Templates avec C++ 2011

Xavier JUVIGNY

ONERA

8 Juillet 2017

Plan du cours

Généricité

Définition

- Générer du code à partir d'un patron et de divers paramètres;
- Généricité fonctionelle : un patron de fonction utilisé pour des types différents. Exemple en C :
  `#define MAX(a,b) ((a)>(b) ? (a) : (b))`
- Macros outil primaire, peut être dangereux. Que penser de `MAX(i++,j)` ?
- Généricité structurelle : une liste d'entier même gestion qu'une liste de réels, de vecteurs, de capteurs,etc.
- Difficile à gérer par macro;
- Solution la plus générale : générer code à partir d'un langage à balise ou d'un DSL (Domain Specific Language).

Généricité en C++

- Généricité en C++ définie à partir de patrons (comme un patron de couture : template)
- **Déclaration** : Mot clef C++ `template` suivi de la liste des paramètres variables du patron entre les signes < et >, suivi de la définition de la fonction ou de la classe patron;
- **Instanciation** : Soit les paramètres templates peuvent être déduits par le compilateur (pour les fonctions et en C++17, pour les classes), soit passés entre les signes < et >
- Lors d'une instanciation de template, le compilateur doit avoir accès à la déclaration mais aussi à la définition de la fonction ou de la classe : impossible de compiler une fonction ou une classe template sans l'instancier pour des paramètres donnés.

Généricité fonctionnelle

Généricité fonctionnelle en C++

- Défini un modèle de fonction ou de méthode de classe;
- **Syntaxe** : modèle précédé du mot clef `template` et de la liste des paramètres du patron entre les signes < et >;
- Exemple de fonction retournant le max de deux valeurs de type comparable

```
template<typename K> inline K max_val( const K& a, const K& b ) {
    return (a > b ? a : b ); }
int main() {
    std::cout << max_val(1,3) << max_val(1.2,4.3-2.1) << max_val('t','l') << std::endl;
    ... }
```

- `K` est le paramètre du template : `typename` précise qu'ici *K* est un paramètre représentant un type de variable;
- Dans ce cas simple, lors de l'instanciation (et de l'appel) d'une fonction à partir du template, le paramètre `K` déduit du type de paramètre passé à la fonction;
- Cependant, `max_val(1.3.5)` ne compilera pas, car types différents, idem pour `max_val("tintin","milou")` (`char[6]` et `char[5]`)
- On peut passer deux paramètres types :

```
template<typename K1,typename K2> auto max_val(const K1& a,const K2& b){return (a>b?a:b);}
```

- Il faut que les valeurs de types `K1` et `K2` soient comparables : `max_val(1.3.5)` et `max_val("tintin","milou")` compilent dans ce cas.
- On renvoit `auto` car déduction de type non trivial avant instanciation de la fonction templatée;
- `max_val("tintin","milou")` peut renvoyer la valeur `"milou"` car comparaison sur adresse chaîne de caractère.

Template et surcharge de fonctions

Surcharge de fonctions

- On a vu que la fonction template `max_val` ne renvoie pas la bonne chaîne de caractère car n'effectue pas la bonne comparaison;
- C++ autorise qu'on définisse la fonction `max_value` pour les chaînes de caractère de type `char*` :

```
std::string max_value( const char* s1, const char* s2 )
{
    if ( strcmp(s1,s2) > 0 ) return std::string(s1); else return std::string(s2);
}
```

- C++ n'instanciera pas une fonction template si une fonction non template existe déjà pour un ou des types de variables données;
- On parle alors de **spécialisation template**;
- Dans notre cas, la fonction `max_value`, grâce à la spécialisation, renvoie bien maintenant la bonne chaîne de caractère...
- **Remarque** : Bien que la fonction soit spécialisée pour des `const char*`, elle est également appelée pour des `char[5]` ou `char[6]` car ces types sont trivialement convertissables en pointeur (sans créer une variable temporaire pour la conversion).

Paramètres template

Les types des paramètres

Le type du paramètre template (qui devra être passé ensuite en constante) peut être :

- un **type générique** : Paramètre est précédé du mot clef `typename` (recommandé) ou `class`;
- un **type intégral** : entier, booléen, mais aussi pointeur, pointeur de fonctions, référenes, etc...mais pas de type réel, etc...
- Un paramètre référence non initialisable avec donnée temporaire ou valeur immédiate lors de l'instanciation.
- Comme pour les fonctions, il est possible de passer des valeurs par défaut aux paramètres template.
- Les paramètres ayant des valeurs par défaut doivent être déclarés à la fin de la liste des paramètres template;

Cas du pointeur nul

Considérons le code suivant :

```
template<typename T> void display_pointer(const T* pt) {
    std::cout << "Pointer at adress " << pt << " with elements of size " << sizeof(T) << std::endl;
}
...
double x; int i;
display_pointer(&x); display_pointer(&i);
display_pointer(NULL); display_pointer(nullptr); // Les deux appels ne compilent pas !
```

Il faut spécialiser la fonction pour le pointeur nul à l'aide du type `std::nullptr_t` :

```
void display_pointer( std::nullptr_t pt ) {
  std::cout << "Null pointer !" << std::endl;
}
display_pointer(nullptr); // Ok, compile maintenant
display_pointer(NULL); // Non portable, non compilable avec certains compilateurs
```

Template de template

Principe

- Le paramètre template possède lui-même un paramètre template;
- Pas possible d'avoir un template de template de template dans les normes actuelles du C++;

```
template<template<typename,typename> class C> C<double,std::allocator<double>> init_vector(int ni) {
    C<double,std::allocator<double>> v;
    for ( int i = 0; i < ni; ++i) v.push_back(i+1.);
    return v;
}

int main() {
    auto z = init_vector<std::list>(10);
    for ( auto x : z ) std::cout << x << " ";
```

- Le paramètre de la fonction au dessus est le type de container, pas le type des valeurs qu'il contient ou le type d'allocation nécessaire pour ces valeurs;
- Il est important de passer deux paramètres templates au paramètre template C pour utiliser un type `vector` car ce dernier réclame deux paramètres templates (un pour le type de valeur contenu, l'autre pour le type d'allocation);
- On peut utiliser des templates de template également pour les structures.

