Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

Funzione `printf(...)`

Definita come

`int printf(char *format, arg list ...)`

`format` è una stringa costante che può contenere :

- whitespaces: spazi,tabulazioni, invii a capo (\n)
- caratteri di testo standard (qualunque sequenza di caratteri non bianchi che non inizi per `%`)
- specificatori di formato (che iniziano per `%`)

Ritorna il numero di caratteri stampati. Se qualcosa non va, restituisce un numero negativo.

Formato (%)	Tipo	Risultato
c	char	singolo carattere
i,d	int	numero decimale
o	int	numero ottale
x,X	int	numero esadecimale
u	int	intero senza segno
s	char *	stampa una stringa
f	double/float	formato -m.ddd...
e,E	double/float	formato scientifico (e.g. -1.34e003)
p	*	valore di un puntatore (ossia indirizzo contenuto)
%	-	stampa il carattere %

per maggiori informazioni

https://cplusplus.com/reference/cstdio/printf/

In [13]:
```c
#include <stdio.h>

#include <unistd.h>

int main()
{
    printf("Hello World");
    sleep(2);
    return 0;
}
```

```
Hello World
```


Domanda Perchè la stringa viene stampata dopo i 2 secondi e non prima?

In [9]:
```c
// altro esempio
#include <stdio.h>


int main()
{
    int n, d;
    printf("Hello World\n");
    printf("numerator:");
    scanf("%d", &n);
    printf("denumerator:");
    scanf("%d", &d);
    printf("result:");
    int q = d/n;
    printf("%d\n", q);

    return 0;
}
```

```
Hello World
numerator:

denumerator:

[C kernel] Executable exited with code -8
```

Domanda

Perchè la stringa `result:` non viene stampata in caso di errore?

Risposta

l'I/O in C è *buffered*. Il dato viene conservato in un'area di memoria, e viene letto/stampato solo se si verificano determinate condizioni.

Questo per evitare un eccesso di chiamate di sistema, pesanti in termini computazionali.

In sostanza, invece di leggere/scrivere tante volte dati, si preferisce leggerli/scriverli tutti in una volta in un'unica chiamata interna.

la stringa `result:` rimane nel buffer in caso di errore.

Per l'output, C permette di utilizzare 3 tipi di strategia:

- *unbuffered*: l'output viene generato immediatamente, senza memorizzare i caratteri in nessun area temporanea
- *block buffered*: i caratteri sono conservati in un'area di memoria temporanea (buffer). Quando il buffer si riempie, il suo contenuto viene scritto in massa sul dispositivo di output (esempio, un file di testo). Tale buffer è svuotato anche in caso di una invocazione di `fflush(...)` o quando il dispositivo di output viene chiuso.
- *line buffered*: come block buffered, ma con in più lo svuotamento del bufferanche in caso di presenza del carattere newline `\n`.

Di default, in C l'output su file è block buffered, sullo standard output (i.e. lo schermo) è invece line buffered. Lo `stderr` è invece unbuffered.

Even though it is not mandatory in standards, functions which read from stdin can flush to the stdout file first.

In [ ]:
```c
#include <stdio.h>


int main()
{
    int n, d;
    printf("Hello World\n");
    printf("numerator:");
    scanf("%d", &n);
    printf("denumerator:");
    scanf("%d", &d);
    printf("result:\n"); // <--- il buffer viene svuotato quando trova un carattere \r
    int q = d/n;
    printf("%d\n", q);

    return 0;
}
```

In C, è possibile "svuotare" (*flush*) qualsiasi stream di output. L'effetto è che qualsiasi dato contenuto viene effettivamente inserito nel relativo stream. la funzione di riferimento è `fflush(...)` ed è definita nell'header `stdio.h`

From linux manual page:

> Output streams that refer to terminal devices are always line buffered by default; pending output to such streams is written automatically whenever an input stream that refers to a terminal device is read.

Domanda: se la funzione `printf` è line buffered, perchè nel caso precedente le stringhe `numerator:` e `denumerator:` vengono stampate subito nonostante la mancanza dello `\n`?

Risposta: anche se non obbligatorio negli standard, è prassi comune che funzioni che leggano dallo standard input "svuotino" in automatico lo stdout prima di essere eseguite.

