

## PRIMI ELEMENTI DI PROGRAMMAZIONE (APPUNTI INTEGRATIVI)

### PSEUDOCODIFICA

E' un altro modo, oltre ai diagrammi di flusso, per rappresentare un algoritmo. Utilizza una sequenza di comandi simile al linguaggio naturale

Esempio: algoritmo giorno del mese.

Questo algoritmo calcola in quale giorno della settimana cade una specifica data acquisita come input:

| FLOW CHART                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | PSEUDOCODIFICA                                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> graph TD     Inizio([Inizio]) --&gt; Leggi[/Leggi giorno, mese, anno/]     Leggi --&gt; Mese{mese &lt; 3}     Mese -- Falso --&gt; Join(( ))     Mese -- Vero --&gt; Anno[anno = anno - 1]     Anno --&gt; MeseAdj[mese = mese + 12]     MeseAdj --&gt; Join     Join --&gt; K[k = anno - (anno/100)*100]     K --&gt; J[j = anno/100]     J --&gt; D1["d = giorno - 1 + ((13*(mese+1))/5) + k + k/4 + j/4 + 5*j"]     D1 --&gt; D2["d = d - (d/7)*7"]     D2 --&gt; Scrivi[/Scrivi Scrivi d/]     Scrivi --&gt; Fine([Fine])         </pre> | <pre> INIZIO Leggi giorno, mese, anno se (mese&lt;=3) {     anno=anno+1     mese=mese+12 } k=anno - (anno/100)*100 j=anno/100; d=giorno-1+((13*(mese+1))/5)+k+k/4+j/4+5*j d=d-(d/7)*7; scrivi d FINE         </pre> |

### INDENTAZIONE

Indica il corretto allineamento di del codice in modo che istruzioni facenti parte dello stesso blocco siano scritte una sotto l'altra allineate. Un codice non indentato viene comunque compilato ma è meno leggibile.

Esempio:

| codice indentato                                                                                        | codice non indentato                                                                            |
|---------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| <pre> in eta=12; if (eta&lt;18)     printf("maggiorenne"); else     printf("minorenne");         </pre> | <pre> in eta=12; if (eta&lt;18) printf("maggiorenne"); else printf("minorenne");         </pre> |

Il codice indentato è più leggibile poiché è immediatamente visibile quali istruzioni verranno eseguite se eta>18 è vero e quali se eta>18 è falso

## LE 5 CARATTERISTICHE DEGLI ALGORITMI

- **Finitezza**

L'algoritmo deve terminare dopo l'esecuzione di un numero finito di istruzioni (passi)

Esempio di algoritmo che **NON rispetta** questa caratteristica:

1. acquisisci un numero naturale N
2. somma 1 a N
3. ripeti il passo 2

- **Non ambiguità (o "determinatezza")**

Ogni istruzione deve essere interpretabile in modo **univoco dall'esecutore**.

Esempio di ambiguità: "se N è molto maggiore di 10..."

- **Completezza**

L'algoritmo deve essere completo, ciò significa che devono essere previste delle istruzioni per qualsiasi valore inserito delle variabili di input.

Ad esempio, non è completo il seguente algoritmo:

1. acquisisci un numero N
2. calcola la radice quadrata

Non è completo perché se  $N < 0$  la radice quadrata non esiste e l'algoritmo non spiega cosa deve fare l'esecutore in questo caso.

Per essere completo è necessario scrivere:

1. acquisisci un numero N
2. se  $N < 0$   
scrivi "impossibile calcolare la radice quadrata"
3. altrimenti  
calcola la radice quadrata  
scrivi il valore calcolato

- **Generalità**

L'algoritmo deve risolvere **UNA CLASSE** di problemi e **NON UN SINGOLO** problema.

Ad esempio, l'algoritmo che calcola il giorno è generale perché determina il giorno della settimana per **OGNI** data fornita come input. Se calcolasse solo il giorno della settimana di una specifica data non sarebbe generale.

- **Determinismo o Riproducibilità**

Eseguendo infinite volte l'algoritmo con gli stessi dati di ingresso, esso deve fornire sempre gli stessi risultati in uscita. Si dice che l'algoritmo deve essere **deterministico**.

