

GUIDA AL LINGUAGGIO DI PROGRAMMAZIONE “C”

1. COS'È IL "C"?

Il "C" è la lingua franca dell'universo dei computer, sviluppata negli anni '70; è usato per scrivere di tutto (dai sistemi operativi ai programmi). Su di esso sono basati molti altri linguaggi (C++, Java, C#, Perl e Python). Le sue caratteristiche sono:

- Linguaggio di "basso livello" → il C fornisce accesso a concetti a livello macchina, ovvero operazioni che corrispondono alle istruzioni proprie del computer;
 - Linguaggio "piccolo" e permissivo;
 - Efficiente, portabile, potente e flessibile con una grande "libreria standard" (punto di forza del C);
 - È però incline agli errori ed i suoi programmi potrebbero essere difficili da capire e da modificare;

Dopo aver scritto un programma in C, affinché il compilatore lo possa eseguire ci sono 3 passi:

- **Preprocessamento**: il programma viene dato ad un “preprocessore” che obbedisce ai comandi che iniziano con #, ovvero le cosiddette direttive;
 - **Compilazione**: dopo il programma passa ad un “compilatore”, che lo traduce in istruzioni macchina;
 - **Linking**: infine il “linker” combina il codice prodotto con del codice addizionale, per renderlo eseguibile;

⚠ Tutto questo a noi è invisibile se usiamo un "IDE" (ambiente di sviluppo integrato), che già fa tutto da solo.

Esempio di programma:

```
1. #include <stdio.h>                                // #include = richiamare librerie di funzioni
2. int main()                                         // int = tipo di dato che ritorna la main (0, ovvero intero)
3. {                                                 // main() = funzione principale ("il programma")
4.     printf("Hello World!");                         // {} = delimitano le istruzioni
5.     return 0;                                       // printf("testo") = stampa a video il messaggio contenuto
6. }                                                 // return 0 = ritorna il valore 0 dopo aver eseguito la main
```

→ La prima riga contiene l'header `<stdio.h>` che contiene le informazioni riguardanti la libreria standard di I/O (Input/Output) del C; la seconda riga contiene invece una funzione, ovvero una serie di istruzioni invocate con un nome → la funzione `main()` è particolare in quanto viene invocata automaticamente all'esecuzione del programma. In questo caso, la funzione `main()`, quando il programma viene eseguito, ritorna il valore "0" che è appunto un intero (come vediamo dal tipo `int` che precede la definizione della funzione). Il `return 0` causa la fine della funzione `main()` e indica appunto che questa restituisce il valore 0, ovvero restituisce un valore trascurabile per l'elaboratore [in alcuni casi, il `return 0` può anche essere omesso].

→ Inoltre, dato che la funzione main() non ha argomenti, ovvero non ha nulla dentro le parentesi tonde, si può anche scrivere come main(**void**), ovvero “mancante” di argomenti.

Se si vuole aggiungere un **COMMENTO** all'interno del codice, in C si hanno due opzioni:

- `/* Commento */` → “inscatolo” il commento
 - `// Commento` → commento valido dalle due sbarrette al termine della stessa riga

Inoltre possiamo anche costruire una vera e propria scatola intorno ai commenti (con lo scopo magari di introdurre il programma); esempio:

```
1. /*****
2. * Nome: potenza.c *
3. * Scopo: stampare la potenza di un numero *
4. *****/
```

2. LE VARIABILI

Le **VARIABILI** sono contenitori che memorizzano i valori/dati durante l'esecuzione del programma; in C le variabili devono essere dichiarate prima di essere inizializzate, ovvero devo prima dichiararne il TIPO:

```

1. #include <stdio.h>
2. int main()
3. {
4.     int x, k;           // Dichiaro la variabile(o le variabili) con il determinato "tipo"
5.     x = 5;             // Inizializzo la variabile assegnandone un valore
6.     k = 4;
7.     char a = 'r';      // Posso anche fare le due cose insieme (dichiaro + inizializzo)
8.     double y = 2.0;
9.     printf("%d", sizeof(y)); // La funzione sizeof() dice quanti byte occupa la nostra variabile
10.    return 0;
11. }                   // Ricorda che se non c'è "#", la riga va chiusa con ";"
```

PRINCIPALI TIPI di dichiarazione:

TIPO della variabile	A COSA mi riferisco	SPAZIO DI MEMORIA occupato
char	Singolo carattere ASCII	1 BYTE (8 bit)
int	Numero INTERO	2 BYTE (16 bit)
short	Numero INTERO CORTO	2 BYTE (16 bit)
long	Numero INTERO LUNGO	4 BYTE (32 bit)
float	Numero REALE (in virgola mobile)	4 BYTE (32 bit)
double	Numero REALE LUNGO	8 BYTE (64 bit)

Inoltre, oltre alle variabili, posso definire anche delle **COSTANTI** (o "macro"), ovvero:

```
#define PI_GRECO 3.14 // #define è una direttiva per il preprocessore (come #include)
```

A differenza delle variabili, dove il nome scelto (o identificatore) può avere anche lettere minuscole (come nella scrittura "Camel Case" → bottigliePerPacco), le costanti sono per convenzione scritte solo con lettere maiuscole nella scrittura "Snake Case" (PI_GRECO).

⚠️ Inoltre, nel dichiarare una variabile, non posso usare particolari termini noti come "keyword", ovvero quelle parole riservate che hanno già una loro ben specifica funzione nel C (es. non posso dichiarare una variabile con il termine "printf").

Quelli nella tabella sono i PRINCIPALI TIPI di valori, ma ce ne sono **MOLTI ALTRI**. Vediamoli più da vicino:

- **TIPI INTERI**: oltre alla dimensione di un intero ("short int" o "long int", o più semplicemente "short" e "long"), abbiamo anche interi "signed" (dove il "Most Significant Bit" [MSB, il primo bit a sinistra] indica il segno del numero → se MSB = 0, numero positivo; se MSB = 1, numero negativo) e "unsigned" (senza segno); di default, le variabili intere sono considerate "signed", ma se le vogliamo "unsigned" basta dichiararlo. Con questa nuova informazione, ora abbiamo **8 POSSIBILI TIPI DI INTERO**:

- "short int" (o solo "short")
- "unsigned short int" (o solo "unsigned short")
- "int"
- "unsigned int"
- "long int" (o solo "long")
- "unsigned long int" (o solo "unsigned long")

- “**long long int**”
- “**unsigned long long int**”

⚠ Un modo per determinare l'intervallo numerico coperto dai vari tipi interi è usare l'header `<limits.h>`, che definisce delle macro per gli estremi di rappresentazione dei vari tipi.

⚠ Il C permette la scrittura di **COSTANTI INTERE** in formato decimale (base 10), ottale (base 8) e esadecimale (base 16):

- **DECIMALI**: contengono cifre tra 0 e 9; non devono iniziare per “0”;
- **OTTALI**: contengono cifre tra 0 e 7; devono iniziare per “0”;
- **ESADECIMALI**: contengono cifre da 0 a 9 + lettere dalla “a” alla “f”; devono iniziare per “0x”;

Per forzare queste costanti ad essere: “long int” va aggiunta una “L” (o “l”) al fondo; “long long int” va aggiunta una “LL” (o “ll”) al fondo; “unsigned” va aggiunta una “U” (o “u”) al fondo.

- **TIPI FLOATING POINT** {usato lo standard IEEE-754 che descrive i numeri reali in 3 parti: segno, esponente (con il suo segno) e mantissa (o parte frazionaria)}:
 - “**float**” (a singola precisione [32 bit])
 - “**double**” (a doppia precisione [64 bit])
 - “**long double**” (con precisione estesa)
 - “floating point complessi” [stessi 3 tipi, ma riferiti ai numeri complessi]

⚠ Il C ammette diverse scritture per le **COSTANTI FLOATING POINT**, ovvero: | 57.0 | 57. | 57.0e0 | 57E0 | 5.7e1 | 5.7e+1 | .57e2 | 570.e-1| etc... Inoltre, per forzare queste costanti ad essere: “float” va aggiunta una “F” (o “f”) al fondo; “long double” va aggiunta una “L” (o “l”) al fondo.

- **TIPI PER CARATTERI** {il set attualmente più diffuso è il codice ASCII, dove 7 bit possono rappresentare 128 caratteri diversi}: ad una variabile di tipo “**char**” può essere assegnato qualsiasi carattere, ma va racchiuso tra apici singoli [char carattere = ‘a’]; lavorare con i caratteri è semplice, in quanto il C tratta i caratteri come dei “piccoli interi”, in quanto prende il valore numerico associato ad ogni carattere.

⚠ Per leggere e scrivere caratteri, si possono usare le funzioni “putchar(carattere)” e “getchar()”:

- “**putchar(carattere);**” → scrive un carattere (come printf, ma per un singolo carattere)
- “**carattere = getchar();**” → legge un carattere che poi restituisce (come scanf, ma per un singolo carattere);

Usare queste due funzioni (invece di printf e scanf) permette di risparmiare tempo e CPU nell'esecuzione di un programma (sono funzioni più semplici per l'elaboratore).

