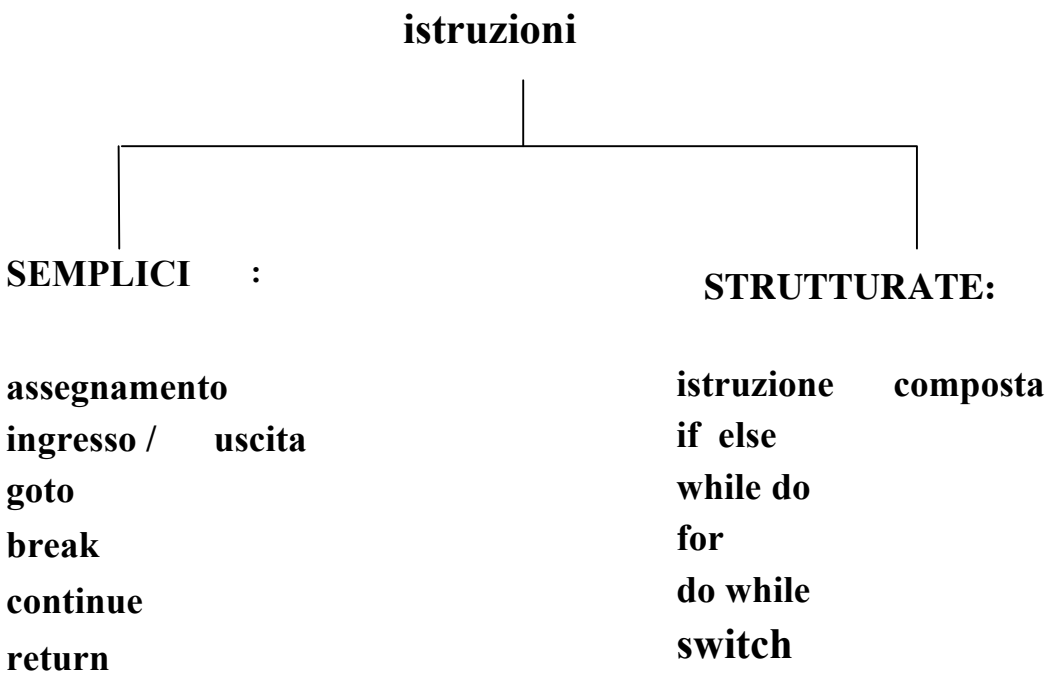

# Linguaggio C: Istruzioni

# Istruzioni: classificazione

In C, le istruzioni possono essere classificate in due categorie:

- istruzioni **semplici**
- istruzioni **strutturate**: si esprimono mediante composizione di altre istruzioni (semplici e/o strutturate).



# Istruzioni Semplici

# Istruzione di Assegnamento

- E` l'istruzione con cui si modifica il valore di una variabile.

- Mediante l'assegnamento, si scrive un nuovo valore nella cella di memoria che rappresenta la variabile specificata.

**Sintassi:**

<istruzione-assegnamento>::= <identificatore-variabile>=<espressione>;

**Ad esempio:**

```
int A, B;

A=20;
B=A*5; /* B=100 */
```

**Compatibilita` di tipo ed assegnamento:**

In un assegnamento, l'identificatore di variabile e l'espressione devono essere dello stesso tipo (eventualmente, conversione implicita oppure ***coercizione***).

# Assegnamento e Coercizione

Facendo riferimento alla gerarchia dei tipi semplici:

```
int < long < float < double < long double
```

consideriamo l'assegnamento:

V=E;

Quando la variabile V e` di un tipo di *rango inferiore* rispetto al tipo dell'espressione E l'assegnamento prevede la conversione forzata (***coercizione***) di E nel tipo di V.

