Calcolatori Elettronici—Prova di programmazione in C
24 gennaio 2022

Si vuole realizzare una variante interattiva del *gioco dell'impiccato*, a giocatore singolo. In questo gioco, il giocatore deve indovinare una parola, inserendo un carattere per volta, avendo a disposizione un numero limitato di tentativi. Per ciascuna parola, è possibile fornire dei suggerimenti, ossia un numero arbitrario di caratteri della parola già inseriti.
Il gioco funziona a *turni*: ogni turno richiede al giocatore di indovinare una parola differente. Il numero di parole in una partita non è definito a priori. Ogni volta che viene inserito un carattere non presente nella parola, il giocatore perde una "vita". Se viene indovinata la parola del turno, si guadagnano un certo numero di "vite".

Il gioco è quindi configurabile: il programma accetta in input (come unico parametro obbligatorio) il percorso ad un file di testo nel formato:

`pAroLa;`*n*

in cui ogni riga rappresenta un turno. In ciascuna riga, prima del carattere '`;`' si trova la parola da indovinare. Dopo il '`;`' si trova il numero di tentativi aggiuntivi ("vite") che si guadagnano qualora la parola sia stata indovinata (un numero intero positivo). Si faccia riferimento al file `config.txt` fornito per un esempio di file di configurazione.
Le parole da indovinare, nel file di configurazione, hanno alcuni caratteri maiuscoli: questi sono i suggerimenti, ossia i caratteri che vengono automaticamente inseriti e resi visibili all'inizio del turno.

Il programma da realizzare quindi, deve seguire i seguenti passi:

1. Il file di configurazione viene analizzato ed i turni di gioco vengono caricati in memoria (in una struttura dati a scelta dello studente);
2. Viene mostrata la parola, sostituendo ai caratteri in minuscolo dei trattini bassi (rispetto all'esempio precedente, verrà mostrato: "`_ A _ _ L _`");
3. Il giocatore inserisce un input da tastiera: se vengono inseriti più caratteri, soltanto il primo sarà considerato come lettera da verificare;
4. Se la lettera inserita è tra quelle da indovinare, viene aggiunta all'insieme delle lettere indovinate (ad esempio, se il giocatore digita '`p`' o '`P`', verrà mostrato "`P A _ _ L _`");
5. Se la lettera inserita non è tra quelle da indovinare, o se è una lettera già indovinata in precedenza, si perde un punto vita;
6. Se i punti vita sono terminati, il giocatore perde la partita;
7. Se la parola è stata indovinata, viene aggiunto *n* ai punti vita e si passa al turno successivo (ripartendo dal punto 2);
8. Quando sono state indovinate tutte le parole, il giocatore ha vinto la partita.

Per semplicità, si assuma che il contenuto del file di configurazione è ben formato, ovvero senza errori: non è necessario effettuare alcun controllo sulla correttezza del formato del file di configurazione.

Soluzione

Il programma da realizzare si compone di due parti principali: l'analisi del file di input con relativa rappresentazione in memoria dei turni ed il vero e proprio gioco. Il fatto che il programma deve necessariamente ricevere il percorso ad un file di configurazione deve essere esplicitamente verificato. Possiamo quindi organizzare la funzione main() come segue.

```c
int main(int argc, char **argv)
{
	// Verifica se è stato passato esattamente un argomento
	if(argc != 2) {
		fprintf(stderr, "Usage: %s [config_file]\n", argv[0]);
		exit(EXIT_FAILURE);
	}
	// Tenta di caricare la configurazione del gioco
	if(!load_game(argv[1])) {
		fprintf(stderr, "Unable to load game configuration file: %s\n", argv[1]);
		exit(EXIT_FAILURE);
	}
	// Avvia il gioco
	game_loop();
	return 0;
}
```

