

Elenco di esercizi C+Unix

Enrico Bini

November 17, 2023

Premessa

Segue un elenco di esercizi assegnati durante l'anno.

Importante Per massimizzare l'apprendimento, si raccomanda di leggere la soluzione soltanto **dopo** aver provato a risolvere l'esercizio autonomamente. Difatti, leggere una soluzione scritta da altri dopo aver provato a risolvere il problema da soli permette di capire meglio la soluzione proposta.

Esercizi C

Es. [es-array-cat-err] Il codice di `es-array-cat-err.c` (disponibile nella cartella “Soluzioni degli esercizi”) dovrebbe concatenare due stringhe. Si corregga il codice affinché vengano effettivamente concatenate le due stringhe `v1` e `v2`. Ai fini di questo esercizio **NON** si possono utilizzare le funzioni di libreria `strcat`, `strncat`, `strlen`, etc. L'obiettivo è confrontarsi con errori tipici (sia di compilazione che run-time) che si possono ottenere, **NON** scrivere il codice di una funzionalità di cui esistono già molte implementazioni.

Es. [es-array-odd-even] Si legga da standard input (con `fgets+strtol`) un array di 7 interi. Si stampino prima tutti gli elementi di indice dispari e poi quelli di indice pari. Se per esempio vengono letti: 11 20 37 45 51 69 75, allora viene stampato: 11 37 51 75 20 45 69.

Es. [es-sum-next] Si legga da standard input (con `fgets+strtol`) un array `v1` di 10 interi. Si costruisca un altro array `v2` in cui:

- il primo elemento è la somma di tutti gli elementi di `v1`
- il secondo elemento è la somma degli elementi di `v1` a partire dal secondo
- ...
- l'ultimo elemento è uguale all'ultimo elemento di `v2`.

Es. [count-char] Si scriva un programma che legga una stringa da `stdin` e, per ogni carattere presente nella stringa, scriva a `stdout` una riga con il numero di occorrenze del carattere nella stringa e il carattere stesso. Per esempio, se la stringa letta è `Ciao a tutti!!`

venga stampato a `stdout`

```
2,  
2,a  
2,i  
1,o  
3,t  
1,C  
1,u  
2,!
```

in ordine a piacere.

Es. [print-after] Si scriva un programma che legge due stringhe di caratteri (**s1** e **s2**) di lunghezza massima di 80 caratteri mediante **fgets**.

Preliminarmente, elimina i caratteri non stampabili da entrambe le stringhe le stringhe **s1** e **s2** scrivendo il byte 0 sul primo byte non stampabile (un byte è stampabile se ha codice ASCII compreso fra 32 e 126). Ricorda: **fgets** memorizza nella stringa anche il carattere “a capo” che deve quindi essere eliminato.

- Se **s2** è contenuta all’interno di **s1**, il programma stampa la parte di **s1** che segue **s2**.
- Se **s2** non è contenuta all’interno di **s1**, non stampa niente.

Per esempio, se le stringhe **s1** e **s2** sono rispettivamente:

Ciao a tutti
ia

allora verrà stampato

o a tutti

Si realizzi tale programma:

1. evitando di includere le funzioni della libreria **string.h**
2. evitando le parentesi quadre per riferire gli elementi di **s1** e **s2**

Es. [get-exponent] Si scriva un programma che:

1. legga un **double** da tastiera,
2. estraiga l’esponente della sua rappresentazione in floating point secondo lo Standard IEEE 754-1985
3. stampi tale esponente in decimale.

Suggerimento: si provi a leggere la memoria dove il numero floating point è memorizzato, come un intero **unsigned long** da cui poi estrarre l’esponente attraverso la manipolazione dei suoi bit.

Es. [binary] Si scriva un programma che legge un intero senza segno da tastiera **stdin** e scrive sul terminale la sua rappresentazione in base 2. Si eviti di usare gli operatori di divisione / e di resto %, preferendo invece gli operatori bitwise e quelli di shift. Si eviti di usare **strtoul(s, NULL, 2)** che fa esattamente questo.

Es. [triangle-star] Scrivere un programma che stampi a video un triangolo rettangolo di ‘*’ la cui base e altezza siano lette da tastiera. Esempio, se vengono inseriti 10 (base) e 4 (altezza), viene stampato quanto segue

```
*
****
*****
*****
```

Es. [caotic-seq] Si consideri la successione generata dal numero n e che calcola il numero successivo come segue:

- se n è pari allora il prossimo numero è la metà di n
- se n è dispari allora il prossimo numero è il triplo più uno.

La sequenza termina quando si raggiunge 1.

