

---

## C: un po' di storia...

Il linguaggio C nasce negli anni '70 nei laboratori Bell; nello stesso periodo e nello stesso luogo, viene scritta la prima versione del sistema operativo Unix.

Dennis Ritchie sviluppò il linguaggio C come evoluzione del linguaggio B di Ken Thompson, autore della prima versione di Unix. A sua volta, il B (interpretato) era una semplificazione del BCPL, linguaggio *portabile*, in altre parole nascondeva al programmatore le caratteristiche della macchina su cui girava.

Questa capacità venne ereditata dal suo discendente C, tanto che ben presto, Unix venne riscritto in C e portato su piattaforme differenti. Si trattò di una scelta innovativa: fino ad allora i sistemi operativi erano normalmente scritti nel linguaggio assembler della macchina, e non con un linguaggio ad alto livello.

Il C ha subito varie revisioni, e la versione standard è detta ANSI C, mentre la versione originale (ormai completamente abbandonata) è detta C K&R (da Kernighan&Ritchie)

**ATTENZIONE:** Ogni compilatore C++ è anche un compilatore C.

---

## Java e C a confronto

Punti di forza del C:

- permette un elevato controllo della macchina, pur essendo un linguaggio ad alto livello: c'è una notevole corrispondenza tra il codice C e il codice assembler generato;
- permette l'accesso diretto a molte delle caratteristiche interne del sistema;
- prestazioni migliori (codice nativo)
- Java (ma non solo) nasconde molti dettagli di basso livello che sono essenziali per realizzare un sistema operativo.
- linguaggio semplice, con pochi costrutti;

D'altra parte, si può osservare che:

- la gestione della memoria, le inizializzazioni delle variabili e la gestione degli errori sono interamente demandate al programmatore;
- c'è un maggiore rischio di scrivere codice non funzionante in C (almeno all'inizio)
- le librerie standard del C sono poche.

In generale, C consente allo sviluppatore che sa quel che vuole fare di farlo rapidamente ed efficientemente. Il sistema di gestione a runtime è molto leggero (praticamente inesistente).

---

## Like Java, like C

La sintassi di Java ricalca quella del C++, a sua volta basata sul C.

### Operatori aritmetici

- `int i = i+1; i++; i--; i *= 2;`
- `+, -, *, /, %,`

### Operatori relazionali e logici

- `<, >, <=, >=, ==, !=`
- `&&, ||, &, |, !`

### Strutture di controllo

- `if ( ) { } else { }`
- `while ( ) { }`
- `do { } while ( );`
- `for(i=1; i <= 100; i++) { }`
- `switch ( ) {case 1: & }`
- `continue; break;`

### Dichiarazione delle funzioni

- `<tipo> nomefunzione ([<tipopar1> <nomepar1>, ...]) { }`
- `int calcola(float a, float b) { ... return(ris); }`

Attenzione: se non viene specificato il tipo restituito da una funzione, questa viene considerata per default di tipo `int`, e non `void` !

---

## OOP: un concetto recente

Le differenze più evidenti del C rispetto a Java sono sicuramente:

- la mancanza di oggetti;
- la mancanza di polimorfismo.

Passare da Java a C è un po' ripercorrere al contrario la storia della programmazione. Il paradigma di programmazione ad oggetti è entrato nell'uso corrente in tempi recenti (a cavallo tra gli anni '80 e '90), ed ha impiegato tempo per imporsi.

Lo stesso si può dire per il concetto di polimorfismo (a volte presente anche in linguaggi non ad oggetti).

Lo stile di programmazione, ovviamente, ne risente. Questo non vuol dire però che non sia possibile scrivere del “buon” codice in C (leggibile, mantenibile, riutilizzabile). Uno stile di programmazione strutturata e *modulare* è sufficiente per raggiungere questi obiettivi.

---

## Struttura di un programma C

Ogni programma C dispone di una funzione main, che

- restituisce int (anche se molti compilatori accettano void)
- può prendere in input due valori, un intero e un vettore di stringhe, che rappresentano rispettivamente il numero di parametri passati al programma e i parametri stessi.

