

# Programmazione 1

## Lezione 7

Vincenzo Marra

`vincenzo.marra@unimi.it`

Dipartimento di Matematica *Federigo Enriques*

Università degli Studi di Milano

12 aprile 2017

## Passaggio di parametri: semantica

Abbiamo visto che un programma in C tipico è costituito da più funzioni che si chiamano 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 chiamano 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.

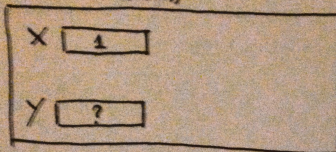
---

```
1  int main(void)
2  {
3      int f(int);    //Accetta un intero, restituisce un intero
4
5      int x=1, y;    //Variabili locali allocate nella
6                    //memoria del main
7      y=f(x);        //Chiamata ad f
8                    //Qui y vale 0, x vale 1
9      return -1;
10 }
11 int f(int z)        //Il parametro formale z potrebbe anche chiamarsi x
12 {                  //senza che ciò causi conflitti con la x del main
13     //Qui è allocata la variabile locale z nella memoria locale
14     //di f (disgiunta da quella del main e di ogni altra procedura)
15     //In z, cioè nelle celle di memoria allocate per z,
16     //è COPIATO il valore (1) del parametro passato dal chiamante.
17     z=-1; //Il valore 1 è sovrascritto da -1
18     return 0; //E' restituito il valore 0, e lo spazio di memoria
19 //locale di f è deallocato: la variabile locale z non esiste più.
20 }
```

---

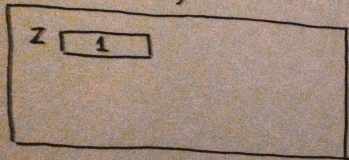
PASSAGGIO DEI PARAMETRI PER COPIA

int main(void)



MEMORIA LOCALE  
DEL MAIN

int f(int z)



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`.

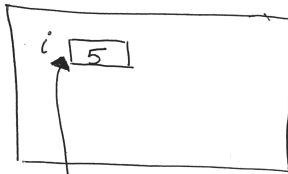
---

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      void f(int *);
6      int i=5;
7
8      printf("Prima: i=%d\n", i);
9      f(&i); /* Passa ***L'INDIRIZZO*** di i */
10     printf("Dopo: i=%d\n", i);
11     return 0;
12 }
13
14 void f(int *p)
15 {
16     (*p)++;
17     return;
18 }
```

---

*Qual è l'output del programma?*

```
int main(void)
```



MEMORIA  
LOCALE  
di main

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



MEMORIA  
LOCALE  
di f

---

par2.c

---

```
1  #include <stdio.h>
2  int main(void)
3  {
4      void f(int *);
5      int i=0;
6      int *ptr;
7      ptr=&i;
8      *ptr=5;
9      printf("Prima: i=%d\n",i);
10     f(ptr); /* Passa ***UN PUNTATORE*** a i */
11     printf("Dopo: i=%d\n", i);
12     return 0;
13 }
14 void f(int *p)
15 {
16     (*p)++;
17     return;
18 }
```

---

*Qual è l'output del programma?*



---

par3.c

---

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      void f(int *);
6      int i=5;
7
8      printf("Prima: i=%d\n", i);
9      f(&i);
10     printf("Dopo: i=%d\n", i);
11     return 0;
12 }
13
14 void f(int *p)
15 {
16     p++;
17     return;
18 }
```

---

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

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

è del tutto coerente col prototipo

```
f(arr);
```

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

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

è del tutto coerente col prototipo

```
f(arr);
```

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

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

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

è del tutto coerente col prototipo

```
f(arr);
```

Si applica la solita semantica del **passaggio per valore**: 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`.

**Nota Bene.** 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. In particolare, il tipo `String` che abbiamo usato in laboratorio è definito a scopi didattici nella libreria `Prog1`, ma è solo un sinonimo per un array di `char` molto grande.



---

```
1      strcpy1.c
2  /* strcpy(char *s, char *t): copia t in s */
3
4  int main(void)
5  {
6      void strcpy1(char*, char*);
7      char s[20];
8      char t[]="Pinco pallino";
9
10     strcpy1(s,t);
11     printf("s: %s\nt: %s\n",&s[0],t); //&s[0] equiv. a s
12     return 0;
13 }
14
15 void strcpy1(char *s, char *t)
16 {
17     int i=0;
18     while ( (s[i]=t[i]) != '\0')
19         i++;
20 }
```

---

---

```
1      strcpy2.c
2  /* strcpy(char *s, char *t): copia t in s */
3  #include<stdio.h>
4  int main(void)
5  {
6      void strcpy2(char*, char*);
7      char s[20];
8      char t[]="Pinco pallino";
9
10     strcpy2(s,t);
11     printf("s: %s\nt: %s\n",s,t);
12     return 0;
13 }
14 void strcpy2(char *s, char *t)
15 {
16     while ( (*s=*t) != '\0')
17     {
18         s++;
19         t++;
20     }
21 }
```

---

---

```
1  /* strcpy(char *s, char *t): copia t in s */
2  #include<stdio.h>
3
4  int main(void)
5  {
6      void strcpy3(char*, char*);
7      char s[20];
8      char t[]="Pinco pallino";
9
10     strcpy3(s,t);
11     printf("s: %s\nt: %s\n",s,t);
12     return 0;
13 }
14
15 void strcpy3(char *s, char *t)
16 {
17     while ( (*s++==*t++) != '\0');
18 }
```

