Programmazione 1

Lezione 7

Dipartimento di Matematica *Federigo Enriques*
Università degli Studi di Milano

Passaggio di parametri: semantica

Abbiamo visto che un programma in C tipico è costituito da più funzioni che si richiamano a vicenda. Le funzioni si scambiano informazioni tramite il passaggio di parametri e la restituzione di valori.

Vediamo ora un poco più in dettaglio con un esempio la semantica del passaggio dei parametri in C. Anticipando le conclusioni, il punto fondamentale da mettere a fuoco sarà questo:

Passaggio di parametri: semantica

Abbiamo visto che un programma in C tipico è costituito da più funzioni che si richiamano a vicenda. Le funzioni si scambiano informazioni tramite il passaggio di parametri e la restituzione di valori.

Vediamo ora un poco più in dettaglio con un esempio la semantica del passaggio dei parametri in C. Anticipando le conclusioni, il punto fondamentale da mettere a fuoco sarà questo:

Il passaggio dei parametri in C

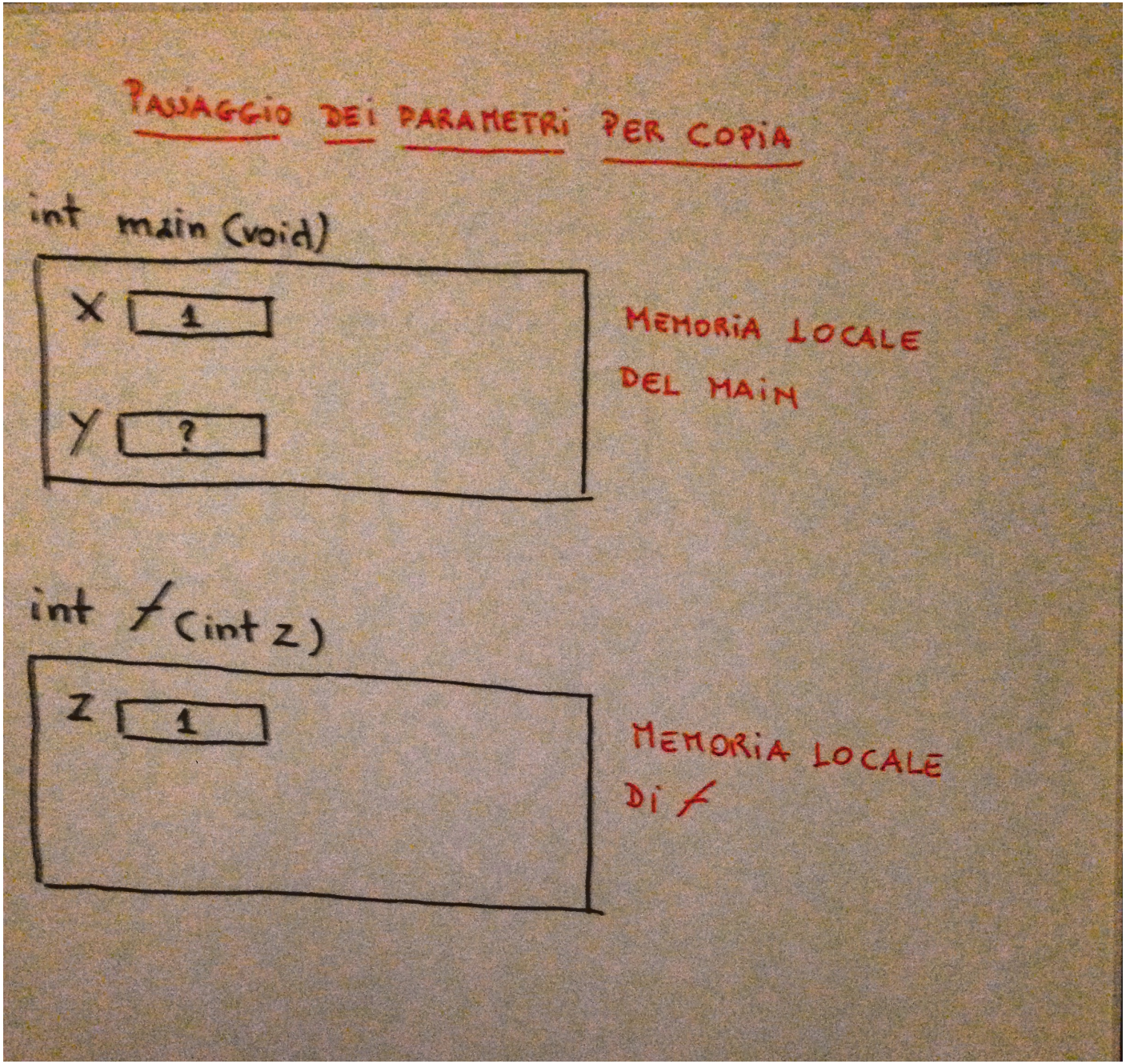
Nel linguaggio C, il linguaggio dei parametri fra procedure avviene esclusivamente per copia: ossia, la funzione chiamata riceve una copia, memorizzata nel suo spazio di memoria locale, del dato passato dalla funzione chiamante.

