Projet *Expressions*
C++

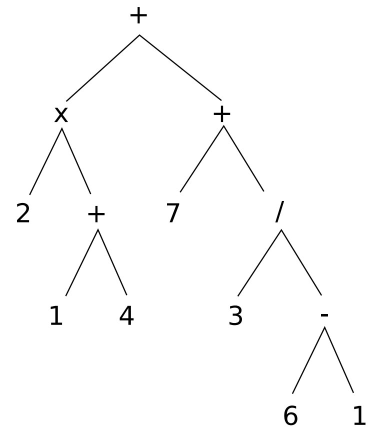
L'objectif de ce projet est de définir un ensemble de classes permettant de représenter et de manipuler des expressions arithmétiques. L'exercice est assez proche de ce que vous avez déjà fait en TP avec les Dipoles et les Figures récursives. Les expressions que nous allons considérer sont constituées de :

— d'opérandes numériques (constantes dans un premier temps),
— d'opérateurs binaires : $+$, $-$, $\times$ et $/$,

Une expression peut être représentée sous la forme d'un arbre dont :

— la racine est l'opérateur le moins prioritaire,
— les nœuds (internes) sont des opérateurs,
— les feuilles sont les opérandes.

Par exemple, l'expression ci-dessous est représentée par l'arbre à droite.



$$(2 \times (1 + 4)) + \left(7 + \frac{3}{6 - 1}\right)$$

Code fourni à télécharger : `https://mquinson.frama.io/ensr-cpp/template-projet-expressions.tgz`

★ Exercice 1: Première approche, sans héritage.

Nous allons d'abord essayer de résoudre le problème sans utiliser d'héritage. On utilisera une classe `FlatExpr` dotée d'un champ spécifique précisant le type d'expression. Par exemple, si ce champ vaut `'+'`, alors l'expression est une addition. Pour simplifier, on considérera uniquement le cas d'expressions constituées d'opérateurs binaires. Nous souhaitons pouvoir réaliser deux traitements sur une expression :

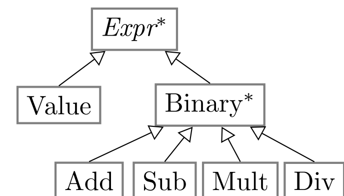
— `double FlatExpr::eval()` qui permet d'évaluer la valeur d'une expression (implémentée avec un `switch`). Ainsi, l'évaluation de l'expression représentée par l'arbre ci-dessus donne la valeur `17.6`
— `std::string FlatExpr::latex()` qui permet de représenter l'expression en LaTeX. On utilisera le symbole `\times` pour la multiplication, et on utilisera la macro `\frac` pour construire les fractions.
— `operator<<` permettant d'afficher (récursivement) un arbre dans un flux.

▷ **Question 1:** Complétez le code de la classe `FlatExpr` fournie afin de passer le premier test fourni. Pour compiler, lancez la commande `cmake .` (sans oublier le point final), puis compilez avec `make`. Pour voir l'affichage des tests qui ne passent pas, on utilisera `ctest --output-on-failure`

Bien que correcte, cette solution ne suit absolument pas la philosophie de la programmation orientée objet. Les chantres de l'objet expliquent que comme la logique est répartie entre plusieurs méthodes, l'ajout d'un nouvel opérateur demande de modifier plusieurs endroits du code. Cette façon de programmer porte un nom : il s'agit de shotgun surgery. Comme le nom l'indique, c'est une mauvaise habitude. Dans la suite de ce projet, on s'attachera donc à implémenter une solution *OOP-friendly* utilisant l'héritage.

★ Exercice 2: Héritage, liaison dynamique et templates.

L'arbre d'héritages de ce problème est assez similaire à celui du problème des dipôles. La principale différence est que nous allons écrire le code de façon générique de façon à pouvoir faire des opérations sur tous-types de données (entiers, doubles, complexes).



Comme nous allons écrire beaucoup de petites classes, le plus simple est de mettre l'implémentation des méthodes directement dans le fichier d'entête afin de ne pas avoir à sauter d'un fichier à l'autre trop souvent. Il n'y aura donc pas de `Expr.cpp`. Les bibliothèques implémentées de cette façon (comme `Catch2`, utilisée pour les tests en Prog2) sont dite *header-only*.

