



C: dai puntatori ai file

Diego Graziati



Indice:

- pag 3-12: i puntatori;
- pag 13-30: i Record (struct);
- pag 31-48: i File.



I PUNTATORI

I puntatori sono una struttura di dati che immagazzina l'indice di una variabile (anche funzione) al suo interno.

I puntatori sono molto leggeri in memoria, pertanto chiamare i puntatori invece che direttamente le variabili salva molta memoria e rende il programma più veloce.

COME USARLI: inizializzazione ed assegnazione

```
int* puntatore;
```

Tipo del vettore:
indica quale tipo di dato
il puntatore potrà
contenere. Un
puntatore **int** non potrà
mai contenere
l'indirizzo di una
variabile **char**!

“Specificatore”:
l’***** serve al
compilatore per
capire che sta
venendo inizializzato
un puntatore e non
una variabile.

Nome del puntatore.

COME USARLI: inizializzazione ed assegnazione

```
puntatore=&A;
```

Nome del puntatore.

Indirizzo:
la “&” specifica al puntatore, come alla funzione “scanf()”, di andare all’indirizzo in memoria della variabile indicata a destra della “&”. In questo caso l’indirizzo della variabile viene salvato nel puntatore.

Variabile:
è il nome della variabile “puntata”.

COME USARLI: il concetto

Adesso il puntatore “puntatore” possiede l’indirizzo della variabile “A”. Ciò è fondamentale:

potremo usare il valore presente nella variabile “A” senza doverla chiamare (operazione che peserebbe sulla memoria), ma chiamando al suo posto il puntatore “puntatore” (operazione che peserebbe poco sulla memoria).

COME USARLI: il “*” unario

Il “*” unario è importantissimo:

- Nell’inizializzazione serve al compilatore per capire che stiamo inizializzando un puntatore e non una variabile;
- Nelle operazioni e negli output serve per far capire al compilatore che stiamo utilizzando il valore contenuto all’indirizzo puntato dal puntatore.

COME USARLI: le operazioni

Il puntatore può essere utilizzato nelle operazioni, con un occhio di riguardo per le moltiplicazioni, nel caso volessimo usare il valore contenuto nell'indirizzo puntato dal puntatore.



```
C=B*(*puntatore);
```

“*” binario:
segno della moltiplicazione.

“*” unario.

COME USARLI: la stampa a schermo

Nella stampa a schermo l'utilizzo dei puntatori è semplice:

```
printf("%d", *puntatore);
```

Specificatore di stampa:
specifica come deve
essere stampato l'output.

"*" unario:
senza in output verrà
stampato l'indirizzo
contenuto nel puntatore.

Nome del puntatore.

CONCATENAZIONE DI PUNTATORI

È possibile concatenare più puntatori:

un puntatore punterà, quindi, l'indirizzo di un altro puntatore, che a sua volta punterà l'indirizzo di un altro puntatore ecc..., finché non si arriverà al puntatore che punta l'indirizzo di una variabile.

CONCATENAZIONE DI PUNTATORI

Inizializzazione:

```
int** puntatore_2;
```

“*” unario:

ne vanno inseriti tanti quanto è il livello di concatenazione, ovvero il numero di puntatori concatenati creati.

CONCATENAZIONE DI PUNTATORI

Assegnazione: identica all'assegnazione di un normale puntatore, tranne che a destra dell'uguale ci andrà il puntatore puntato;

Stampa a schermo:

```
printf("%d", **puntatore_2);
```



“*” unario:

ne vanno inseriti tanti quanto è il livello di concatenazione, ovvero il numero di puntatori concatenati creati.

Ometterne uno manderà in stampa l'indirizzo dell'ultimo puntatore. Più se ne omettono e più si scala la “scala di concatenazione”, fino a stampare l'indirizzo del puntatore utilizzato per la stampa.



I RECORD (STRUCT)

Un record è una struttura di dati che al suo interno possiede una lista di variabili di tipi di dati.

Questo, similmente alle funzioni, permette di organizzare meglio il codice e renderlo più leggibile.

CREAZIONE

Creare un record è simile al creare una funzione, senza i parametri e con un “;” dopo la “}”:

```
struct ciao{  
    int num;  
    int bello;  
};
```

IMPORTANTISSIMO: non si possono assegnare dei valori alle variabili direttamente nei record!



TYPDEF

Il **typedef** ci permette di modificare il nome di un tipo di dati(**int**, **char**...) o una struttura di dati. Se risulta praticamente inutile nel primo caso, nel secondo caso, invece, risulta quasi indispensabile, in quanto semplifica di molto la vita.