param.c

```c
int main(void) {
    int f(int);    //accetta un intero, restituisce un intero
    /* Variabili locali allocate nella
    memoria del main */
    int x=1, y;

    y=f(x);        //chiamata ad f
    /* In questo punto y vale 0, x vale 1 */
    return -1;
}
int f(int z) {
    /* Il parametro formale z potrebbe anche chiamarsi x senza che cio'
    causi conflitti con la x del main. Alla chimamara di f, la variabile
    locale z e' allocata nella memoria locale di f (disgiunta da quella
    del main e di ogni altra procedura). In z, cioè nelle celle di
    memoria allocate per z, e' COPIATO il valore del parametro passato
    dal chiamante; in questo esempio, 1. */
    z=-1;         //Il valore 1 è sovrascritto da -1

    return 0;     /* E' restituito il valore 0, e lo spazio di memoria
                  locale di f è deallocato: la variabile locale z non
                  esiste più. */
}
```


PASSAGGIO DEI PARAMETRI PER COPIA
int main (void)
X 1
Y ?
MEMORIA LOCALE DEL MAIN
int f (int z)
z 1
MEMORIA LOCALE DI f

Passaggio di parametri e puntatori

Abbiamo visto che i puntatori sono variabili — con un ruolo specifico, ma pur sempre variabili. Possono quindi essere passati come parametri alle funzioni.

Passaggio di parametri e puntatori

Abbiamo visto che i puntatori sono variabili — con un ruolo specifico, ma pur sempre variabili. Possono quindi essere passati come parametri alle funzioni.

Date le dichiarazioni e l'assegnazione:

```
int *ptr;
int i=5;
ptr=&i;
```

consideriamo la funzione di prototipo:

```
void f(int *p);
```

Passaggio di parametri e puntatori

Abbiamo visto che i puntatori sono variabili — con un ruolo specifico, ma pur sempre variabili. Possono quindi essere passati come parametri alle funzioni.

Date le dichiarazioni e l'assegnazione:

```
int *ptr;
int i=5;
ptr=&i;
```

consideriamo la funzione di prototipo:

```
void f(int *p);
```

Allora l'invocazione

```
f(ptr);
```

è sintatticamente corretta. Qual è la sua semantica?

Abbiamo già visto:

Passaggio dei parametri

L'unica modalità di passaggio dei parametri alle funzioni contemplata dal linguaggio C è quella per copia.

Abbiamo già visto:

Passaggio dei parametri

L'unica modalità di passaggio dei parametri alle funzioni contemplata dal linguaggio C è quella per copia.

