Templates

Introduction C++

Xavier JUVIGNY, AKOU, DAAA, ONERA xavier.juvigny@onera.fr

Initiation au C++
- 4 Novembre 2023 -

Plan du cours

Overview

Généricité

Définition

- Pouvoir définir des **algorithmes** ou des **types** pouvant opérer ou gérer sur plusieurs types de données différents;
- Permet d'abstraire un ensemble de concepts cohérents pour construire des algorithmes au–dessus;
- **Exemple** : Fonction de tri besoin d'accès aléatoires à une collection de valeurs et une fonction de comparaison;
- La fonction devient utilisable dans un très grand nombre de contextes;

Principe

- Générer du code à partir d'un patron (≡ patron de couture) 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++

- Généricité en C++ définie à partir de patrons (comme un patron de couture : template en anglais);
- **Syntaxe** : Mot clef `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** :
  - On fait suivre le type de la valeur ou le nom de la fonction par la liste des paramètres templates entre $<$ et $>$;
  - Pour les fonctions, on laisse le C++ déduire les paramètres templates en fonction du type des paramètres passés en argument;
  - Depuis C++ 17, si une valeur est initialisée à sa déclaration, le C++ déduit les paramètres passés à partir du type des paramètres passés pour l'initialisation;
- 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;
- On peut néanmoins "précompiler" des fonctions ou classes templates pour des types spécifiques.

Généricité fonctionnelle

Généricité fonctionnelle en C++

- Définie 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'appel d'une fonction à partir du template, le paramètre `K` déduit du type de paramètre passé;
- `max_val(1.3.5)` ne compile pas, car types $\neq$, 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 types `K1` et `K2` soient comparables : `max_val(1.3.5)` et `max_val("tintin","milou")` compilent.
- 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 templates

Les types des paramètres

Les types de paramètres templates autorisés :

- **Type générique** : Paramètre précédé du mot clef `typename` (recommandé) ou `class`;

```
template<typename K> auto sum( std::vector<K> const &u ) {
  K s(0);
  for (auto const& valeur : u ) s += valeur;
  return s;
}
...
std::vector<double> u; ... auto vsomme = sum(u.begin(), u.end());
```

- **Type intégral** : entier, booléen, pointeur, pointeur de fonctions, références, etc...mais interdiction type réel, etc...

```
template<std::size_t dim> auto sqNorm( std::array<double,dim> const& u ) {
  double s = 0;
  for (std::size_t i=0; i < dim; ++i ) s += u[i]*u[i];
  return s;
}
```

Paramètres templates...

- Possible d'avoir des types génériques et intégraux :

```
template<typename K, std::size_t dim> K sqNorm( std::array<K,dim> const& u ) {
  K s = 0;
  for (std::size_t i=0; i < dim; ++i ) s += u[i]*u[i];
  return s;
}
```

- Beaucoup de subtilités quand on utilise les templates :

Ne compile pas

```
template<typename Iterator>
auto sum( Iterator const &itBeg,
          Iterator const &itEnd ) {
  decltype(*itBeg) s=0;
  for (Iterator it = itBeg; it != itEnd; ++it )
    s += *it;
  return s;
} ...
std::vector<double> u; ...
auto vsomme = sum(u.begin(), u.end());
```

Compile

```
template<typename Iterator>
auto sum( Iterator const &itBeg,
          Iterator const &itEnd ) {
  std::remove_reference_t<decltype(*itBeg)> s=0;
  for (Iterator it = itBeg; it != itEnd; ++it )
    s += *it;
  return s;
} ...
std::vector<double> u; ...
auto vsomme = sum(u.begin(), u.end());
```

Paramètres templates par défaut

- 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;

```
template<typename K, int dim=3> auto createCanonicalBase() {
  std::vector<std::array<K,dim>> base; base.reserve(dim);
  for (int i=0; i<dim; ++i) {
    std::array<K,dim> ei;
    ei.fill(0); ei[i] = 1;
    base.push_back(ei);
  }
  return base;
}
...
auto base3D = createCanonicalBase<double>();
auto base2D = createCanonicalBase<int,2>();
```

Remarque : Dans cet exemple, on ne peut pas déduire les paramètres templates à partir des paramètres de la fonction. On est donc obligé d'expliciter ici au moins le type.