**Ad esempio:**

```
float k=1.5;
int x;
x=k;   /*il valore di k viene convertito forzatamente nella
   sua parte intera: x assume il valore 1  ! */
```

## Esempio:

```
main()
{/*definizioni: */
char y='a'; /*codice(a)=97*/
int     x,X,Y;
unsigned int Z;
float SUM;
double r;

/* parte istruzioni: */
X=27;
Y=343;
Z = X + Y -300;
X = Z / 10 + 23;
Y = (X + Z) / 10 * 10; /* qui X=30, Y=100, Z=70 */
X = X + 70;
Y  = Y % 10;
Z = Z + X -70;
SUM = Z * 10;  /* X=100, Y=0, Z=100 , SUM=1000.0*/
x=y;     /* char -> int: x=97*/
x=y+x; /*x=194*/
r=y+1.33; /* char -> int -> double*/
x=r; /* coercizione -> troncamento: x=98*/
}
```

# Assegnamento come operatore

Formalmente, **l'istruzione di assegnamento è un'espressione.**
Infatti:

- Il simbolo = e` un operatore:
  - → l'istruzione di assegnamento e` una espressione
  - → ritorna un valore:
    - il valore ritornato e` quello assegnato alla variabile a sinistra del simbolo =
    - il tipo del valore ritornato e` lo stesso tipo della variabile oggetto dell'assegnamento

**Ad esempio:**

```
int valore=122;
int K, M;

K=valore+100; /* K=122;l'espressione
                 produce il
                 risultato 222 */
M=(K=K/2)+1;  /* K=111, M=112*/
```

# Assegnamento abbreviato

In C sono disponibili operatori che realizzano particolari forme di assegnamento:

- operatori di incremento e decremento
- operatori di assegnamento abbreviato
- operatore sequenziale

- **Operatori di incremento e decremento:**
  Determinano l'incremento/decremento del valore della variabile a cui sono applicati.
  Restituiscono come risultato il valore incrementato/decrementato della variabile a cui sono applicati.

```
int A=10;

A++;/*equivale a: A=A+1; */
A--;/*equivale a: A=A-1; */
```

**Differenza tra notazione prefissa e postfissa:**

- Notazione **Prefissa**: (ad esempio, ++A) significa che l'incremento viene fatto prima dell'impiego del valore di A nella espressione.
- Notazione **Postfissa**: (ad esempio, A++) significa che l'incremento viene effettuato dopo l'impiego del valore di A nella espressione.

# Incremento & decremento: esempi

```
int A=10, B;
char C='a';

B=++A;          /*A e B valgono 11 */
B=A++;          /* A=12, B=11 */
C++;                    /* C vale 'b' */

int i, j, k;
k = 5;
i = ++k; /*  i = 6, k = 6 */
j = i + k++; /* j=12, i=6,k=7 */

j = i + k++; /*equivale a:j=i+k; k=k+1;*/
```

- In C l'ordine di valutazione degli operandi non e' indicato dallo standard: si possono scrivere espressioni il cui valore e` difficile da predire:

```
k = 5;
j = ++k * k++; /* quale effetto ?*/
```

# Operatori di assegnamento abbreviato

E` un modo sintetico per modificare il valore di una variabile.

Sia v una variabile, op un'operatore (ad esempio, +,-,/, etc.), ed e una espressione.

***v op = e***

è quasi equivalente a:

***v = v op (e)***

**Ad esempio:**

```
k += j;              /*  equivale a k = k + j  */
k *= a + b;          /*  equivale a k = k * (a + b)  */
```

Le due forme sono ***quasi*** equivalenti perchè:

- in ***v op= e*** v viene valutato una sola volta;
- in ***v = v op (e)*** v viene valutato due volte.

# Operatore sequenziale

Un'espressione sequenziale (o di ***concatenazione***) si ottiene concatenando tra loro piu` espressioni con l'operatore virgola (,).

**(<espr1>, <espr2>, <espr3>, .. <esprN>)**

- Il risultato prodotto da un'espressione sequenziale e` il risultato ottenuto dall'ultima espressione della sequenza.
- La valutazione dell'espressione avviene valutando nell'ordine testuale le espressioni componenti, **da sinistra verso destra**.

**Esempio:**

```
int A=1;
char B;
A=(B='k', ++A, A*2);        /* A=4 */
```

# Precedenza e Associativita` degli Operatori

In ogni espressione, gli operatori sono valutati secondo una **precedenza** stabilita dallo standard, seguendo opportune regole di **associativita`**:

- La **precedenza** (o priorita`) indica l'ordine con cui vengono valutati operatori diversi;
- L'**associativita`** indica l'ordine in cui operatori di pari priorita` (cioe`, stessa precedenza) vengono valutati.

➔ E` possibile forzare le regole di precedenza mediante l'uso delle parentesi.