Applichiamo la semantica per copia al nostro esempio. Dopo la chiamata

```
f(ptr);
```

la funzione `f` ha a sua disposizione (nella memoria locale ad essa riservata) una *copia locale* del valore della variabile originaria `ptr`.

Abbiamo già visto:

Passaggio dei parametri

L'unica modalità di passaggio dei parametri alle funzioni contemplata dal linguaggio C è quella per copia.

Applichiamo la semantica per copia al nostro esempio. Dopo la chiamata

```
f(ptr);
```

la funzione `f` ha a sua disposizione (nella memoria locale ad essa riservata) una *copia locale* del valore della variabile originaria `ptr`.

Ma il valore di `ptr` è un indirizzo della memoria locale riservata alla funzione chiamante. Quindi, la funzione `f` *ha accesso* tramite la sua variabile locale `p` alla memoria della funzione chiamante.

Puntatori come parametri delle funzioni

Usare i puntatori come parametri delle funzioni è un modo per condividere memoria fra la funzione chiamante e quella chiamata.

Puntatori come parametri delle funzioni

Usare i puntatori come parametri delle funzioni è un modo per condividere memoria fra la funzione chiamante e quella chiamata.

Nota in particolare che ciò permette di *restituire al chiamante* un risultato della computazione eseguita dalla funzione `f`, nonostante il fatto che la funzione chiamata `f` abbia prototipo

```
void f(int *);
```

e dunque, formalmente, *non restituisca alcun valore*.

Nota pure che avremmo anche potuto passare direttamente alla funzione `f` l'*indirizzo* della variabile intera `i` tramite *referencing*:

```
f(&i);
```

se `i` è di tipo `int`.

par1.c

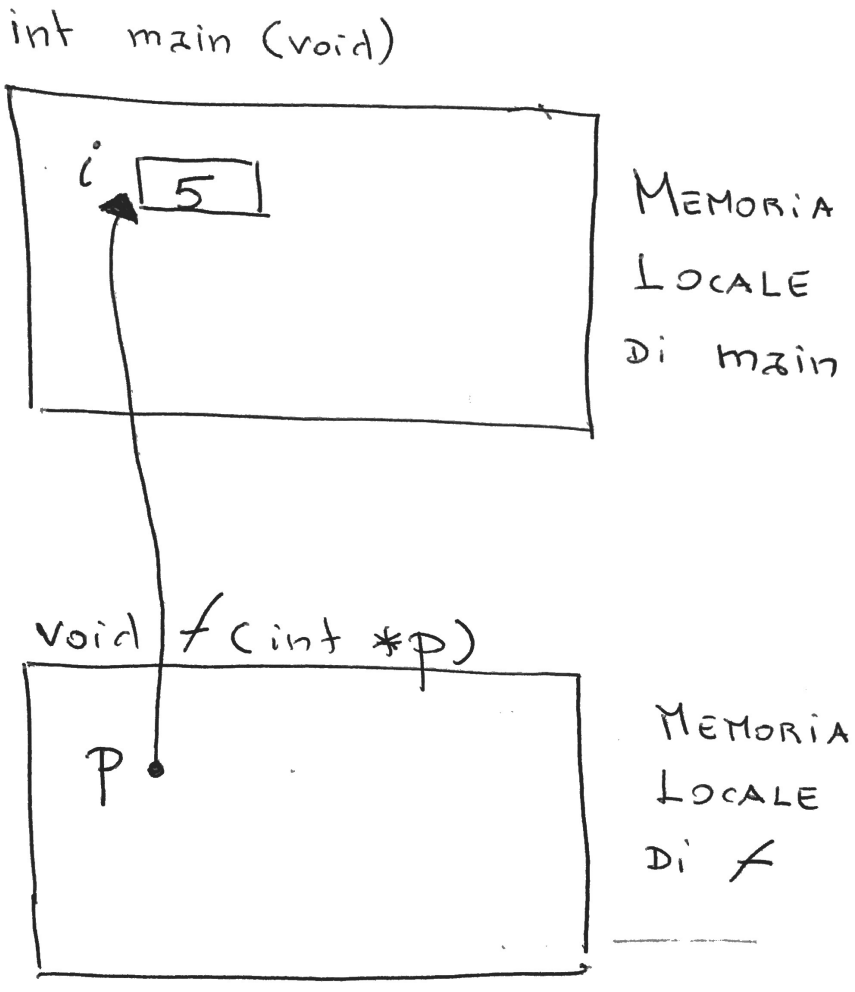
```c
#include <stdio.h>

int main(void)
{
    void f(int *);
    int i=5;

    printf("Prima: i=%d\n", i);
    f(&i); /* Passa ***L'INDIRIZZO*** di i */
    printf("Dopo: i=%d\n", i);
    return 0;
}

void f(int *p)
{
    (*p)++;
    return;
}
```