TYPEDEF: uso 1

```
typedef struct libro_struttura libro;
```

Richiamo di **typedef**.

Vecchio nome della
struttura di dati
precedentemente
creata.

Nuovo nome della
struttura di dati.

TYPEDEF: uso 2

Richiamo di **typedef**.

Vecchio nome della struttura di dati, in questo caso creata nello stesso momento in cui la rinominiamo.

```
typedef struct libro_struttura{  
    int pagine;  
    char titolo[50];  
} libro;
```

Variabili di tipi di dati contenute nel Record.

Nuovo nome della struttura di dati.



IL CONCETTO

Ottimo, abbiamo creato un Record “libro_struttura”. Tramite l’inizializzazione di una variabile alla struttura di dati “libro_struttura” potremo accedere alle variabili, tramite “dot notation”, contenute nel Record. Questo varia se abbiamo o meno utilizzato il **typedef**.

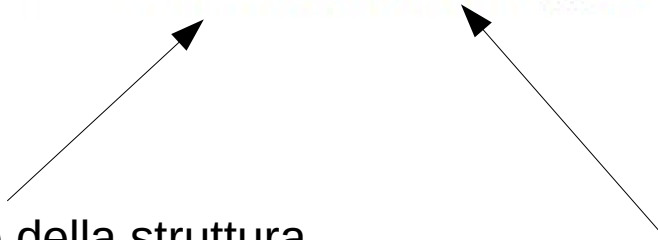
INIZIALIZZAZIONE VARIABILI ALLA STRUTTURA DATI

Con **typedef**:

```
typedef struct libro_struttura libro;
```

```
int main(){  
    libro variabile_1;  
}
```

Nome della struttura
dati dopo il cambio di
nome.



Nome della variabile inizializzata
alla struttura di dati.

INIZIALIZZAZIONE VARIABILI ALLA STRUTTURA DATI

Senza **typedef** (1):

```
struct libro_struttura variabile_1;
```

↑
Specificatore:
fa capire al
compilatore che si
tratti di un Record.

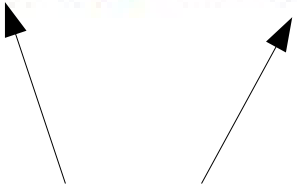
↑
Nome effettivo del Record.

↑
Nome della variabile
inizializzata alla struttura
di dati.

INIZIALIZZAZIONE VARIABILI ALLA STRUTTURA DATI

Senza **typedef** (2):

```
struct libro_struttura{  
    int pagine;  
    char titolo[50];  
} libro_1, libro_2;
```



Nome delle variabili inizializzate al Record.

ATTENZIONE: inizializzare una variabile direttamente alla creazione del Record non preclude l'inizializzazione di altre variabili all'interno del "main()".

INIZIALIZZAZIONE VARIABILI ALLA STRUTTURA DATI: note

Inizializzare una variabile, nel caso dei record, è ben diverso dall'assegnarle un valore!

Non si potrà quindi inizializzare una variabile ad un Record ed assegnarle un valore su un'unica riga, ma si dovranno separare le singole azioni.

Questo perché le variabili inizializzate ad un Record non sono altro che ponti di collegamento alle variabili contenute nel Record, accessibili tramite “dot notation”.

ASSEGNAZIONE: numeri e caratteri

Per assegnare i numeri ed i caratteri ad una variabile nel Record la procedura è questa:

```
variabile_1.pagine=30;
```

Variable inizializzata al Record.

“.”:
la “dot notation” fa capire al compilatore che io sto richiamando la variabile a destra del “.”.

Nome della variabile, presente nel Record, che sto modificando.

Valore assegnato alla variabile nel Record.

ASSEGNAZIONE: Stringhe

Per assegnare una stringa ad un'array di caratteri nel Record bisogna fare così:

```
strcpy(variable_1.titolo, "Il camminatore del Mississippi!");
```

Nome della
funzione.

Nome della
variabile
"ponte".

Nome della
variabile nel
Record che sta
venendo
modificata.

Stringa che deve essere inserita
nella variabile specificata a
sinistra della ",".

PUNTATORI E RECORD: inizializzazione ed assegnazione

Nome del Record

"*" unario.

Nome del puntatore.

```
libro* puntatore_struttura;  
puntatore_struttura=&variabile_1;
```

Nome del puntatore.

"&":
indirizzo della variabile
a destra.

Variabile puntata dal
puntatore.

“->”: COS’È?!