# Precedenza & associativita`
# degli operatori C

| Precedenza | Operatore | Simbolo | Associatività |
|---|---|---|---|
| 1 (max) | chiamate a funzione<br>selezioni | `()`<br>`[]` `->` `.` | a sinistra |
| 2 | operatori unari: | | a destra |
| | op. negazione | `!` `~` | |
| | op. aritmetici unari | `+` `-` | |
| | op. incr. / decr. | `++` `--` | |
| | op. indir. e deref. | `&` `*` | |
| | op. sizeof | `sizeof` | |
| 3 | op. moltiplicativi | `*` `/` `%` | a sinistra |
| 4 | op. additivi | `+` `-` | a sinistra |

# Precedenza & associativita`
# degli operatori C (continua)

| Precedenza | Operatore | Simbolo | Associatività |
|---|---|---|---|
| 5 | op. di shift | >> << | a sinistra |
| 6 | op. relazionali | < <= > >= | a sinistra |
| 7 | op. uguaglianza | == != | a sinistra |
| 8 | op. di AND bit a bit | & | a sinistra |
| 9 | op. di XOR bit a bit | ^ | a sinistra |
| 10 | op. di OR bit a bit | \| | a sinistra |
| 11 | op. di AND logico | && | a sinistra |
| 12 | op. di OR logico | \|\| | a sinistra |
| 13 | op. condizionale | ?...: | a destra |
| 14 | op. assegnamento e sue varianti | = += -= *= /= %= &= ^= \|= <<= >>= | a destra |
| 15 (min) | op. concatenazione | , | a sinistra |

# Esempi

Sia V=5, A=17, B=34. Determinare il valore delle seguenti espressioni :

| | | |
|---|---|---|
| A<=20 \|\| A>=40 | → | ***vero*** {A<20} |
| ! (B=A*2) | → | ***falso*** {B=34=17*2} |
| A<=B && A<=V | → | ***falso*** {A>V} |
| A<=(B&&A)<=V | → | ***vero*** {A<=1<=V} |
| A>=B && A>=V | → | ***falso*** {A<B} |
| ! (A<=B && A<=V) | → | ***vero*** |
| ! (A>=B) \|\| !(A<=V) | → | ***vero*** |
| (A++, B=A, V++, A+B+V++) | → | ***42*** (A=18, B=18, V=7) |

# INPUT/OUTPUT

- L'immissione dei dati di un programma e l'uscita dei suoi risultati avvengono attraverso operazioni di lettura e scrittura.

- Il C non ha istruzioni predefinite per l'input/output.

- In ogni versione ANSI C, esiste una *Libreria Standard* (**stdio**) che mette a disposizione alcune funzioni (dette *funzioni di libreria*) per effettuare l'input e l'output da e verso dispositivi.

- **Dispositivi standard di input e di output:**
  - per ogni macchina, sono periferiche predefinite (generalmente tastiera e video).

# INPUT/OUTPUT

Le dichiarazioni delle funzioni messe a disposizione da tale libreria devono essere *incluse* nel programma:

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

- **`#include`** e` una direttiva per il **preprocessore** C:
- nella fase precedente alla compilazione del programma ogni direttiva "**#**..." viene eseguita, provocando delle modifiche testuali al programma sorgente.
- Nel caso di **`#include <nomefile>:`**

  viene sostituita l'istruzione stessa con il contenuto del file specificato.

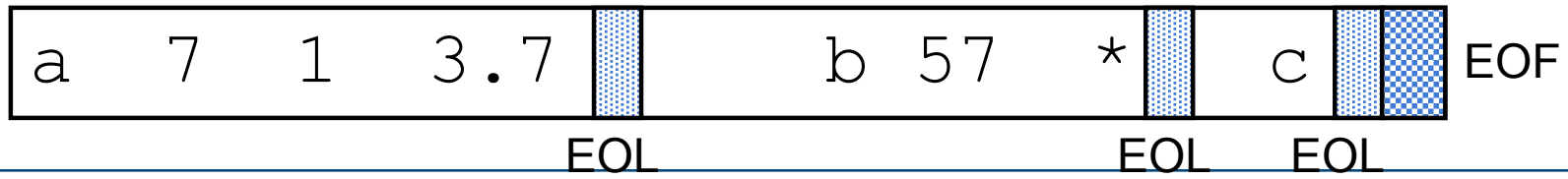
# INPUT/OUTPUT

Il C vede le informazioni lette/scritte da/verso i dispositivi standard di I/O come file *sequenziali*, cioè **sequenze di caratteri** (o *stream*).

