Université de Paris **Langages à objets avancé**
M1 **Année 2020—2021**

TP Noté n° 2

Ce travail est à réaliser chez vous et individuellement et à déposer sur Moodle au plus tard le vendredi 30 Octobre à 23h59 sous forme d'une archive [1]. Un fichier `main.cpp` vous est fourni pour tester votre code. Votre archive devra contenir un `Makefile` permettant de **compiler votre code**, et ce, **sans erreurs**. L'absence de ce Makefile ou l'existence d'erreurs de compilation par les commandes qu'il exécute vous fera avoir la note 0 automatiquement. Dans le cas où vous n'arrivez pas à faire toutes les questions, vous êtes invités à

— mettre des définitions vides pour les méthodes que vous n'avez pas réussi à écrire (juste des `return ...` de valeurs arbitraires),

— ou commenter les parties du `main` de `main.cpp` que votre code ne gère pas.

Cependant, dans l'idéal, votre code doit pouvoir être exécuté avec le fichier `main.cpp` fourni.

Réfléchissez bien à ce qui doit être déclaré constant dans votre programme, à ce qui doit être passé par référence en argument de vos méthodes et fonctions (cela influencera votre note finale). Vous veillerez à empêcher les inclusions circulaires de fichiers `.hpp` en utilisant la technique standard

```
#ifndef XXX_HPP
#define XXX_HPP
...
#endif
```

Aussi, vous utiliserez la classe `string` pour manipuler les chaînes de caractères.

Système de fichiers en mémoire

Dans ce TP, vous allez écrire un système de fichiers très simplifié tenant complètement en mémoire vive. Pour cela, vous écrirez les quatre classes suivantes :

— une classe `FileSystem`, qui représente une instance de ce système de fichiers,

— une classe `Node`, qui représente un nœud de l'arborescence, c'est-à-dire un fichier ou un dossier,

— une classe `File`, qui représente un fichier dans l'arborescence,

— une classe `Directory`, qui représente un dossier dans l'arborescence.

1. `tar cvf mon_fichier.tar fic1 fic2 ...` pour créer l'archive `mon_fichier.tar` composée des fichiers `fic1`, `fic2`, ...

La classe `FileSystem`

Dans cette section, on écrit le code associé à la classe `FileSystem` qui décrit un système de fichiers. Un système de fichiers a un nom et un dossier racine à partir duquel on peut accéder au reste de l'arborescence. De plus, un système de fichiers fournit une méthode permettant de générer des identifiants uniques qui seront utilisés pour numéroter les nœuds de l'arborescence.

1. Créez deux fichiers `FileSystem.hpp` et `FileSystem.cpp` associés à la définition d'une classe `FileSystem` décrivant un système de fichiers. Déclarez et définissez un destructeur.
2. Ajoutez un attribut `name` à la classe représentant le nom du système de fichiers et ajoutez un accesseur `get_name()`. Déclarez et définissez un constructeur qui prend en argument un tel nom.
3. Ajoutez un attribut `Directory *root` qui est un pointeur vers un objet de la classe `Directory` définie plus loin. On utilisera la syntaxe adéquate avant la déclaration de la classe `FileSystem` pour autoriser un tel attribut alors que la classe `Directory` n'est pas encore déclarée. Ajoutez de plus un accesseur `get_root()`.
4. Déclarez et définissez une méthode non publique `int get_fresh_uid()` qui génère un nouvel identifiant à chaque appel qui est différent des précédents renvoyés. On pourra utiliser un autre attribut non publique pour définir cette méthode.

La classe `Node`

Dans cette section, on écrit le code associé à la classe `Node` dont les objets représentent les nœuds (fichiers ou dossiers) de l'arborescence associée à un système de fichiers. Un nœud contient un pointeur vers le système de fichiers auquel il appartient, un identifiant unique (qui est un entier), un nom et un pointeur vers le dossier parent qui le contient. Les classes `File` et `Directory` définies plus loin hériteront de `Node`. Étant donné un nœud, on aimerait pouvoir savoir si c'est un dossier ou non et, suivant cette information, pouvoir convertir un pointeur `Node*` vers ce nœud en un pointeur `Directory*` ou `File*`. Aussi, on aimerait pouvoir connaître la "taille" de ce nœud dans l'arborescence.

1. Créez deux fichiers `Node.hpp` et `Node.cpp` associés à la définition d'une classe `Node` décrivant un nœud dans un système de fichiers. Cette classe aura comme attributs un pointeur `fs` vers un objet `FileSystem`, un entier `uid`, une chaîne de caractères `name`, et un pointeur `parent` vers un objet `Directory`. On utilisera la même syntaxe que dans la section précedente pour pouvoir déclarer l'attribut `parent` sans que la classe `Directory` ne soit encore déclarée. Déclarez et définissez un constructeur prenant comme argument ces quatre attributs (on donnera une valeur par défaut à `parent` qui sera `nullptr`). Déclarez et définissez un destructeur. Définissez un accesseur publique pour `name`.
2. Modifiez la visibilité du constructeur et du destructeur de sorte qu'un utilisateur extérieur ne puisse ni créer ni détruire de `Node` directement.

