Programmazione di Sistema

Lab 2 - Gestione della Memoria e Ereditarietà

Esercizio 1

Si implementi e testi una classe C++ che fornisca l'implementazione di un container `lista` data la seguente definizione:

```
#ifndef MYLIST_H
#define MYLIST_H

class myList
{
public:
    myList();
    ~myList();
    /* Insert an element at the head of the list. Current element moved to
        the new head.*/
    void push_front(const MyClass &val);
    /* Return the value of the node at the head of the list and delete the
        head node. Current element moved to the new head. */
    MyClass popNode();
    /* Insert an element at the tail of the list. Current element moved to
        the tail. */
    void push_back(const MyClass &val);
    /* Insert an element at the pos-th element of the list. Current
       element moved to the new element */
    void insertNode(const MyClass &val, int pos);
    /* Insert an element in a sorted list with increasing values. Current
       element moved to the new element.*/
    void insertIncOrderedNode(const MyClass &val);
    /* Insert an element in a sorted list with decreasing values. Current
       element moved to the new element.*/
    void insertDecOrderedNode(const MyClass &val);
    /* Delete the pos-th node of the list. Current element moved to the
       pos-th previous element. */
    void deleteNodeByPos(const int &pos);
    /* Delete all the nodes of the list containing val. Current element
       moved to the previous element of the deleted element.*/
    void deleteNodeByVal(const MyClass &val);
```

```
    /* Swap the content of nodes at pos0 and pos1. Current element moved
       to the pos0-th element. */
    void swapNodes(const int &pos0, const int &pos1);
    /* Return the value of the pos-th node. Current element moved to the
       found element.*/
    MyClass findNodeByPos(const int &pos) const;
    /* Return the position of the first occurrence of a node containing
       val. Current element moved to the found element. */
    int findNodeByVal(const char &val) const;

private:
    typedef struct listNode
    {
        MyClass value;
        struct listNode *next;
    } nodes;

    nodes *_head, *_tail, *_current_el;

    //Return the pointer to a new node
    nodes *_new_node();
};


#endif // MYLIST_H
```

Per il momento si utilizzi per il contenitore un tipo di dato fisso, scelto dal programmatore, i.e., definendo opportunamente un istanza della classe MyClass. Si scriva un programma per testare il corretto funzionamento della lista implementata.

Spunto di Riflessione

Si verifichino le differenze di prestazioni quando i metodi che vedono i tipi di base come argomento, siano passati per valore invece di una const reference.

Esercizio 2

Si implementi un contenitore C++ per la gestione di vettori, si realizzi il contenitore in modo che, lato utente, sia possibile fare tutto quello che il contenitore `vector` consente di fare, esclusa la gestione degli iteratori. L'aggiunta di un nuovi elementi deve corrispondere alla allocazione di ulteriore spazio in memoria per il vettore, sono possibili due approcci:

- Allocare solamente lo spazio necessario all'aggiunta dei nuovi elementi
- Allocare lo spazio per chunk di memoria. In questo modo il vettore avrà una dimensione fissa di partenza `size` e, qualora l'aggiunta di un nuovo elemento richieda l'aggiunta di memoria si procederà ad allocare un nuovo vettore di dimensione `dim_vettore + size`.

Si ragioni specialmente sul significato e sulle operazione di `resize`, valutando costi e benefici dei due approcci, e su tutti i costruttori di copia ed operatori di assegnazione.
Al fine di provare il funzionamento del contenitore, si scriva una funzione di ordinamento per un vettore.
Per il momento si utilizzi per il contenitore un tipo di dato fisso, scelto dal programmatore, i.e., un vettore di interi, piuttosto che di caratteri.

Esercizio 3

Si realizzi una libreria per la descrizione di un file system contenente directory e file regolari. Per lo svolgimento dell'esercitazione è necessario utilizzare **smart pointers** e **contenitori della STL**.
Per questa esercitazione procederemo con un approccio "bottom up", ovvero implementeremo prima le funzioni base di file e directory e poi ne generalizzeremo la struttura.
Nella prima parte vedremo come realizzare un albero di directory con degli smart pointer.

Parte 1 - Smart Pointer e Gestione di un Albero di Oggetti con Smart Pointer

Gli oggetti di tipo `Directory` hanno come attributi:

- **nome** rappresentato con **std::string name**
- **un riferimento a tutti i figli e alla directory padre**

Inoltre hanno:

- un metodo `void ls(int indent)` che stampa il nome con indentazione pari ad `indent` spazi e, ricorsivamente, tutti gli elementi figli contenuti con un indentazione pari a `indent + 4` spazi.
- un metodo `<SomeKindOfPointer> get(std::string name)` che permette di navigare tra il padre ("..") e i figli, dove *SomeKindOfPointer* è un tipo che permette di fare accesso indiretto al contenuto richiesto.