Cas d'utilisation des templates

Passage de fonctions lambdas

- En C++, chaque fonction lambda possède son propre type inconnu du programmeur;
- Problème pour pouvoir passer une fonction lambda en paramètre d'une fonction;
- En C++ 20, on peut utiliser un paramètre déclaré `auto`;
- Sinon, on doit passer par un template :

```
template<typename EvalFctType> auto
computeVector( int dim, EvalFctType const& evalFct ) {
  std::vector<decltype(evalFct(0))> u(dim);
  for (std::size_t i=0; i<dim; ++i) u[i] = evalFct(i);
  return u; } ...
auto u1 = computeVector(5, [](int i){ return 1.5*i; } );
```

Avec cette technique, on peut également passer des foncteurs (des objets avec opérateur d'évaluation) :

```
struct Kernel { double kWave; std::vector<double>  radius;
  std::complex<double> operator () (int i) const  { return std::polar(1./radius[i], kWave*radius[i]); }
};
Kernel kernelFct{ 1.1, std::vector{1.,3.,5.,7.,13.}};
auto u2 = computeVector( kernelFct.radius.size(), kernelFct );
```

Spécialisation template

Spécialisation template

Possibilité de définir une mise en œuvre particulière pour des types ou valeurs particuliers des paramètres templates.
Deux possibilités :

- Si les paramètres templates ne portent que sur des types associés aux types des arguments passés à la fonction, on peut spécialiser le template avec une fonction "classique"
  Exemple :

```
template<typename K>
auto dot( std::vector<K> const& u, std::vector<K> const& v ) {
  assert(u.size() == v.size());
  typename std::vector<K>::value_type sum = 0;
  for (std::size_t i=0; i< u.size(); ++i ) sum += u[i]*v[i];
  return sum;
}
double dot( std::vector<double> const& u, std::vector<double> const& v) {
  return ddot_(u.size(), u.data(), 1, v.data(), 1);// Utilisation du BLAS
}
```

Spécialisation template (suite)

Spécialisation template (suite)

- Sinon, il faut utiliser un template "spécialisé"
  Syntaxe:

```
template<> (retour fonction) (fonction)<param1,param2,...>( arguments... ) {
  ...
}
```

Exemple de spécialisation template sur une valeur entière :

```
template<long n> long fact() { return n * fact<n-1>(); }
template<> long fact<0>() { return 1L; }
// L'évaluation de la factorielle peut se faire à la compilation (mais pas obligatoire)...
std::cout << fact<10>() << std::endl;
```

Remarque 1 : Pour évaluer une factorielle à la compilation, il est préférable d'utiliser une fonction `constexpr` !
Remarque 2 : Néanmoins, pour des algorithmes plus complexes, cette technique template peut avoir son utilité.

Spécialisation partielle

Spécialisation partielle

- En spécifiant uniquement qu'une partie des paramètres templates

```
template<typename K1, typename K2>  auto prod(std::vector<K1> const& u, std::vector<K2> const& v) {
      std::vector<decltype(std::conj(u[0])*v[0])> res(u.size());
      for (std::size_t i=0; i<u.size(); ++i) res[i] = std::conj(u[i])*v[i];
      return res; }
  template<typename K2> auto prod( std::vector<double> const& u, std::vector<K2> const& v ) {
      std::vector<decltype(K2(0)*1.)> res(u.size());
      for (std::size_t i=0; i<u.size(); ++i) res[i] = u[i]*v[i];
      return res; }
```

- En spécialisant la fonction pour une sous-classe de type

```
template<typename K> auto dot( std::vector<K> const& u, std::vector<K> const& v ) {
  K sum = 0; for (std::size_t i=0; i< u.size(); ++i ) sum += u[i]*v[i];
  return sum; }
template<typename K> auto dot( std::vector<std::complex<K>> const& u,
                               std::vector<std::complex<K>> const& v ) {
  std::complex<K> sum = 0.; for (std::size_t i=0; i<u.size(); ++i) sum += std::conj(u[i])*v[i];
  return sum; }
```

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;
```

Spécialisations partielles

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 !
```

Spécialisation template : principe de fonctionnement (SFINAE)

Principe d'instanciation d'un compilateur pour les spécialisations template

- Le compilateur recherche si une fonction sans paramètre template est définie et l'instancie le cas échéant;
- Si ce n'est pas le cas, recherche si une version partiellement spécialisée est définie et peut être utilisée sans échec par le compilateur. Si c'est le cas, le compilateur instancie la fonction à partir de cette spécialisation partielle.
- Enfin, en dernier lieu, le compilateur cherche à instancier la fonction à partir d'une version template générale (il peut y en avoir plusieurs). Si cela se traduit par un échec, le compilateur renvoie une erreur;
- Le comportement du compilateur est largement utilisé par les programmeurs C++ pour contrôler le comportement du compilateur;
- Cela permet entre autre de faire de l'introspection avec C++ durant la compilation.

Exercice sur les template de fonction

Puissance nième

Calculer la puissance *n* ième (*n* entier > 0) d'une valeur de type *K* (pouvant être un scalaire, une matrice...)