CONVERSIONI: un elaboratore può fare operazioni solo tra operandi con lo stesso numero di bit; dunque, affinché l'elaboratore possa operare, il compilatore C deve generare istruzioni per convertire gli operandi e renderli “interagibili” per l'hardware. Abbiamo due tipi di conversioni del compilatore:

- **CONVERSIONI IMPLICITE**: applicate automaticamente, senza l'intervento del programmatore; viene effettuata una “**promozione**” dell'operando con il tipo “più piccolo” (ovvero che occupa il minor numero di bit) al tipo dell'operando “più grande” → in ordine di grandezza crescente:
 - float => double => long double
 - int => unsigned int => long int => unsigned long int
- **CONVERSIONI ESPLICITE**: effettuate dal programmatore, usando l'operatore di “**casting**” → forzare il compilatore a fare le conversioni che vogliamo:

```

1. float quoziente;           // Dicho il quoziente come "float", ma per ottenere un
2. int dividendo, divisore;   // risultato più preciso devo convertire anche dividendo e
3. quoziente = (float) dividendo / divisore; // divisore in "float" prima di effettuare la divisione

```

- **VALORI BOOLEANI:** prima della versione C99, il tipo "booleano" (ovvero True and False) non esisteva nel linguaggio C; infatti i programmatori che usavano C89 usavano definire delle macro con nomi come TRUE e FALSE (#define TRUE 1 → assegno a TRUE il valore 1; #define FALSE 0 → assegno a FALSE il valore 0).

Nel C99 è stato aggiunto il tipo "_Bool", che corrisponde ad un "unsigned int" (intero senza segno): se il valore assegnato ad una variabile _Bool è 0, allora viene memorizzato il valore 0 (corrispondente a FALSE); se invece le viene assegnato un valore diverso da 0, allora viene memorizzato il valore 1 (corrispondente a TRUE).

Oltre alla definizione del tipo "_Bool", il C99 fornisce l'header <stdbool.h> che fornisce la macro "**bool**" corrispondente a _Bool (quindi semplicemente una scrittura più facile); inoltre fornisce anche delle macro chiamate "**true**" e "**false**" che corrispondono rispettivamente ad 1 e 0.

⚠ Posso anche creare un tipo "booleano" con la funzionalità detta "**definizione di tipo**" (type definition), tramite cui il compilatore aggiunge il nuovo tipo alla lista dei tipi che è in grado di riconoscere:

```
typedef int Bool;
```

⚠ Altro importante operatore è l'operatore "sizeof (nome_del_tipo)", il quale ha come valore un intero senza segno che rappresenta il numero di Byte richiesti per un determinato tipo di valori; inoltre, il tipo dell'operatore "sizeof" è chiamato "size_t" e per essere stampato a video va aggiunta una "z" tra "%" e la "u" degli unsigned int: printf("Size of int: %zu", sizeof(int));

3. INPUT/OUTPUT

printf("stringa + %operatori di formato", espressioni) → funzione che stampa a video la stringa (ovvero il primo argomento), facendo però riferimenti alle espressioni/variabili contenute nel secondo argomento:

```

1. #include <stdio.h>           // L'header <stdio.h> contiene le funzioni di I/O (come la printf)
2. int main()                   //
3. {                           //
4.     int i = 2;               // Il "\n" contenuto nella printf indica una "new line" (a capo)
5.     printf("%d \n", i);      // In questo caso mi viene stampato a video il valore contenuto
6. }                           // nella variabile "i"; il "%d" è l'operatore di formato per interi
                                // che assume il ruolo e il valore di "i" nella funzione printf()

```

Operatori di formato (o specifiche di conversione): per ogni tipo di variabile/espressione, c'è uno specifico operatore di formato:

- "%d" o "%i" → per interi in formato decimale
- "%f" → per numeri reali (float o double)
- "%c" → per un carattere
- "%s" → per una stringa di caratteri
- "%o" → per un numero in formato ottale
- "%x" o "%X" → per numero in formato esadecimale
- "%u" → per interi "unsigned" (senza segno)
- "%e" o "%E" → stampa numero reale (float/double) in notazione scientifica (esponenziale)
- "%f" → stampa numero reale sia in formato scientifico che decimale (a seconda della dimensione)
- "%%" → per stampare "%"
- "%p" → stampa un valore " void * "

⚠ Per interi "short" devo mettere una "h" tra "%" e l'operatore di conversione (d, o, u, x); per interi "long" devo mettere una "l"; per interi "long long" devo mettere una "ll".

⚠ Per un valore "double" devo mettere una "l" tra "%" e l'operatore di conversione (e, f, g); per un valore "long double" una "L".

Esempio:

```
int i = 40;
float x = 839.21
printf("%5.3d", i);           // Viene stampato | 040|, ovvero su 5 spazi con 3 cifre
printf("%-10g", x);          // Viene stampato |839.21    |, ovvero su 10 spazi ma a sinistra per il "-"
```

Sequenze di escape: permettono alle stringhe di contenere dei caratteri particolari (che causerebbero problemi al compilatore); abbiamo 2 tipi di sequenze di escape:

- **Caratteri di escape:**
 - “\a” → Alert (suono, campanella)
 - “\b” → Backspace
 - “\f” → Form feed
 - “\n” → New line (a capo)
 - “\r” → Carriage return
 - “\t” → Tab orizzontale
 - “\v” → Tab verticale
 - “\\” → Backslash
 - “\?” → Punto di domanda
 - “\’ ” → Apice singolo
 - “\” ” → Apice doppio
- **Escape numerici:** permettono di rappresentare qualsiasi carattere; per farlo dobbiamo guardare in una tabella il valore ottale/esadecimale del carattere e fare una sequenza di escape numerica con “\” seguita dal valore ottale oppure “\x” seguita dal valore esadecimale

scanf("stringa + %operatori di formato", &variabili) → funzione che legge l'input da tastiera secondo un particolare formato contenuto nella **stringa**; nelle **variabili** viene memorizzato il valore in input (le variabili vanno comunque prima dichiarate). Il carattere “&” precede gli argomenti e indica l'indirizzo della variabile.

⚠ Bisogna fare attenzione che il numero di conversioni combaci con il numero di variabili e che il tipo di operatore sia adatto a quella variabile! Inoltre, la scanf cerca nell'input il primo carattere del tipo di valore dichiarato dall'operatore, saltando dunque i caratteri di spaziatura.

⚠ Quello che viene scritto nella stringa con gli operatori di formato deve combaciare con l'input; esempio: se nella stringa della scanf ho “%d/%d” e l'input è | 5 /96|, la scanf salta il primo spazio e associa il 1° “%d” con 5, ma poi cerca di far combaciare lo spazio dopo il 5 con “/” e, dato che non combaciano, la scanf si ferma lì, lasciando il resto dei caratteri nell'input.

Ora vediamo nel dettaglio ciò che è contenuto nella parte della libreria standard dedicata all'input/output, che abbiamo sopra accennato [**<stdio.h>**, che contiene le “funzioni di I/O dei byte”]. Nel C, indichiamo con “**STREAM**” una sorgente di input o una destinazione di output; l'accesso ad uno “stream” avviene attraverso un “**puntatore a file**” che è di tipo “**FILE ***” (tipo definito in **<stdio.h>**). Gli stream standard sono:

stdin	Standard input (tastiera)	stdout	Standard output (schermo)	stderr	Standard error (schermo)
--------------	---------------------------	---------------	---------------------------	---------------	--------------------------

Questi stream standard possono essere modificati attraverso un “reindirizzamento”:

- “reindirizzamento dell’input” → **programma <stream_input.file**
- “reindirizzamento dell’output” → **programma >stream_output.file**
- “reindirizzamento di input e output insieme” → **programma <stream_input.file >stream_output.file**

Inoltre, se reindirizziamo l’output ad un file, non ci possiamo accorgere degli errori che si creano fino a che non apriamo lo stream di output: per questo motivo, possiamo scrivere gli errori su “**stderr**” in modo che, anche se **stdout** viene reindirizzato, gli errori compariranno ugualmente sullo schermo.

L’header **<stdio.h>** supporta 2 tipi di file:

- “**file testuale**” → solo caratteri; suddiviso in righe; ogni riga termina con dei caratteri speciali (in Windows termina con ‘\x0d\x0a’), mentre il file termina con un segnalatore di “EOF” [end-of-file, in Windows ‘\x1a’];
- “**file binario**” → non sono divisi in righe e non contengono segnalatori di fine riga o file;

⚠️ Se non conosciamo il tipo di un file è meglio supporre che questo sia binario.

