Variadic templates

X.Juvigny

November 10, 2017

Contents

1 Introduction

Parmi le lot de nouveautés que le C++ a apporté, les variadic templates permettent au C++ d'accéder à une abstraction supérieure. Ils permettent 0 une meilleure généricité, des optimisations impossibles avant, et une plus grande souplesse dans la conception des interfaces, permettant de proposer des interfaces plus simples et plus naturelles à l'utilisateur de nos modules.

2 Paramètres variadiques

La conception de paramètres variadiques existent depuis l'origine du langage C. Une fonction de la librairie standard du C qui utilise cette notion est par exemple la fonction `printf` qui prend un nombre variable d'arguments dont les types sont quelconques.

Le principe est le suivant : on passe en premier les paramètres fixes de la fonction (ceux dont on connaît d'avance le type), puis les paramètres variables sont passés uniquement à l'aide de trois petits points (…). Des fonctions fournies par le C permettent ensuite à l'exécution :

1. de pouvoir parcourir les différents paramètres passés en tant que paramètres variadiques,

2. de pouvoir lire la valeurs de chaque paramètre en indiquant le type de la valeur attendue.

Toute cette machinerie se fait à l'aide de cinq macros définis dans la librairie standard `stdarg` :

- `va_list lst_params` : c'est une structure qui permet de parcourir les différents paramètres passés en arguments variadiques;
- `va_start(lst_params,last_arg)` : permet d'initialiser la structure ci–dessus sur le premier argument de la fonction variadique. Il prend en argument le dernier paramètre non variadique de la fonction (une fonction avec paramètre variadique doit avoir au moins un paramètre non variadique et les paramètres variadiques sont obligatoirement déclarés en derniers paramètres.);
- `val = va_arg(lst_params, type )` : Permet de lire la valeur courante du paramètre pointé par la liste des paramètres variadiques en précisant le type attendu;
- `va_end(lst_params)` : A appeler à la fin du parcours des valeurs de la liste des paramètres variadiques, empêchant tout appel ultérieur de `va_arg`;
- `va_copy(lst_params,lst_copy_params)` : Parfois, il est nécessaire de pouvoir revenir en arrière pour relire la valeur d'un argument passé en variadique. `va_copy` permet de copier l'état de la liste des paramètres variadique à un instant donné. Cette fonction n'est définie que dans la norme C Ansi 99, et ne sera pas valable pour des versions antérieures du C.

Voici un exemple d'utilisation des paramètres variadiques en C :

```
# include <stdarg.h>
# include <iostream>
# include <list>
# include <string>

void add_to_dico( std::list<std::string>& dico, ... )
{
    va_list params;
    va_start(params,dico);
    char* s;
    s = va_arg( params, char*);
    while ( s != "\0" ) {
        dico.push_back(std::string(s));
        s = va_arg(params, char*);
    }
    va_end(params);
}

int main()
{
    std::list<std::string> dico;
    add_to_dico(dico, "Tintin", "Spirou", "Milou", "\0" );
    for ( auto s : dico ) std::cout << s << " ";
    std::cout << std::endl;
    return EXIT_SUCCESS;
}
```

Cette technique pose plusieurs inconvénients :

- L'analyse des paramètres passés à la fonction se fait à chaque appel de la fonction, alors que ces paramètres sont connus à la compilation (ce sont les paramètres passés à la fonction) ce qui empêche toute possibilité d'optimisation de la part du compilateur;
- La gestion de ces paramètres est fait par le concepteur de la fonction, et peut générer des bogues très difficiles à détecter (il n'y a pas de contrôle par le compilateur des types des paramètres passés à la fonction);
- Pour connaître quand arrêter de parcourir les paramètres variadiques, il n'y a pas de convention imposée par le langage et c'est au concepteur de la fonction de déterminer cette convention (dans l'exemple ci–dessus, la convention est de terminer les paramètres variadiques par une chaîne de caractère avec le caractère nul;
- Enfin, cette technique ne s'applique qu'aux fonctions, et ne permet pas, entre autre, comme on le verra ici, de pouvoir définir des structures contenant un nombre quelconque d'attributs !

