Esercitazione 5

28-30 Maggio 2025

Esercizio 1: linked list

I file **es0501.rs** contiene due moduli List1 e List2, con due modelli alternativi per implementare una linked list.
Il primo modello si basa su una enum, su un modello simile a quello visto a lezione, con una definizione ricorsiva della lista. Il secondo modello invece utilizza un layout basato su una struct Node con un approccio più simile ad una implementazione C like: la struct contiene l'elemento incapsulato e un attributo next che punta al nodo successivo

1. Implementare entrambe le interfacce come descritte nei commenti del file.
2. Inoltre con il file è fornito un modulo **mem_inspect** che contiene due funzioni **dump_object** e **dump_memory** che permettono di ispezionare oggetti generici e gli smart pointer. Queste funzioni sono da usare, una volta risolto l'esercizio per ispezionare i nodi e gli smart pointer usati:
   - ```
     let mut l1 = List1::List::<i32>::new();
     l1.push(10);
     let mut l2 = List2::List::<i32>::new();
     l2.push(10)
     ```

   Dove sono allocate le head di l1 e l2?
   Che differenze ci sono nell'ultimo nodo tra l1 e l2?
   Qual è il layout di memoria dei nodi?

3. Infine modificare List2 per renderla una lista doppio linkata:
   a. si avranno due puntatori, alla testa (head) e alla coda (tail) della lista e ogni nodo deve puntare anche al precedente nella catena, oltre che al successivo.
   b. si dovrà poter fare push e pop da cima e coda della lista (aggiungere i metodi push_front e pop_front, push_back, pop_back)
   c. inoltre aggiungere in metodo `fn popn(&mut self, n: usize) -> Option<T>` che rimuove l'elemento ennesimo della lista e lo restituisce. Attenzione ad aggiornare correttamente head o tail se il nodo eliminato è all'inizio o alla fine.

   Può ancora essere usato Box per puntare ai nodi adiacenti? Come va modificata la struttura di Node per avere più riferimenti? Vedere i suggerimenti nei commenti del file allegato
     ```

Esercizio 2: albero file system

Realizzare una struct che implementi in memoria la struttura di un file system. Deve essere in grado di poter replicare qualsiasi directory e i suoi contenuti presenti su disco.

La struttura del file system è un albero, che contiene tre tipi di elementi principali:

1. File: i file sono foglie
2. Directory: possono avere altri contenuti.
3. Link simbolici: un link simbolico è un file speciale che contiene un riferimento ad un altro file in qualsiasi punto del file system (usare **fs::read_link(path)** per leggere il valore). Il riferimento nel link può essere anche non valido, attenzione! (*suggerimento: aggiungere i link solo in una seconda fase, all'inizio ignorare i link*)

L'elemento del file system quindi può essere definito con una enum del tipo:

```
enum FSItem {
	Directory(Directory), // Directory contiene nome, i figli, eventuali metadati, il padre
	File(File), // File contiene il nome, eventuali metadati (es dimensione, owner, ecc), il padre
	SymLink(Link) // Il link simbolico contiene il Path a cui punta e il padre
}
```

L'oggetto FileSystem inoltre gestisce il concetto di directory corrente, che può essere selezionata e cambiata. All'inizio è la radice
Ogni operazione supporta path assoluti e relativi, rispetto la directory corrente:

1. path assoluti: iniziano con "/" e partono sempre dalla radice del fs
2. path relativi: tutti quelli che non iniziano per "/" e partono sempre dalla directory corrente. In più sono da gestire due path relativi particolari:
    a. "." la dir corrente
    b. ".." la dir padre

Metodi da implementare per il file system (i Result sono da definire):

```
impl FileSystem {
	// crea un nuovo FS vuoto
	pub fn new() -> Self

	// crea un nuovo FS replicando la struttura su disco
	pub fn from_disk() -> Self

	// cambia la directory corrente, path come in tutti gli altri metodi
	// può essere assoluto o relativo;
	// es: "../sibling" vuol dire torna su di uno e scendi in sibling
	pub fn change_dir(&mut self, path: String) -> Result

	// crea la dir in memoria e su disco
	pub fn make_dir(&self, path: String, name: String) -> Result
```

```
        // crea un file vuoto in memoria e su disco
        pub fn make_dir(&self, path: String, name: String) -> Result

        // rinonima file / dir in memoria e su disco
        pub fn rename(&self, path: String, new_name: String) -> Result

        // cancella file / dir in memoria e su disco, se è una dir cancella tutto
il contenuto
        pub fn delete(&self, path: String) -> Result

        // cerca l'elemento indicato dal path e restituisci un riferimento
        pub find(&self, path: String) -> Result
}
```

Essendo necessario poter risalire anche al padre di un nodo occorre una struttura che non solo permetta di scendere verso i figli, ma anche di risalire verso il padre.
Attenzione quindi ai cicli e fare riferimento all'albero visto a lezione.
Notare che molti metodi che sembrano comportare una modifica alla struttura ricevono &self e non &mut self come parametro. Perché?
Anche in questo esercizio utilizzare dump_object e dump_memory per disegnare il layout degli smart pointer usati per gestire i nodi