Spunto di Riflessione
Prima di generalizzare la struttura analizziamo come implementare la relazione padre/figlio tra directory utilizzando degli smart pointer.
Ogni directory ha un solo padre, mentre può avere un numero arbitrario di figli, ciascuno identificato da un nome univoco.
È conveniente poter fare riferimento ad una directory tramite puntatori. Le directory, infatti, hanno la necessità di poter essere navigate sia verso il basso (selezionando una sotto-directory sulla base del suo nome) che verso l'alto (la radice del file system) attraverso lo pseudo-nome ".."; inoltre una cartella deve poter fare riferimento a se stessa tramite lo pseudo-nome ".". Tuttavia non vogliamo usare i puntatori nativi per via della loro intrinseca ambiguità e i rischi sulla gestione della memoria che un loro utilizzo improprio può causare.

- Qual è il contenitore STL più adatto per contenitore i figli? Per una lista completa dei contenitori disponibili si faccia riferimento alla pagina: `https://en.cppreference.com/w/cpp/container`
- Per memorizzare il puntatore al padre è meglio usare un `weak_ptr<Directory>` o uno `shared_ptr<Directory>`?
- Poiché gli smart pointer gestiscono il rilascio del blocco di memoria di cui incapsulano il puntatore tramite l'operatore delete, se un oggetto deve essere manipolato tramite smart pointer, occorre garantire che sia allocato sullo heap. Come si fa ad impedire che possano esistere istanze allocate sullo stack o come variabili globali? (suggerimento: cercate *Named Constructor Idiom* e fatevi ritornare un puntatore ad un nuovo oggetto allocato sullo heap)
- Ogni oggetto ha accesso al proprio puntatore nativo: questo non è altro che il valore della parola chiave `this`. Parimenti, è facile in un oggetto costruire uno smart pointer che punti ad un altro oggetto, ad esempio con la funzione di libreria `std::make_shared<T>()`. Tuttavia, come fa un oggetto ad avere uno smart (*weak* o *shared*) pointer a se stesso? Basta passare `this` al costruttore di uno smart pointer[1]?

[1] Si faccia riferimento a: `https://en.cppreference.com/w/cpp/memory/enable_shared_from_this/shared_from_this`

- Si verifichi la risposta precedente con il seguente esempio

```
#include <iostream>
#include <memory>

using namespace std;

class D : public enable_shared_from_this<D> {
    weak_ptr<D> parent;
public:
    D(weak_ptr<D> parent): parent(parent){
        cout<<"D() @ "<<this<<'\n';
    }
    shared_ptr<D> addChild(){
        shared_ptr<D> child = make_shared<D>(shared_ptr<D> (this));

        return child;
    }
    ~D(){
        cout<<"~D() @ "<<this<<'\n';
    } };
int main(){
    D root(shared_ptr<D>(nullptr));

    shared_ptr<D> child = root.addChild();
}
```

Prima di procedere alla costruzione dell'albero di directory ricordiamo alcune proprietà degli smart pointer:

- quando due oggetti si puntano vicendevolmente (es: d1 $->$ d2 e d2 $->$ d1) almeno uno deve essere un `weak_prt<T>`
- si può ottenere un `weak_ptr<T>` solo da uno `shared_ptr<T>`
- è meglio ottenere uno `shared_ptr<T>` al di fuori dell'oggetto puntato, per evitare una sua distruzione accidentale

Quindi per creare una directory e aggiungerla come figlio creare il metodo: `std::shared_ptr<Directory> addDirectory(const std::string &nome)` Questo metodo deve:

- creare un nuovo oggetto `Directory` figlio che ha due `weak_ref<Directory>`: uno al padre `parent` e uno a se stesso, `self`
- per costruirlo in modo corretto:
  - rendere privato il costruttore (si faccia riferimento a quanto visto sopra con *Named Constructor Idiom*)
  - scrivere un funzione factory statica `makeDirectory(const std::string& name, weak_ptr<Directory > parent)` che al suo interno crei uno `shared_ptr<Directory> dir)` e lo assegni a `dirr->self` come `weak_ptr<Directory>`, salvi il `parent` in una opportuna variabile istanza e restituisca `dir` al chiamante.
- memorizzare il `dir` così ottenuto tra i figli

Si implementi quindi il metodo `static std::shared_ptr<Directory> getRoot()` che crea, se ancora non esiste, l'oggetto di tipo `Directory` radice (”/”) e ne restituisce lo smart pointer. Tale metodo deve appoggiarsi ad una variabile statica della classe `Directory` in cui conservare l'oggetto creato. Come si dichiara una variabile statica? Come e dove la si definisce?
Finire di implementare i metodi `ls` e `get` e provare a costruire e navigare in un albero di directory, verificando che tutti gli oggetti allocati vengono correttamente rilasciati al termine del programma. Per approfondire i temi relativi all'uso della memoria dinamica, si veda il documento: `https://isocpp.org/wiki/faq/freestore-mgmt` Per approfondire i temi relativi agli smart pointer, si vedano i documenti: `https://isocpp.org/wiki/faq/cpp11-library` e `https://medium.com/pranayaggarwal25/a-tale-of-two-allocations-f61aa0bf71fc` Per i problemi legati alla creazione di cicli in strutture gestite da smart pointer, vedere `https://www.modernescpp.com/index.php/std-weak-ptr`

