

Introduzione al linguaggio C

- Le sue origini risalgono al 1969 quando Ken Thompson creò il linguaggio B per scrivere il sistema operativo UNIX su piattaforma *PDP-7*
- Nel 1972 il linguaggio B venne perfezionato dando luogo ad una variante detta NB che permise di scrivere il sistema operativo UNIX per piattaforma *PDP-11*
- Dal 1972 in poi venne comunemente denominato *linguaggio C*
- Ancora oggi il linguaggio C risulta il più usato al mondo (Gli stessi sistemi operativi Linux e Windows sono scritti in C)

Peculiarità

- Dimensioni del codice e dell'eseguibile ridotte: il codice è un normale file di testo di tipo ASCII e l'eseguibile è costituito da un insieme di istruzioni macchina
- Efficienza: la gestione della memoria avviene a basso livello
- Portabilità: è possibile compilarlo per la maggior parte delle macchine oggi in commercio
- Linguaggio di alto livello: si basa su primitive intuitive
- Può integrare codice assembly
- Segue il paradigma di programmazione *imperativo (procedurale)*

Formato di un programma in C

```
<direttive al preprocessore>
<dichiarazioni>
<intestazione della funzione main>
{
    <dichiarazioni>
    <istruzioni>
}
```

Il programma minimo

```
#include <stdio.h> // Direttiva al preprocessore
int main(void)    // intestazione funzione main
{
    printf("Hello World!\n"); // istruzioni
    return(0);
}
```

Dettagli

- *Direttiva del preprocessore*: comando che permette di richiamare le librerie standard del C. (nell'esempio *stdio.h*)
- *Funzione main*: è la funzione base di ogni programma in C, è indispensabile ed unica e deve esserci sempre.
- *Le parentesi graffe* servono, invece, per delimitare le istruzioni, o come vengono abitualmente chiamate "statement", che devono essere eseguite in ordine, da quella più in alto, giù fino all'ultima.
- *Il punto e virgola*, invece, serve per "chiudere" un'istruzione, per far capire che dopo quel simbolo inizia una nuova istruzione.
- *printf()*: stampa a video tutto quello che gli viene passato come argomento; fa parte della libreria *<stdio.h>*

Compilazione

- Il codice sorgente viene controllato dal preprocessore che ha i seguenti compiti:
 - rimuovere eventuali commenti presenti nel sorgente;
 - interpretare speciali direttive denotate da "#", come *#include*.
 - controllare eventuali errori del codice
- Il risultato del preprocessore sarà un nuovo codice sorgente "pulito" ed "espanso" che viene tradotto dal compilatore C in codice assembly;
- L'assembler sarà incaricato di creare il codice oggetto salvandolo in un file (.o sotto Unix/Linux e .obj in Dos/Windows);
- *Il Link editor*, infine, ha il compito di collegare tutti i file oggetto risolvendo eventuali dipendenze e creando il programma (che non è altro che un file eseguibile).

Variabili

- Dichiarazione di variabili: tutte le variabili prima di essere utilizzate devono essere dichiarate
 - tipo della variabile (identificativo standard)
 - nome variabile (identificativo scelto dall'utente)
- Inizializzazione: spesso alla variabile deve essere assegnato un valore iniziale
- Dichiarazione e inizializzazione possono essere fatte nella stessa istruzione

- ES:

```
/* solo dichiarazione */
int x;
/* inizializzazione */
x = 10;
/* dichiarazione ed inizializzazione allo stesso
tempo */
int y = 15;
```

Tipi di variabili (identificativi standard)

Tipo	Rappresentazione	N. di byte
char	Carattere	1 (8 bit)
int	Numero intero	2 (16 bit)
short	Numero intero "corto"	2 (16 bit)
long	Numero intero "lungo"	4 (32 bit)
float	Numero reale	4 (32 bit)
double	Numero reale "lungo"	8 (64 bit)

Operatori

- Servono a combinare le variabili/costanti in operazioni matematiche/logiche
- Diversi tipi di operatori:
 - Matematici
 - Di assegnamento
 - Di confronto
 - Logici