3.1. OPERAZIONI SUI FILE

- **APRIRE UN FILE** → “**FILE * fopen(nome_file, modalità_di_apertura)**”
 - **FILE *** è il tipo della **fopen**, ovvero la **fopen** restituisce un **puntatore a file** (quando non può aprire un file, restituisce un puntatore nullo);
 - **nome_file** è una stringa con in nome del file che deve essere aperto;
 - **modalità_di_apertura** è un carattere/stringa che specifica le operazioni da compiere sul file;

⚠️ Nel **nome_file** possiamo trovare 3 scritture:

- “**nome_file**”, ovvero solo il nome del file;
- “**./nome_file**”, ovvero una scrittura con **percorso relativo**; in questa scrittura se troviamo “**..**” prima del nome del file stiamo dicendo al programma di salire di 1 directory (di un cartella), mentre se troviamo “**..**” dopo il nome del file stiamo dicendo al programma di scendere di 1 directory (di andare in una sottocartella);
- “**C:/Users/nome_file**”, ovvero il **percorso assoluto** dove viene indicato in che cartella si trova il file;

Da notare che usiamo “**/**” al posto di “****” (che sarebbe il vero indicatore di percorso in Windows), in quanto il carattere “****” introduce delle sequenze di escape; un’alternativa sarebbe usare “****” al posto di “****”.

⚠️ Possiamo **memorizzare** il puntatore a file restituito dalla **fopen** in una variabile (per poi eseguire operazioni sul file) → **fp = fopen(“nome_file”, “modalità_di_apertura”);**

Per la “**modalità_di_apertura**” abbiamo diverse tipologie:

Modalità di apertura per i FILE TESTUALI	
“ r ”	Apre file in lettura (il file deve esistere)
“ w ”	Apre file in scrittura (il file può anche non esistere)
“ a ”	Apre il file in accodamento (può anche non esistere)
“ r+ ”	Apre il file in lettura e scrittura (parte dall’inizio)
“ w+ ”	Apre il file in lettura e scrittura (tronca il file se esiste)
“ a+ ”	Apre il file in lettura e scrittura (accoda il file se esiste)

Modalità di apertura per i FILE BINARI	
“rb”	Apre file in lettura (il file deve esistere)
“wb”	Apre file in scrittura (il file può anche non esistere)
“ab”	Apre il file in accodamento (può anche non esistere)
“rb+” oppure “r+b”	Apre il file in lettura e scrittura (parte dall'inizio)
“wb+” oppure “w+b”	Apre il file in lettura e scrittura (tronca il file se esiste)
“ab+” oppure “a+b”	Apre il file in lettura e scrittura (accoda il file se esiste)

- CHIUDERE UN FILE → “**fclose**(puntatore_a_file)”

- L'argomento di “fclose”, ovvero il **puntatore_a_file**, è il puntatore a file ottenuto dalla “fopen” (o dalla “freopen” che vedremo tra poco);
- La “**fclose**” restituisce:
 - **0** se il file è stato chiuso con successo;
 - la macro **EOF** se c'è stato un errore;

⚠ Esempio:

```

1. int main(void) {
2.   FILE * fp = fopen(nome_file, "r");
3.   if (fp == NULL) {
4.     printf("Can't open %s\n", nome_file);
5.     exit(EXIT_FAILURE);
6.   }
7.   fclose(fp);
8.   return 0;
9. }
```

// Apro il file “**nome_file**” e lo associo alla variabile
// puntatore “fp”; se “fp” è uguale a NULL (macro
// che indica qualcosa di “vuoto”), viene chiamata
// la funzione **exit** con la macro “EXIT_FAILURE”
// (entrambe definite in <stdlib.h>, di cui parlerò
// in seguito). Poi il file puntato da fp viene chiuso
// con la funzione **fclose**(fp).

- COLLEGARE UN FILE CON UNO STREAM APERTO → “**freopen**(nome_file, modalità_apertura, stream)”

La “**freopen**” collega un **file** diverso con uno stream che è già stato aperto; è molto utile se vogliamo associare un file con uno degli stream standard; il file viene dunque collegato allo **stream** scritto nella **freopen**.

- FILE TEMPORANEI = file che esistono fino a che il programma è in esecuzione:

- La funzione “**tmpfile**”, associata ad un puntatore a file, crea un file temporaneo (aperto in modalità “wb+”) associato a quel puntatore:
 1. FILE * file_temporaneo;
 2. file_temporaneo = **tmpfile**(); //Creato il file temporaneo associato a file_temporaneo

Se la creazione del file non va a buon fine, la funzione **tmpfile**() restituisce un puntatore nullo, ma questa funzione crea un file di cui non possiamo conoscere il nome;
- Associando la funzione “**tmpnam**” ad un puntatore con il nome del file, possiamo dare il nome al file temporaneo creato dalla “**tmpfile**”:
 1. char *nome_file;
 2. nome_file = **tmpnam**(NULL); //Creato il file temporaneo associato a file_temporaneo

- FILE BUFFERING: dato che trasferire dati da un disco o in un disco è un'operazione lenta, si usa il “**buffering**”, ovvero i dati scritti su uno stream vengono mantenuti in un'area della memoria (buffer) e quando quest'area è piena avviene il “**flush**” del buffer (cioè il buffer viene svuotato e viene trascritto il suo contenuto nell'output). Le funzioni presenti in <stdio.h> eseguono automaticamente il buffering quando questo risulta vantaggioso, ma se volessimo operare sul buffer manualmente?

- La funzione “**fflush**” svuota il buffer del file puntato dal “**puntatore_a_file**” [**fflush(fp);**] oppure svuota tutti i buffer con [**fflush(NULL);**]. La funzione restituisce 0 (tutto ok) o EOF (errore);

- La funzione “**setvbuf**” permette di controllare il buffer → **setvbuf(stream, buffer, tipo_buffer, N);**
 - “**buffer**” = indirizzo del buffer;
 - “**N**” = numero di byte presenti nel buffer;
 - “**tipo_buffer**” = tipo di buffering desiderato, che può essere:
 - “**_IOFBF**” → full buffering (trascrizione buffer quando è pieno);
 - “**_IOLBF**” → line buffering (una riga alla volta);
 - “**_IONBF**” → no buffering;

Chiamando la **setvbuf** usando un puntatore nullo come secondo argomento, la funzione crea un buffer della dimensione specificata.

⚠ C’è anche una funzione chiamata “**setbuf(stream, buffer)**” considerata obsoleta in quanto non permette di decidere la modalità e la dimensione del buffer.

– OPERAZIONE VARIE SU FILE:

- La funzione “**remove(nome_file)**” cancella il file chiamato “**nome_file**”;
- La funzione “**rename(vecchio_nome_file, nuovo_nome_file)**” rinomina un file dal suo nome vecchio al nuovo; bisogna assicurarsi di chiudere il file prima di rinominarlo.

3.2. INPUT/OUTPUT FORMATTATO

– FUNZIONI “...printf” (controllare l’output):

- Funzione “**printf(stringa_da_printare + operatori di formato”, variabili/espressioni)**”: scrive sempre sullo **stdout** (in generale lo schermo). Come abbiamo già visto, l’**operatore di formato** (“% + carattere”) controlla come si presenta l’output;
 - Funzione “**fprintf(stream_output, “stringa + operatori”, variabili/espressioni)**”: scrive sullo **stream** specificato.
- ⚠ printf è uguale a fprintf con stdout come 1° argomento!

– OPERATORE DI FORMATO (specifiche di conversione)[[nel dettaglio](#)] → %#012.5Lg

- “**%**” → INTRODUTTORE DI FORMATO;
- “**#0**” → FLAG:

“ - ”	Allinea a sinistra
“ + ”	I numeri prodotti dalla conversioni con segno iniziano con un “+” o con un “-”
“ spazio ”	I numeri positivi prodotti dalla conversioni vengono preceduti da uno spazio
“ # ”	Gli ottali iniziano con “0”, gli esadecimali con “0x”
“ 0 ”	I numeri vengono gonfiati con degli zeri fino a riempire la larghezza del campo

- “**12**” → LARGHEZZA MINIMA DEL CAMPO;
- “**.5**” → PRECISIONE (per interi vengono messi degli 0, mentre per i float è il numero di cifre dopo il separatore decimale [.]);
- “**L**” → MODIFICATORE DI LUNGHEZZA (come gli “h”, “l”, “ll”... che abbiamo visto prima);
- “**g**” → SPECIFICATORE CONVERSIONE (visti prima);
 - ⚠ Mettendo un “*” nell’operatore di formato, specifichiamo dopo la stringa il valore che deve sostituire l’asterisco: “printf(“%6.4d”, i);” equivale a “printf(“%*.4d”, **6**, i);”.

– FUNZIONI “...scanf” (leggere l’input):

- Funzione “**scanf(stringa_da_leggere + operatori di formato”, puntatori)**”: legge sempre dallo **stdin** (in generale la tastiera). Come abbiamo già visto, l’**operatore di formato** (“% + carattere”) controlla come si presenta l’input;
- Funzione “**fscanf(stream_input, “stringa + operatori”, puntatori)**”: legge dallo **stream** specificato.

⚠ Noi non abbiamo ancora visto i puntatori, ma pensiamo ad essi come degli oggetti che indicano l'indirizzo di memoria delle variabili in cui voglio salvare i valori in input, dunque a differenza delle printf che facevano riferimento alle variabili, le scanf fanno riferimento ai loro indirizzi!

⚠ Le scanf terminano prematuramente se si verifica un “**input failure**” o un “**matching failure**” (ovvero la stringa non combacia con l’input).

⚠ Le scanf restituiscono il numero di dati che sono stati letti e assegnati dall’input.