- Gli *stream* di input/output possono contenere dei caratteri di controllo:
  - End Of File (EOF)
  - End Of Line (EOL)

**Sono disponibili funzioni di libreria per:**

- Input/Output a caratteri
- Input/Output a stringhe di caratteri
- Input/Output con formato



# INPUT/OUTPUT CON FORMATO

- Nell'I/O con formato occorre specificare il formato (*tipo*) dei dati che si vogliono leggere oppure stampare.

- Il formato stabilisce:
  - come interpretare la sequenza dei caratteri immessi dal dispositivo di ingresso (nel caso della lettura)
  - con quale sequenza di caratteri rappresentare in uscita i valori da stampare (nel caso di scrittura)
- Il formato viene specificato mediante apposite ***direttive di formato***; ad esempio %d, %f, %s ecc.

# LETTURA CON FORMATO: `scanf`

E' una particolare forma di assegnamento: la scanf assegna i valori letti alle variabili specificate come argomenti (nell'ordine di lettura).

```
scanf(<stringa-formato>, <sequenza-variabili>);
```

**Ad esempio:**

```
int X;
float Y;
scanf("%d%f", &X, &Y);
```

# LETTURA CON FORMATO: scanf

```
scanf(<stringa-formato>, <sequenza-variabili>);
```

- **scanf** legge una serie di valori in base alle specifiche contenute in **<stringa-formato>** e memorizza i valori letti nelle variabili specificate in **<sequenza-variabili>.**
- Se la **<stringa-formato>** contiene N direttive (del tipo %..), è necessario che le variabili specificate nella **<sequenza-variabili>** siano esattamente N.

- restituisce il numero di valori letti e memorizzati, oppure EOF in caso di end of file :

```
int X, Y, K;
K = scanf("%d%d", &X, &Y);
```

  - se vengono immessi da input i due valori 100 e -25, le variabili X,Y e K assumeranno i seguenti valori:

    **X=100  Y=-25  K=2**

# scanf & formato

Ogni direttiva di formato prevede dei separatori specifici:

| Tipo di dato | Direttive di formato | Separatori |
|---|---|---|
| Intero | %d, %x, %u, etc. | Spazio, EOL, EOF. |
| Reale | %f %g etc. | Spazio, EOL, EOF |
| Carattere | %c | Nessuno |
| Stringa | %s | Spazio, EOL, EOF |

```
int  X; float Y; char Z;
scanf("%d%f%c", &X, &Y, &Z);
```

**Osservazioni:**

- La <stringa-formato> puo` contenere dei caratteri qualsiasi (che vengono scartati, durante la lettura), che rappresentano separatori aggiuntivi rispetto a quelli standard.

  Ad esempio: `scanf("%d:%d:%d", &A, &B, &C);`

  richiede che i tre dati da leggere vengano immessi separati dal carattere "**:**".

# SCRITTURA CON FORMATO: `printf`

La **`printf`** viene utilizzata per fornire in uscita il valore di una variabile, o, più in generale, il risultato di una espressione:

```
printf(<stringa-formato>,<sequenza-elementi>);
```

- Anche in scrittura e` necessario specificare (mediante una **`<stringa-formato>`**) il formato dei dati che si vogliono stampare.
- **`<sequenza-elementi>`** e` una lista di ***espressioni*** (tante quante le direttive di formato contenute nella **`<stringa-formato>`**) .

**Ad esempio:**

```
int X=19;
float Y=2.5;
printf("%d%f", X, X+Y);
```

# printf

```
printf(<stringa-formato>,<sequenza-elementi>);
```

- **printf** scrive una serie di valori in base alle specifiche contenute in **<stringa-formato>.**
- I valori visualizzati sono i risultati delle espressioni indicate nella **<sequenza-elementi>.**
- La **printf** restituisce il numero di caratteri scritti.
- La stringa di formato della **printf** può contenere sequenze costanti di caratteri da stampare (nell'ordine indicato).