Operatori matematici

Operazione	Simbolo
Addizione	+
Sottrazione	-
Moltiplicazione	*
Divisione	/
Divisione con modulo (solo numeri interi)	%
incremento unitario	++
decremento unitario	--

Utilizzo degli operatori di incremento e decremento

```
x++; /* equivale a scrivere x = x + 1; */
x--; /* equivale a scrivere x = x - 1; */
```

- Possono trovarsi in due posizioni:
 - Post-fissi `x++`
 - Pre-fissi `++x`
- Post-fissi:** il valore viene aggiornato "dopo" la valutazione dell'espressione in cui si trova
- Pre-fissi:** il valore viene aggiornato "prima"

Forma contratta per operatori aritmetici

- espressione1 = espressione1 <operatore> espressione2
/* questa risulta equivalente alla sottostante */
- espressione1 <operatore> = espressione2

```
x = x + 3; /* diventa */
x += 3;
```

```
y = y * 5; /* diventa */
y *= 5;
```

Operatori di confronto

Simbolo	Significato	Utilizzo
<code>==</code>	uguale a	<code>a == b</code>
<code>!=</code>	diverso da	<code>a != b</code>
<code><</code>	minore	<code>a < b</code>
<code>></code>	maggiore	<code>a > b</code>
<code><=</code>	minore o uguale	<code>a <= b</code>
<code>>=</code>	maggiore o uguale	<code>a >= b</code>

Operatori logici

Simbolo	Significato	Utilizzo
<code>&&</code>	AND logico	<code>a && b</code>
<code> </code>	OR logico	<code>a b</code>

Proprietà' degli operatori

- *Posizione*: *prefisso* se compare prima degli operandi, *postfisso* se compare dopo e *infixo* se compare tra gli operandi;
- *Arietà*: numero di argomenti che un operatore può accettare.
- *Precedenza (o priorità)*: stabilisce quali operazioni debbano essere eseguite per prime
- *Associatività*: a parità di priorità stabilisce quale sia l'ordine con cui bisogna eseguire i vari operatori.
 - associativo a sinistra: si scorre da sinistra verso destra
 - associativo a destra si scorre destra verso sinistra

Funzioni standard di input/output <stdio.h>

- `printf()`: stampa sullo standard output (video) la lista di argomenti che le vengono passati conformemente al formato descritto dalla stringa
- `scanf()`: legge dallo standard input (tastiera) la lista di argomenti che le vengono passati nel formato descritto dalla stringa

printf(.....)

- Dichiarazione:
`int printf(char *formato, lista argomenti ...)`
- La *stringa di formato* ha due tipi di argomenti:
 - *caratteri ordinari*: questi vengono copiati nell'output (tutto cio' che vogliamo scrivere come stringa);
 - *specificazioni di conversione*: contraddistinte dal carattere percentuale "%" e da un carattere che specifica il formato con il quale stampare le variabili presenti nella lista di argomenti

Specificatori di conversione

Specificatore	Tipo di dato
<code>%d</code>	Intero (int)
<code>%c</code>	Un carattere (char)
<code>%s</code>	Una stringa di caratteri (*char)
<code>%x</code>	Numero esadecimale
<code>%f</code>	Numero reale (float o double)
<code>%e</code>	Formato scientifico valore in virgola mobile (float)
<code>%g</code>	Valore in virgola fissa o mobile (double)

Sequenze di escape

- Servono a rappresentare quei caratteri "speciali" presenti nella codifica ASCII
- Iniziano sempre con il carattere *backslash* (\)

Sequenza di escape	Descrizione
<code>\n</code>	Ritorno a capo
<code>\t</code>	Tabulazione orizzontale
<code>\b</code>	Tabulazione verticale
<code>\a</code>	Torna indietro di uno spazio
<code>\f</code>	Salto pagina

scanf(.....)