Depuis la norme C99, il existe des paramètres variadiques pour les macros. Si au niveau des macros, cette prise en compte des paramètres est bien faite à la compilation, l'appel au final à une fonction acceptant des arguments variadique reste nécessaire. Cela simplifie simplement la gestion des paramètres variadiques au niveau de la macro. De plus, les macros à paramètres variadiques ne permettent toujours pas d'avoir des structures à nombre de champs variables.

L'utilisation des variadiques dans une macro se fait simplement : on définit le nombre variable d'arguments à l'aide de l'ellipse ... et l'**expension** de ces variables à l'aide de la macro `__VA_ARGS__`, comme dans l'exemple donné ci–dessous.

```
#define error_log(format,...) fprintf(stderr,format,__VA_ARGS__)

int main()
{
    error_log("Une erreur bien fictive : %d\n",0);
    ...
}
```

Avant la norme 2011 du C++, il était possible avec certaines limites de définir des templates avec un nombre variable de paramètres. Pour les fonctions, il était nécessaire de définir autant de surcharge template de la fonction que du nombre maximal de paramètres qu'on imposait à l'utilisateur.

```
template<typename Container>
void fill_inplace( Container& c )
{
    using K=typename std::remove_reference<decltype(*std::begin(c))>::type;
    for ( typename Container::iterator it = std::begin(c); it != std::end(c); ++it ) {
        (*it) = K();
    }
}

template<typename Container, typename T0 >
void fill_inplace( Container& c, const T0& t0 )
{
    using K=typename std::remove_reference<decltype(*std::begin(c))>::type;
    for ( typename Container::iterator it = std::begin(c); it != std::end(c); ++it ) {
        (*it) = K(t0);
    }
}

template<typename Container, typename T0, typename T1 >
```

```
void fill_inplace( Container& c, const T0& t0, const T1& t1 )
{
    using K= typename std::remove_reference<decltype(*std::begin(c))>::type;
    for (typename Container::iterator it = std::begin(c); it != std::end(c); ++it ) {
        (*it) = K{t0,t1};
    }
}

int main()
{
    std::vector<double> u(5);
    fill_inplace(u,3.14);
    std::vector<std::complex<double>> z(5);
    fill_inplace(z,3.14,1.414);
    ...
```

Ainsi, dans le wrapper C++ d'OpenCL, pour l'appel à un noyau OpenCL, il a été défini 32 versions de la fonction permettant d'appeler un noyau en passant directement les arguments demandés par le noyau, pour les trente deux cas possibles : appel d'un noyau sans paramètre, avec un paramètre, avec deux, ..., avec trente et un paramètres ! Si par malheur notre noyau attend trente trois paramètres, il faut faire alors appel aux fonctions C d'OpenCL pour rentrer un par un les trente trois paramètres...

En revanche, cette fois ci, les templates permettent :

1. de s'assurer que le type attendu par la fonction est en adéquation avec le type passé (on peut contrôler les types attendus par la fonction avec les templates, voir plus haut...);

2. de résoudre les nombre et le types des paramètres passés à la compilation, et non plus à l'exécution.

Pour les structures à nombre d'attributs variables, une autre méthode permet de pouvoir les définir, mais toujours avec un nombre maximal de paramètre template en C++98. L'astuce consiste à définir une structure vide, puis passer un nombre conséquent de paramètres templates qui ont pour type par défaut cette structure vide.

```
struct void_type {
    void_type() {}
};

/** Un tuple contient au moins un élément dans cette version
*/
template<typename T0            , typename T1=void_type, typename T2=void_type, typename T3=void_type,
         typename T4 = void_type, typename T5=void_type, typename T6=void_type, typename T7=void_type>
struct tuple_t
{
    T0 t0; T1 t1; T2 t2; T3 t3; T4 t4; T5 t5; T6 t6; T7 t7;
    tuple_t(const T0& v0, const T1& v1=void_type(), const T2& v2=void_type(), const T3& v3=void_type(),
            const T4& v4=void_type(), const T5& v5=void_type(), const T6& v6=void_type(), const T7& v7=void_type()) :
            t0(v0),t1(v1),t2(v2),t3(v3),t4(v4),t5(v5),t6(v6),t7(v7)
    {}
};

int main()
{
    tuple_t<int,double,std::string> tple(3,3.14,"pi");
    std::cout << tple.t0 << ", " << tple.t1 << ", " << tple.t2 << std::endl;
    ...
}
```