Parte 2 - Polimorfismo mediante ereditarietà

Dal momento che una directory può contenere, al suo interno, sia directory che file, andremo ad utilizzare la classe astratta `Base`, non instanziabile, che è la base comune da cui derivano `Directory` e `File`.
La classe `Base` offre le seguenti funzioni membro pubbliche:

- `std::string getName()const`: restituisce il nome dell'oggetto
- `virtual in mType()const = 0`: metodo virtuale puro di cui fare override nelle classi derivate; restituisce il tipo dell'istanza (`Directory` of `File`) codificato come un intero.
- `virtual void ls(int indent)const = 0`: metodo virtuale puro di cui fare override nelle classi derivate.

Si implementi quindi tale classe in un apposito file chiamato “Base.h“
La classe `Directory` deriva da `Base` e, come nella parte 1 dell'esercizio, ha il costruttore protetto, mantiene come membri privati una collezione di `shared_ptr` ad altri elementi di tipo `Directory` e `File`, un `weak_ptr` alla directory genitore e un `weak_ptr` a se stessa. Il metodo statico `getRoot()` permette di accedere a una singola istanza (corrispondente alla cartella “/“) attraverso la quale si può interagire con il modello di file system implementato. La classe `Directory` espone le seguenti funzioni membro pubbliche:

- `static std::shared_ptr<Directory> getRoot()`: crea, se ancora non esiste, l'oggetto di tipo `Directory` e ne restituisce lo smart pointer. Questo oggetto è il mezzo tramite cui interagire con il modello di file system.
- `std::shared_ptr<Directory> addDirectory(const std::string& nome): crea un nuovo oggetto di tipo \lstinline{Directory`, il cui nome è desunto dal parametro e lo aggiunge alla cartella corrente. Se risulta già presente, nella cartella corrente, un oggetto con il nome indicato, restituisce uno smart pointer vuoto (attenzione ai nomi riservati “.“ e “..“).
- `std::shared_ptr<File> addFile(const std::string& name, uintmax_t size)`: aggiunge alla `Directory` un nuovo oggetto di tipo `File`, ricevendone come parametri il nome e la dimensione in byte; l'aggiunta di un `File` con nome già presente nella cartella corrente non è permessa e il metodo deve quindi restituire uno smart pointe vuoto (idem come sopra).
- `std::shared_ptr<Base> get(const std::string& name)`: restituisce uno smart pointer all'oggetto (`Directory` o `File`) di nome `name` contenuto nella directory corrente. Se inesistente, restituisce uno `shared_ptr` vuoto. I nomi speciali “..“ e “.“ permettono di ottenere rispettivamente lo `shared_ptr` alla directory genitore di quella corrente e il puntatore all'istanza stessa.

- `std::shared_ptr<Directory> getDir(const std::string& name): funziona come il metodo \ textit{get(name)` facendo un `dynamic_pointer_cast> dal tipo \lstinline{Base` al tipo `Directory`

- `std::shared_ptr<File> getFile(const std::string& name)`: funziona come il metodo *get(name)*, facendo un `dynamic_pointer_cast` dal tipo `Base` al tipo `File`

- `bool remove(const std::string& name)`: rimuove dalla collezione di figli della directory corrente l'oggetto (Directory o File) di nome `name`, se esiste, restituendo `true`. Se l'oggetto indicato non esiste e se si tenta di rimuovere ".." o "." viene restituito `false`.

- `void ls (int indent)const override`: implementa il metodo virtuale puro della classe `Base`. Questo metodo elenca, ricorsivamente, file e directory figli della directory corrente, indentati in modo appropriato.

Si implementi quindi tale classe, ponendo la dichiarazione nel file "Directory.h" e l'implementazione nel file "Directory.cpp".
Anche la classe `File` deriva da `Base` e offre le seguenti funzioni membro pubbliche aggiuntive:

- `uintmax_t getSize()const`: restituisce la dimensione del file.

- `void ls (int indent)const override`: implementa il metodo virtuale puro della classe `Base`. Questo metodo stampa nome e dimensione del file con indentazione appropriata.

Si implementi quindi tale classe, ponendo la dichiarazione nel file "File.h" e l'implementazione nel file "File.cpp". Si testi la libreria utilizzando il supporto per l'accesso al file system introdotto in C++17. Si deve quindi leggere una directory e il suo contenuto con `recursive_directory_iterator` e costruire la struttura corrispondente in memoria (`https://en.cppreference.com/w/cpp/filesystem/recursive_directory_iterator`).
NOTA chi utilizza MacOS deve avere la versione 10.15 se vuole utilizzare il compilatore di default clang++, oppure deve installare GCC 9, come indicato in questo documento: `https://solarianprogrammer.com/2017/05/21/compiling-gcc-macos/`