## PRINCIPALI TIPI DI DATO IN C

Sapere quanti byte occupano:

- **int**                      numero intero in complemento a 2 (ca2)                      2,4,8 byte  
dipende dalla piattaforma HW/SW
- **short**                    numero intero in ca2                      2 byte
- **long**                     numero intero in ca2                      4 byte
- **float**                    numero floating point a precisione singola                      4 byte
- **double**                  numero floating point a precisione doppia                      8 byte
- **char**                    carattere (memorizzato come numero intero ASCII)                      1 byte
- **bool**                    valore booleano ("true" o "false") (richiede l'inclusione del file <stdbool.h>) 1 byte

Una stringa viene memorizzata come array di caratteri:

char nome[]="Antonio";

nome →

|   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  |
| A | n | t | o | n | i | o | \0 |

Un array è una sequenza di variabili dello stesso tipo, in questo caso char, caratterizzata da un unico identificatore (in questo caso "nome") e in cui ogni singolo elemento è identificato dalla posizione nell'array. In questo caso

nome[2] è il carattere 't'

## SPECIFICATORI DI FORMATO

Gli specificatori di formato servono per specificare quale tipo di dato si vuole mostrare in un'operazione di output o acquisire in un'operazione di input che utilizzeremo (da imparare) sono i seguenti:

`"%i" o "%d"` → per i tipi di dato `int` (eventualmente anche per i `bool`)

`"%hi"` → per i tipi di dato `short`

`"%li"` → per i tipi di dato `long`

`"%f"` → per i tipi di dato `float`

`"%lf"` → per i tipi di dato `double`

`"%c"` → per i tipi di dati `char` ('A')

`"%s"` → per mostrare una stringa (vedremo più avanti)

```
char s[5]="ciao";
```

```
printf("%s",s);
```

`"%g"` → va bene per qualsiasi formato numerico ed elimina gli zero inutili dopo la virgola

## OPERATORI ALGEBRICI E LOGICI

### Operatori algebrici

Sono +, -, \*, /, % (% fa il MOD = resto della divisione intera)

non ci sono la potenza e la radice quadrata, per svolgere queste operazioni è necessario utilizzare le relative funzioni dopo aver importato la libreria matematica **<math.h>**

**pow(base, esponente);**                      potenza (attenzione restituisce un float)

**sqrt(valore);**                              radice quadrata (square root)

### Operatori di autoincremento/autodecremento

Notazione postfissa (l'operatore si trova dopo la variabile):

**y++**      (incrementa la variabile y di 1)      L'operatore è ++. La variabile è y

**y--**      (decrementa la variabile y di 1)      L'operatore è --. La variabile è y

Con la notazione postfissa (la più utilizzata), l'operazione avviene DOPO la valutazione dell'espressione

Esempio:

`printf("y=%i\n", y++);`    →    prima mostra il valore di y  
                                         poi incrementa y            `y=y+1;`

se l'operatore di autoincremento precede la variabile (notazione *prefissa*) l'operazione avviene PRIMA della valutazione di una espressione.

**++y**      (incrementa la variabile y di 1)

**--y**      (decrementa la variabile y di 1)      L'operatore è --. La variabile è y

Esempio:

`printf("y=%i\n", ++y);`    →    prima incrementa y    `y=y+1`  
                                         poi mostra il valore di y

Esempio:

`c=++a + b--`                      →    `a=a+1;`                      (prima fa questo)

`c=a+b;`                      (poi fa questo)

`b=b-1;`                      (poi fa questo)

**Esercizio:** provare il codice precedente assegnando dei valori a piacere ad a,b,c. Prevedere l'output. Mostrare il risultato sul monitor e verificare la previsione.

## Operatori algebrici compatti

`a+=b`       $\rightarrow$       `a=a+b;`

`a-=b`       $\rightarrow$       `a=a-b;`

`a*=b`       $\rightarrow$       `a=a*b;`

`a/=b`       $\rightarrow$       `a=a/b;`

## Operatori di confronto

Per la verifica delle condizioni si utilizzano i seguenti operatori di confronto:

`a==0`      **(Attenzione perché il compilatore non segnala un errore se metti un solo =, segnala solamente un warning! Ma se metti un solo = stai facendo un assegnamento errore molto frequente!)**

`a!=0`      a diverso da 0

`a<0`

`a<=0`

`a>0`

`a>=0`

```
4
5  int main()
6  {
7      int a=10;
8
9      if (a=11)
10         printf("vero");
11     else
12         printf("falso");
13
14     return 0;
15 }
```

**Errore!**  
**Scrive vero!**

## Esercizio:

Cosa mostra come output il seguente codice? (Verificare con codeblocks)

```
6  int main()
7  {
8      int a=0;
9      if (a=0)
10         printf("vero");
11     else
12         printf("falso");
13     return 0;
14 }
```

## Operatori logici (booleani)

Operatore      Simbolo in C

NOT

!