Si le C++98 permet de résoudre les problèmes posés par le C dans le cas de paramètres variadiques, mais on est obligé de définir un nombre maximal de paramètres admis et cette technique entraine une lourdeur de programmation non négligeable.

La norme 2011 du C++ a introduit la notion de template variadique, c'est à dire avec un nombre de paramètres templates variables. Nous allons voir que cela permet de résoudre certains des problèmes qui se posent encore avec le C++ 98.

3 Principe des templates variadiques

Comme pour le C, le symbole ... défini les paramètres variadiques pour les templates. Par contre, l'ensemble des paramètres symbolisés par ... peut être nommé par le programmeur :

```
template<typename ... Args> void f(Args... args)
{
```

Dans le listing au-dessus, `Args` est le **paquet** de paramètres templates et `args` le **paquet** d'arguments passés à la fonction (dont le type de chaque argument correspondra un à un aux paramètres templates qui seront passés.

Pour déployer les arguments passés, on utilisera les `...` après le nom du paquet de paramètre. Par exemple :

```
# include <iostream>

/** paramètres templates variadiques                    Arguments variadiques
template<typename ...Args> void print_err(const char* fmt, Args ...args)
{
   fprintf(stderr, fmt, args...);// Déployement des arguments variadiques
}

int main()
{
    // Ci dessous, Args prendra comme types [int, double] et args [404,4.04]
    print_err("Ceci n'est pas une erreur %d ni %g\n",404,4.04);
    return 0;
}
```

Lors de l'invocation de `args...`, le compilateur C++ remplace `args...` par la liste des valeurs des arguments passés à la fonction séparés par des virgules.

En fait, les trois points peuvent ne pas suivre immédiatement `args`. Dans ce cas, les `...` remplacent le pattern contenant args par une liste de ce pattern répliqué plusieurs fois dont chaque membre est séparé par une virgule (du moins jusqu'au C++ 2017 qui permet de choisir le symbole de séparation). Par exemple :

```
# include <iostream>
# include <string>

template<typename Head, typename ...Args> std::ostream& print(std::ostream& out, Head const & head, Args const & ...args)
{
    out << head;
    // On utilise les propriétés de l'opérateur , pour construire
    // une liste d'initialisation d'entiers nuls tout en affichant nos arguments :
    auto tmp = {(out << ", " << args,0)...};
    return out;
}

int main()
{
    print_err("Ceci n'est pas une erreur",404,"ni",4.04,"\n");
    return 0;
}
```

Quelques remarques sur le listing ci–dessus :

- Le paquet de paramètre `args` est passé en référence constante : `Args const & ...` pour éviter des copies;

- Il n'était pas possible d'écrire directement `std::cerr << args...;` car dans ce cas, `args` aurait été remplacé par `arg1,arg2,...,argn` ce qui n'aurait pas été syntaxiquement correct avec les entrées sorties de la bibliothèque `iostream`;
- Plus subtile, il était également impossible d'écrire le code suivant :

```
template<typename ...Args> void print_err(Args const & ...args)
{
    auto tmp = {(0, std::cerr << args)...};
}
```

  car cela aurait engendré une copie de `std::cerr` ce qui est interdit par la bibliothèque standard.

