Funzioni:

```
tipo nome(arg1, arg2, ..., argN) {
    ≡
    return valore;
}
```

Esempio: prova.c

```
#include <stdio.h>          } printf(...)
#include <math.h>             scanf(...)
                              fprintf(...)
int dado(int);          // dichiarazione funzione
double uniforme(double, double); // dichiarazione funzione uniforme
int main() {
    int n = dado(32);
    double x = uniforme(0.9, 3.2);
    double y = uniforme(0, 2);
    ≡
    return 0;
}                                   } implementa main()
int dado(int nfac) {
    ≡
}                                   } implementa dado()
double uniforme(double a, double b) {
    ≡
    return valore;
}                                   } implementa uniform
```

creare [funzioni.h]

Solo dichiarazioni:

```
int dado(int);
double uniforme(double, double);
```

[funzioni.c]

contiene solo implementazioni funzioni:

nessun main()

```
int dado(int n) {
    ...
}
double uniforme(double a,
                double b) {
    ...
}
```

gcc -c funzioni.c ⇒ crea funzioni.o

compila ma non crea eseguibile

[prova.c]

```
#include <stdio.h>
#include "funzioni.h"

int main() {
    ...
}
```

gcc -o app.exe prova.c funzioni.o

contiene dado() uniforme() già compilate

programma.c

```
// dichiarazione funzioni:
   ===
int main() {
   ===
}
// implementazione funzioni:
   ===
```

Dichiarazione ed implementazione devono avere
- lo stesso tipo di ritorno
- lo stesso numero di argom.
- gli stessi tipi di argomenti
- lo stesso ordine di argomenti

```
double mysqrt(double y) {
double risultato;
   ===  } metodo babilonese
```

Calcolo risultato

```
return 0;
```
errore

```
return risultato;
}
```

```
double z = mysqrt(2);        // z = 0.
z = mysqrt(3.1);             // z = 0
```

chiede $L \in [0,d]$ con

```
do {
    printf("inserisci L tra %f e %f: ", 0, d);
    scanf("%lf", &L);
} while( L < 0 || L > d );
```

$d \in [0,5]$ con

input: estremo inferiore
estremo superiore
stringa di carattere

output: double

```
double d = insert("distanza righe", 0, 5);
double L = insert("lunghezza asse", 0, d);
double t = insert("tempo volo", 0, 2.3);
```

Dichiarazione:

```
double insert( char[100], double, double );
double insert( char*, double, double );
```

equivalenti

Implementazione:

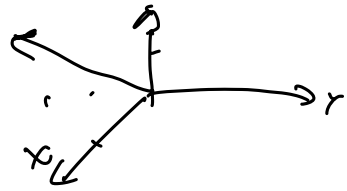
```
double insert( char nome[100], double x1, double x2){
    double risultato;
    do {
        printf("%s in [%f, %f]: ", nome, x1, x2 );
        scanf("%lf", &risultato);
    } while( risultato < x1 || risultato > x2);
    return risultato;
}
```

Chiamata a uso della funzione:

```
double z = insert("pippo", -2, 3.1);
```

z sarà in [-2, 3.1];

```
double vettore[3] = {2, -2, 3};
```




```
double modulo = magnitude(vettore, 3);
```

$$|\vec{v}| = \sqrt{\sum_i v_i^2}$$

```
double v2[3] = {1, 1, 13};
modulo = magnitude(v2, 3);
```

$\vec{v_i} \cdot \vec{vett} = v_{i1} \cdot vett_1 + v_{i2} \cdot vett_2 + v_{i3} \cdot vett_3$

```
double r = scalar(vettore, v2, 3);
```

Dichiarazione:

```
double magnitude(double [3], int);
double scalar(double [3], double[3], int);
```

metodo alternativo molto più diffuso

```
double magnitude(double *, int);
double scalar(double *, double*, int);
```

Implementazione:

lunghezza array

```
double magnitude( double *V, int N ) {
    double r = 0;
    int i;
    for (i=0; i<N; i++) {
        r += V[i]*V[i];
    }
    r = sqrt(r);
    return r;
}
```

// $r = \sum_{i}^{3} V_i^2$

// $r = \sqrt{r}$

V[i] ≡ *(V+i);

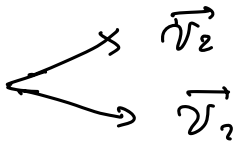
```
double V[3], W[7], Z[19];
double mv = magnitude( V, 3);
mw = magnitude( W, 7);
mz = magnitude( Z, 19);

x = magnitude( W, 6 )
```