- Dichiarazione:
`int scanf(char *formato, lista argomenti ...)`
- La *stringa di formato* è costituita dai soli *specificatori di conversione*
- Le variabili argomento devono essere precedute dal simbolo `&`, perché la *scanf* necessita dell'indirizzo di memoria della variabile
- L'operatore `&` ritorna l'indirizzo di memoria di una variabile/puntatore

Specificatori di conversione scanf

Specificatore	Tipo di dato
<code>%d</code>	Intero (int)
<code>%c</code>	Un carattere (char)
<code>%s</code>	Una stringa di caratteri (*char)
<code>%f</code>	Numero reale virgola fissa (double)
<code>%le</code>	Numero reale virgola mobile (double)
<code>%lg</code>	Valore in virgola fissa o mobile (double)

Istruzioni di scelta

- In linguaggio C esistono due tipi di strutture di controllo condizionale (istruzioni di scelta):
 - `if-else` : permette di scegliere tra due possibili alternative
 - `switch` : permette di scegliere tra *n* possibili alternative

if - else

- Sintassi:

```
if (espressione)
{
    istruzione1
    istruzione2
}
```

- oppure:

```
if (espressione)
{
    istruzione1
}
else
{
    istruzione2
}
```

if – else in serie

```
if (risultato_esame >=18)
{
    printf ("Complimenti hai superato l'esame");
}
else if (risultato_esame >=15)
{
    printf ("Devi sostenere l'orale per questo esame");
}
else
{
    printf ("Non hai superato l'esame");
}
```

switch

- Sintassi:

```
switch (espressione)
{
    case elem_1:
        istruzione_1;
        break;
    case elem_2:
        istruzione_2;
        break;
    ...
    case elem_n:
        istruzione_n;
        break;
    default:
        istruzione;
        break;
}
```

Strutture iterative

- Le strutture iterative in linguaggio C sono tre:
 - il **while**, che continua il suo ciclo fino a quando l'espressione associata non risulta falsa
 - il **do-while**, che agisce come il **while**, ma assicura l'esecuzione delle istruzioni associate almeno una volta
 - il **for**, che è il costrutto più usato, versatile e potente tra i tre.

while

- La struttura del **while** è la seguente:

```
while (condizione)
{
    istruzione1;
    istruzione2;
}
```

- Le istruzioni all'interno del **while** agiscono sulla condizione modificandola (altrimenti il ciclo non terminerebbe).

do - while

- La struttura è la seguente:

```
do
{
    istruzione1;
    istruzione2;
}while (condizione)
```

- Le istruzioni all'interno del blocco vengono eseguite almeno una volta indipendentemente dalla condizione

for

- La struttura è la seguente:

```
for (inizializzazione ; condizione ; incremento)
{
    istruzione1;
    istruzione2;
}
```

- Le espressioni all'interno del **for** hanno diversi compiti, e sono separate da un *punto e virgola*:
 - La prima viene eseguita una sola volta prima di entrare nel ciclo: inizializza la variabile di controllo del ciclo.
 - La seconda è la condizione (che coinvolge anche la variabile di controllo), che se risulta falsa interrompe l'esecuzione del ciclo.
 - La terza costituisce l'istruzione di incremento della variabile di controllo che viene eseguita dopo ogni ciclo del **for**.

Strutture dati contigue: gli array

- In linguaggio C possiamo dichiarare un array di numeri interi in questo modo:

```
int myarray[10];
```

- tipo di dati
- nome della variabile
- [dimensione]

Inizializzazione degli array

- Caso 1:

```
int numeri[] = {7, 23, 44, 5};
```

- Caso 2:

```
int numeri[100];
int i;

for(i = 0; i < 100; i++)
{
    numeri[i] = 1;
}
/* a tutti gli elementi assegno 1 */
```

Array multi-dimensionali

- Un array *bi-dimensionale* è costituito da un array dove ogni suo elemento è anch'esso, a sua volta, un array
- E' possibile definire array tri-dimensionali, quadri-dimensionali, ecc.