Spécialisation template

Spécialisation template

Possibilité de spécialiser un template pour des valeurs particulières de ses paramètres :

```
template<long n> long fact() { return n * fact<n-1>(); }
template<> long fact<0>() { return 1L; }

std::cout << fact<10>() << std::endl; // L'évaluation de la factorielle se fait à la compilation...
```

Remarque : l'instanciation de la fonction template oblige ici à passer les paramètres entre les symboles < et > car l'entier ne peut se déduire des paramètres passés à la fonction templatée.

Spécialisation partielle

Supposons qu'on veuille calculer la dérivée directionnelle sur un corps K d'une fonction f. Une réalisation possible en template est la suivante :

```
template<typename Func, typename K> K df_s_dh(const Func& f,const K& h,const K& x,
                                              const decltype((std::abs(h))) eps=1.E-6){
    auto nrm_h = std::abs(h); // On devrait traiter erreur si nrm_h < error_epsilon
    K dh = eps*h/nrm_h;
    return K(1./eps) * (f(x+dh) - f(x));
}
```

Formule générale, mais pas optimale pour les réels. On spécialise donc la fonction pour les doubles par exemple :

```
template<typename Func> double df_s_dh(const Func& f,const double& h,const double& x) {
    return (1./h)*(f(x+h)-f(x));
}
```

Spécialisation template et constexpr (C++14)

Template et constexpr

- À partir de C++14, il est possible de templater les expressions constantes
- Permet de définir des valeurs associées à des types génériques;
- Mais aussi de calculer des expressions à l'aide des templates;

```
template<typename I, long n> constexpr I factoriel = I(n) * factoriel<I,n-1>;
template<typename I> constexpr I factoriel<I,0> = I(1);
...
std::cout << factoriel<double,20> << std::endl;
```

Règles sur les spécialisations partielles

- Permet de spécialiser une fonction, une expression constante ou une classe/structure (voir plus loin);
- Permet également de spécialiser selon la nature du type (par exemple spécialiser dans le cas où c'est un pointeur);
- Une valeur ne peut pas être exprimée en fonction d'un paramètre template de la spécialisation :

```
template<int I, int J> struct B { ... };
template<int I> struct B<I,2*I> { ... }; // Erreur, dépendance entre paramètres templates
```

- Le type d'une des valeurs de la spécialisation ne peut pas dépendre d'un autre paramètre :

```
template<typename T, T t> class B { ... };
template<typename T> class B<T,1> { ... }; // Erreur, t dépend de T !
```

Exercice sur les template de fonction

Puissance nième

Calculer la puissance *n* ième (*n* entier positif) d'une valeur de type *K* (pouvant être aussi bien un scalaire qu'une matrice par exemple)

Puissance nième

Calculer la puissance *n* ième (*n* entier positif) d'un double par succession d'appels récursifs résolus à la compilation;

Norme 2D

Écrire une fonction calculant la norme d'un vecteur 2D sur un corps K (réel, complexe, etc...)

Problème

On définit la suite de *fibraction* :

$$\begin{cases} u_1 & = & 1 \\ u_2 & = & 2 \\ u_n & = & u_{n-2} - u_{n-1} \end{cases}$$

Le but est de calculer u_{32} à la compilation et afficher le résultat à l'exécution

Astuce

Les fonctions templatées ne sont générées qu'une fois pour une valeur donnée dans le cadre de la récursion.

Généricté structurelle

Déclaration d'une structure template

- Déclaration et définition semblables à celles d'une fonction template

```
template<paramètres templates> class|struct|union;
```

- Les méthodes peuvent être définies au sein de la déclaration ou bien à l'extérieur de la déclaration;
- Dans le dernier cas, elle doivent être elles-mêmes également déclarées template lors de leur définition;

```
template<typename K> class A { ...
    K func( const K& k ) const;
};
template<typename K> K A<K>::func( const K& k ) const { ... }
```

- les chevrons $<$ et $>$ sont là pour spécifier que c'est la classe qui est template et non la méthode de la classe;
- De manière générale, il faudra toujours référencer une classe template avec la liste de ses paramètres, sauf si on la référence dans une méthode de la classe elle-même;
- Lors de l'instanciation d'une classe template, le compilateur doit avoir accès à la déclaration **et** à la définition de la classe;
- Il est possible d'avoir une méthode elle-même template dans une classe template.

