PILLOLE DI PROGRAMMAZIONE IN LINGUAGGIO C

Con questa dispensa affrontiamo le basi della programmazione in linguaggio C e per farlo utilizzeremo un ambiente di sviluppo integrato (I.D.E. Integrated Development Environment) chiamato Codeblocks.

Questo tutorial non può ovviamente sostituire un libro sulla programmazione in linguaggio C, ma vuol fornire le nozioni essenziali e gli strumenti per poter realizzare rapidamente semplici programmi.

PARADIGMI DI PROGRAMMAZIONE

Un **paradigma di programmazione** è un modello che definisce la struttura di un programma. Ogni paradigma prevede modalità differenti per organizzare il codice e per rappresentare i dati.

Esistono diversi paradigmi di programmazione; tra i più noti troviamo quello **imperativo** e quello **dichiarativo**.

Nel primo caso, il paradigma imperativo (probabilmente il più diffuso) consiste nel fornire al computer una sequenza di istruzioni da eseguire per raggiungere un determinato risultato. Questo implica la necessità di tenere traccia dello **stato** del programma e delle operazioni da svolgere passo dopo passo.

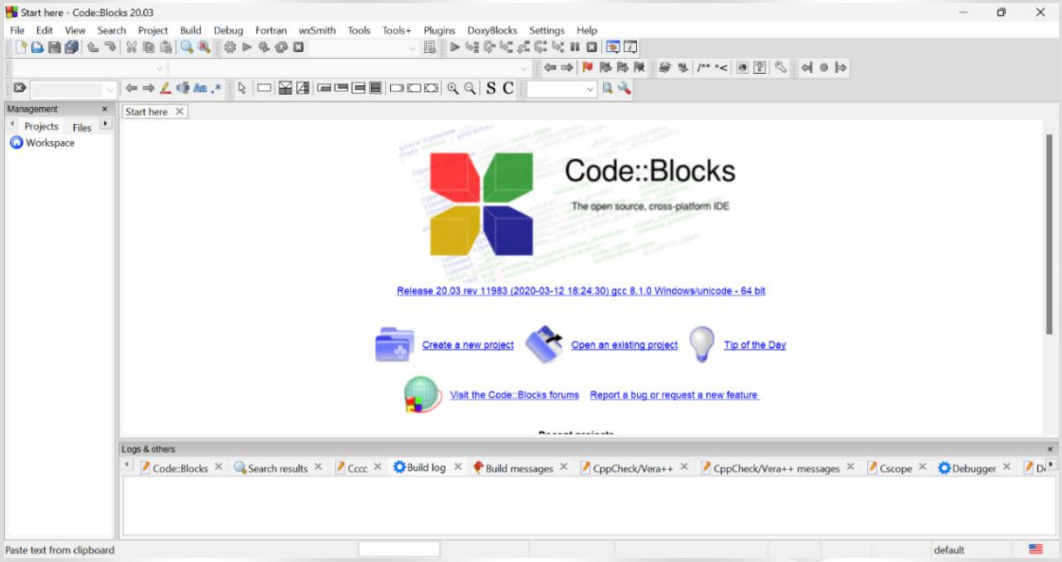
Nel secondo caso, il paradigma dichiarativo si basa sull'indicare alla macchina **che cosa** ottenere, senza specificare **come** procedere per ottenerlo. È quindi il sistema stesso a determinare la sequenza delle operazioni necessarie.

Nel nostro caso andremo ad utilizzare un linguaggio di programmazione basato sul paradigma di programmazione IMPERATIVO. Pertanto sarà compito del programmatore stabilire la tipologia di dati da utilizzare e organizzare la sequenza delle operazioni che la macchina dovrà eseguire per raggiungere il risultato richiesto.

Il programma è scaricabile al seguente link: https://www.codeblocks.org/downloads/binaries/

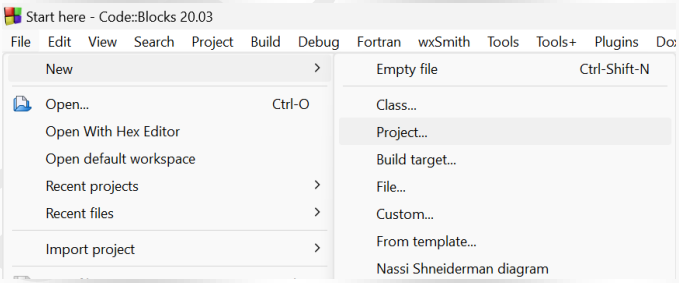
Scaricare la versione completa di compilatore codeblocks-25.03mingw-setup.exe

All'avvio del programma si presenta la seguente finestra.

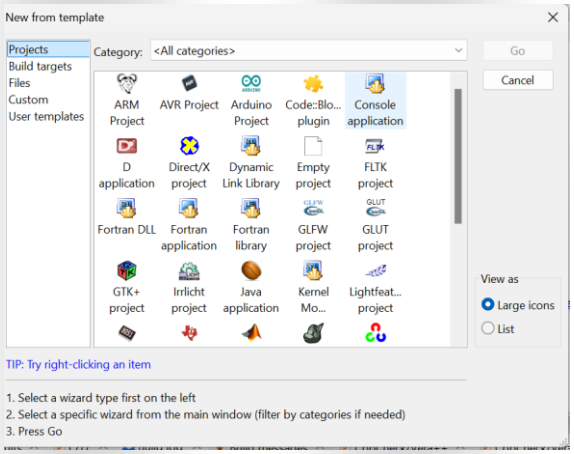


1) Prima di iniziare un nuovo progetto, occorre creare una cartella dove salvare tutti i file.

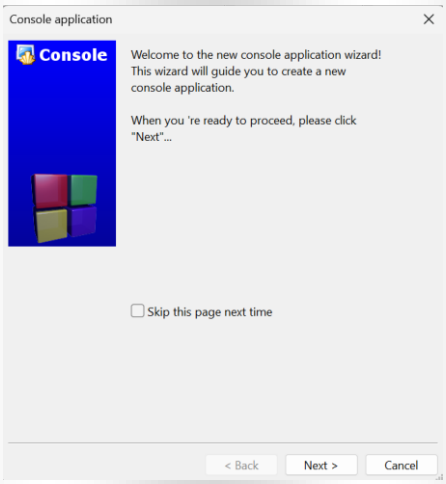
2) Successivamente dal menu File, scegliere New - Project.

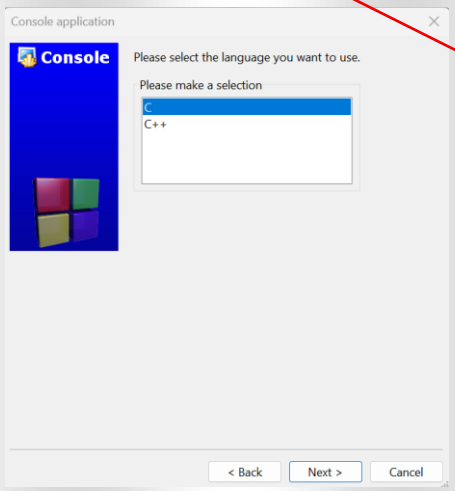


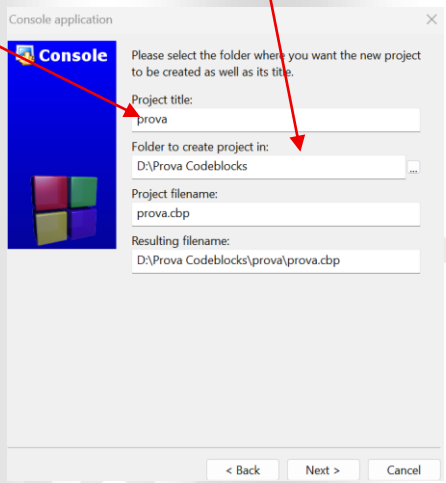
3) Si aprirà una finestra all'interno della quale ci sono i vari tipo di progetto che possiamo realizzare, noi sceglieremo **Console Application** e poi faremo clic sul pulsante **GO**.



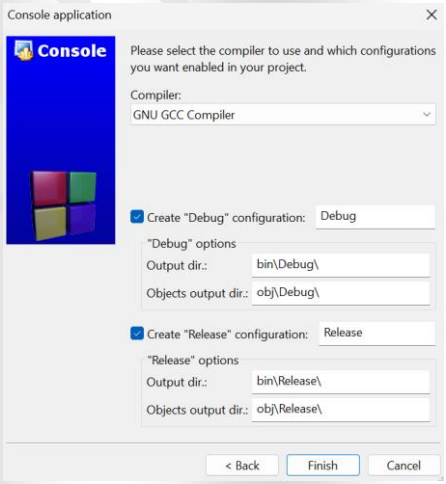
4) Partira il Wizard (l'aiuto nella creazione del progetto). Faremo click su NEXT scegliendo poi il **linguaggio C.** Successivamente daremo un nome al progetto e selezioneremo la cartella che avevamo creato inizialmente.





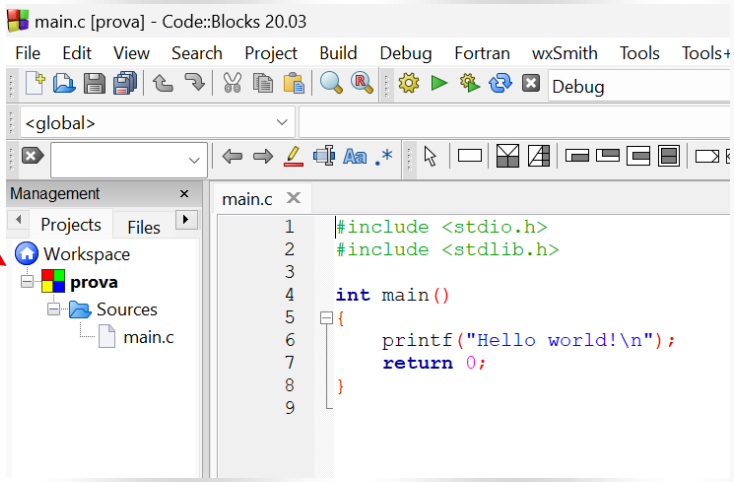


Proseguire con NEXT fino ad arrivare alla scelta del compilatore. Lasciamo le impostazioni predefinite e facciamo FINISH.



> ***Approfondimento:***
> *Il compilatore è un programma che traduce il codice sorgente scritto in un linguaggio di programmazione di alto livello (come C, C++, Java, ecc.) in un linguaggio macchina o in un codice intermedio che può essere eseguito direttamente dal computer o da un interprete.*
> *In altre parole, il compilatore converte il codice leggibile dall'uomo in istruzioni comprensibili dal processore.*
> *La scelta consigliata per Codeblocks, è GNU GCC Compiler", che è uno degli strumenti Open Source, più diffusi e affidabili per la compilazione di programmi in C e C++.*

Al termine nella parte sinistra della schermataavremo il progetto con i relativi files.
Inizialmente troveremo solo il file main.c dove scrivere il programma.



