Input e Output

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

Operazioni di I/O

- Le operazioni di I/O sono fondamentali in qualsiasi programma
  - Interazione con l'utente
  - Salvataggio su file
  - Caricamento di dati da file

- Finora ci siamo limitati ad utilizzare printf() per stampare su schermo
- La libreria standard (con l'header stdio.h) fornisce tutte le primitive necessarie ad interagire con la consolle e con i file su disco

Il tipo FILE *

- FILE * è un tipo incompleto che fornisce una definizione opaca di una struttura dati che permette di operare su file
- La filosofia Unix è "*everything is a file*":
  - Un file è un file regolare su disco
  - Un file è un canale di input
  - Un file è un canale di output su un terminale
  - Un file è una periferica

- Esistono dei *descrittori di file* predefiniti, che vengono aperti automaticamente al caricamento del programma:
  - **stdin**: lo standard input (i byte che arrivano dalla tastiera)
  - **stdout**: lo standard output (il terminale)
  - **stderr**: ancora il terminale, ma è un canale "logico" diverso per mostrare stampe di errore

scanf()

- La funzione scanf():
  - legge dei caratteri dal canale standard di input
  - li converte secondo le specifiche di formattazione fornite dalla stringa di formato
  - memorizza i valori ottenuti negli argomenti seguenti (puntatori)

- La sintassi della scanf() ricorda molto quella della printf()
  - Gli specificatori di formato sono gli stessi
  - La semantica degli specificatori non è perfettamente la stessa

- Utilizzare la scanf() è complesso e può portare a molti undefined behavior

Alcuni pericoli della scanf()

- Se il numero di specificatori di conversione nel formato supera il numero di argomenti puntatore, vi è undefined behavior
- Se il numero di argomenti puntatore supera il numero di specifiche di conversione, gli argomenti puntatore in eccesso vengono valutati, ma vengono altrimenti ignorati.
- Per la scanf() una stringa finisce dove si incontra uno spazio
- Se non si ottiene una corrispondenza con lo specificatore di formato, i caratteri vengono lasciati sullo stdin (potenzialmente causando problemi in chiamate successive)

Esempio 1: voglio leggere un numero

```c
#include <stdio.h>

int main(void)
{
    int a;
    printf("enter a number: ");
    scanf("%d", &a);
    printf("You entered %d.\n", a);
}
```

- Eseguitelo inserendo come input: 42
- Eseguitelo inserendo come input: abcdefgh

Esempio 2: voglio leggere un numero

```c
#include <stdio.h>

int main(void)
{
    int a;
    printf("enter a number: ");
    while (scanf("%d", &a) != 1) {
        // input was not a number, ask again:
        printf("enter a number: ");
    }
    printf("You entered %d.\n", a);
}
```

- Eseguitelo inserendo come input: abc

Esempio 3: voglio leggere un numero

```c
#include <stdio.h>

int main(void)
{
    int a;
    printf("enter a number: ");
    while (scanf("%d", &a) != 1) {
        // input was not a number, ask again:
        fflush(stdin); // <- never ever do that!
        printf("enter a number: ");
    }
    printf("You entered %d.\n", a);
}
```

- Eseguitelo inserendo come input: abc

Esempio 4: voglio leggere una stringa

```c
#include <stdio.h>

int main(void)
{
    char name[12];
    printf("What's your name? ");
    scanf("%s", name);
    printf("Hello %s!\n", name);
}
```

- Eseguitelo inserendo come input: Paolo
- Eseguitelo inserendo come input: Bruno Liegi Bastonliegi
- Eseguitelo inserendo come input: Bruno-Liegi-Bastonliegi

Esempio 5: voglio leggere una stringa

```c
#include <stdio.h>

int main(void)
{
    char name[40];
    printf("What's your name? ");
    scanf("%39[^\n]", name);
    printf("Hello %s!\n", name);
}
```

- Eseguitelo con input: Bruno Liegi Bastonliegi
- Eseguitelo schiacciando solo return (eventualmente con valgrind)

Esempio 6: voglio leggere una stringa

```c
#include <stdio.h>

int main(void)
{
    char name[40];
    printf("What's your name? ");
    scanf(" %39[^\n]", name);
    //     ^ note the space here, matching any whitespace
    printf("Hello %s!\n", name);
}
```

- Eseguitelo schiacciando solo return
- Eseguitelo con input vuoto (CTRL+D su Linux, eventualmente con valgrind)

Esempio 7: ok, voglio leggere *qualcosa*

```c
#include <stdio.h>

int main(void)
{
    char name[40];
    printf("What's your name? ");
    if (fgets(name, 40, stdin)) {
        printf("Hello %s!\n", name);
    }
}
```

- Eseguitelo con qualsiasi input

Esempio 8: ci siamo!