Le template fourni `Expr.hpp` est un peu complexe, mais vous avez tous les éléments pour le comprendre. La classe de base `Expr` est une template car on ne connaît pas le type de retour de la méthode `eval()`. L'opérateur `<<` est défini directement dans la classe bien que ce ne soit pas une méthode de cette classe. C'est ce que signifie le `friend` devant le prototype, et c'est la solution la plus simple dans le cas d'un template de classe. Cela reste un peu surprenant, et les curieux pourront lire la page de référence correspondante. Je n'ai pas trouvé comment écrire cela encore plus simplement.

▷ **Question 1:** Complétez la classe `Value` afin que son constructeur sauvegarde la valeur passée en paramètre, et que ses autres méthodes utilisent cette valeur.

▷ **Question 2:** Complétez la classe `Binary` afin que ses constructeurs sauvegardent les paramètres. Le constructeur recevant directement deux `T` doit créer des `Value` explicitement.

Dans cet exercice, on ne cherche pas à éviter les fuites mémoires, mais simplement à faire fonctionner les tests fournis. Votre code doit être lisible, mais il est très difficile (et parfaitement hors sujet) de l'écrire sous une forme qui permette aux tests de passer sans fuite détectée par valgrind. La propreté mémoire sera rajoutée à l'exercice suivant.

▷ **Question 3:** Complétez les classes `Add`, `Sub` et `Div` afin de pouvoir tester votre travail avec le test fourni.

Une fois votre code complet, supprimez le constructeur par défaut `Binary::Binary()` qui n'est là que pour permettre au template de compiler, mais qu'il n'est pas raisonnable de garder puisque les champs privés de cette classe n'ont pas de valeur par défaut raisonnable.

▷ **Question 4:** Écrivez une classe `Mult` (pour la multiplication) sur le modèle des autres opérations. Décommentez le test fourni pour la tester.

▷ **Question 5: (optionnelle)** Permettez l'usage de nombres complexes de la bibliothèque standard dans vos expressions (un test est fourni pour les `std::complex<int>`). Pour cela, `Value<std::complex<int>>::latex()` est un peu difficile à écrire car il n'existe pas de solution standard pour transformer un complexe standard en `std::string`. Une solution est d'implémenter une fonction templatée `myprint<T>` dont le prototype est fourni en commentaire, et implémenter manuellement les *spécialisations totales* nécessaires de cette fonction.

★ Exercice 3: Smart Pointers, surcharge d'opérateurs et variables.

L'objectif de cet exercice est d'améliorer et simplifier le code écrit à l'exercice précédent grâce à des constructions spécifiques au langage C++.

▷ **Question 1:** Complétez le template fourni dans le fichier `SmartExpr.hpp` pour passer le premier test du fichier `SmartExprTest.cpp` Pour rendre l'exercice moins répétitif, vous n'avez pas à implémenter l'opérateur de sérialisation (i.e. `operator<<`) dans ce cas-là. Seule la méthode `latex()` est utilisée dans les tests.

On défini le type `ExprPtr<T>` comme un raccourci bienvenu pour `std::shared_ptr<Expr<T>>` Utiliser `using` plutôt que `typedef` prend tout son sens ici, puisque notre alias de type est une template. Ce serait impossible avec un `typedef`.

Le code demandé est très comparable à ce que vous aviez écrit à l'exercice précédent, avec des différences subtiles qui rendent le copie/colle moins efficace que la réécriture manuelle. Les plus grandes différences seront dans la classe `Binary` qui doit maintenant stocker les opérandes sous forme de smart pointers, et au niveau des constructeurs des classes filles `Add`, `Sub`, etc. Pensez aussi à supprimer `Binary::Binary()` comme précédemment. Vérifiez l'absence de fuite mémoire avec la commande `valgrind ./smart_test` Si vous n'avez pas valgrind, ajoutez l'option `-fsanitize=address` dans les options de compilation (et d'édition de lien, si elles sont séparées) pour vérifier avec ASan à la place.