Possono essere definite nello stesso file C diverse funzioni, indipendenti l'una dall'altra, senza relazioni gerarchiche.

In questo caso, se una funzione (come SayHelloTo() sotto) è invocata in un punto che ne precede la definizione, essa deve prima essere *dichiarata*, introducendone il *prototipo*.

Il prototipo o *firma* non è altro che l'intestazione della funzione, terminato da un ";".

Ciò informa il compilatore dell'*interfaccia* della funzione e gli permette di stabilire se essa è invocata correttamente.

Ogni funzione definita nel file è pubblica per tutte le altre funzioni del file.

```
#include <stdio.h>
void SayHelloTo(char *); // prototipo

int main(int argc, char *argv[])
{
    SayHelloTo("World");
    return 0;
}
void SayHelloTo(char *thing)
{
    printf("Hello, %s\n", thing);
    return;
}
```

---

## Struttura di un programma C

È possibile suddividere un programma C in differenti file sorgente (“moduli”).

È probabile che qualche modulo contenga:

- (definizioni di) funzioni e/o
  - variabili esterne a definizioni di funzioni (vedi pag. 9)
- destinati ad essere utilizzati in altri moduli.

I prototipi di queste funzioni e le dichiarazioni delle variabili, insieme a eventuali definizioni di tipi e di costanti, possono essere inserite in “include file” con estensione *.h*, che i moduli possono includere tramite l'apposita direttiva **#include**.

L'inclusione permette al modulo che la effettua di invocare funzioni dichiarate nel file *.h* incluso e definite in altri moduli.

Questa è più che altro una comodità per evitare di riscrivere il prototipo di una funzione in ogni modulo che la invoca.

Il linguaggio C non prevede limitazioni all'accesso alle funzioni definite in altri moduli (come *public/private/protected*): il compilatore permette a qualunque modulo di chiamare qualsiasi funzione (al più con un warning se non coincide con il prototipo), il linker poi collega a qualsiasi funzione abbia il nome desiderato.

L'unico sistema per ovviare a questo problema, è l'utilizzo del qualificatore *static*. Se un modulo qualifica *static* una funzione in esso definita, ne impedisce l'uso da parte di altri moduli (anche se questi dovessero includere i prototipi della funzione *static*).

---

## Struttura di un programma C: esempio

sayhello.h:

```
void SayHelloTo(char *); /* prototipo */
```

sayhello.c

```
#include <stdio.h>
#include "sayhello.h"

// se si premettesse static, SayHelloTo
// non sarebbe invocabile da main.c

void SayHelloTo(char *thing)
{
    printf("Hello, %s\n", thing);
    return;
}
```

main.c:

```
#include <stdio.h>
#include "sayhello.h"

int main(int argc, char *argv[])
{
    SayHelloTo("World");
    return 0;
}
```

---

## Variabili e costanti

Le variabili sono tipizzate, ma il compilatore non effettua controlli in caso di conversione errata di tipi.

```
float reale = 10000000000.0;
int intero;
...
intero = (int)reale; // type cast
```

Il **type cast** (conversione) causa una perdita di precisione, che non è però segnalata a compile- né a run-time

Le variabili dichiarate all'interno del blocco di una funzione si dicono locali (per la funzione).

Attenzione: tali dichiarazioni possono figurare esclusivamente all'inizio del blocco (prima delle istruzioni).

Per quanto riguarda lo scope, le variabili locali nascondono quelle esterne (vedi pag. 9).

Il C permette di definire costanti (tipizzate) tramite il costrutto `const`. Ad una variabile dichiarata *const tipo* può essere assegnato il valore solo in fase di dichiarazione.

```
const int nonmitoccare = 3;
...
nonmitoccare = 2; /* errore in compilazione !*/
```

L'alternativa, comunemente adottata, è la direttiva `#define` che fa in realtà ricorso al preprocessore (vedere pag.25).

Modificatore `static`: le variabili interne `static` di una funzione rimangono in vita anche dopo l'uscita dalla funzione. Le variabili esterne `static` sono visibili solo nel file in cui sono dichiarate (private).