- RILEVARE LA FINE DI UN FILE → tutti gli stream possiedono 2 indicatori: un “**indicatore di errore**” e un “**indicatore di end-of-file**” [EOF]. Una volta che gli indicatori sono stati impostati, rimangono quelli fino a che non vengono azzerati (per esempio con una chiamata alla funzione “**clearerr(fp)**” che azzera entrambi gli indicatori del file puntato da fp). Abbiamo anche le funzioni:
 - “**feof(fp)**” → dà un valore ≠ 0 se l’EOF associato al file puntato da fp è ancora impostato (altrimenti dà 0);
 - “**ferror(fp)**” → dà un valore ≠ 0 se l’indicatore di errore associato al file puntato da fp è ancora impostato (altrimenti dà 0).

3.3. I/O DI CARATTERI

- OUTPUT:
 - “**putchar(carattere)**” → scrive un carattere nello stdout (generalmente lo schermo);
 - “**fputc(carattere, stream)**” → scrive un carattere nello stream specificato;
 - “**putc(carattere, stream)**” → scrive un carattere nello stream specificato.
- INPUT:
 - “**getchar(carattere)**” → legge un carattere dallo stdin (generalmente la tastiera);
 - “**fgetc(carattere, stream)**” → legge un carattere dallo stream specificato;
 - “**getc(carattere, stream)**” → legge un carattere dallo stream specificato.

⚠ Ultima funzione è “**ungetc(carattere, stream)**” che rimette a posto l’ultimo carattere letto dallo stream e azzera l’indicatore di EOF.

3.4. I/O DI RIGHE

- OUTPUT:
 - “**puts(stringa)**” → scrive una stringa nello stdout (generalmente lo schermo); aggiunge sempre il carattere “\n” a fine stringa;
 - “**fputs(stringa, stream)**” → scrive una stringa nello stream specificato; non aggiunge il carattere “\n”;
- INPUT:
 - “**gets(stringa)**” → legge una riga dallo stdin (generalmente la tastiera); quindi l’input deve terminare con il carattere “\n” (dove la gets smette di leggere la riga e scarta il “\n”);
 - “**fgets(stringa, sizeof(stringa), stream)**” → legge una riga dallo stream specificato; il **sizeof** pone un limite al numero di caratteri che verranno salvati nella stringa: infatti la fgets legge i caratteri fino al “\n” oppure fino a che siano stati letti “**sizeof(stringa) – 1**” caratteri.

3.5. I/O DI BLOCCHI

- OUTPUT:
 - “**fwrite(a, sizeof(a[0]), sizeof(a)/sizeof(a[0]), stream)**” → copiare un vettore dalla memoria ad uno stream:
 - “**a**” = indirizzo del vettore;
 - “**sizeof(a[0])**” = dimensione di ogni elemento del vettore in byte;

- “`sizeof(a)/sizeof(a[0])`” = numero di elementi;
- INPUT:
 - “`fread(a, sizeof(a[0]), sizeof(a)/sizeof(a[0]), stream)`” → legge gli elementi di un vettore da uno stream; la funzione restituisce il numero di elementi (non i byte), che dovrebbe coincidere con il terzo argomento.

⚠ Queste 2 funzioni leggono anche non vettori!

3.6. I/O DI STRINGHE

- OUTPUT:
 - “`sprintf(stringa, “stringa + operatori”, variabili/espressioni)`” → scrive il suo output non in uno stream (come lo schermo), ma in una stringa. Restituisce il numero di caratteri salvati;
 - “`snprintf(stringa, n, “stringa + operatori”, variabili/espressioni)`” → come la sprintf ma ha il parametro aggiuntivo “n”, ovvero nella stringa verranno scritti massimo “n – 1” caratteri.
- INPUT:
 - “`sscanf(stringa, “stringa + operatori”, variabili/espressioni)`” → legge non da uno stream, ma da una stringa puntata dal suo primo argomento;

3.7. POSIZIONAMENTO NEI FILE

Ogni stream è associato con una posizione; quando viene eseguita un'operazione, la posizione avanza automaticamente in modo sequenziale. Ma se vogliamo saltare in una certa posizione del file?

- “`fseek(puntatore_a_file/stream, N-byte, da dove devo calcolare la nuova posizione)`” → il 3° argomento, che indica da dove calcolare la nuova posizione, può essere:
 - SEEK_SET = inizio del file;
 - SEEK_CUR = posizione corrente;
 - SEEK_END = fine del file.

⚠ Per gli stream testuali, possiamo usare la fseek solo per spostarci a inizio o fine dello stream (o in una posizione in cui siamo già stati).

- “`ftell(puntatore_a_file/stream)`” → dà la posizione del file corrente sotto forma di long int (oppure dà il numero “-1L” se si verifica un errore);
- “`rewind(puntatore_a_file/stream)`” → imposta la posizione del file al suo inizio;

Se dobbiamo lavorare con file molto grandi, abbiamo 2 funzioni aggiuntive:

- “`fgetpos(puntatore_a_file/stream, &file_pos)`” → salva la posizione del file nella variabile `file_pos`;
- “`fsetpos(puntatore_a_file/stream, &file_pos)`” → imposta la posizione del file al valore contenuto nella variabile `file_pos`.

4. ESPRESSIONI

Quando scriviamo un'espressione in C, possiamo affidarci a diversi tipi di OPERATORI:

- ARITMETICI:
 - “+” (unario, ovvero come segno del numero)
 - “-” (unario, ovvero come segno del numero)
 - “+” (somma)

- “ - ” (sottrazione)
- “ * ” (moltiplicazione)
- “ / ” (divisione)
- “ % ” (resto della divisione) [solo con numeri interi]

⚠ Il risultato di una divisione viene sempre arrotondato verso lo zero e il valore di $(i \% j)$ ha sempre lo stesso segno di “i” $[-9 \% 7 = -2]$.

– DI ASSEGNAZIONE:

- Assegnamento Semplice → “=” (es. `int i = 72.99f /*i = 72*/;`) richiede un “Lvalue”, ovvero un contenitore (per esempio una variabile), come operando sinistro;
- Assegnamento Composto:
 - “ a `+= b` ” → somma b ad a e memorizza risultato in a
 - “ a `-= b` ” → sottrae b ad a e memorizza risultato in a
 - “ a `*= b` ” → moltiplica a per b e memorizza risultato in a
 - “ a `/= b` ” → divide a per b e memorizza risultato in a
 - “ a `%= b` ” → calcola il resto della divisione (a/b) e memorizza il risultato in a
- DI INCREMENTO E DECREMENTO: gli operatori “`++`” (incrementa di 1) e “`--`” (decrementa di 1) possono essere usati sia come prefissi $[++i / --i]$ sia come suffissi $(i++ / i--)$; come prefissi il valore della variabile cambia già durante l’operazione di incremento/decremento, mentre come suffissi il valore cambia dopo l’operazione.

⚠ Gli operatori hanno uno rispetto all’altro una certa “gerarchia di precedenza” e un’associatività (ovvero da quale parte è associato l’operatore):

Precedenza	Nome	Simbolo	Associatività
1°	incremento (suffisso)	<code>++</code>	sinistra
	decremento (suffisso)	<code>--</code>	
2°	incremento (prefisso)	<code>++</code>	destra
	decremento (prefisso)	<code>--</code>	
	più (segno unario)	<code>+</code>	
	meno (segno unario)	<code>-</code>	
3°	moltiplicativi	<code>* / %</code>	sinistra
4°	additivi	<code>+ -</code>	sinistra
5°	assegnamento	<code>= += -= *= /= %=</code>	destra

⚠ Qualsiasi espressione può essere usata come un’istruzione, ma il suo valore viene scartato se essa non è associata ad una variabile o non fa parte di un’espressione più grande!

– DI CONFRONTO (tra due argomenti)

○ RELAZIONALI:

- “ `<` ” → minore
- “ `<=` ” → minore o uguale
- “ `>` ” → maggiore
- “ `>=` ” → maggiore o uguale

- DI UGUAGLIANZA:
 - “`==`” → uguale a
 - “`!=`” → diverso da
- LOGICI (operazioni logiche restituiscono 1 quando sono vere e 0 quando sono false):
 - “`!`” → negazione (“not”)
 - “`&&`” → “and” (entrambe verificate)
 - “`||`” → “or” (una delle due verificate)

5. ISTRUZIONI DI SELEZIONE

→ permettono al programma di selezionare un percorso da eseguire:

- ISTRUZIONE CONDIZIONALE “if”:

```
1. if (risultato_esame >= 18) {           // Il blocco “if (condizione)” contiene la condizione per
2.     printf ("Superato");               // eseguire le istruzioni dopo; se l'istruzione è 1, posso
3. }                                         // omettere le graffe, altrimenti devo accorparle nelle {}
```

- CLAUSOLA “else”: l'istruzione contenuta nella “blocco-else” viene eseguita quando non si verifica la condizione del “blocco-if” (ovvero quando la condizione dell'if è falsa, cioè il suo valore = 0):

```
1. if (risultato_esame >= 18) {
2.     printf ("Superato");
3. } else {                                // Quando risultato_esame < 18, si passa al “blocco-else”
4.     printf ("Non superato");             // e viene stampato “Non superato”
5. }
```