- On verra à la fin du document qu'en C++ 2017, la mise en œuvre de la fonction peut être simplifiée :

```
template<typename ...Args> auto print_err(Args const & ...args)
{
    std::cerr << args << ... << " ";
}
```

- Par rapport aux versions précédentes en C ou en C++ 98, plusieurs avantages se dégagent :
  - Plus besoin d'une chaîne de formatage comme dans la première version, ce qui évite pas mal d'erreurs de programmation;
  - Le nombre d'arguments passés à la fonction est vraiment quelconque contrairement à une version écrite en C++ 98;
  - La lourdeur de programmation s'en trouve largement allégée (pas plusieurs versions de la même fonction).

Si on reprend l'exemple de la fonction qui rajoute des mots à un dictionnaire (une liste ici) qu'on avait écrit à l'aide des variadiques du langage C, on peut écrire une version simplifiée et moins sujette aux erreurs de programmation :

```
# include <stdarg.h>
# include <iostream>
# include <list>
# include <string>

template<typename ...Args>
void add_to_dico( std::list<std::string>& dico, Args const &... args)
{
    auto l = std::list<std::string>{args...};
    dico.insert(dico.end(),l.begin(),l.end());
}

int main()
{
    std::list<std::string> dico;
    add_to_dico(dico, "Tintin", "Spirou", "Milou" );
    for ( auto s : dico ) std::cout << s << " ";
    std::cout << std::endl;
    return EXIT_SUCCESS;
}
```

Remarque : La première ligne de la fonction `add_to_dico` initialise une liste de chaîne de caractères en passant les arguments variadiques dans une liste d'initialisation (symbolisée par `{ }`. Cela suppose implicitement que tous les

arguments variadiques passés à la fonction doivent être ici des chaînes de caractère. Si cela n'est pas vérifié, le compilateur générera une erreur de compilation (plus ou moins compréhensible selon les compilateurs).

Il est tout à fait possible de développer les arguments variadiques combinée avec une expression, du moment que les opérateurs **,** séparant les diverses expressions permettent une syntaxe cohérente.

```
# include <tuple>
# include <iostream>

// Effectue le produit par eux-mêmes de chaque arguments passés à la fonction
template<typename ...Args> auto squared( Args const & ... args)
{
    return std::make_tuple(args*args...);
}

int main()
{
    int n, x, i, l;
    std::tie(n,x,i,l) = squared(3,1.2, 4u, 3L);
    std::cout << n << " " << x << " " << i << " " << l << std :: endl;
    return 0;
}
```

Il est également permis d'utiliser plusieurs fois le déployement des arguments variadiques. Par exemple, le programme suivant effectue le produit cartésien d'ensembles et affiche ensuite le tuple résultat :

```
# include <iostream>
# include <tuple>

/** Construit un tuple de paires d'éléments, chaque élément étant constitué de t et d'une variable
    de l'ensemble des variables variadique contenues dans args
*/
template<typename T, typename... Args>
auto pairWithRest( T const& t, Args const&... args ) {
    return std::make_tuple(std::make_pair(t,args)...);
}

/** Construit un tuple contenant le produit cartésien des éléments passés en arguments
*/
template<typename... Args> auto
selfCartesianProduct( Args const&... args ) {
    // std::tuple_cat concataine des tuples...
    return std::tuple_cat(pairWithRest(args,args...)...);
}

/** Affiche un tuple à l'écran */
template<typename Tuple, std::size_t N>
struct TuplePrinter {
    static void print(const Tuple& t ) {
        TuplePrinter<Tuple, N-1>::print(t);
        std::cout << ", " << std::get<N-1>(t);
    }
};
// Spécialisation quand N=1
template<typename Tuple>
struct TuplePrinter<Tuple,1>{
    static void print( const Tuple& t ) {
        std::cout << std::get<0>(t);
    }
};

/** Affiche un tuple contenant un nombre quelconque de valeurs de types indéterminés */
template<typename... Args> void
print(const std::tuple<Args...>& t ) {
    std::cout << "(";
    TuplePrinter<decltype(t),sizeof...(Args)>::print(t);
    std::cout << ")";
}


/** Affiche une paire d'éléments à l'écran */
template<typename S, typename T> std::ostream&
operator << (std::ostream& out, std::pair<S,T> const & p )
{
    out << "{" << p.first << "," << p.second << "}";
    return out;
}

int main()
{
```

```
    auto cp = selfCartesianProduct(1,"2",3u,"quatre",5.0);
    print(cp);
    return EXIT_SUCCESS;
}
```