#include <stdio.h> include la libreria standard di input/output (serve per funzioni come printf e scanf).

#include <stdlib.h> → include la libreria standard (qui non è ancora usata, ma viene inserita per convenzione).

int main() → è la funzione principale del programma C, da cui parte l'esecuzione.

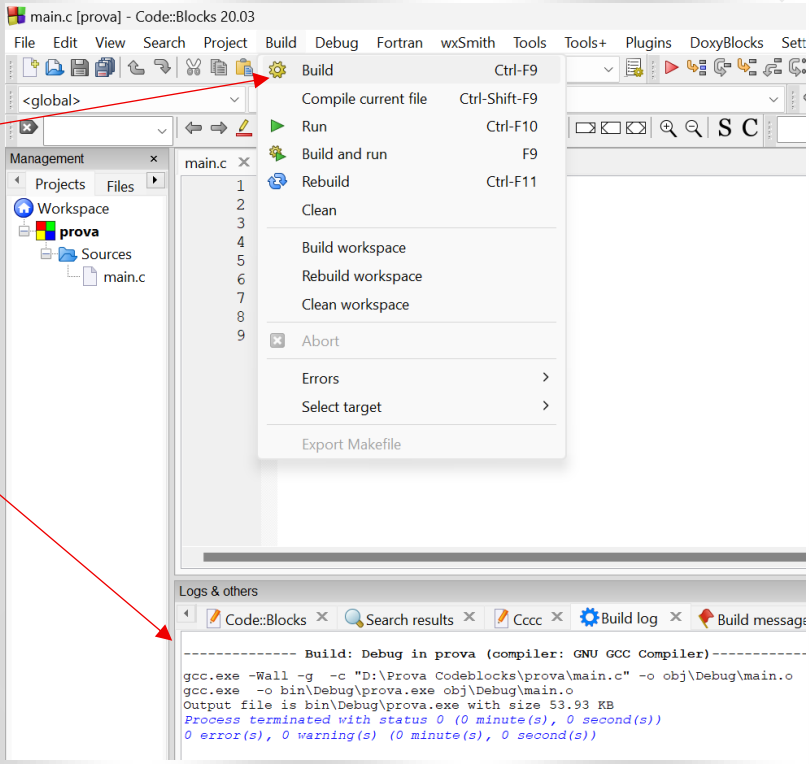
printf("Hello world!\n"); → stampa sullo schermo la stringa "Hello world!" seguita da un carattere ASCII "a capo" (\n).

return 0; → indica che il programma è terminato con successo (0 = nessun errore).

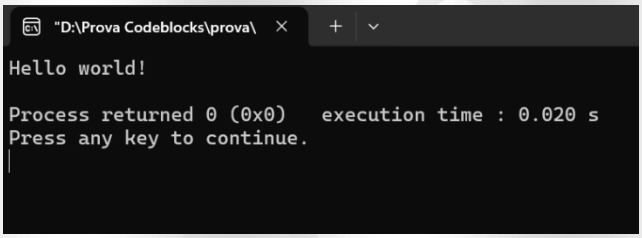
Approfondimento:
*Il codice ASCII è l'acronimo di **American Standard Code for Information Interchange**, cioè **Codice Standard Americano per lo Scambio di Informazioni**.*
È un sistema di codifica che associa a ogni carattere (lettera, cifra, simbolo o comando di controllo) un numero intero compreso tra 0 e 127.
Questi numeri vengono poi rappresentati in binario (cioè con 0 e 1) nei computer.
Ad esempio nel codice ASCII la lettera A corrisponde al valore 65, il numero 1 è visto come un carattere e corrisponde al numero decimale 49 e così via.
Nel codice ASCII ci sono anche comandi come ad esempio nuova linea che corrisponde al numero 10.

A questo punto possiamo già compilare il programma mediante il comando BUILD.

Se ci sono errori verranno segnalati nella finestra di Log in basso.



Dopo aver compilato possiamo cliccare sul pulsante RUN, o fare direttamente le due operazioni insieme con BUILD e RUN.
Il programma verrà eseguito e nel nostro caso vedremo solamente la scritta Hello World

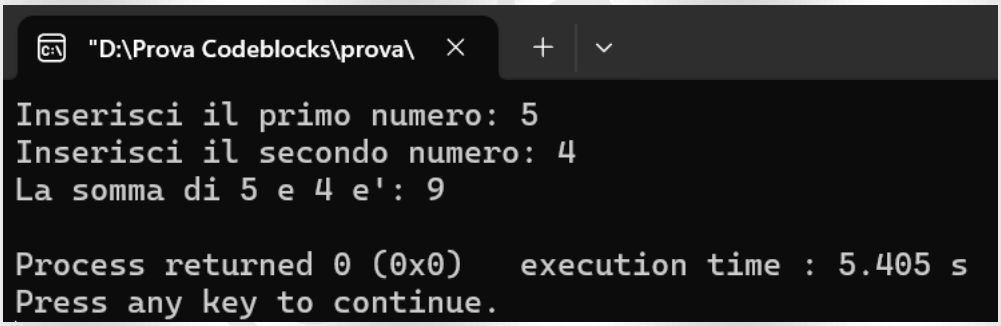


Esempio n.1

Prima di affrontare la programmazione in linguaggio C, proviamo un altro esempio.

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

int main()
{
    // Dichiarazione delle variabili
    int numero1, numero2, somma;

    // Acquisizione dei due numeri da tastiera
    printf("Inserisci il primo numero: ");
    scanf("%d", &numero1);

    printf("Inserisci il secondo numero: ");
    scanf("%d", &numero2);

    // Calcolo della somma
    somma = numero1 + numero2;

    // Visualizzazione del risultato
    printf("La somma di %d e %d e': %d\n", numero1, numero2, somma);

    return 0;
}
```

Anche in questo caso con il pulsante BUILD e RUN, possiamo compilare e mandare in esecuzione il programma ed anche in questo caso l'esecuzione del programma avverrà sulla generica schermata nera dove possiamo solamente inserire dei valori e vedere dei risultati.



Prima di affrontare la programmazione per un'interfaccia grafica realizzeremo programmi in questa modalità.

Ma per farlo dovremo innanzitutto conoscere il linguaggio C.

Di seguito affronteremo i seguenti argomenti:

- Variabili.
- Array monodimensionali e multidimensionali.
- Operatori.
- Funzioni Scanf e Printf.
- Strutture di controllo.
- Funzioni.
- Puntatori.
- Files.

VARIABILI

Nel linguaggio C e C++, le variabili devono sempre essere dichiarate prima del loro utilizzo.
Dichiarare una variabile significa indicare al compilatore il nome della variabile (identificatore) e il suo tipo.
Il compilatore riserverà uno spazio in memoria di dimensioni adeguate a contenere quel tipo di dato.
Nel linguaggio C i tipi di variabili fondamentali sono i seguenti:

TIPO	SPAZIO OCCUPATO	RANGE DI VALORI	
		min	max
• **char**	1 byte	-128	+127
• **unsigned char**	1 byte	0	+255
• **short Int** • **short**	2 byte	-32.768	+32.767
• **int** **sistemi a 16 bit**	2 byte	-32.768	+32.767
• **int** **sistemi a 32 bit**	4 byte	–2 147 483 648	+2 147 483 647
• **unigned int** **sistemi a 16 bit**	2 byte	0	+65.535
• **unigned int** **sistemi a 32 bit**	4 byte	–2 147 483 648	+2 147 483 647
• **long**	4 byte	-2.147.483.648	+2.147.483.647
• **unsigned long**	4 byte	0	+4.294.967.295
• **long long** **sistemi a 64 bit**	8 byte	–9 223 372 036 854 775 808	+9 223 372 036 854 775 807
• **unsigned long** **sistemi a 64 bit**	8 byte	0	18 446 744 073 709 551 615
• **float**	4 byte *7 cifre significative* *8 bit esponente* *23 bit mantissa*	$\pm 1.175\,494 \times 10^{-38}$	$\pm 3.402\,823 \times 10^{38}$
• **double**	8 byte *15 cifre significative* *11 bit esponente* *52 bit mantissa*	$\pm 2.225\,074 \times 10^{-308}$	$\pm 1.797\,693 \times 10^{308}$
• **long double**	10 byte *19 cifre significative* *15 bit esponente* *64 bit mantissa* In base all'architettura del processore esiste anche da 16 byte		

I numeri senza virgola, char, int, short e long, vengono memorizzati utilizzando il sistema binario, per comprendere in quale maniera, possiamo considerare il numero più piccolo, cioè il tipo char, con due esempi:

se il numero è positivo: char prova=98; in binario=0110 0010
se il numero è negativo: char prova=-98; in binario=1001 1110

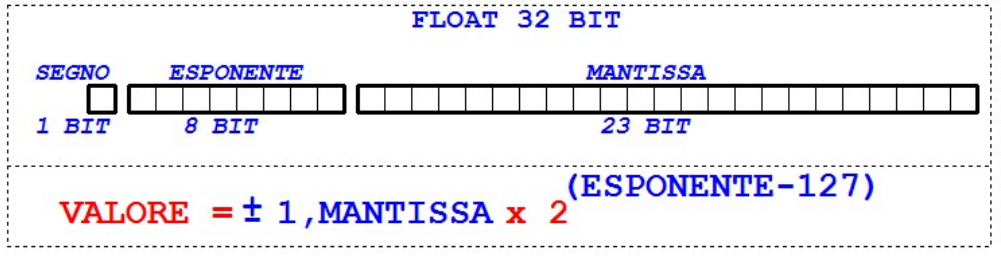
Nel primo caso il numero è positivo ed il codice binario è la semplice trasformazione del valore in decimale:

98=0110 0010 $=0\times2^7+1\times2^6+1\times2^5+0\times2^4+0\times2^3+0\times2^2+1\times2^1+0\times2^0=$
$= 0 + 64 + 32 + 0 + 0 + 0 + 2 + 0 =$
= 64+ 32+2=98

Nel secondo caso il numero negativo viene rappresentato facendo il complemento a due del valore 98 convertito in binario, cioè si ricava il complemento del numero binario invertendo ogni singolo bit e successivamente si somma il valore 1:

complemento di 0110 0010 = 1001 1101 (si invertono i singoli bit)
1001 1101 + 1 = 1001 1110

I numeri con la virgola vengono memorizzati utilizzando il sistema a virgola mobile che prevede una struttura di bit composta da segno-esponente-mantissa. Anche in questo caso consideriamo il più piccolo dei numeri con la virgola e cioè il tipo float:



Lo standard **IEEE754** definisce quanto segue:

SEGNO: *0=positivo 1=negativo*

ESPONENTE: *L'esponente deve rappresentare valori positivi e negativi, per fare questo nello spazio dedicato viene messo il valore dell'esponente reale sommato al valore 127.*
In questo modo volendo memorizzare un esponente pari a +2 ,nel campo troveremo 129 che in binario corrisponde a 1000 0001.
Volendo invece memorizzare un esponente pari a -2, nel campo troveremo 125 che in binario corrisponde a 0111 1101.

MANTISSA: *La mantissa è normalizzata, la normalizzaione si ottiene moltiplicando per 2 l'effettivo valore (in binario la moltiplicazione per 2 avviene shiftando a sinistra di una posizione) fino a quando il bit più a sinistra della mantissa diventa 1, eliminando in questo modo tutti i bit a zero non significativi a sinistra. Ciò significa che il bit più a sinistra vale sempre 1, per questo motivo è inutile memorizzare questa informazione, i 23 bit pertanto serviranno solo a rappresentare la parte frazionaria del numero.*
Ad esempio una mantissa pari a ***100 1011 0010 0000 0011 0001*** *trasformata come parte frazionaria in decimale corrisponde a:*

MANTISSA (PARTE FRAZIONARIA)

1	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	1
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-16}	2^{-17}	2^{-18}	2^{-19}	2^{-20}	2^{-21}	2^{-22}	2^{-23}

$$\frac{1}{2^1} + \frac{1}{2^4} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^{10}} + \frac{1}{2^{18}} + \frac{1}{2^{19}} + \frac{1}{2^{23}} =$$

$$= 0,586919903755187988281 25 + 1 =$$

$$= 1,586919903755187988281 25$$

Se ad esempio nei 4 byte abbiamo la seguente sequenza ***0100 0100 0100 1011 0010 0000 0011 0001*** *,*

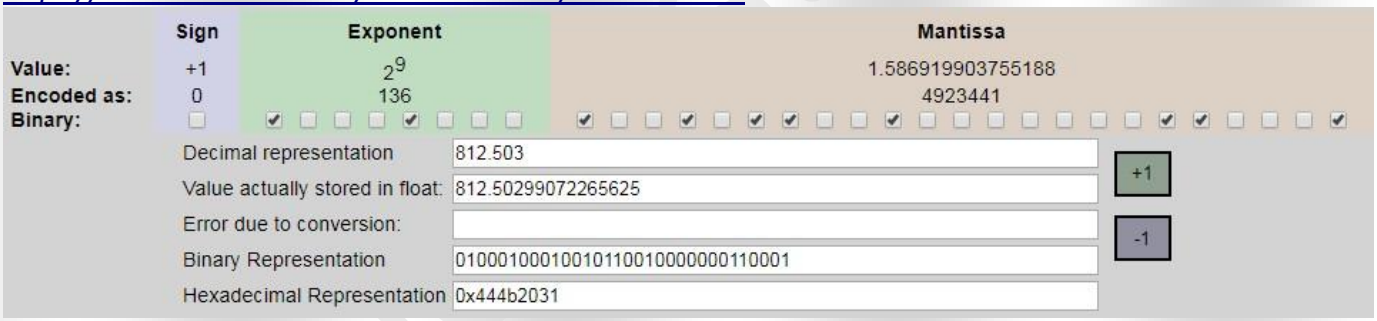
significa che il valore sarà ottenuto come segue: $VALORE = \pm 1,M \times 2^{(E-127)}$

S =0*=+*
E=100 0100 0*=136*
M=100 1011 0010 0000 0011 0001 *=0,586919903755187988281 25*

Sostituendo i valori otteniamo: $VALORE = +1,586919903755187988281 25 \times 2^{(136-127)}$

VALORE = 812,50299072265625

Al seguente link, un convertitore online dove è possibile verificare e provare la conversione in virgola mobile.
https://www.h-schmidt.net/FloatConverter/IEEE754.html



DICHIARAZIONE DI VARIABILI

La dichiarazione di variabili serve a riservare uno spazio nella memoria RAM adatto al tipo di dato che la variabile dovrà contenere.
Si possono dichiarare variabili **globali**, cioè accessibili da ogni parte del programma, oppure variabili **locali**, dichiarate all'interno di una funzione e utilizzabili solo al suo interno.
La dichiarazione di una variabile globale avviene generalmente all'inizio del programma, **prima del main.**

esempio: *int contatore=5;*
float valore;

Nel programma potrò cambiare il valore delle due variabili assegnando loro un valore o il risultato di un'operazione.

Con lo stesso meccanismo si possono dichiarare variabili **che non potranno mai cambiare il proprio valore**, cioè le **costanti**.

esempio: *const int giorni=7;*

Nel programma la variabile **giorni** non cambierà mai il proprio valore.

Durante l'esecuzione di un programma, il valore di una variabile può essere **modificato** assegnandole direttamente un nuovo valore oppure il risultato di un'operazione, utilizzando l'operatore =