---

## Variabili globali

Una variabile definita esternamente alle funzioni sarà visibile a tutte le funzioni definite nel file, e viene detta **globale**. L'uso di variabili globali ha effetti negativi sulla leggibilità del codice (la modifica di una variabile globale può influenzare ogni funzione che ne fa uso).

Tuttavia esso non è infrequente, per motivi di efficienza, nella programmazione di sistema a basso livello.

Per utilizzare una variabile globale *x*, in un file nel quale non è definita, è necessario dichiararla con la direttiva *extern x* (o verrebbe allocata memoria per *x* di nuovo).

Notate la distinzione:

- definizione (di funzioni o variabili): alloca memoria
- dichiarazione (di funzioni o variabili): non alloca memoria

global.h:

```
extern int globale; // dichiarazione della variabile globale
void stampa();      // dichiarazione della funzione stampa()
```

global.c:

```
#include <stdio.h>
#include "global.h"
void stampa()      // definizione della funzione stampa()
{
    printf("%d\n", globale);
}
```

main.c:

```
#include <stdio.h>
#include "global.h" //include la dichiarazione della funzione stampa()
int globale;       //definizione della variabile globale (memoria allocata)
int main()
{
    globale = 2; stampa();
    globale = 3; stampa();
    return(0);
}
```

---

## Definizione di tipi e di struct.

Anche in assenza di oggetti, è possibile definire delle strutture dati avanzate (quasi come definire degli oggetti, ma senza i relativi metodi).

Tramite la parola riservata **struct** è possibile aggregare dei *membri*, cioè dati di tipi eterogenei (un po' come un *record* di un db aggrega degli *attributi* o *campi*).

```
struct complex {double real; double imag;};
....
struct complex a,b; //NB: complex da solo non è un tipo
.....           //il tipo è: struct complex
a.real=2.0; a.imag=3.0;
```

È possibile definire dei tipi personalizzati, per rendere più chiaro il codice:

```
typedef char[20] mybuffer;
typedef char carattere;
...
mybuffer b;
carattere c;
```

Si può (e conviene) ribattezzare un tipo struct.

(qui non serve la *label* complex che negli esempi sopra segue struct)

```
typedef struct {double real; double imag;} Complex;
Complex a, b;
```

L'operatore `sizeof(<nomevar>)` restituisce la dimensione in byte occupata dalla variabile specificata. Viene risolto a tempo di compilazione.

---

## Tipi di dati enumerati

È possibile definire dei tipi di dato enumerati:

```
enum giorniSettimana {  
    LUNEDI,  
    MARTEDI,  
    MERCOLEDI,  
    GIOVEDI,  
    VENERDI,  
    SABATO,  
    DOMENICA  
}
```

LUNEDI vale 0, MARTEDI 1 e così via.

Si può anche imporre il valore iniziale della sequenza:

```
enum giorniSettimana {  
    LUNEDI = 1,  
    MARTEDI,  
    MERCOLEDI,  
    GIOVEDI,  
    VENERDI,  
    SABATO,  
    DOMENICA  
}
```

L'uso di enum può rendere più leggibile il codice.

Infatti, si interpone un livello di astrazione tra la rappresentazione del dato e la sua semantica.

```
int giornoNoEnum;  
giorniSettimana giornoEnum;  
  
giornoNoEnum = 1;          /* ?????? */  
giornoEnum = LUNEDI;      /* ok */
```

---

## Variabili e allocazione della memoria

In Java, il comportamento del gestore di memoria differisce in base al tipo di variabile definita:

- se è un tipo base (int, float, char,...) viene allocata staticamente nello stack del blocco di codice corrispondente.
- se è un'istanza di un oggetto, viene riservato lo spazio necessario per mantenere un **riferimento** all'oggetto stesso. Tramite l'operatore new è possibile allocare effettivamente lo spazio richiesto, da una zona differente di memoria (detta heap).