Généricité structurelle

Exemple généricité structurelle en C++

```
namespace Algebra {
    template<typename K>
    class Vecteur {
    public:
        using value=K;
        using container=std::vector<K>;
        using reference=typename container::reference;
        using const_reference=typename container::const_reference;

        Vecteur() = default;
        Vecteur( std::size_t dim ) : m_arr_coefs(dim)  {}
        Vecteur( const std::initializer_list<K>& l ) : m_arr_coefs(l) {}
        Vecteur( const Vecteur& u ) = default;
        Vecteur( Vecteur&& u ) = default;
        ~Vecteur() = default;

        std::size_t dim() const { return m_arr_coefs.size(); }
        reference operator []( std::size_t i)
        { return m_arr_coefs[i]; }
        const_reference operator [] ( std::size_t i ) const
        { return m_arr_coefs[i]; }
        Vecteur operator + ( const Vecteur& u ) const;
    private:
        std::vector<K> m_arr_coefs;
    };
```

Généricité structurelle

Exemple généricité structurelle en C++

```
    template<typename K> Vecteur<K>
    Vecteur<K>::operator + ( const Vecteur<K>& u ) const
    {
        Vecteur w(dim());
        for (std::size_t i = 0; i < dim(); ++i )
            w[i] = u[i]+(*this)[i];
        return w;
    }
    template<typename K> std::ostream& operator << ( std::ostream& out, const Vecteur<K>& u )
    {
        out << "{ ";
        for ( std::size_t i = 0; i < u.dim(); ++i ) out << u[i] << " ";
        out << "}";
        return out;
    }
}

int main()
{
    using Algebra::Vecteur;
    Vecteur<double> u={1.,2.,3.};
    Vecteur<double> v={4.,5.,6.};
    auto w = u + v;
    std::cout << u << "+" << v << "=" << w << std::endl;
```

Organisation, production de code avec les templates

Problématique

- À chaque instanciation d'une classe template, on génère le code adéquat;
- Il arrive souvent de regénérer plusieurs fois le même code pour le même jeu de paramètres templates;
- Le temps de production (compilation) risque de devenir une étape lourde et longue sur des codes importants de type industriel;
- Il faut donc essayer de générer une classe template qu'une seule fois pour un jeu de paramètre donné.

Solution

- Pour instancier un template, le compilateur doit avoir accès aux définitions des méthodes;
- Séparer en deux fichiers la déclaration et la définition (mise en œuvre); : `Vecteur.hpp` et `Vecteur.tpp`
- Créer un troisième fichier (`Vecteur`) incluant le fichier de déclaration et le fichier de définition;
- On peut instancier explicitement une structure ou une fonction template : `template Algebra::Vecteur<double>;`
- Créer un fichier (`Vecteur.cpp`) instanciant les classes templates avec les paramètres courants (bibliothèque ou module)/voulus (application);

Déclaration,Définition des méthodes d'une classe template

Listing 1: Vecteur.hpp

```
namespace Algebra {
template<typename K> class Vecteur :
                     public std::vector<K> {
public:
    using real_t=decltype( std::abs(K(0)) );
    Vecteur( std::size_t dim = 0 ) :
          std::vector<K>(dim) {}
    Vecteur(std::initializer_list<K> coefs);
    Vecteur( const Vecteur& u )=default;
    Vecteur( Vecteur&& u ) = default;
    ~Vecteur();

    //C++ 2014
    auto normL2() const;
    //C++ 2011
    real_t normL2() const;
    Vecteur
    operator + (const Vecteur& u) const;
};
}
```

lateur de

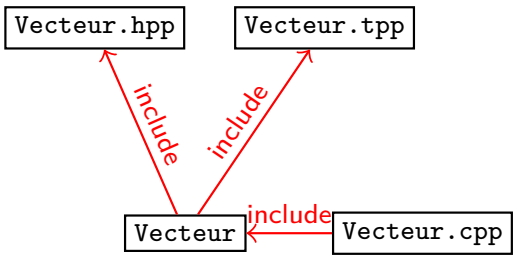
Listing 2: Vecteur.tpp

```
namespace Algebra{
  template<typename K>
  Vecteur<K>::Vecteur(std::initializer_list<K> coefs) :
         std::vector<K>(coefs)
  {}
  template<typename K> Vecteur<K>::~Vecteur() {}
  template<typename K>
  Vecteur<K>::real_t Vecteur<K>::normL2() const {
      Vecteur<K>::real_t nrm{0};
      for ( const auto& val : *this )
          nrm += std::norm(val);
      return std::sqrt(nrm);
  }
}
```

Production avec les templates

- Les définitions (ici fichiers .tpp) doivent être accessibles aux parties de codes les utilisant;
- On peut néanmoins "précompiler" des templates pour certains paramêtres spécifiques et ne pas utiliser dans ce cas le fichier de définition.

Architecture fichiers pour bien gérer les templates



Description des fichiers

- Fichier de déclaration `Vecteur.hpp`

```
namespace Algebra {
    template<typename K> class Vecteur { ... }; }
```

- Fichier de définition `Vecteur.tpp`

```
namespace Algebra {
  template<typename K> Vecteur<K>::Vecteur(...) {...} }
```

- Fichier d'inclusion (pour générer nouveaux types de vecteurs) `Vecteur`

```
#include "Vecteur.hpp"
#include "Vecteur.tpp"
```

- Fichier d'instanciation de types "standards" `Vecteur.cpp`

```
# include "Vecteur"
template Algebra::Vecteur<double>;
template Algebra::Vecteur<float>;
```

Référence universelle

Problématiques

- Considérons le code suivant :

```
template<typename K> class Wrapper : public Base_wrapper
{ public:
    Wrapper( K& a1 ) : m_obj(a1) {}
    Wrapper( K& a1, K& a2) : m_obj(a1,a2) {}
  private:
    K m_obj; };
 Wrapper<std::pair<double>> p( 3., 4. );
```