Puissance nième (2)

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;
```

- Tous les attributs et méthodes peuvent utiliser les paramètres templates déclarés au niveau de la classe;
- Les constructeurs sont déclarés avec le nom de la classe **sans les paramètres templates**;
- On peut définir les méthodes au sein de la déclaration de la classe mais aussi en dehors de la déclaration;
- Les méthodes définies hors classe doivent être elles-mêmes déclarées la classe portant la méthode template :

```
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 même, une structure ou une classe déclarée dans une classe template peut utiliser les paramètres de la classe template;

Instanciation d'une classe template

Règles à suivre pour l'instanciation

- Une partie du code déclarant une valeur, instanciation de la classe template, doit avoir accès à toutes les déclarations et définition de la classe et de ses méthodes !
- Lors de la déclaration d'une valeur de type la classe template :
  - On peut éviter de passer les paramètres templates pour la classe si la déclaration est faite au sein de la classe ou d'une de ses méthodes.
  - Dans ce cas, les paramètres templates sont implicitement les mêmes que celles déclarées pour la méthode ou la classe au sein où elle est déclarée;
  - Dans les autres cas : en dehors de la classe ou une de ses méthodes ou si les paramètres doivent être différents de celles déclarées pour la méthode ou la classe au sein desquelles elle est déclarée;

```
template<typename K> class A {...
  A( A const& a ) = default; // Constructeur de copie

  A& operator = ( A const& ) const = default; // Opérateur de copie
  A f( A const& a ) const;
};
template<typename K> auto A<K>::f( A const& a ) -> A { A b; ... return b; }
```

Constructeurs et méthodes templates

- Il est possible de déclarer et définir des méthodes et constructeurs templates dans une classe (qu'elle soit elle-même templatée ou non);
- Il est par contre impossible de définir des méthodes virtuelles templates !

Exemple avec classe non templatée:

```
class A { ...
  template<typename K> A( K const& valeur );
  ...
  template<typename K> K f( K const& valeur );
};
template<typename K> A::A( K const& valeur ) {...}
template<typename K> K f ( K const& valeur ) {...}
...
A a1(3.);
A a2("Valeur");
a1.f("Trois");
a2.f(4.+3.i);
```

Exemple avec classe templatée:

```
template<typename K1> class B { ...
  template<typename K2> B( K2 const& valeur );
  template<typename K2> K2 f( K2 const& valeur );
};
template<typename K1> template<typename K2>
B<K1>::B( K2 const& valeur ) { ... }

template<typename K1> template<typename K2>
K2 B<K1>::f( K2 const& valeur  ) { ... }

B<int> b1(3.); B<double> b2("Trois");
b1.template f<double>(3.); b2.f(3.+2.i);
```

Instance de template

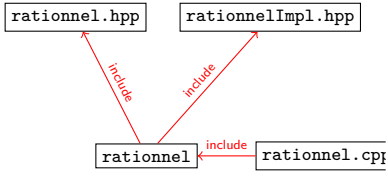
- Lors de la compilation d'un fichier de mise en œuvre (fichier `.cpp`) le compilateur C++ ne compile qu'une seule fois pour un jeux de paramêtres template donné la classe templatée;
- Pour chaque fichier de mise eu œuvre, le C++ recompile une classe templatée même si on l'a instanciée avec le même jeu de paramètre;
- Possible d'instancier explicitement une classe template :

```
#include "rationnelImpl.hpp"

template class Rationnel<std::int16_t>;
template class Rationnel<std::int32_t>;
template class Rationnel<std::int64_t>;
```

- En séparant déclaration et définition, on peut ne compiler qu'une seule fois chaque instance de la classe templatée pour un jeu de parmêtre templatée donnée :
  - `rationnel.hpp` : Déclaration de la classe template `Rationnel`;
  - `rationnelImpl.hpp` : Définition des méthodes de la classe template : inclu la classe `rationnel.hpp`;
  - `rationnel.cpp` : Instanciations de la classe template : inclu le fichier `rationnelImpl.hpp`
  - Pour instancier la classe template avec des paramêtres non instanciés : include le fichier `rationnelImpl.hpp` sinon inclure le fichier `rationnel.hpp`.

Architecture fichiers pour bien gérer les templates



Description des fichiers

- Fichier de déclaration `rationnel.hpp`

```
template<typename Int> class Rationnel { ... };
```

- Fichier de définition `rationnelImpl.hpp`

```
template<typename Int> Rationnel<Int>::Rationnel(...) {...}
```

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

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

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

```
# include "rationnel"
template Rationnel<std::int_32>;
template Rationnel<std::int_64>;
```

Exemple complet classe template : nombres rationnels

Déclaration : `rationnel.hpp`

Partie concernant les constructeurs :

```
template<typename Integer> class Rationnel {
public:
  // Constructeurs et destructeur
  Rationnel()                    = default;
  Rationnel( Integer t_numérateur, Integer t_dénominateur = 1 );
  explicit Rationnel( double t_réel, double t_epsilon = 1.E-14 );
  template<typename Integer2> Rationnel( Rationnel<Integer2> const& q );
  Rationnel( Rationnel const& ) = default;
  Rationnel( Rationnel      && ) = default;
  ~Rationnel()                   = default;
  ...
};
```

Remarque : Noter l'emploi du mot clef `explicit` pour le $3^{\text{ème}}$ constructeur, utilisé uniquement lors de la déclaration. Précise au C++ qu'il lui est **interdit de convertir implicitement** un réel en rationnel. La conversion devra être déclarée explicitement.