Tabella della verità

| A | NOT A |
|---|-------|
| F | V     |
| V | F     |

AND

&&

| A | B | A AND B |
|---|---|---------|
| F | F | F       |
| F | V | F       |
| V | F | F       |
| V | V | V       |

OR

||

| A | B | A OR B |
|---|---|--------|
| F | F | F      |
| F | V | V      |
| V | F | V      |
| V | V | V      |

```

5  int main()
6  {
7      int a=10;
8      int b=9;
9      if (a>10 || b>10)
10         printf("vero");
11     else
12         printf("falso");
13
14     return 0;
15 }
```

Scrivi vero o falso?

Provare

In C una variabile intera viene interpretata come un valore booleano dove il valore **0** rappresenta **false** e qualsiasi altro valore intero (1,2,...) rappresenta **true**.

Esempio:

```

5  int main()
6  {
7      int a=10;
8
9      if (a)
10         printf("vero");
11     else
12         printf("falso");
13
14     return 0;
15 }
```

Scrivi vero o falso?

Provare

## SEQUENZE DI ESCAPE

Le **sequenze di escape** sono particolari combinazioni di caratteri che vengono interpretati in modo speciale dal compilatore quando sono poste all'interno della funzione **printf()**.

Innanzitutto si sottolinea come tali sequenze siano formate da due caratteri (backslash + lettera) ma interpretate come un singolo carattere dal compilatore, le sequenze di escape sono le seguenti (tabella p.109):

|                           |   |                                              |                                                                                                                                                                                                                                                                                            |
|---------------------------|---|----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>printf("\n")</code> | → | va a capo                                    | (\n è il carattere ASCII 10, significa "line feed" e corrisponde al Comando "avanzamento riga" che, nelle telescriventi meccaniche, faceva arrotolare il foglio su cui veniva scritto il testo)                                                                                            |
| <code>printf("\r")</code> | → | torna all'inizio e sovrascrive l'ultima riga | (\r è il carattere ASCII 13, significa "line feed" significa return e corrisponde al comando che, nelle telescriventi meccaniche, faceva lentamente tornare il carrello all'inizio. Se veniva mandato questo comando senza '\n' l'ultima riga precedentemente scritta veniva sovrascritta) |
| <code>printf("\t")</code> | → | inserisce uno spazio di tabulazione ("tab")  |                                                                                                                                                                                                                                                                                            |
| <code>printf("\"")</code> | → | inserisce carattere virgolette "             |                                                                                                                                                                                                                                                                                            |
| <code>printf("\\")</code> | → | inserisce carattere backslash \              |                                                                                                                                                                                                                                                                                            |



## SVUOTAMENTO BUFFER DI INPUT

In alcuni casi, come abbiamo visto nell'esercizio sull'acquisizione dei dati di input, c'è un problema nell'acquisizione di un tipo di dato char. Il problema è il seguente:

Quando si eseguono due operazioni di input consecutive e la seconda operazione riguarda l'acquisizione di un tipo di dato char, l'esecutore non "attende" l'inserimento del char da tastiera ma prosegue con l'esecuzione. Ad esempio:

```
5  int main()
6  {
7      int x;
8      char c;
9      printf("Inserisci l'intero x: ");
10     scanf("%d",&x);
11     printf("Inserisci il carattere c: ");
12     scanf("%c",&c);
13     printf(" i due valori inseriti sono \nx=%d \nc=%c",x,c);
14     return 0;
15 }
16
```

Risultato:

L'esecutore non "attende"  
l'inserimento del carattere da tastiera

```
"D:\OneDrive - Istituto Olivelli Putelli\Scuola\AS 2022 2023\Materiale informatica\3_Linguaggio C\
Inserisci l'intero x: 10
Inserisci il carattere c: i due valori inseriti sono
x=10
c=
Process returned 0 (0x0) execution time : 2.374 s
Press any key to continue.
```

Cerchiamo di capire perché questo avviene e come risolvere il problema.

Quando si utilizza la funzione `scanf()` l'esecutore rimane in attesa dei dati inseriti con la tastiera. Ciò che viene digitato sulla tastiera non viene immediatamente assegnato alla variabile ma viene memorizzato in una memoria chiamata **buffer di input** fino a quando viene premuto il tasto "invio". Anche il tasto "invio" viene inserito nel buffer di input come carattere ascii 10 (il famoso '\n' visto prima).

