Phase n°1

Parser d'expression mathématique "simple"

À chaque phase, vous devez rendre

- Un *rapport* (avec une vue générale des choix que vous avez eu à faire et des difficultés rencontrées)
- Votre *code* (qui doit compiler, et de préférence être commenté)
- Un système de build (CMake, Makefile, Meson)

Introduction

Vous allez implémenter, au cours du projet C++, un évaluateur d'expressions tout d'abord simple, puis au fur et à mesure un peu plus sophistiqué. Cette phase n°1 TP à principalement pour but de vous faire pratiquer et, pour certains, découvrir le language C++ et plus spécifiquement le C++ "moderne" (C++11 et supérieur), mais aussi et surtout de *constituer la base de code qui vous servira pour le projet.* À la fin de ce TP, votre programme devra pouvra pouvoir lire des expressions sur l'entrée standard comme ceci

```
$ ./calculatrice
1+1 <enter>
2
3*4 <enter>
12
^D
$
```

ou, de la même manière, comme ceci

```
$ echo '1+1' >> expressions.txt
$ echo '3*4' >> expressions.txt
$ cat expressions.txt | ./calculatrice
2
12
$
```

Tout au long du sujet, nous vous invitons (mais ce **n'est pas noté**) à créer des fichiers de code C++ illustrant un aspect précis du language (pour, par exemple, mettre en évidence un bug). Ce genre de fichiers s'apparente aux *minimal working examples*, qui servent entre autres à décrire des bugs sur des forums tels que Stack Overflow. Le but est ici de vous approprier des notions du C++ de manière isolées, et de vous aiguiller sur des solutions pour le projet. Ces interludes sont marqués par des "Tip" comme ci-dessous:

> **Tip**
> Ce n'est pas la peine d'inclure les MWE dans vos systèmes de build, c'est juste pour vous et c'est *optionnel*.

Autre remarque: tout au long du sujet, des prototypes de fonctions sont décrits. **Ce ne sont pas des prototypes exacts**. La plupart du temps il vous faudra les modifier un peu, de manière à éviter les copies inutiles, par exemple.

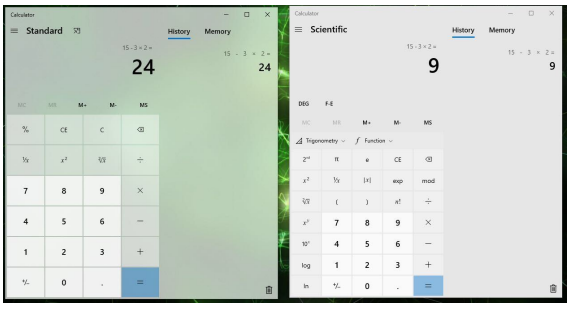

Figure 1: La calculatrice officielle de Windows, au comportement curieux. Aussi étonnant que ça puisse paraître, ce n'est pas un bug mais une fonctionnalité.

Base

Commencez par créer la structure de base de votre projet (voir l'annexe sur la structuration d'un projet C++, que vous pouvez tout à fait éviter si vous étiez à l'aise en C). Vous devriez assez rapidement pouvoir obtenir une boucle lisant plusieurs lignes à partir du flux d'entrée standard et se comportant un peu comme le programme `cat` sans argument. Par la même occasion, vous vous familiariserez un peu avec des bases de C++. Vous aurez besoin d'un flux d'entrée (`std::istream`) comme `std::cin`. Notez que l'affichage, en C++, se fait de manière analogue, avec un flux de sortie comme `std::cout` et `std::cerr`, que vous trouverez dans `<iostream>`.

Maintenant que vous disposez d'un programme capable de boucler sur des entrées utilisateur, il est temps de passer au cœur du sujet, *la conversion d'une chaîne de caractères, une expression, en une valeur.*

Conversion

Type Expression

Afin de représenter et évaluer ces expressions, constituées d'entiers et des opérateurs `+`, `-`, `*` et `/` avec les priorités habituelles, créez la classe `Expression` devant fournir à minima l'interface

```
constructeur :: const char*
  -> Expression
eval :: ? -> int
print :: () -> ()
```

qui correspond en C++ à une classe contenant *entre autres*

```
class Expression {
public:
    /// Construit l'expression
    /// depuis une chaîne de
    /// caractères.
    Expression(const char* str);

    /// Retourne le résultat de
    /// votre expression.
    int eval();

    /// Affiche la représentation
    /// interne de votre expression.
    void print();

    // ...
} ;
```

qui *pourrait* être utilisée comme suit

```
int main(int argv, char* argv[]) {
    Expression e(argv[1]);
    std::cout << "x = "
              << e.eval()
              << std::endl;
    e.print();
}
```

en n'omettant évidemment pas les tests qui s'imposent ici (et que, naturellement, le sujet omet).