Exemple complet classe template : nombres rationnels (2)

Déclaration : `rationnel.hpp`

```
template<typename Integer> class Rationnel {
public:
  // Constructeurs et destructeur...
  // Opérateurs arithmétiques
  Rationnel& operator += ( Rationnel const& );
  Rationnel& operator -= ( Rationnel const& );
  Rationnel& operator *= ( Rationnel const& );
  Rationnel& operator /= ( Rationnel const& );
  Rationnel operator +   ( Rationnel const& ) const;
  Rationnel operator -   ( Rationnel const& ) const;
  Rationnel operator *   ( Rationnel const& ) const;
  Rationnel operator /   ( Rationnel const& ) const;
  Rationnel operator -   (                  ) const;
  ...
};
```

Exemple complet classe template : nombres rationnels (3)

Déclaration : `rationnel.hpp`

```
template<typename Integer> class Rationnel {
public:
  // Constructeurs et destructeur ...
  // Opérateurs arithmétiques ...
  // Autres opérateurs
  Rationnel& operator = ( Rationnel const& ) = default;
  Rationnel& operator = ( Rationnel     && ) = default;
  // Opérateur spatial (spaceship operator) pour les comparaisons (C++ 20)
  decltype(Int(1)<=>Int(2)) operator <=> ( Rationnel const& ) const;
  // Pour dire qu'on utilise l'opérateur <=> sinon faut le redéfinir
  bool operator == ( Rationnel const& ) const
  { return ((numérateur() == q.numérateur()) && (dénominateur() == q.dénominateur())); };
  explicit operator double() const { return double(m_numérateur)/m_dénominateur; }
  explicit operator std::string() const
  { return "(" + std::to_string(this->m_numérateur) + "/" + std::to_string(this->m_dénominateur) + ")"; }
  ... };
```

Remarque : L'opérateur <=> permet en une seule fonction de définir les opérarateurs de comparaison (sauf l'égalité si on ne le met pas à défaut). Le type renvoyé par l'opérateur <=> décrit l'ordonnancement entre les deux valeurs comparées.

Exemple complet classe template : nombres rationnels (4)

Déclaration : `rationnel.hpp`

```
template<typename Integer> class Rationnel {
public:
  // Constructeurs et destructeur ...
  // Opérateurs arithmétiques ...
  // Autres opérateurs ...
  // Accesseurs et modifieurs ...
  struct ValeurStruct { Integer numérateur, dénominateur; };
  ValeurStruct  valeurs() const;
private:
  ValeurStruct m_valeurs;
};
```

Remarque : On a choisit de créer une structure interne portant les deux entiers numérateur et dénominateur. Cela permet de renvoyer la paire d'entier en précisant sémantiquement leurs rôles :

```
auto vals = q.valeurs();
std::cout << "numérateur : " << vals.numérateur << std::endl;
auto [p,q] = q.valeurs();
std::cout << "numérateur : " << p << std::endl;
```

Exemple complet classe template : nombres rationnels (5)

Déclaration : `rationnel.hpp`

```
template<typename Integer> class Rationnel { ... };
// Opérateurs définis à l'extérieur de la classe
template<typename Integer> inline Rationnel<Integer> operator + (Integer val,Rationnel<Integer> const& p)
{ return p + val; }

template<typename Integer> inline Rationnel<Integer> operator - (Integer val,Rationnel<Integer> const& p)
{ return -(p - val); }

template<typename Integer> inline Rationnel<Integer> operator * (Integer val,Rationnel<Integer> const& p)
{ return p * val; }

template<typename Integer> inline Rationnel<Integer> operator / (Integer val,Rationnel<Integer> const& p)
{ return Rationnel<Integer>(val)/p; }

template<typename Integer> inline
std::ostream& operator << (std::ostream& out,Rationnel<Integer> const& p) {
    auto [pn,pd] = p.valeurs();
    out << pn << " " << pd;
    return out;                                                    }
```

Espace de nommage anonyme (suite exemple nombres rationnels)

- Déclarer dans la partie définition un espace de nommage (`namespace`) n'ayant aucun nom.
- Les fonctions et les variables "globales" définies dans cet espace de nommage ne seront visibles que pour les fonctions qui seront définies dans le même fichier (au include près);