**Ad esempio:**

```
int X=19, K;
float Y=2.5;
K=printf("Risultato: %d%f\n", X, X+Y);
```

**Effetti:**

```
int X=19, K;
float Y=2.5
```

# FORMATI COMUNI

- Formati più comuni:

| | |
|---|---|
| `int` | `%d` |
| `float` | `%f` |
| `carattere singolo` | `%c` |
| `stringa di caratteri` | `%s` |

- Caratteri di controllo:

| | |
|---|---|
| `newline` | `\n` |
| `tab` | `\t` |
| `backspace` | `\b` |
| `form feed` | `\f` |
| `carriage return` | `\r` |

- Per la stampa del carattere ' `%` ' si usa: `%%`

# ESEMPIO

```
#include <stdio.h>
main()
{int      k;
scanf("%d",&k);
printf("Quadrato di %d: %d\n",k,k*k);
}
```

**Esempio:**

Se in ingresso viene immesso il dato: `3`

La `printf` stampa:

```
Quadrato di 3: 9

_
```

# ESEMPIO

Rivediamo l'esempio visto inizialmente:

```
/*programma che, letti due numeri a terminale, ne stampa
  la somma*/

#include <stdio.h>

main()
{ int X,Y;  /* p. dichiarativa */

  scanf("%d%d",&X,&Y);/*lettura dei due dati*/
  printf("%d",X+Y);/* stampa della loro somma */
}
```

**Dati da input i due valori 26 e -32, il programma stampa:**

```
-6
```

# ESEMPIO

```
scanf("%c%c%c%d%f", &c1,&c2,&c3,&i,&x);
```

- Se in ingresso vengono dati:

```
ABC 3 7.345
```

- la **scanf** effettua i seguenti assegnamenti:

```
char c1   'A'
char c2   'B'
char c3   'C'
int i          3
float x    7.345
```

# ESEMPIO

```
#include <stdio.h>
main()
{char Nome='A';
char Cognome='C';
printf("%s\n%c.  %c.  \n%s\n","Programma  scritto  da:",  Nome,
   Cognome,"Fine");
}
```

**Stampa:**

```
Programma scritto da:
A. C.
Fine
-
```

# Esempio

**Esempio:**

stampa della codifica (decimale, ottale e esadecimale) di un carattere dato da input.

```
#include <stdio.h>

main()
{ char a;

printf("Inserire un carattere:  ");
scanf("%c",&a);
printf("\n%c vale %d in decimale, %o in ottale \
       e %x in hex.\n",a, a, a, a);
}
```

## Effetti dell'esecuzione:

```
Inserire un carattere:  A

A vale 65 in decimale, 101 in ottale e 41 in hex.
```

# Esercizio

## Calcolo dell'orario previsto di arrivo.

Scrivere un programma che legga tre interi positivi da terminale, rappresentanti l'orario di partenza (ore, minuti, secondi) di un vettore aereo, legga un quarto intero positivo rappresentante il tempo di volo in secondi e calcoli quindi l'orario di arrivo.

## Prima specifica:

```
main()
{  /*dichiarazione variabili: occorrono tre
   variabili intere per l'orario di partenza ed
   una variabile intera per i secondi di volo.  */
   /*leggi i dati di ingresso */
   /*calcola l'orario di arrivo */
   /*stampa l'orario di arrivo */
}
```

## Soluzione:

```
#include <stdio.h>
main()
{ /* dichiarazione dati */
long unsigned int Ore, Minuti, Secondi, TempoDiVolo;
/* leggi i dati di ingresso*/
printf("%s\n","Orario di partenza (hh,mm,ss)?");
scanf("%ld%ld%ld", &Ore, &Minuti, &Secondi);
printf("%s\n","Tempo di volo (in sec.)?");
scanf("%ld", &TempoDiVolo);
/* calcola l'orario di arrivo*/
Secondi = Secondi + TempoDiVolo;
Minuti = Minuti + Secondi / 60;
Secondi = Secondi % 60;
Ore = Ore + Minuti / 60;
Minuti = Minuti % 60;
Ore = Ore % 24;
/* stampa l'orario di arrivo*/
printf("%s\n","Arrivo previsto alle (hh,mm,ss):");
printf("%ld%c%ld%c%ld\n",Ore,':',Minuti,':', Secondi);
}
```

# Dichiarazioni e Definizioni

Nella parti dichiarative di un programma C possiamo incontrare:

- definizioni (di variabile, o di funzione)
- dichiarazioni (di tipo o di funzione)

**Definizione:**

Descrive le proprieta` dell'oggetto definito e ne determina l'esistenza.

