Programmazione di sistema

Anno accademico 2020-21

Ereditarietà
Polimorfismo
Smart Pointer

Esercitazione 2

Questo è esercizio è da svolgere due volte, con due implementazioni alternative

- una utilizzando puntatori classici
- una seconda trasformando i puntatori in smart pointer, evidenziando vantaggi ed eventuali problematiche

Problema

Si realizzi una libreria per la descrizione e la gestione (in memoria) di un filesystem contenente directory e file regolari.

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

Parte 1: gestione di un albero di oggetti con puntatori

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.

Ogni directory ha un solo padre, mentre può avere un numero arbitrario di figli, ciascuno identificato da un nome univoco.

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 '..' (punto punto); inoltre una cartella deve poter fare riferimento a se stessa tramite lo pseudo-nome '.' (punto).

In questa prima implementazione usiamo direttamente i puntatori nativi alla directory, facendo attenzione a non generare memory leak durante la gestione delle directory.

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

2. Funzioni membro pubbliche da realizzare:
   - **static Directory* getRoot()**: restituisce la radice
   - **Directory* addDirectory(const std::string &name)**: aggiunge una directory come figlio, se la directory esiste restituisce il puntatore alla directory trovata
   - **Directory* get(const std::string &name)**: restituisce una directory figlio il cui nome corrisponde a name e permette di navigare tra i figli o il padre (“..”)
   - **bool remove(const std::string &name)**: rimuove dalla collezione di figli della directory corrente l’oggetto di nome “name”, se esiste, restituendo true (**attenzione alla memoria!**).
     Se l’oggetto indicato non esiste o se si tenta di rimuovere “..” e “.” viene restituito false.
   - **bool move(const std::string &name, Directory *target)**: spostare la directory name da dentro la directory corrente dentro la directory target
   - **bool copy(const std::string &name, Directory *target)**: copiare la directory name da dentro la directory corrente dentro la directory target (attenzione alla memoria!)
   - **void ls(int indent)**: stampa l’albero di directory a partire della directory corrente

3. Che problematica porterebbe avere un costruttore pubblico di Directory e un metodo Directory::add(Directory *child)? A cosa bisognerebbe fare attenzione
4. Analogamente Directory::add(Directory &target)?
5. Implementare una strategia per contare gli oggetti allocati verificare che siano state chiamate tutte le delete attese. Quale strategia si può utilizzare per evitare di sporcare distruttori e costruttori di Directory con un count? Hint: pensare all’ereditarietà o alla composizione: cosa succede a costruttore e distruttore di classe base o di un attributo quando un oggetto viene costruito/distrutto?

Esempio di uso

```
Directory* root = Directory::getRoot();
auto alfa = root->addDirectory("alfa");
root->addDirectory("beta")->addDirectory("beta1");
root->getDir("beta")->addDirectory("beta2");
alfa->getDir("..")->ls(4);
alfa->remove("beta");
root->ls(4);
```

Output

```
[+] /
      [+] alfa
      [+] beta
```

```
            beta1
            beta2
[+] /
      [+] alfa
```

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.

Si implementi tale classe in un apposito file chiamato "Base.h", così come per le classi successive (.h e .cpp)

Anche la classe **File** deriva da Base e oltre al suo nome ha una dimensione e la data di creazione. La classe **File** offre le seguenti funzioni membro pubbliche aggiuntive:

- **uintmax_t getSize() const** – restituisce la dimensione del file
- **uintmax_t getDate() const** – restituisce la data di creazione
- **void ls**(int indent) **const override** – implementa il metodo virtuale puro della classe Base; stampa nome e dimensione del file con indentazione appropriata

Invece la nuova implementazione di Directory è:

- **static Directory* getRoot()**: restituisce la radice
- **Directory* addDirectory(const std::string &name)**: aggiunge una directory come figlio, se la directory esiste restituisce il puntatore alla directory trovata, se esiste già un file con lo stesso nome invece null
- **File* addFile(const std::string &name, uintamx_t size)**: aggiunge un File, ritorna null se non lo può creare
- **Base* get(const std::string &name)**: restituisce un oggetto figlio il cui nome corrisponde a name e permette di navigare tra i figli o il padre ("..")
- **Directory* getDirectory(const std::string &name)**: restituisce una directory figlia il cui nome corrisponde a name e permette di navigare tra i figli o il padre ("..")
- **File* getFile(const std::string &name)**: restituisce un file figlio
- **bool remove(const std::string &name)**: rimuove dalla collezione di figli della directory corrente l'oggetto di nome "name", se esiste, restituendo true

Se l'oggetto indicato non esiste o se si tenta di rimuovere "**..**" e "**.**" viene restituito false.

- **bool move(const std::string &name, Directory *target)**: spostart la directory name da dentro la directory corrente dentro la directory target
- **void ls(int indent) const override**: stampa l'albero di directory a partire della directory corrente

Testare la libreria utilizzato 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 (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/)

Parte 3: uso di smart pointer

Utilizzare direttamente i puntatori nativi può portare ad errori nella gestione della memoria difficilmente identificabili.
Il C++ offre un costrutto, gli smart pointer, che permette di tracciare i riferimenti agli oggetti allocati e distruggerli automaticamente quando non sono più referenziati da nessuno.

Ci sono tre tipi base di smart pointer

- **unique_ptr<T>**: accetta un solo riferimento a T*, non può essere copiato ma supporta la move; quando esce dallo scope viene chiamata delete T
- **shared_ptr<T>**: accetta n riferimenti a T*, può essere copiato, delete T viene chiamata quando esce dallo scope l'ultimo riferimento T
- **weak_ptr<T>**: si ottengono dagli shared_ptr, hanno un riferimento a T* ma non partecipa al reference count; sono utili quando due oggetti posso *puntarsi* in modo circolare (es una lista doppio linkata, padre e figlio nella struttura delle directory): se si usassero shared_ptr si avrebbe un loop e non verrebbero mai deallocati, invece se in una direzione si usa un weak_ptr (es verso il padre), una volta eliminato lo shared (es da figlio a padre) può venire tutto deallocato.

Per una guida introduttiva leggere questo link:

- https://www.internalpointers.com/post/beginner-s-look-smart-pointers-modern-c

Trasformare Base, Directory e File per poter usare gli smart pointer. In questa nuova implementazione Directory mantiene come membri privati:

- una collezione di shared_ptr ad altri elementi di tipo Directory e File,
- uno weak_ptr alla directory genitore
- uno weak_ptr a sé stessa.

Punti di attenzione:

1. 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)

2. 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?

3. 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?

4. 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

5. 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?

6. 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

Nuova interfaccia della classe Directory usando gli smart pointer:

- **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.
- **bool move**(const std::string& nome, **std::shared_ptr<Directory> dest**)
- **bool copy**(const std::string& nome, **std::shared_ptr<Directory> dest**): cosa cambia con gli smart pointer? (*attenzione domanda "Ingannevole", pensare anche alla logica della copy*)
- **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

Eseguire gli stessi test fatti con la implementazione precedente