▷ **Question 2:** On souhaite maintenant surcharger les opérateurs classiques du C++ de façon à créer automatiquement une `Expr` à partir d'une expression C++. On souhaite pouvoir écrire des choses comme ci-contre. `expr->latex()` doit alors retourner la chaîne `"((5+4)*3)"`

```
ExprPtr<int> value(5);
ExprPtr<int> sum = value + 4;
auto expr = sum * 3;
```

L'objectif de cette question est donc d'écrire les opérateurs C++ nécessaires. Le premier problème est que comme dans le cas de l'opérateur de sérialisation, ces opérateurs ne peuvent pas être ajoutés à la classe servant de membre gauche à l'opération. Il est en effet impossible d'étendre la classe `std::shared_ptr<Expr<T>>` car `std::shared_ptr<>` est dans la bibliothèque standard. Il faut donc définir ces opérateurs sous forme de fonctions, un peu comme nous avons fait avec `operator<<()` dans les TP sans template de classes.

Le code fourni donne le prototype des opérateurs à définir pour l'opération d'addition (l'un avec un `ExprPtr<T>` en second paramètre et l'autre avec une valeur scalaire en second paramètre), ainsi que celui d'une fonction `make_value` permettant de convertir facilement une valeur scalaire en une `std::shared_ptr<Value<T>>`.

Décommentez le test *Operators for automatic parsing* une fois que vous avez écrit le code nécessaire.

▷ **Question 3:** On souhaite maintenant ajouter un nouveau type d'expression `Var` représentant des variables libres nommées. Ces expressions ne sont pas évaluables, puisque la valeur d'une variable n'est pas connue à priori. Il convient donc de lever une exception si la méthode `Var::eval()` est invoquée.

Ensuite, il convient d'ajouter une méthode virtuelle `subst()` (dont le prototype est donné en commentaire dans le code fourni) à toute la hiérarchie de classes. Cette méthode permet de remplacer les variables dont le nom correspond au paramètre par la valeur passée en paramètre. Dans tous les cas, cette méthode doit renvoyer un `ExprPtr` vers un nouvel objet. En effet, l'objet géré par le smart pointer n'ayant pas de référence à l'objet gestionnaire, il ne peut pas fabriquer de pointeur partagé sur l'objet gestionnaire.

Il est un peu dommage de devoir créer des copies des parties de l'arbre d'expression qui ne sont pas modifiées, mais c'est inévitable avec les smart pointers de la bibliothèque standard. Les `boost::intrusive_ptr` placent le compteur de référence dans l'objet géré et non dans un objet gestionnaire séparé. On peut alors créer un smart pointer partagé depuis l'objet géré, ce qui permet de parfois réutiliser cet objet au lieu d'imposer sa duplication systématique. Ce mécanisme est cependant parfaitement hors programme et cité ici pour votre curiosité.

Décommentez le test *single variable* et vérifiez avec `valgrind` l'absence de fuite mémoire.

▷ **Question 4: (optionnelle)** Ajoutez une surcharge non virtuelle de la fonction `Expr::subst(...)` permettant de passer le test *multiple variables* **sans modifier le test**. Une partie de la question est de trouver quel type de collection elle doit prendre en paramètre pour permettre au test de fonctionner. Il convient ensuite d'implémenter cette fonction.

▷ **Question 5:** On souhaite maintenant implémenter une méthode virtuelle `Expr::simplify()` retournant une forme simplifiée de l'expression courante. En particulier, si l'expression ne contient pas de variables libres, il faut la remplacer par une valeur unique. Quand certaines parties de l'expression contiennent des variables, il faut quand même simplifier les autres parties si possible. Il peut être utile d'ajouter d'autres méthodes en plus de `simplify()` pour résoudre cette question.

★ Logistique du projet

Ce projet est à réaliser en binôme. Le rapport et les sources du programme doivent être placés dans un **projet git privé** sur le gitlab de l'istic (ou un autre hébergeur comme gitlab.com ou framagit). Vous ajouterez les deux enseignants du module (`mquinson` et `kbarrere`) à la liste des développeurs de votre projet. Le rapport en pdf doit être dans le git (ou en artefact du CI), et l'intégralité de votre programme doit être écrit en C++.