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

int main(void)
{
    char name[40];
    printf("What's your name? ");
    if (fgets(name, 40, stdin)) {
        name[strcspn(name, "\n")] = 0;
        printf("Hello %s!\n", name);
    }
}
```

- Eseguitelo con qualsiasi input

Esempio 9: come faccio a leggere un numero?

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

int main(void)
{
    int a;
     char buf[1024]; // use 1KiB just to be sure

    do {
        printf("enter a number: ");
        if (!fgets(buf, 1024, stdin)) {
            // reading input failed, give up:
            return 1;
        }

        // have some input, convert it to integer:
        a = atoi(buf);
    } while (a == 0); // repeat until we got a valid number

    printf("You entered %d.\n", a);
}
```

Esempio 10: come faccio a leggere un numero?

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

int main(void)
{
    long a;
    char buf[1024]; // use 1KiB just to be sure
    int success; // flag for successful conversion

    do {
        printf("enter a number: ");
        if (!fgets(buf, 1024, stdin)) {
            // reading input failed:
            return 1;
        }
```

Esempio 10: come faccio a leggere un numero?

```
        // have some input, convert it to integer:
        char *endptr;

        errno = 0; // reset error number
        a = strtol(buf, &endptr, 10);
        if (errno == ERANGE) {
            printf("Sorry, this number is too small or too large.\n");
            success = 0;
        } else if (endptr == buf) {
            // no character was read
            success = 0;
        } else if (*endptr && *endptr != '\n') {
            // *endptr is neither end of string nor newline,
            // so we didn't convert the *whole* input
            success = 0;
        } else {
            success = 1;
        }
    } while (!success); // repeat until we got a valid number

    printf("You entered %ld.\n", a);
}
```

Le regole di uso della scanf()

- Regola 1: scanf() non serve a leggere input, ma ad effettuarne un'interpretazione.
- Regola 2: scanf() può essere pericolosa se utilizzata in maniera incauta. Utilizzata con le stringhe ed una lunghezza di buffer fissa è il suo uso migliore.
- Regola 3: Anche se le stringhe di formato della scanf() assomigliano molto a quelle della printf(), hanno una semantica leggermente differente (leggete il manuale!)
- Regola 4: Anche con stringhe a dimensione fissa, è possibile che vengano lasciati dei caratteri su stdin. Se questo è un problema, conviene utilizzare fgets().

Operazioni su filesystem

- fopen: apre un file, creando un descrittore corrispondente
- fclose: chiude un descrittore di file aperto
- fseek: imposta l'indicatore di posizione nel file
- ftell: restituisce il valore corrende dell'indicatore di posizione
- rewind: equivalente a fseek(stream, 0L, SEEK_SET)
- fwrite: scrive un buffer su file, a partire dall'indicatore di posizione corrente
- fread: legge dati in un buffer da un file, a partire dall'indicatore di posizione corrente

Un esempio

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

#define SIZE 1
#define NUMELEM 5

int main(void)
{
        FILE *fd = NULL;
        char buff[100];
        memset(buff, 0, sizeof(buff));

        if ((fd = fopen("test.txt", "rw+")) == NULL) {
                printf("fopen() error!\n");
                return 1;
        }

        if (fread(buff, SIZE, NUMELEM, fd) != SIZE * NUMELEM) {
                printf("\n fread() failed\n");
                return 1;
        }
        printf("Read [%s] from file\n", buff);
```

Un esempio

```
        if (fseek(fd, 11, SEEK_CUR) != 0) {
                printf("\n fseek() failed\n");
                return 1;
        }

        if (fwrite(buff, SIZE, strlen(buff), fd) != SIZE * NUMELEM) {
                printf("\n fwrite() failed\n");
                return 1;
        }

        fclose(fd);

        return 0;
}
```

gets()

SYNOPSIS

```
#include <stdio.h>
char *gets(char *s);
```

DESCRIPTION

Never use this function.

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

CONFORMING TO

LSB deprecates gets(). POSIX.1-2008 marks gets() obsolescent. ISO C11 removes the specification of gets() from the C language, and since version 2.16, glibc header files don't expose the function declaration if the _ISOC11_SOURCE feature test macro is defined.

BUGS

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.