La meta-programmation en C++

Date de publication : 24/11/2004 , Date de mise a jour : 01/09/2005

Par Laurent Gomila

Dans cet article, nous allons explorer une partie peu connnue mais pourtant extrêmement puissante du C++ : la meta-programmation.

1. La base de la meta-programmation : les templates

La meta-programmation en C++ s'appuie sur un concept puissant de ce langage : les templates. Avant d'aller explorer plus en profondeur la notion de meta-programmation, je ne peux donc que vous conseiller de revoir vos bases sur les templates.

Si vous n'avez vraiment jamais entendu parler de templates, vous pouvez aller jeter un oeil à nos cours de C++ en ligne, ainsi qu'à la rubrique sur les templates dans notre FAQ C++. Tous deux sont très bien faits et vous donnerons les bases nécessaires. Attention cependant, si vous débutez complétement en C++ vous aurez peut-être du mal à suivre cet article, certaines techniques que nous allons decouvrir ici ne sont pas évidentes à appréhender.

Bien, cette mise au point effectuée, passons à ce qui nous interesse dans cet article : la meta-programmation.

2. Mais... Qu'est-ce donc que la meta-programmation ?

Avec l'arrivée des templates, le C++ s'est doté d'un outil aussi puissant que complexe. En effet au travers de fonctions et classes templates nous sommes maintenant capables d'écrire du code qui sera interprété non pas à l'execution, mais pendant la compilation. Un peu comme les macros en C, mais juste un peu : les templates sont beaucoup plus complexes et puissantes.

Habituellement on se sert des templates pour créer des classes ou fonctions génériques, c'est-à-dire qui acceptent n'importe quel type pour peu qu'il soit compatible avec le code produit. Prenons un exemple typique : la fonction Max, qui fonctionnera pour tout type dont les valeurs peuvent être comparé à l'aide de l'opérateur >.

```
template <class T>
const T& Max(const T& x, const T& y)
{
    return x > y ? x : y;
}

int i = Max(5, 9);
double d = Max(1.5, 8.6);
std::string s = Max(std::string("bonjour"), std::string("hello"));
```

Ici nous n'avons écrit qu'une version de notre fonction Max. Pourtant, lorsque le compilateur va rencontrer notre code, il va générer les spécialisations correspondantes pour les types utilisés : **int**, **double** et **std::string**. C'est exactement comme si nous avions écrit ceci :

```
const int& Max(const int& x, const int& y)
{
    return x > y ? x : y;
}

const double& Max(const double& x, const double& y)
{
    return x > y ? x : y;
}

const std::string& Max(const std::string& x, const std::string& y)
{
    return x > y ? x : y;
}
```

... Sauf que nous ne l'avons pas écrit ! Nous avons relégué cette tâche ingrâte au compilateur, qui s'en charge d'ailleurs mieux et plus vite que nous.

Et bien la meta-programmation c'est exactement ça : manipuler des données génériques et mettre à contribution notre compilateur pour générer le code final voulu. Ainsi tout ce que nous allons voir par la suite se passe principalement **pendant la phase de compilation**, l'execution des programmes ne servira qu'à vérifier que notre compilateur a bien généré le résultat attendu. L'un des inconvénients direct de ce genre de programmation est donc un temps de compilation nettement accru, mais ce n'est rien en comparaison de tout le temps gagné au final : aussi bien à l'écriture du code que pendant l'execution du programme. Un autre inconvénient est qu'ici tout ce que nous allons écrire devra être connu du compilateur : certaines données que nous allons manipuler ne pourrons assurément pas être entrées par l'utilisateur lors de l'execution.

Très bien, nous pouvons maintenant utiliser nos templates pour écrire des fonctions et classes acceptant plusieurs types. Belle économie de temps et de réecriture, certes. Mais... est-ce bien tout ? Ne pouvons-nous pas aller encore plus loin ? La réponse est évidemment oui (sinon cet article n'aurait pas de raison d'être), et pour vous mettre en appétit je vous donnerais simplement quelques exemples de ce que l'on peut faire en meta-programmation :

1. On peut faire faire tout un tas de calculs mathématiques à notre compilateur : sinus, exponentielle, factorielle, ... et booster les performances de ce genre de calculs généralement lents.
2. On peut réécrire des programmes habituellement pauvres en performances et leur donner un second souffle, par exemple le tri-bulle (*bubble sort*).
3. On peut procéder à des optimisations intéressantes et certainement insoupçonnées sur nos calculs, notamment le calcul matriciel avec les *expression templates*.
4. On peut générer automatiquement du code, notamment pour implanter des *design patterns* tels que les fabriques ou le visiteur sans effort et en gardant un typage fort.
5. On peut écrire des outils puissants tels que des analyseurs syntaxiques.

Aux vues de ces quelques exemples, on peut déjà diviser la meta-programmation en deux parties : l'une portant sur le calcul et l'optimisation mathématique (templates principalement à base d'entiers), et l'autre portant sur la génération automatique de code (templates à base de types).