Analyse

Dans un premier temps, il vous faut être capable de *(1)* séparer la chaîne en entrée en tokens et *(2)* les organiser sous une forme propice à l'évaluation.

La constitution même des tokens est un des points centraux du TP. Ces derniers doivent pouvoir représenter, à ce point du TP, ou bien des entiers ou des opérateurs. On doit être capable, à l'éxécution (et non pas à la compilation), de distinguer la catégorie à laquelle ils appartiennent, et de les évaluer en fonction de cette catégorie (la fonction `int eval(...)`), et ce *sans* avoir recours à du RTTI, comme on pourrait le faire dans d'autres langages. Cette forme de polymorphisme était réalisée en C via des `union`. Le C++ dispose de ces mêmes `union` ainsi que de `std::variant`, mais il possède encore un autre moyen de la mettre en oeuvre, *à base d'héritage.*

Quoi que vous choisissiez, vous devrez pouvoir disposer d'objets `Token` qui offrent une fonction `int eval(...)`. Par exemple, on s'attend, quel que soit le type du token, à pouvoir faire

```
Token* tok = // ...
std::cout << tok->eval(/* ... */)
          << std::endl;
```

Tip

Créez un object `Rectangle`, et une classe dérivée `Square`, toutes deux avec une méthode `print()` virtuelle. Créez un vecteur de `Rectangle` et peuplez-le avec des carrés et des rectangles, et appelez la méthode print sur tous les éléments. Constatez que les carrés s'affichent comme des rectangle ; stockez cette fois des *pointeurs de* rectangles dans votre vecteur, et affichez de nouveau. Que constatez-vous ? Comme vu en TD, ce comportement est dû à la table virtuelle des objets stockés. On dit que la méthode print est polymorphique.

Découpe / ségmentation en tokens / lexer

Crééz une fonction `tokensFromString` (ou le nom qu'il vous plaira) et de prototype

```
std::vector<Token*> tokensFromString(
       const std::string& s, char delim);
```

qui sépare la chaîne s en plusieurs tokens. *Ces tokens ne sont pas forcément séparés par des espaces.*

La méthode la plus simple consiste à reproduire le comportement d'un lexer réel, qui procède caractère par caractère, et qui renvoie, à la fin d'un léxème, le token correspondant. Vous pouvez (et devriez) représenter ce côté machine à état explicitement dans votre code, où les états sont ici le type du léxème courant et les transitions sont le passage d'un léxème à un autre. Par exemple

```
1+2         → transition →   1+2
 ^ opérateur                   ^ nombre
```

Note

La class `std::stringstream` et la fonction `std::getline` peuvent vous être utiles (entre autres). Notez aussi que dans le sujet, le vecteur contient ici des Token directement ; cela n'est pas une nécessité et peut même être une mauvaise idée. À vous de voir. L'annexe sur les smart pointers, ainsi que vos exercices de TD, pourraient vous être utiles à ce point du TP.

Analyse syntaxique / parser

Vos tokens étant constitués et rangés par ordre d'arrivée, il vous faut encore les organiser sous une forme qui soit évaluable facilement (autrement dit, vous aller *compiler* votre suite de tokens vers une représentation interne plus simple). Vous profiterez également de cette étape pour rendre votre programme aussi robuste que possible aux erreurs de syntaxe potentielles.

Les règles à observer sont les suivantes:

- la division et la multiplication sont prioritaires sur l'addition et la soustraction
- à priorité équivalente, évaluer de *gauche à droite*

de sorte que

- $3 + 2 * 4 = 3 + (2 * 4) = 11$
- $17 - 24/4 * 3 + 2 = (17 - ((24/4) * 3)) + 2 = 1$

La représentation interne que vous devez implémenter s'appelle la représentation polonaise inversée (*RPN* ou *reverse polish notation*). L'algorithme standard qui permet la conversion d'une expression écrite naturellement en une RPN s'appelle l'algorithme du shunting-yard (en raison d'une analogie avec le mécanisme qui sert à changer l'ordre des wagons dans un train). Une expression simple telle que `a + b * c` devient `a b c * +` et la suite de tokens précédente `17 - 24 / 4 * 3 + 2` devient `17 24 4 / 3 * - 2 +`.

La construction de cette représentation se fait "aisément" avec une pile, et sera réalisée par une fonction plus ou moins comme

```
Container<Token> parse(std::vector<Token>)
```

où le type `Container` sera un conteneur quelconque de `std::`, celui qui vous semble le plus adapté pour représenter une RPN.