⚠ Se le istruzioni contenute nei blocchi sono singole e “corte”, posso essere messe sulla stessa riga della condizione if e dell'else (sconsigliato perché più disordinato, ma possibile):

```
1. if (i > j) max = i;
2. else max = j;
```

⚠ Possiamo anche “if” con dentro altri “if” nelle istruzioni: in questo caso parliamo di “nested-if” (o istruzioni if annidate); bisogna far particolare attenzione a queste strutture, perché il “blocco-else” presente in esse si lega al “blocco-if” libero più vicino sopra l'else:

```
1. if (i > j)                         // Proprio a causa della confusione del codice, è buona norma per
2.     if (i > k)                     // il programmatore inserire le varie istruzioni in delle {}
3.         max = i;
4.     else
5.         max = k;
```

- ISTRUZIONI “if” IN CASCATA (o “if - else if - else”): quando dobbiamo verificare più condizioni e casi, al posto di usare tanti “if” annidati, posso usare la struttura:

```
1. if (risultato_esame >= 18) {          // Se risultato_esame >= 18, viene stampato “Superato”,
2.     printf ("Superato");              // altrimenti si passa all'if seguente, ovvero quello nel
3. } else if (risultato_esame >= 15) {    // blocco “else if” e, se risultato_esame >= 15, viene
```

```

4.     printf ("Orale");
      // stampato "Orale", altrimenti si passa all'if seguente;
5. } else {
      // in questo caso non ci sono altri if e dunque si passa
6.     printf ("Non superato");
      // all'else finale che stamperà "Non superato".
7. }
```

- **OPERATORE CONDIZIONALE** (detto anche “operatore TERNARIO” perché è l’unico operatore in C che richiede 3 operandi): ha struttura => **espressione1 ? espressione2 : espressione3** => serve a scrivere in maniera più compressa un’espressione condizionale; il “blocco-if” seguente equivale all’operatore ternario affianco:

```

1. if (i > j)
2.     return i;
3. else
4.     return j;
```

=

```
1. return (i > j) ? i : j;
```

- **ISTRUZIONE “switch”**: l’istruzione “switch (variabile)” è come una serie di “else if” regolati dal valore che può assumere la variabile; esempio:

```

1. switch(voto) {
2.   case 3:
3.     printf("Ottimo");
4.     break;
5.   case 2:
6.     printf("Buono");
7.     break;
8.   case 1:
9.     printf("Discreto");
10.    break;
11.   case 0:
12.     printf("Bocciato");
13.     break;
14.   default:
15.     printf("Voto senza senso");
16.     break;
17. }
```

// L’istruzione **switch** salta tra i vari **casi** a seconda del // valore assunto dalla variabile (**voto**); se **voto = 3**, viene // stampato “**Ottimo**” e il percorso si interrompe lì con il // **break**; altrimenti, se il **voto** è diverso da **3**, si va avanti // verso il basso al **prossimo caso** (come in cascata) e così // via. Se il valore assunto da **voto** non è nessuno dei casi, // si passa al caso “**default**” e viene stampato “**Voto senza // senso**”.

⚠ Negli switch non si possono usare come variabili i numeri float e le stringhe

⚠ Inoltre, per risparmiare spazio, si possono mettere più etichette “case” sulla stessa riga:

```

1. switch (voto) {
2.   case 4: case 3: case 2: case 1: printf("Promosso");
3.             break;
4.   case 0: printf("Bocciato");
5.             break;
6.   default: printf("Voto senza senso");
7.             break;
8. }
```

6. CICLI

Un ciclo serve a eseguire ripetutamente delle istruzioni (contenute nel **corpo del ciclo**). In C, ogni ciclo possiede un'espressione di controllo che viene analizzata ad ogni esecuzione ("**iterazione**") del ciclo; se l'espressione è vera (valore espressione $\neq 0$), il ciclo continua ad iterare. Ci sono 3 tipi di cicli:

- **CICLO "while"** → continua ad iterare fino a che l'espressione di controllo è vera (valore $\neq 0$):

```
1. int i = 1;
2. while (i < 8) {                                // Il ciclo while continua ad iterare fino a che l'espressione di
3.     i *= 2;                                     // controllo è vera, ovvero fino a quando "i" è minore di 8 e ad
4. }                                              // ogni iterazione il valore di "i" viene moltiplicato per 2.
```

⚠ Anche qui, se l'istruzione nel corpo del while è unica, posso anche omettere le graffe, altrimenti vanno obbligatoriamente messe, in quanto deve risultare soltanto 1 il corpo del while (le varie istruzioni vanno accorpate).

⚠ Se dobbiamo creare un "**ciclo infinito**", ovvero che itera all'infinito e che si può fermare solo con una specifica condizione/azione dell'utente, allora posso creare un ciclo while con una condizione di controllo "sempre vera", ovvero che il suo valore sia sempre diverso da 0 [del tipo **while (1)**].

Esercizio utile: /*Stampare la tavola dei quadrati con un while*/

```
1. #include <stdio.h>
2. int main()
3. {
4.     int i, n;
5.     printf("Questo programma stampa la tavola dei quadrati.\n");
6.     printf("Inserisci il numero dove vuoi arrivare a vedere il quadrato: ");
7.     scanf("%d", &n);
8.
9.     i = 1;
10.    while (i <= n) {
11.        printf("%10d%10d\n", i, i*i);           // Stampo "i" ed il suo quadrato (i*i)
12.        i++;                                 // Ad ogni iterazione, "i" viene aumentato di 1
13.    }
14.    return 0;
15. }
```

- **CICLO "do"** → è uguale all'istruzione while, ma l'espressione di controllo viene testata dopo l'iterazione del ciclo; questo consente al "ciclo do" di eseguire le istruzioni contenute nel **corpo del ciclo** almeno 1 volta:

```
1. int i = 1;
2. do {                                         // Il ciclo do esegue l'istruzione contenuta nel {corpo del ciclo} e
3.     i *= 2;                                    // solo dopo la prima iterazione, viene verificata l'espressione
4. } while (i < 8);                           // di controllo contenuta nell'istruzione while.
```

- **CICLO "for"** → ha 3 parametri e la seguente struttura:

```
1. int i;                                       // Il ciclo for itera per un certo numero di volte a seconda delle
2. for (i = 10; i > 0; i--) {                   // istruzioni contenute nei parametri; i parametri sono:
3.     printf("%d \n", i);                      // 1°) INIZIALIZZAZIONE VARIABILE DI CONTROLLO
```

```

4. } // 2°) CONDIZIONE PER ITERARE
5.   // 3°) INCREMENTO/DECREMENTO DELLA VARIABILE

```

⚠ Possiamo usare anche più di una variabile nel ciclo for, ottenendo così un “ciclo dinamico”, ovvero:

```

1. int high, low; // Tramite l'operatore "virgola", abbiamo
2. for (high=100, low=0; high >= low; high--, low++) // 2 variabili di controllo ("high" e "low")
3.   printf("H = %d - L = %d \n", high, low);

```

⚠ Ed inoltre anche qui possiamo omettere le graffe se l'istruzione contenuta nel corpo del ciclo è unica!

⚠ Se ometto il 1° (inizializzazione) e il 3° (incremento/decremento) parametro, allora il ciclo for è un ciclo while “sotto mentite spoglie”; se viene omesso il 2° (condizione per iterare), la condizione è considerata vera di default e quindi il ciclo for in questione è un “**ciclo infinito**” (infatti il “ciclo for infinito” assume forma [for (; ;)].

⚠ Inoltre, il 1° parametro (inizializzazione della variabile di controllo), può contenere anche la dichiarazione della variabile stessa [for (**int i = 0**; i < n; i++)].

- [USCIRE DA UN CICLO? 3 ISTRUZIONI](#) per farlo:

- **ISTRUZIONE “break”** → già vista nell'istruzione “switch”, permette di stoppare l'iterazione ed uscire dal ciclo (non avverrà un'altra iterazione dopo il break);
- **ISTRUZIONE “continue”** → conclude l'iterazione corrente e salta alla prossima iterazione del ciclo (non esce dal ciclo);
- **ISTRUZIONE “goto”** → permette di saltare verso una qualsiasi istruzione (provista di una label, ovvero di un identificatore), contenuta nella funzione; ha struttura “**goto istruzione;**” con “**istruzione**” = l'istruzione verso cui dobbiamo saltare, e l'istruzione deve avere un identificatore, ovvero deve presentarsi nella forma “**identificatore : istruzione**”.
- **ISTRUZIONE VUOTA** (o “null statement”): istruzione sprovvista di simboli ad eccezione del “;” alla fine; è utile se non dobbiamo dare istruzioni in un ciclo, per esempio, ma va fatto solamente iterare (come nei cicli infiniti)

7. VETTORI (Array)

Il C supporta anche 2 **variabili aggregate** (che memorizzano delle collezioni di valori): i vettori e le strutture. I vettori contengono elementi tutti dello stesso tipo; questi elementi possono essere selezionati individualmente tramite la loro posizione (“indice”) nel vettore. I vettori si possono classificare in base alle loro dimensioni in:

- **VETTORI UNIDIMENSIONALI** (1 dimensione): per dichiarare un vettore dobbiamo specificarne il tipo e il numero dei suoi elementi (**int a[9]**). Per accedere, invece, ad un elemento del vettore, dobbiamo scrivere il nome del vettore seguito dal numero intero che indica la posizione (“indice”) dell'elemento nel vettore (**a[2]**); gli **INDICI** degli elementi di un vettore di lunghezza “N” sono compresi tra $0 \leq \text{indice} \leq N - 1$ (per esempio, se il vettore contiene 10 elementi, questi hanno indici da 0 a 9).