3. Calcul et optimisation mathématique

L'un des exemples les plus célèbres et "simples" de fonction mathématique remaniée à la sauce template est certainement la factorielle :

```
template<unsigned int N> struct Fact
{
    enum {Value = N * Fact<N - 1>::Value};
};
template<> struct Fact<0>
{
    enum {Value = 1};
```

```
};

// Ici, x vaudra 24 avant même que vous ne lanciez votre programme - coût à l'execution : 0 sesterce
unsigned int x = Fact<4>::Value;
```

Comme vous pouvez déjà le voir dans ce petit extrait, la syntaxe de la meta-programmation est un peu particulière, pleine de spécialisations et d'enum comme vous n'avez pas l'habitude d'en voir. Nous allons donc voir quelques règles de base de cette syntaxe un peu particulière.

3.1. Quelques règles de meta-programmation

Comme tout type de programmation, la meta-programmation se base sur certaines règles et une certaine syntaxe. La règle de base est que **tout** doit **toujours** rester au maximum connu du compilateur. Si vous introduisez du code ne pouvant être évalué qu'à l'execution, soit votre meta-programme est cassé et le compilateur ne se gêne pas pour vous le faire remarquer, soit c'est voulu (on ne peut pas toujours tout précalculer), et dans ce cas cela s'en ressentira sur l'execution du programme. Il faut donc suivre assez rigoureusement certains principes de base, mais vous verrez : une fois que l'on s'y est habitué un peu cela coule tout seul après. Voici donc quelques éléments qui vous aideront à écrire vos chefs d'oeuvre de meta-programmation.

1- Utiliser struct & enum pour travailler sur des constantes entières

Pour notre première règle de meta-programmation, nous allons d'ores et déjà frapper un grand coup : exit les fonctions ! En effet rappelez-vous, tout ce qui est laissé à l'execution n'est plus exploitable par notre bon vieux compilateur, et ce sont des informations que nos meta-programmes ne pourront jamais exploiter. Nous pouvons donc oublier de passer nos paramètres à des fonctions, tout comme récupérer nos résultat via la valeur de retour. Mézalors, comment faire ? Eh bien nos fonctions seront désormais des **classes** (ou plutôt des **structures**, c'est équivalent en C++ mais cela nous évitera d'écrire un fastidieux "*public :*" à répétition), et nos valeurs de retour ne seront rien de moins que des **types énumérés** (enum).

```
Exemple de fonction évaluée à la compilation
// Une fonction totalement inutile : elle renvoit son paramètre
int Identite(int N)
{
    return N;
}
unsigned int x = Identite(5); // x ne sera connu que lorsque le programme sera executé

// La même en meta-programme
template<int N> struct Identite
{
    enum {Value = N;}
};
unsigned int x = Identite<5>::Value; // la valeur de x est connue de notre compilateur
```

Dans notre second exemple, x est connu du compilateur et pourra donc très bien être passé comme paramètre template à un meta-programme plus évolué par exemple. Notre premier x quant à lui est totalement inexploitable, sa valeur ne sera connue que lors de l'execution du programme.

2- Ne pas oublier l'inlining

Bien entendu nous ne travaillerons pas toujours avec des types entiers et des constantes. Je vous ai dit tout à l'heure qu'on ne pouvait pas renseigner notre compilateur en ce qui concerne les nombre réels et les variables connues seulement à l'execution, et bien pas tout à fait. Nous disposons d'un moyen assez efficace : *l'inlining*. Une fonction inline aura le même comportement qu'une macro : un appel à une telle fonction se verra remplacé par le code correspondant, sans les inconvénients des macros (comme l'évaluation multiple des paramètres entre autre). Attention toutefois, l'inlining est totalement contrôlé par le compilateur, si votre fonction ne lui plait pas il ne sera pas obligé de l'inliner. Même chose lorsque vous compilerez en mode debug ou sans optimisation : il y a de fortes chances que vos fonctions ne soient pas inlinées. Mais rassurez-vous : nos fonctions mathématiques ne feront pas plus d'une ou deux lignes de code, ce qui en fait des candidates parfaites pour l'inlining, et il n'y a donc aucune raison que votre compilo n'en veuille pas.

```
Exemple de fonction inline
inline int Add(int X, int Y)
{
    return Y == 0 ? X : Add(X, Y - 1) + 1;
}

int Somme = Add(5, 3);
```

Ce code sera probablement remplacé par **int Somme = 5 + 1 + 1 + 1** c'est-à-dire **int Somme = 8**. S'il n'avait pas été inliné, il aurait abouti à des appels successifs à Add, ce qui est très coûteux.

3- Les conditions : spécialisation et opérateur ternaire

Dans nos programmes nous aurons très certainement besoin de tester certaines conditions. Mais si nous utilisons le

classique **if / else**, celui-ci ne pourra être évalué qu'à l'execution. Une solution est de créer une classe dont le paramètre template est un booléen, puis de la spécialiser pour **true** et pour **false**, comme dans l'exemple ci-dessous :

```
Exemple de conditionnelle
template<bool Condition> struct Test {};

template<> struct Test<true>
{
    static void Do()
    {
        DoSomething();
    }
};

template<> struct Test<false>
{
    static void Do()
    {
        DoSomethingElse();
    }
};

// Code habituel
if (Condition)
    DoSomething();
else
    DoSomethingElse();

// Avec notre structure Test
Test<Condition>::Do();
```

Je me répète mais c'est le principe de base de la meta-programmation : ici *Condition* devra pouvoir être évaluée par le compilateur, sinon nous ne pouvons bien sûr pas utiliser ce mécanisme.