“->” è simile al “*” unario, con l’unica differenza che può essere utilizzato solo nei Record e specifica che il puntatore sta accedendo, tramite l’indirizzo della variabile ponte, al contenuto di una delle variabili nel Record, quest’ultima specificata a destra del “->”.

IMPORTANTISSIMO:

non si può utilizzare il “*” unario insieme a “->”.

OPERAZIONI E RECORD: usare le variabili nei record

```
variabile_2=variabile_1.pagine+3;
```

Operare usando una delle variabili contenute nel Record è identico all'operare normale, con la differenza che bisogna usare la "dot notation" per specificare quale delle variabili all'interno del Record verrà utilizzata nell'operazione.

OPERAZIONI E RECORD: usare le variabili nei record tramite i puntatori

```
variabile_2=puntatore_struttura->pagine+3;
```

Esattamente come abbiamo fatto precedentemente, per operare con un puntatore bisogna ricordarsi semplicemente di utilizzare “->” per specificare quale delle variabili presenti nel Record andremo ad adoperare nell’operazione.

STAMPA A SCHERMO

La stampa a schermo è identica alle normali stampe a schermo, con le dovute precisazioni:

```
printf("%d\n", variabile_1.pagine);  
printf("%d", puntatore_struttura->pagine);
```

INPUT

Anche l'input è identico al normale input da tastiera, sempre però osservando le varie precisazioni:

```
scanf("%d", &variabile_1.pagine);  
fflush(stdin);  
scanf("%s", puntatore_struttura->titolo);
```

Opzionale. Fa la pulizia dell'input da tastiera. Non inficia l'input in sé.

Prende in input la prima stringa scritta dall'utente.



I FILE

Gestire i file in C è semplice, ma può risultare lungo e tedioso, oltre che frustrante, senza i requisiti base, come i puntatori o la gestione delle stringhe.

Nelle prossime slide spiegherò come aprire e chiudere un file, come modificarlo e come ricevere un input.

I PUNTATORI DI TIPO “FILE”

Per poter utilizzare un file bisogna prima di tutto salvare all'interno di un puntatore il percorso al file desiderato.

Naturalmente, queste operazioni sono diverse le une dalle altre, pertanto per ora vedremo come inizializzare un puntatore al tipo di dato “FILE”:

```
FILE* puntatore_file;
```

↑
Tipo di dato.

↑
“*” unario.

↑
Nome del puntatore.

ASSEGNAZIONE: salvare il percorso ad un file nel puntatore di tipo "FILE"

Assegnare il percorso di un file ad un puntatore, ovvero aprire un file, possiede tutta una sua funzione (nota: il file deve essere già presente):

```
puntatore_file=fopen("Esercitazione/input.txt","r");
```

Nome del puntatore.

Nome della
funzione di
apertura file.

Percorso al file,
compreso del
nome del file.

Modalità di apertura
del file.

CHIUSURA DI UN FILE

Dimenticarsi di chiudere un file è spesso considerato un errore, pertanto si consiglia vivamente di scrivere la funzione di chiusura subito dopo quella di apertura, così da non commettere errori (naturalmente, nel mezzo tra una funzione e l'altra si possono scrivere quante funzioni si vogliano):

```
fclose(puntatore_file);
```



Nome della funzione.

Nome del puntatore.

MODALITÀ DI APERTURA

Le modalità di apertura sono 2:

- read (“r”): un file aperto in modalità “read” è un file di input. Il file verrà, quindi, semplicemente letto e non potrà essere modificato;
- write (“w”): un file aperto in modalità “write” è un file di output. Il file verrà, quindi, utilizzato per la stampa e ciò che è già presente nel file verrà sovrascritto.

CONTROLLO CORRETTA APERTURA DEL FILE

Può succedere che il file non venga aperto correttamente. Capire se ciò è successo è semplice:

```
if(puntatore_file==NULL){  
    printf("File non aperto correttamente!");  
    exit(0);  
}
```

Nome del puntatore.

Termina il processo corrente.

NULL= NULL è il valore assegnato ad un puntatore a file nel qual caso questi non possenga nulla al suo interno. Naturalmente, un puntatore a file con valore NULL significa che l'operazione di apertura del file in questione è risultata fallimentare. In questo caso bisogna controllare la sintassi nella funzione di apertura oppure il percorso utilizzato per il file che si vuole aprire.

INPUT DA FILE: `fscanf()`, `fgetc()`, `fgets()`

Queste funzioni di input sono identiche a quelle per il normale input da tastiera.