```
esempio:    int contatore;      //dichiarazione della variabile senza valore
            int risultato;      //dichiarazione della variabile senza valore
            int valore1=10;  //dichiarazione della variabile con valore iniziale
            int valore2=5 ;  //dichiarazione della variabile con valore iniziale

            int main(){
                    contatore=5    //assegnazione diretta del valore 5 alla variabile contatore
                    contatore=contatore-1;  //assegno un nuovo valore in questo caso 4
                    risultato=valore1*valore2;  //assegno alla variabile il risultato di un'operazione
             }
```

COMMENTI

Nel precedente esempio possiamo anche vedere che sono stato inseriti dei **commenti**, per rendere leggibile il programma, i commenti vengono sempre preceduti dal doppio slash //, in questo caso tutto ciò che segue il doppio slash non viene considerato dal compilatore.

Si possono anche commentare più righe con il singolo slash insieme all'asterisco nel seguente modo:

```
esempio:    /*
            tutto questo è un commento
            anche se è scritto su più righe
            */
```

I **commenti** servono a fornire informazioni aggiuntive o a spiegare il funzionamento del programma. Il compilatore li ignora completamente e non li traduce in istruzioni eseguibili.

ARRAY

Un ARRAY è un insieme di variabili dello stesso tipo memorizzate in memoria in maniera contigua.
L'ARRAY viene dichiarato con un unico nome e l'accesso alle singole variabili avviene scrivendo il nome e l'indice della variable.

```
esempio:    int  valori[5];
```

in questo caso l'ARRAY valori contiene 5 elementi di tipo intero, ogni elemento può essere scritto e letto inserendo vicino al nome l'indice che va da 0 a 4, perciò le 5 variabili dell'array saranno le seguenti:

valori[0] valori[1] valori[2] valori[3] valori[4]

L'indice dell'array potrebbe essere anch'esso una variabile come nel seguente esempio:

```
esempio:    int  valori[5];
            int  indice=0;
            valore[indice]=2;       //scrivo nella variabile valore[0] il numero 2
            indice++;               //incremento di uno la variabile indice
            valore[indice]=2;       //scrivo nella variabile valore[1] il numero 2
```

E' possibile ovviamente dichiarare ARRAY anche di tipo FLOAT, LONG, CHAR ecc…

Nel caso di ARRAY di caratteri si può dichiarare il nome ed il contenuto, ed automaticamente la dimensione sarà adeguata per il contenuto come ad esempio: *char nome[] = "Mario";*

avremo che: ***nome[0]='M' nome[1]='a' nome[2]='r' nome[3]='i' nome[4]='o'*** **nome[5]=/0 (carattere nullo)**

In questo caso ogni array conterrà dopo i singoli caratteri sempre il carattere ASCII NULL indicato con /0 che come valore ha 0.

ARRAY MULTIDIMENSIONALI

Un array multidimensionale è, come nel caso di un array normale, un insieme di variabili dello stesso tipo memorizzate in memoria in maniera contigua.
A differenza di un array monodimensionale, le variabili sono organizzate **in righe e colonne**, come in una tabella. L'array viene dichiarato con un unico nome, e l'accesso alle singole variabili avviene tramite **il nome seguito da più indici** (uno per ciascuna dimensione).

esempio: `int valori[2][3];`

in questo caso l'ARRAY valori contiene 6 elementi di tipo intero, ogni elemento può essere scritto e letto inserendo vicino al nome i due indici:

	colonna 0	colonna 1	colonna 2
riga 0	*valori[0][0]*	*valori[0][1]*	*valori[0][2]*
riga 1	*valori[1][0]*	*valori[1][1]*	*valori[2][2]*

Volendo scrivere o leggere in una casella dovremo inserire il nome dell'ARRAY seguito dall'indice riga e poi colonna

esempio. `valori[0][1]=5; //scrivo il valore 5 nella casella centrale della prima riga`

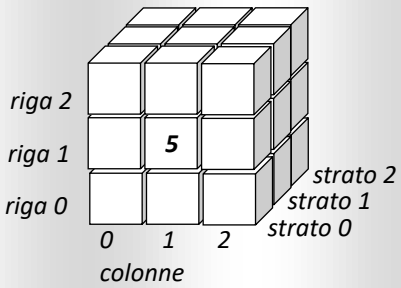
	colonna 0	colonna 1	colonna 2
riga 0		*5*	
riga 1			

Possiamo avere più di due dimensioni ad esempio nel seguente caso abbiamo un ARRAY a 3 dimensioni, questo può essere paragonato ad una tabella con più strati. La dichiarazione avviene sempre allo stesso modo, indicando il nome e gli elementi di ogni dimensione:

esempio: `int valori[3][3][3];`

Possiamo immaginare questo ARRAY come un cubo, come fosse una tabella 3 righe e 3 colonne presente su 3 strati

esempio: `valori[2][1][0]=5;`



Con lo stesso sistema si possono dichiarare ARRAY di n. dimensioni.

Anche in questo caso il tipo della variabile può essere di qualsiasi tipo non necessariamente intera come negli esempi ed anche in questo caso gli indici possono essere a loro volta delle variabili.

VETTORI DI STRINGHE

Un particolare caso d vettore bidimensionale è il vettore di stringhe. La sintassi è la stessa di un vettore di caratteri a due dimensioni:

char nome_vettore[num_stringhe][lunghezza_stringhe]

La lunghezza della stringa deve sempre tener conto della presenza del carattere terminatore.
Il vettore può essere inizializzato nel seguente modo:

```
char elenco_nomi[3][20] = {
    "Mario Rossi",
    "Luigi Bianchi",
    "Giuseppe Verdi"
};
```

STRUTTURE

I **vettori (array)** in C contengono elementi **tutti dello stesso tipo** (tutti int, tutti float, tutti char, ecc.). Quando si vuole creare un dato composto che includa **tipi diversi** (es. numeri, caratteri, stringhe), il linguaggio C mette a disposizione le **strutture**.
Una **struttura** (struct) è un tipo di dato aggregato che permette di raggruppare variabili di tipo diverso sotto un unico nome.
La sintassi della dichiarazione è la seguente:

```
struct NOME {
  tipo1  identificatore1;
  tipo2  identificatore2;
  tipo2  identificatore3;
};
```

Membri

Ad esempio:

```
struct MOTORE {
  char      sigla[3];
  float     potenza;
  unsigned  numero_giri;
  float     aliimentazione;
};
```

Una volta che la struttura è stata creata occorre dichiarare una variabile associata a quel tipo di struttura:

- Inizialmente viene creata la struttura.
- Viene dichiarata una variabile con quel tipo di struttura.
- Vengono assegnati i valori ai membri della struttura.
- Vengono visualizzati i valori dei membri.

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

struct MOTORE {
   char         sigla[3];
   float        potenza;
   unsigned     numero_giri;
   float        alimentazione;
};

struct MOTORE nastro;

int main() {

   nastro.sigla[0]='M';
   nastro.sigla[1]='1';
   nastro.sigla[2]='\0';//terminatore stringa
   nastro.potenza = 1000;
   nastro.numero_giri = 2800;
   nastro.alimentazione = 400;

   printf("Sigla = %s\n", nastro.sigla);
   printf("Potenza = %.2f\n", nastro.potenza);
   printf("Numero giri = %u\n", nastro.numero_giri);
   printf("Alimentazione = %.2f\n", nastro.alimentazione);

   return 0;
}
```

Il risultato sarà il seguente:

```
Sigla = M1
Potenza = 1000.00
Numero giri = 2800
Alimentazione = 400.00
```

SCANF E PRINTF
Lettura da Standard Input ed Output (libreria stdio.h)

Per realizzare i primi semplici programmi, come mostrato nei due esempi precedenti, è necessario immettere dati da tastiera e visualizzarli sullo schermo.
A questo scopo si utilizzano le funzioni scanf e printf, che appartengono alla libreria stdio.h e servono rispettivamente a leggere e scrivere dati dallo standard input e sullo standard output.