Mais pour tester un booléen, nous pouvons également utiliser l'opérateur ternaire **?:** qui est parfois bien plus léger à écrire :

```
Exemple de conditionnelles bis
template <int I> struct NumericTests
{
    enum
    {
        IsPair = (I % 2 ? false : true),
        IsZero = (I == 0 ? true : false)
    };
};

bool b1 = NumericTests<45>::IsPair; // 0 (false)
bool b2 = NumericTests<0>::IsZero; // 1 (true)
```

4- La récursion et la spécialisation pour boucler

Autre élément syntaxique d'importance : les boucles. Je crois que vous l'aurez compris maintenant, même pas la peine de penser aux **for** et autres **while**, il va falloir faire sans si l'on veut que notre code soit généré automatiquement (c'est-à-dire ici, nos boucles déroulées). Pour ce faire nous allons utiliser la récursion (pour boucler) et la spécialisation (pour s'arrêter de boucler).

```
Exemple de boucles
template <int Begin, int End> struct Boucle
{
    static void Do()
    {
        DoSomething();
        Boucle<Begin + 1, End>::Do();
    }
};

template <int N> struct Boucle<N, N>
{
    static void Do() {}
};

// Méthode habituelle
for (int i = 5; i < 15; ++i)
    DoSomething();

// Avec notre structure Boucle
Boucle<5, 15>::Do();
```

5- N'oubliez pas les variables temporaires

Lorsque nos programmes vont commencer à grossir, il sera plus confortable de travailler avec des variables temporaires. La syntaxe template est parfois lourde à comprendre, essayons de la rendre le plus sympathique possible. De la même manière que pour stocker un résultat, ce sont nos fidèles **enum** qui vont s'acquitter de cette tâche :

```
Exemple de variables temporaires
template <int X, int Y> struct SomeVeryComplicatedComputations
{
    enum
    {
        Temp1 = X + 2 * Y,
        Temp2 = Y + 2 * X,
        Temp3 = X < Y ? 0 : 1,

        Value = Temp3 ? Temp1 : Temp2
    };
};
```

3.2. Réecriture des fonctions mathématiques

Comme déjà suggéré plus tôt dans cet article, les templates peuvent nous aider à réécrire certains calculs mathématiques de manière inhabituelle et beaucoup plus optimisée. Si vous ne savez pas comment sont implémentés les sinus, exponentielles et autres joyeusetés mathématiques je ne vous expliquerez pas ça en détail, sachez juste que ce sont généralement les développements en série entière qui sont utilisés. C'est à dire qu'on peut exprimer certaines fonctions mathématiques sous forme de polynôme de degré N ; plus N sera élevé, meilleure sera l'approximation. C'est bien souvent ce degré N qui sera notre paramètre template, ainsi nous pourrons même contrôler à volonté la précision du résultat. Une fois que vous aurez trouvé un bon compromis rapidité de compilation / précision vous pourrez fixer N une bonne fois pour toute et éviter de le trimballer dans tous vos appels de fonction.
De plus, ces polynômes approchants sont déterminés de manière itérative, donc facilement de manière récursive, ce qui les rend assez adaptés à la meta-programmation. Voici quelques exemples typiques de fonctions, une fois que vous en aurez vu quelques uns et assimilé le processus, vous pourrez aisément écrire les vôtres (pour peu que vous ayiez la formule mathématique associée bien sûr).

La factorielle

$$x! = x \times (x-1)!$$
$$0! = 1$$

```
template<unsigned int N> struct Factorielle
{
    enum {Value = N * Factorielle<N - 1>::Value};
};
template<> struct Factorielle<0>
{
    enum {Value = 1};
};
```

Attention, cette version est 100% calculée à la compilation, elle ne servira donc qu'à calculer des constantes, et ne sera donc pas utilisée à l'execution. Voici la même mais en version "fonction inline", ce qui nous permettra de renvoyer un double et donc de manipuler des nombres beaucoup plus grands (on est limité à 12! avec la version ci-dessus)

```
template <int I> inline double Factorielle()
{
    return I * Factorielle<I - 1>();
}
template <> inline double Factorielle<0>()
{
    return 1.0;
}
```

La puissance

$$x^y = x \times x^{y-1}$$
$$x^0 = 1$$

```
template <int N> inline double Puissance(double x)
{
    return x * Puissance<N - 1>(x);
}
template <> inline double Puissance<0>(double x)
{
    return 1.0;
}
```

L'exponentielle

$$e^x = \sum_{n \in N} \frac{x^n}{n!}$$

$$e^{-x} = \frac{1}{e^x}$$

```
// Ici N représente l'ordre, autrement dit la précision du calcul

template <int I> inline double Exp_(double x)
{
    return Exp_<I - 1>(x) + Puissance<I>(x) / Factorielle<I>();
}
template <> inline double Exp_<0>(double x)
{
    return 0.0;
}

template <int N> inline double Exponentielle(double x)
{
    return x < 0.0 ? 1.0 / Exp_<N>(-x) : Exp_<N>(x);
}
```

Le cosinus

$$\cos(x) = \sum_{n \in N} \frac{(-1)^n x^{2n}}{(2n)!}$$

```
// Ici aussi, N resprésente l'ordre de développement

template <int N> inline double Cosinus(double x)
{
    return Cosinus<N - 1>(x) + (N % 2 ? -1 : 1) * Puissance<2 * N>(x) / Factorielle<2 * N>();
}
template <> inline double Cosinus<0>(double x)
{
    return 1.0;
}
```

L'arctangente hyperbolique

$$arctanh(x) = \sum_{n \in N} \frac{x^{2n+1}}{2n+1}$$

```
template <int I> inline double Atanh(double x)
{
    return Atanh<I - 1>(x) + Puissance<2 * I + 1>(x) / (2 * I + 1);
}
template <> inline double Atanh<1>(double x)
{
    return x;
}
template <> inline double Atanh<0>(double x)
{
    return 0.0;
}
```

Essayez un Atanh<50>(0.5) par exemple, et regardez le code généré : ce n'est rien d'autre qu'une immense suite de multiplications de constantes !