---

---

```
1  /* strcpy(char *s, char *t): copia t in s */
2  #include<stdio.h>
3
4  int main(void)
5  {
6      void strcpy4(char*, char*);
7      char s[20];
8      char t[]="Pinco pallino";
9
10     strcpy4(s,t);
11     printf("s: %s\nt: %s\n",s,t);
12     return 0;
13 }
14
15 void strcpy4(char *s, char *t)
16 {
17     while ( (*s++==*t++) );
18 }
```

---

## Lunghezza delle stringhe

Scrivere una funzione di prototipo

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

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

---

lung1.c

---

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int lungstr(char *);
6      char t[]="Ave";
7
8      printf("La stringa e' lunga %d\n",lungstr(t));
9      return 0;
10 }
11
12 int lungstr(char *s)
13 {
14     int i;
15     for(i=0; s[i]!='\0'; i++);
16     return i;
17 }
```

---

## 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

---

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int lungstr(char *);
6      char t[]="Ave";
7
8      printf("La stringa e' lunga %d\n",lungstr(t));
9      return 0;
10 }
11
12 int lungstr(char *s)
13 {
14     if (s==NULL)
15         return -1;
16
17     int i;
18     for(i=0; s[i]!='\0'; i++);
19     return i;
20 }
```

---

## 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:

- ❶ `stdin` – associato d'ufficio ai dati in **ingresso** provenienti dalla tastiera.
- ❷ `stdout` – associato d'ufficio ai dati in **uscita** visualizzati sul monitor del calcolatore.
- ❸ `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 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.
- 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.

## Esercizio

Studia il codice sorgente delle funzioni della libreria didattica Prog1. Poi prova a scrivere delle tue funzioni di lettura per tipi diversi di dati, introducendo varianti utili alle funzioni della libreria.

## Esercizio

Studia il codice sorgente delle funzioni della libreria didattica Prog1. Poi prova a scrivere delle tue funzioni di lettura per tipi diversi di dati, introducendo varianti utili alle funzioni della libreria.

## Nota Bene

All'esame è permesso usare solamente le funzioni della libreria standard del C. Quindi è importante saper scrivere rapidamente e senza difficoltà il codice per leggere interi, caratteri, float o double, stringhe. Vedremo ora i primi esempi; altri esercizi saranno svolti in laboratorio.

---

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i; char c; float f; double r;
6
7      scanf("%d*c%c%f%lf", &i, &c, &f, &r);
8      printf("int:\t%d\nchar:\t%c\nfloat:\t%g\ndouble:\t%g\n", i,c,f,r);
9
10     return 0;
11 }
```

---

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

---

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char s[BUFSIZ]; //Array di char grande: BUFSIZ definito in stdio.h
6
7      printf("Dimmi il tuo nome.\n");
8      scanf("%s", s); //NOTA: s, non &s
9      printf("Ma ciao, %s!\n", s);
10
11     return 0;
12 }
```

---

*Uso di scanf per leggere una stringa.*

---

```
1  _____ indovinello.c _____
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7      char s[BUFSIZ]; //Array di char grande: BUFSIZ definito in stdio.h
8
9      printf("Dimmi il tuo nome.\n");
10     scanf("%s", s); //NOTA: s, non &s
11     printf("Ma ciao, %s!\n", s);
12     printf("Dimmi si' (S) o (N).\n");
13     char c = getchar();
14     if (c=='S')
15         printf("Mi hai detto di si'!\n");
16     else
17         printf ("Mi hai detto %c.\n",c);
18
19     return 0;
20 }
```

---

*Perché non funziona come ti aspetteresti a prima vista?*

---

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char s[BUFSIZ]; //Array di char grande: BUFSIZ definito in stdio.h
6      char c;
7
8      printf("Dimmi il tuo nome.\n");
9      scanf("%s%c", s, &c); //NOTA: s, non &s; e &c, non c
10     printf("Ma ciao, %s!\n", s);
11     printf("Dimmi si' (S) o (N).\n");
12     c = getchar();
13     if (c=='S')
14         printf("Mi hai detto di si'!\n");
15     else printf ("Mi hai detto %c.\n",c);
16     return 0;
17 }
```

---

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



---

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char s[BUFSIZ]; //Array di char grande: BUFSIZ definito in stdio.h
6
7      printf("Dimmi una frase.\n");
8      fgets(s, BUFSIZ, stdin);
9      printf("%s",s); //NOTA: Non aggiungo \n alla stampa
10
11     return 0;
12 }
```

---

*Uso di fgets per leggere una stringa.*

---

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char s[BUFSIZ]; //Array di char grande: BUFSIZ definito in stdio.h
6
7      printf("Dimmi una frase.\n");
8      if (fgets(s, BUFSIZ, stdin) == NULL) //ERRORE o EOF
9          printf("Errore in lettura oppure EOF\n");
10     else
11         printf("%s",s); //NOTA: Non aggiungo \n alla stampa
12
13     return 0;
14 }
```

---

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

---

EOF.c

---

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int c;          //Deve essere int, non char, per testare EOF secondo
6                      //lo standard C
7
8      printf("Dimmi EOF! (Cioe' CTRL+D.)\n");
9      if ( (c=getchar()) == EOF ) //EOF definita in stdio.h
10         printf("L'hai detto!\n");
11     else
12         printf("Non l'hai detto...\n");
13
14     return 0;
15 }
```

---

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