Mise en œuvre : `rationnelImpl.hpp`

```
#include "rationnel.hpp"
...
namespace {
// Calcul du pgcd par algorithme d'Euclide
template<typename Int> Int pgcd( Int p, Int q) { ...
}
// Calcul une approximation rationnelle d'un nombre réel
template<typename Int> Rationnel<Int>
approximation_rationnelle(double x, const Rationnel<Int>& pn,
                                    const Rationnel<Int>& qn, double tol ) {
...
}
}// Fin espace de nommage anonyme
```

Exemple complet classe template : nombres rationnels (6)

Mise en œuvre : `rationnelImpl.hpp`

```
#include "rationnel.hpp"
...
template<typename Int> Rationnel<Int>::rationnel( Int num, Int dénom)
  :   m_valeurs{num, dénom}
{ ... }
template<typename Int> Rationnel<Int>::rationnel( double real, double epsilon)
  :   m_valeurs{0, 1}
{ ... }
...
template<typename Int>  decltype(Int(1)<=>Int(2))
Rationnel<Int>::operator <=> ( const rationnel& p) const {
  assert(m_valeurs.dénominateur != 0); assert(p.valeurs().dénominateur != 0);
  if ((m_valeurs.numérateur==p.valeurs().numérateur)&&(m_valeurs.dénominateur==p.valeurs().dénominateur))
    return std::strong_ordering::equal;
  if (m_valeurs.numérateur*p.valeurs().dénominateur<p.valeurs().numérateur*m_valeurs.dénominateur)
    return std::strong_ordering::less;
  assert(m_valeurs.numérateur*p.valeurs().dénominateur>p.valeurs().numérateur*m_valeurs.dénominateur);
  return std::strong_ordering::greater;
}
```

Référence universelle

Problématiques

- Considérons le code suivant :

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

Référence universelle...

Solution non satisfaisante à la problématique précédente

- 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)) {}
```

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 ou les quaternions par exemple).

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

Overview

Notion de fonctions variadiques en C

Notions

- Pour certaines fonctions, la liste des arguments peuvent ne pas être fixe.
  Exemple : la fonction `printf` en C

```
#include <cstdio>
...
printf("Hello world\n");
printf("On est le %d, du mois %d, de l'annee %d\n",jour, mois, annee);
printf("La distance entre les deux points sont : %lg\n", distance);
...
```

- Syntaxe à la déclaration/définition :
  - Passer en premier les paramètres fixes;
  - Pour les paramètres variables (variadiques), on passe simplement `...`;
- Principe:
  - À l'appel, parcourir la liste des paramètres variadiques;
  - Pour chaque paramètre parcouru, lire sa valeur en précisant le type attendu;

Notion de fonctions variadiques en C...

Macros définies dans la composante stdarg de la std

- `va_list lst_params` : Structure permettant de parcourir les $\neq$ paramêtres variadiques passées à la fonction;
- `va_start(lst_params,last_arg)` : Initialise liste paramètres var. en donnant dernier paramètre fixe dans `last_arg`;
- `val = va_arg(lst_params, type)` : Lire valeur paramètre courant. Passe ensuite paramètre suivant;
- `va_end(lst_params)` : À appeler après avoir parcouru paramètres variadiques pour détruire la structure;

Exemple code C

```
void createKeywords( std::list<std::string>> keys, ... ) {
  va_list params; va_start(params, keys);
  std::string s = va_arg(params, std::string);
  while (s != "\0") { keys.push_back(s); s = va_arg(params, std::string); }
  va_end(params);
}
...
std::list<std::string> keywords;
createKeywords(keywords, "Tintin"s, "Milou"s, "Spirou"s, "\0"s);
```

Problèmes des fonctions variadiques en C...

Problèmes posés par les fonctions variadiques en C

- On ne peut pas connaître le nombre de variables variadiques : une variable fixe doit permettre de savoir quand arrêter de lire des variables variadiques;
- Il n'y a aucun contrôle sur les types des paramètres passés !
- Donc, possibilité de faire un grand nombre de bogue avec les fonctions variadiques en C !

Exemple de bogues possibles avec l'exemple précédent

```
// Oubli de passer le "\0" à la fin : pas d'arrêt dans le parcours des templates variadiques
createKeywords(keywords, "Tintin"s, "Milou"s, "Spirou"s);
// On passe des arguments qui ne sont pas des std::string (un char * et un int)
createKeywords(keywords, "Tintin"s, "Milou", "Spirou"s, 0);
```

template variadique

Définition

- Avant C++ 11, utilisation des fonctions variadiques du C indispensables;
- Traitement des arguments variadiques résolus à l'exécution alors que les arguments variadiques étaient connus à la compilation d'un exécutable;
- C++ 11 introduit les templates variadiques;
- Permet entre autre de gérer les fonctions à nombre variable d'arguments à la compilation;
- Gestion des templates variadiques en général pensée de façon récursive;
- Fonction déployée à la compilation

Exemple

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

Comment ça marche ?

Exemple de déploiement de la fonction adder