La fonction `selfCartesianProduct` utilise bien deux fois le déploiement des arguments variadiques pour effectuer le produit cartésien. Une seconde remarque sur ce code : la fonction `print` utilise un ensemble de types variadiques au sein des arguments templates de la classe `tuple` : c'est justement le sujet de la prochaine section : pouvoir définir des structures pouvant contenir un nombre variable d'attributs qui seront définis par les paramètres variadiques template.

4 Classes et structures à paramètres variadiques

Imaginons que nous voulons pouvoir à la compilation calculer la valeur d'une variable qui doit être la somme de n entiers, n pouvant varier selon les cas.

Une solution serait de concevoir une fonction constexpr qui prend en argument une liste d'initialisation et qui en fait la somme :

```
# include <iostream>

constexpr long f_adder( const std::initializer_list<long>& l )
{
    long sum = 0;
    for ( auto it = l.begin(); it != l.end(); ++it )
        sum += *it;
    return sum;
}

int main()
{
    constexpr long sum = f_adder({1,2,3,4});
    std::cout << "1+2+3+4=" << sum << std::endl;
    return 0;
}
```

Ce programme répond bien à nos attentes, mais il a cependant quelques petits défaut :

- On a du écrire une fonction uniquement pour calculer des sommes d'entier à la compilation : à chaque besoin d'une somme d'entier à résoudre à la compilation, on est obligé de créer une variable constexpr;
- Il est facile d'oublier le `constexpr`, et dans ce cas, on se retrouve avec une fonction visible par le linker et qui va évaluer à l'exécution la somme des entiers pour la variable dont on a oublié le constexpr.

On va donc créer directement des variables templates à l'aide du C++ 14, qui seront évaluées à la compilation sans faire appel à une seule fonction constexpr. Ces variables seront définies à l'aide d'un paramètre variadique template sous forme de long :

```
# include <iostream>

template<long n0, long... n> constexpr long adder = n0 + adder<n...>;

template<long n0> constexpr long adder<n0> = n0;

int main()
{
    std::cout << "1+2+3+4=" << adder<1,2,3,4> << std::endl;
    return 0;
}
```

Notons que nous avons été obligé de spécialiser `adder` pour prendre en compte le cas où une seule variable template est passée et arrêter dans ce cas la récursion. Cette fois ci, nous avons bien une variable définie à la compilation et dont on est assuré qu'elle ne sera pas évaluée à l'exécution. La constexpr nous permet également de nous assurer de ne pas créer de variables, en particuliers des variables intermédiaires dûes à la définition récursive (si le constexpr n'avait pas été déclaré, le compilateur aurait créé les variables `adder<1,2,3,4>, adder<2,3,4>, adder<3,4>` et `adder<4>`. Cet exemple montre que les paramètres variadiques permettent de jouer sur les types, mais ne montre pas comment définir une classe template contenant un nombre quelconque d'attributs. C'est ce que montrera l'exemple suivant.