Ad esempio:

```
int V; /* definizione della variabile intera V */
```

**Dichiarazione:**

Descrive soltanto delle proprieta` di oggetti, che verranno (eventualmente) creati mediante definizione.

Ad esempio: *dichiarazione* di un tipo di dato non primitivo:

```
typedef .... newT; /* newT e` un tipo non primitivo*/
```

# Dichiarazione di tipo

La dichiarazione di tipo serve per introdurre tipi non primitivi.

```
typedef <descrizione-nuovo-tipo> newT;
```

- si utilizza la parola chiave **`typedef`**.
- la dichiarazione associa ad un tipo di dato non primitivo un identificatore arbitrario (**`newT`**)
- le caratteristiche del nuovo tipo sono indicate in **`<descrizione-nuovo-tipo>`**

L'introduzione di tipi non primitivi aumenta la **leggibilita`** e **modificabilita`** del programma.

# Tipi scalari non primitivi

In C sono possibili dichiarazioni di tipi scalari non primitivi:

- tipi **ridefiniti**
- [tipi **enumerati**] (non li tratteremo)

**Tipo ridefinito:**

Si ottiene associando un nuovo identificatore a un tipo gia` esistente (primitivo o non).

**Sintassi:**

**typedef <id-tipo-esistente> <id-nuovo-tipo>;**

**Esempio:**

```
typedef int MioIntero; /*     MioIntero e` un tipo non
                               primitivo che ridefinsce il
                               tipo int*/
MioIntero X;           /* X e` di tipo MioIntero */
int Y;                 /* Y e` di tipo int */
```

➔ X e Y rappresentano entrambi valori interi, ma ***nominalmente*** sono di tipo diverso.

# Equivalenza tra tipi di dato

Quando due variabili hanno lo stesso tipo?
Dipende dalla ***realizzazione*** del linguaggio.

In generale, vi sono due possibilita` :

- equivalenza **strutturale**
- equivalenza **nominale**

**Equivalenza strutturale:**

due dati sono considerati di tipo equivalente se hanno la stessa struttura.

Ad esempio:

```
typedef int MioIntero;
MioIntero X;
int Y;
```

**Se la realizzazione di C prevede equivalenza strutturale:**

X e Y sono **dello stesso tipo**.

# Equivalenza tra tipi di dato

**Equivalenza nominale:**

due dati sono considerati di tipo equivalente se sono stati definiti usando lo stesso identificatore di tipo.

Ad esempio:

```
typedef int MioIntero;
MioIntero X;
int   Y;
```

**Se la realizzazione di C prevede equivalenza nominale:**

X e Y sono **di tipo diverso**.

# Equivalenza tra tipi di dato

L'equivalenza nominale e` piu` restrittiva:

non e` detto che dati strutturalmente equivalenti siano anche nominalmente equivalenti (dipende dalla realizzazione del linguaggio!)

## Equivalenza di tipo in C:

- Lo standard non stabilisce il tipo di equivalenza da adottare.
- Per garantire la portabilita` , e` necessario sviluppare programmi che presuppongano una realizzazione basata su equivalenza nominale.

# Programmazione strutturata (Dijkstra, 1969)

La programmazione strutturata nasce come proposta per regolamentare e standardizzare le metodologie di programmazione.

**Obiettivo**:

rendere piu' facile la lettura dei programmi (e quindi la loro modifica e manutenzione).

**Idea di base:**

Ogni programma viene visto come un comando (complesso, o *strutturato*) ottenuto componendo altri comandi mediante alcune regole di composizione (***strutture di controllo***).

**Strutture di controllo:**

- **concatenazione** (o composizione, blocco);
- **alternativa** (o istruzione condizionale)
- **ripetizione** (o iterazione)

Si puo` dimostrare (teorema Jacopini-Böhm) che queste strutture sono sufficienti per esprimere qualunque algoritmo.