La conversion en elle même se fait en recopiant la liste en entrée dans une autre tout en extrayant les opérateurs et en les réinjectant au bon moment. Pour chaque élément:

- si c'est un littéral, on l'envoie sur la sortie
- si c'est un opérateur,
  - on dépile sur la sortie tous les opérateurs en attente précédents qui sont de priorité supérieure ou égale

    - on place cet opérateur sur la pile d'attente

- à la fin on peut dépiler tous les opérateurs restants à la fin de la sortie.

Il existe une représentation alternative, voir la partie AST optionnelle si vous êtes curieux.

Évaluation de la RPN

Maintenant que vous pouvez transformer une chaîne de caractères en une suite de tokens, il est temps de procéder à l'évaluation. Avec la représentation dont vous disposez, cela se fait de manière semblable, avec une pile. Pour chaque élément de la RPN:

- si c'est un littéral, on le met sur la pile
- si c'est un opérateur (binaire):
    - on dépile deux éléments de la pile
    - on calcule le résultat
    - on empile le résultat sur la pile

À la fin seul reste le résultat final de l'expression.

Nombre réels, parenthèses

Jusque là, votre programme devait au moins lire les entiers (`[0-9]+`). Modifiez votre lexer pour permettre la compréhension par votre programme de nombres réels au format usuel (`[0-9]+.[0-9]*`).

Ajoutez également le support des parenthèses.

Séquence d'expressions

Faites en sorte que votre programme puisse lire plusieurs expressions à la suite, à raison d'une par ligne. De manière analogue à MATLAB®, si une ligne se termine par un ';' le résultat ne doit pas être affiché. En revanche on s'attend à voir le résultat sur la sortie sinon.

Nous vous proposons de considérer une classe d'objets `Program`, qui stockera cette suite d'expression, et dont l'éxécution correspondra à l'évaluation séquentielle de ses expressions. Par exemple, une entrée telle que

```
(4*2+3*6)/13;
8*9-1
3+1
```

doit produire la sortie

```
71
4
```

Réalisez cette classe qui prendra en entrée un flux d'entrée standard (c'est à dire en C++ `std::istream`), et affichera les résultats des expressions.

Mémoire

Ajoutez enfin une mémoire à votre code, de sorte que la définition d'une variable dans une expression permette son utilisation ultérieure. La syntaxe d'affectation est `ID = ...`, où `ID` sera un identificateur devant commencer par une lettre (ou éventuellement un caractère ne provoquant pas de conflit avec les opérateurs). La suite d'entrées

```
deuxpi = 2 * 3.1415926536;
rayon = 3*8;
circonference = deuxpi * rayon;
circonference
```

doit produire en sortie

```
150.79644738
```

Pour faire cela, nous vous demandons d'utiliser un tableau associatif (dictionnaire), présent dans la `std::`, comme `std::map` ou `std::unordered_map` (à vous de choisir laquelle utiliser).

Représentation en AST (optionnel)

L'algorithme Shunting Yard peut produire, de manière analogue à la RPN, un *abstract syntaxic tree (AST)*, qui est l'une des représentations internes usuelles des compilateurs communs que vous utilisez. Dans ce cas précis, cela revient à représenter votre expression comme un arbre, chaque nœud étant un opérateur et chaque feuille étant une valeur.

Si il vous reste du temps, et que ça vous intéresse, mettez en place cette structure à la place de la RPN. N'hésitez pas à utiliser une structure d'arbre simpliste, le but de l'exercice n'étant pas réellement là.

Annexes

Structuration d'un programme en C++

À l'instar du C, un programme C++ ne possède pas de structure spécifique. Cela dit, toujours comme le C, la plupart des projets sont constitués d'un système de build (GNU Make, CMake, SCons, Meson, Yaf, etc...) et d'un nombre arbitraire de couples source/header. Comme en C, les headers ont pour vocation de fournir uniquement des descriptions (déclarations) de types et de fonctions et d'être inclus dans les sources qui seront compilées fichier par fichier, donnant des fichiers objet (`.o`) qui seront ensuite reliés (linker).

Si vous êtes curieux, essayez de faire un `nm <un_fichier_objet> | c++filt` et vous observerez alors le contenu d'un fichier objet (les lignes marquées `U`, *undefined*, seront un jour peut-être la cause de vos futures erreurs de linker).