Funzioni

- Le funzioni esistono più o meno in tutti i linguaggi e vengono chiamate anche *procedure* o *subroutine*.
- Alcuni linguaggi fanno distinzione tra funzioni che ritornano un valore e quelle che, invece, non ritornano valori;
- il C assume che ogni funzione ritorni un valore, questo accade utilizzando l'istruzione **return** seguita, eventualmente, da un valore.

Esempio di funzione

```
double elevamento_potenza(double base, int potenza)
{
    double valore_ritorno = 1.0;
    int i;

    for(i=0; i<potenza; i++)
    {
        valore_ritorno *= base;
    }
    return(valore_ritorno);
}
```

Esempio funzione(2)

```
double elevamento_potenza(double base, int potenza)
```

- Questa è la definizione della funzione, dove vengono dichiarati: il tipo del valore di ritorno (**double**), il nome della funzione (**elevamento_potenza**) e la lista degli argomenti. Per ogni argomento della funzione è necessario dichiarare il tipo (**double** e **int** nell'esempio) ed il nome della variabile (nell'esempio **base** e **potenza**) corrispondente.
- Esistono anche funzioni che non ritornano alcun valore, in questo caso si parla di funzioni che ritornano **void**. Una funzione void non deve avere necessariamente un'istruzione "return" anche se è buona regola terminare la funzione con un'istruzione di return senza alcun parametro.

Prototipi di funzione

- Va fatta una piccola nota riguardante la prototipazione delle funzioni, ovvero la creazione di prototipi.
- Un prototipo di una funzione non è altro che la sua dichiarazione, senza specificare il corpo della funzione stessa (**implementazione**);
- si scrive, quindi, solo la dichiarazione iniziale comprendente il nome ed il tipo restituito dalla funzione, e gli argomenti passati come parametro;
- questo avviene perché ogni funzione è utilizzabile solamente quanto è stata dichiarata, quindi se in un pezzo di codice, prima di dichiarare la funzione, voglio usare tale funzione, non posso farlo, perché "fisicamente" non è ancora stata creata

Esempio prototipo

```
#include <stdio.h>

void stampa_doppio(int variabile); /* dichiarazione del prototipo */

int main(void)
{
    int var = 5;
    stampa_doppio(var);
    return(0);
}

void stampa_doppio(int variabile) /* implementazione funzione */
{
    printf("Il doppio di %d è %d ", variabile, 2*variabile);
    return;
}
```

Puntatori

- I puntatori sono uno strumento molto potente nella programmazione in C. I puntatori permettono di lavorare "a basso livello" (cioè agendo su singole istruzioni del processore), mantenendo però una praticità unica nel suo genere.
- Un puntatore non è altro che **una variabile che contiene l'indirizzo di memoria di un'altra variabile**. Quando dichiariamo una variabile, a questa verrà riservato un indirizzo di memoria, ad esempio la posizione 1000; un puntatore a questa variabile contiene, appunto, l'indirizzo di memoria di tale variabile (quindi il valore 1000). L'importanza risiede nel fatto che si possono manipolare sia il puntatore che la variabile puntata (cioè la variabile memorizzata a quell'indirizzo di memoria).
- Per definire un puntatore è necessario seguire la seguente sintassi:

```
/* variabile normale */
int variabile;

/* puntatore */
int *puntatore;
```

Puntatori(2)

- L'asterisco * viene chiamato **operatore di indirezione o deferenziamento** e restituisce il contenuto dell'oggetto puntato dal puntatore; mentre l'operatore & (già visto nell'utilizzo della scanf) **restituisce l'indirizzo di memoria della variabile** e viene normalmente usato per assegnare ad un puntatore l'indirizzo di una variabile nel seguente modo:

```
/* assegno al puntatore l'indirizzo di variabile */
puntatore = &variabile;
```

Da notare che un puntatore deve essere dello stesso tipo della variabile puntata. Ad esempio un puntatore ad un variabile di tipo double dovrà essere dichiarato come:

```
double *puntatore;
```