```
adder(1,3,5,7,13)
=> { 1 + adder(3,5,7,13) }                    first = 1, args... = 3,5,7,13
=> { 1 + { 3 + adder(5,7,13)} }               first = 3, args... = 5,7,13
=> { 1 + { 3 + { 5 + adder(7,13) } } }        first = 5, args... = 7,13
=> { 1 + { 3 + { 5 + { 7 + adder(13) } } } }  first = 7, args... = 13
=> { 1 + { 3 + { 5 + { 7 + { 13 } } } } }     Appel fonction spécialisée
```

- Dans le cas où on a passé des options d'optimisation, il est quasi sûr que l'appel de adder sera réduit à la valeur finale obtenue
- Si il y a des variables passées dans la fonction adder, le C++ rendra une expression déployée sous forme de somme :

```
adder(x,y,z)
=> { x + adder(y,z) }      first = x, args... = y,z
=> { x + {y + adder(z)} }  first = y, args... = z
=> { x + { y + { z } } }   // Appel fonction spécialisée
```

Paquet de paramètres et paquet d'arguments

- `Ts` est le paquet de paramètres templates;
- `Ts...` déploie les types contenus dans `Ts`:
  Si `Ts` contient `{int, std::string, double}`, alors `Ts... args` sera remplacé pour instancier la fonction par :
  `int arg1, std::string arg2, double arg3`
  et `arg` contiendra donc à l'appel de cette fonction
  `arg1 : int, arg2 : std::string, arg3 : double;`
- `args` est le paquet de paramètres de la fonction;
- `args...` déploie les paramètres :
  Si `args` contient `{3, "tintin", 3.14}` alors `args...` sera remplacé par `3,"tintin",3.14;`
  avec les types associés (comme plus haut);
- **Exemple** :

```
template<typename... Ts> void printError(const char* fmt, Ts ... args )
{ fprintf(stderr, fmt, args... ); }
...
// Ts : int, double | args = 3, 3.1415
printError("Une erreur etrange : %d => %g", 3, 3.1415 );
```

Paquet d'arguments (déploiement général)

- En fait, les ... peuvent ne pas suivre immédiatement le paquet d'argument args (peut porter un autre nom que args);
- Par défaut, un **motif** suivi par ..., contenant au moins un paquet de paramètres est *déployé* en zéro à plusieurs instanciations du motif où le nom du paquet de paramètre est remplacé par chaque élément du paquet, dans l'ordre.
- L'expansion se fera avec une virgule ou un espace pour séparer chaque élément du paquet de paramètres;
  - Avec sizeof. Évalue à la compilation nombre d'éléments dans le paquet de paramètre : `sizeof...(args)`;
  - Utilisé comme liste d'arguments pour l'appel d'une fonction :

```
f(args...);            // Déployé comme f(arg1,arg2,...,argN);
f(n,++args...);        // Déployé comme f(n, ++arg1, ++arg2, ..., ++argN);
f(h(args...)+args...); // Déployé comme f(g(arg1)+arg1,...,g(argN)+argN)
```

  - Utilisé dans liste de paramètres servant d'initialisation entre parenthèses. Mêmes règles que appel de fonctions :

```
A a(n, args...); // Déployé comme A a(n,arg1,arg2,...,argN);
```

  - Utilisé dans une liste d'initialisation :

```
std::array<int,sizeof...(args)  > indices{args...};
std::array<int,sizeof...(args)+2> indices{-1,args...,5};
int dummy[sizeof...(Ts)] = { (std::cout << args, 0)...};
```

Exemple d'utilisation des templates variadiques

- Par défaut, les ... remplacent le **motif** contenant le paquet d'argument args par une liste de ce motif répliqué plusieurs fois, chaque membre séparé par une virgule
- Exemple:

```
template<typename HeadType, typename... Ts> std::ostream&
print(std::ostream& out, HeadType const& head, Ts const& ...args) {
  out << head;
  // Déploiement->{(out << ", " << arg1,0),(out << ", " << arg2,0),...,(out << ", " << argn),0}
  auto tmp = { (out << " " << args,0)... };
  return out; }
print("Message : ", 404, " est ", 4.04, "\n");
```

- args passé en référence constante pour éviter copie;
- Impossible d'écrire directement `std::cerr << args...;` car serait remplacé par `std::cerr << arg1,arg2,...,argn;`
- Utilise les caractéristiques de l'opérateur , : retourne dernière valeur. Exemple `auto a = (3.14,4);` ⇒ `a : int = 4` .
- Pour d'autres contextes d'emplois des paquets de paramètres, voir ▸ ce lien

Déploiement d'expressions (C++ 17)

- Déploie un paquet de paramètre en utilisant un opérateur binaire;
- Principe : remplace l'opérateur , par un autre opérateur binaire (+, *, ...);
- Permet de beaucoup simplifier l'emploie des templates variadiques;
- Syntaxe :

  (pack = {e_1 ,..., e_n} expression de paquet de paramètres, op opérateur binaire, init → I expression sans variadiques)
  - **Déploiement unitaire droit :** (pack op ...) remplacé par (e_1 op (... op (e_{n-1} op e_n)))
  - **Déploiement unitaire gauche :** (... op pack) remplacé par (((e_1 op e_2) op ...) op e_n)
  - **Déploiement binaire droit :** (pack op ... op init) remplacé par (e_1 op (... op (e_{n-1} op (e_n op I))))
  - **Déploiement binaire gauche :** (init op ... op pack) remplacé par ((((I op e_1) op e_2) op ...) op e_n)
- **Exemples** :

```
template<typename... Ts> auto adder( Ts const& ...args) { return (args + ...); }

template<typename HeadType, typename... Ts> std::ostream&
print(std::ostream& out, HeadType const& head, Ts const& ... args ) {
  out << head; ((out << " " << args),...);
  return out;
}
```

Exercices sur les fonctions templates variadiques

- Modifier la fonction `adder` (version avec déploiement d'expression) en donnant une valeur initiale à laquelle on rajoute les valeurs passées dans le paquet de paramètre;
- En s'inspirant de la fonction `adder` donné avec et sans déploiement d'expression, écrire une fonction qui concatène *n* chaîne de caractères en une chaîne, en insérant entre chaque chaîne un espace (avec et sans déploiement);
- Écrire une fonction recherchant le maximum parmi *n* valeurs de types comparables;
- Une fonction vérifiant que les valeurs passées (de types comparables) sont bien données dans l'ordre croissant. On écrira une version sans déploiement d'expression et une version avec déploiement d'expression;
- Une fonction `testAny` qui prend *n* booléens (ou valeurs assimilables à un booléen) et teste si au moins un booléen est vrai; Écrire de même une fonction `testAll` qui prend *n* booléens (ou assimilables) et teste si tous sont vrais.

Overview

Stratégies template

Politique template

- **Principe** : Changer le "comportement" d'une fonction en fonction du type passé comme paramètre template
- **Exemple simple** :

```
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::vector<K> const& buffer, int dest, int tag = MPI_ANY_TAG )
{    MPI_Send( buffer.data(), nbItems, MPI_Type<K>::id_type(), dest, tag, MPI_COMM_WORLD);    }
```

- **Exemple plus compliqué** :

```
class OpenGLRenderer { ... static void render( ... ); ... };
class VulkanRenderer { ... static void render( ... ); ... }
...
template<typename HardwareRenderer>
class SceneRenderer { ...
void display( ... ) { ... HardwareRenderer::render(...); ... }
};
```

Stratégies templates : comment restreindre les types possibles ?

- Se base sur le SFINAE (**S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror)
- **Principe** : Définir par défaut une structure template générant une erreur mais qu'on spécialise pour les types permis pour un paramètre template
- **Exemple simple** :

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

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

- **Note** : En C++ 20, cette stratégie template est devenue inutile car le C++ 20 introduit la notion de `concept`

```
#include <concepts>
std::floating_point auto distance( std::floating_point auto a1, std::floating_point auto a2,
                                   std::floating_point auto b1, std::floating_point auto b2 )
{    ...    }
// Autre façon d'utiliser un concept :
template<std::floating_point Real> Real distance( Real a1, Real a2, Real b1, Real b2 ) {    ...    }
```

Restreindre des types possibles à l'aide des concepts (C++ 20)?

- La composante concepts de la bibliothèque standard contient :
  - `std::derived_from<BaseClass>` : Restreint aux types dérivant d'une classe passée en argument template;
  - `std::convertible_to<Type>` : Restreint aux types convertibles en `Type`;
  - `std::integral` : Restreint aux entiers;
  - `std::signed_integral` : Restreint aux entiers signés;
  - `std::unsigned_integral` : Restreint aux entiers non signés;
  - `std::totally_ordered` : Restreint aux types appartenant à un ensemble totalement ordonné;
  - `std::copyable` : Restreint aux type copyable;
  - `std::invocable` : Restreint aux types foncteurs (possédant un opérateur d'évaluation);
- Possible de concevoir ses propres concepts (mais pas abordé dans ce cours);
- **Exemples** :

```
template<std::integral I> I pgcd( I a, I b ) { ... }
template<std::totally_ordered K> void sort( std::vector<K>& a ) { ... }
```

- Permet de mieux documenter sa fonction ou type template !

Introspection par template (outils nécessaires)

Techniques nécessaires

- On va utiliser le principe SFINAE et des expressions constantes (constexpr);
- Utilisation de `std::declval<K>()` et `std::decltype(expr)` :
- `std::decltype(expr)` permet de déclarer un type (pour une valeur par exemple) à partir d'une expression donnée;
- Pour récupérer type renvoyé par méthode dans une classe, créer valeur via constructeur par défaut et appeler la méthode:

```
struct Default { int foo() { return 1; } };// Struct avec constructeur par défaut...
decltype(Default().foo()) n1 = 1; //ok, n1 est un entier
```

- Que faire si classe sans constructeur par défaut ? Données nécessaires pour construire valeur ? Effets de bords possibles...