Per quanto riguarda il caricamento della configurazione del gioco, occorre notare che la specifica richiede esplicitamente di poter gestire un numero arbitrario di parole da indovinare. Inoltre, ciascuna parola è associata ad un numero di "vite" guadagnate differente. Poiché il programma deve implementare una singola partita (seppur organizzata in turni), possiamo prevedere una struttura dati per la descrizione di un singolo turno. I vari turni possono essere organizzati in una lista singolarmente concatenata. La partita può essere descritta utilizzando una sola variabile globale (la testa della lista). Dobbiamo quindi definire una struttura dati per rappresentare il nodo della lista di parole da indovinare.
Di seguito si propone una *possibile* soluzione all'esercizio proposto.
Occorre prestare attenzione al fatto che la lunghezza di una parola non è nota a priori: per gestire questo scenario è possibile utilizzare un vettore flessibile. Poiché trattiamo stringhe, non è necessario mantenere un membro che descriva la lunghezza della stringa, poiché questa è sempre terminata da un byte nullo. Si ottiene quindi la seguente variabile globale:

```
struct match {
      struct match *next;
      unsigned points;
      char word[];
} *game_config = NULL;
```

La funzione di caricamento della lista deve leggere il file una riga alla volta. Per semplicità, in questa implementazione si assume che una riga non sia più lunga di 256 caratteri. Se si vuole rilassare questa assunzione è possibile utilizzare memoria dinamica al posto del buffer di linea allocato su stack.
Il caricamento in memoria del file di configurazione legge quindi il file una riga alla volta ed alloca un nodo della lista che viene inserito in coda. La dimensione del vettore flessibile può essere sovrastimata considerando tutta la lunghezza della riga (vi è uno spreco di byte legato alla lunghezza del numero di vite aggiuntive in caratteri, ma è un costo tollerabile al fine di ridurre la complessità dell'operazione di caricamento). Per determinare la posizione del carattere ; all'intero della riga si può utilizzare la funzione di libreria standard `strcspn()` vista a lezione. Un'alternativa non discussa a lezione è l'utilizzo della funzione `strtok()`, che rende comunque più complesso il codice per lo specifico obiettivo. La posizione del carattere ; può essere utilizzata sia per inserire il terminatore di stringa nel vettore della parola, sia per effettuare la conversione da stringa a intero del numero di vite che si ottengono (si noti che in questo caso si è utilizzato `atoi()` poiché la specifica sottolineava il fatto che il file di configurazione è ben formato, pertanto non è necessario effettuare alcun controllo di errore).

```
static bool load_game(char *path)
{
      FILE *file;
      char line[256]; // Data la struttura del file descritta, possiamo assumere che una riga entri in 256 byte

      // Provo ad aprire il file in lettura e verifico che l'apertura sia andata a buon fine
      file = fopen(path, "r");
      if(file == NULL)
            return false;

      // Leggo una linea alla volta e inizializzo i nodi nella lista
      struct match **curr = &game_config;
      while(fgets(line, sizeof(line), file)) {
            *curr = malloc(sizeof(struct match) + strlen(line));
            strcpy((*curr)->word, line);
            unsigned short delim = strcspn((*curr)->word, ";");
            (*curr)->word[delim] = '\0';
            (*curr)->points = atoi(&(*curr)->word[delim + 1]);
            (*curr)->next = NULL;
            curr = &(*curr)->next;
      }
      fclose(file);

      return true;
}
```

Possiamo ora discutere la logica di gioco. La lista di parole da indovinare (che è stata caricata precedentemente) viene scandita un nodo alla volta. Per ciascuna parola viene eseguito un turno di gioco (implementato in seguito). Se il turno ha successo, ovvero il giocatore non termina le "vite", si procede al turno successivo. All'uscita dal ciclo che implementa il gioco, se l'ultimo turno è andato a buon fine, il giocatore ha vinto la partita, altrimenti esso ha perso.
Si noti il ciclo finale per liberare tutta la memoria utilizzata in caso di partita persa.

```
static void game_loop(void)
{
	struct match *curr = game_config;
	struct match *to_free;
	int life = INITIAL_LIFE;
	bool alive = true;

	while(curr) {
		if(!(alive = turn(curr, &life))) {
			break;
		}

		// Questa parola è stata indovinata
		to_free = curr;
		curr = curr->next;
		free(to_free);
	}

	clrscr();

	if(!alive) {
		while(curr) {
			to_free = curr;
			curr = curr->next;
			free(to_free);
		}

		printf("GAME OVER!\n");
	} else {
		printf("YOU WON!\n");
	}
}
```

