


- [Aucun ami connecté](#)
- [863 Zéros connectés](#)
- [280 626 Zéros inscrits](#)
- 
- [pylaterreur](#)
 - 
 - [Profil](#)
 - [Identifiants](#)
 - [Études](#)
 - [Réglages](#)
 - [Contacts](#)
 - [Commandes](#)
 -
- [Lisez vos Messages Personnels](#)



De la généricité contrôlée à la compilation, grâce à l'utilisation de conteneurs génériques, listes de types et assertions statiques 🖋️



Par  [pylaterreur](#)

Difficulté : Difficile  Durée d'étude : 3 heures



Salut, dans ce tutoriel, on va coder un système permettant de créer, d'ajouter et de récupérer des éléments dans un AST représentant un header C++ (simplifié, bien sûr, c'est un minituto).



Un AST est un arbre représentant différents éléments, de manière hiérarchique. Si vous avez des doutes sur ce qu'est un AST, je vous conseille la lecture de l'article Wikipedia [AST](#) (en anglais) sur la question.

Pour ce faire, on utilisera des conteneurs génériques ([boost::any](#)), ainsi que des listes de types ([Loki::Typelist](#)) et des assertions statiques ([Loki::CompileTimeError](#)).

[Retour en haut](#)

Sommaire du tutoriel :



- [Problème](#)
- [Solution](#)

- [Tests](#)

[Retour en haut](#)

Problème

Pour cela, voila un sujet, assez simple : nous allons écrire un système capable de stocker l'ast d'un header C++ simplifié.

Nous aurons les classes suivantes, dans le namespace AST, avec des contraintes (en **vert**, la liste des éléments que l'on peut y stocker, et en **rouge** les éléments qu'on ne peut PAS y stocker)

- **AST** : peut être anonyme ; **Namespace**, **Class**, **Enum** ; **AST**
- **Namespace** : doit avoir un nom ; **Namespace**, **Class**, **Enum** ; **AST**
- **Class** : doit avoir un nom ; **Class**, **Enum** ; **AST**, **Namespace**
- **Enum** : peut être anonyme ; **AST**, **Namespace**, **Class**, **Enum**

Pour insérer une copie de l'élément A dans un élément B, nous utiliserons la syntaxe `B << A`. Pour récupérer une référence sur l'élément A contenu par l'élément B, nous utiliserons la syntaxe `B >> C`, avec C un objet du même type et du même nom que A.



On n'écrira qu'un seul `operator<<` et un `operator>>` dans notre code. C'est là que les typelists vont nous aider à séparer vérification de types et types autorisés.

Je vous donne ci-après des exemples d'utilisations, valides ou invalides.

Exemple 1 : valide

On devra pouvoir faire compiler et fonctionner le code suivant :

Code : C++ - exemple 1 - [Sélectionner](#)

```
{  
  
    AST::AST ast;  
  
    ast << AST::Namespace("namespace1");  
  
    (ast >> AST::Namespace("namespace1")) << AST::Class("class1");  
  
    (ast >> AST::Namespace("namespace1")) >> AST::Class("class1");  
}
```

Exemple 2 : invalide

Il faudra interdire la compilation du code suivant :

Code : C++ - exemple 2 - [Sélectionner](#)

```

{
    AST::AST ast;
    ast << AST::Namespace("namespace1");
    (ast >> AST::Namespace("namespace1")) << AST::Class("class1");

    ((ast >> AST::Namespace("namespace1")) >> AST::Class("class1"))
    << AST::Namespace("namespace2");
}

```

Exemple 3 : invalide

Le code suivant, est correct en termes de règles, mais doit générer une exception :

Code : C++ - exemple 3 - [Sélectionner](#)

```

{
    AST::AST ast;
    ast << AST::Namespace("namespace1");
    (ast >> AST::Namespace("namespace1")) << AST::Class("class1");

    (ast >> AST::Namespace("toto")) >> AST::Class("class2");
}

```

[Retour en haut](#)

Solution

Avant de continuer, je vous conseille de lire la documentation de Loki, au moins sur `Loki::TL::MakeTypelist`, `Loki::TL::IndexOf`, `Loki::CompileTimeError`. Les 2 premières servent pour les typelists, la première pour créer une typelist, la deuxième pour, entre autres choses, vérifier si un type existe dans une typelist. La dernière effectue une assertion statique, ce qui permet d'avoir des erreurs à la compilation, et non au moment de l'exécution.

Nous allons utiliser un `std::vector<boost::any>` pour stocker un élément dans un autre. Pour récupérer l'élément stocké, on utilisera `boost::any_cast` (qui génère une exception si le type qu'on spécifie n'est pas le type de l'objet stocké, mais on trouvera une solution pour que ça n'arrive jamais 😊).