Compila? Se sì, qual è l'output del programma?


int main (void)
i
5
MEMORIA LOCALE DI main
void f(int *p)
p
MEMORIA LOCALE DI f

par2.c

```c
#include <stdio.h>
int main(void)
{
    void f(int *);
    int i=0;
    int *ptr;
    ptr=&i;
    *ptr=5;
    printf("Prima: i=%d\n",i);
    f(ptr); /* Passa ***UN PUNTATORE*** a i */
    printf("Dopo: i=%d\n", i);
    return 0;
}
void f(int *p)
{
    (*p)++;
    return;
}
```

Compila? Se sì, qual è l'output del programma?

par3.c

```c
#include <stdio.h>
int main(void)
{
    void f(int *);
    int i=5;

    printf("Prima: i=%d\n", i);
    f(&i);
    printf("Dopo: i=%d\n", i);
    return 0;
}

void f(int *p)
{
    p++; /* Questo incremento e' un errore grave,
         dato main come sopra. Perche'? (Segmentation fault) */
    return;
}
```

Compila? Se sì, qual è l'output del programma?

Passaggio di parametri e array

Quando si passa il nome di un array come parametro ad una funzione, ciò che è passato è *l'indirizzo della prima locazione dell'array*.

Passaggio di parametri e array

Quando si passa il nome di un array come parametro ad una funzione, ciò che è passato è *l'indirizzo della prima locazione dell'array*.

Supponiamo di dichiarare l'array:

```
int arr[10];
```

Una funzione dichiarata così:

```
void f(int *p);
```

può essere invocata in questo modo per passarle in ingresso l'array `arr`:

```
f(arr);
```

Poiché, come appena detto, passare `arr` ad una funzione *equivale* a passarle `&arr[0]`, l'invocazione

```
f(arr);
```

è del tutto coerente col prototipo

```
void f(int *p);
```

Poiché, come appena detto, passare `arr` ad una funzione *equivale* a passarle `&arr[0]`, l'invocazione

```
f(arr);
```

è del tutto coerente col prototipo

```
void f(int *p);
```

Si applica la solita semantica del passaggio per copia: ciò che la funzione ottiene è una copia locale dell'indirizzo passato `&arr[0]`, contenuta nella variabile locale `p`.

Poiché, come appena detto, passare `arr` ad una funzione *equivale* a passarle `&arr[0]`, l'invocazione

```
f(arr);
```

è del tutto coerente col prototipo

```
void f(int *p);
```

Si applica la solita semantica del passaggio per copia: ciò che la funzione ottiene è una copia locale dell'indirizzo passato `&arr[0]`, contenuta nella variabile locale `p`.

Poiché la funzione chiamata ha dunque una sua variabile locale che contiene un indirizzo, essa ha a tutti gli effetti un *puntatore* (*del tipo appropriato*) *all'array originale*.

Array come parametri delle funzioni.

Passare un nome di array come parametro ad una funzione equivale a passarle un *puntatore* (del tipo appropriato) *al primo elemento* dell'array.

Array come parametri delle funzioni.