Il singolo turno richiede di mostrare la parola (nascondendo le lettere non ancora indovinate) ed accettare un carattere in input. Nella soluzione proposta vengono anche mostrate il numero di vite rimaste.
Queste operazioni devono essere ripetute fintantoché il giocatore non indovina la parola, oppure fino all'esaurirsi del numero di vite.
Per mantenere lo stato della partita, viene utilizzato lo stesso buffer in cui era stata caricata la parola: se viene indovinata una lettera, questa viene convertita in maiuscolo: in questo modo verrà automaticamente mostrata all'iterazione successiva.
Occorre acquisire da tastiera un carattere per volta. Per questo viene utilizzata la funzione di libreria `getchar()`. Tuttavia, la funzione in questione restituisce un carattere per volta, ma l'utente potrebbe averne inserito più di uno (o nessuno). Occorre quindi gestire esplicitamente questo caso: se il giocatore ha inserito almeno un carattere, in realtà ce ne sarà almeno un altro su standard input (il ritorno a capo). Se invece non ha inserito alcun carattere, `getchar()` restituirà esattamente il ritorno a capo. Se il carattere restituito è quindi diverso dal ritorno a capo (ossia: è stato inserito almeno un carattere da tastiera), allora occorre svuotare lo standard input: questo può essere fatto chiamando iterativamente `getchar()`, fino ad incontrare il carattere di ritorno a capo.

```
static bool turn(struct match *curr, int *life)
{
	while(true) {
		bool correct = false;
		int to_find = 0;

		clrscr();

		// Stampa lo stato corrente del turno
		printf("LIFE: %d\n\n", *life);
		char *word = curr->word;
		while(*word) {
			char c = *word++;
			if(islower(c))
				c = '_';
			printf("%c ", c);
		}
		printf("\n");

		// Acquisisci un carattere da standard input e svuota il buffer se sono stati inseriti più caratteri
		char input = (char)tolower((char)getchar());
```

```
            if(input != '\n')
                  while(getchar() != '\n');

            // Verifica se il carattere inserito è nella parola e segnalo come trovato.
            // Contemporaneamente, conta il numero di caratteri che devono ancora essere indovinati.
            word = curr->word;
            while(*word) {
                  if(*word == input) {
                        correct = true;
                        *word = (char)toupper(*word);
                  }
                  to_find += islower(*word) ? 1 : 0;
                  word++;
            }

            if(to_find == 0)
                  return true;
            if(!correct)
                  (*life)--;
            if(*life == 0)
                  return false;
      }
}
```

Alcune considerazioni finali. Vengono riportati gli header necessari ad utilizzare le funzioni di libreria presenti nella soluzione qui sopra. Inoltre, viene definita una macro per parametrizzare a tempo di compilazione il numero di vite iniziali del giocatore.
Viene anche definita una macro `clrscr()` per cancellare lo schermo, per dare l'effetto di un "gioco" a riga di comando (anche se questo non era stato richiesto dalla specifica). Questa macro utilizza degli *ANSI escape code*, ossia delle sequenze di caratteri introdotti dalla telescrivente VT100 del 1978 e ad oggi standardizzati (per quanto il supporto su sistemi non-Unix non è sempre stato stabile o presente).

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>

#define INITIAL_LIFE 10
#define clrscr() printf("\e[1;1H\e[2J")
```