RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

C++ Moderne
Templates et Conceptes

Xavier JUVIGNY, SN2A, DAAA, ONERA
xavier.juvigny@onera.fr

Formation C++
- 8 Septembre 2022 -

[1] ONERA, [2] DAAA

Plan du cours

Sommaire

Référence universelle

Définition

Une référence universelle représente soit une *rvalue* soit une référence.
Syntaxe : `<type>&&`
Quand est-ce qu'une variable est une r-value ou une référence universelle ?

Règles à connaître

- Si variable ou paramètre doit être déduite d'un paramètre template T comme `T&&` ou bien comme `auto&&`, alors c'est une référence universelle.
- Une référence universelle devient une référence r-value quand elle est affectée à un objet temporaire sinon elle devient une simple référence (l-value).

Attention : `std::vector<T>&&` est une r-value !
Remarque : Une valeur littérale est considérée comme temporaire.

Exemples de références universelles (ou non)

Exemples

```
Matrix&& mat1 = a_matrix; // Référence r-value
auto&&   mat2 = another_matrix; // Référence universelle (l-value)
template<typename T> void f(std::vector<T>&& u); // Référence r-value
template<typename T> void g(T&& u); // Référence-universelle
std::vector<double> tab;
auto&& a = tab[0]; // référence universelle (l-value).
auto&& b = 10;     // référence universelle (r-value)
g(10); // Le paramètre u de g devient une r-value.
int x = 10; g(x);// Paramètre de g devient une l-value
template<typename T>
class Vecteur { ...
    Vecteur( Vecteur&& rhs ); // Référence r-value
    template<typename Container> Vecteur( Container&& c ); // Référence universelle
    void setCoef(T&& x); // r-value (T non déduit)
...};
template<typename T> void f(const T&& a); // r-value !
```

Variables et r-values

À savoir

Les noms des variables de références r-values sont des l-values.

Exemples

```
void g(T&& u); // Référence-universelle
auto&& b = 10;      // référence universelle (r-value)
g(b);  // Le paramètre u de g devient une l-value
```

Forwarding

Néanmoins possible transmettre type l-value ou r-value à une variable avec `std::forward`

Exemples

```
g(std::forward(b));// Paramètre u de g devient une r-value
auto&& d = std::forward(b); // c'est une référence r-value.
```

Condition statique

Possibilité à la compilation de choisir selon une condition évaluable à la compilation de choisir de compiler une partie de code ou une autre.
Syntaxe : `if constexpr (cond) { ... } else { ... }`

Exemples

```
template<typename Integer, Integer n>
Integer factorial() {
    if constexpr (n==0)
        return Integer(1);
    else
        return n*factorial<Integer,n-1>();
}
    auto factval = factorial<long long, 10LL>();
```

Exercices sur les conditions statiques

- Écrire une fonction template prenant en argument template un entier et qui calcule la suite de Fibonacci $F_{n+2} = F_{n+1} + F_n$ avec $F_0 = 0$ et $F_1 = 1$. Ne pas utiliser de spécialisation template. Dans un premier temps écrire une fonction classique et observer les fonctions générées puis s'assurer que le calcul est bien effectué pendant la compilation !
- Écrire une fonction constexpr prenant deux entiers a et b en paramètres template. Si a divise b, alors la fonction renvoie le résultat de $\frac{a}{b}$ sous forme d'un entier, sinon renvoie le quotient et le reste sous forme d'une paire d'entier. Afficher le résultat sous forme d'un entier ou sous la forme $b \times q + r$ selon que b divise a ou non.

Notion de variadique

Définition

Qui contient un nombre arbitraire de paramètres.