In C non esistono oggetti; è però possibile definire esplicitamente variabili di tipo **puntatore**, che permettono di allocare in un secondo tempo, in base all'effettiva necessità, un qualsiasi tipo di dati. I puntatori sono *tipizzati* (puntatore a interi, a reali,...), anche se è possibile effettuare dei cast.

```
int *p;
Complex *c;
struct prova *a;
```

Le funzioni utili per la gestione della memoria (definite nella libreria stdlib.h) sono:

- `void *calloc(size_t nmemb, size_t size);`  
alloca la memoria necessaria per contenere nmemb elementi ciascuno di dimensione size, e ne azzera il contenuto; restituisce il puntatore alla memoria allocata.

- 
- `void *malloc(size_t size);`  
alloca la memoria necessaria per contenere `size` byte;  
restituisce il puntatore alla memoria allocata.
  - `void free(void *ptr);`  
dealloca la memoria utilizzata. Al contrario di Java, in C non esiste nativamente un sistema di garbage collection, quindi questa funzione deve essere chiamata esplicitamente, per liberare la memoria.
  - `void *realloc(void *ptr, size_t size);`  
Tenta di cambiare la dimensione della memoria allocata da `*ptr` dalla dimensione corrente a `size`.

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

int main(int argc, char *argv[])
{
    int *a, *b;

    a = malloc(sizeof(int));
    b = malloc(sizeof(int));

    *a = 2; *b = 3;
    free(a); /* rilascia la memoria allocata da a */
    a = b;   /* entrambi i puntatori fanno riferimento
              alla stessa zona di memoria */
    free(b);
}
```

Una variabile puntatore contiene quindi l'indirizzo dell'“oggetto” a cui fa riferimento.

---

In un'espressione come:

\*p

viene effettuata la *dereferenziazione* del puntatore p: se p, variabile di tipo puntatore al tipo T, contiene l'indirizzo di un "oggetto" di tipo T, allora \*p è l'"oggetto" di tipo T referenziato da p.



Viceversa, data una variabile x di tipo T, il suo indirizzo si ottiene tramite l'operatore &:



Riprendendo l'esempio precedente, per effettuare il confronto tra a e b non si può scrivere:

```
if (a == b) ....
```

perché questa operazione confronta tra loro i puntatori, non le zone di memoria a cui essi fanno riferimento; in maniera del tutto analoga a quanto succede in Java nel confronto tra due oggetti.

Bisogna invece scrivere:

```
if (*a == *b) ....
```

Per l'accesso a elementi di struct referenziati tramite puntatore, è prevista una sintassi alternativa semplificata.

```
Complex *c;  
c = (Complex *)malloc(sizeof(Complex));  
c->real = 2.0; c->imag = 3.0;  
/* equivalenti a (*c).real = 2.0; (*c).imag = 3.0; */
```

---

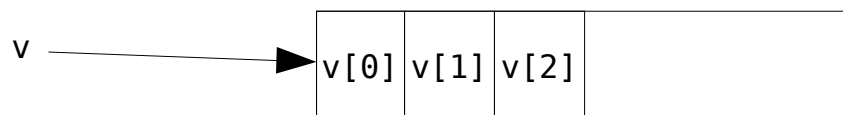
## Vettori

Per definire un vettore, il compilatore deve conoscerne l'esatta dimensione. La dimensione deve essere espressa attraverso una costante.

```
#define DIMENSIONE 500
int v[1000];
int altrovettore[DIMENSIONE];
```

Un vettore in C non conserva informazioni sulla propria dimensione: e' semplicemente un blocco di memoria. Bisogna prestare attenzione a non uscire dai limiti, il compilatore e il sistema di runtime non segnalano errori.

In generale, un vettore può essere visto come un puntatore dal valore immutabile, che fa riferimento al primo elemento del vettore.



Grazie a questa rappresentazione, e' possibile aggirare la limitazione sull'allocazione statica dei vettori, dichiarando un puntatore ed allocandolo (con malloc/calloc) dinamicamente.