Puisque nous allons renvoyer une référence sur un élément du `std::vector`, on ne peut pas stocker directement les éléments, il faut les stocker sous forme de pointeurs sur le tas (avec l'opérateur `new`). En effet, si on garde une référence sur un élément et que le vector est agrandi, la référence sera alors invalide. Pour éviter les leaks, on utilisera un smart pointer boost : `shared_ptr`. Par contre, il faudra garder en tête que si on utilise un élément contenu dans un élément père, il faut que son parent ne soit pas déjà détruit.

On a donc :

- un `<<` qui stocke un élément, qui ne peut fail qu'à la compilation
- un `>>` qui peut lever une exception dans le cas où l'élément que l'on veut récupérer n'existe pas

dans l'élément père.

Ces 2 opérateurs pourraient recevoir un `boost::any`. Cependant, nous voulons un maximum de vérification à la compilation, et le moins possible au runtime. Un paramètre template, correspondant au type reçu par ces méthodes, est une solution. On regarde si le type de la référence constante qu'on reçoit est dans une typelist préalablement définie (une `AST::Class` aurait une typelist `{AST::Class,AST::Enum}`).

En terme d'attributs, on va avoir une `std::string` pour le nom de l'élément, et un `std::vector<boost::any>` pour les éléments fils.

Pourquoi pas faire une classe de base, `BasicElement`, qui contienne ces 2 attributs et ces 2 opérateurs, ainsi qu'un constructeur pour initialiser la string (avec un paramètre par défaut égal à chaîne vide), une méthode `name()` pour récupérer le nom, et une méthode `name(const std::string &)` pour modifier le nom ? Cette classe aurait un paramètre template : le type "fils" (`AST::AST`, `AST::Class`, ...). La typelist des types autorisés sera donc un typedef dans chacun des types "fils".

Pour résumer, nous avons donc :

- `<Element>`, la classe de base, avec `<<` et `>>` templates, le type étant vérifié par rapport à la typelist contenue dans le type template `Element`, sous forme de typedef.
- `BasicElement<Class>`, la classe permettant la création de `Class`, contenant une typelist autorisant `Class` et `Enum`
- les autres classes basées en gros sur le modèle de la classe `Class` (un typedef de la liste des types fils autorisés, quelques using dont les `operator<<` et `operator>>`)

Code : C++ - [Sélectionner](#)

```
#ifndef AST_HPP_
# define AST_HPP_

# include <string>
# include <vector>
# include <utility>

# include <boost/any.hpp>

# include <loki/Typelist.h>
# include <loki/static_check.h>

# include <boost/shared_ptr.hpp>

    boost::shared_ptr;

    < , >
    hasSameType
{
    { value = };
};

    < >
    hasSameType<T, T>
{
```

```

        { value =          };
};

        <
        >
        toPointer
{
        T* Type;
};

        AST
{
        ;
        ;
        ;

        <
        >

{
        :
        BasicElement(          std::string &name = "") : _name(name)
        {}

        ~BasicElement()
        {}

        <
        >
        BasicElement&          <<(          C &c)
        {
                Loki::CompileTimeError<Loki::TL::IndexOf<
Element::AuthorizedTypes::Result, C>::value != -1> IsAuthorized;

                shared_ptr<C> p(          C(c));
                _elements.push_back(p);
                (*          );
                IsAuthorized invalid_type;
                (void)invalid_type;
        }

        <
        >
        C&          >>(          C &c)
        {
                Loki::CompileTimeError<Loki::TL::IndexOf<
Element::AuthorizedTypes::Result, C>::value != -1> IsAuthorized;
                (std::vector<boost::any>::iterator b(_elements.begin()),
e(_elements.end())); b != e; ++b)
                {
                        (b->type() ==          (shared_ptr<C>) && c.name() ==
(boost::any_cast<shared_ptr<C> >(*b))->name())
                        {
                                (* (boost::any_cast<shared_ptr<C> >(*b)));
                        }
                }

                1;
                IsAuthorized invalid_type;
                (void)invalid_type;
        }

        std::string &name()
        {
                (_name);
        }

        void name(          std::string &name)
        {