Passare un nome di array come parametro ad una funzione equivale a passarle un *puntatore* (del tipo appropriato) *al primo elemento* dell'array.

Illustreremo la cosa con un esempio. La specifica del problema è questa:

Copia di Stringhe

Date due stringhe `s` e `t`, copiare la stringa `t` sulla stringa `s`.

Memento. Come già detto, in C non esiste un tipo specifico per le stringhe. Una stringa è semplicemente un array di `char` terminato da `'\0'` per convenzione.

strcpy1.c

```c
/* strcpy(char *s, char *t): copia t in s */
#include<stdio.h>

int main(void)
{
    void strcpy1(char*, char*);
    char s[20];
    char t[]="Pinco pallino";

    strcpy1(s,t);
    printf("s: %s\nt: %s\n",&s[0],t); //&s[0] equiv. a s
    return 0;
}

void strcpy1(char *s, char *t)
{
    int i=0;
    while ( (s[i]=t[i]) != '\0')
        i++;
}
```

strcpy1comm.c

```c
/* strcpy(char *s, char *t): copia t in s */
#include<stdio.h>

int main(void)
{
    void strcpy1(char*, char*);
    char s[20];
    char t[]="Pinco pallino";

    strcpy1(s,t);
    printf("s: %s\nt: %s\n",&s[0],t); //&s[0] equiv. a s
    return 0;
}

void strcpy1(char *s, char *t)
{
    int i=0; // questa variabile e' ridondante: eliminiamola
    while ( (s[i]=t[i]) != '\0')
        i++;
}
```

strcpy2.c

```c
/* strcpy(char *s, char *t): copia t in s */
#include<stdio.h>
int main(void)
{
    void strcpy2(char*, char*);
    char s[20];
    char t[]="Pinco pallino";

    strcpy2(s,t);
    printf("s: %s\nt: %s\n",s,t);
    return 0;
}
void strcpy2(char *s, char *t)
{
    while ( (*s=*t) != '\0')
    {
        s++;
        t++;
    }
}
```

strcpy2comm.c

```c
/* strcpy(char *s, char *t): copia t in s */
#include<stdio.h>
int main(void)
{
    void strcpy2(char*, char*);
    char s[20];
    char t[]="Pinco pallino";

    strcpy2(s,t);
    printf("s: %s\nt: %s\n",s,t);
    return 0;
}
void strcpy2(char *s, char *t)
{/* Persino questa versione si puo' rendere piu' concisa,
 al costo di perdere leggibilita' */
    while ( (*s=*t) != '\0')
    {
        s++;
        t++;
    }
}
```

strcpy3.c

```c
/* strcpy(char *s, char *t): copia t in s */
#include<stdio.h>

int main(void)
{
    void strcpy3(char*, char*);
    char s[20];
    char t[]="Pinco pallino";

    strcpy3(s,t);
    printf("s: %s\nt: %s\n",s,t);
    return 0;
}

void strcpy3(char *s, char *t)
{
    //funziona col postincremento, ma non con il preincremento!
    while ( (*s++=*t++) != '\0');
}
```

strcpy4.c

```c
/* strcpy(char *s, char *t): copia t in s */
#include<stdio.h>

int main(void)
{
    void strcpy4(char*, char*);
    char s[20];
    char t[]="Pinco pallino";

    strcpy4(s,t);
    printf("s: %s\nt: %s\n",s,t);
    return 0;
}

void strcpy4(char *s, char *t)
{
    //sfrutta il fatto che '\0'==0: codice non robusto.
    while ( (*s++=*t++) );
}
```

Lunghezza delle stringhe

Scrivere una funzione di prototipo

```
int lungstr(char *)
```

che accetti come parametro una stringa e ne restituisca la lunghezza.

lung1.c

```c
#include <stdio.h>

int main(void)
{
    int lungstr(char *);
    char t[]="Ave";

    printf("La stringa e' lunga %d\n",lungstr(t));
    return 0;
}

int lungstr(char *s)
{
    int i;
    for(i=0; s[i]!='\0'; i++);
    return i;
}
```

Lunghezza delle stringhe

Scrivere una funzione di prototipo

```
int lungstr(char *)
```

che accetti come parametro una stringa e ne restituisca la lunghezza.