- Le constructeur de `Wrapper` a besoin de passer une liste arbitraire d'arguments à l'objet de type `K`;
- Le code ne compile pas car on ne peut pas passer des valeurs par référence;
- Une solution est de passer plutôt des références constantes, mais si l'objet K demande une référence non constante ?
- On peut alors convertir a1 et a2 en référence non constante : `const_cast<K&>(a1);`
- Problème: maintenant, Wrapper peut modifier des valeurs constantes (au travers de l'objet K).
- Une solution qui marche : considérer toutes les combinaisons de références constantes ou non :

```
template<typename K> class Wrapper : public Base_wrapper
{public:
  ...
  Wrapper( K& a1, K& a2) : m_obj(a1,a2) {}
  Wrapper( const K& a1, K& a2) : m_obj(a1, a2) {}
  Wrapper( K& a1, const K& a2) : m_obj(a1, a2) {}
  Wrapper( const K& a1, const K& a2 ) : m_obj(a1, a2) {} ...
```

- Problème : 2^N combinaisons à écrire (N = nbre arguments)

Référence universelle

Référence universelle

```
Wrapper( K&& a1 ) : m_obj(a1) {}
Wrapper( K&& a1, K&& a2 ) : m_obj(a1, a2) {}
```

- Passage d'une variable : passée par référence
- Passage d'une valeur : passée par rvalue (id. référence constante)

Forwarding

- On veut parfois garder le type de passage d'un argument (pour faire un retour par déplacement par exemple ou si la fonction appelée demande une référence universelle);

```
Wrapper( K&& a1 ) : m_obj(std::forward<K>(a1)) {}
```

Méthodes templates

Template de méthodes

- Dans une classe, on peut définir des méthodes template;
- La classe peut-être template ou non.

```
class Matrix {
    Matrix( std::size_t& nrow, std::size_t& ncol );
    template<typename InputIterator>
    Matrix( std::size_t& nrow, std::size_t& ncol, const InputIterator& it );
    template<typename Vec> Vec multVec( const Vec& u ) {
        assert(u.size() != nb_columns());
        Vec v(nb_rows());
        for ( int i = 0; i < nb_rows(); ++ i ) {
            v[i] = 0.;
            for ( int j = 0; j < nb_columns(); ++j ) {
                v[i] += (*this)(i,j) * u[j];
            }
        }
        return v;
    }

    ...
};
```

Exercices

Matrice de rotation 2D

Écrire une classe représentant une rotation 2D d'angle α (le corps de base pouvant être réel ou complexe);

Polynôme

Écrire une classe représentant un polynôme (l'anneau considéré pouvant être non commutatif comme pour les matrices par exemple).

- Calcul dérivée et primitive;
- Évaluation du polynôme en une valeur;
- Addition, soustraction et multiplication de deux polynôme
- Sauvegarde et affichage des polynômes.

Les expressions template

Problématique

- Soit une classe représentant des vecteurs algébriques : on y a programmé l'addition, le produit par un scalaire et le produit scalaire;

```
template<typename K>
class Vecteur<K> { ...
    Vecteur operator + ( const Vecteur& u ) const;
    K operator | ( const Vecteur& u ) const;
    Vecteur homothetie( const K& alpha ) const;
};
template<typename K> Vecteur<K> operator * ( const K& alpha, const Vecteur<K>& u ) {
    return u.homothetie(alpha); }
```

- Que penser de l'expression : `w = v - (v|u)*u` ?
- Le compilateur va générer : un scalaire intermédiaire (le produit scalaire), un premier vecteur intermédiaire (pour calculer `(v|u)*u`, un second vecteur pour calculer `w = v - (v|u)*u`.
- Le constructeur et l'opérateur de déplacement sauve un peu la mise, mais : on créer deux vecteurs intermédiaires et chaque boucle effectuant les opérations sur les vecteurs demandent peu d'instructions...

Première approche : Calcul d'une factorielle par le compilateur

- On utilise les variables templates permises par C++ 14;
- On exprime la factorielle sous forme de récursion sur un paramètre template entier;

```
template<typename T, long n> constexpr T factorial = factorial<T,n-1>*T(n);
template<typename T> constexpr T factorial<T,0> = 1L;
int main() {
    std::cout << factorial<long, 5> << std::endl;
```

Les expressions templates...

Autre exemple un peu plus complexe

Calcul de l'approximation entière supérieure d'une racine carrée par encadrement

- Cas un peu plus complexe : il faut utiliser d'autres variables templates intermédiaires;
- Qu'on peut regrouper au sein d'une structure : allège et clarifie le code;

```
template<std::size_t N, std::size_t Low, std::size_t Upp>
struct IRoot_gen{
    static constexpr std::size_t mean = (Low+Upp)/2;
    static constexpr std::size_t down = ((mean*mean)>=N);
};
template<std::size_t N, std::size_t Low=1, std::size_t Upp=N> constexpr std::size_t IRoot =
            IRoot<N, (IRoot_gen<N,Low,Upp>::down ? Low : IRoot_gen<N,Low,Upp>::mean+1),
                     (IRoot_gen<N,Low,Upp>::down ? IRoot_gen<N,Low,Upp>::mean : Upp)>;
template<std::size_t N, std::size_t Mean> constexpr std::size_t IRoot<N,Mean,Mean> = Mean;
int main() {
    std::cout << "Racine entiere superieure de 24 = " << IRoot<24> << std::endl;
    return 0;
```

Remarques

- Pour l'intant, les exemples donnés auraient pu être programmés plus simplement à l'aide d'expressions constantes...
- Mais les templates permettent bien plus que cela...

Les expressions templates...