Pour conclure cette partie, je tiens à vous signaler que ces fonctions ne sont encore pas optimisées à fond. En effet les fans de bricolage pourrons s'en donner à coeur joie, pour notamment réduire la profondeur de récursion ou éliminer les branches de récursion inutiles. De même en remaniant les formules mathématiques utilisées pour les rendre plus sympathiques pour nos templates, il y a encore là matière à optimiser.

3.3. Les "expression templates", ou comment booster vos calculs matriciels

Vous avez certainement déjà entendu parler de bibliothèques de calcul matriciel, telles que la MTL ou boost::uBlas par exemple. On vous dit qu'elles sont optimisées, et que leurs performances dépassent celles des bibliothèques classiques de calcul mathématique. Mais comment diantre font-elles ? Quel est ce procédé mystique qui leur permet de telles

performances ? Et bien toutes deux sont basées sur le principe des *expression templates*, que je vais vous détailler ici.

Imaginez ce bout de code, utilisation typique d'une classe de vecteurs mathématiques :

```
Vector V1, V2, V3, V4;

V4 = (V1 / 2.5) - (V2 + V3) * 5;
```

Mettons-nous maintenant à la place de notre compilateur, et examinons ce code.

1. **V2 + V3** sera d'abord évalué, un vecteur temporaire **Temp1** sera généré après appel à l'opérateur + surchargé.
2. **V1 / 2.5** sera ensuite évalué, un nouveau vecteur temporaire **Temp2** sera créé après appel à l'opérateur /.
3. **Temp1 * 5** sera calculé, et créera un résultat **Temp3** après appel à l'opérateur *.
4. **Temp2 - Temp3** sera le dernier calcul effectué, produisant une nouvelle variable temporaire **Temp4** après passage par l'opérateur -.
5. **V4 = Temp4** sera finalement évalué, affectant les valeurs de Temp4 à V4 via l'opérateur =.

Ce que nous pouvons donc résumer en :

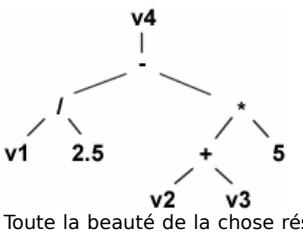
```
Vector V1, V2, V3, V4;

// operator +(const Vector&, const Vector&)
Vector Temp1;
for (int i = 0; i < 3; ++i)
    Temp1[i] = V2[i] + V3[i];

// operator /(const Vector&, double)
Vector Temp2;
for (int i = 0; i < 3; ++i)
    Temp2[i] = V1[i] / 2.5;

// operator *(const Vector&, double)
Vector Temp3;
for (int i = 0; i < 3; ++i)
    Temp3[i] = Temp1[i] * 5;

// operator -(const Vector&, const Vector&)
Vector Temp4;
for (int i = 0; i < 3; ++i)
    Temp4[i] = Temp2[i] - Temp3[i];

// Vector::operator =(const Vector&)
for (int i = 0; i < 3; ++i)
    V4[i] = Temp4[i];
```

Vous le voyez, un expression aussi simple qu'une ligne de calcul vectoriel va générer en fait de nombreux appels de fonctions, variables temporaires et calculs inutiles.

C'est là qu'interviennent les *expression templates*, qui vont nous permettre de générer directement le résultat en une seule passe :

```
Vector V1, V2, V3, V4;

// Vector::operator =(???)
for (int i = 0; i < 3; ++i)
    V4[i] = (V1[i] / 2.5) - (V2[i] + V3[i]) * 5;
```

Belle optimisation n'est-ce pas. Mais comment cela va-t-il être possible ? Il va nous falloir différer l'évaluation des calculs, c'est-à-dire ne rien faire dans nos opérateurs surchargés et laisser tout le boulot à l'opérateur =. Au lieu de calculer des résultats intermédiaires, nos surcharges d'opérateurs vont maintenant construire **l'arbre syntaxique** de notre expression, à l'aide de classes templates. L'opérateur = va ensuite parcourir cet arbre syntaxique et effectuer en un seul coup le calcul. Voici l'arbre correspondant à notre expression :



Toute la beauté de la chose réside dans les templates et l'inlining : vous pourrez penser que la construction d'un tel arbre va se révéler encore plus coûteuse que le code original, ce qui est vrai. Mais ici notre arbre ne sera pas physique, il sera construit et parcouru par le compilateur. Une fois notre programme compilé il n'en restera que la ligne de calcul donnée plus haut. Rappelez-vous, nous faisons ici de la meta-programmation !

Pour arriver à nos fins il va nous falloir quelques classes templates pleines de vide, comme nous les aimons. Bon bien sûr elles ne seront pas vraiment vides, mais après compilation il n'en restera presque (voire totalement) rien, notre programme n'y aura vu que du feu.

