin(double)` e `double cos(double)` definite nel file di intestazione `math.h` della libreria standard.) Scrivete poi una funzione di prototipo:

```
void rot(Tri *, double a)
```

cha applichi la rotazione al triangolo passato come argomento.

Scrivete un programma che permetta all'utente di specificare un triangolo nel piano tramite le coordinate dei suoi vertici, assieme a un angolo $\theta$ (espresso in radianti e codificato come tipo `double`). Il programma calcola e visualizza l'area del triangolo — si usi la funzione scritta per risolvere l'Esercizio 6 — applica al triangolo la rotazione antioraria di angolo $\theta$, calcola e visualizza nuovamente l'area del triangolo dopo la rotazione, e termina.

## Esercizio 8

*Lettura di triangoli da file.*

Tempo: 40 min.

Scrivete un programma che accetti dalla riga di comando il nome di un file. Se non vi sono argomenti sulla riga di comando il programma termina con un messaggio d'errore appropriato. Se vi è più di un argomento sulla riga di comando, il primo argomento è considerato il nome del file e i successivi argomenti sono ignorati. Il programma tenta di aprire il file specificato sulla riga di comando in lettura. Se l'apertura non va a buon fine il programma termina con un messaggio d'errore. Altrimenti, una volta aperto il file, il programma assume che il file sia strutturato nel modo seguente. La prima riga del file contiene un intero $n$ positivo. Il file contiene poi altre $n$ righe, ciascuna delle quali è nel formato

```
x1 y1 x2 y2 x3 y3
```

dove $x_i$ ed $y_i$ sono valori di tipo `double` che rappresentano le ascisse e le ordinate di tre punti nel piano. Il programma interpreta i valori in ciascuna riga come le coordinate dei vertici di un triangolo nel piano. Di conseguenza, dopo aver aperto il file ed aver letto il valore $n$, il programma dichiara un array di triangoli (cioè, un array di `Tri`) di dimensione $n$ e vi memorizza $n$ triangoli in accordo con le coordinate lette dal file. Completata la lettura, il programma chiude il file.

Il programma visualizza infine le informazioni seguenti sul terminale, e termina.

(1) Il numero totale di triangoli presenti sul file.
(2) Il numero totale di triangoli degeneri.
(3) La media aritmetica delle aree dei triangoli.
(4) L'indice del triangolo di area minima, **esclusi i triangoli degeneri**. (Nel caso vi siano più triangoli non degeneri di area minima, è sufficiente che il programma visualizzi l'indice di uno di essi.)

Per testare il vostro programma, preparate con un editor di testi un file da usare per le prove.

Osservate che per calcolare i valori (1–4) richiesti dall'Esercizio 8 non sarebbe necessario memorizzare i triangoli letti da file in un array, come richiesto invece dalla traccia: per esempio, per calcolare la media delle aree basterebbe sommare l'area del triangolo letto alla riga $i$ alle aree precedentemente calcolate, e alla fine della lettura dividere per $n$. La soluzione dell'Esercizio 9 seguente, invece, non può prescindere dalla memorizzazione dei triangoli.

## Esercizio 9

*Ordinamento dei triangoli letti da file.*

Tempo: 30 min.

Ampliate il programma scritto per risolvere l'Esercizio 8 di modo che, prima di terminare, visualizzi l'elenco dei triangoli letti da file in ordine crescente di superficie. Come algoritmo di ordinamento usate Bubble Sort, che avete implementato nell'Esercizio 11 della Lezione 4.

Osservate che, nell'Esercizio 9, invece di ordinare effettivamente il vettore di triangoli, il programma potrebbe ordinare un vettore ausiliario di *puntatori* ai triangoli. Questa seconda soluzione è in realtà l'unica disponibile nel caso in cui ciascuna unità dei dati da ordinare abbia dimensioni tali da rendere impraticabile l'esecuzione degli scambi richiesti dall'algoritmo di ordinamento.

Dipartimento di Matematica *Federigo Enriques*, Università degli Studi di Milano, via Cesare Saldini, 50, I-20133 Milano