

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 opérateurs binaires `+` `-` `*` `/`
- Les parenthèses `()`
- Les priorités opératoires
- Les séquences 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 rencontrées

Au début du projet, la compréhension de la fonction `eval(...)` et son implémentation polymorphique sont sûrement 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 adaptés 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 trois de la classe abstraite `Token`. Les méthodes `eval` `print` et `parse` de la classe `Expression` exploitent ce polymorphisme.

Les fonctions sont gérées par la partie analyseur syntaxique, lorsque une définition `ID = ...;` est trouvée, 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 déjà 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, créés 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 :

TP1 done
master 97250a9

Re-run jobs

▼ CMake
on: push

build
succeeded 18 hours ago in 45s

Search logs

- ✓ Set up job 2s
- ✓ Run actions/checkout@v2 2s
- ✓ Create Build Environment 0s
- ✓ Install Valgrind 14s
- ✓ Configure CMake 3s
- ✓ Build 23s
- ✓ Test 0s
- ✓ Valgrind Memcheck 1s
- ✓ Post Run actions/checkout@v2 0s
- ✓ Complete job 0s

- Echec de la tache Valgrind Memcheck :

working parenthesis
master 42e2f03

Re-run jobs

▼ CMake
on: push

build
failed 3 days ago in 52s

Search logs

✗ build

▼ ✗ Valgrind Memcheck 6s

```

23 ==3448== by 0x10A068: main (in /home/runner/work/TP_CPP_M1/build/src/calculator_run)
24 ==3448== If you believe this happened as a result of a stack
25 ==3448== overflow in your program's main thread (unlikely but
26 ==3448== possible), you can try to increase the size of the
27 ==3448== main thread stack using the --main-stacksize= flag.
28 ==3448== The main thread stack size used in this run was 8388608.
29 ==3448==
30 ==3448== HEAP SUMMARY:
31 ==3448== in use at exit: 1,168 bytes in 25 blocks
32 ==3448== total heap usage: 28 allocs, 3 frees, 82,064 bytes allocated
33 ==3448==
34 ==3448== LEAK SUMMARY:
35 ==3448== definitely lost: 0 bytes in 0 blocks
36 ==3448== indirectly lost: 0 bytes in 0 blocks
37 ==3448== possibly lost: 0 bytes in 0 blocks
38 ==3448== still reachable: 1,168 bytes in 25 blocks
39 ==3448== suppressed: 0 bytes in 0 blocks
40 ==3448== Rerun with --leak-check=full to see details of leaked memory
41 ==3448==
42 ==3448== For counts of detected and suppressed errors, rerun with: -v
43 ==3448== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
44 /home/runner/work/_temp/b7bc3157-c4ba-4987-a3ba-9656cecb6c82.sh: line 1: 3448 Segmentation fault
(core dumped) valgrind ./src/calculator_run 100/2+33-7
45 Eval:
46 Error: Process completed with exit code 139.

```

- ✓ Post Run actions/checkout@v2 1s
- ✓ Complete job 0s