- `fscanf()` prende un input qualsiasi, tramite lo specificatore (es. “%d”), dal primo spazio allo spazio successivo. Perciò, `fscanf()` non è consigliato per un input di più valori separati da degli spazi, in quanto prenderà solo il primo valore ed ignorerà tutti gli altri;

- `fgetc()` prende in input esclusivamente un singolo carattere;

- `fgets()` prende in input una stringa, compresa di spazi.

INPUT DA FILE: `fscanf()`, `fgetc()`, `fgets()`

NOTA: `fgets()` riempirà l'array di caratteri con tutti i caratteri presenti sulla linea di testo che `fgets()` sta controllando. Infatti, `fgets()` analizza il file linea per linea, pertanto per utilizzare la funzione bisogna usare anche un ciclo "while".

FSCANF() E FGETC(): utilizzo

FSCANF():

Nome della
funzione.

Specificatore di formato:
specifica in quale formato(**char**, **int**...) riceveremo in input.

```
fscanf(puntatore_file, "%d", &A);
```

Nome del puntatore che contiene l'indirizzo al file dal quale preleviamo il valore che ci serve.

Variabile in cui andremo ad immagazzinare il valore ricevuto in input.

FSCANF() E FGETC(): utilizzo

FGETC():

Nome della funzione.

```
B=fgetc(puntatore_file);
```

Variable che immagazzinerà
il valore prelevato dalla
funzione.

Nome del puntatore.

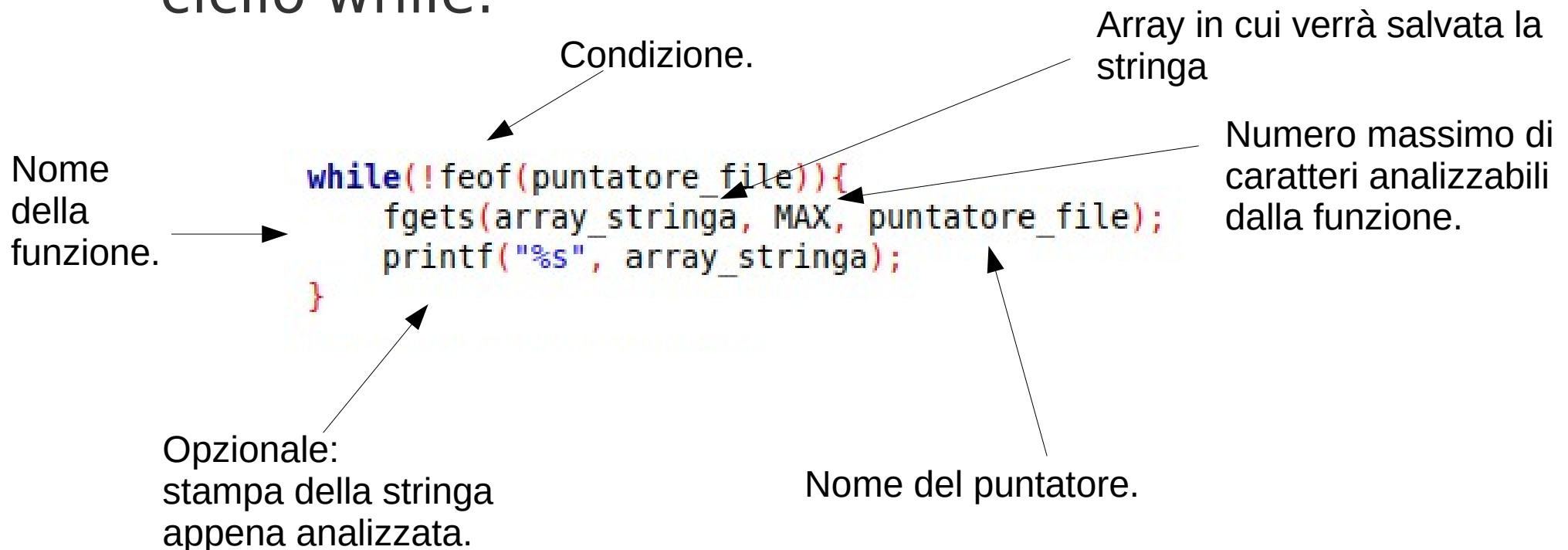
EOF: End Of File

L'End Of File è l'acapo subito sottostante all'ultima riga di testo.

Tramite la funzione “feof(<nome del puntatore>)” potrò sapere se avrò o meno raggiunto la fine del file.