Exemple

En C, on peut définir des fonctions variadiques (`printf` en est un bon exemple) :

```
double max( int nb_vals, double x0, ... ) {
    int index;
    va_list args;
    double valmax = x0;
    va_start(args, x0);
    for (index=0; index<nb_vals-1; ++index) {
        double xi = va_arg(args,double);
        valmax = (valmax < xi ? xi : valmax);
    }
    va_end(args);
    return valmax;
}

int main() {
    double val = max(6, -1., 3., 4., 2., -7., 0.);
    printf("La valeur maximale trouvee est : %lg\n", val);
    return EXIT_SUCCESS;
}
```

Templates variadiques

Principe

À partir de C++ 11, possibilité de définir un nombre arbitraire de paramètres templates à une classe ou une fonction.

Syntaxe : `template<typename ... Ts>`

Remarque : On peut définir d'autres paramètres templates (non variadiques) **avant** le paramètre template variadique !

Exemple

```
template<typename T0, typename T1>
auto max( T0 const& arg0, T1 const& arg1 ) -> decltype(arg0+arg1)
{    return (arg0>arg1 ? arg0 : arg1);    }

template<typename T0, typename ... Tn>  auto max( T0 const& arg0, Tn const&... args ) {
    auto valmax = max(args...);
    return (decltype(valmax+arg0))(arg0>valmax ? arg0 : valmax);
}

    auto valmax = max(0, 3.5, 2., -6, 4LL, 8.3);
```

Un peu de vocabulaire

Paquet de paramètres

“Paramètre” `Ts` contenant l’ensemble des paramètres du template variadique. `Ts` peut être vu comme un “conteneur” possédant *n* paramètres templates : `Ts=[T1,T2,...,Tn]`.
Syntaxe : `typename ... <Nom paquet paramètres>`

Paquet d’arguments

“Argument” `args` contenant variables `argi` type respectif `Ti`. Rajout Possible spécifications type.
Syntaxe : `<Nom paquet paramètres> [spécification types] ... <nom paquet argument>`.

Exemple de paquets d’arguments pour un paquet de paramètres `Ts`

- `Ts ... args` ≡ `args = [T1 arg1, T2 arg2, ..., Tn argn]` ;
- `Ts&... args` ≡ `args = [T1& arg1, T2& arg2, ..., Tn& argn]` ;
- `Ts&&... args` ≡ `args = [T1&& arg1, T2&& arg2, ..., Tn&& argn]` ;
- `Ts const&... args` ≡ `args = [T1 const& arg1, ..., Tn const& argn]` ;

Déploiement du paquet d'argument

Principe

Remplace le paquet d'argument par la liste des arguments le contenant, séparés par des virgules

Syntaxe : `<nom paquet arguments>...`

Ainsi `args...` ⇔ `arg1,arg2,...,argn`

Retour sur le code précédent

La ligne `auto valmax = max(args...);` est donc remplacée par le compilateur par `auto valmax = max(arg1,arg2,...,argn);`.

L'opérateur `sizeof...`

Principe

Permet de connaître la taille d'un paquet de paramètres.
Syntaxe : `sizeof...(<nom paquet paramètres>)`.

Exemple d'application

```
template<typename T0, typename ... Tn>
auto max( T0 const& arg0, Tn const&... args ) {
    if constexpr (sizeof...(args) == 0)
        return arg0;
    else {
        auto valmax = max(args...);
        return (decltype(valmax+arg0))(arg0>valmax ? arg0 : valmax);
    }
}
```

Un peu plus de généralités sur les paramètres variadiques

Quels types de paramètres templates peuvent être variadiques ?

A peu près tous les paramètres autorisés dans un template !

Exemples

```
template<std::size_t size1, std::size_t... sizes>
double gen_sqrnorms( std::array<double,size1> const& array1,
                     std::array<double,sizes> const&... arrays )
{
    double nrm = 0.;
    for ( std::size_t iCoefs=0; iCoefs<size1; ++iCoefs)
        nrm += array1[iCoefs]*array1[iCoefs];
    if constexpr (sizeof...(arrays))
        return nrm + std::move(gen_sqrnorms(arrays...));
    else
        return nrm;
}
```

Contrôle des arguments

Ce qu'on souhaite

- Avoir le même type de donnés comme paramètres ;
- Avoir un nombre pair ou impair d'arguments