Nota. Come si comporta la funzione se il puntatore passato come argomento è stato dichiarato ma non definito? E se esso vale `NULL`?

Lunghezza delle stringhe

Scrivere una funzione di prototipo

```
int lungstr(char *)
```

che accetti come parametro una stringa e ne restituisca la lunghezza.

Nota. Come si comporta la funzione se il puntatore passato come argomento è stato dichiarato ma non definito? E se esso vale NULL?

```
1 char *str; //str e' indeterminato
2 printf("%d", lungstr(str));
```

Lunghezza delle stringhe

Scrivere una funzione di prototipo

```
int lungstr(char *)
```

che accetti come parametro una stringa e ne restituisca la lunghezza.

Nota. Come si comporta la funzione se il puntatore passato come argomento è stato dichiarato ma non definito? E se esso vale NULL?

```
1 /* Altri casi particolari da considerare */
2 char *str1; //str1 e' indeterminato
3 char str2[] = "Ave, Cesare"; //str2 non e' NULL
4 char *str3 = NULL; //str3 e' NULL
5 char str4[3] = {'A', 'v', 'e'}; //str4 non e' \0-terminated
6 printf("%d", lungstr(str1)); //Idem str2, str3, str4
```

lung2.c

```c
#include <stdio.h>

int main(void)
{
    int lungstr(char *);
    char t[]="Ave";

    printf("La stringa e' lunga %d\n",lungstr(t));
    return 0;
}

int lungstr(char *s)
{
    if (s==NULL)
        return -1;

    int i;
    for(i=0; s[i]!='\0'; i++);
    return i;
}
```

Input/Output standard

Il compilatore C assume l'esistenza nel sistema di tre flussi di dati standard:

1. `stdin` – associato d'ufficio ai dati in ingresso provenienti dalla tastiera.
2. `stdout` – associato d'ufficio ai dati in uscita visualizzati sul monitor del calcolatore.
3. `stderr` – associato d'ufficio ai dati in uscita visualizzati sul monitor del calcolatore. (Non ne parleremo in dettaglio.)

Input/Output standard

Il compilatore C assume l'esistenza nel sistema di tre flussi di dati standard:

1. `stdin` – associato d'ufficio ai dati in ingresso provenienti dalla tastiera.
2. `stdout` – associato d'ufficio ai dati in uscita visualizzati sul monitor del calcolatore.
3. `stderr` – associato d'ufficio ai dati in uscita visualizzati sul monitor del calcolatore. (Non ne parleremo in dettaglio.)

Le funzioni della libreria standard per l'I/O che discuteremo in questa lezione, definite nel file di intestazione `stdio.h`, leggono da e scrivono su `stdin` e `stdout`, rispettivamente. Hanno controparti in grado di leggere da e scrivere su flussi di dati diversi, associati ai file, di cui parleremo più avanti.

Lettura di caratteri singoli

- `int getchar(void)`
- Restituisce il successivo carattere del flusso `stdin` come intero (ossia il suo codice ASCII convertito da `char` a `int`), oppure `EOF` se incontra la fine del flusso.
- La costante intera `EOF` è definita in `stdio.h`. Di solito vale -1, ma è necessario usare sempre la forma simbolica per non far dipendere la correttezza del codice dallo specifico valore della costante.
- Il carattere restituito da una chiamata a `getchar` è *rimosso* da `stdin`.
- Nel caso in cui `stdin` non contenga alcun carattere, la chiamata a `getchar` blocca il programma in attesa dell'input dell'utente da tastiera, input che consiste di tutto quanto l'utente digita fino al primo invio (carattere `'\n'`).