Si scriva un programma che, accettato un valore numerico intero N da tastiera, stampi la lunghezza di tutte le sequenze generate per ciascun valore di partenza da 1 a N .

Es. [es-is-number] Scrivere un programma che prenda in input da `stdin` una stringa `string` puntata da un `(char *)` e due valori interi `a` e `b` (con `fgets + strtol`) e che stampi un messaggio che dica se la sottostringa da `string[a]` a `string[b]`, estremi inclusi, rappresenta o no un numero. Non è consentito l'utilizzo di funzioni di libreria.

Es. [es-my-strtoi] Scrivere un programma che prenda in input da `stdin` una stringa puntata da un `(char *)` rappresentante un numero intero in base decimale e ne salvi il valore in intero corrispondente in una variabile, per poi stamparla. Non è consentito l'utilizzo di funzioni di libreria che svolgono già tale funzione.

Es. [es-my-bstrtoi] Modificare il programma scritto per [es-my-strtoi] in modo che prenda in input un intero `b` da 2 a 36 e una stringa `(char *)` rappresentante un numero intero in base `b`, e ne salvi il valore in intero corrispondente in una variabile, per poi stamparla.

Es. [exam-2019.01.28] Implementare la funzione con prototipo

```
int range_of_even(int * nums, int length, int *min, int *max);
```

La funzione ha quattro parametri:

- `nums` è un array di numeri interi;
- `length` è la dimensione di `nums`;
- `min` e `max` sono puntatori usati dalla funzione per restituire degli interi al chiamante.

La funzione deve determinare il valore massimo e minimo **dei valori pari** presenti in `nums`. Se tali limiti esistono allora la funzione deve restituirli tramite i puntatori `min` e `max` al chiamante e restituire 1. Se l'array non contiene alcun numero pari, la funzione deve restituire 0 e i valori in `*min` e `*max` non saranno significativi.

Es. [fibo] Si realizzi la funzione con prototipo

```
int * fibo(int n);
```

la quale alloca e restituisce un array di `n` interi contenente i primi `n` numeri della successione di Fibonacci (https://it.wikipedia.org/wiki/Successione_di_Fibonacci).

Inoltre si scriva la funzione `main` che legge `n` da tastiera, stampa gli elementi di `fibo(n)` e infine dealloca l'array.

Es. [sort-record] Data la seguente `struct`

```
typedef struct {
    char * name;
    int age;
} record;
```

si scriva il corpo delle seguenti funzioni:

```
record * rec_rand_create(int n);
void rec_sort(record * v, int n);
void rec_print(record * v, int n);
void rec_free(record * v, int n);
```

- La funzione `rec_rand_create` alloca e restituisce un array di `n` elementi di tipo `record` in cui
 - ogni stringa `name` contiene caratteri casuali e ha lunghezza casuale fra 1 e `MAX_LEN` (costante del pre-processor opportunamente definita)
 - ogni campo `age` è casuale fra `MIN_AGE` e `MAX_AGE`

Si veda `man 3 rand` per la generazione di numeri casuali

- la funzione `rec_sort` ordina gli elementi dell'array `v` di lunghezza `n` secondo il campo `age` crescente
- la funzione `rec_print` stampa l'array
- la funzione `rec_free` dealloca la struttura dati

Si realizzi quindi un `main` che testi le tre funzioni.

Es. [list] Si estenda il file `test-list.c` aggiungendo le seguenti funzioni:

1. la funzione

```
list list_insert_ordered(list p, int val);
```

che riceve in input una lista ordinata per valori crescenti puntata da `p` e inserisce il nuovo elemento `val` nella lista mantenendo l'ordinamento;

2. la funzione

```
list list_cat(list before, list after);
```

che riceve in input due liste `before` e `after` e restituisce in uscita la lista `before` a cui è stata aggiunta in coda la lista `after`

3. la funzione

```
list list_insert_tail(list p, int val);
```

che inserisce l'elemento `val` in coda alla lista puntata da `p` e ritorna la lista modificata

Es. [list-more] A partire dal file `es-list.c` realizzato nell'esercizio [list], si realizzino anche le seguenti funzioni (ispirate a esercizi dello scritto):

1. la funzione

```
list list_delete_if(list head, int to_delete);
```

la quale cancella e dealloca il primo nodo della lista il cui valore del campo `value` è uguale al parametro `to_delete`. La funzione restituisce la lista così modificata.

2. la funzione

```
list list_delete_odd(list head);
```

la quale rimuova dalla lista ogni elemento in posizione dispari (il primo, il terzo, etc.). La funzione restituisce la lista così modificata.