Poiché la funzione `scanf()`, quando acquisisce input numerici, preleva dal buffer di input solamente i caratteri numerici (e non "preleva" spazio, tab, a capo ecc.), quando viene acquisito un input numerico, nel buffer rimane il carattere "\n" (invio). Pertanto, se dopo l'input numerico si cerca di acquisire un dato di tipo char, la funzione `scanf` assegna alla variabile di tipo char il carattere "\n", impedendo all'utente di inserire il valore desiderato.

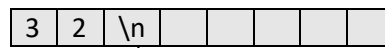
Esempio:

```
int x;
```

```
char c;
```

```
printf ("inserisci x (intero) ");
```

ipotizziamo che con la tastiera si inserisca il valore 32+invio

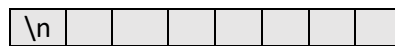


valori nel buffer input

```
scanf ("%i",&x);
```

invio

Dopo lo scanf() ecco cosa rimane nel buffer:

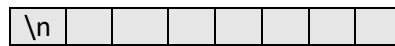


valori nel buffer input

32 è stato assegnato ad x, ma nel buffer rimane \n

```
printf ("inserisci il carattere c ");
```

```
scanf ("%c",&c);
```



valori nel buffer input

alla variabile c viene assegnato il carattere \n

```
printf ("x = %i \n c= %c ", x,c);
```

Per risolvere il problema, prima di acquisire il dato di tipo char con scanf, è necessario "svuotare" il buffer di input.

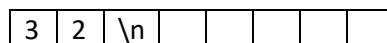
**Quindi la soluzione è svuotare il buffer di input. Per farlo si utilizza il seguente ciclo che "consuma" tutto il buffer fino a quando legge il \n. Quando viene letto \n l'esecutore esce dal ciclo.**

```
while(getchar() != '\n');
```

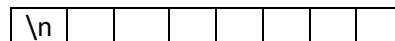
```
int x;
```

```
char c;
```

```
printf ("inserisci x (intero) ");
```

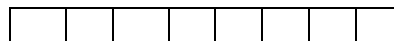


```
scanf ("%i",&x);
```



```
printf ("inserisci il carattere c ");
```

```
while(getchar() != '\n');
```



```
scanf ("%c",&c);
```



nuovo input da tastiera da parte dell'utente 'A' + invio

Alla variabile C viene assegnato il valore A

```
printf ("x = %i \n c= %c ", x,c);
```



Esempio con problema risolto:

```
4  int main()  
5  {  
6      int x;  
7      char c;  
8      printf("Inserisci un intero: ");  
9      scanf("%i",&x);  
10     printf("Inserisci un carattere: ");  
11     while(getchar() != '\n');  
12     scanf("%c",&c);  
13     printf("valori inseriti \nx:%i \nc:%c",x,c);  
14 }
```

Svuotamento buffer di input

## SCOPE: L'AMBITO DI VISIBILITA' DELLE VARIABILI

Le variabili (e le costanti) possono essere *globali* o *locali*.

Una variabile **globale** è una variabile che “esiste” (cioè che può essere utilizzata per variarne il valore o della quale si può utilizzare il valore) DA QUANDO VIENE DICHIARATA fino alla fine del codice contenuto in un file.

La dichiarazione di variabili globali è l'unica tipologia di istruzioni che può essere scritta all'esterno del blocco principale di un programma (sopra il main)

Una variabile **locale** invece “esiste” solamente all'interno **del blocco** in cui è stata dichiarata.

Il fatto che una variabile sia globale o locale dipende quindi da dove viene dichiarata.

Una variabile locale può assumere lo stesso nome (identificatore) di una variabile globale, in tal caso **nel blocco interno in cui è dichiarata la variabile locale, essa “copre” la variabile globale.**

L'ambito di visibilità di una variabile (ossia il blocco in cui è visibile) viene indicato con il termine di **scope**.

Esempio:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int x=200;
5  int main()
6  {
7      int x=10;
8      if(true)
9      {
10         double x=8.2;
11         printf("Dentro al blocco x vale %f\n",x);
12     }
13     printf("Fuori dal blocco x vale %i",x);
14     return 0;
15 }
16
```

E' possibile dichiarare variabili con lo stesso nome solamente se hanno scope diversi. Se si dichiarano due variabili con lo stesso nome all'interno dei uno stesso blocco il compilatore segnala un errore.

Esercizi interessanti:

Cosa mostra l'output nei seguenti casi?

```
int main()
{
    int x=20;
    if (true)
    {
        x=200;
    }
    printf("%i",x);

    return 0;
}
```

200

```
6  int main()
7  {
8      int x=20;
9      if (true)
10     {
11         int x=200;
12     }
13     printf("%i",x);
14
15     return 0;
16 }
```

20