Tout d'abord définissons notre classe de vecteurs, **Vector** :

```
struct Vector
{
    // Définition des types iterateurs
    typedef double* iterator;
    typedef const double* const_iterator;

    // Fonctions de récupération des itérateurs de début et de fin
    iterator begin() {return Elements;}
    iterator end()   {return Elements + 3;}
    const_iterator begin() const {return Elements;}
    const_iterator end()   const {return Elements + 3;}

    // Et enfin les données
    double Elements[3];
};
```

Bien sûr cette classe ne contient que le strict nécessaire pour accompagner cet article, une vraie classe de vecteurs mathématiques sera plus complète.
Notre classe contient donc ses éléments (dimension 3 ici), et des accesseurs permettant de récupérer un itérateur sur le début des données ainsi que sur la fin, style STL. Cela va nous servir plus tard à parcourir nos éléments et effectuer les calculs.

Il nous faut maintenant nos fameuses classes template, qui vont nous permettre de construire l'arbre syntaxique. Ici chaque noeud est un opérateur, unaire ou binaire, et dont les fils peuvent a priori avoir n'importe quel type. Nous allons donc construire les classes correspondantes :

```
Nos opérateurs
enum Operators
{
    Add,   // addition
    Sub,   // soustraction
    Mult,  // multiplication
    Div,   // division
    Plus,  // + unaire
    Minus, // - unaire
    Const  // Constante
};
```

```
Quelques opérateurs unaires
// Modèle de classe pour nos opérateurs unaires
template <class T, int Op> struct UnaryOp {};

// Classe de base pour nos opérateurs unaires
template <class T> struct UnaryOpBase
{
    UnaryOpBase(T t) : It(t) {}
    inline void operator ++() {++It;}

    T It;
};

// Spécialisation pour le moins unaire - nous devrons spécialiser nos classes pour chaque opérateur
template <class T> struct UnaryOp<T, Minus> : public UnaryOpBase<T>
{
    UnaryOp(T t) : UnaryOpBase<T>(t) {}
    inline double operator *() {return -(*It);}
};

// Spécialisation pour les constantes
template <class T> struct UnaryOp<T, Const> : public UnaryOpBase<T>
{
    UnaryOp(T t) : UnaryOpBase<T>(t) {}
    inline void operator ++() {}
    inline double operator *() {return It;}
};
```

```
Quelques opérateurs binaires
// Modèle de classe pour nos opérateurs binaires
template <class T, class U, int Op> struct BinaryOp {};

// Classe de base pour nos opérateurs binaires
template <class T, class U> struct BinaryOpBase
```

```
{
    BinaryOpBase(T I1, U I2) : It1(I1), It2(I2) {}
    inline void operator ++() {++It1; ++It2;}

    T It1;
    U It2;
};

// Spécialisation pour l'addition
template <class T, class U> struct BinaryOp<T, U, Add> : public BinaryOpBase<T, U>
{
    BinaryOp(T I1, U I2) : BinaryOpBase<T, U>(I1, I2) {}
    inline double operator *() {return *It1 + *It2;}
};

// Spécialisation pour la multiplication
template <class T, class U> struct BinaryOp<T, U, Mult> : public BinaryOpBase<T, U>
{
    BinaryOp(T I1, U I2) : BinaryOpBase<T, U>(I1, I2) {}
    inline double operator *() {return *It1 * *It2;}
};
```

Ce code a priori pas évident est en fait assez simple : nous créons pour chaque opérateur une spécialisation de la classe correspondante. Tout ce que nous avons à spécilaliser est l'opérateur *, qui va effectuer le calcul effectif (l'addition des deux fils pour Add, l'inversion du fils pour Minus, ...). Les noeuds de l'arbre sont représentés dans nos classes par des itérateurs, qui seront incrémentés à chaque boucle (dans notre opérateur =), et qui renverront le résultat de l'opérateur associé via l'opérateur *. Si les fils sont eux-même des BinaryOp ou UnaryOp, l'appel à l'opérateur * va appeler l'opérateur * des fils, récursivement jusqu'à tomber sur une feuille de l'arbre, dont les itérateur pointeront cette fois vers les "vraies" valeurs de notre calcul (les éléments des vecteurs, ou les valeurs des constantes).
Si tout cela n'est pas encore très clair dans votre esprit ne vous inquietez pas, lisez la suite et essayez d'implémenter ces classes par vous-même sur des exemples simples, vous saisirez vite comment ça marche. Le plus important est de ne pas se laisser impressionner par la syntaxe rebutante, bien que légère, de tous ces templates.

Bien, maintenant que nous avons défini le type des noeuds de notre arbre, voyons à quoi va ressembler le type de celui-ci avec notre expression d'exemple :

```
BinaryOp<
    BinaryOp<
        Vector::const_iterator,
        UnaryOp<
            double,
            Const>,
        Div>,
    BinaryOp<
        BinaryOp<
            Vector::const_iterator,
            Vector::const_iterator,
            Add>,
        UnaryOp<
            double,
            Const>,
        Mult>,
    Minus>
```

Heureusement tout cela c'est pour notre compilateur, nous ne verrons pas tous ces types à rallonge (sauf lorsqu'on aura affaire à une erreur de compilation. Si vous devez être attentif à ce que vous codez c'est donc bien ici, une malheureuse faute de frappe peut vous mener à un message d'erreur tellement long que vous mettrez quelques heures à vous rendre compte que c'est juste une faute de frappe !).