SCANF (scan formatted, cioè leggi con formato=
Quando viene chiamata questa funzione, il programma si ferma in attesa che l'utente inserisca un dato da tastiera nella console in cui il programma è in esecuzione. Il dato inserito viene poi memorizzato nella variabile indicata nel comando.

La sintassi è la seguente: ***scanf("formato", &variabile1, &variabile2, …);***

I formati possibili sono i seguenti:

%d intero (int)	*es.scanf("%d",&variabile);*	***%c carattere singolo (chat)***	*es.scanf("%c",&variabile);*
%f num.reale (float)	*es.scanf("%f",&variabile);*	***%s stringa di testo***	*es.scanf("%s",variabile);*
%lf num.reale (double)	*es.scanf("%lf",&variabile);*		

Il simbolo **&** presente prima del nome della variabile indica l'indirizzo di memoria della variabile e non va messo nel caso di una stringa di testo, in quanto in quel caso il nome della variabile rappresenta già l'indirizzo del primo carattere.
In maniera sintetica possiamo dire che la funzione scanf serve a leggere un dato dalla tastiera e a salvarlo in una variabile.

esempio
```
int numero;
scanf("%d", &numero);
```

PRINTF
Quando viene chiamata questa funzione, il programma **stampa sulla console** un valore o una stringa nel formato specificato. I formati sono gli stessi visti per la funzione scanf.

La sintassi è la seguente: **printf*("testo e formato", variabile1, variabile2, …);***

In questo caso non occorre il simbolo & che indica l'indirizzo in memoria della variabile, ma va scritto solo il nome.

esempio
```
int numero = 5;
printf("Il numero vale: %d", numero);
```

Come si può vedere nell'esempio nella parte iniziale dentro la parentesi, quella indicata nella sintassi con "testo e formato", è possibile inserire un testo e successivamente ad esso il formato
Pertanto da questo comando otterremo quanto segue: `Il numero vale: 5`

E' possibile scrivere più variabili, ad esempio:

```
int num1=5,num2=3;
int main(){
printf("numero1=%d, numero2=%d", num1, num2);
}
```

In questo caso otterremo: `numero1=5, numero2=3`

Se si hanno numeri con la virgola si possono anche indicare il numero di decimali, scrivendolo prima della lettera che indica il tipo di variabile, come ad esempio nel seguente caso:

```
float pi = 3.14159;
int main(){
  printf("Pi con 2 decimali: %.2f", pi);
}
```

In questo caso otterremo: `Pi con 2 decimali: 3.14`

OPERATORI

Gli operatori in C sono simboli che permettono di eseguire operazioni su variabili e valori. Si possono suddividere in diverse categorie in base alla funzione.

TIPOLOGIA	**SIMBOLO**	**OPERATORE**	**ESEMPIO** *(a,b e c sono variabili)*
Operatori aritmetici	+	*Addizione*	*c=a+b* ***//c** sarà uguale alla somma tra **a** e **b***
	-	*Sottrazione*	*c=a-b* ***//c** sarà uguale alla differenza tra **a** e **b***
	*	*Moltiplicazione*	*c=a*b* ***//c** sarà uguale alla moltiplicazione tra **a** e **b***
	/	*Divisione*	*c=a/b* ***//c** sarà uguale alla somma tra **a** e **b***
	%	*Modulo (resto della divisione)*	*c=a%b* ***//c** sarà uguale alla resto della divisione tra **a** e **b***
	++	*Incremento*	*c++* ***//c** sarà incrementato di 1*
	--	*Decremento*	*c--* ***//c** sarà decrementato di 1*
Operatori di assegnazione	=	*Assegnazione*	c=5 ***//c** sarà uguale a 5*
	+=	*Somma ed assegna*	c+=5 ***//c** sarà incrementato di 5*
	-=	*Sottrai ed assegna*	c-=5 ***//c** sarà decrementato di 5*
	*=	*Moltiplica ed assegna*	c*=5 ***//il valore di c** sarà moltiplicato per 5*
	/=	*Dividi ed assegna*	c/=5 ***//il valore di c** sarà diviso per 5*
	%=	*Trova il resto ed assegna*	c%=5 ***// c** conterrà il resto della divisione c/5*
Operatori bitwise (a livello di bit)	&	AND bit a bit	c=a&b //Esegue l'operazione di and tra i bit di **a** e **b**
	|	OR bit a bit	c=a|b //Esegue l'operazione di or tra i bit di **a** e **b**
	^	XOR bit a bit	c=a^b //Esegue l'operazione di xor tra i bit di **a** e **b**
	~	NOT bit a bit	c=~a //Inverte tutti i bit di **a** e mette il risultato in **c**
	>>	Shift a sinistra di n. bit	c=a>>2 //i bit di **a** vengono shiftati a destra di 2 posizioni
	<<	Shift a destra di n. bit	c=a<<2 //i bit di **a** vengono shiftati a sinistra di 2 posizioni
Operatori di confronto	==	Uguale	a==b ***//controlla se a è uguale a b***
	!=	Diverso	a!=b ***//controlla se a è diverso da b***
	>	Maggiore	a>b ***//controlla se a è maggiore di b***
	<	Minore	a<b ***//controlla se a è minore di b***
	>=	Maggiore o uguale	a>=b ***//controlla se a è maggiore o uguale a b***
	<=	Minore o uguale	a<=b ***//controlla se a è minore o uguale a b***
Operatori logici	*&&*	*Operatore logico and*	*(a>0)&&(b>0)* ***//controlla se a=0 e b=0***
	||	*Operatore logico or*	*(a>0)||(b>0)* ***//controlla se a=0 o b=0***
	!	*Operatore logico not*	*!(a>0)* ***//controlla se a è minore di zero***
Operatori speciali	*Sizeof*	Dimensione di una variabile	*c=sizeof(a)* ***//c** conterrà il numero di byte utilizzati da **a***
	&	Indirizzo di una variabile	***c=&a //c** conterrà l'indirizzo in memoria della variabile **a***

Prima di vedere l'utilizzo di questi operatori, è indispensabile cominciare a parlare delle **strutture di controllo**.

STRUTTURE DI CONTROLLO

In C, le **strutture di controllo** servono a **gestire il flusso di esecuzione** di un programma, permettendo decisioni, ripetizioni o salti condizionali. Si possono dividere in tre categorie principali: **selezione, iterazione e salto**.

Strutture di selezione (condizionali)	**Strutture di iterazione (cicli)**	**Istruzioni di salto**
○ if…else ○ else if ○ switch…case	○ for ○ while ○ do…while	○ break ○ continue ○ return ○ goto

STRUTTURE DI SELEZIONE

IF…ELSE

Questa è una **struttura di controllo condizionale** che permette di **eseguire blocchi di codice diversi a seconda che una condizione sia vera o falsa**.

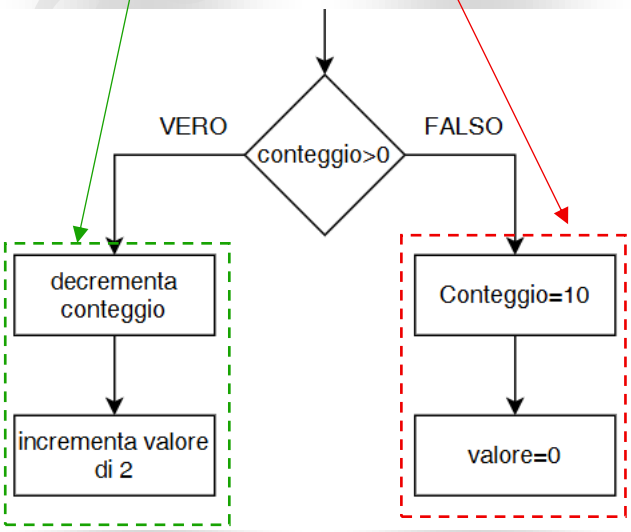
Sintassi:

```
if (condizione) {
    // codice eseguito se la condizione è vera
} else{
    // codice eseguito se la condizione è falsa
}
```

Se la condizione tra le parentesi è vera (true, o 1) si esegue il codice contenuto tra le prime parentesi graffe, altrimenti si esegue il codice tra le parentesi dell'else.

esempio:
In questo caso la condizione è che il valore della variabile conteggio sia maggiore di 0 in questo caso la stessa variabile viene decrementata e la variabile valore viene incrementata di due unità.
Quando la condizione non è più vera, cioè quando conteggio diventa uguale a zero, allora si esegue ciò che è nelle parentesi dell'else.

```
int conteggio=10,valore=0;

if (conteggio>0) {
    conteggio--;
    valore+=2;
} else{
    conteggio=10;
    valore=0;
}
```

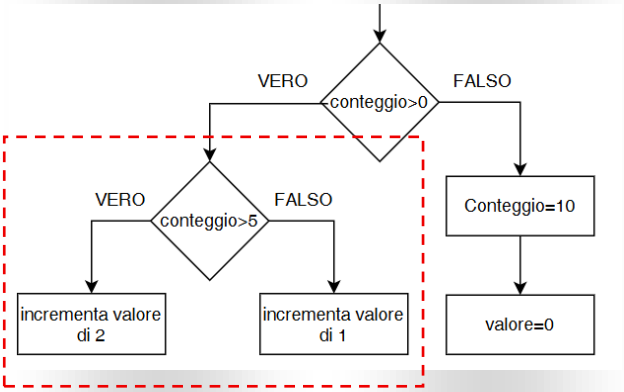


Il risultato della condizione può essere solamente TRUE o FALSE, o anche 1 o 0 e la condizione può interessare una variabile di qualsiasi tipo.