3. la funzione

```
list list_cut_below(list head, int cut_value);
```

la quale rimuova dalla lista ogni elemento che abbia valore inferiore al valore `cut_value` passato come parametro.

4. la funzione

```
list list_dup(list head);
```

la quale ritorna una copia dalla lista (copia di ogni elemento).

Es. [file-csv] Scrivere un programma che legge il nome di un file come parametro da riga di comando. Il programma legge le righe del file `argv[1]` in formato CSV (comma separated values). Inoltre calcola i valori minimo, massimo e medio per ciascuna colonna dei dati letti dal file. Per questo calcolo, si utilizzi una funzione con prototipo

```
void data_column_process(int * data, int length, int * min, int * max, double * avg);
```

che prende in input un array puntato da `data` di lunghezza `length` e restituisce i valori minimo, massimo e medio attraverso parametri `min`, `max` e `avg` passati per riferimento dell'array.

Per esempio, se il contenuto del file è:

```
1, 2
3, 1
4, -3
```

allora il risultato è

- colonna 1: min=1, max=4, avg=2.666666
- colonna 2: min=-3, max=2, avg=0.

Es. [file-shuffle-rows] Scrivere un programma che legge il nome di due file da riga di comando. Il programma scrive nel file `argv[2]` le righe del file `argv[1]` in ordine casuale (si veda `man 3 rand` per numeri casuali). Una riga si identifica come una sequenza di byte terminata dal carattere “a capo”.

Es. [list-module] A partire dal codice dell'esercizio [list-more], si realizzi un modulo “list”, ovvero

- l'header file `list-module.h`,
- il file `list-module.c` con il corpo delle funzioni,
- un file `test-list-module.c` che contenga la funzione `main` che testa il funzionamento del modulo,
- un `Makefile` per tutti i target di interesse.

Esercizi Unix

Es. [kids-write-file] Si scriva un programma che legga da riga di comando 3 command-line arguments:

1. il primo è un nome di file da aprire in scrittura
2. il secondo è un numero `n_kids` di processi figlio da creare
3. il terzo è un numero `n_writes` di scritture che ogni processo figlio deve fare

Il programma, apre il file (con nome passato a riga di comando) in scrittura e crea `n_kids` processi figlio. Ogni processo figlio scrive `n_writes` volte nel file il proprio PID ed il PID del parent sulla stessa riga. Si provi a eseguire con:

```
./es-kids-write-file out.csv 10 1000
```

Investigare:

- i valori minimi di `n_kids` e `n_writes` per cui i PID stampati da un figlio sono interrotti da altri PID
- come la rimozione della bufferizzazione delle scritture possa far comparire stringhe inattese (dovute all'intrleaving delle stampe dei processi figlio)

Es. [sum-rand-kids] Sia `NUM_KIDS` una macro definita con `#define` (di valore 20, per esempio). Si scriva un programma in cui il processo padre genera `NUM_KIDS` processi figlio. Ogni processo figlio genera casualmente un numero intero n da 1 a 6, stampa il suo PID e n , ed esce con exit status uguale al numero casuale estratto (`man 3 rand` per la generazione di numeri interi casuali. Si usi `srand(getpid())` per l'inizializzazione del seed del random. Perché la soluzione di stackoverflow `srand(time(NULL))` non funziona?). Il processo padre attende la terminazione di tutti i processi figli (con `wait`) e stampa la somma dei valori di uscita dei propri figli.

Suggerimento Si provi a scrivere il codice a partire dall'esempio `test-fork-wait.c`, nel seguente modo:

1. modificare il codice dei processi figli come richiesto dall'esercizio
2. modificare il codice in cui il padre fa le `wait(&status)` sui figli. In questo caso il padre deve estrarre da `status` l'exit status dei figli attraverso macro `WEXITSTATUS(status)` (si veda `test-fork-waitpid.c` o le slide per il suo utilizzo) e farci quanto richiesto dall'esercizio.

Es. [kids-count-or-die] Si scriva il codice di un eseguibile di nome `count-loop` che legga da riga di comando 2 command-line arguments:

- il primo è una durata `timeout`
- il secondo è un numero `n_iter` di iterazioni da svolgere in un ciclo `for` vuoto: `for(int i=0; i<n_iter; i++);`

Il programma `count-loop`, imposta un timer di durata `timeout` con `alarm`. Se riesce a terminare tutte le iterazioni `n_iter`, termina normalmente. Altrimenti, il segnale `SIGALRM` determina la sua terminazione.

Si scriva quindi il programma `kids-count-or-die` il quale legga da riga di comando 3 command-line arguments:

1. il primo è una durata `timeout`
2. il secondo è un numero `n_iter` di iterazioni
3. il terzo è un numero `n_kids` di processi figlio da creare.