Calcola modulo con i primi 6 elementi di W

Implementazione:

```
double scalar(double* V1, double * V2, int N) {
    double x=0;
    for(int i=0; i<N; i++) {
        x += V1[i] * V2[i];
    }
    return x;
}
```

$$\vec{v}_1 \cdot \vec{v}_2 = |\vec{v}_1| \cdot |\vec{v}_2| \cos\theta_{12}$$




```
double costeta = scalar(V1, V2, 3) / magnitude(V1, 3)
                                   / magnitude(V2, 3);

scalar(V1,V2,3) / (magnitude(V1,3) * magnitude(V2,3))

double thetaV(double* V1, double* V2, int N) {
  double x =
    scalar(V1,V2,3) / (magnitude(V1,3) * magnitude(V2,3));
  return acos(x);
}

    double theta = thetaV(V1, V2, 3);

Calcolo media:
    double dati[] = {1, 2, -1, -2, -3, 2, 4, 9};
    double m = media(dati, 8);

Dichiar:  double media(double*, int);

Implementazione:

double media(double* V, int N) {
    double somma = 0;
    for(int i=0; i<N; i++) {
        somma += *(V+i); // V[i]
    }
    somma /= N;
    return somma;
}
```

```
int main() {
    → calcolatto( . — );
}
```

inutile

```
int stamp (char *m) {
    printf("%s", m);
    printf("\n");
    return 0;
}
```

printf("%s\n", m);

Funzione inutile

Funzioni senza valore di ritorno: void

```
void func(--) {
    =
    // non serve return.
}

int x = func(--);
```

```
int main() {
    func(--);
    srand48(time(0));
    int n = erand48(); // erand48() non void
    scriviSuFile(dati, N, "dati.txt");
}
```

```
void scriviSuFile(double* V, int N, char* nome) {
    FILE* pf;
    pf = fopen(nome, "w+");
    if(!pf) {
        printf("errore! exit\n");
        exit(-1);
    }
    for(int i=0; i<N; i++) {
        fprintf(pf, "%.3f\n", V[i]);
    }
    fclose(pf);
} // fine scriviSuFile
```

!pf ≡ 1 => pf=0.

errore apertura file

Passaggio per valore e per puntatore

```
int main() {
    double x = 2;
    double z = sqrt(x);
}
```

main: x
2

main: z
1.4...

x non modificato da sqrt()

```
double sqrt(double a) {
```

passaggio per valore.

sqrt: a
2

```
double prova(double *a) {
    double r = sqrt(*a);
    *a = -2; //
    return r;
}

int main() {
    double x = 9;
    printf("x= %f\n", x);
    double y = &x;
    double z = prova(y);
    printf("z = %f\n", z);
    printf("x = %f\n", x);
}
```

passaggio per puntatore

scrittura che modifica argomento

prova modifica x tramite y

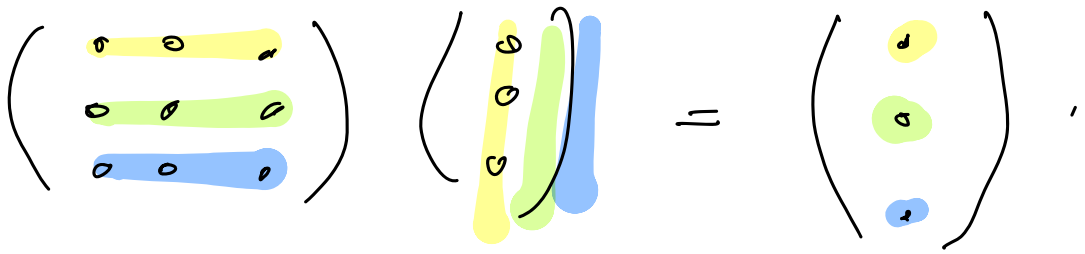
```
x = 9
z = 3
x = -2
```

prova() ha modificato variabile x di main()

```
double dati[1000] = {-1, -1, ---, -1};
for (int i=0; i<1000; i++) {
    dati[i] = -1;
}

inizializza(dati, 1000, -1);
double voti[160];
inizializza(voti, 160, 18);
```

```
void inizializza(double* v, int N, double val){
    for(int i=0; i<N; i++){
        v[i] = val;
    }
}

double mat[7][4];
assegnaMat(mat, 7, 4, 1);

void assegnaMat(double mat[][4], int nR, int nC
                , double val){
    for(int i=0; i<nR; i++){
        for(int j=0; j<nC; j++){
            *(*(mat+i)+j) = val;
        } // ciclo colonne
    } // ciclo righe
}

void printMat(double m[][10], int nR, int nC){
    for(int i=0; i<nR; i++){
        for(int j=0; j<nC; j++){
            printf("%.3f\t", *(*(m+i)+j));
        } // ciclo colonne
        printf("\n");
    } // ciclo righe
```

nC ∈ [0, 10]



```
double mat[3][3];
double v[3];
```

Funzioni non restituiscono array

```
double risultato[3];
prodotto(mat, v, 3, risultato);
                    risultato del prodotto.
```

```
void prodotto(double m[][3], double v[3], int N,
                                , double r[3]    ) {
  for(int i=0; i<N; i++) {
    r[i]=0;
    for(int j=0; j<N; j++) {
      r[i] += m[i][j] * v[j];
    }
  }
}
```

array / puntatore usato per restituire un array
possibile con funzione void in C;