```
int *v, dim;
...
// leggi il valore della variabile dim dalla tastiera
v = (int*)malloc(dim*sizeof(int));
// Ora v è (punta a) un array di dim int
```

---

## Aritmetica dei puntatori

Ogni operazione che fa riferimento a indici di un array, può essere riscritta in forma più veloce usando i puntatori,

Se  $p$  e  $q$  puntano a due elementi di un array:

- $p+n$  punta all' $n$ -simo elemento dopo quello puntato da  $p$ ;
- $*(p+n)$  si può anche scrivere  $p[n]$
- $p-n$  punta all' $n$ -simo elemento prima di quello puntato da  $p$
- $p-q$  e' il numero di elementi tra quelli puntati da  $p$  e  $q$
- $p < q$  e' vero se l'elemento puntato da  $p$  viene prima di quello puntato da  $q$  nell'array (vale anche per  $\leq$   $>$   $\geq$ ).
- $p$  e  $q$  non vanno combinati se puntano a elementi di array diversi.

Ad es., dati:

```
int a[10];  
p = &a[0];
```

$*(p+1)$ ,  $p[1]$  e  $a[1]$  sono lo stesso dato.

Inoltre, in C, un nome di vettore  $a$  fa da puntatore ad  $a[0]$ , quindi:

- $a$ ,  $p$  e  $\&a[0]$  sono lo stesso puntatore;
- $a+1$  e  $\&a[1]$  sono lo stesso puntatore (ad  $a[1]$ );
- $*(a+1)$  e  $a[1]$  sono lo stesso elemento.

Così, in generale, l'indirizzo dell' $i$ -esimo elemento del vettore viene calcolato come:

$\langle \text{indirizzo } 1^\circ \text{ elemento} \rangle + \langle \text{indice } i \rangle * \langle \text{dimensione elemento} \rangle$

---

## Dichiarazioni composte

In C una dichiarazione di un identificatore ha la forma:

tipo   dichiaratore

e dichiaratore ha una delle 5 forme:

- identificatore
- (dichiaratore)
- dichiaratore()
- \*dichiaratore
- dichiaratore[espressione-costante]

Per esempio:

```
int *a[5];  
char (*b)[5];  
char x[3][5];
```

\*   ()   [e]   si dicono costruttori e si leggono così:

*	si legge	puntatore a
()		funzione che restituisce
[e]		array di e

In un dichiaratore composto, [] lega più forte di (), e () di \*. Per interpretare un dichiaratore composto, si leggono, nell'ordine:

- l'identificatore
- "è un"
- i costruttori dal più interno al più esterno
- il tipo

Esempi:

int *a[5]	dà	a è un array di 5 puntatori a int,
char (*b)[5]		b è un puntatore a un array di 5 char
char x[3][5]		x è un array di 3 array di 5 char.

---

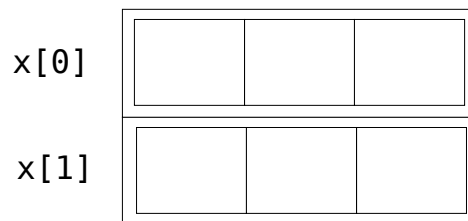
## Array a più dimensioni

Un array a 2 dimensioni, p. es. 2 righe e 3 colonne, si può dichiarare con:

```
static int x[2][3]={ /* nel C originale, l'inizializzazione è */
    {1, 2, 3},      /* ammessa solo se l'array è static */
    {3, 5, 6}
};
```

Questa dichiarazione va letta così:

x è un: array di: 2 array di 3 int, cioè:



Convenzionalmente, in `x[2][3]`, il primo indice dà il n. di righe, dunque un array a due dimensioni è in realtà un array di array riga.

Vediamo come viene tradotto un riferimento a, p.es., `x[1][2]`, secondo le regole già viste:

- `x+1` è l'indirizzo del 2° elemento di `x`;
- `x[1]` è `*(x+1)`, cioè il 2° elemento di `x`; si tratta di un array di 3 int;
- in quanto array, `x[1]` viene tradotto nell'indirizzo del 1° dei suoi 3 elementi;
- `x[1]+2` è l'indirizzo del 3° elemento dell'array `x[1]`;
- `x[1][2]` è `*(x[1]+2)`, cioè il 3° elemento dell'array `x[1]`; si tratta dunque di un int.

## Stringhe

---

In C non esiste un tipo stringa nativo. Le stringhe sono rappresentate tramite vettori di caratteri terminati dal carattere 0 (NULL).

Per quanto visto precedentemente, un vettore è in realtà un puntatore al primo elemento del vettore.

Una stringa può essere (come vettore) inizializzata al momento della definizione:

```
char a[10] = "ciao";  
char b[] = "prova";
```

Analogamente, dati s e t stringhe, è errato scrivere s=t per assegnare ad s il valore di t, perché verrebbe copiato il puntatore a t. Per copiare una stringa, occorre copiarne i singoli elementi, facendo attenzione a non “uscire” dalla memoria allocata.

Per la manipolazione delle stringhe, il C mette a disposizione alcune funzioni di libreria definite in string.h.

Copia di stringhe (src in dest) (strncpy si ferma ai primi n caratteri):

```
char *strncpy(char *dest, const char *src);  
char *strncpy(char *dest, const char *src, size_t n);
```

Confronto di stringhe (restituisce valori risp. <0 , =0 e >0 se s1<s2, s1=s2 o s1>s2) (strncmp si ferma ai primi n caratteri)

```
int strcmp(const char *s1, const char *s2);  
int strncmp(const char *s1, const char *s2, size_t n);
```

Concatenazione di stringhe (appende src alla fine di dest, che deve essere sufficientemente grande; strncat si ferma ai primi n caratteri)

```
char *strncat(char *dest, const char *src);  
char *strncat(char *dest, const char *src, size_t n);
```

È consigliato l'uso delle versioni con controllo della dimensione.

---

## Funzioni

I metodi di Java e C++ sono, a tutti gli effetti, direttamente derivati dalle funzioni del C.

```
tipo nomefunzione([tipopar1 par1, tipopar2 par2, ...])
```

Punti critici da ricordare:

- Le variabili locali vengono allocate alla chiamata della funzione, e rilasciate all'uscita (ogni dato ivi contenuto va perso).
- **In C esiste unicamente il passaggio per valore.**

Come si può ovviare a questa limitazione?

In C++ ci sono i reference, in Java ogni oggetto viene passato implicitamente per puntatore.

Basta esplicitare il puntatore: alla funzione si passa non un "oggetto", ma un puntatore ad esso (che è comunque sempre una variabile). Non è possibile modificare il puntatore, ma l'"oggetto" puntato sì.

Esempio:

```
swap(int *p, *q) {  
    int temp;  
    temp = *p;  
    *p = *q;  
    *q = temp;  
}  
...  
swap(&a, &b);
```

---

## Esempio: getint

```
/**
 * Restituisce un intero, letto da tastiera
 * @param pn Puntatore ad intero (il valore letto)
 * @return Il segno del valore letto
 */
char getint(int * pn) {
    int s, c, sign;

    /* Salta eventuali spazi bianchi iniziali: */
    while ((c = s = getchar()) == ' ' ||
           c == '\n' || c == '\t');

    /* Determinare il segno del numero da leggere: */
    sign = 1;
    if ( s == '+' || s == '-' ) {
        /* c è un segno prima di num. */
        sign = (s=='+') ? 1 : -1;
        c = getchar();
    } else s = ' ';

    /* Ciclo di elaborazione: per porre in *pn
     l'intero in input, p.es. 435:
     - si legge una nuova cifra (finché ce ne sono)
       p.es. '5'
     - a questo punto, *pn dovrebbe valere 43; si
       moltiplica *pn per 10 e si somma 5, calcolato con
       '5'-'0':
     */

    for ( *pn = 0; c >='0' && c <= '9'; c = getchar() )
        *pn = 10 * (*pn) + c-'0';
    *pn *= sign;

    /* Alla fine, si restituisce il segno:*/
    return(s);
}
```

---

## Array e funzioni

Dato un array `a` di elementi di tipo `T`, una chiamata `f(a)` passa a `f` il valore dell'argomento `a`, cioè l'indirizzo di `a[0]`, che ha tipo `T`.

Quindi il parametro `x` di `f` sarà un puntatore a `T` di valore iniziale `a`.

Così, attraverso `x`, `f` può modificare gli elementi di `a`:

```
main() {
    int a[]={0, 2};
    f(a);
}
f(int *x) {
    x[0] = 1;
}
```

Anche se dichiarato come array, `x` resta un puntatore variabile.

Si può ovviamente utilizzare l'aritmetica dei puntatori per accedere ai vari elementi:

```
int lung(char x[])
/* restituisce la lunghezza della stringa x */
{
    int n;

    /* x++ varia l'indirizzo in x */
    for (n = 0; *x != '\0'; x++)
        n++;
    return(n);
}
```

Oppure, sfruttando la differenza tra puntatori:

```
int lung(char x[]) {
    char *p = x;
    while (*p != '\0')
        p++;
    return(p-x);
}
```

Si noti che, chiamando ad es. `lung(a+2)`, si ottiene la lunghezza del sottoarray `a[2]`.

---

## Esempio: array di puntatori a stringhe

```
/**
 * La funzione readlines(lineptr,maxlines):
 * legge le righe di input, finché non ne trova una vuota;
 * la riga letta n viene memorizzata nella memoria statica e
 * un puntatore ad essa viene posto in lineptr[n-1];
 * @param lineptr è un array (di puntatori a stringhe), per cui
 * l'effetto di readlines sarà visibile alla funzione che
 * la chiama.
 * @param maxlines numero massimo di linee da leggere.
 * @return restituisce il numero di righe lette, o -1 se
 * - le righe lette superano maxlines o
 * - la memoria disponibile per le stringhe lette si è esaurita.
 */

#define MAXLEN 1000

readlines(char *lineptr[], int maxlines) {
    /* lineptr[n] punterà alla riga letta n (da 0) */
    int len, nlines=0; char *p, line[MAXLEN];

    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines) return(-1);
        else if ((p = malloc(len)) == NULL)
            /* len byte allocati in mem. statica */
            return(-1); /* memoria esaurita */
        else {
            line[len-1] = '\0'; /* elimina il '\n' da riga letta */
            strcpy(p, line); /*copia riga in mem. non locale */
            lineptr[nlines++] = p;
        }
    return(nlines);
}
```

NB: la memoria statica puntata da lineptr va richiesta dinamicamente, con malloc, da readlines che legge le righe e sa quanta ne serve. L'alternativa è dichiarare:

```
char lineptr[MAXLIN][MAXLEN]
```

Perché introdurre p e non scrivere lineptr[nlines++] = line?

Perché p punta a memoria non locale, mentre la memoria puntata da line ha esistenza locale.

---

ATTENZIONE: NON restituire mai puntatori a variabili allocate staticamente (non tramite malloc) all'interno di una funzione. A tutti gli effetti, quella porzione di memoria viene riassegnata per altro uso, all'uscita della funzione.

---

## Preprocessore

Si occupa di processare le macro definite e modificare il programma di conseguenza. Agisce prima della compilazione, *espandendo* le macro (il compilatore vede il codice esclusivamente dopo le modifiche effettuate dal preprocessore).

**#define**: Permette di definire simboli (macro).

- **#define name expression**
- **#define name(param1, param2, ...) expression**
  - definisce *name* rispettivamente come espressione o come macro (pseudo funzione)
- **#undef name**
  - cancella la definizione di un simbolo.

Esempio:

```
#define PROVA 3
#define ALTRAPROVA (2+3)
#define quad(x) (x*x)
```

Il preprocessore provvede a sostituire *a tempo di compilazione* tutti i riferimenti alla stringa "PROVA" con la stringa "3", così come sostituisce le occorrenze di "ALTRAPROVA" con "(2+3)", e le occorrenze di `quad(x)` (dove *x* stavolta può essere qualsiasi cosa) con `(x*2)`.

Se nel codice si trova

```
a = 5 + quad(a);
```

il compilatore vedrà

```
a = 5 + (a * a);
```

---

Questo approccio presenta dei problemi. Le macro assomigliano molto alle funzioni, ed i programmatori sono portati a trattarle come tali. In realtà, la semantica è completamente differente. Consideriamo il seguente codice:

```
a = 5 + quad( a-1 );
```

 diventa 

```
a = 5 + ( a-1 * a-1 );
```

Bisogna scrivere correttamente:

```
a = 5 + quad( (a-1) );
```

 cioè 

```
a = 5 + ( (a-1) * (a-1) );
```

oppure, cambiare la definizione

```
#define quad(x) ((x)*(x))
```

```
a = 5 + quad( a-1 );
```

 cioè 

```
a = 5 + ( (a-1) * (a-1) );
```

Non è l'unico problema; confrontiamo le due definizioni:

```
int valid(x) { return (x > 0 && x < 20); }  
#define valid(x) (x>0 && x<20)
```

e la chiamata

```
if (valid(x++)) { ... }
```

- Se `valid` è una funzione, verrà richiamata con il parametro `x`, ne viene valutato il risultato, quindi `x` viene incrementato.
- Se `valid` è una macro, viene espansa come segue:  

```
if (x++>0 && x++<20) { ... }
```

con ovvie ripercussioni sul risultato.

---

**#include:** inserisce il contenuto del file specificato (normalmente un file header .h) nell'output.

Utilizzato soprattutto per importare le definizioni delle funzioni, e in questo simile ad import in Java. Attenzione: i file include NON sono librerie, contengono solo prototipi delle funzioni e le variabili esportate.

- la versione <...> fa riferimento ai file presenti nella cartella di sistema dei file include (/usr/include)
- la versione tra "..." fa riferimento ai file nella cartella corrente.

**#if,#ifdef,#ifndef:** Costrutti di compilazione condizionale  
Il preprocessore esclude dall'output le sezioni di codice che

```
#if expression
code1
#else
code2
#endif
```

```
#ifdef expression
code1
#else
code2
#endif
```

```
#ifndef expression
code1
#else
code2
#endif
```

non soddisfano le condizioni specificate.

Es.

```
#define DEBUG
....
#ifdef DEBUG
printf("attenzione: punto di controllo\n");
#endif
```

```
#if 0
Block of code to be removed
#endif
```

Il blocco non viene compilato!

---

## Input e Output

La funzione più nota per l'output di dati su standard output è

```
int printf(const char *format, ...); // print formatted
```

Essa prende in input :

- una stringa di caratteri, che specifica come i parametri successivi devono essere rappresentati;
- le variabili e/o espressioni da rappresentare.

Le espressioni di tipo `%x`, dove `x` è una lettera, eventualmente presenti all'interno della stringa di formato, sono sostituite dal parametro corrispondente per posizione, rappresentato come specificato dal *conversion specifier* `x`. Tra i conversion specifier più usati, troviamo:

- `%d` : intero con segno;
- `%u` : intero senza segno;
- `%f` : float (reale singola precisione);
- `%e`, `%lf`: double (reale doppia precisione);
- `%c` : carattere.

```
public class esempio
{
    public static void main(String argv[])
    {
        int intero=2;
        float reale=3.0;
        System.out.println("Il valore intero "+
            "e' "+intero+", mentre quello reale "+
            "e' "+reale+"\n");
    }
}
```

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int intero=2;
    float reale=3.0;
    printf("Il valore intero "
        "e' %d, mentre quello reale "
        "e' %f\n",intero,reale);
}
```

Nella stringa di formato possono essere usate sequenze di escape come `\n`, `\t`, ...

---

## Input e Output

La funzione simmetrica di printf per prelevare dati da standard input è:

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

La stringa di formato ha lo stesso comportamento di quella presente nella printf.

I parametri successivi sono dei puntatori alla zona di memoria destinata a contenere il valore letto.

```
#include <stdio.h>

int main()
{
    int prova;
    scanf("%d", &prova);
    printf(" Risultato: %d", prova+1);
}
```

Per l'input di semplici caratteri, e' disponibile anche

```
int getchar(void);
```

che restituisce il successivo carattere inserito da tastiera.

```
#include <stdio.h>

int main() /* copia stdin su stdout */
{
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```