Operatori: Relazionali ==, !=, <, >, <=, >= **Logici** ! (NOT) && (AND) || (OR)

Struttura:
if (espressione) { seq.1 di istruzioni } else { seq.2 di istruzioni }

```
while (condition)
        statement;

cont = 0;
while (cont < N) {
        …;
        cont++;
}
```

```
for(init_expr; condition; loop_expr)
        statement;

for (cont = 0; cont < N; cont++) {
        …;
        …;
}
```

#include <stdio.h> printf (*stringa_controllo, elemento, ...*); scanf(*stringa_controllo, &elemento, ...*);
%d intero decimale, %f floating point, %c carattere, %s stringa

Array
int n[5]; int n[5] = {1, 2, 3, 4, 5};
int n[5] = {13}; tutti gli altri elementi sono posti a 0
n[0] = 3; printf ("%d", n[0]); stampa 3
scanf ("%d", &n[1]); Se l'utente inserisce 16, n[1] assume valore 16
char stringa[] = "word"; L'array stringa contiene i caratteri 'w', 'o', 'r', 'd', '\0' (terminatore di stringa).

#include <string.h>
strcat(char *dest, const char *src); Concatena src alla stringa dest.
strcpy(char *s1, const char *s2); Copia s2 in s1, incluso il carattere di terminazione \0.
int strlen(const char *s); Restituisce la lunghezza della stringa s.
int strcmp(const char *s1, const char *s2); Confronta la stringa s1 con s2, restituisce 0 se sono uguali, un numero negative se s1 precede s2, un numero positivo altrimenti.

Matrici
int M[5][5]; float F[20][20][30]; scanf("%d",&M[i][j]); printf("%d",M[i][j]);

Struct
```
struct {  char via[20], citta[20];
        int numero, CAP; } indirizzo;
strcpy(indirizzo.via, "Ponzio");
indirizzo.numero = 34;
strcpy(indirizzo.citta, "Milano");
printf("%d\n", indirizzo.numero); > 34
printf("%s\n", indirizzo.citta); > Milano
scanf("%s", indirizzo.via); > Ponzio
scanf("%d", &indirizzo.CAP); > 20133
struct …  rec1, rec2;
rec2 = rec1;//campi ordinatamente copiati
rec1==rec2  è sintatticamente errato
```

Typedef typedef struct { int giorno; int mese; int anno; } data;

Puntatori typedef int *intRef; intRef primoPunt, secondoPunt; equivalgono a int * terzoPunt;
primoPunt = &x; dereferenziazione: *primoPunt equivale a x
p->PrimoCampo = 12; equivale a (*p).PrimoCampo = 12;

Funzioni

```
<tipo restituito><nome della funzione>(<lista degli argomenti>) {
        <variabili locali>
        <corpo della funzione>
}
```

```
double sum (TipoArray a, int n){…}  double sum (double a[], int n) {…}  double sum (double *a, int n) {…}
sum (V, 50);

int f(int m[][N], int v[]){…}     f(mat, vet);
```

File

```
FILE * fopen (char * nomefile, char * modalità)    "r" lettura, "w" scrittura, "a" scrittura fine file (append)
int fclose (FILE * fp)
int feof (FILE * fp)  restituisce 0 (falso) se NON si è alla fine
int fprintf (FILE * fp, str_di_controllo, elementi)
int fscanf (FILE * fp, str_di_controllo, indirizzo_elementi)
```

Liste

```
ref = (PTD) malloc( sizeof(TipoDato) );          free(ref);

char * ptr;
ptr = (char *) malloc( sizeof(char) );
*ptr = 'a';
printf("Carattere: %c\n", *ptr);
free( ptr );
ptr=NULL;

typedef struct EL {
        TipoElemento info;
        struct EL * prox;
} ElemLista;
typedef ElemLista * ListaDiElem;
```

```
int Dimensione(ListaDiElem lista) {
        int count = 0;
        while( lista!=NULL) {
          lista = lista–>prox;
          count++;
        }
        return count;
}
```

```
ListaDiElem InsInTesta ( Lista lista,
                        TipoElemento elem ) {
        ListaDiElem punt;

        punt = (Lista) malloc(sizeof(ElemLista));
        punt–>info = elem;
        punt–>prox = lista;

        return  punt;
}

Chiamata: lista1 = InsInTesta( lista1, elemento );
```

```
ListaDiElem InsInFondo( Lista lista,TipoElemento elem ) {
  Lista punt;
  if( lista==NULL ) {
      punt = malloc( sizeof(ElemLista) );
      punt–>prox = NULL;
      punt–>info = elem;
      return  punt;
  }
  else {   lista–>prox = InsInFondo( lista–>prox, elem );
      return lista;   }
}
Chiamata :    lista1 = InsInFondo( lista1, Elemento );
```

```
ListaDiElem InsInOrd( ListaDiElem lista, TipoElemento elem ) {
        ListaDiElem punt, puntCor = lista, puntPrec = NULL;
        while ( puntCor != NULL  &&  elem > puntCor–>info ) {
            puntPrec = puntCor;     puntCor = puntCor->prox;
        }
        punt = (ListaDiElem) malloc(sizeof(ElemLista));
        punt–>info = elem; punt–>prox = puntCor;
        if ( puntPrec != NULL ) {  puntPrec–>prox = punt; return lista; } else return punt;
   }
Chiamata : lista1 = InsInOrd( lista1, elemento );

ListaDiElem Cancella( ListaDiElem lista, TipoElemento elem ) {
        ListaDiElem puntTemp;
        if( lista!=NULL)
                if( lista–>info == elem ) {
                        puntTemp = lista–>prox;
                        free( lista );
                        return puntTemp; // per cancellare tutte le istanze: return Cancella(PuntTemp, Elem);
                }
                else
                        lista–>prox = Cancella( lista–>prox, elem );
        return lista;
}
Chiamata : lista1 = Cancella( lista1, elemento );

void DistruggiListaRic ( ListaDiElem lista ) {
   if ( lista!=NULL ) {
        DistruggiListaRic( lista–>prox ); free( lista );
   }
}

void VisualizzaLista( ListaDiElem lista ) {
   if ( lista==NULL )
       printf(" ---| \n");
   else {
         printf(" %d\n ---> ", lista–>info);
         VisualizzaLista( lista–>prox );
   }
}
```

Alberi

```
typedef struct Nodo {
        Tipo dato;
        struct Nodo *left, *right;
} nodo;
typedef nodo * tree;
```