Il flusso del programma può avere due direzioni, ma volendo si possono annidare più strutture if dando più possibili percorsi al programma.

esempio:
In questo caso se la condizione è vera si va a controllare una seconda condizione e cioè se conteggio>5, in questo caso si incrementa valore di 2 ed in caso contrario si incrementa di 1.
Da notare che nel secondo IF annidato dentro al primo, non sono state messe le parentesi graffe, questo perché si possono evitare quando nell'IF o nell'ELSE c'è una sola istruzione.

```
int conteggio=10,valore=0;

if (conteggio>0) {
    conteggio--;
    if(conteggio>5) valore+=2;
    else            valore+=1;
} else{
    conteggio=10;
    valore=0;
}
```



ELSEIF

Questa struttura serve per **verificare più condizioni in sequenza**, una dopo l'altra. È una **forma estesa dell'istruzione if...else**, utile quando ci sono **più possibilità di scelta**.

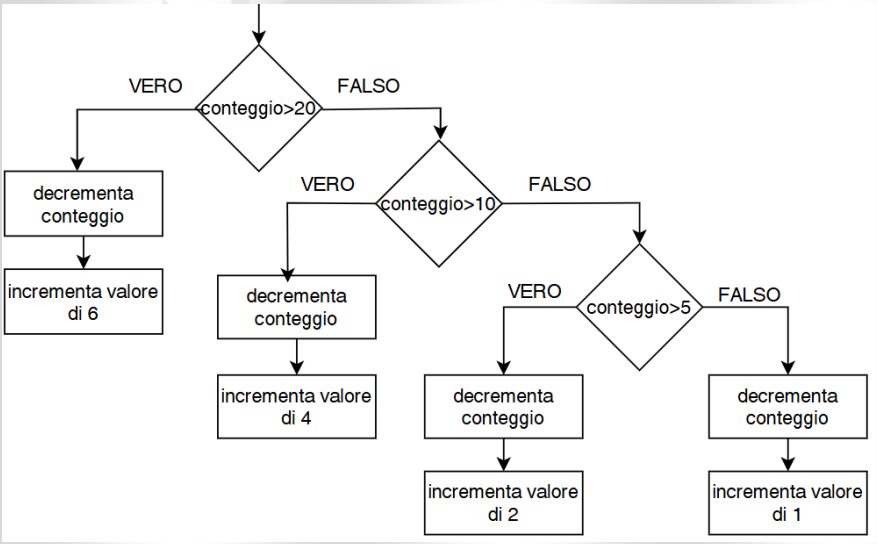
Sintassi:

```
if (condizione1) {
    // codice eseguito se condizione1 è vera
}
else if (condizione2) {
    // codice eseguito se condizione2 è vera
}
else if (condizione3) {
    // codice eseguito se condizione3 è vera
}
else {
    // codice eseguito se nessuna condizione è vera
}
```

Le condizioni tra parentesi tonde sono più di una, il programma verifica ogni condizione ed esegue il contenuto tra le parentesi graffe del relativo else.

esempio:
in base al valore della variabile conteggio viene incrementata diversamente la variabile valore.

```
if (conteggio>20) {
    conteggio--;
    valore=valore+6;
}
else if (conteggio>10) {
    conteggio--;
    valore=valore+4;
}
else if (conteggio>5) {
    conteggio--;
    valore=valore+2;
}
else {
    conteggio--;
    valore=valore+1;
}
```



La condizione può essere anche combinata quando si uniscono più condizioni utilizzando gli operatori logici visti prima.

esempio: *se conteggio è compreso tra 2 ed 8 (estremi esclusi) si può scrivere in questo modo.*

```
if((conteggio>2)&&(conteggio<8)){
    //istruzioni da eseguire se la condizione è vera
}
```

se invece voglio verificare di stare fuori da questo intervallo (estremi compresi) scriverò in questo modo.

```
if((conteggio>=8)||(conteggio<=2)){
    //istruzioni da eseguire se la condizione è vera
}
```

Le condizioni combinate possono interessare qualsiasi variabile anche differenti tra di loro all'interno della stessa condizione, inoltre il ciclo IF potrebbe anche non avere l'ELSE come visto negli ultimi due esempi.

SWITCH…CASE

È una struttura di controllo condizionale che permette di eseguire blocchi di codice diversi in base al valore assunto da una variabile o da un'espressione.
A differenza dell'istruzione **IF**, nel costrutto **switch** il controllo viene effettuato una sola volta sul valore della variabile, confrontandolo con i valori specificati nei vari **case**.
Quando si trova una corrispondenza, viene eseguito il blocco di codice relativo fino a un'eventuale istruzione **break**, che serve a terminare lo switch.

Sintassi:

```
switch (espressione) {
    case valore1:
        // istruzioni se espressione == valore1
        break;

    case valore2:
        // istruzioni se espressione == valore2
        break;

    case valore3:
        // istruzioni se espressione == valore3
        break;

    default:
        // istruzioni se nessun case corrisponde
}
```

- L'espressione dentro switch() viene valutata.
- Il programma confronta il suo valore con i vari case.
- Quando trova un case uguale, esegue le istruzioni da lì in poi.
- Il comando break serve a uscire dallo switch e non eseguire i casi successivi.
- Il blocco default (facoltativo) si esegue se nessun valore corrisponde.

esempio:

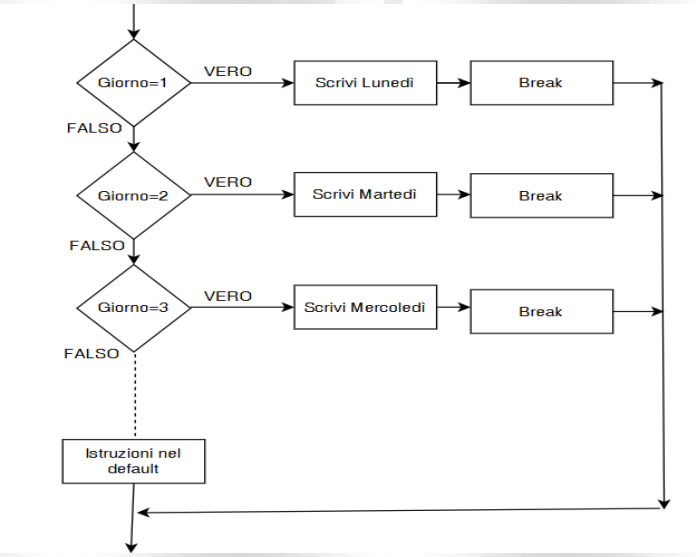
```
int giorno = 3;

switch (giorno) {
    case 1:
        printf("Lunedi\n");
        break;
    case 2:
        printf("Martedi\n");
        break;
    case 3:
        printf("Mercoledi\n");
        break;
    case 4:
        printf("Giovedi\n");
        break;
    case 5:
        printf("Venerdi\n");
        break;
    case 6:
        printf("Sabato\n");
        break;
    case 7:
        printf("Domenica\n");
        break;
    default:
        printf("Numero non valido (1-7)\n");
        break;
}
```

nell'esempio viene testata la variabile intera "giorno" in base al suo valore viene visualizzato a video il relativo giorno della settimana.

Nel caso dell'esempio giorno=3, pertanto verranno eseguite le istruzioni dopo il case 2 fino al break. Il break interromperà i successivi case.

Se il valore non è tra 1 e 7, cioè i numeri indicati nei vari case, allora viene eseguito ciò che segue il default.

Il diagramma di flusso dell'esempio potrebbe essere rappresentato come nel seguente modo:



In realtà non vengono effettuati più controlli della variabile, ma il programma salta direttamente nel case che contiene il valore corretto.

STRUTTURE DI ITERAZIONE

WHILE…DO

La struttura WHILE…DO è un ciclo che verifica una condizione booleana (TRUE o FALSE) prima di eseguire il blocco di istruzioni.
Se la condizione è TRUE, il blocco viene eseguito e poi la condizione viene nuovamente controllata. La sequenza di istruzioni viene ripetuta finché la condizione rimane vera.

Sintassi:

```
while(condizione da testare){
    istruzione 1;
    istruzione 2;
    istruzione 3;
    .
    .
    istruzione n;
}
```

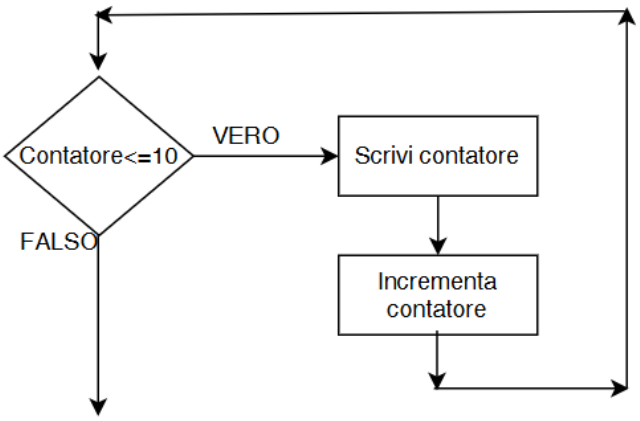
- Viene valutata la condizione del While.
- Se vera (TRUE) allora si eseguono tutte le istruzioni.
- Si torna a valutare la condizione del While.
- Si eseguono le istruzioni fino a quando la condizione è vera.

esempio:

nell'esempio viene controllato il valore della variabile contatore se il valore è inferiore o uguale a 10, viene visualizzato sullo schermo e successivamente viene incrementato di un'unità.
Il risultato sarà

```
int contatore=0;

while(contatore<=10){
    printf("Valore=%d\n",contatore);
    contatore++;
}
```

Il ciclo WHILE esegue il controllo in testa al blocco di Istruzioni, pertanto se la condizione inizialmente non è rispettata Il blocco di istruzioni non verrà mai eseguito.



Il risultato dell'esempio sarà il seguente:

```
Valore=0
Valore=1
Valore=2
Valore=3
Valore=4
Valore=5
Valore=6
Valore=7
Valore=8
Valore=9
Valore=10
```

DO…WHILE

La struttura DO…WHILE a differenza della precedente è un ciclo che verifica la condizione booleana (TRUE o FALSE) dopo aver eseguito il blocco di istruzioni.
Se la condizione è TRUE, il blocco viene eseguito nuovamente. La sequenza di istruzioni viene ripetuta finché la condizione rimane vera.