Produit scalaire déroulé à la compilation

- On va se baser sur le fait que le produit scalaire de deux vecteurs de dimension N peut être calculer par le produit de deux scalaires sommé au produit scalaire de deux vecteurs de dimension $N-1$...
- La boucle sur les éléments des deux vecteurs sera ainsi complètement déroulée par le compilateur !
- Plus d'appel de fonction, le produit scalaire sera directement remplacé par l'expression arithmétique correspondante.

```
template<typename InputIterator, std::size_t N> struct DotProduct_generator {
    static auto eval( const InputIterator& iter_a, const InputIterator& iter_b ) {
        return DotProduct_generator<InputIterator,1>::eval(iter_a,iter_b) +
               DotProduct_generator<InputIterator,N-1>::eval(std::next(iter_a),std::next(iter_b));
    }
};
template<typename InputIterator> struct DotProduct_generator<InputIterator,1> {
    static auto eval( const InputIterator& iter_a, const InputIterator& iter_b ) {
        return (*iter_a)*(*iter_b);
    }
};
template<std::size_t N, typename InputIterator> inline auto dot( const InputIterator& u, const
      InputIterator& v ) {
    return DotProduct_generator<InputIterator,N>::eval(u,v);
}
int main() {
    std::array<double,4> u = {1.,2.,3.,4.};
    std::array<double,4> v = {4.,3.,2.,1.};
    std::cout << "(u|v) = " << dot<4>(u.begin(),v.begin()) << std::endl;
    std::list<int> li = {1,2,3,4};
    std::list<int> lj = {4,3,2,1};
    std::cout << "(u|v) = " << dot<4>(li.begin(),lj.begin()) << std::endl;
```

Les expressions templates...

expression template et arbre AST

- Généralement, les expressions templates génèrent un arbre AST (Abstract Syntax Tree) à partir d'un chaînage d'opérateurs (+, -, x, /, etc...)
- Permet, pour les vecteurs, par exemple, d'exprimer simplement par la suite le calcul à faire sur chaque composante, sans calcul de vecteurs intermédiaires;
- Mise en œuvre complexe mais de nombreuses littératures sur le sujet;
- Il existe une librairie générale permettant "facilement" de mettre en œuvre des expressions templates : Boost.Proto d'Éric Niebler ;
- De nombreux tutorials et exposés là dessus sur internet : Serge Sanspaille (`http://linuxfr.org/users/serge_ss_paille/journaux/c-14-expressions-template-pour-les_nuls` et Joël Falcou (CppCon 2014) ainsi que le tutorial d'Eric Niebler sur Boost.Proto (`http://ericniebler.com/category/proto/`)

Exercice sur les expressions templates

Addition de vecteurs de dimension N

Le but de cet exercice est d'écrire une expression template qui déroulera toute la boucle pour additionner deux vecteurs dans un troisième vecteur.

- En s'inspirant de l'expression template permettant le produit scalaire de deux vecteurs, écrire une expression template permettant l'addition de deux vecteur dans un troisième.
- Vérifier qu'on ne génère pas de fonction si on choisit l'option d'optimisation `-O2`;
- Essayer l'expression template en utilisant divers conteneurs (tableaux statiques, listes, etc...).

Stratégies template

Politique

- À l'aide de la spécialisation, permet de changer de comportement selon le type.

```
template<typename K> struct MPI_Type { static MPI_Datatype id_type() { return MPI_PACKED; } };
template<> struct MPI_Type<short> { static MPI_Datatype id_type() { return MPI_SHORT; } };
template<> struct MPI_Type<double> { static MPI_Datatype id_type() { return MPI_DOUBLE; } };
...
template<typename K> bool send( std::size_t nbItems, const K* buffer, int dest, int tag ) {
    MPI_Send( buffer, nbItems, MPI_Type<K>::id_type(), dest, tag, MPI_COMM_WORLD);
}
```

SFINAE : Substitution failure is not an error

Permet de restreindre un template à certains types

```
template<typename K> struct restrictor        {                        };
template<>           struct restrictor<float> { typedef float  result; };
template<>           struct restrictor<double>{ typedef double result; };

template<typename Real> typename restrictor<Real>::result
distance( K a1, K a2, K b1, K b2 ) { ... }
```

Introspection par template

Techniques nécessaires

- SFINAE + constexpr
- Utilisation de `std::declval<K>()` et `std::decltype(expr)` :

```
#include <utility>
struct Default { int foo() { return 1; } };// Struct avec constructeur par défaut
struct NonDefault { NonDefault(int i) {} // Struct sans constructeur par défaut
                    int foo() { return 1; } };
int main() {
    decltype(Default().foo()) n1 = 1; //ok, n1 est un entier
    decltype(NonDefault().foo()) n2 = 2;// Erreur, NonDefault n'a aucun constructeur par défaut
    // std::declval transforme un type en référence sur obj de ce type
    decltype(std::declval<NonDefault>().foo()) n3 = 3;// ok, n3 est un entier
}
```

Introspection

Teste si une classe possède une méthode particulière :

```
template<class K> struct has_serialize {
    template<typename C> static constexpr
    decltype(std::declval<C>().serialize(std::cout), bool()) test(int){
        return true; } // symbole , évalue de gauche à droite
    template<typename C> static constexpr bool test(...) {
        return false; }
    // La valeur booléenne finale évaluée par le compilateur pour savoir si
    // C contient une méthode serialize
    static constexpr bool value = test<K>(int());
};
```

Utilisation de l'instrospection

Problème

- Considérons le programme suivant :

```
template<typename C> std::ostream& serialize(const C& obj, std::ostream& out) {
  if (has_serialize<C>::value) return obj.serialize(out);
  else return obj.fmtSave(out);
}
```

- Ne compile pas car même si `has_serialize<C>::value` faux à la compilation, il compile quand même la branche morte;

Solution

- Utiliser `std::enable_if` avec template:

```
template<typename K> typename std::enable_if<has_serialize<K>::value,std::ostream&>::type
serialize( const K& obj, std::ostream& out ) {
    out << "Serialisation de obj..." << std::endl;
    return out; }
template<typename K> typename std::enable_if<!has_serialize<K>::value,std::ostream&>::type
serialize( const K& obj, std::ostream& out ) {
    out << "Pas de serialisation de obj..." << std::endl;
    return out; }
```

- `std::enable_if` peut-être passé en type de retour, en paramêtre de fonction "stupide" ou en paramêtre template (avec spécialisation).