```

```

        _name = name;
    }

    :
    std::string _name;
    std::vector<boost::any> _elements;
};

    :      BasicElement<AST>
{
    :
    Loki::TL::MakeTypelist<Namespace, Class>
AuthorizedTypes;

    BasicElement::      <<;
    BasicElement::      >>;
    BasicElement::name;

    AST(      std::string &name = "") : BasicElement<AST>(name)
    {}
    ~AST()
    {}
};

    :      BasicElement<Class>
{
    :
    Loki::TL::MakeTypelist<Class> AuthorizedTypes;

    BasicElement::      <<;
    BasicElement::      >>;
    BasicElement::name;

    Class(      std::string &name) : BasicElement<Class>(name)
    {}

    ~Class()
    {}
};

    :      BasicElement<Namespace>
{
    :
    Loki::TL::MakeTypelist<Namespace, Class>
AuthorizedTypes;

    BasicElement::      <<;
    BasicElement::      >>;
    BasicElement::name;

    Namespace(      std::string &name) :
BasicElement<Namespace>(name)
    {}
    ~Namespace()
    {}
};

    :      BasicElement<Enum>
{
    :
    Loki::TL::MakeTypelist<> AuthorizedTypes;

    BasicElement::      <<;
    BasicElement::      >>;
    BasicElement::name;

```

```

        Enum(          std::string &name = "") : BasicElement<Enum>(name)
        {
            ~Enum()
            {}
        };
    }

#endif

```

Dans les `<< et >>`, la ligne `Loki::CompileTimeError<Loki::TL::IndexOf<Element::AuthorizedTypes::Result, C>::value != -1> IsAuthorized;` définit un type `IsAuthorized`, qui sera considéré comme **invalide** par le compilateur dans le cas où le type template `C` n'est pas dans la typelist de l'élément en question. Pour générer l'erreur, la création d'un typedef ne suffit pas. Par contre la création d'objet de ce type, si. C'est pour ça que l'on crée des objets de type `IsAuthorized`, auxquels on donne un nom explicite pour faire générer une erreur presque compréhensible par un humain. Ces objets sont déclarés après le return (ou le throw), ce qui ne posera pas de problème de performance au moment de l'exécution.

[Retour en haut](#)

Tests

Reprenons les exemples de la première sous-partie.

Exemple 1

Code : C++ - main1.cpp - [Sélectionner](#)

```

#include "ast.hpp"

int main(void)
{
    AST::AST ast;

    ast << AST::Namespace("namespace1");

    (ast >> AST::Namespace("namespace1")) << AST::Class("class1");

    (ast >> AST::Namespace("namespace1")) >> AST::Class("class1");
}

```

Sous Linux, la commande `g++ -W -Wall main1.cpp && ./a.out` n'output rien. Ca compile, et ça marche !

Exemple 2

Code : C++ - main2.cpp - [Sélectionner](#)

```
#include "ast.hpp"

int main(void)
{
    AST::AST ast;
    ast << AST::Namespace("namespace1");
    (ast >> AST::Namespace("namespace1")) << AST::Class("class1");

    ((ast >> AST::Namespace("namespace1")) >> AST::Class("class1"))
    << AST::Namespace("namespace2");
}
```

g++ -W -Wall main2.cpp && ./a.out déclenche l'erreur de compilation suivante :

Code : Console - Sortie standard et d'erreur de la compilation et l'exécution du binaire de main2.cpp -
[Sélectionner](#)

```
In file included from main2.cpp:1:0:
ast.hpp: In member function 'AST::BasicElement<Element>& AST::BasicElement<Element>::operator<<<
main2.cpp:9:97:   instantiated from here
ast.hpp:59:20: error: 'invalid_type' has incomplete type
```

Exemple 3

Code : C++ - main3.cpp - [Sélectionner](#)

```
#include "ast.hpp"

int main(void)
{
    AST::AST ast;
    ast << AST::Namespace("namespace1");
    (ast >> AST::Namespace("namespace1")) << AST::Class("class1");

    (ast >> AST::Namespace("toto")) >> AST::Class("class2");
}
```

g++ -W -Wall main3.cpp && ./a.out déclenche l'erreur suivante, à l'exécution de ./a.out

Code : Console - [Sélectionner](#)

```
terminate called after throwing an instance of 'int'
Aborted
```

[Retour en haut](#)

J'espère que cet exemple pratique a pu vous aider dans vos conceptions C++.

A l'heure où j'écris ces lignes, l'exemple que nous venons de voir est encore frais pour moi, en effet, c'est un problème réel que j'ai rencontré dans mon projet de fin d'études (qui consiste en un langage de modélisation graphique de C++, ainsi que son éditeur, que mon groupe de projet a nommés tous les deux C++ML, à partir des langages UML et C++).

Avant de nous quitter, je vous conseille la lecture du livre Modern C++ Design d'Andrei Alexandrescu, l'auteur de la bibliothèque Loki. Loki et ce livre sont intimement liés, mais je vous laisse le découvrir par vous-même 😊 .