Exemple homogénéisation type des paramètres

```
template<typename T, typename ... Ts>
T max( T const& val1, T const& val2, Ts const&... vals ) {
    if constexpr (sizeof...(vals) == 0)
        return std::max(val1, val2);
    else {
        T valmax = max(val2, vals...);
        return std::max(val1, valmax);
    }
}

    double dmax = max(3.5,-4.6, 6.5, 3.2, 1.7);
    std::cout << "dmax : " << dmax << std::endl;
    // Erreur de compilation ci--dessous car type paramètres inhomogènes
    //auto emax = max(4, -3, 7.5, -6, 8.9);
```

Contrôle parité nombre paramètres

Exemple forçant à avoir un nombre pair de paramètres

```
template<typename T, typename ... Ts>
auto pair_wise_addition( T const& val1, T const& val2, Ts const&... vals )
{
    if constexpr (sizeof...(vals) == 0)
        return std::make_tuple(val1+val2);
    else
    {
        auto res = pair_wise_addition(vals...);
        return std::tuple_cat(std::make_tuple(val1+val2), res);
    }
}
    auto res = pair_wise_addition( 1, 3, 4.6, -1.2, "tintin et "s, "milou"s );
    std::cout << std::get<0>(res) << ", " << std::get<1>(res) << ", " << std::get<2>(res) << std::endl;
```

Exercices simples sur les templates variadiques

- Écrire une fonction prenant *n* valeurs de type `T` en paramètre et qui renvoie le max de ces valeurs ;
- Écrire une fonction prenant une fonction générique qui sera appliquée à *n* variables de types variés. Tester en passant une fonction anonyme générique qui additionne la variable avec elle-même ainsi que plusieurs variables de types différents(entier, réel, chaîne de caractère).

Transmission parfait du type de retour dans les codes générique

Problématiques

- fonction utilisant paramètre template : on ne sait pas si elle renvoie une valeur (r-value) ou une référence
- Dans des templates récursifs, un type de retour déduit d'une expression explicite peut mener à une récurrence infinie ;

Solution apportée par le C++ 14

Possibilité de faire une transmission parfaite du type de retour et de pouvoir différer le type de retour.
Syntaxe : `decltype(auto)`

Règles de `decltype(auto)`

Règles et différences avec `auto`

On considère deux variables :

```
int i; int&& f();
```

On a alors :

- `auto x = i;` : `x` est de type `int` ;
- `decltype(auto) x = i;` : `x` est de type `int` ;
- `auto x = (i);` : `x` est de type `int` ;
- `decltype(auto) x = (i);` : `x` est de type `int&` ;
- `auto x = f();` : `x` est de type `int` ;
- `decltype(auto) x = f();` : `x` est de type `int&&` ;

Remarque : `i` et `(i)` ne signifient plus la même chose en C++ 14 !

Exemples d'utilisation de `decltype(auto)`

```
// Permet de bien prendre en compte du type exact de retour de la fonction.
template<class F, class... Args>
decltype(auto) caller(F f, Args&&... args)
{    return f(std::forward<Args>(args)...);     }
//#########################################################
// Le decltype(auto) permet de reporter la déduction du type de retour
// -> permet d'éviter une boucle de déduction infinie ayant lieu si
//    on avait utilisé decltype(enumerate(Enumeration<i-1>{}))
template<int i> struct Enumeration {};

constexpr auto enumerate(Enumeration<0>) { return Enumeration<0> };
template<int i> constexpr auto enumerate(Enumeration<i>) -> decltype(auto)
{ return enumerate(Enumeration<i-1>{}); }

int main() { decltype(enumerate(Enumeration<10>{})) a; }
```

Application des templates variadiques à des classes

Principe

Permettre à une classe/structure d'avoir un nombre paramétrables d'attributs, etc.. . .

Techniques utilisées

- Utiliser en général une structure récurrente ;
- Avec une spécialisation servant de condition d'arrêt à la récursion.
- Éventuellement, utiliser l'héritage pour la récurrence.

