Calcolatori Elettronici—Prova di programmazione in C
02 settembre 2021

Esercizio 1
Si chiede di realizzare una struttura dati in C, secondo lo schema fornito nel file `esercizio1.c`, che implementi una stringa dinamica.
Una stringa dinamica è una stringa che può essere utilizzata in una funzione di `append()`, tale per cui diventa possibile concatenare più stringhe tra di loro durante l'esecuzione del programma.

Soluzione

La stringa dinamica la cui implementazione è stata richiesta ricorda molto il vettore dinamico. È possibile quindi prevedere una struct in cui siano definiti: un puntatore ad un buffer di memoria utilizzato per memorizzare la vera e propria stringa; un contatore che tiene traccia dei caratteri totali disponibili all'interno del buffer di memoria; un contatore che tiene traccia di quanti caratteri sono stati effettivamente utilizzati fino a quel momento. La funzione di inizializzazione richiesta può quindi preallocare un buffer di dimensione richiesta, inizializzare il buffer come stringa (inserendo il terminatore), ed impostare i contatori ai valori corretti:

```
struct dynstr {
	char *buff; // The current string
	size_t size; // The total number of available characters
	size_t used; // The current number of used characters
};

void dynstr_init(struct dynstr *ds, size_t len)
{
	assert(len > 0);
	ds->buff = malloc(len);
	ds->buff[0] = '\0';
	ds->size = len;
	ds->used = 1;

}
```

La funzione di "append" richiesta deve concatenare la stringa conservata nel buffer con il suffisso passato come argomento. Così come nel caso del vettore dinamico, è necessario verificare se la memoria disponibile nel buffer è sufficiente a contenere la nuova stringa. In caso negativo è necessario effettuare una riallocazione. A differenza dell'esempio "classico" del vettore dinamico, però, non è sufficiente raddoppiare lo spazio: infatti, il suffisso da concatenare può avere dimensione arbitraria e non entrare in un buffer raddoppiato una sola volta. È quindi necessario prendere in considerazione anche questo caso.

```
void append(struct dynstr *ds, const char *suffix)
{
	size_t len = strlen(suffix);

	if(ds->used + len >= ds->size) {
		do {
			ds->size *= 2;
		} while(ds->used + len >= ds->size);
		ds->buff = realloc(ds->buff, ds->size);
	}

	strncat(ds->buff, suffix, len);
	ds->used += len;
}
```

La funzione di stampa è immediata, poiché serve soltanto a rendere opaco il buffer contenuto all'interno della struttura dynstr:

```
void print(struct dynstr *ds)
{
	printf("%s", ds->buff);
}
```

Con questa implementazione, la funzione main() fornita ha a disposizione tutte le funzioni necessarie al suo funzionamento:

```
int main(void)
{
	struct dynstr ds;
	dynstr_init(&ds, 8);

	append(&ds, "Hello ");
	append(&ds, "world, ");
	append(&ds, "this ");
	append(&ds, "is ");
	append(&ds, "my ");
	append(&ds, "program");

	print(&ds); // Expected print: "Hello world, this is my program"

	return 0;
}
```

Esercizio 2
Data una struttura dati lista, secondo lo schema riportato nel file `esercizio2.c`, implementare una funzione che ordini i nodi della lista per chiavi crescenti.
Se necessario, è possibile popolare (per motivi di test) la lista con nodi arbitrari all'interno della funzione `main()`.

Soluzione

Un qualsiasi algoritmo di ordinamento è valido per risolvere questo esercizio. Nella soluzione proposta, viene fornita un'implementazione del bubble sort. La difficoltà di questo esercizio sta nella gestione della lista doppiamente concatenata, che richiede di aggiornare 6 puntatori differenti per effettuare uno scambio di elementi. L'utilizzo (suggerito nel codice fornito) di un nodo sentinella consente di semplificare lo scambio, andando a controllare un solo corner case. Inoltre, utilizzando i doppi puntatori, non è necessario utilizzare alcuna variabile d'appoggio per effettuare lo scambio, avendo cura di effettuare l'aggiornamento dei puntatori nell'ordine corretto.
Nella soluzione qui proposta viene anche preinizializzata (in forma cablata) una lista non ordinata ed una funzione di stampa, non strettamente richieste dall'esercizio in sede d'esame.