Puntatori e array

- L'aspetto che accomuna i puntatori e gli array è sostanzialmente il fatto che entrambi sono memorizzati in locazioni di memoria continue, ed è quindi possibile agire su un array (se lo si vede come una serie di blocchi ad indirizzi di memoria sequenziali) come se si stesse agendo su un puntatore (e viceversa); ad esempio se dichiariamo un array "alfa" ed un puntatore "pointer":

```
/* dichiaro un array ed una variabile intera */
int alfa[20];
int x;

/* dichiaro un puntatore a intero */
int *pointer;

/* assegno al puntatore l'indirizzo di memoria della prima
cella dell'array cioè alfa[0] */
pointer = &alfa[0];

/* metto tutti gli elementi dell'array uguali a 0 */
for(i = 0; i < 10; i++)
{
    *(pointer + i) = 0;
}
```

La funzione malloc()

- La funzione malloc() permette di allocare della memoria durante l'esecuzione di un programma. La sintassi della funzione malloc() è la seguente:

```
*void malloc(int size)
```

- In particolare tale funzione vuole come argomento un parametro di tipo **int** che rappresenta la **dimensione in byte dello spazio di memoria** contiguo da allocare e ritorna un **puntatore di tipo void** alla prima posizione della memoria appena allocata.
- Nel caso in cui l'allocazione non vada a buon fine (la memoria sulla macchina non è sufficiente ad allocare lo spazio richiesto) la funzione malloc ritorna un **puntatore nullo (NULL)** indicando un errore.

La funzione sizeof()

- La funzione **sizeof()** permette di determinare la dimensione di un tipo di dato rappresentato sull'architettura corrente.
- Tale funzione vuole come argomento **un tipo di dato** e ritorna **un valore intero** che rappresenta il numero di byte con cui tale tipo di dato viene rappresentato sull'architettura corrente.
- Quindi su un processore della famiglia x86 l'istruzione **sizeof(double)**, ad esempio, ritornerà il valore intero 8 che corrisponde ad una rappresentazione a 8 byte = 64 bit per le variabili di tipo double

La funzione free()

- La funzione **free()** permette di liberare una zona di memoria preventivamente allocata, ad esempio con una chiamata malloc().
- La funzione **free()** vuole come argomento il puntatore alla prima locazione di memoria da liberare.

Esempio di allocazione dinamica

```
int *array; /* dichiaro un puntatore ad intero che mi
            rappresenterà un array */

array = (int *)malloc(sizeof(int) * 10);
/* questa istruzione alloca memoria necessaria a contenere
10 elementi di tipo intero */

if(array == NULL) /* controlla se c'era memoria sufficiente
                  per allocare l'array */
{
    printf("Memoria esaurita\n");
    exit(1);
}

free(array); /* libero la memoria associata all'array quando
              non serve più */
```

La funzione fopen()

- I file sono la parte più importante degli stream perché, come già detto, sono un elemento essenziale per permettere al programmatore di fare applicazioni interattive. Come menzionato prima, la prima cosa da fare è aprire uno stream su file. Per fare ciò si usa la funzione fopen(), la cui sintassi è la seguente:

```
FILE *fopen(char *nome, char *modo);
```

- Com'è possibile notare, la funzione **fopen()** prevede due argomenti di tipo **puntatore a carattere** (cioè una stringa di testo).
- Il primo argomento identifica il **nome del file** che vogliamo aprire mentre il secondo argomento indica la modalità di apertura del file e può assumere i seguenti valori:
 - "r" - lettura;
 - "w" - scrittura;
 - "a" - scrittura in fondo al file (append).

Note su apertura file

- La funzione **fopen()** restituisce un puntatore al tipo di dato descritto dall'identificativo privato **FILE**.
- Il puntatore restituito **servirà, dopo l'apertura, per poter effettuare** le operazioni di lettura e scrittura sul file stesso.
- Nel caso in cui si verificano problemi nell'apertura del file **viene restituito un puntatore nullo (NULL)**.