Sintassi:

```
do{
    istruzione 1;
    istruzione 2;
    istruzione 3;
    .
    .
    istruzione n;
}while(condizione da testare);
```

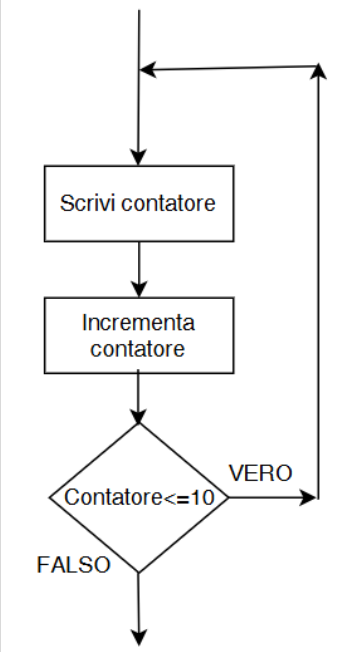
- Vengono eseguite tutte le istruzioni.
- Viene valutata la condizione del While.
- Se vera (TRUE) allora si torna ad eseguire tutte le istruzioni.
- Si eseguono le istruzioni fino a quando la condizione è vera.

esempio:

nell'esempio viene visualizzato il valore della variabile contatore e successivamente viene incrementato. Al termine si controlla se il valore è inferiore o uguale a 10, in tal caso vengono ripetute le istruzioni.

```
int contatore=0;

do{
    printf("Valore=%d\n",contatore);
    contatore++;
}while(contatore<=10);
```

Il ciclo WHILE esegue il controllo in coda, dopo al blocco di Istruzioni, pertanto il blocco di istruzioni non verrà eseguito almeno una volta.



Il risultato dell'esempio sarà il seguente:

```
Valore=0
Valore=1
Valore=2
Valore=3
Valore=4
Valore=5
Valore=6
Valore=7
Valore=8
Valore=9
Valore=10
```

FOR

Il ciclo FOR è una struttura di controllo di tipo iterativo che permette di ripetere un blocco di istruzioni per un numero definito di volte, oppure finché una certa condizione rimane vera.
All'interno della struttura vengono identificate 3 parti: l'inizializzazione, la condizione e l'aggiornamento.

- L'inizializzazione viene eseguita una sola volta all'inizio del ciclo e si usa per dichiarare ed assegnare un valore alla variabile di controllo.
- La condizione viene valutata prima di ogni iterazione (in testa) se è *vera* (non zero), il ciclo continua; se è *falsa* (0), il ciclo termina.
- L'aggiornamento viene eseguito alla fine di ogni iterazione del ciclo, serve per modificare la variabile di controllo.

Sintassi:

```
for (inizializzazione; condizione; aggiornamento) {
   // blocco di istruzioni da ripetere
}
```

esempio:

nell'esempioviene visualizzato il valore della variabile i che varia da 0 a 10

INIZIALIZZAZIONE CONDIZIONE AGGIORNAMENTO

```
int i;
for (i = 0; i <= 10; i++){
   printf("Valore=%d\n",i);
}
```

Il risultato dell'esempio sarà il seguente:

```
Valore=0
Valore=1
Valore=2
Valore=3
Valore=4
Valore=5
Valore=6
Valore=7
Valore=8
Valore=9
Valore=10
```

E' possibile omettere una delle 3 parti del FOR, ad esempio è possibile avere le seguenti condizioni:

- Ciclo FOR senza INIZIALIZZAZIONE.
  La variabile viene inizializzata esternamente al ciclo FOR

```
int i = 0;
for (; i < 5; i++) {
    printf("%d\n", i);
}
```

- Ciclo FOR senza CONDIZIONE.
  La condizione viene testata internamente come una qualsiasi istruzione.

```
for (int i = 0; ; i++) {
   // ciclo infinito fino a break
   if (i == 5) break;
}
```

- Ciclo FOR senza AGGIORNAMENTO.
  L'aggiornamento viene eseguito internamente come una qualsiasi istruzione.

```
for (int i = 0; i < 5; ) {
   printf("%d\n", i);
   i++; // aggiornamento manuale
}
```

ISTRUZIONI DI SALTO

BREAK

E' un'istruzione di controllo che serve per interrompere immediatamente l'esecuzione del ciclo o dello *switch* in cui si trova.

Questa istruzione termina subito:

- un ciclo for
- un ciclo while
- un ciclo do...while
- una struttura switch

Dopo il break, l'esecuzione prosegue dalla prima istruzione dopo il blocco interrotto.

Di seguito qualche esempio di utilizzo del BREAK.

```c
for (int i = 0; i < 10; i++) {
  if (i == 5) {
      break;   //interrompe il ciclo quando i arriva a 5
  }
  printf("%d\n", i);
}
```

```
0
1
2
3
4
```

```c
int valore=1;

switch (valore) {
   case 1:
       printf("Uno");
       break;     // evita di eseguire anche i case successivi
   case 2:
       printf("Due");
       break;
   case 3:
       printf("Due");
       break;
}
```

```c
int i;

while (1) {  // ciclo potenzialmente infinito
    printf("i = %d\n", i);
    i++;
    if (i == 5) { // quando i arriva a 5...
        break;    // ...interrompo il ciclo
    }
}
```

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

CONTINUE

E' un'istruzione che serve per saltare il resto del corpo del ciclo e passare subito all'iterazione successiva.

- In un **for**, salta direttamente all'**aggiornamento** e poi al controllo della **condizione.**
- In un **while** o **do…while**, salta direttamente al **controllo della condizione.**

Esempio nel ciclo FOR

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue;    // salta il printf per i = 5
    }
    printf("%d\n", i);
}
```

```
0
1
2
3
4
6
7
8
9
```

Esempio nel ciclo WHILE

```
int i = 0;
while (i < 10) {
    i++;
    if (i == 5) {
        continue; //salta alla condizione i<10
    }
    printf("%d\n", i);
}
```

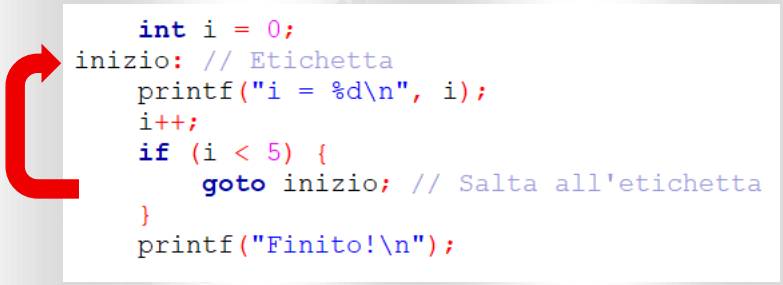
```
1
2
3
4
6
7
8
9
10
```

GOTO

E' un'istruzione di salto incondizionato: quando viene eseguita, il flusso del programma prosegue dalla posizione dell'etichetta indicata.

L'etichetta è un nome seguito da due punti (:), simile a una variabile, ma non può essere una parola chiave.

Esempio:



```
    int i = 0;
inizio: // Etichetta
    printf("i = %d\n", i);
    i++;
    if (i < 5) {
        goto inizio; // Salta all'etichetta
    }
    printf("Finito!\n");
```

```
i = 0
i = 1
i = 2
i = 3
i = 4
Finito!
```

RETURN

Questa istruzione viene usata per terminare una funzione e restituire un valore (se la funzione non è void).

Successivamente vedremo le funzioni dove RETURN verrà utilizzato più volte, di seguito un semplice esempio con una funzione che esegue la somma di due numeri e restituisce con l'istruzione RETURN il risultato.

```
int somma(int a, int b) {
  return(a + b);    // restituisce la somma
}
```

ESPRESSIONI CONDIZIONALI COMPOSTE

Nei precedenti esempi abbiamo visto come utilizzare le operazioni di confronto per verificare una condizione.

Prendiamo l'esempio visto precedentemente nel ciclo WHILE:

```
int contatore=0;

while(contatore<=10){
    printf("Valore=%d\n",contatore);
    contatore++;
}
```

In questo caso il blocco delle istruzioni contenute nel ciclo vengono eseguite se la condizione (contatore<=10) è TRUE (vera).
Quando il valore di contatore è inferiore o uguale a 10 pertanto viene scritto sullo schermo il valore della variabile.

In questo caso la condizione è una sola, vediamo ora un caso di condizione composta e lo facciamo con un ciclo IF all'interno del WHILE.

```
int contatore=0;

while(contatore<=10){
  if( (contatore>3) && (contatore<7) ){
      printf("%d\n", contatore);
  }
  contatore++;
}
```

Il risultato sarà:

```
4
5
6
```

LA condizione per cui viene eseguito il WHILE rimane la stessa, ma all'interno del ciclo c'è una struttura IF con una condizione composta da (contatore>3) e (contatore<7).
Visto che le due condizioni sono legate dall'operatore AND && significa che il corpo dell'IF verrà eseguito solo se entrambe le condizioni sono vere. Il risultato infatti dimostra che vengono stampati solo i numeri 4,5, e 6.

Se invece cambiamo l'operatore e mettiamo l'operatore OR || ed allo stesso tempo modifichiamo il maggiore e minore nelle condizioni avremo quanto segue:

```
int contatore=0;

while(contatore<=10){
  if( (contatore<3) || (contatore>7) ){
      printf("%d\n", contatore);
  }
  contatore++;
}
```

Il risultato sarà:

```
0
1
2
8
9
10
```

In questo caso viene stampato il valore solo se è inferiore a 3 e superiore a 7.

Le condizioni composte possono essere anche più di due e possono anche utilizzare operatori differenti, come nel seguente esempio:

```
int contatore=0;

while(contatore<=10){
  if( ((contatore>2) && (contatore<5)) || ((contatore>6) && (contatore<9)) ){
      printf("%d\n", contatore);
  }
  contatore++;
}
```

```
3
4
7
8
```

In questo caso viene effettuata la stampa della variabile se il valore è compreso tra 2 **e** 5 **oppure** se è compreso tra 6 **e** 9.

FUNZIONI

In ogni linguaggio di programmazione esiste la possibilità di organizzare il codice mediante **funzioni**. Le funzioni servono a:

- evitare ripetizioni di istruzioni,
- rendere il programma più leggibile,
- migliorare la modularità e la struttura del codice.

Ogni funzione possiede:

- **un NOME**,
- **un TIPO di ritorno** (cioè il tipo di dato che restituisce),
- **eventuali PARAMETRI** che le vengono passati.

Il **contenuto della funzione** (la sua definizione) va scritto **al di fuori del main()**.

- Se la funzione è **scritta prima del main**, non occorre altro, il compilatore la conosce già.
- Se invece la funzione è **definita dopo il main**, allora è necessario dichiararne prima il **PROTOTIPO**, cioè una sua anticipazione che indica:
  - il tipo di ritorno,
  - il nome della funzione,
  - i parametri che accetta.

Questo permette al compilatore di sapere come utilizzare la funzione anche prima di incontrarne la definizione.

Per meglio comprendere quanto detto, consideriamo una funzione che esegue la somma tra due valori che gli vengono passati e restituisce il risultato.

PROTOTIPO, viene descritto il nome i due parametri passati **a** e **b** ed il tipo di valore restituito INT**.**

CHIAMATA ALLA FUNZIONE, viene chiamata la funzione a cui vengono passati due numeri, 3 e 4, ed il risultato restituito andrà nella variabile intera **r**.

In questo caso la variabile **r** è stata dichiarata dentro al MAIN.

DEFINIZIONE, viene scritto il codice contenuto nella funzione, indicando i parametri ed il tipo come nel prototipo.
Con l'istruzione return, si restituisce il risultato.