Exemple classique de structure avec templates variadiques

Exemple de tuple simplifié

```
template<typename T, typename... Types> struct Tuple : public Tuple<Types...> {
    T current_obj;
    Tuple( T const& current_value, Types const &... other_values)
        :   Tuple<Types...>(other_values...), current_obj(current_value) {}
    template<std::size_t n> auto const& get()
    { if constexpr (n==0) return current_obj; else  return Tuple<Types...>:: template get<n-1>(); }
};

template<typename T> struct Tuple<T> {
    T current_obj;
    Tuple( T const& current_value ) :   current_obj(current_value)    {}
    template<std::size_t n> auto const& get()
    { static_assert(n==0, "Error, n out of range"); return current_obj; }
};

int main() {
    Tuple<double, std::string,int,char> tuple(3.5, "Milou", 42, 'c');
    std::cout << tuple.get<0>() << " " << tuple.get<1>() << " "
              << tuple.get<2>() << " " << tuple.get<3>() << std::endl;
```

Exemple de classes utilisant des templates variadiques

Exemple

```
template<typename K, std::size_t dimension_n, std::size_t... dimensions> class Tenseur {
public:
    Tenseur() = default;

    K& coefs(std::size_t indexn,decltype(dimensions)... indices)
    {    return m_coefs[indexn].coefs(indices...);    }

    std::array<Tenseur<K,dimensions...>,dimension_n> m_coefs;                          };
template<typename K, std::size_t dimension> class Tenseur<K,dimension> {

    K& coefs( std::size_t index ) {   return m_coefs[index];   }

Tenseur<double,3,3,3> tenseur;

for ( std::size_t ind0=0; ind0<3; ++ind0)
    for ( std::size_t ind1=0; ind1<3; ++ind1)
        for ( std::size_t ind2=0; ind2<3; ++ind2)
            tenseur.coefs(ind0,ind1,ind2) = ind0 + 4*ind1 + 16*ind2;
```

Exercice sur les templates variadiques

Toutes les fonctions devront pouvoir être interprétées à la compilation (constexpr).

Gestion d'une liste statique

Le but ici est de gérer une liste d'entier durant la compilation.

- Définir liste statique comme une structure template vide ayant pour paramètres templates une liste d'entiers (on spécialisera pour une liste ne contenant qu'un seul entier).
- Écrire une fonction permettant d'afficher la liste statique.
- Définir deux fonction permettant de rajouter un élément soit en première position (pushFront) soit en dernière (pushBack) ;
- Définir une fonction renvoyant un booléen : vrai si la liste est triée croissante, faux sinon.
- Définir une fonction triant la liste (On utilisera un tri à bulle) ;
- Définir une fonction permettant de trier une liste tout en enlevant ses doublons.

Règle pour le déploiement d'un paquet de variable

Principe

Par défaut, le déploiement d'un paquet `as` *equiv* `a1,a2,...,an` de types `Ts` $\equiv$ `T1,T2,...,Tn` suit la règle suivante :

`expression(as)` se déploie comme `expression(a1), expression(a2), ..., expression(an)`

Quelques exemples

- `Ts const&... as` $\Rightarrow$ `T1 const& a1, ..., Tn const& an`
- `std::vector<Ts> const&...as` $\Rightarrow$
  `std::vector<T1> const& a1, ...,std::vector<Tn> const& an`

Produit cartésien de deux ensembles hétérogènes

Exercice

L'idée est d'avoir en entrée deux tuples et de construire un tuple de tuple contenant des pairs constituant le produit cartésien des deux tuples.
Exemple : `{3,'x'}` $\times$ `{0.5,1}` $\Rightarrow$ `{{(3,0.5),(3,1)},{('x',0.5),(`x',1)}}`
Pour l'exercice, il faut utiliser :

- `std::index_sequence<entiers...>` : Proche de la liste statique de l'exercice précédent. Contient au travers de paramètres templates n entiers. Possible de passer un paquet de paramètres entiers.
- `std::index_sequence_for<Paramètres templates...>` : Génère pour n paramètres une séquence d'indices de type `std::index_sequence` allant de 0 à $n-1$. Accepte les paquets de paramètres.