Le funzioni fscanf() e fprintf()

- Una volta che si è aperto un file con la funzione **fopen()**, possiamo usare due funzioni per accedervi, queste sono la **fprintf()** e la **fscanf()** che operano come la **printf()** e la **scanf()** con la sola differenza che agiscono su uno stream aperto da **fopen()**. La sintassi delle due funzioni è la seguente:

```
int fprintf(FILE *f_pointer, char *formato, argomenti ...);
int fscanf(FILE *f_pointer, char *formato, argomenti ...);
```

- Come si può intuire la **fprintf()** scrive sul file mentre la **fscanf()** legge.
- L'unica differenza rispetto alla printf() e scanf() che abbiamo già visto sta nel fatto che come primo argomento prevedono **gli venga passato un puntatore a file**.

Le funzioni fflush() e fclose()

- Infine gli stream, qualunque uso ne sia stato fatto, devono essere prima **"puliti"** e poi **chiusi**, utilizzando le funzione **fflush()** che svuota il buffer dello stream pulendolo e **fclose()** che chiude lo stream.
- La sintassi delle funzioni è la seguente:

```
fflush(FILE *f_pointer);
fclose(FILE *f_pointer);
```

Errori comuni e regole di stile in C

- Errori comuni per chi è ai primi approcci con il linguaggio C possono essere così riassunti:
- **Assegnazione (=)** al posto del **confronto (==)**. Bisogna porre attenzione, quando, utilizzando una struttura condizionale come if-else, scriviamo l'operazione di confronto (==), poiché essa può, per un errore di battitura, diventare un assegnamento (=);
- **Mancanza di ()** per una funzione - Il programmatore inesperto tende a credere che una funzione alla quale non si passano parametri (void) non debba avere le parentesi tonde; ciò è errato, le parentesi tonde, anche senza parametri, devono essere sempre messe.
- **Indici di Array** - Quando si inizializzano o si usano gli array, bisogna porre attenzione agli indici utilizzati (e quindi alla quantità di elementi) poiché se si inizializza un array con N elementi, il suo indice deve avere un intervallo tra 0 (che è il primo elemento) ed N-1 (che è l'n-esimo elemento).
- **Il C è Case Sensitive**: fa distinzione tra lettere maiuscole e minuscole, interpretandole come due caratteri diversi; bisogna quindi stare attenti ad utilizzare i nomi giusti per le variabili.
- Il ";" chiude ogni istruzione - E' un errore tanto comune che non può non essere citato, ogni istruzione deve essere chiusa con un punto e virgola; questa facile dimenticanza (Nda: che colpisce anche i più esperti ;), segnalata dal compilatore, può far perdere del tempo prezioso ed appesantisce inutilmente il lavoro del programmatore.

Buone regole di programmazione

- Utilizzando poche regole di buona programmazione possiamo scrivere del codice facilmente leggibile da altri programmatori e più facilmente correggibile dal docente. Ecco un elenco di alcune di queste regole:
- **I nomi delle variabili**, strutture, costanti e funzioni devono essere significativi e normalmente scritti con caratteri minuscoli. Inoltre sarebbe buona norma, se il nome è composto da più parole, evidenziare le due parole separandole da un carattere di underscore "_".
- **Le costanti**, invece, è buona regola che siano scritte tutte in maiuscolo e separate, se formate da più di una parola, dal carattere underscore "_".
- **Uso, funzione e posizione dei commenti** - I commenti devono accompagnare quasi tutte le istruzioni, per spiegare il significato di ciò che stiamo facendo, inoltre devono essere sintetici e devono essere aggiornati appena modifichiamo un'istruzione. I commenti devono essere più estesi, invece, se devono spiegare un determinato algoritmo o se accompagnano una funzione.
- **Ogni sorgente** dovrebbe contenere, nell'ordine:
 - **Commento iniziale** (con nome del file, nome dell'autore, data, testo del problema ed eventuali algoritmi usati);
 - **Istruzioni #include** per tutti i file da includere (bisogna includere solo i file necessari);
 - **Dichiarazioni** di costanti, strutture e tipi enumerativi;
 - **Definizione** di variabili;
 - **Prototipi** delle funzioni;
 - **Definizione delle funzioni** (con la funzione main messa come prima funzione).