Curiously Reccuring Template Pattern (CRTP)

Principes

- Formalisé dans les années 1980 comme le *F-bounded quantification*;
- Une classe `X` dérive d'une classe template ayant pour argument template la classe `X` elle-même;

```
template<typename K> struct Base { };        // Méthode Base utilise template pour
class Derived : public Base<Derived> { ... };// accéder membres classes dérivées
```

Exemple d'utilisation

- Polymorphisme statique

```
template <typename T> struct Base {
    void implementation() { ... static_cast<T*>(this)->implementation(); ... }
    static void static_func() { ... T::static_sub_func(); ... } };
struct Derived : Base<Derived> { void implementation();
                                 static void static_sub_func(); };
```

- Compteur d'objets :

```
template <typename T> struct counter {
    static int objects_created; static int objects_alive;
    counter() { ++objects_created; ++objects_alive; }
    counter(const counter&) { ++objects_created; ++objects_alive; }
protected: // objects should never be removed through pointers of this type
    ~counter() { --objects_alive; } };
template <typename T> int counter<T>::objects_created( 0 );
template <typename T> int counter<T>::objects_alive( 0 );
class X : counter<X> { ... };
```

CRTP (suite)

Autres exemples

- Constructeur de copie polymorphique

```
class Shape {// Base class has a pure virtual function for cloning
public:
    virtual ~Shape() {};
    virtual std::shared_ptr<Shape> clone() const = 0; };
template <typename Derived> class Shape_CRTP : public Shape {
public: // This CRTP class implements clone() for Derived
    virtual std::shared_ptr<Shape> clone() const override
    { return std::make_shared<Derived>(static_cast<Derived const&>(*this)); } };
// Nice macro which ensures correct CRTP usage
#define Derive_Shape_CRTP(Type) class Type: public Shape_CRTP<Type>
Derive_Shape_CRTP(Square) {};// Every derived class inherits from Shape_CRTP instead of Shape
Derive_Shape_CRTP(Circle) {};
```

- Chaînage polymorphique

```
template <typename ConcretePrinter> class Printer { // Base class
public:
    Printer(ostream& pstream) : m_stream(pstream) {}
    template <typename T> ConcretePrinter& print(T&& t) {
        m_stream << t; return static_cast<ConcretePrinter&>(*this); }
    template <typename T> ConcretePrinter& println(T&& t) {
        m_stream << t << endl; return static_cast<ConcretePrinter&>(*this); }
private:
    ostream& m_stream; };
class CoutPrinter : public Printer<CoutPrinter> { // Derived class
public:
    CoutPrinter() : Printer(cout) {}
    CoutPrinter& SetConsoleColor(Color c) { ... return *this; } };
// usage CoutPrinter().print("Hello ").SetConsoleColor(Color.red).println("Printer!");
```

Autres utilitaires templates fournis par C++11/14

Propriétés types basiques type_traits

- is_void
- is_null_pointer std::nullptr_t ?
- is_integral
- is_floating_point
- is_array
- is_enum
- is_union
- is_class
- is_function
- is_pointer
- is_lvalue_reference
- is_rvalue_reference
- is_member_object_pointer
- is_member_function_pointer

Propriétés types composés

- is_fundamental
- is_arithmetic
- is_scalar
- is_object
- is_compound
- is_reference
- is_member_pointer

Propriété des types

- is_const
- is_volatile
- is_trivial : objet possède un constructeur et une copie définis par défaut
- is_trivially_copyable
- is_standard_layout : Type pouvant être échangé avec d'autres langages
- is_pod : Plain Old Data type?
- is_empty : no data in object ?
- is_polymorphic
- is_abstract

Propriété des types (suite)

- is_final
- is_signed
- is_unsigned

Propriétés opérations

- is_constructible
- is_default_constructible
- is_copy_constructible
- is_move_constructible
- is_assignable
- is_copy_assignable
- is_move_assignable
- is_destructible
- has_virtual_destructor

Autres utilitaires templates fournis par C++11/14 (suite)

Propriétés types

- `alignment_of`
- `rank` : Nombre dimensions tableau statique
- `extent` : Taille tableau statique dans un dimension
- `is_same` :vrai si deux types sont les mêmes
- `is_base_of` : Vrai si un type dérive d'un autre
- `is_convertible` : Vrai si type convertissable en autre type

Modification de type

Créé nouvelle définition de type en applicant modifications sur paramêtre template.

- `remove_cv` : Enlève spécif. const ou volatile au type
- `remove_const`
- `remove_volatile`
- `add_cv` Rajoute spécifs const et volatile au type
- `add_const`
- `add_volatile`
- `remove_reference`
- `add_lvalue_reference`
- `add_rvalue_reference`
- `remove_pointer`
- `add_pointer`

Modifications type (Suite)

- `make_signed` : type entier devient signé
- `make_unsigned`
- `remove_extent` : type contenu dans un tableau
- `remove_all_extents` : type contenu dans tableaux imbriqués
- `aligned_storage` : type adapté comme stockage non initialisé pour un type;
- `aligned_union` : type adapté comme stockage non initialisé pour divers types donnés
- `decay` : type basique du type passé en paramètre: Exemple, type tableau sur T converti en `T*`;
- `enable_if` : Cache surcharge ou spécialisation de fonction selon une valeur booléenne à la compilation;
- `conditional` : Choisi un type ou un autre selon booléen
- `common_type` : type commun d'un groupe de types;
- `underlying_type` : Entier sous jacent pour type énumération donné
- `result_of` : type résultat d'un objet appelable avec un ensemble d'arguments. Attention : template obsolète en C++17, remplacé par `invoke_result`.