In [10]:
```c
#include <stdio.h>


int main()
{
    int n, d;
    printf("Hello World\n");
    printf("numerator:");
    scanf("%d", &n);
    printf("denumerator:");
    scanf("%d", &d);
    printf("result:");
    fflush(stdout); // alternativa: fflush svuota i buffer di output
    int q = d/n;
    printf("%d\n", q);

    return 0;
}
```

```
Hello World
numerator:

denumerator:

result:
[C kernel] Executable exited with code -8
```

Funzione `scanf(...)`

```
int scanf ( const char * format, ... );
```


da https://cplusplus.com/reference/cstdio/scanf/ :

> - in format, the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters).
> - Non-whitespace character, except format specifier (%): Any character that is not either a whitespace character **(blank, newline or tab)** or part of a format specifier (which begin with a % character) causes the function to read the next character from the stream, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of format. If the character does not match, the function fails, returning and leaving subsequent characters of the stream unread.
> - Format specifiers: A sequence formed by an initial percentage sign (%) indicates a format specifier, which is used to specify the type and format of the data to be retrieved from the stream and stored into the locations pointed by the additional arguments.

> On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the end-of-file.

In [6]:
```c
#include<stdio.h>

int main(){
    int a,b;

    printf("inserisci interi:");
    scanf("%d%d",&a,&b);
    printf("%d %d", a, b);
    return 0;
}
```

```
inserisci interi:

1 5
```

```c
#include<stdio.h>

int main(){
    int a,b;

    printf("inserisci intero:");
    scanf("%d\n",&a);

    printf("inserisci intero:");
    scanf("%d\n",&b);

    printf("%d %d\n",a, b);
    return 0;
}
```

```
inserisci intero:1



2
inserisci intero:3
1 2
```

Nella `scanf(...)` far seguire uno spazio ad uno specificatore di formato può creare problemi. Questo perchè la `scanf(...)`, quando incontra uno spazio nella sua stringa di formato, non termina finché non viene inserito un carattere diverso da `space` o `newline`. E' necessario quindi inserire un utelriore carattere (e.g., `2`) per far terminare la `scanf(...)`. Il carattere `2` inserito per far terminare la `scanf("%d\n",&a)` non viene però "consumato", rimanendo nel buffer. La seconda chiamata `scanf("%d ",&b)` "consumerà" il carattere `2`, ponendolo nella variabile `b` (al posto del carattere `3`, che rimane nel buffer, e viene utilizzato solo per far terminare la `scanf("%d\n",&b)`, in quanto anch'essa contiene uno spazio dopo lo specificatore).

A whitespace (blank, newline, tab) character in a scanf format causes it to explicitly read and ignore as many whitespace characters as it can. So with `scanf("%d ", ...)`, after reading a number, it will continue to read characters, discarding all whitespace until it sees a non-whitespace character on the input. That non-whitespace character will be left as the next character to be read by an input function.

```c
#include<stdio.h>

int main(){
    int a;

    char car;

    printf("inserisci intero:");
    scanf("%d", &a);
    printf("inserisci carattere:");
    scanf("%c", &car);
    printf("Hai inserito il carattere %c", car);
    return 0;
}
```

Problema: la `scanf("%d",&a);` acquisisce e mette nella variabile `a` il valore inserito da tastiera.

La conferma del valore viene data con il tasto <Enter> che corrisponde ad un carattere di `\n`

Tale valore però **non viene elaborato** rimanendo nel *buffer di tastiera* (che ricordo essere un'area di memoria dedicata ai caratteri acquisiti da tastiera), aspettando la prossima acquisizione.

La successiva acqusizione è `scanf("%c", &car);`, ossia l'acqusizione di un carattere (`%c`). Tale acquisizione troverà il precedente carattere `\n`, considerandolo come carattere da mettere nella variabile `car`, e quindi impedendo all'utente di fare l'acqusizione effettiva.

Possibile soluzione:

- fare un'acquisizione a vuoto per consumare il carattere rimanente
  - `getchar();` oppure `getch(stdin);` nel caso di un solo carattere nel buffer

più in generale:


```c
#include <stdio.h>
void clean_stdin(void)
{
    int c;
    do
    {
        c = getchar();
    }
    while (c != '\n' && c != EOF);
}
```

Esempio di NON soluzione

- In alcuni casi, viene riportata come possibile soluzione l'utilizzo della fuzione `fflush(stdin);`. Tuttavia, tale soluzione potrebbe avere comportamento indefinito in alcune implementazioni/versioni delle librerie che la contengono.
- Questo perchè la funzione `fflush(...)` non nasce per lavorare su buffer di input (come la tastiera), ma su buffer di output --> soluzione **non valida** in quanto non standard.

```c
#include<stdio.h>

int main(){
    int a;
    int b;

    printf("inserisci a e b:");
    scanf("%d%d",&a, &b);
    return 0;
}
```