The (Duel)Lists – come usare le Linked List in C – pt.2

dottor Jacquin: *[rivolgendosi a Armand d'Hubert] Ho studiato a fondo l'uomo militare, gli ho letteralmente scrutato il cervello; ma il mio fato è quello di rabberciarlo di continuo senza avere alcuna idea di come funzioni.*

Allora: dove eravamo rimasti? Ah, si: le **linked list**. **Nell'ultimo articolo** avevo spiegato l'importanza di questi oggetti nel nostro linguaggio preferito e avevo anche fornito un esempio molto elementare (*ma funzionale!*). È giunto ora il momento di estendere un po' l'esempio (*l'avevo promesso, e "ogni promessa è debito"*). Leggete attentamente l'articolo perché l'argomento è moderatamente intricato e intrigante e, mi raccomando, non fate come il **dottor Jacquin** di **The Duellists** (**I duellanti**) di **Ridley Scott**: se qualcosa non vi è chiaro provate e riprovate fino a trovare il bandolo della matassa, non vi arrendete. Come avrete già intuito, anche in questo articolo parleremo di **linked list**!



…Ci risiamo…

La promessa da mantenere consisteva in aggiungere, all'esempio fornito, qualche funzione di utilità e, soprattutto, permettere di liberare la memoria occupata con **malloc(3)** usando le opportune **free(3)**. Alla fine ho pensato di aggiungere quattro funzioni:

- **findNode()** – cerca un *node* specifico. In questo esempio il *node* è identificato dal campo *"data"* che funge anche da identificativo (vedi note più avanti).
- **delNode()** – cancella un *node* specifico (e libera la memoria). In questo esempio il *node* è identificato dal campo *"data"* che funge anche da identificativo (vedi note più avanti).
- **cleanList()** – cancella l'intera lista (e libera la memoria).
- **printList()** – mostra il contenuto dell'intera lista.

Ebbene si, credo che sia giunto il momento di mostrare la nuova versione del programma. Vai col codice!

```
// linklist.c - un programma di esempio sulle linked list
#include <stdlib.h>
#include <stdio.h> // node di una linked list singola con campo dati
typedef struct snode { int data; struct snode *next;
} node_t; // prototipi locali
void addNode(node_t **head, int data);
void appendNode(node_t **head, int data);
void findNode(node_t **head, int data);
void delNode(node_t **head, int data);
void cleanList(node_t **head);
void printList(node_t **head); // funzione main
int main()
{ // init lista vuota e inserisceo con addNode() 5 nodi con data = indice del loop node_t *head = NULL; for (int i = 0; i < 5; i++) addNode(&head, i); // mostro la lista printList(&head); // cerco un node: quello con data == 3 findNode(&head, 3); // cancello un node: quello con data == 3 delNode(&head, 3); // cerco il node cancellato (quello con data == 3) findNode(&head, 3); // mostro di nuovo la lista:
```

```
deve mancare il node con data == 3 printList(&head); // cancello l'intera lista cleanList(&head); // mostro di nuovo la lista: deve essere vuota printList(&head); return 0;
} // addNode - alloca in testa a una lista un node con dati e pointer al prossimo elemento
void addNode(node_t **head, int data)
{ // alloco un nuovo node node_t *node = malloc(sizeof(node_t)); node->data = data; node->next = *head; // assegno la head lista al nuovo node *head = node;
} // appendNode - alloca in fondo a una lista un node con dati e pointer al prossimo elemento
void appendNode(node_t **head, int data)
{ // alloca un nuovo node node_t *node = malloc(sizeof(node_t)); node->data = data; node->next = NULL; // appende il nuovo node node_t *current = *head; if (current == NULL) { // caso speciale per lista vuota: assegna head lista al nuovo node *head = node; } else { // scorro la lista per trovare l'ultimo node while (current->next != NULL) current = current->next; // assegno all'ultimo node il nuovo node current->next = node; }
} // findNode - trova un node nella lista
void findNode(node_t **head, int data)
{ // scorro la lista node_t *current = *head; while (current) { // cerco il node con il <data> corrispondente if (current->data == data) { // node trovato! printf("node trovato: data=%d (myhead=%p next=%p)\n", current->data, (void *)current, (void *)current->next); return; } // passo al priossimo node current = current->next; } // node non trovato! printf("node con data=%d non trovato!\n\n", data); return;
} // delNode - rimuove un node dalla lista
void delNode(node_t **head, int data)
{ // scorro la lista node_t *current = *head; node_t *previous = NULL; while (current) { // cerco il node con il <data> corrispondente if (current->data == data) { // modifico il riferimento <next> per eliminare dall'elenco il node trovato if (previous) previous->next = current->next; else *head = current->next; // cancello il node e ritorno printf("%s: cancello data=%d (current=%p next=%p)\n", __func__, current->data, (void *)current, (void *)current->next); free(current); return; } // passo al prossimo node previous = current; current = current->next; }
```

```
} // cleanList - reset completo della lista
void cleanList(node_t **head)
{ // scorro la lista node_t *current = *head; node_t *next; while (current) { // salvo il node successivo prima di liberare quello corrente next = current->next; // cancello il node printf("%s: cancello data=%d (current=%p next=%p)\n", __func__, current->data, (void *)current, (void *)current->next); free(current); // mi sposto sul node successivo current = next; } // imposto la head lista a NULL per indicare che la lista è vuota *head = NULL;
} // printList - mostra la lista
void printList(node_t **head)
{ // scorro la lista e stampo i valori node_t *current = *head; printf("%s: lista in %p - elementi contenuti:\n", __func__, (void *)current); while (current) { printf("data=%d (current=%p next=%p)\n", current->data, (void *)current, (void *)current->next); current = current->next; } printf("\n");
}
```