Classe d'aide

`integral_constant` : Défini une constante spécifique d'un type entier spécifique

Exercice sur l'introspection

Exercice sur le SFINAE

- Écrire une fonction calculant le median d'un ensemble de valeurs contenues dans un conteneur;
- Cette fonction devra marcher pour tout conteneur ayant une méthode `size()` et des itérateurs;
- On optimisera cette fonction pour tous les conteneurs à accés direct, c'est à dire possédant l'opérateur `[]`.
- Astuce : Regarder les fonctions `std::begin`, `std::advance` et `std::next`.

template variable

Définition

- Avant C++ 11, pour avoir arguments variables dans une fonction : utilisation des ellipses et des macros va_...;
- Traitement des arguments variables résolus à l'exécution alors que les arguments étaient connus à la compilation d'un exécutable;
- C++ 11 introduit les templates variables
- Permet entre autre de gérer les fonctions à nombre variable d'arguments à la compilation

Exemple

```
template<typename K> K adder( K val ) { return val; }
template<typename K, typename ... Args> K adder( K first, Args ... Args ) {
    return first + adder(args...);
}

int main() {
    ...
    std::cout << adder(1,3,5,7,13) << std::endl;
    std::string a1("tin"), a2("et"), a3("milou");
    std::cout << adder(a1, a1, a2, a3) << std::endl;
    ...
}
```

- adder accepte un nombre quelconque de paramètres; Écrit sous forme récursive;
- Compile sans problème si paramètres acceptent addition; Fonctions évaluées à la compilation.
- `typename ... Args` : **paquet de paramètre template**;
- `Args ... args` : **paquet d'arguments de fonction**.

Templates variables et pattern matching

Problématique et solution

- Vérification des arguments du template à la compilation (par exemple : même type deux à deux)

```
template<typename T> bool pair_compare( const T& a, const T& b ) { return a == b; }
template<typename K,typename ... Args> bool pair_compage( const K& a,const K& b,Args... args ) {
    return (a==b) && pair_compare(args...);
}
int main() {
   ...
   pair_compare( 3, 3, 1.5, 1.5, 'a', 'a');// retourne true
   pair_compare( 3, 3, 1.5, 1.5, 2 );// Ne compile pas, nbre impair args
   pair_compare( 3, 3., 4, 4 ); // Ne compile pas, type(3) != type(3.)
   ...
}
```

Un exemple un peu plus complexe

```
template <typename ...T> std::list<std::tuple<T...>> simple_zip(std::list<T>... lst) {
  std::list<std::tuple<T...>>  result;
  struct {
    void operator()(std::list<std::tuple<T...>> &t, int c, typename std::list<T>::iterator ...it) {
      if(c == 0) return;
      t.emplace_back(std::move(*it++)...);
      (*this)(t, c-1, it...); }
  } zip;
  zip(result, std::min({lst.size()...}), lst.begin()...);
  return result;
}
std::list<std::tuple<>> simple_zip() { return {}; }
```

Utilisation des variadic template pour initialisation inplace

Problématique

- Pouvoir initialiser tous les objets d'un conteneur à l'aide d'une fonction, sans copie d'un objet externe
- On ne connaît pas *a priori* les arguments nécessaires à cette fonction

Solution

- Utilisation des variadic template :

```
class Vecteur : public std::vector<double> {
public:
    Vecteur( int dim, double value = 0 ) : std::vector<double>(dim,value) {}
    template<typename... Args>
    Vecteur( int dim, double (*f)(int, Args...), Args... args ) : std::vector<double>()
    {
        reserve(dim); for ( int i = 0;  i < dim; ++i ) emplace_back(f(i,args...));
    }
};
double f(int i, double scal ) { return scal * i * i; }
int main() {
    Vecteur u{10, 3.14};
    Vecteur v{10, f, 0.5};
    ...
}
```

- `emplace_back` n'était pas vraiment nécessaire ici, car que des doubles
- Mais serait nécessaire si les éléments du vecteur étaient des objets contenant beaucoup de données;
- Ici on a utilisé un pointeur de fonction, mais en fait, un paramêtre template sur la fonction serait beaucoup plus souple.

Exercices

- **Composition de n fonctions**
  - Écrire une fonction template qui évalue la composition de n fonctions données en paramètres
  - Écrire une fonction template qui additionne aux valeurs contenues dans un conteneur les valeurs d'un autre conteneur sur lequel on a appliqué n fonctions;
  - Il faut s'assurer de ne pas créer de structures intermédiaires.
- **Génération de nuage de points procédurale**
  - Reprendre la classe Nuage de points fait aux TPs sur les classes;
  - Rajouter un constructeur qui prend une fonction qui selon divers paramètres et un indice i va générer N points.