Scrittura di caratteri singoli

- `int putchar(int c)`
- Scrive il carattere `c`, convertito in `int` sul flusso `stdout`, e restituisce il carattere scritto oppure `EOF` se incorre in un errore.

Scrittura di caratteri singoli

- `int putchar(int c)`
- Scrive il carattere `c`, convertito in `int` sul flusso `stdout`, e restituisce il carattere scritto oppure `EOF` se incorre in un errore.

Possiamo ora chiarire un punto che avevamo lasciato in sospeso:

`getchar`, `putchar` e `int`

`EOF` è un `int`, non un `char`: è per questo che `getchar` e `putchar` restituiscono `int` e non `char`.

Scrittura di dati formattati

- `int printf (const char *formato, ...)`
- Converte e scrive su `stdout` dati formattati, secondo le specifiche della stringa di controllo `format`. Restituisce il numero di caratteri scritto, o un valore negativo in caso d'errore. se incontra la fine del flusso, o se incorre in un errore prima di aver assegnato una qualsiasi conversione. Altrimenti, restituisce il numero di argomenti convertiti e assegnati con successo.
- Abbiamo già usato a lungo questa funzione. I dettagli completi si trovano alle pp. 260–262 del K&R.

Lettura di dati formattati

- `int scanf (const char *formato, ...)`
- Legge da `stdin` sotto il controllo della stringa `formato`, e assegna i valori convertiti attraverso i successivi argomenti, ognuno dei quali deve essere un *puntatore*. Vi deve essere corrispondenza in numero e tipo fra i caratteri di conversione in `formato` e i puntatori. Termina quando `formato` è stata completamente esaminata. Restituisce `EOF` se incontra la fine del flusso, o se incorre in un errore prima di aver assegnato una qualsiasi conversione. Altrimenti, restituisce il numero di argomenti convertiti e assegnati con successo.
- La stringa `formato` è simile a quella usata per `printf`. Gli spazi sono ignorati se `formato` non contiene spazi; altrimenti, tutti gli spazi nel corrispondente punto dell'input vengono saltati.

- I caratteri ordinari in `formato` (eccetto %) devono collimare con l'input.
- Nel caso in cui `stdin` non contenga alcun carattere, la chiamata a `scanf` blocca il programma in attesa dell'input dell'utente da tastiera, input che consiste di tutto quanto l'utente digita fino al primo invio (carattere `'\n'`).
- La stringa `format` può contenere specifiche di conversione come `%*s`, in cui il carattere `*` indica che va letta una stringa, ma non va assegnata ad alcuna variabile.
- Per esempio,

  ```
  scanf("%d+%f", &x, &y);
  ```

  legge un `int` e un `float` separati da +, e li memorizza nelle variabili `x` e `y`, rispettivamente.
- Per dettagli completi sulla semantica di `scanf` si vedano le pp. 262–264 del K&R.

Lettura di stringhe

- `char *gets(char *s)` — **Non si usa più**: può causare `buffer overflow`. La si può però ancora incontrare in vecchi sorgenti.
- Legge da `stdin` la riga in ingresso successiva, e la memorizza nell'array `s`.
- Sostituisce il carattere `'\n'` con `'\0'`.
- Restituisce `s`, oppure `NULL` se incontra `EOF` (chiusura dello stream) oppure se incorre in un errore.
- *Versione sicura*, da usare sempre: `int fgets(char *s, n, FILE *stream)`, utile anche perché permette di imporre il limite massimo `n-1` al numero di caratteri letti. Qui `stream` deve valere `stdin` per leggere da tastiera.
- Oltre a quanto detto, differisce da `gets` anche perché include `'\n'` fra i caratteri letti, e gli aggiunge `'\0'` in coda.