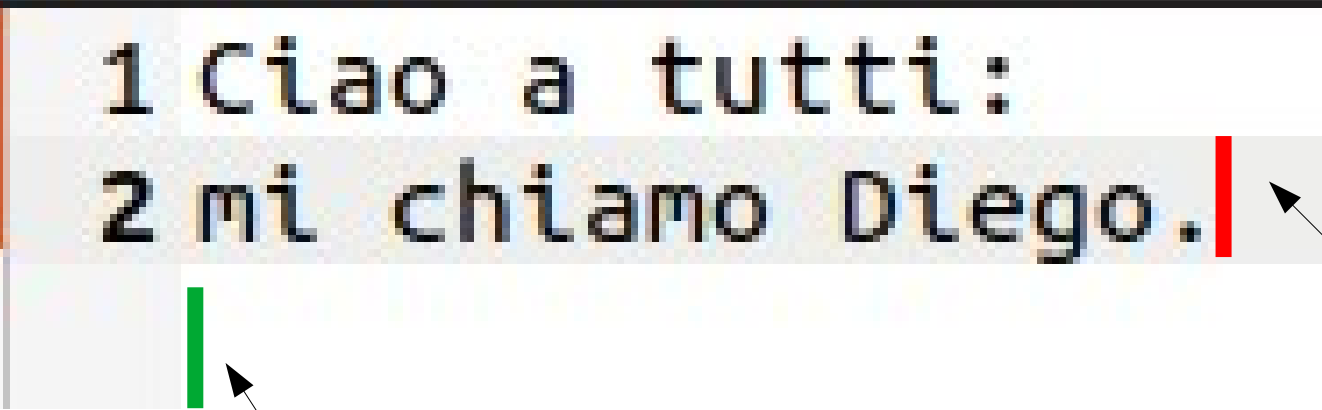
FGETS(): utilizzo(1)

`fgets()`, essendo che possiede al suo interno un array, richiede per forza l'utilizzo di un ciclo `while`:



FGETS(): utilizzo(1). Problema.

A primo acchitto utilizzare l'eof per utilizzare l'"fgets()" potrà sembrare furbo, ma non lo è affatto ed ecco perché:



```
1 Ciao a tutti:
2 mi chiamo Diego.
```

Fine analisi da parte di fgets(): fgets() analizza linea per linea.

Eof:
l'End Of File varia in base alla formattazione del testo. È un dato inaffidabile.

FGETS(): utilizzo(1). Problema.

L'fgets() analizzerà l'ultima riga, per poi provare ad analizzare un'altra riga, per non trovare nulla e quindi rianalizzare la stessa riga precedente.

Questo può portare alla ristampa di una stessa riga, come nell'esempio precedente.

FGETS(): utilizzo(2)

Per ovviare a questo problema si può sfruttare il valore “NULL” dato in output da “fgets()” quando finisce di analizzare l’ultima riga di file:

```
while(fgets(array_stringa, MAX, puntatore_file)!=NULL){  
    printf("%s", array_stringa);  
}
```

Condizione ed analisi.

Stampa a schermo: opzionale.

SSCANF(): la scanf selezionatrice

sscanf() ci permette di analizzare un array già presente, specificando la quantità di elementi, a partire dal primo, che si vogliono esaminare:

```
sscanf(array_stringa, "%s %s", array_parola_1, array_parola_2);
```

Nome della
funzione.

Nome
dell'array da
cui
estraiamo gli
elementi.

Formato e
numero di
elementi da
analizzare.

Array/variabili in cui vengono
salvati gli elementi
analizzati. Ce ne devono
essere tanti quanti sono gli
elementi analizzati!

SSCANF(): la scanf selezionatrice

NOTA:

la `sscanf()` possiede un secondo valore di output, ovvero la quantità effettiva di elementi analizzati nell'array. Il valore sarà minore degli elementi che andrebbero analizzati nel qualcaso l'array avesse meno elementi di quelli che la `sscanf()` dovrebbe analizzare. Nel caso gli elementi nell'array fossero maggiori rispetto a quelli che la `sscanf()` dovrebbe analizzare, allora l'output sarà sempre uguale al numero di elementi che la `sscanf()` dovrebbe analizzare.

FPRINTF(): utilizzo

L'fprintf() è la funzione di output su file. Il file in questione dovrà essere stato precedentemente aperto in modalità "write(w)":

```
fprintf(puntatore_file_2, "%s", array_parola_1);
```

Nome della funzione.

Puntatore che contiene il percorso al file di output.

Formato di stampa.

Array/variabile da stampare.