Come avrete notato è molto simile a quello dell'ultimo articolo ma, anche grazie alla nuova funzione **printList()**, il codice è un po' più lineare e leggibile. Credo che grazie ai **commenti esagerati** sia tutto molto chiaro, ma qualche nota da aggiungere c'è: questo è un esempio molto semplificato, quindi non è il caso di chiedersi a che serve una *linked list* con un solo dato (che è, oltretutto, un semplice *int):* evidentemente una *linked list* può contenere (oltre all'indispensabile campo *"next"*) quanti altri campi semplici si vogliono, oppure un solo campo *data* che può essere, a sua volta, una struttura complessa, oppure più strutture complesse, ecc.

Però la architettura del codice rimarrá la stessa, semplicemente sarà un po' più complicato aggiungere e cancellare elementi nel caso che ci siano altre **malloc(3)**/**free(3)** da fare, ma niente di cui preoccuparsi. E anche la **findNode()** e la **delNode()** potrebbero cercare un semplice campo identificativo oppure cercare una struttura complessa: il meccanismo si può semplificare o complicare a piacere, ma alla fin fine l'architettura di base è quella che

vi ho appena mostrato.

Ah, un'ultima cosa, e scusatemi se mi ripeto: quello sopra è un esempio molto semplificato ma, ovviamente, se si vorrà usare come codice di produzione bisognerà aggiungere gli opportuni controlli di esecuzione (specialmente sull'esito delle **malloc(3)**) e gli opportuni trattamenti degli errori del caso. *Non dimenticatevene mai!*

Cosa manca per finire l'articolo? Ah, si, compilare ed eseguire. Vediamo un po:

```
aldo@Linux $ gcc -Wall -Wextra -pedantic linklist.c -o linklist
aldo@Linux $ ./linklist
printList: lista in 0x56a7786e6320 - elementi contenuti:
data=4 (current=0x56a7786e6320 next=0x56a7786e6300)
data=3 (current=0x56a7786e6300 next=0x56a7786e62e0)
data=2 (current=0x56a7786e62e0 next=0x56a7786e62c0)
data=1 (current=0x56a7786e62c0 next=0x56a7786e62a0)
data=0 (current=0x56a7786e62a0 next=(nil)) node trovato:
data=3 (myhead=0x56a7786e6300 next=0x56a7786e62e0)
delNode: cancello data=3 (current=0x56a7786e6300 next=0x56a7786e62e0)
node con data=3 non trovato! printList: lista in 0x56a7786e6320 - elementi contenuti:
data=4 (current=0x56a7786e6320 next=0x56a7786e62e0)
data=2 (current=0x56a7786e62e0 next=0x56a7786e62c0)
data=1 (current=0x56a7786e62c0 next=0x56a7786e62a0)
data=0 (current=0x56a7786e62a0 next=(nil)) cleanList: cancello
data=4 (current=0x56a7786e6320 next=0x56a7786e62e0)
cleanList: cancello data=2 (current=0x56a7786e62e0 next=0x56a7786e62c0)
cleanList: cancello data=1 (current=0x56a7786e62c0 next=0x56a7786e62a0)
cleanList: cancello data=0 (current=0x56a7786e62a0 next=(nil))
printList: lista in (nil) - elementi contenuti:
```

Beh, i risultati parlano chiaro, no? Corrispondono ai passi indicati nel codice:

1. Creo una lista di cinque elementi e stampo il contenuto (*indirizzi di memoria inclusi! Per chi ha dubbi sul funzionamento delle linked list…*)
2. Cerco un *node* (in questo caso quello con *data* == *3*) e mostro cosa ho trovato.
3. Cancello un *node* (in questo caso quello con *data* == *3*).
4. Cerco un *node* (in questo caso quello con *data* == *3*) e mostro che non l'ho trovato (*certo che se lo trovavo era un problema…*).
5. Mostro di nuovo la lista: deve mancare il *node* con *data* == *3* (*speriamo…*)
6. Cancello l'intera lista e stampo il contenuto: non deve apparire nulla (*di nuovo: speriamo…*).

Funziona!

Quando ho cominciato a scrivere lo scorso articolo pensavo a una cosa semplice semplice in una sola parte e via… poi mi è sembrato il caso di aggiungere questa seconda parte e… ma si, prolungherò ulteriormente la descrizione: nel prossimo articolo parlerò di **doubly linked list**! Così mi tolgo il pensiero e non dovrò ritornare sull'argomento chissà quando. *Restate sintonizzati!*

Ciao, e al prossimo post!

https://italiancoders.it/the-duellists-come-usare-le-linked-list-in-c-pt-2/