Voir la documentation de Microsoft

Produit cartésien de deux ensembles hétérogènes (suite...)

Les étapes :

- Écrire une fonction permettant d'afficher une paire de valeurs (utilisant `std::pair`) ;
- Écrire deux fonctions : une fonction printTupleImpl qui affiche les valeurs d'un tuple en suivant une séquence d'index et un opérateur affichant un tuple en utilisant la fonction précédente ;
- Écrire une fonction permettant le produit "cartésien" entre un objet de type T et un tuple en utilisant une séquence d'entiers ;
- Écrire une fonction permettant le produit "cartésien" d'un tuple P_1 utilisant une séquence d'entiers avec un tuple P_2 ;
- Écrire une fonction effectuant le produit "cartésien" de deux tuples.

Généralisation du déploiement des paquets d'argument (C++ 17)

Principe

Remplacer la virgule générée par le déploiement par un autre symbole.

Syntaxes

`op` est le symbole devant remplacer la virgule, `init` une valeur initiale.

- `(paquet ... op)` $\Rightarrow p_0$ op $(\ldots$ op $(p_{n-1}$ op $p_n)\ldots)$;
- `(... op paquet)` $\Rightarrow (\ldots(p_1$ op $p_2)$ op $\ldots)$ op $p_n)$;
- `(paquet op ... op init)` $\Rightarrow (p_1$ op $(\ldots$ op $(p_{n-1}$ op $(p_n$ op init $)\ldots)$;
- `(init op ... op paquet)` $\Rightarrow (\ldots($init op $p_1)$ op $p_2)$ op $\ldots)$ op $p_n)$.

Exemples de déploiement généralisé (C++ 17)

Quelques exemples

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

template<typename ... Ts> void cppPrintf( Ts const&... values )
{   (std::cout << ... << values );   }

template<typename Func, typename ... Ts>
auto fadd( double initValue, Func&& func, Ts const&... values )
{   return (func(values)+...+initValue);    }
```

Variables templates (C++ 14)

Principe

Possibilité de définir directement des patrons de variables.

Exemple

```
template<typename K, std::uint32_t n> constexpr K fact = n*fact<K,n-1>;
template<typename K> constexpr K fact<K,0> = K(1);

int main() {
    printf("10! = %12.6lf\n", fact<double,10>);
```

Paramètres templates implicites (C++ 17)

Principe

Passer des paramètres non typés dont on déduit le paramètre template

Exemple

```
template<auto value1, auto... values> struct static_tuple : public static_tuple<values...> {
    template<auto index> auto get() const {
        if constexpr (index==0) return value1;
        else return static_tuple<values...>::template get<index-1>();
    }
};
template<auto value> struct static_tuple<value> {
    template<auto index> auto get() const {
        static_assert(index==0, "Out of bound index value");
        return value;
    }
};

    static_tuple<3, 7LL, 3.14> stuple;
    printf("stuple : (%d, %lld, %lf)\n", stuple.get<0>(), stuple.get<1U>(), stuple.get<2ULL>());
```

Exercices

- Écrire une variable template `constante1` prenant en paramètres templates le type et la valeur de la constante.
- Écrire une variable template `constante2` prenant un paramètre template auto pour la valeur de la constante.
- Définir sous forme de variable la suite de Heron : $u_{n+1} = 0.5.\left(u_n + \frac{A}{u_n}\right)$ où A est la valeur dont on veut calculer la racine carrée.

Remarque : clang ne supporte pas encore des valeurs en paramètre template non entières (version 15 comprise) !

Déduction automatique des paramètres templates (C++ 17)

Principe

- Généralise la déduction des paramètes templates des fonctions templates aux classes et structures.
- On peut définir ses propres règles pour la déduction de type à partir des constructeurs ;
- Déjà fait pour les classes de la STL.

Exemple d'utilisation pour des classes de la STL