⚠ Bisogna fare attenzione che l'indice non vada fuori dall'intervallo ammesso per il vettore!

⚠ Inoltre, espressioni della forma **a[i]** (con **i** = indice dell'elemento) sono degli “Lvalue”, dunque possono essere usate come variabili e posso assegnare ad un elemento del vettore nella posizione “**i**” un valore che scelgo → **a[i] = 5**

Posso inizializzare un vettore tramite un “**INIZIALIZZATORE**”, ovvero una lista di espressioni costanti racchiuse tra parentesi graffe e separate da virgole → `int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};` se l’inizializzatore è più corto della lunghezza del vettore annunciata nelle parentesi quadre, allora agli elementi restanti viene imposto il valore zero. Dunque possiamo inizializzare un vettore con l’enunciato `int a[10] = {0},` che equivale a `int a[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};`

⚠ L’inizializzatore non può essere completamente vuoto (`int a[10] = {}` è sbagliato!) e non può essere più lungo della lunghezza del vettore stesso; inoltre se è presente un inizializzatore, si può omettere la lunghezza del vettore nella sua definizione → `int a[] = {1, 2, 3, 4, 5, 6, 7} →` in questo modo, il vettore assume la lunghezza dell’inizializzatore (ovvero equivale a `int a[7];`).

⚠ Se vogliamo inizializzare solo una parte del vettore, possiamo usare i “**designatori inizializzati**”, ovvero:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48}           // I numeri tra parentesi quadre sono le posizioni degli elementi
                                                       // del vettore a cui va associato il valore a destra dell’uguale; agli
                                                       // altri elementi del vettore viene associato il valore 0
```

Se omettiamo la lunghezza del vettore con i designatori inizializzati, la lunghezza del vettore è = alla posizione dell’ultimo elemento dei designatori + 1 (nel caso di sopra sarebbe 15, in quanto l’ultimo elemento è alla posizione [14]).

⚠ Un vettore può usare contemporaneamente gli inizializzatori e i “**designatori inizializzati**” (misto).

Possiamo anche usare l’operatore “sizeof” per misurare la dimensione di un vettore (in Byte) o la dimensione di un elemento del vettore; per ottenere la **lunghezza** del vettore (“len”, ovvero il suo numero di elementi), ci basta dividere la dimensione del vettore per la dimensione di un elemento:

$$\text{len}(a) = \frac{\text{sizeof}(a)}{\text{sizeof}(a[0])}$$

Per esempio, se non sappiamo quanto è lungo il vettore “`a`”, possiamo usare in un ciclo “`for`” (che richiede la sua lunghezza come parametro) l’espressione: `for (i = 0; i < (int)(sizeof(a) / sizeof(a[0])); i++) ...`

- **VETTORI MULTIDIMENSIONALI:** un vettore può avere un qualsiasi numero di dimensioni; con la dichiarazione “`int a[5][9];`” creo per esempio un **vettore bidimensionale** (una **MATRICE** o tabella) con 5 righe e 9 colonne. Per accedere ad uno **specifico elemento** della matrice “`a`” che si trova alla riga “`i`” e alla colonna “`j`” dobbiamo scrivere “`a[i][j];`”

⚠ Inoltre, per inizializzare un vettore bidimensionale, dobbiamo usare un inizializzatore del tipo:

1. `int a[5][9] = {{1,2,2,3,3,4,4,0,6},` // Annidiamo gli inizializzatori unidimensionali
`{1,1,2,3,5,4,7,5,8},`
`{1,4,2,3,3,4,4,1,5},`
`{1,9,2,3,7,4,3,5,6},`
`{1,1,2,3,3,4,2,3,3}};`

Gli inizializzatori dei vettori multidimensionali, essendo generati a partire dall’annidamento degli inizializzatori unidimensionali, seguono le loro stesse regole, ma con particolari accorgimenti:

- Se un inizializzatore non è abbastanza grande per riempire l’intero vettore, allora gli elementi rimanenti vengono imposti a “0”;
- Se sono specificate le lunghezze di righe e colonne, possiamo omettere le parentesi graffe interne, in quanto una volta riempita una riga di elementi, passa alla seconda e negli spazi rimanenti mette degli 0;
- Designatori inizializzati per vettori bidimensionali → `int a[2][2] = {[0][0] = 1, [1][1] = 2};`
- **VETTORI A LUNGHEZZA VARIABILE (VLA)** → la lunghezza viene calcolata quando il programma è in esecuzione e non quando viene compilato:

```
int a[n];                                // Lunghezza del vettore dipende dalla variabile “n”
```

8. FUNZIONI

Le **funzioni** sono piccoli programmi con le loro dichiarazioni e istruzioni, implementate nel programma principale: mi permettono dunque di suddividere il programma in parti più piccole e quindi più facili da scrivere e da modificare in caso di errore (in quanto si vede subito da quale parte del programma esce l'errore); un'altra utilità è che sono riutilizzabili in altri programmi.

- Definire una funzione:

```
1. tipo_restituito nome_funzione(parametro1, parametro2, ...)// tipo del valore che restituisce la funzione
2. {
3.     dichiarazioni e istruzioni;                                // nome della funzione (che serve a invocarla)
4. }                                                               // nome e tipo dei parametri usati dalla funzione
                                                               // corpo della funzione
```

⚠ Le funzioni **non possono restituire vettori**; inoltre se la funzione non restituisce valori, il tipo è “**void**”

⚠ Per ogni parametro deve essere **specificato il tipo separatamente**; se una funzione non ha nessun parametro, tra le parentesi deve comparire un “**void**”

⚠ Il corpo di una funzione il cui tipo restituito è “**void**” **può anche essere vuoto** (viene detta “**funzione void**”)

- Chiamare una funzione → una funzione si invoca con “**nome_funzione(argomenti)**”; gli argomenti sarebbero i valori che forniamo alla funzione e prendono il posto dei parametri (parametro = valore “finto” con cui viene scritta la funzione; argomento = valore “effettivo” che va usato nella chiamata alla funzione).

Esempio:

```
1. /* Controlla se numero è primo */
2. #include <stdbool.h>                                     // Includo le librerie standard dei booleani e di I/O
3. #include <stdio.h>
4. bool is_prime(int n)                                       // Definisco la funzione “is_prime” di tipo “booleano” in
5. {                                                       // quanto mi restituisce il valore “true”
6.     int divisor;
7.     if (n <= 1)
8.         return false;
9.     for (divisor = 2; divisor * divisor <= n; divisor++)
10.        if (n % divisor == 0)
11.            return false;
12.    return true;                                         // return “valore” → indica il valore che mi restituisce “is_prime”
13. }
14.
15. int main(void)
16. {
17.     int n;
18.     printf("Enter a number: ");
19.     scanf("%d", &n);
20.     if (is_prime(n))                                      // Qui viene chiamata (invocata) la funzione “is_prime”
21.         printf("Prime \n");                                // che prima ho definito
22.     else
23.         printf("Not Prime \n");
24.     return 0;
25. }
```

⚠ Se però non voglio / non posso definire la funzione prima di invocarla, come posso fare? In questo caso, se la definizione della funzione si trova dopo la sua “chiamata nel main”, allora posso anteporre al main stesso una **DICHIARAZIONE della funzione**, ovvero:

```

1. /* Controlla se numero è primo */
2. #include <stdbool.h>
3. #include <stdio.h>
4. bool is_prime(int n);           // Dichiara la funzione "is_prime", altrimenti il main non
5. int main(void)                // riconosce nulla alla chiamata di "is_prime" (se non la
6. {                            // dichiarassi nemmeno, il main si chiederebbe "ma che
7.     ...                      // cosa diavolo è is_prime?")
8.     if (is_prime(n))          // La chiamata di "is_prime" si trova sopra la definizione
9.     ...
10. }
11.
12. bool is_prime(int n)          // La definizione di "is_prime" si trova sotto la chiamata
13. {
14. ...
15. }
16.

```

Come abbiamo detto prima gli argomenti di una funzione sono i valori effettivi con cui viene chiamata la funzione (prendono il posto dei parametri usati nella definizione); ma cosa succede se il tipo di un argomento non coincide con il tipo del parametro usato nella definizione della funzione? Avvengono delle **CONVERSIONI**:

- Se il compilatore ha incontrato la definizione/dichiarazione della funzione, gli argomenti vengono convertiti nel tipo dei parametri;
- Se il compilatore non ha incontrato nulla della funzione, gli argomenti subiscono le “promozioni” al tipo dei parametri (dunque questo non bisogna mai farlo accadere → **bisogna sempre anteporre alla chiamata di una funzione la sua definizione o per lo meno la sua dichiarazione**).

⚠ Se un parametro è costituito da un vettore multidimensionale, nella dichiarazione del parametro si può omettere solo la prima dimensione; usando un VLA come parametro, possiamo indicare specificatamente che la variabile “n” è la lunghezza del vettore “a” → int somma_vettori(int n, int a[n])... Possiamo però anche mettere come lunghezza del vettore un asterisco (int a[*]).

⚠ Inoltre, se vogliamo chiamare una funzione che ha come parametro un vettore, ma il vettore non è stato dichiarato prima della chiamata alla funzione, allora possiamo usare un “**letterale composto**” (un vettore creato al volo specificando semplicemente gli elementi che contiene) → totale = somma_vettori([int[]]{3, 0, 3, 4, 5}, 5).

- **L'espressione “return”** → una funzione “non-void” deve usare l'espressione “return” per specificare il valore che verrà restituito al terminare della chiamata alla funzione stessa

⚠ Il “return” si può anche usare nelle funzioni void, ammesso che non venga messo nulla dopo il “return” (quindi in modo che ci sia solo l'espressione “return;” per far terminare la chiamata della funzione).

- **L'espressione “exit”** → appartiene all'header `<stdlib.h>`; per indicare che il programma è terminato normalmente usiamo “exit(0);” oppure “exit(EXIT_SUCCESS);”, mentre per indicare che il programma è terminato in maniera anormale usiamo “exit(EXIT_FAILURE);”.

⚠ La differenza tra “return” ed “exit” è che “exit” causa la fine del programma in qualsiasi funzione essa si trovi, mentre “return” causa la fine del programma solo se si trova nella funzione “main”!

- **Funzione “ricorsiva”** = funzione che chiama se stessa (troviamo la funzione chiamata nel corpo della funzione stessa); esempio → il FATTORIALE ($n!$):

```

1. int fact(int n)
2. {
3.   if (n <= 1)
4.     return 1;
5.   else
6.     return n * fact(n - 1);           // La funzione dichiara se stessa nel suo corpo
7. }
```

8.1. VARIABILI LOCALI [vs] GLOBALE

- **Variabile LOCALE** = dichiarata nel corpo di una funzione:
 - Il suo spazio viene allocato quando la funzione che la contiene viene invocata e deallocato quando questa ha termine
 - Il suo “scope” (ovvero la porzione di codice entro cui si può fare riferimento alla variabile) è “di blocco”, ovvero la variabile locale è visibile dalla sua dichiarazione alla fine del corpo della funzione che la contiene

⚠ Posso usare l'enunciato “static” davanti alla dichiarazione di una variabile locale per usarla come un posto in cui nascondere dei dati alle altre funzioni del programma e mantenerli per le future chiamate della funzione.

⚠ I parametri contenuti nella definizione della funzione hanno le stesse caratteristiche delle variabili locali.

- **Variabile GLOBALE** (o esterna) = dichiarata fuori dal corpo di qualsiasi funzione (anche fuori dal main):
 - Ha memorizzazione statica (come per le variabili locali “static”)
 - Il suo “scope” è “di file”, ovvero visibile dalla sua dichiarazione fino alla fine del file che la contiene

⚠ Svantaggi delle variabili esterne:

- Modificando una variabile esterna, potrebbero esserci ripercussioni su tutto il programma;
- Se ad una variabile esterna fosse assegnato un valore non corretto, sarebbe difficile identificare la funzione responsabile (potrebbero essere state tutte, andrebbero dunque controllate una per una);
- Funzioni basate su variabili esterne sono difficili da implementare in altri programmi.

⚠ Suggerimenti per un “realizzare un buon programma”:

- Organizzare il programma in blocchi (blocco = {dichiarazioni + istruzioni}, ovvero come il corpo di una funzione);
- Far precedere ogni definizione di funzione da un commento contenente il nome, lo scopo e i parametri della funzione.

9. LA LIBRERIA STANDARD

La libreria standard del C contiene 24 header (prototipi di funzioni, definizioni di tipi e di macro):

<assert.h>	Autodiagnosi dei programmi	<signal.h>	Condizioni eccezionali (interruzioni e errori di “run-time”)
<complex.h>	Numeri complessi	<stdarg.h>	Scrivere funzioni con un numero variabile di argomenti
<ctype.h>	Classificazione dei caratteri + conversione (maiuscolo – minuscolo)	<stdbool.h>	Bool, true, false
<errno.h>	Controllare errori durante le chiamate alle funzioni	<stddef.h>	Definizioni dei tipi e delle macro più usate
<fenv.h>	Flag di stato e modi di controllo per i “floating-point”	<stdint.h>	Caratteristiche dei tipi interi

<code><float.h></code>	Caratteristiche dei “floating-point”	<code><stdio.h></code>	Input/Output (+ operazioni sui file)
<code><inttypes.h></code>	Macro delle stringhe di formato per particolari tipi di interi (+ grandi)	<code><stdlib.h></code>	Funzioni escluse dagli altri header
<code><iso646.h></code>	Ortografie alternative (per mancanza di alcuni caratteri)	<code><string.h></code>	Gestione delle stringhe
<code><limits.h></code>	Caratteristiche degli interi	<code><tgmath.h></code>	Gestisce le chiamate a “math” o a “complex” a seconda dei dati
<code><locale.h></code>	Adattare programma a regione geografica	<code><time.h></code>	Data e ora in C
<code><math.h></code>	Funzioni matematiche	<code><wchar.h></code>	Caratteri multibyte e wide-char
<code><setjmp.h></code>	Saltare attraverso il programma	<code><wctype.h></code>	Classificazione dei wide-char

In particolare, noi vediamo:

- `<stddef.h>`: non contiene funzioni ma solo macro e tipi più usati, come:
 - “ptrdiff_t” → tipo derivato dalla sottrazione di 2 puntatori (signed);
 - “size_t” → tipo restituito dall’operatore “sizeof” (unsigned);
 - “wchar_t” → tipo sufficientemente grande da poter rappresentare tutti i possibili caratteri;
 - “NULL” → macro che rappresenta il “null pointer”;
 - “offsetof” → macro che calcola il numero di byte compresi tra l’inizio di una struttura e il membro specificato;
- `<stdbool.h>`: già visto in precedenza;
- `<stdio.h>`: già visto in precedenza;

11. STRINGHE

Una **stringa letterale** è una sequenza di caratteri racchiusa tra doppi apici (“ ”); le stringhe possono contenere anche sequenze di escape. Se troviamo una stringa che è **troppo lunga** per essere scritta su una riga, possiamo continuare nella riga successiva attraverso un “backslash” (\); un altro modo è chiudere le virgolette a fine riga e riaprirle nella riga sotto per continuare la stringa.

In C una stringa letterale di lunghezza n viene allocata in $n + 1$ byte di memoria (ultimo byte per il carattere terminatore di stringa, ovvero il “**carattere null**” (\0) → un byte con 8 bit = 0). Per esempio, la stringa vuota (“”) viene memorizzata con un singolo carattere null.

Dato che la stringa viene memorizzata come un **vettore** di caratteri, il compilatore la tratta come un **puntatore** di tipo **“char *”** (char *p = “abc”;). Dato che sono vettori, possiamo **indicizzare** le stringhe (char ch = “abc”[1]; ovvero assegno alla variabile ch di tipo char il carattere ‘b’ (ricorda che con “abc”[3] si richiama il carattere null). Per memorizzare una stringa di “N” caratteri dobbiamo avere un **vettore unidimensionale di caratteri grande “N+1”** (l’elemento in più è il carattere null (\0) → stringa = vettore di caratteri che termina con il carattere null). Dunque possiamo inizializzare una variabile stringa con una scrittura del tipo:

```
char stringa[5] = “ciao” → {‘c’, ‘i’, ‘a’, ‘o’, ‘\0’} // La grandezza del vettore è 5 per contenere anche il “\0”
```

Se un inizializzatore di stringa (come “ciao”) è **più corto** della grandezza del vettore, il compilatore aggiunge negli spazi vuoti dei caratteri null aggiuntivi:

```
char stringa[7] = “ciao” → {‘c’, ‘i’, ‘a’, ‘o’, ‘\0’, ‘\0’, ‘\0’}
```

Se invece un inizializzatore è **più lungo** della grandezza del vettore, il C non ci farà accedere al vettore creato come se volessimo accedere una stringa (dato che non sarà presente il carattere null, quel vettore non è una stringa!).

Da ciò abbiamo capito che possiamo definire una stringa sia come un vettore di caratteri sia come un puntatore:

```
char stringa[ ] = "ciao";           // L'unica differenza tra le due scritture è che una stringa come vettore
char *stringa = "ciao";            // può essere modificata modificandone gli elementi.
```

Abbiamo già visto l'I/O di stringhe nel capitolo di I/O, dunque ora lo saltiamo, ma vediamo com'è possibile leggere le stringhe carattere per carattere:

```
1. int read_line(char str[], int n);
2. {
3.     int ch, i = 0;
4.     while ((ch = getchar()) != "\n")
5.         if (i < n)
6.             str[i++] = ch;
7.     str[i] = "\0";           // Terminatore di stringa (\0);
8.     return i;              // Ritorna il numero di caratteri memorizzati.
9. }
```

FUNZIONI PER STRINGHE (header `<string.h>`) → #include <string.h> (i parametri costituiti da stringhe hanno il tipo “`char *`”):

- “`strcpy(stringa1, stringa2);`” → copia la stringa puntata da `stringa2` nel vettore puntato da `stringa1`, ovvero copia in `stringa1` i caratteri presenti in `stringa2` fino al 1° carattere null incontrato in `stringa2`. Questa funzione è utile in quanto non potendo usare l’assegnamento (`stringa1 = "abc";`) possiamo usare questa funzione (`strcpy(stringa1, "abc");`) ma se i caratteri della `stringa2` sono troppi per essere contenuti nel vettore puntato da `stringa1`? Beh, in questo caso possiamo usare la funzione “`strncpy(stringa1, stringa2, sizeof(stringa1));`”.
- “`len_stringa = strlen(stringa);`” → restituisce come intero `unsigned` di tipo “`size_t`” la lunghezza della stringa (numero caratteri fino al null escluso).
- “`strcat(stringa1, stringa2);`” → concatena 2 stringhe (unire 2 stringhe in 1 accodandole); se vogliamo porre un limite al numero di caratteri copiati, dobbiamo usare la “`strncat(stringa1, stringa2, sizeof(stringa1) - strlen(stringa1) - 1);`”.
- “`strcmp(stringa1, stringa2);`” → restituisce un valore:
 - `strcmp(stringa1, stringa2) < 0` se `stringa1 < stringa2`;
 - `strcmp(stringa1, stringa2) > 0` se `stringa1 > stringa2`;
 - `strcmp(stringa1, stringa2) = 0` se `stringa1 = stringa2`;

Si basa sull’ordinamento lessicografico e sulla dimensione della stringa.

⚠ Ricorda che un ‘carattere’ è rappresentato da un intero, mentre una “stringa” è rappresentata da un puntatore! (`'a' ≠ "a"`).

VETTORI DI STRINGHE → per rappresentare un vettore di stringhe, potremmo usare un vettore bidimensionale (matrice); ma la scelta migliore è un “rugged array” (vettore frastagliato), ovvero le cui varie righe hanno lunghezza diversa a seconda della stringa: per farlo lo definiamo come un vettore di puntatori a stringa:

```
1. char *pianeti[] = {"Mercurio", "Venere", "Terra",           // In questo modo non sprechiamo spazio
2.                      "Marte", "Giove", "Saturno", // nella tabella!
3.                      "Urano", "Nettuno"};
```

ARGOMENTI DELLA RIGA DI COMANDO: sulla prima riga del programma, spesso capita di dover dare delle particolari istruzioni al programma e queste istruzioni vengono dette “argomenti della riga di comando” (o

parametri del programma). Per accedere a questi argomenti, dobbiamo definire il main come una funzione con 2 parametri “int **main(int argc, char *argv[])**”:

- “**int argc**” → “argument count” = numero di argomenti della riga di comando (incluso il nome del programma come primo argomento);
- “**char *argv[]**” → “argument vector” = vettore di puntatori agli argomenti della riga di comando (memorizzati come stringhe).

⚠ “**argv[0]**” punta al nome del programma, mentre “**argv[argc]**” è un puntatore nullo (gli altri argomenti della riga di comando vanno da “**argv[1]**” a “**argv[argc - 1]**”).

10. PUNTATORI

La memoria è divisa in BYTE (8 bit); ogni byte possiede un INDIRIZZO univoco: se abbiamo n byte, avremo indirizzi da 0 a $n - 1$. Ogni variabile nei nostri programmi occupa dei byte in memoria e l'indirizzo del suo 1° byte viene considerato l'indirizzo della variabile stessa.

Con le **VARIABILI PUNTATORE** memorizziamo l'indirizzo di una variabile i in una variabile p che “punta” ad i ; dunque un **PUNTATORE** è un indirizzo e una variabile puntatore memorizza quell'indirizzo. Quando dichiaro una variabile puntatore devo anteporre il nome della variabile con un **asterisco [*]**; il **tipo dichiarato** indica che il puntatore punta a oggetti di quel tipo (può puntare solo al tipo dichiarato!).

Per trovare invece l'**INDIRIZZO** di una variabile useremo l'operatore “**&**” (indirizzo) [se x è una variabile, &x è il suo indirizzo di memoria]; mentre, come spiegato prima, per accedere all'oggetto puntato da un puntatore useremo “*****” (indirection) [se p è un puntatore *p è l'oggetto puntato da p]. Dunque dopo aver dichiarato una variabile puntatore con “**tipo_dichiarato *p;**”, la si può inizializzare assegnandole l'indirizzo di una variabile con “**p = &i;**”. Possiamo anche unire dichiarazione e inizializzazione con “**tipo_dichiarato *p = &i;**”.

⚠ Fino a quando p punta ad i , $*p$ è un **ALIAS** per i (modificando il valore di $*p$, modifico anche il valore di i).

⚠ Allo stesso oggetto possono puntare più variabili puntatore!

I puntatori sono comodi per passarli alle funzioni come alias di variabili, ovvero invece di passare la variabile “ x ” alla funzione, le passo $*p$ (con $p = &x$). In questo modo la funzione potrà leggere e modificare “ x ” tramite $*p$. Se invece non vogliamo che una funzione modifichi il $*p$, devo dichiarare il parametro della funzione preceduto da “**const**” [$f(const tipo_dichiarato *p)$].

⚠ Possiamo anche scrivere **funzioni che restituiscono puntatori**; esempio:

1. **int *max(int *a, int *b)**
2. {
3. if (*a > *b) **return** a;
4. else **return** b;
5. }
6. **int *p, i, j;**
7. **p = max(&i, &j);** // p ora punterà al maggiore tra “i” e “j”

10.1. PUNTATORI E VETTORI

Il C permette di eseguire dell'aritmetica sui puntatori che puntano a elementi di un vettore. Questo porta ad un modo alternativo per elaborare i vettori, nel quale i puntatori prendono il posto degli indici. Per esempio:

1. **int a[10], *p;**
2. **p = &a[0];** // p ora punta al primo elemento del vettore a
3. ***p = 5;** // Attraverso p abbiamo memorizzato il valore 5 in a[0]

OPERAZIONI CON I PUNTATORI:

- **PUNTATORE ± INTERO:** se p punta all'elemento $a[i]$, $p + j$ punta all'elemento $a[i + j]$ (ammesso che $a[i + j]$ esista); allo stesso modo $p - j$ punta all'elemento $a[i - j]$.
- **PUNTATORE - PUNTATORE:** se p punta all'elemento $a[i]$ e se q punta all'elemento $a[j]$, $p - q =$ alla distanza tra i due indici ($p - q = i - j$); dunque sottrarre due puntatori è valido se i puntatori puntano agli elementi dello stesso vettore.
- **CONFRONTARE PUNTATORI:** posso usare gli operatori relazionali e di uguaglianza per confrontare puntatori, ma è utile solo se i puntatori puntano ad elementi dello stesso vettore:

```
1. p = &a[5];           // p <= q = 0 perché 5 > 1, mentre p >= q = 1 perché 5 > 1
2. q = &a[1];
```

- **PUNTATORI A LETTERALI COMPOSTI** → al posto di scritture come:

```
1. int a[] = {3, 0, 3, 4, 1};
2. int *p = &a[0];
```

posso usare il letterale composto “**int *p = [int []] {3, 0, 3, 4, 1};**” che fa puntare a p il 1° elemento del vettore.

⚠ Dunque l'aritmetica dei puntatori ci permette di visitare tutti gli elementi di un vettore incrementando ripetutamente una variabile puntatore (come se fosse il nostro indice del vettore); esempio:

```
1. #define N 10
2. int a[N], sum = 0, *p;
3. for (p = &a[0]; p < &a[N]; p++)      // Facendo avanzare il puntatore fino all'elemento a[N],
4.     sum += *p;                         // scandisco tutto il vettore (ed in questo caso ne sommo i valori).
```

Questo ciclo sarebbe equivalente al ciclo “while”:

```
1. #define N 10
2. int a[N], sum = 0, *p;
3. p = &a[0];                          // Trovo prima l'inizializzazione di p, poi la condizione e infine
4. while (p < &a[N])                 // l'incremento di p per iterare tutto il vettore.
5.     sum += *p++;
```

⚠ Attenzione all'uso degli operatori di incremento e decremento su puntatori:

- Con “**(*p)++**” incremento il valore della variabile a cui è associato il puntatore p ;
- Con “***p++**” incremento invece il puntatore che itera sul vettore.

⚠ Posso anche usare il nome di un vettore come un puntatore al 1° elemento del vettore stesso:

```
1. *a = 7;                           // Equivalenti a dire "a[0] = 7;"
2. *(a+1) = 12;                     // Equivalenti a dire "a[1] = 12;"
```

Il ciclo prima visto può infatti essere semplificato con:

```
1. #define N 10
2. int a[N], sum = 0, *p;
3. for (p = a; p < a + N; p++)
4.     sum += *p;
```