Structure de données variable

Caractéristiques

- Les structures/classes utilisateurs en C/C++ sont définies et fixées à la compilation;
- Impossible à l'exécution de rajouter de nouveaux champs à une structure;
- Les templates variables peuvent définir des structures de données avec un nombre de champs arbitraire;

Définition d'un tuple

```
template<typename... Ts> struct tuple{};
template<typename T, typename... Ts> struct tuple<T,Ts...>: tuple<Ts...> {
    tuple(T t, Ts... ts) : tuple<Ts...>(ts...), tail(t) {}

    T tail;
};
tuple<double, uint64_t, const char*> t1(12.2, 42, "big");
```

- Comment accéder aux champs d'une structure à nombre variable de champs ?
- On sait accéder au premier champs n (`tail`)
- Par récurrence, on peut accéder au $k^{\text{ème}}$ champs;
- Il va falloir également savoir quel type retourner pour chaque champs...
- Par récurrence également; Le tout sera résolu à la compilation.

Exemple de structure variadic : le tuple

Accès au type du $k^{\text{ème}}$ élément

- On crée une structure d'aide pour accèder au type du $k^{\text{ème}}$ élément
- Crée de manière récursive

```
template<typename T,typename... Ts> struct elem_type_holder<0, tuple<T,Ts...>> {
    typedef T type;
};
template<size_t k, typename T,typename... Ts> struct elem_type_holder<k, tuple<T,Ts...>> {
    typedef typename elem_type_holder<k-1,tuple<Ts...>>::type type;
};
```

Accesseur au $k^{\text{ème}}$ élément

- On se sert de la structure d'aide + définition récursive :

```
template <size_t k, class... Ts>
typename std::enable_if<k==0, typename elem_type_holder<0, tuple<Ts...>>::type&>::type
get(tuple<Ts...>& t) { return t.tail; }
// ................................................
template <size_t k, class T, class... Ts>
typename std::enable_if<k!=0, typename elem_type_holder<k, tuple<T, Ts...>>::type&>::type
get(tuple<T, Ts...>& t) {
  tuple<Ts...>& base = t; return get<k - 1>(base); }
```

Exercice sur les templates

- Tableau statique à *n* dimensions
  - Á l'aide d'un variadique sur des `size_t`, concevoir un tableau statique à N entrées;
- Vecteur procédural
  - Créer une classe Vecteur algébrique avec élément de type générique;
  - Définir une méthode d'orthonormalisation d'une base de ces vecteurs, évalué à la compilation, à l'aide d'un algorithme de gram-schmidt pouvant prendre en argument un nombre quelconque de vecteurs
  - Algorithme de gram-schmidt (version récursive en pseudo syntaxe !) :

```
void axspy( Vecteur<K>& y, K& a, const Vecteur<K>& x ) { y -= a*x; }
void axspy( Vecteur<K>& y, K& a, const Vecteur<K>& x, ... ) { y -= a*x; axspy(y,...); }
void gram_schmidt( Vecteur& u ) { u.normalize(); }
void gram_schmidt( Vecteur& v, Vecteur& u ) { gram_schmidt(u); v = v - (v|u).u;
                                              v.normalize(); }
void gram_schmidt( Vecteur& v, Vecteur& u, ... )
{ gram_schmidt(u,...); axspy(v,x,...); v.normalize();}
```

  - Attention lorsque K est un complexe !
  - Tester l'algorithme sur un jeu de paramètre réel :

```
Vecteur<double> u1{1.,1.,1.,1.}, u2{2.,1.,1.,1.}, u3{2.,2.,1.,1.}, u4{2.,2.,2.,1.};
```

  - Puis sur un jeu de paramètre complexe :

```
// Syntaxe C++ 14 pour les complexes :
Vecteur<std::complex<double>> z1{1.+1i,1.+1i,1.+1i,1.+1i},
                              z2{1.+2i,1.+1i,1.+1i,1.+1i},
                              z3{1.+1i,1.+2i,1.+1i,1.+1i},
                              z4{1.+1i,1.+1i,1.+2i,1.+1i},
```

Exercices (suite...)

Produit cartésien d'ensemble

- Écrire un programme qui permet à partir d'un ensemble de valeurs homogènes ou non de former le produit cartésien de cet ensemble par lui-même.
- Par exemple, l'ensemble $\{"Tin", 3.14, "Mi"\}$ donne par le produit cartésien par lui-même :

$$\begin{array}{ll}\{ & ["Tin","Tin"],["Tin",3.14],["Tin","Mi"],[3.14,"Tin"],\\ & [3.14,3.14],[3.14,"Mi"],["Mi","Tin"],["Mi",3.14],["Mi","Mi"]\ \}\end{array}$$

- On utilise pour former le tuple de ces pairs d'entités, la fonction `std::tuple_cat` qui concatène *n* tuples en un seul tuple.
- On programmera également une fonction permettant d'affichier l'ensemble obtenu par le produit cartésien
- Exemple de programme principal :

```
auto cp = selfCartesianProduct(1,2,3,4,5);
print(cp);
```

Conclusion temporaire

C++ un langage mixte interprété/compilé ?

- Template + expressions constantes : permet d'effectuer des traitements complexes sur des données **statiques**;
- Les données dynamiques seront traitées par l'exécutable produit par le compilateur;
- Si le code ne contient que des données statiques : le calcul complet peut être fait par le compilateur;
- les résultats peuvent être extraits de l'assembleur ou du binaire produit par l'exécutable;
- Tous les caractéristiques d'un interpréteur évolué : introspection, typage anonyme des paramètres de fonction, etc... : ce qu'on a dans python.
- Avec une syntaxe néanmoins plus complexe !

Le C++ comme EDSL

- EDSL : Embeded Domain Specific Language;
- Expression templates : Permet de rajouter des extensions au langage en fabricant un arbre AST (Abstract Syntax Tree);
- Voir pour cela la bibliothèque Boost.Proto : permet de créer une grammaire et de nouvelles fonctionnalités évaluées par expression template;