Nota

Per la prova di laboratorio, è importante saper scrivere rapidamente e senza difficoltà il codice per leggere interi, caratteri, float o double, stringhe, ecc. Se mi ci vuole un'ora a scrivere e far funzionare il codice necessario alla lettura e alla memorizzazione dei dati, o più in generale alla gestione dell'input/output, non ho *alcuna chance* di superare la prova di laboratorio. Vedremo ora i primi esempi; altri esercizi saranno svolti in laboratorio. È bene che vi esercitiate anche da soli fino a quando non vi serva più consultare la documentazione per eseguire input da terminale.

eco.c

```c
#include <stdio.h>

int main(void)
{
  int i; char c; float f; double r;

  scanf("%d%*c%c%f%lf", &i, &c, &f, &r);
  printf("int:\t%d\nchar:\t%c\nfloat:\t%g\ndouble:\t%g\n", i,c,f,r);

  return 0;
}
```

Uso di `scanf` *per leggere dati di tipo primitivo.*

strscan.c

```c
#include <stdio.h>

int main(void)
{
  char s[BUFSIZ]; //Array di char grande: BUFSIZ definito in stdio.h

  printf("Dimmi il tuo nome.\n");
  scanf("%s", s); //NOTA: s, non &s
  printf("Ma ciao, %s!\n", s);

  return 0;
}
```

Uso di `scanf` *per leggere una stringa.*

indovinello.c

```c
#include <stdio.h>

int main(void)
{
  char s[BUFSIZ]; //Array di char grande: BUFSIZ definito in stdio.h

  printf("Dimmi il tuo nome.\n");
  scanf("%s", s); //NOTA: s, non &s
  printf("Ma ciao, %s!\n", s);
  printf("Dimmi si' (S) o (N).\n");
  char c = getchar();
  if (c=='S')
    printf("Mi hai detto di si'!\n");
  else
    printf ("Mi hai detto %c.\n",c);

  return 0;
}
```

Perché non funziona come ti aspetteresti a prima vista?

indovinellosol.c

```c
#include <stdio.h>

int main(void)
{
  char s[BUFSIZ]; //Array di char grande: BUFSIZ definito in stdio.h
  char c;

  printf("Dimmi il tuo nome.\n");
  scanf("%s%*c", s); //NOTA: s, non &s
  printf("Ma ciao, %s!\n", s);
  printf("Dimmi si' (S) o (N).\n");
  c = getchar();
  if (c=='S')
    printf("Mi hai detto di si'!\n");
  else printf ("Mi hai detto %c.\n",c);
  return 0;
}
```

Perché rimane un `'\n'` *sullo stream* `stdin`.

leggistr.c

```c
#include <stdio.h>

int main(void)
{
  char s[BUFSIZ]; //Array di char grande: BUFSIZ definito in stdio.h

  printf("Dimmi una frase.\n");
  fgets(s, BUFSIZ, stdin);
  printf("%s",s); //NOTA: Non aggiungo \n alla stampa

  return 0;
}
```

Uso di `fgets` *per leggere una stringa.*

leggistrerr.c

```c
#include <stdio.h>

int main(void)
{
  char s[BUFSIZ]; //Array di char grande: BUFSIZ definito in stdio.h

  printf("Dimmi una frase.\n");
  if (fgets(s, BUFSIZ, stdin) == NULL) //ERRORE o EOF
    printf("Errore in lettura oppure EOF\n");
  else
    printf("%s",s); //NOTA: Non aggiungo \n alla stampa

  return 0;
}
```

Uso di `fgets` *per leggere una stringa, con controllo d'errore.*

EOF.c

```c
#include <stdio.h>

int main(void)
{
  int c;    //Deve essere int, non char, per testare EOF secondo
        //lo standard C

  printf("Dimmi EOF! (Cioe' CTRL+D (Unix) o CTRL+Z (Windows).)\n");
  if ( (c=getchar()) == EOF ) //EOF definita in stdio.h
    printf("L'hai detto!\n");
  else
    printf("Non l'hai detto...\n");

  return 0;
}
```

Uso di `getchar()` *per controllare la fine del flusso* (`EOF`).