Un éxécutable est produit une fois que tous les liens ont pu être établis et qu'un symbole correspondant à la fonction `int main()` est trouvée. Voilà un exemple de projet utilisant CMake:

```
. projet
├── CMakeLists.txt
└── src
    ├── main.cpp
    ├── truc.cpp
    └── truc.h
```

dont le `CMakeLists.txt` contient

```
projet("Truc")
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_FLAGS
    "${CMAKE_CXX_FLAGS}
    -Wall -Werror -g -ggdb")
add_executable(truc
    src/main.cpp
    src/truc.cpp
    )
```

et qu'on peut compiler et lancer en faisant

```
mkdir build
cd build
cmake ..
make [-j NB_THREADS]
./truc
```

Tout ceci n'est qu'un exemple. Vous être libre d'utiliser ce qu'il vous plaît, cet aspect n'étant pas le but du projet. Assurez-vous tout de même que l'on puisse le compiler sans trop de difficulté.

Débugguage

Ce qui débugguait du C débugguera du C++. Vous avez donc entre autres à disposition les classiques

- valgrind
- lldb
- gdb
- ddd

Espaces de noms

Depuis le début de ce TP vous voyez partout des `std::`. `std` est l'espace de nom correspondant à la librairie standard du C++ (sous linux, si ça vous intéresse, vous pouvez observer des headers dans `/usr/include/c++/...`), et `::` correspond à l'*opérateur de résolution d'espace de nom*. Très intuitivement, cela permet de ranger les noms de symboles dans des cases ; en C tout devait être nommé différement (en général on préfixe par le nom du type, comme `vec2f_add` pour l'addition de deux vecteurs 2D en flottant).

Vous pouvez, dans les fichiers sources (mais pas les headers), vous épargner des `::` en utilisant les mot-clés `using namespace`. Techniquement, vous pouvez le faire dans les header également, mais vous polluerez l'environement de tout fichier qui incluera votre header.

Vecteurs

Un des conteneurs les plus pratiques et les usités est `std::vector<T>`, qui au passage illustre à merveille l'utilité des templates. Notez que ça n'a que très peu à voir avec les vecteurs rencontrés en maths.

```
std::vector<int> loto ;
loto.push_back( 4) ;
loto.push_back( 8) ;
loto.push_back(15) ;
loto.push_back(16) ;
loto.push_back(23) ;
loto.push_back(43) ;
```

Il a des caractéristiques intéressantes. C'est un tableau dynamiquement alloué, qui possède une capacité effective m (taille en mémoire) et une capacité utilisée n (nombre d'éléments valides, placés dans les n premiers emplacements). Lorsqu'il n'y a plus de place ($n = m$), le buffer est réalloué. Les éléments sont donc rangés en mémoire de manière contigue, ce qui permet un accès en $O(1)$. De plus, pour peu qu'on connaisse à l'avance le nombre d'éléments (ou une estimation) on peut utiliser la méthode `reserve(new_m)` qui réalloue le buffer en conséquence. Les ajouts suivants (`push_back()`, `emplace_back()`) n'induiront alors pas de réallocation.

Smart pointers

Vous avez normalement appris à désallouer la mémoire que allouée dynamiquement via des fonctions telles que `malloc()`, de la librairie standard C. La mise en place de ces `free()` étant au mieux fastidieuse, et au pire oubliée, est la cause principale des fuites mémoires.

En C++, les mécanismes de construction/destruction peuvent être exploités pour éviter cela et rendre plus explicite la notion de "propriété" ("à *qui* appartient la mémoire"). Les objets wrappers les plus connus sont les `std::unique_ptr<Type>` et `std::shared_ptr<Type>`, comme vous l'avez normalement vu en cours/TD. Un `unique_ptr` est l'unique propriétaire de sa ressource, et il la détruit à sa propre destruction. Le `shared_ptr`, par contre, maintient un compteur de références vers le même objet, et ne détruit la ressource que si il est le dernier à s'en servir. On pourrait imaginer l'implémentation simpliste

```
template <typename T>
class shared_ptr {
    // ...

    ~shared_ptr() {
        -- _ref_counter ;
        if (_ref_counter == 0)
            delete _ressource ;
    }

    // ...
}
```

Vous pouvez trouver la vraie implémentation dans un dossier du style `/usr/include/c++/9.2.0/bits/shared_ptr.h` (valide au moment de l'écriture, sur ArchLinux).

En bref, vous pouvez avantageusement vous servir des ces classes d'objets en remplacement des pointeurs C (appelés "raw pointers"), et c'est même une recommandation dans la communauté C++. Les opérateurs sont surchargés de sorte que vous puissiez les utiliser comme des pointeurs C. Par exemple:

```
struct Machin {
    Machin(int a)
    : _a {a} {
        // Nothing.
    }

    int _a ;
} ;

// Alloue un Machin et le stocke en interne
// (sur ce qui est en C++ le freestore).
std::unique_ptr<Machin> machin_ptr { 3 } ;

// Affiche 3.
std::cout << machin_ptr->_a << std::endl ;

// Le Machin est détruit en fin de scope,
// avec le unique_ptr.
```