Avant d'attaquer la surcharge de nos opérateurs pour construire notre arbre, munissons-nous de quelques fonctions qui vont nous simplifier leur écriture, notamment grace à la détection automatique de templates. Ce genre de fonction est très utilisé par exemple dans la STL, pour éviter d'écrire tous les paramètres templates d'une classe inutilement.

```
template <class T, class U> inline BinaryOp<T, U, Add> MakeAdd(const T& t, const U& u)
{
    return BinaryOp<T, U, Add>(t, u);
}
template <class T, class U> inline BinaryOp<T, U, Mult> MakeMult(const T& t, const U& u)
{
    return BinaryOp<T, U, Mult>(t, u);
}
template <class T> inline UnaryOp<T, Minus> MakeMinus(const T& t)
{
    return UnaryOp<T, Minus>(t);
}
template <class T> inline UnaryOp<T, Const> MakeConst(const T& t)
{
    return UnaryOp<T, Const>(t);
```

```
}
```

Bien, passons maintenant à la **surcharge de nos opérateurs**, pour construire l'arbre en question :

```
Surcharges de l'opérateur +
template <class T, class U> BinaryOp<T, U, Add> operator +(const T& v1, const U& v2)
{
    return MakeAdd(v1, v2);
}
template <class T> BinaryOp<Vector::const_iterator, T, Add> operator +(const Vector& v1, const T& v2)
{
    return MakeAdd(v1.begin(), v2);
}
template <class T> BinaryOp<T, Vector::const_iterator, Add> operator +(const T& v1, const Vector& v2)
{
    return MakeAdd(v1, v2.begin());
}
BinaryOp<Vector::const_iterator, Vector::const_iterator, Add> operator +(const Vector& v1, const Vector& v2)
{
    return MakeAdd(v1.begin(), v2.begin());
}

Surcharges de l'opérateur *
template <class T>BinaryOp<T, UnaryOp<double, Const>, Mult> operator *(const T& v1, double d)
{
    return MakeMult(v1, MakeConst(d));
}
BinaryOp<Vector::const_iterator, UnaryOp<double, Const>, Mult> operator *(const Vector& v1, double d)
{
    return MakeMult(v1.begin(), MakeConst(d));
}

Surcharges de l'opérateur -
template <class T> UnaryOp<T, Minus> operator -(const T& v1)
{
    return MakeMinus(v1);
}
UnaryOp<Vector::const_iterator, Minus> operator -(const Vector& v1)
{
    return MakeMinus(v1.begin());
}
```

Ici chaque opérateur construit juste le noeud de l'arbre correspondant, aucun calcul n'est effectué. Nous devons écrire des surcharges différentes lorsque nous rencontrons un Vector, car dans ce cas la syntaxe sera différente (on va stocker non pas un T mais un Vector::const_iterator, qui sera récupéré grâce à v.begin()). On pourrait imaginer bidouiller nos templates pour donner une syntaxe similaire aux Vector, et ainsi n'écrire qu'une version de nos opérateurs. Mes quelques essais ne se sont pas révélés concluants, mais libre à vous de vous amuser avec vos templates pour rendre la syntaxe la plus concise et la plus claire possible.
Pour ne pas encombrer cet article avec trop de code je n'ai mis en exemple que quelques opérateurs, mais il faudrait bien sûr tous les coder. Le code sera plus ou moins identique, et une fois qu'on a saisi le principe cela ne demande presque aucun effort supplémentaire.

Bien, nous avons donc maintenant un arbre syntaxique pour notre expression, et nous avons tout ce qu'il faut pour effectuer le calcul grace à nos opérateurs *. Il ne nous reste plus qu'à le faire, et c'est cette fois le boulot de **l'opérateur =** de notre classe Vector :

```
Surcharge de l'opérateur d'affectation
template <class T> const Vector& Vector::operator =(T Expr)
{
    for (iterator i = begin(); i != end(); ++i, ++Expr)
        *i = *Expr;

    return *this;
}
```

Une fois ce code inliné et compilé, tout ce qu'il en restera sera la ligne correspondante à notre calcul mathématique déjà citée plus haut. Mission accomplie !

Cette section est assez longue et pleine de code template, prenez bien le temps de vous familiariser avec et vous verrez que ce concept puissant n'est pas si dur que cela à appréhender. Je vous ai parlé ici du calcul matriciel, mais on peut imaginer bien d'autres applications aux *expression templates* : calcul d'intégrales, et en général toute sorte de calculs mathématiques, ou bien encore les opérations sur les chaînes de caractères comme la concaténation. Vous pourrez ainsi imaginer toute sorte d'optimisation auxquelles vous n'auriez pas penser, avec cette nouvelle technique.

4. Génération automatique de code

Nous avons vu jusqu'à présent comment la meta-programmation pouvait nous aider à optimiser nos calculs, la plupart du temps mathématiques. Mais ce n'est que l'une des nombreuses utilisations de ce type de programmation, nous allons pouvoir effectuer des choses de plus en plus étonnantes et pratiques, à commencer par la génération automatique de code. C'est un domaine vaste, et dont la seule limite sera votre imagination. Cette partie sera donc tout sauf exhaustive, je vous donnerai quelques éléments de base et quelques exemples qui je l'espère vous aiderons à développer vos propres chefs d'oeuvre de meta-programmation.