Code. Vous porterez un soin particulier à l'écriture du code. *Pensez à nettoyer pour ne pas rendre un brouillon.* Attachez-vous à factoriser le maximum de code possible dans la hiérarchie de classes, sans sacrifier la lisibilité du code. Celui-ci doit être commenté et bien écrit pour être facilement lisible. Vous êtes libres d'ajouter des méthodes et classes à votre projet pour améliorer sa lisibilité. Il est rappelé que l'on écrit un programme pour que d'autres humains puissent le lire, et (accidentellement seulement) pour que les machines puissent l'exécuter. Nous passerons plus de temps à lire votre code qu'à exécuter votre programme. Ajouter des tests unitaires est apprécié, surtout si vous pouvez justifier de leur intérêt dans votre rapport : ce qu'ils testent et en quoi ils diffèrent de ceux existant.

Rapport. Vous devez apporter une réponse claire et détaillée aux questions du sujet le nécessitant, sans reprendre trop de code. Pour les questions optionnelles, vous pouvez donner une idée de solution même si vous ne l'avez pas implémenté. Le rapport (de 5 pages maximum) doit contenir une courte introduction générale, une réponse pour chaque question explicitant et justifiant vos choix d'implémentation, une synthèse générale, et une bibliographie précise de toutes les sources (sites, livres ou individus) qui vous ont aidé, avec quelques mots de ce que vous en avez retiré. Indiquez dans votre rapport le temps passé par chaque membre du binôme sur ce projet : nous utilisons cette information pour ajuster le sujet d'une année sur l'autre. Si vous avez des propositions d'amélioration pour ce projet, ajoutez-les en annexe du rapport.

Triche. La frontière est mince entre *l'oubli* de sources et le plagiat. N'oubliez rien, ne trichez pas. Voler des idées, maquiller des résultats ou mal citer ses sources sont des péchés mortels en recherche. Tricher, c'est reconnaître qu'on va devoir changer de métier, car ceux pris à tricher en science n'ont pas de seconde chance.

En revanche, il est conseillé de discuter du projet et d'échanger des idées avec tous vos collègues. Pour ne pas franchir la ligne jaune, **ne lisez jamais de code écrit par un autre groupe, et ne permettez pas aux autres groupes de lire votre code**. Vous ne pouvez rendre que du code écrit par vous-même, et vous devez détailler brievement vos sources d'inspiration sur internet dans la partie bibliographie de votre rapport. Soyez spécifiques : si par exemple vous avez trouvé des réponses sur Stack Overflow, pointez les pages utilisées.

Collaboration. La science moderne étant un sport d'équipe, nous évaluerons votre capacité à collaborer. Un projet composé de parties individuelles mal assemblées recevra une évaluation sévère, et nous attendons que tous les membres du projet commitent du code dans l'historique du projet. Savoir aider les autres et demander de l'aide efficacement sont des compétences fondamentales pour être scientifique de haut niveau. Quel que soit le contexte, collaborer efficacement est indispensable pour parvenir à produire des logiciels intéressants.

Nous ferons un "checkout" de vos projets git le **vendredi 19 avril à 19h** heure de Paris.

Il n'y aura aucune extension. Attention au commit de dernière minute qui casse tout.

★ Annexe : le problème de l'expression.

Ce projet porte sur ce que l'on nomme *Expression problem* [1] dans la communauté des langages de programmation. Il s'agit de définir un type abstrait de données (TAD) typé statiquement, mais extensible de deux façons orthogonales : on veut d'une part permettre d'ajouter de nouvelles fonctions à notre TAD, et d'autres part de faire en sorte que le TAD soit utilisable pour d'autres types élémentaires. Par exemple, ajouter une nouvelle fonction au TAD `Expr` pourrait consister à implémenter le calcul de la dérivé ou de l'intégrale d'une expression donnée. Ajouter des types élémentaires à notre TAD consisterait à implémenter de nouveaux opérateurs comme la moyenne ou les fonctions trigonométriques. Ce problème revient donc à concilier le typage statique avec deux objectifs :

1. Utiliser un type fourni par une bibliothèque externe, et introduire de nouvelles opérations sur ces types ;
2. Utiliser des opérations fournies par une bibliothèque externe, et introduire de nouveaux types de données sur lesquels les opérations de la bibliothèque peut s'appliquer.

Dans les langages orientés objets, le premier objectif est difficile, puisqu'il faut modifier tous les types fournis pour ajouter la nouvelle fonction. C'est ce qu'il faudrait faire pour réajouter l'opérateur de sérialisation à toutes les sous-classes de `Expr<>` dans l'exercice 3. En revanche, le second objectif du problème est trivial dans un langage OO, puisqu'on peut toujours sous-classer les types fournis par la bibliothèque pour ajouter définir de nouveaux types.