```
struct NonDefault { NonDefault(int i) {} // Struct sans constructeur par défaut
                    int foo() { return 1; } }; ...
decltype(NonDefault().foo()) n2 = 2;// Erreur, NonDefault n'a aucun constructeur par défaut
```

- `std::declval<K>()` (dans `utility` de la std) permet de "créer" virtuellement valeur de type classe et permettre d'appeler méthode dont on veut connaître type de retour :

```
#include <utility>
// 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 (principe)

Principe introspection

- On veut une structure permettant de tester à la compilation si une classe contient une méthode ou non;
- Cette structure va contenir
  - une méthode de classe templatée (pour utiliser le SFINAE) tentant d'appeler la méthode en question dans le retour et retournant `true`;
  - une méthode de classe templatée du même nom que la précédente avec signature différente (mais compatible) retournant simplement `false`;
  - Une valeur booléenne constexpr initialisée en appelant la méthode de classe avec le paramètre de la classe pour paramètre de la méthode;
- **Exemple** :

```
template<class K> struct has_serialize {
    template<typename C> static constexpr
    decltype(std::declval<C>().serialize(std::cout), bool()) test(int){ return true; }
    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

- L'instrospection ne peut se faire qu'au moment de la compilation;
- Impossible de l'utiliser dans un `if` évalué à l'exécution
- Le code ci-dessous ne pourra jamais compilé :

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

- Utiliser `if constexpr` pour évaluer la branche de compilation :

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

Exercice sur l'introspection

- 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`.

Expressions templates : problématique

Problématiques

- Imaginons que nous avons mise en œuvre un vecteur algébrique doté de la sosutraction, du produit par un scalaire et du 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 fait dans ce cas le C++ avec l'expression : `auto w = v - (v|u)*u` ?
  - $(u|v)$ range le résultat dans un scalaire α sur la pile;
  - $\alpha.u$ range le résultat dans un vecteur u' sur la pile;
  - $v - u'$ range le résultat dans un vecteur v' sur la pile;
  - $w = v'$ déplace les données de v' dans w.
- Bilan : Au moins un vecteur intermédiaire non nécessaire de créer :

```
auto scal = (v|u); for (std::size_t i=0; i<v.dim(); ++i) w[i] = v[i] - scal*u[i];
```

Comment éviter le vecteur intermédiaire ?

- Jouer avec les déplacements;
- Remarquer qu'un vecteur intermédiaire (temporaire) peut être modifié pour mettre à jour le calcul;

```
template<typename K> class Vecteur<K> { ...
template<typename K> Vecteur<K>&& operator - ( Vecteur<K>&& u ) {
  for (std::size_t i=0; i<dim(); ++i) u[i] = m_coefs[i] - u[i];
  return std::move(u);
}...
};
```

- Que fait dans ce cas le C++ avec l'expression : `auto w = v - (v|u)*u` ?
  - $(u|v)$ range le résultat dans un scalaire α sur la pile;
  - $\alpha.u$ range le résultat dans un vecteur u' sur la pile;
  - $v - u'$ stocke le résultat dans u';
  - $w = u'$ déplace les données de u' dans w.
- On voit qu'on ne crée pas le vecteur intermédiaire par rapport à l'ancienne version;
- Mieux, mais performance toujours $<$ à :

```
auto scal = (v|u); for (std::size_t i=0; i<v.dim(); ++i) w[i] = v[i] - scal*u[i];
```

Expressions templates : Idée

- **Idée** : Utiliser les templates pour construire un arbre abstrait syntaxique (principe des compilateurs);
- Technique avancée et pas toujours facile à mettre en place;
- Plusieurs bibliothèque proposent des bibliothèques d'expression templates :
  - Dans `boost-Yap`: Permet de construire soi-même des expressions templates assez "naturellement";
  - Dans `Eigen` : Les expressions mathématiques sur des vecteurs ou des matrices sont automatiquement optimisés par la bibliothèque en utilisant si possible du BLAS par derrière;
  - ...
- Nous allons voir le principe sur des exemples très simples où on peut s'affranchir de construire un arbre syntaxique.