```c
//prototipo della funzione
int somma(int a, int b);


//corpo del main
int main() {
    int r = somma(3, 4);
    printf("%d\n", r);
    return 0;
}


// definizione della funzione
int somma(int a, int b) {
    return (a + b);
}
```

Se la DEFINIZIONE fosse stata inserita prima del MAIN non occorreva il PROTOTIPO, come nel seguente caso.

I parametri passati possono essere anche inferiori a 2 o superiori, e di diversa tipologia, non necessariamente tutti INT come nell'esempio.

```c
// definizione della funzione
int somma(int a, int b) {
    return (a + b);
}

//corpo del main
int main() {
    int r = somma(3, 4);
    printf("%d\n", r);
    return 0;
}
```

Come le variabili, le funzioni possono essere di vario tipo, INT, FLOAT, LONG, ecc… Ma potrebbero anche non avere tipo e cioè non restituire alcun valore, in questo caso la funzione viene definita **VOID** e non ci sarà bisogno del **return**.

Ad esempio la stessa cosa vista sopra la si potrebbe realizzare con una funzione VOID nel seguente modo.

Il valore non viene restituito ma messo in una variabile globale.

```
int r;

// definizione della funzione
void somma(int a, int b) {
    r=(a + b);
}

//corpo del main
int main() {
    somma(3, 4);
    printf("%d\n", r);
    return 0;
}
```

La precedente soluzione non è sicuramente la più ottimale in quanto si potrebbe non utilizzare alcuna variabile globale.

In questo esempio vediamo la funzione inserita all'interno del printf, come fosse una variabile.
Una **funzione che restituisce un valore** può essere vista *come un'espressione* che produce un risultato.
Quindi, **puo essere usata come fosse una variabile o un valore**, perché quando viene chiamata "diventa" il valore che restituisce.

```
// definizione della funzione
int somma(int a, int b) {
   return(a + b);
}

//corpo del main
int main() {
    printf("%d\n", somma(3, 4));
    return 0;
}
```

Quando si scrive somma(3, 4), il *valore* della funzione è **7**. Per questo può essere inserita:

- in un'assegnazione
- in un confronto
- in un'espressione matematica
- come argomento di un'altra funzione

Ma attenzione perché una funzione:

- **non conserva un valore** permanente come una variabile,
- **non può essere modificata** come una variabile,
- **produce un valore solo quando viene eseguita,**

Quindi è più corretto dire:
"Una funzione con valore di ritorno è come una *formula* che produce un valore quando viene chiamata."

CASTING

Il **casting** è un'operazione per forzare la conversione di un dato da un tipo a un altro. Serve ad indicare al compilatore in che maniera interpretare un dato.

Supponiamo il seguente esempio:

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

int num1,num2;
float risultato;

int main() {

    num1=10;
    num2=3;
    risultato=num1/num2;
    printf("Risultato=%f\n", risultato);
    return 0;
}
```

*In questo caso nonostante la variabile **risultato** sia di tipo float, il valore scritto a schermo è il seguente:*

```
Risultato=3.000000
```

Questo perché le variabili utilizzate num1 e num2 sono di tipo intero, pertanto l'operazione di divisione tra le due non è in grado di memorizzare un numero con la virgola.

La soluzione in questo caso è quella di fare il casting dell'operazione indicando il tipo float.
In questo caso il risultato ottenuto è il seguente:

```c
int num1,num2;
float risultato;

int main() {

    num1=10;
    num2=3;
    risultato=(float)num1/num2;
    printf("Risultato=%f\n", risultato);
    return 0;
}
```

```
Risultato=3.333333
```

Ovviamente il casting is può effettuare con tutte le diverse tipologie di variabili e non solo dall'intero al float come nell'esempio precedente.

Si parla di **casting implicito**, quando la conversione avviene senza indicare nulla come quando una variabile intera viene passato ad una variabile double, in questo caso si passa da un tipo più piccolo (int) ad un tipo più grande (double), nell'esempio b avrà lo stesso valore di a, ma b è un numero reale.

```c
int a = 10;

double b = a;
```

Quando invece il valore di una variabile reale viene messo in una intera (da un tipo più grande ad uno più piccolo) si ha il **casting esplicito**. Nell'esempio la variabile y intera non potrà contenere il valore dopo la virgola ed il suo valore sara pari a 10.

```c
double x = 10.75;

int y = (int) x;
```

PUNTATORI

Ogni variabile può essere vista come una porzione di memoria RAM con un nome identificativo, contenente il valore che assegniamo alla variabile.

Possiamo pertanto immaginare la memoria RAM come una serie di cassetti, ognuno di questi cassetti ha un suo indirizzo.

Indirizzo	*Contenuto*
0000	
0001	
0002	
0003	
0004	

Se dichiariamo una variabile PROVA di tipo INT, nel caso in cui la memoria sia organizzata in byte, questa occuperà 2 byte.

Indirizzo	*Contenuto*
0000	
0001	
0002	valore di PROVA
0003	
0004	

Fatta questa premessa, definiamo ora un nuovo tipo di variabile definita PUNTATORE.

Un **puntatore** è una variabile che **contiene l'indirizzo di memoria** di un'altra variabile.
Come visto sopra ogni variabile ha:

1. un **valore**
2. un **indirizzo** (dove è memorizzata)

Un puntatore memorizza proprio questo **indirizzo** e nell'esempio visto sopra della variabile PROVA il puntatore a questa variabile conterrà il valore 0002.

Per dichiarare un PUNTATORE, si utilizza l'operatore *****: ***int *p;***

Per indicare l'indirizzo di una variabile si utilizza l'operatore **&**: &PROVA indica **l'indirizzo** di PROVA

Pertanto scrivendo:

```
int PROVA = 10;
int *p = &PROVA;
```

Avremo che **p** contiene l'indirizzo di PROVA e ***p** rappresenta il **valore di PROVA**

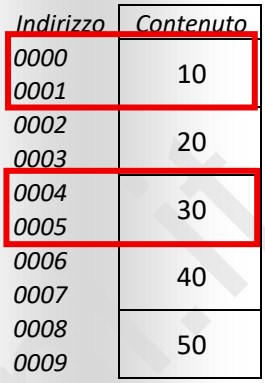
E' possibile anche modificare il valore della variabile utilizzando il puntatore, come nel seguente esempio:

```
int PROVA = 10;
int *p = &PROVA;

int main() {
  *p = 20;     // cambia il valore di PROVA
  printf("%d", PROVA);  // stampa 20
}
```

N.B.
E' anche possibile dichiarare un puntatore che non punta ad alcuna variabile, in questo modo il puntatore dichiarato non contiene l'indirizzo di nessuna variabile: int *p = NULL;

Nel caso di un array di valori, il puntatore all'array punta di fatto al primo elemento:

```
int vettore[5] = {10, 20, 30, 40, 50};
int *p = vettore;  //equivalente a int *p=&vettore[0];

int main() {
  printf("%d", *p);    // stampa 10
}
```



Le operazioni su un puntatore tengono conto della dimensione in byte della variabile.
Ad esempio nel caso dell'esempio abbiamo una variabile int (2byte) sommando al puntatore il valore 2 andremmo a puntare l'indirizzo 0004.

```
int vettore[5] = {10, 20, 30, 40, 50};
int *p = vettore;  //equivalente a int *p=&vettore[0];

int main() {
  printf("%d", *(p+2));    // stampa 30
}
```

In pratica il puntatore inizialmente puntava alla riga 0 e sommando 2 punterà alla riga 4 in quanto 2 variabili INT sono 4 byte.

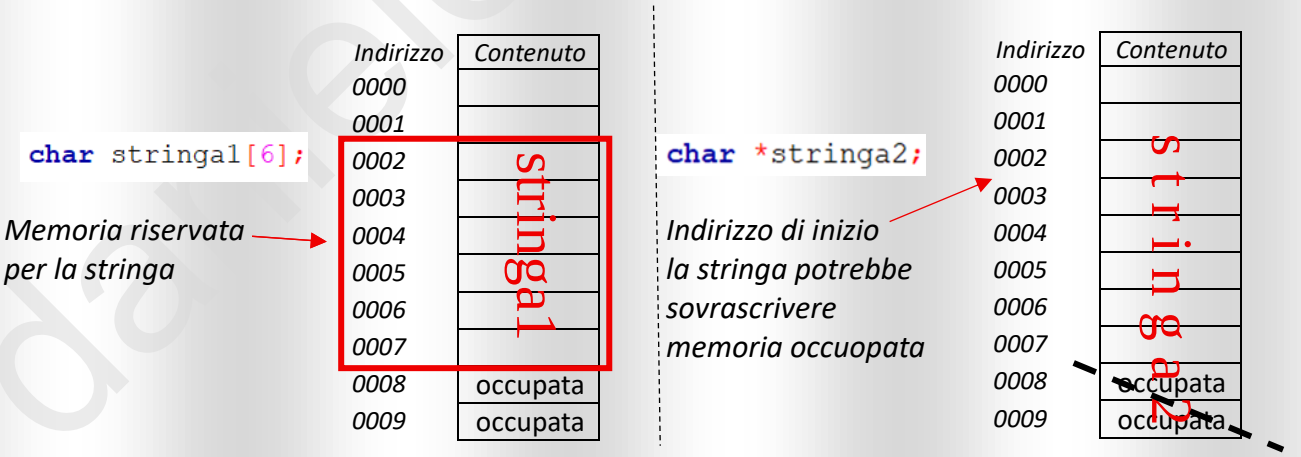
STRINGHE E PUNTATORI

Una stringa è un vettore di char terminato dal carattere '\0'. Valgono pertanto le stesse considerazioni viste in precedenza tranne che per qualche distinzione.

Una Stringa può essere dichiarata come vettore di caratteri: `char stringa1[6];`

ma può essere dichiarata anche come puntatore ad un vettore di tipo char: `char *stringa2;`

La differenza sta nel fatto che nel primo caso viene allocata una dimensione ben definita della memoria per contenere il numero di caratteri indicato, nel secondo caso invece viene allocata la locazione di memoria necessaria a contenere l'indirizzo del primo carattere, pertanto nel secondo caso scrivendo nella stringa si rischia di sovrascrivere un'area di memoria già occupata.



In entrambi i casi le stringhe si possono inizializzare al momento della dichiarazione:

```
char stringa1[]="prova";
char *stringa2="prova";
```

Anche in questo caso bisogna precisare che nel primo caso i caratteri della stringa1 "prova" vengono memorizzati nel vettore stringa1 , nel secondo caso invece alla variabile stringa2 viene assegnato solo l'indirizzo del primo elemento.

STRUTTURE E PUNTATORI

Un puntatore ad una struttura contiene l'indirizzo in memoria della struttura.
Per comprendere come vengono utilizzati i puntatori con le strutture andiamo a vedere direttamente un esempio.