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

struct node {
	int key;
	struct node *next;
	struct node *prev;
};
```

```c
// La seguente implementazione è quella di un bubble sort
void sort(struct node **head)
{
	bool swapped;
	struct node **ptr;

	if(!*head)
		return;

	do {
		swapped = false;
		ptr = head;

		while ((*ptr)->next) {
			if ((*ptr)->key > (*ptr)->next->key) {
				if((*ptr)->next->next)
					(*ptr)->next->next->prev = *ptr;
				(*ptr)->next->prev = (*ptr)->prev;
				(*ptr)->prev = (*ptr)->next;
				(*ptr)->next = (*ptr)->next->next;
				(*ptr)->prev->next = *ptr;
				*ptr = (*ptr)->prev;
				swapped = true;
			}
			ptr = &(*ptr)->next;
		}
	} while(swapped);
}

struct node n  = {30, NULL, NULL};
struct node n1 = {10, NULL, &n};
struct node n2 = {11, NULL, &n1};
struct node n3 = {15, NULL, &n2};
struct node n4 = {9, NULL, &n3};
struct node n5 = {26, NULL, &n4};
struct node n6 = {12, NULL, &n5};
struct node n7 = {0, NULL, &n6};
struct node *head = &n;

void print(void)
{
	struct node *n = head;
	while(n) {
		printf("%d ", n->key);
		n = n->next;
	}
	puts("");
}

int main(void)
{
	n.next = &n1;
	n1.next = &n2;
	n2.next = &n3;
	n3.next = &n4;
	n4.next = &n5;
	n5.next = &n6;
	n6.next = &n7;

	print();
	sort(&head);
	print();

	return 0;
}
```

Esercizio 3
Si richiede di realizzare una funzione in C che permetta di modificare un vettore di interi andando a rimuovere tutti gli elementi dispari.
La rimozione deve essere realizzata *in place*, ovverosia non è consentito effettuare copie del vettore. Il contatore di elementi "validi" all'interno del vettore, preinizializzato all'interno del programma, deve essere aggiornato durante l'eliminazione degli elementi.

Soluzione

Per risolvere questo esercizio è necessario:
1. Individuare nel vettore tutti e soli gli elementi dispari;
2. Rimuovere un elemento dispari spostando tutti i successivi e decrementando il contatore degli elementi validi.

Per quanto riguarda il punto 2, è necessario tenere a mente che se si effettua lo spostamento tramite funzioni di libreria, non è possibile utilizzare la memcpy() perché sorgente e destinazione si sovrappongono in memoria: è necessario quindi utilizzare la funzione memmove(). Effettuare la traslazione a mano è un'operazione nettamente meno efficiente (le funzioni di libreria sono ottimizzate).

```c
#include <stddef.h>
#include <string.h>
#include <stdio.h>

int vector[] = {21, 75, 73, 97, 98, 48,  2, 66, 52, 47, 44,
                6, 40, 41, 50, 24, 29,  1, 25, 46, 14, 42,
               93, 71, 96, 61, 58, 10, 68, 20, 95, 32, 16};
size_t counter = sizeof(vector)/sizeof(int);

void no_odds(int *v)
{
	size_t size = counter;
	for(size_t i = 0; i < size; i++) {
		if(v[i] % 2 == 1) {
			memmove(&v[i], &v[i + 1], (size - i) * sizeof(int));
			counter--;
			i--;
		}
	}
}

void print()
{
	for(int i = 0; i < counter; i++)
		printf("%d ", vector[i]);
	printf("\n");
}

int main(void)
{
	print();
	no_odds(vector);
	print();

	return 0;
}
```