3. Déclarez les quatre méthodes virtuelles suivantes :
   — une méthode `is_directory()` qui renvoit un booléen,
   — une méthode `to_directory()` qui renvoit un pointeur sur un objet `Directory`,
   — une méthode `to_file()` qui renvoit un pointeur sur un objet `File`,
   — une méthode `size()` qui renvoit un entier.

   Comme les objets `Node` ne seront instanciés que comme des `Directory` ou `File`, et que ces deux classes redéfinieront ces quatre méthodes, vous pouvez donner une définition arbitraire aux quatres méthodes précédentes (ou mieux, vous pouvez déclarer ces méthodes comme étant *virtuelles pures*, en rajoutant `= 0` à la fin de la déclaration [2]).

La classe `File`

Dans cette section, on écrit le code de la classe `File` dont les objets représentent les fichiers de l'arborescence associée à un système de fichiers. Un fichier est tout simplement un nœud étendu avec un attribut de type `string` représentant son contenu.

1. Créez deux fichiers `File.hpp` et `File.cpp` associés à la définition d'une classe `File` décrivant un fichier dans un système de fichiers. Cette classe héritera de `Node` et aura un attribut supplémentaire `content` qui sera une chaîne de caractères. Déclarez et définissez
   — un constructeur avec quatre paramètres qui appelle celui de `Node`,
   — un destructeur,
   — un accesseur `get_content()` et un mutateur `set_content(const std::string&)` pour `content`.

   Jouez sur la visibilité du constructeur et destructeur afin qu'un utilisateur extérieur ne puisse ni créer ni détruire des fichiers directement.
2. Afin de pouvoir redéfinir pour la classe `File` les quatre méthodes virtuelles déclarées dans la classe `Node`, redéclarez-les dans la classe `File` et définissez-les en suivant les instructions suivantes :
   — on prendra la taille de `content` comme valeur devant être retournée par `size()` ;
   — la méthode `is_directory()` devra renvoyer `false` (car un objet `File` n'est pas un dossier !) ;
   — la méthode `to_directory()` devra renvoyer `nullptr` ;
   — la méthode `to_file()` devra renvoyer `this`.

2. c.f. `https://cpp.developpez.com/faq/cpp/?page=Les-fonctions-membres-virtuelles#Qu-est-ce-qu-une-fonction-virtuelle-pure`

La classe `Directory`

Dans cette section, on écrit le code de la classe `Directory` dont les objets représentent les dossiers de l'arborescence associée à un système de fichiers. Un dossier est un nœud étendu avec un attribut représentant la liste des nœuds qu'il contient (ses « enfants »).

1. Créez deux fichiers `Directory.hpp` et `Directory.cpp` associés à la définition d'une classe `Directory` décrivant un dossier dans un système de fichiers. Cette classe héritera de `Node` et aura un attribut supplémentaire `children` qui sera un tableau `std::vector<Node*>` de pointeurs sur des nœuds. Déclarez et définissez un constructeur avec quatre paramètres qui appelle celui de `Node` (on gardera le même argument optionnel que pour `Node`), et un destructeur. Votre destructeur devra détruire les nœuds « enfants » du dossier. Jouez sur la visibilité du constructeur et destructeur afin qu'un utilisateur extérieur ne puisse ni créer ni détruire des dossiers directement.
2. Afin de pouvoir redéfinir pour la classe `Directory` les quatre méthodes virtuelles déclarées dans la classe `Node`, redéclarez-les dans la classe `Directory` et définissez-les en suivant les instructions suivantes :
   — on prendra la somme des tailles des nœuds « enfants » du dossier comme valeur devant être retournée par `size()` ;
   — la méthode `is_directory()` devra renvoyer **true** ;
   — la méthode `to_directory()` devra renvoyer **this** ;
   — la méthode `to_file()` devra renvoyer `nullptr`.
3. Déclarez et définissez les méthodes suivantes qui permettent d'accéder et de modifier le contenu d'un dossier :
   — une méthode `add_file`, qui prend en argument un `std::string` représentant un nom de fichier. Cette méthode créera et ajoutera un nouveau fichier (dont le nom est celui précisé par l'argument) comme « enfant » du dossier courant et renverra un pointeur `File*` vers le fichier créé. Cette méthode vérifiera qu'un nœud du même nom n'existe pas déjà dans le dossier (sinon, elle renverra `nullptr`). On fera attention à attribuer un identifiant unique au nouveau fichier en utilisant les méthodes de `fs` (afin d'être autorisé à utiliser ces méthodes depuis `Directory`, on déclarera une relation d'amitié entre les classes `FileSystem` et `Directory`) ;
   — une méthode `add_directory` qui prend en argument un `std::string` représentant un nom de dossier. Cette méthode créera et ajoutera un nouveau dossier (dont le nom est celui précisé par l'argument) comme « enfant » du dossier courant et renverra un pointeur `Directory*` vers le dossier créé. Cette méthode vérifiera qu'un nœud du même nom n'existe pas déjà dans le dossier (sinon, on renverra `nullptr`). Ici aussi, on vérifiera qu'on initialise le nouveau dossier avec un identifiant unique obtenu avec les méthodes de `fs` ;
   — une méthode `remove_node` qui prend en argument un `std::string` représentant un nom de nœud à éliminer. Cette méthode cherchera un nœud enfant