Il programma crea `n_kids` processi figlio che eseguono `count-loop` con `execve` a cui vengono passati gli stessi parametri `timeout` e `n_iter`.

A questo punto aspetta la terminazione dei figli:

- se il figlio `count-loop` termina normalmente, bene,
- altrimenti, se il figlio termina a causa del segnale ricevuto dal proprio timer, il padre genera un nuovo processo figlio con stesso `timeout` e un valore `n_iter` uguale alla metà di quello usato quando il processo era stato creato.

Quando tutti i processi sono terminati, il processo padre stampa, per ogni processo figlio, il valore `n_iter` con cui il processo figlio è riuscito a terminare prima del timer.

Es. [guess-number] Scrivere un programma che realizzi un semplice gioco. Il programma seleziona un numero casuale tra 0 e `argv[1]` (il primo argomento passato a riga di comando), e l'utente deve indovinare questo numero. Per fare questo, viene realizzato un ciclo in cui il programma legge da tastiera un numero inserito dall'utente:

- se il numero è stato indovinato, il gioco finisce;
- se il numero è maggiore o minore di quello estratto casualmente, viene stampato a video la scritta "maggiore" o "minore", rispettivamente.

Se il giocatore non indovina entro `argv[2]` secondi (da realizzare con `alarm` e gestendo il segnale `SIGALRM`), il programma stampa a video "tempo scaduto", ed esce.

Es. [loop-zero] Scrivere un programma che genera `argv[1]` processi figlio. Ogni processo, compreso il processo padre incrementa per sempre una propria variabile `var` da zero fino `argv[2]` come segue

$$0, 1, 2, \dots, \text{argv}[2] - 1, \text{argv}[2], 0, 1, 2, \dots$$

(si ricorda che sopra per `argv[i]` si intende l'intero scritto nella stringa, non la stringa).

- Ogni volta che `var` è uguale a zero, il processo figlio invia `SIGUSR1` al processo padre.
- Quando il processo padre riceve tale segnale `SIGUSR1`, se la propria variabile `var` è minore o uguale a `argv[3]`, invia un segnale di terminazione ad un processo figlio scelto a caso.
- Il processo padre termina quando anche il suo ultimo figlio ha terminato.

Es. [string-kids] Si scriva il codice di un eseguibile di nome `char-loop` che:

1. legge da riga di comando un argomento;
2. inizializza una variabile `unsigned char c` al primo carattere del primo argomento passato a riga di comando;
3. incrementa `c` per sempre (forever loop). Quando `c` supera il valore del codice ASCII 126 (decimale), allora viene resettata al valore di 33 (decimale);
4. quando viene premuto `Ctrl+C` (corrispondente al segnale `SIGINT`), il programma termina restituendo come exit status il valore corrente di `c`.

Si scriva quindi un programma `string-kids` che:

1. crea `argv[1]` processi figlio (con la system call `fork()`) che eseguono `char-loop` (con la system call `execve(...)`);
2. invia a tutti loro un segnale `SIGINT` (con la system call `kill(...)`);
3. scrive il carattere corrispondente all'exit status (recuperato con la system call `wait(...)`) su una stringa, che viene poi stampata.

Provare a inserire una `sleep(1)` prima dell'invio dei segnali e cerca di capire cosa cambia e perché. (Nota: `sleep(1)` NON è una primitiva di sincronizzazione fra processi. A questo punto del programma, però, è l'unica cosa che si può utilizzare al posto dei semafori.)

Es. [string-kids-alarm] Un processo padre crea `argv[1]` processi “`char-loop`” come descritti nell'esercizio `[string-kids]`. Quando ha terminato la creazione dei processi figlio, attende la ricezione di un segnale `SIGALRM` dopo un secondo (mediante chiamata `alarm(1)`) e si mette in attesa della terminazione dei figli (con `wait(...)/waitpid(...)`).

- Quando riceve il segnale `SIGALRM`, invia il segnale `SIGINT` ad uno dei suoi figli scelto a caso (si ricorda che la ricezione di `SIGINT` da parte di un processo “`char-loop`” determina la sua terminazione con una `exit(...)`);
- Il padre quindi
 1. legge l'exit status del figlio terminato,
 2. stampa la stringa con i caratteri corrispondenti agli exit status di tutti i figli terminati fino ad ora, e
 - (a) se la somma degli exit status dei figli terminati fino ad ora è pari a zero (modulo 256), termina i figli ancora vivi, effettua le `wait` sul loro exit status e termina,
 - (b) altrimenti
 - i. crea un nuovo figlio di tipo “`char-loop`” che rimpiazza (con `execve(...)`) quello terminato
 - ii. richiede la ricezione di un nuovo `SIGALRM` fra un secondo con `alarm(1)`
 - iii. si mette nuovamente in attesa della terminazione di un figlio.