Dans cet exemple, nous allons définir un type de tas (FILO) contenant des données de type hétérogène :

```
# include <iostream>
# include <string>

template<typename T1, typename ...Tn>
class hheap
{
public:
    using value_type=T1;
    hheap(const T1& t1, const hheap<Tn...>& heap) : value(t1), m_subheap(heap)
    {}
    hheap(const hheap&) = default;
    template<typename T0> auto
    push( const T0& t0 ) const {
        return hheap<T0,T1,Tn...>(t0,*this);
    }
    const auto& pop() const {
        return m_subheap;
    }
    T1 get_value() const { return value; }
    bool is_empty() const { return false; }
private:
    T1 value;
    hheap<Tn...> m_subheap;
};

struct void_t {};

template<> class hheap<void_t>
{
public:
    hheap() {}
    hheap(const hheap&) = default;
    template<typename T0> auto
    push( const T0& t0 ) const {
        return hheap<T0>(t0);
    }
    bool is_empty() const { return true; }
};

template<typename T1> class hheap<T1>
{
public:
    using value_type=T1;
    hheap(const T1& t1) : value(t1)
    {}
    hheap(const hheap&) = default;
    template<typename T0> auto
    push( const T0& t0 ) const {
        return hheap<T0,T1>(t0,*this);
    }
    const auto pop() const {
        return hheap<void_t>{};
    }
    T1 get_value() const { return value; }
private:
    T1 value;
};

template<typename Heap, typename T0, typename ...Args> auto make_heap_helper( Heap& heap, const T0& t0, Args const & ... args )
{
    auto heap_next = heap.push(t0);
    return make_heap_helper(heap_next, args...);
}
template<typename Heap, typename T0> auto make_heap_helper( Heap& heap, const T0& t0 )
```

```
{
    auto heap_next = heap.push(t0);
    return heap_next;
}

template<typename T0, typename ...Args> auto make_heap(const T0& t0, Args const&... args )
{
    hheap<void_t> a;
    return make_heap_helper(a,t0,args...);
}

template<typename Heap> std::ostream& print(std::ostream& out, const Heap& heap)
{
    out << heap.get_value() << ",";
    return print(out, heap.pop());
}
template<typename T>
std::ostream& print(std::ostream& out, const hheap<T>& heap)
{
    out << heap.get_value();
    return out;
}


int main()
{
    auto a = make_heap(std::string("toto"), 3, 4.15);
    auto b = hheap<void_t>{}.push(3.14).push(std::string("ans, quand il calcula que pi =")
                                        ).push(4).push(std::string("Il avait "));
    print(std::cout, a) << std::endl;
    print(std::cout, b) << std::endl;
    return EXIT_SUCCESS;
}
```

Quelques remarques sur cette classe :

1. À chaque `pop` ou chaque `push`, le type du tas est modifié, et doit donc être affecté à une nouvelle variable, puisque C++ ne supporte pas à l'exécution du programme compilé les types dynamiques;

2. En particuliers, il n'est pas possible de faire une boucle (de type `for` ou `while`) comme pour le tas de la STL, et on doit donc remplacer les boucles par des appels récursifs;

3. De même, les données de notre tas ne peuvent être instanciées que statiquement, et non dynamiquement durant l'exécution du programme : il faut bien voir les templates, à l'instar des expressions constantes comme faisant parti du langage interprété par le compilateur, et ne pouvant être traité par le programme résultant de la phase d'interprétation (de compilation).

5 Extension des variadiques templates en C++ 2017

On a vu qu'en C++ 2011/2014, le déploiement des arguments issus de paramètres variadiques était effectué en séparant chaque argument par l'opérateur `,`. En C++ 2017, il est possible dans une certaine limite de remplacer cet opérateur par un autre opérateur. Par exemple, pour afficher plusieurs variables de types hétérogènes :

```
# include <iostream>

template<typename T, typename ...Tn>
std::ostream& print( std::ostream& out, const T& t, Tn const & ...tn )
{
```

```
    out << t;
    out << "," << ... << tn;
    return out;
}

int main()
{
    print(std::cerr, "tintin", 3.14, 4 );
    return EXIT_SUCCESS;
}
```

De même, pour additionner n variables :

```
# include <iostream>

template<typename T, typename ...Tn> auto add(const T& t, Tn const & ...tn )
{
    return (t+...+tn);
}
```

En somme, si cette extension ne rajoute pas en soit de nouvelles fonctionnalités au langage (on peut toujours remplacer le code ci–dessus par des appels templates récursifs), elle permet une bien meilleure lisibilité et simplicité du code, ce qui est toujours bon à prendre en soi !