

Programmazione di sistema

Anno accademico 2019-20

Esercitazione 2

Si realizzi una libreria per la descrizione di un filesystem contenente directory e file regolari. Per lo svolgimento dell'esercitazione è **necessario utilizzare contenitori della STL e smart pointer**.

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 attributo nome (**std::string name**), inoltre devono contenere: un riferimento a tutti i figli (file o directory) 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 i figli o il padre ("..") dove *SomeKindOfPointer* è un tipo che permette di fare accesso indiretto al contenuto richiesto

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.

1. **Qual è il contenitore STL più adatto per contenere i figli?** (Si consulti la pagina <https://en.cppreference.com/w/cpp/container>)

E' conveniente poter fare riferimento ad una directory tramite puntatori. Le directory, infatti, hanno 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.

2. Per memorizzare il puntatore al padre è meglio usare un `weak_ptr<Directory>` o uno `shared_ptr<Directory>`? Perché della scelta?
3. 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)
4. 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?

5. Verificare la risposta precedente con questo esempio:

```
class D {
    weak_ptr<D> parent;
public:
    D(weak_ptr<D> parent): parent(parent){
        cout<<"D() @ "<<this<<endl;
    }
    shared_ptr<D> addChild(){
        shared_ptr<D> child= make_shared<D>(
                                shared_ptr<D> (this));
        return child;
    }
    ~D(){
        cout<<"~D() @ "<<this<<endl;
    }
};

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

Quante volte vengono chiamati il costruttore ed il distruttore di `D`? Su quali indirizzi? Chi li ha allocati?

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

- quando due oggetti si puntano vicendevolmente (es: `d1 -> d2` e `d2 -> d1`) almeno uno deve essere un `weak_ptr<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
6. Quindi per creare una directory e aggiungerla come figlio creare il metodo:
- ```
std::shared_ptr<Directory> addDirectory(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
    - scrivere una funzione factory statica **`makeDirectory(string name, weak_ptr<Directory> parent)`** che al suo interno crei uno **`shared_ptr<Directory> dir`**, lo assegni a **`dir->self`** come `weak_ptr<Directory>`, salvi il parent nella opportuna variabile istanza e restituisca **`dir`** al chiamante
  - memorizzare il **`dir`** così ottenuto tra i figli
7. Implementare il metodo `static std::shared_ptr<Directory> getRoot()` – crea, se ancora non esiste, l'oggetto di tipo Directory radice (nome `"/`) 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?**
8. 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

Una directory può contenere directory e file, per questo utilizziamo classe astratta **Base**, che è la base comune da cui derivano Directory e File e non è istanziabile. Offre le seguenti funzioni membro pubbliche:

- **std::string getName() const** – restituisce il nome dell'oggetto
  - **virtual int mType() const = 0** – metodo virtuale puro di cui fare override nelle classi derivate; restituisce il tipo dell'istanza (Directory o File) codificato come intero
  - **virtual void ls(int indent) const = 0** – metodo virtuale puro di cui fare override nelle classi derivate.
9. Si implementi tale classe in un apposito file chiamato "Base.h"

La classe **Directory** deriva da Base e come nella parte 1, ed ha il costruttore protetto. Mantiene come membri privati una collezione di `shared_ptr` ad altri elementi di tipo Directory e File, uno `weak_ptr` alla directory genitore e uno `weak_ptr` a sé stessa. Il metodo statico `getRoot()` permette di accedere a una singola istanza (corrispondente alla cartella "/") attraverso la quale si può interagire con il modello.

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.
- **std::shared\_ptr<Directory> addDirectory(const std::string& nome)** – crea un nuovo oggetto di tipo 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& nome, 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 fa restituire uno smart pointer 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 quello all'istanza stessa.
- **std::shared\_ptr<Directory> getDir(const std::string& name)** – funziona come il metodo `get(nome)`, facendo un `dynamic_pointer_cast` dal tipo Base al tipo Directory
- **std::shared\_ptr<File> getFile(const std::string& name)** – funziona come il metodo `get(nome)`, facendo un `dynamic_pointer_cast` dal tipo Base al tipo File
- **bool remove(const std::string& nome)** – rimuove dalla collezione di figli della directory corrente l'oggetto (Directory o File) di nome "nome", se esiste, restituendo true. Se l'oggetto indicato non esiste o se si tenta di rimuovere ".." e "." viene restituito false.

- `void ls(int indent) const override` – implementa il metodo virtuale puro della classe Base; elenca ricorsivamente File e Directory figli della directory corrente, indentati in modo appropriato

10. Si implementi 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; stampa nome e dimensione del file con indentazione appropriata

11. Si implementi tale classe, ponendo la dichiarazione nel file “File.h” e l’implementazione nel file “File.cpp”

12. Testare la libreria utilizzando il supporto per l’accesso al file system introdotto in C++17.

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](https://en.cppreference.com/w/cpp/filesystem/recursive_directory_iterator) (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/>)

## Esempio di uso

```
std::shared_ptr<Directory> root = Directory::getRoot();
auto alfa = root->addDirectory("alfa");
alfa->addDirectory("beta")->addFile("beta1",100);
alfa->getDir("beta")->addFile("beta2", 200);
alfa->getDir("..")->ls();
alfa->remove("beta");
root->ls(0);
```

## Output

```
[+] /
 [+] alfa
 [+] beta
 beta1 100
 beta2 200
[+] /
 [+] alfa
```

## Competenze da acquisire

- Eredità e polimorfismo
- Uso di smart pointer
- Uso base dei contenitori della Standard Template Library