4.1. Les typelists

Les *typelists* (listes de types) sont à la base de beaucoup de concepts de meta-programmation, notamment pour la génération de code. Ils se présentent sous une forme tellement simple que je vais vous la donner sans plus attendre :

```
Implémentation des typelists
template <class T1, class T2>
struct TypeList
{
    typedef T1 Head;
    typedef T2 Tail;
};
```

That's all. Comme vous le voyez une typelist n'est rien d'autre qu'une structure template définissant une tête (Head) et une queue (Tail). Ce ne sont même pas des valeurs, juste des types. La construction de nos liste sera récursive, à savoir qu'une liste sera de la forme **TypeList<Type1, TypeList<Type2, TypeList<... TypeList<TypeN, NullType> > > >**. NullType est une structure spéciale qui définira la fin de notre liste.

```
struct NullType {};
```

Ainsi, si nous voulons construire la liste [double, float, int] par exemple, il faudra écrire :

```
typedef TypeList<double, TypeList<float, TypeList<int, NullType> > > MyList;
```

Vous le voyez, la construction d'une typelist peut très vite devenir laborieuse et illisible pour les autres. On peut imaginer plusieurs méthodes pour simplifier ce procédé, en voici une à base de macros :

```
#define TYPELIST_1(t1)                          TypeList<t1, NullType>
#define TYPELIST_2(t1, t2)                      TypeList<t1, TYPELIST_1(t2) >
#define TYPELIST_3(t1, t2, t3)                  TypeList<t1, TYPELIST_2(t2, t3) >
#define TYPELIST_4(t1, t2, t3, t4)              TypeList<t1, TYPELIST_3(t2, t3, t4) >
#define TYPELIST_5(t1, t2, t3, t4, t5)          TypeList<t1, TYPELIST_4(t2, t3, t4, t5) >
#define TYPELIST_6(t1, t2, t3, t4, t5, t6)      TypeList<t1, TYPELIST_5(t2, t3, t4, t5, t6) >
#define TYPELIST_7(t1, t2, t3, t4, t5, t6, t7) TypeList<t1, TYPELIST_6(t2, t3, t4, t5, t6, t7) >
...

// Notre liste devient alors
typedef TYPELIST_3(double, float, int) MyList;
```

Comme tout type de liste, nous allons maintenant pouvoir effectuer des opérations de base sur nos typelists, comme la recherche, la concaténation, l'inversion, etc... Tout à base de template, comme d'habitude. Je ne donne pas ici le code correspondant, car nous n'aurons pas besoin de ces opérations pour la suite. Le détail de toutes ces méthodes se trouve par exemple dans l'excellent livre *Modern C++ Design* d'Andrei Alexandrescu, chez Addisson-Wesley.

Nous avons maintenant des listes de types, quelques opérations de base sur ces listes si l'on veut, mais que va t-on faire de tout cela ? Et bien n'oubliez pas que nous étudions la meta-programmation, tout ce que nous allons réaliser sera du travail pour le compilateur, et ce qui concerne l'execution nous importera peu ici. En l'occurence nos typelists nous seront utiles en tant que types, jamais nous n'en instancierons pour leur donner des valeurs (ce qui serait d'ailleurs impossible car elles ne comportent aucune donnée).

Je ne vais pas vous expliquer les milles et une applications des typelists ici, mais vous pourrez trouver des implémentations de *designs patterns* par exemple, dans cet article d'Andrei Alexandrescu sur les typelists, ou encore dans son livre cité ci-dessus.

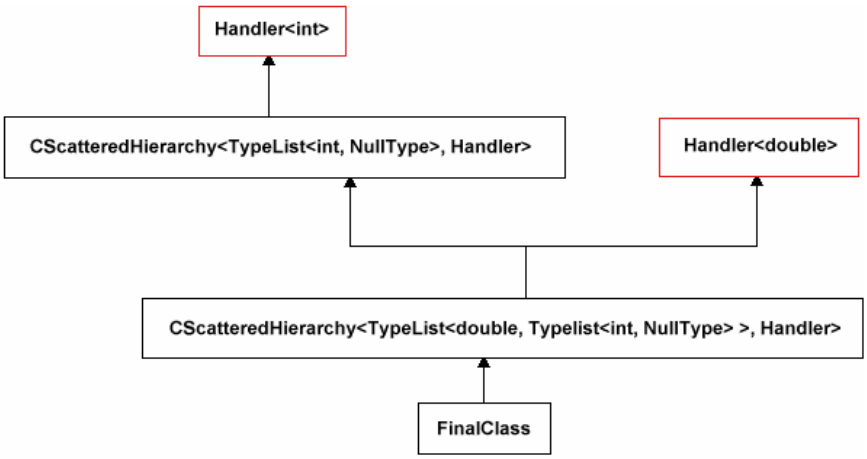
4.2. Les hiérarchies automatiques

Ici nous allons nous intéresser à une application très particulière (mais souvent utilisées) des typelists : la génération de classe via la génération automatique de hiérarchies. Imaginez une classe contenant les mêmes fonctions ou membres, mais en plusieurs exemplaires, un par type. On peut trouver ce genre de situation lorsque l'on code le *design pattern* du visiteur par exemple où une fonction Visit devra être écrite pour chaque type à visiter. Le but sera donc de n'écrire ce code redondant qu'une et une seule fois, puis de le "dupliquer" à l'aide de nos typelists et de notre génération de hiérarchie. Voici le principe : nous allons créer une classe générique (template) qui va contenir notre code redondant appliqué au type T template, puis nous allons instancier cette classe générique pour chaque type concerné, et enfin faire dériver notre

classe finale de toutes ces instanciations. Ainsi notre classe contiendra bien une version du code pour chaque type. Pour que le procédé d'héritage multiple soit lui aussi automatique, il va nous falloir l'outil approprié à savoir, ô étonnement, une classe template, que voici :