Attenzione Si gestisca correttamente il caso in cui il processo padre viene svegliato dalla ricezione del segnale `SIGALRM` quando in attesa sulla `wait`.

Es. [hot-potato] Un processo padre crea `argv[1]` processi e `argv[1]` pipe. Il processo figlio i -esimo legge dalla read end della pipe i -esima e scrive sulla write end della pipe $(i + 1)$ -esima (oppure la pipe 0-esima, nel caso dell'ultimo processo figlio). Si realizza quindi una catena fra processi figlio.

Quando un processo legge dalla pipe:

- se legge zero allora scrive zero anche nella pipe di scrittura e termina,
- altrimenti decrementa il valore letto e lo scrive nella pipe di scrittura.

Il processo padre avvia il “passaggio” del numero scrivendo un (`int`) a caso fra 1 e `argv[2]` nella prima pipe.

Es. [hot-potato-msg] Un processo padre crea `argv[1]` processi figlio e una coda di messaggi. Il messaggio, oltre al tipo `long` obbligatorio, contiene un `int`.

Con i che va da 1 a `argv[1]`, il processo figlio i -esimo riceve dalla coda un messaggio di tipo i , e spedisce in coda un messaggio di tipo $i + 1$ (oppure un messaggio di tipo 1, nel caso dell'ultimo processo figlio). Sia n l'intero contenuto nel messaggio ricevuto. Allora, il contenuto del messaggio spedito è:

- n con probabilità 0.5, oppure
- $n - 1$ con probabilità 0.5.

Si realizza quindi una catena fra processi figlio.

Il processo che riceve il messaggio con valore 0 termina. Gli altri processi terminano dopo che per un secondo non hanno ricevuto alcun messaggio.

Il processo padre avvia il “passaggio” inviando un messaggio di tipo 1 contenente un intero casuale fra 1 e `argv[2]`. Infine attende la terminazione dei figli, stampa il PID del figlio terminato a causa della ricezione del messaggio con valore 0 ed elimina la coda.

Es. [schedule-kids] Si scriva il codice di un eseguibile di nome `count-print` che effettua un numero `n_iter` di stampe, con `n_iter` estratto casualmente fra 1 e `argv[1]`. La stampa contiene il PID e l'indice dell'iterazione. Prima dell'estrazione casuale, si deve invocare `srand(getpid())` per inizializzare il seed della generazione di numeri casuali.

Si scriva quindi il programma `schedule-kids` il quale legga da riga di comando un numero `n_kids` di processi figlio da creare. Prima della creazione dei figli, il programma estrae una sequenza casuale `sched` di indici da 0 a `n_kids-1` (se siete a corto di fantasia su come farlo, potete ispirarvi al modo in cui vengono rimescolate le righe in `es-file-shuffle-rows`).

Il programma crea `n_kids` processi figlio che eseguono `count-print` con `execve` a cui viene passato `argv[1]`, in modo che `count-print` possa decidere quante stampe effettuare (come descritto sopra).

Attraverso l'uso di **semafori**, il processo padre schedula l'ordine con cui i processi figlio `count-print` effettuano le loro stampe. Se per esempio `n_kids` è uguale a 3 e la sequenza `sched` è `sched[0]=2, sched[1]=0, sched[2]=1`, allora le stampe saranno nel seguente ordine:

1. PID figlio 2, stampa iterazione 0
2. PID figlio 0, stampa iterazione 0
3. PID figlio 1, stampa iterazione 0
4. PID figlio 2, stampa iterazione 1
5. PID figlio 0, stampa iterazione 1
6. PID figlio 1, stampa iterazione 1
7. PID figlio 2, stampa iterazione 2
8. PID figlio 0, stampa iterazione 2
9. PID figlio 1, stampa iterazione 2
10. ...

Attenzione Il numero di iterazioni compiute da ogni processo figlio è diverso. Quindi, quando un figlio ha terminato l'esecuzione e, secondo l'ordine `sched`, toccherebbe a lui stampare, viene semplicemente saltato. Il programma termina quando non ci sono più figli.

Suggerimenti Visto che il processo padre si dovrà sincronizzare sia con la fine della stampa dei figli in ogni iterazione, che con la loro terminazione, una possibile soluzione potrebbe essere quella di sincronizzarsi con la terminazione dei figli in modo asincrono, con il segnale `SIGCHLD` ricevuto dal padre ogni volta che un figlio termina.