L'organisation typique dans un langage fonctionnel est celle vue au premier exercice : On définit une fonction par opération du TAD, et dans chaque fonction, on utilise un `switch`, un `match` ou équivalent pour différencier entre les types sur lesquels l'opération peut s'appliquer. Le premier objectif du problème de l'expression est alors facile puisqu'il suffit d'ajouter de nouvelles fonctions à l'ensemble, tandis que le deuxième objectif devient difficile car il faut changer toutes les fonctions existante quand on ajoute un type de données.

On peut donc dire qu'il s'agit de Shotgun Surgery dans les deux cas, mais pas sur le même axe d'extension.

De nombreuses solutions ont été proposées dans la littérature, ainsi que dans les différents langages de programmation. Ce rapport technique [2] par les auteurs du langage Scala donne une vue d'ensemble de la littérature et propose des pistes pour résoudre le premier objectif en suivant l'approche OO, ainsi que des pistes pour résoudre le deuxième objectif en suivant l'approche fonctionnelle.

Si on laisse de coté la contrainte de typage statique (et la sécurité qui en découle), on peut résoudre ce problème avec l'approche dite de *monkey patching* des langages dynamiques comme Python ou Javascript. Cela consiste à injecter dynamiquement du nouveau code dans une classe existante, au runtime. En plus des problèmes de performance dûs à la dynamicité (qui rendent cette approche incompatible avec la philosophie C++), le code modifié ainsi devient vite embrouillé et difficile à maintenir. Les classes ouvertes du langage Ruby sont une tentative pour rendre ces modifications dynamiques plus propres et plus contrôlables, même si cela ne corrige pas les performances.

En Lisp ou Julia, on peut utiliser des multi-méthodes [3] pour résoudre le problème de l'expression (Closure utilisé dans l'article est un Lisp moderne, mais comme il n'y a pas typage statique en Lisp, aucune solution complète au problème ne peut s'écrire en Lisp). Les multiméthodes sont des méthodes de classes qui s'écrivent comme les fonctions que l'on utilise en C++ pour définir les opérateurs. Ces fonctions sont hors du corps de la classe et les paramètres explicitent les receveurs de l'appel. On défini ensuite plusieurs surcharges au sens C++ pour les différents types de receveur à accepter. La différence de ce *double dispatch* avec la surcharge C++ est qu'il s'agit ici de polymorphisme dynamique : une vtable complexe est construite pour décider à l'exécution de la méthode à utiliser en fonction du type dynamique des différents paramètres surchargés.

Si on tente d'utiliser la même approche en C++ (en utilisant la notation fonctionnelle `meth(obj, ..)` sur des fonctions surchargées au lieu de la notation pointée `obj.meth(..)` plus idomatique du C++), le résultat ressemble à de l'overwrite. Une surcharge pour la classe ancêtre masque les surcharges des classes filles puisque tout est géré à la compilation. La surcharge du type statique (pointeur vers l'ancêtre) est utilisée, et les classes filles ne peuvent pas redéfinir de comportements ainsi définis. Le pire est que ce comportement n'est pas systématique. Si on écrit `eval(new Plus(.., ..))`, le type statique est `Plus*` et la bonne surcharge sera utilisée. Mais rares sont les cas où le compilateur peut déterminer le type statique attendu par le programmeur.

Le langage Closure propose des mécanismes pour ajouter une méthode au "protocole" définissant l'interface d'une classe de façon statique mais séparée de la définition de cette classe. C# propose la notion d'*extension method* pour augmenter statiquement une classe (avec ce qui ressemble à une fonction surchargée prenant un objet de la classe en premier argument) tout en permettant l'invocation de ces méthodes avec la notation pointée des objets. Mais comme c'est un mécanisme statique, cela ne constitue pas non plus une solution au problème.

Et non, pour une fois, je ne vais pas expliquer les *trait objects* de Rust. C'est relativement complexe, et ça ne résout pas complètement le problème non plus.

1. Expression problem : `https://en.wikipedia.org/wiki/Expression_problem`
2. `http://lampwww.epfl.ch/papers/IC_TECH_REPORT_200433.pdf`
3. `https://eli.thegreenplace.net/2016/the-expression-problem-and-its-solutions/`