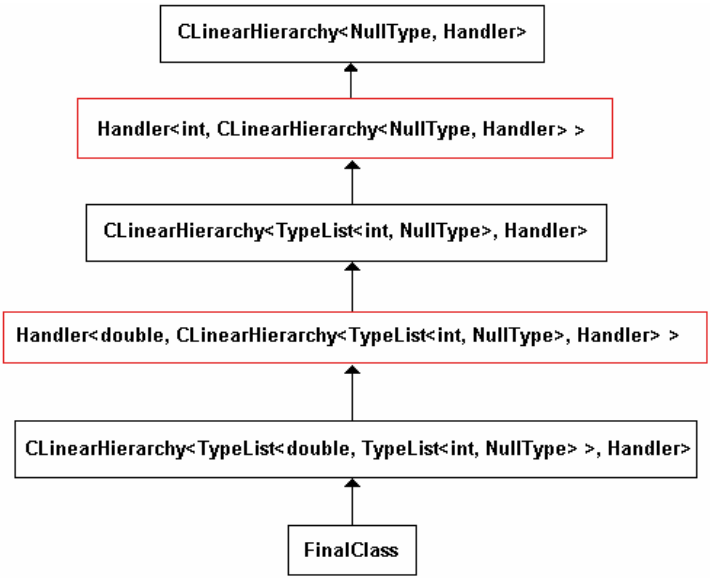
```
template <class TList, template <class> class Handler> class CScatteredHierarchy;

// Notre liste n'est pas vide : on hérite de Handler<T1> et on continue
// le processus par récurrence, avec T2 qui est la suite de notre liste.
template
<
    class T1,
    class T2,
    template <class> class Handler
>
class CScatteredHierarchy<TypeList<T1, T2>, Handler> : public Handler<T1>,
                                                       public CScatteredHierarchy<T2, Handler>
{
};

// Cas d'arrêt : on a épuisé tous les types de notre liste
template
<
    class T,
    template <class> class Handler
>
class CScatteredHierarchy<TypeList<T, NullType>, Handler> : public Handler<T>
{
};
```

Le paramètre template **Handler** sera la classe générique qui contient le code, **TList** va lui être notre typelist contenant les types pour lesquels instancier notre classe générique.
Notre classe finale n'aura ensuite plus qu'à dériver de **CScatteredHierarchy<MyTypeList, MyHandler>**
Le fonctionnement de CScatteredHierarchy peut vous paraître ambigü, voici un schéma détaillant les multiples héritages qui vont être effectués avec par exemple une typelist [double, int] :



Comme vous le voyez notre classe FinalClass va au final bien hériter de **Handler<double>** et **Handler<int>**. Le reste étant vide, cela ne perturbera pas le comportement de notre classe.

Il existe une variante à CScatteredHierarchy, **CLinearHierarchy** qui génère de manière équivalente notre classe, mais sans héritage multiple. La seule contrainte supplémentaire est de rajouter un paramètre template à notre classe Handler, et de la faire dériver de cette classe.
Je ne la détaillerai pas ici, voici seulement le schéma correspondant, pour le même exemple :



4.3. Quelques exemples concrets

Tout ceci doit vous paraître horriblement abstrait, et vous vous demandez toujours comment ce code bizarroïde va vous aider à booster le codage de vos projets. Pour conclure cette partie, voici donc quelques exemples concrets d'utilisation de tous ces nouveaux mécanismes. Le premier est un gestionnaire de ressources, dont vous trouverez le détail dans un autre article concernant mon moteur 3D, "La gestion des ressources" . A la fin de l'article sur les typelists cité plus haut, vous trouverez également des exemples d'utilisation avec le *design pattern* du visiteur, et si vous possédez le livre du même auteur (*Modern C++ Design*) vous trouverez bien d'autres applications.

5. Conclusion

C'en est donc fini de ce tour d'horizon de la meta-programmation. J'ai essayé de couvrir au maximum les possibilités de ce concept, en donnant les bases nécessaires. J'espère également que tout cela aura éveillé en vous un nouvel intêret, surtout si ce genre de programmation vous était inconnu. C'est vraiment un côté peu connu du C++ mais pourtant si puissant, et qui ne demande qu'à être exploité. Et ne vous y trompez pas, tous les mécanismes présentés dans cet article ne sont pas un genre de "gadgets" réservés aux fanatiques du C++, bien maitrisés ils peuvent vraiment se révéler utiles et même surpasser n'importe quelle autre approche plus classique. La meta-programmation est un sujet peu connu mais pourtant bien présent dans des projets ou des outils utilisés couramment, à commencer par la plus célèbre des bibliothèques de C++ : boost.

Une version PDF de cet article est disponible : Télécharger (144 Ko)

Si vous avez des suggestions, remarques, critiques, si vous avez remarqué une erreur, ou bien si vous souhaitez des informations complémentaires, n'hésitez pas à me contacter !