dont le nom correspond à l'argument. Dans le cas où un tel nœud est trouvé, ce nœud sera enlevé des nœuds enfants et détruit, et la méthode renverra **true**. Sinon, la méthode renverra **false** ;

— une méthode `find_node` qui prend en argument un `std::string` représentant un nom de nœud à trouver. Cette méthode cherchera un nœud du même nom parmi les nœuds « enfants » du dossier. Dans le cas où un tel nœud est trouvé, un pointeur `Node*` vers ce nœud sera renvoyé. Sinon, `nullptr` sera renvoyé.

4. Modifiez le constructeur de `FileSystem` de façon à initialiser `root` à un pointeur vers un dossier sans nœuds « enfants » dont le nom est `"root"` et dont le pointeur `parent` est `nullptr` et auquel on attribuera un identifiant unique. Similairement, modifiez le destructeur de `FileSystem` de sorte que le dossier `root` soit supprimé à la destruction. Afin de pouvoir créer et détruire un `Directory` depuis `FileSystem`, on déclarera une relation d'amitié entre les classes `Directory` et `FileSystem`.

Affichage

Dans cette section, on écrit une fonction d'affichage pour la classe `FileSystem`. Cette fonction affichera le nom du système de fichiers, puis affichera l'arborescence des dossiers et fichiers contenus. Pour cela, on affichera les lignes correspondant aux fichiers et aux dossiers en les précédant d'une marge constituée d'un nombre d'espaces qui variera selon la profondeur à laquelle on se trouve dans l'arborescence.

1. Dans la classe `Node`, déclarez une méthode virtuelle protégée `print_to` qui prend en argument une référence vers `std::ostream` et un **int**, et dont le type de retour est **void**. Définissez-la de façon arbitraire (ou mieux, déclarez-la comme *virtuelle pure*).
2. Dans la classe `File`, redéclarez la méthode `print_to` et définissez-la. Cette méthode écrira une ligne dans l'objet de type `ostream` qui commencera par `n` espaces, où `n` est le deuxième argument de la méthode, suivis d'une chaîne de la forme

   ```
   + file: "[name]", uid: [uid], size: [size], content: "[content]"
   ```

   où l'on aura remplacé `[name]`, `[uid]` et `[content]` par les valeurs des attributs correspondants, et `[size]` par la valeur renvoyée par la méthode `size()`.
3. Dans la classe `Directory`, redéclarez la méthode `print_to` et définissez-la. Cette méthode écrira une ligne dans l'objet de type `ostream` qui commencera par `n` espaces, où `n` est le deuxième argument de la méthode, suivis d'une chaîne de la forme

   ```
   + directory: "[name]", uid: [uid], size: [size]
   ```

   où l'on aura remplacé `[name]`, `[uid]` et `[size]` par les valeurs adéquates. Puis, on appelera la méthode `print_to` de chacun des nœuds « enfants » du dossier, en utilisant `n+1` pour le deuxième argument.

4. Surchargez l'opérateur `<<` pour la classe `Directory`. La fonction d'affichage associée devra appeler `print_to` avec `0` comme deuxième argument. Afin que la surcharge de `<<` ait accès à la méthode `print_to`, on déclarera une relation d'amitié entre la classe `Directory` et la surcharge de `<<`.
5. Surchargez l'opérateur `<<` pour la classe `FileSystem`. Pour un objet nommé `fs`, la fonction d'affichage associée devra écrire une ligne de la forme

   ```
   filesystem "[name]"
   ```

   puis demander l'affichage du dossier pointé par `root` en utilisant la surcharge de l'opérateur `<<` sur `*fs.get_root()`.

On pourra regarder le fichier `main.cpp` fourni pour avoir un exemple d'affichage.