Etudiant : Théo Dubus 22008507

TP1 C++

Cette version implémente

- L'analyse de tokens
- La conversion en représentation RPN
- L'évaluation d'une suite de tokens en RPN
- Les nombres réels
- Les opérateur binaires `+ - * /`
- Les parenthèses `( )`
- Les priorité opératoires
- Les séquence d'expression, avec `;` pour ne pas afficher l'expression
- Les identifiants de la forme `[a-z A-Z][a-z A-Z 0-9 '_']*`
- Une mémoire
    - déclaration avec `ID = expression;`
    - modification avec `ID_1 = ID_2`
    - appel avec `ID`

Difficultés rencontrés

Au début du projet, la compréhension de la fonction `eval(...)` et son implémentation polymorphique sont surement les 2 étapes qui m'ont pris le plus de temps.
La gestion des fuites mémoire a aussi été complexe, j'ai essayé d'utiliser les `unique_ptr` mais après de nombreux problèmes, le temps étant compté je me suis tourné vers une gestion avec `new` et `delete` qui ne me plaisent pas vraiment. Je prévois d'utiliser les `shared_ptr` lors du prochain TP qui sont plus adapté au projet car elles permettent la création de plusieurs pointeurs vers la même ressource.

Choix d'implémentation

Les classes `Literal` (nombres réels et entiers) `BinOp` (+,-,*,/) `Par` (Parenthèses) héritent toutes les tois de la classe abstraite `Token`. Les méthodes `eval` `print` et `parse` de la classe `Expression` exploitent ce polymorphisme.
Les fonction sont gérés par la partie analyseur syntaxique, lorsque une définition`ID = ...;` est trouvé, on traite la partie après le `=` jusqu'à `;` comme un sous programme puis on évalue et stocke le résultat dans une map `{id, valeur}`. Si l'ID est déja dans la map on modifie la valeur.

Lors de l'affichage d'une variable `ID`, on retrouve la valeur stocké dans la map `{id, valeur}` et on ajoute un Token `Literal` contenant cette valeur.

Compilation du projet

Le projet utilise le système de build CMake. Un script `buildrun.sh` permet de compiler, lancer les test et la calculatrice en 1 commande.

Le projet compile sous :

- `clang 10.0.0` sur Windows/Ubuntu
- `gcc 7.5.0` sur Ubuntu ou WSL (sous-système Windows pour Linux)

Tests

Les tests sont regroupés dans le répertoire `\test`. J'ai choisi d'utiliser la libairie de test Googletest qui est l'une des plus populaire (Catch, Boost.Test et Ctest sont aussi de bons candidats).

Les test sont séparés en 2 fichiers `.cpp` :

- `expressionTest.cpp` : Les tests d'analyse de tokens, crées au début du projet pour vérifier le fonctionnement de `tokensFromString`
- `programTest.cpp` : Les tests du fonctionnement du programme complet
    - programme en une ligne (addition, priorités opératoires, parenthèses)
    - programme multi lignes (variables, affichage ou non avec `;` , calcul du volume d'un cylindre...)

Fuites mémoire

Le projet a été testé avec `valgrind` pour détecter la présence ou non de 'memleaks'. Un script `testValgrind.sh` permet de tester les fuites sur un programme type composé de toutes les possibilités du langage.

Continuous integration

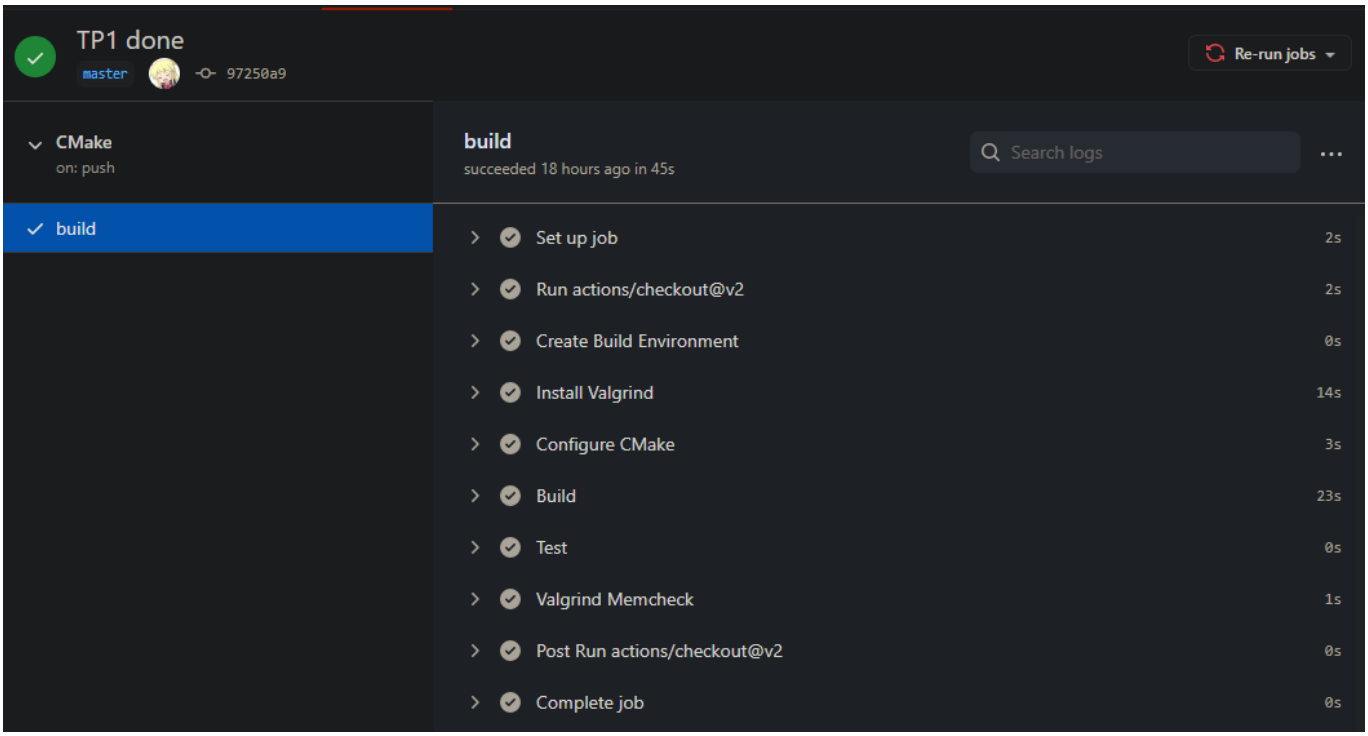
Le repo de ce projet est hébergé sur github.com (en privé pour éviter la copie), j'en ai donc profité pour expérimenter la continuous intégration avec Github Actions qui permet d'effectuer une série de test pour valider mes commit à chaque push.

`.github/workflow/cmake.yml` contient le fichier yaml qui s'occupe de cette continuous integration.
Il y est effectué en autre:

- la compilation du projet avec CMake.
- Le lancement des tests Googletest.
- La vérification de présence des fuites mémoires avec Valgrind.

Exemples d'execution de CI : \

- Execution du "workflow" avec succès :



- Echec de la tache Valgrind Memcheck :