*Dichiaro una struttura chiamata **MOTORE** contenente i 4 campi indicati.*

*Dichiaro una variabile **M1** del tipo struttura **MOTORE**, e la inizializzo con dei valori.*

*Dichiaro un puntatore **ptr** che contiene l'indirizzo della struttura **M1** di tipo **MOTORE**.*

*Accedo ai singoli campi della struttura tramite il puntatore **ptr** tramite l'operatore **->***

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


struct MOTORE {
   char sigla[3];
   float potenza;
   unsigned numero_giri;
   float alimentazione;
};

int main() {
   struct MOTORE M1 = {"M1", 1000.0, 2800 , 380.0};
   struct MOTORE *ptr = &M1;

   printf("Sigla: %s\n", ptr->sigla);
   printf("Potenza: %.2f\n", ptr->potenza);
   printf("Numero di giri: %d\n", ptr->numero_giri);
   printf("Alimentazione: %.2f\n", ptr->alimentazione);

   return 0;
}
```

VETTORE DI PUNTATORI

E' possibile anche avere dei vettori di puntatori, cioè un **array i cui elementi sono puntatori** (non valori).

Nell'esempio vengono dichiarate due variabili; **valore1** e **valore2** ed un puntatore chiamato **vettore** con due elementi che conterranno gli indirizzi delle due variabili.

```c
int valore1 = 10, valore2 = 20;
int *vettore[2];

vettore[0] = &valore1;
vettore[1] = &valore2;
```

ALLOCAZIONE DINAMICA DELLA MEMORIA

Lavorando con i puntatori, potrebbe essere utile organizzare in maniera logica lo spazio di memoria per le variabili utilizzate.
Quando si lavora con i puntatori, si gestiscono direttamente gli indirizzi di memoria, pertanto è importante sapere dove sono memorizzati i dati e come sono organizzati.

Quanto dichiaramo una variabile normalmente, ad esempio; si stabilisce già in fase di compilazione uno spazio in memoria.

```c
int valore=10;
int vettore[6];
```

C'è la possibilità invece di allocare dinamicamente la memoria da assegnare alle variabili in maniera dinamica, durante l'esecuzione (runtime) del programma.
Per farlo su utilizzano le seguenti funzioni presenti nella libreria **stdlib.h**.

- malloc() alloca memoria non inizializzata
- calloc() alloca memoria inizializzata a 0
- realloc() ridimensiona memoria già allocata
- free() libera la memoria

MALLOC e FREE

Vediamo un possibile utilizzo di MALLOC.
Supponiamo di dover calcolare la media di un numero di valori non conosciuto inserito da tastiera.

Inizialmente viene chiesto il numero di valori da inserire.

*Successivamente viene allocato lo spazio necessario per contenere n valori il puntatore **v** conterrà l'indirizzo del primo elemento.*
*In pratica è come se avessimo dichiarato un vettore con **n** elementi puntato da **v**.*

*Successivamente nel ciclo for, vengono inseriti i valori che di volta in volta vengono messi nell'indirizzo dell'elemento puntato da **v**.*

*Al termine nel secondo ciclo for, viene calcolata la somma dei vari elementi puntati da **v**.*

Al termine viene calcolata e visualizzata la media.
Successivamente con free viene liberata la memoria allocata.

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

int n; //dichiaro una variabile intera
int *v; //dichiaro un puntatore ad una variabile intera

int main() {
   printf("Quanti numeri? ");
   scanf("%d", &n);
   /* allocazione dinamica */
   v = malloc(n * sizeof(int));
   if (v == NULL) {
       printf("Errore di allocazione\n");
       return 1;
   }
   /* inserimento dati */
   for (int i = 0; i < n; i++) {
       printf("Inserisci numero %d: ", i + 1);
       scanf("%d", &v[i]);
   }
   /* calcolo media */
   int somma = 0;
   for (int i = 0; i < n; i++){
       somma += v[i];
   }
   printf("Media = %.2f\n", (float)somma / n);
   /* rilascio memoria */
   free(v);

   return 0;
}
```

Se avessimo inserito 3 numeri avremmo ottenuto la seguente situazione.

		Indirizzo	Contenuto
		0000	
		0001	
Puntatore **v=0002**	**v[0]**	0002	Numero1
		0003	
	v[1]	0004	Numero2
		0005	
	v[2]	0006	Numero3
		0007	
		0008	
		0009	

I 3 elementi puntati da v sono v[0], v[1] e v[2] e contengono l'indirizzo dei 3 elementi.
*Con **scanf** utilizzo l'operatore **&** in quanto con questa funzione occorre indicare l'indirizzo di memoria dell'elemento.*
Nel calcolo della media leggo direttamente il valore dei 3 elementi senza l'utilizzo dell'operatore &.

CALLOC

A differenza di MALLOC questa istruzione alloca la memoria indicata e la inizializza al valore di 0.

Viene allocata nua quantità di memoria per contenere 5 interi messi inizialmente al valore di 0.

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

int *v;
int n = 5;

int main(){
   // allocazione dinamica e inizializzazione a 0
   v = calloc(n, sizeof(int));
   for (int i = 0; i < n; i++)     printf("val= %d\n", i, v[i]);
   free(v);
   return 0;
}
```

REALLOC

Nel seguente esempio si va a reallocare lo spazio del vettore v con un numero fisso di 10 interi.

```c
realloc(void *v, 10 * sizeof(int));
```

LIBRERIA stdlib.h

La libreria **stdlib.h** del linguaggio **C** contiene funzioni "di utilità generale", usate molto spesso sia su PC sia su **microcontrollori (Arduino, ESP8266, ecc.)**.
Di seguito una **descrizione ordinata per categorie**, con le funzioni più importanti.

Conversione di stringhe in numeri.

- Conversione da stringa ad intero **atoi(const char *str);**

```
char s[] = "123";
int x = atoi(s);    // x = 123
```

- Conversione da stringa a long **atol(const char *str);**

```
char s[] = "123";
long x = atoi(s);    // x = 123
```

- Conversione da stringa a float **atof(const char *str);**

```
char s[] = "123";
float x = atof(s);    // x = 123.0000
```

Conversioni avanzate.

- Conversione da stringa ad intero controllata. **strtol(const char *str, char **endptr, int base);**
  In questo caso si passa alla funzione la stringa da convertire, endptr è invece un puntatore ad un puntatore, cioè endptr contiene l'indirizzo del puntatore che contiene l'indirizzo dell'ultimo carattere non convertito.
  In base va invece messa la base numerica del numero.

  Consideriamo ad esempio la seguente istruzione:

```
char *end;
long v = strtol("123abc", &end, 10);
```

  La stringa da convertire è "**123abc**" ed ovviamente la conversione si fermerà dopo i primi 3 caratteri quelli numerici.
  Il secondo parametro è l'indirizzo del puntatore **end** dichiarato inizialmente come puntatore.
  Il terzo parametro è la base 10.

  Il secondo parametro end conterrà l'indirizzo del primo carattere non convertito.

  Stringa = '1' '2' '3' 'a' 'b' 'c'

  end punta al 4° carattere e conterrà l'indirizzo di **a**

- Conversione da stringa a long int, controllata. **strtoul(const char *str, char **endptr, int base);**
  Come la precedente ma converte in **long int solo positivi.**

- Conversione da stringa a numeri in virgola mobile **strtod(const char *str, char **endptr);**

Gestione memoria dinamica, viste precedentemente

- malloc(size_t size);
- calloc(size_t num, size_t size);
- realloc(void *ptr, size_t new_size);
- free(void *ptr);

Numeri casuali

- Generatore di numeri casuali **rand(void);**
  Genera un numero tra 0 e la costante della libreria RAND_MAX
- Generatore di numeri casuali con sequenze differenti **srand(unsigned int seed);**
  Genera un numero casuale come nel precedente caso, ma gli viene passato il seme (seed) per generare un numero casuale partendo da un valore differente.

```
srand(time(NULL));
```

  Mettendo **time(NULL)** come seme generatore della sequenza, mette un valore che indica il tempo attuale.
  Per utilizzare la funzione **time**, occorre includere la libreria **time.h**

Valori assoluti e divisioni

- Restituisce il valore assoluto di un numero intero **abs(int x);**
- Restituisce il valore assoluto di un numero long **labs(long x);**
- Esegue la divisione tra interi restituendo quoziente e resto. **div(int num, int den);**
  La funzione restituisce questa struttura

```
typedef struct {
   int quot;    // quoziente
   int rem;     // resto
} div_t;
```

Esempio di utilizzo della funzione:

Viene dichiarata la struttura col nome risultato, gli vengono passati i numeri 17 e 5, il quoziente ed il resto saranno nei membri della struttura:

- risultato.quot
- risultato.rem

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

int main() {
   div_t risultato = div(17, 5);
   printf("Quoziente: %d\n", risultato.quot);
   printf("Resto: %d\n", risultato.rem);
   return 0;
}
```

- Esegue la divisione tra long restituendo quoziente e resto. **ldiv(long num, long den);**
  Come nel caso precedente, ma i membri della struttura restituita sono di tipo long.

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

int main() {
   ldiv_t risultato = ldiv(17, 5);
   printf("Quoziente: %d\n", risultato.quot);
   printf("Resto: %d\n", risultato.rem);
   return 0;
}
```

Controllo del programma

- Esce e termina il programma restituendo un valore. **exit(int status);**

A differenza di return che vale solo nelle funzioni, exit vale ovunque ed esce dal programma.

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

int main() {
   printf("Programma avviato\n");
   exit(0);
   printf("Questa riga non verrà mai eseguita\n");
}
```

- Esce e termina immediatamente ed in modo anomalo il programma segnalando un errore grave. **abort(void);**

Gestione delle stringhe

- **Strcpy.** Copia una stringa terminata da \0.

Nell'esempio il contenuto della stringa sorgente **src**, viene copiato nella stringa destinazione **dest**.

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

int main() {
   char src[] = "Ciao";
   char dest[10];
   strcpy(dest, src);
   printf("%s\n", dest);
   return 0;
}
```

- **Strcmp(.** Compara due stringhe terminate da \0.

  Nell'esempio il contenuto della stringa **cmd**, viene confrontato con la strunga "START".
  Il confronto restituisce 0 se le stringhe sono uguali.

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

int main() {
   char cmd[] = "START";
   if (strcmp(cmd, "START") == 0) {
       printf("Comando START riconosciuto\n");
   }
   return 0;
}
```

- **Strcat.** Concatena due stringhe. Con il termine concatenare, si intendere appendere una stringa di seguito ad un'altra.

  Nell'esempio, le stringhe "Ciao " e "mondo", vengono concatenate, aggiunte una di seguito all'altra, per ottenere la stringa "Ciao mondo".
  Il contenuto della stringa risultante va nella prima stringa usata.

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

int main() {
   char a[20] = "Ciao ";
   char b[] = "mondo";
   strcat(a, b);
   printf("%s\n", a);  // Ciao mondo
   return 0;
}
```

- **Strlen.** Restituisce la lunghezza in caratteri di una stringa.

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

int main() {
   char s[] = "Hello";
   printf("Lunghezza: %d\n", strlen(s));
   return 0;
}
```

WORKING IN PROGRESS !!!!

- GESTIONE DEI FILES