```
[[maybe_unused]] std::vector u(5,0.);
[[maybe_unused]] std::vector v{1.,2.,3.,4.};

[[maybe_unused]] std::list lu(10,0.);
[[maybe_unused]] std::list lv{2.,3.,5.,7.,11.};

[[maybe_unused]] std::map dico{ std::pair{"Mercure",88.}, {"Venus", 224.7},
                                {"Terre"  ,365.256}, {"Mars" , 687.} };
```

Définir ses propres règles de déduction (C++ 17)

Principe

Pour certains constructeurs d'une classe, indiquer au compilateur comment déduire ses paramètres templates à partir des paramètres du constructeur. Ainsi pour une classe `A` :

Syntaxe :

```
template<paramètres...> A(arguments contenant paramètres) -> A<paramètres...>;
```

Exemple

```
template<typename K> struct Matrix {
    Matrix( std::array<std::uint32_t,2> const& t_dimensions, K const& t_initValues );
    Matrix( std::array<std::uint32_t,2> const& t_dimensions, std::vector<K> const& t_values );
    ~Matrix() = default;
    std::vector<K> m_coefs;
};
template<typename K>
Matrix(std::array<std::uint32_t,2> const& t_dimensions,K const& t_initValues) -> Matrix<K>;
template<typename K>
Matrix(std::array<std::uint32_t,2> const& t_dimensions,std::vector<K> const& t_values ) -> Matrix<K>;
```

Paramètres templates pour les fonctions anonymes

Intérêts

- Mieux contrôler les paramètres génériques que `auto` ;
- Créer des fonctions anonymes génériques ne possédant pas d'arguments paramétrés.

Exemple de contrôle des paramètres génériques

```
template<typename T> auto    tpl_adder = [] ( T const& val1, T const& val2 ) -> T
{   return val1+val2;    };
auto auto_adder= [] ( auto val1, decltype(val1) const& val2 ) -> decltype(val1)
{   return val1+val2;    };
auto abizzare= auto_adder(3, 5.2);// Addition d'entiers !
auto tbizzare= tpl_adder<double>(3, 5.2);// Addition de réels
```

Exemple de fonction anonyme avec paramètre template

Exemple d'adaptateur de construction

```
template<typename K> struct Matrix {
    Matrix( std::uint32_t nbRows, std::uint32_t nbCols )
    {
        std::cout << "Construit matrice (" << nbRows << "x" << nbCols
                  << ") containing " << typeid(K).name() << std::endl;
    }
};
template<typename K> auto matrixBuilderAdaptater = [] ( std::array<uint32_t,2> const& dimensions )
{    return Matrix<K>( dimensions[0], dimensions[1] );    };
auto matBuilder = matrixBuilderAdaptater<double>;
matBuilder({ 5, 8 });
```

Sommaire

Introduction aux concepts

Idées principales

- Pouvoir utiliser la simplicité des paramètres auto pour des fonctions classiques ;
- Mais rajouter des contraintes pour contrôler facilement le type de paramètres autorisés !

Comment ça marche ?

- Utiliser `auto` pour spécifier le type d'un paramètre dans une fonction ;
- Rajouter un attribut (un concept) permettant de mettre des contraintes sur les paramètres ;
- Ce concept peut également être utilisé comme paramètre template classique.

Quelques exemples de concepts

De nombreux concepts sont déjà proposés par la librairie standard `concepts`.

Exemples

```
template<std::derived_from<std::ostream> OS>
OS& repr(OS& output, auto const& object)
{ output << "<" << type_name<decltype(object)>()
         << ">{" << object << "}"; return output; }

auto maFonction( std::floating_point auto const& x )
{ return x*x/3 - x/7 + 3; }

template<typename T, typename ContainerT>
    requires std::same_as<T,typename ContainerT::value_type>
void add( T const& value, ContainerT& container )
{   for (auto& coef : container ) coef += value;    }
```

Exercices utilisant les concepts

Pour les exercices, aller voir la référence à la librairie concepts.

- Écrire une fonction calculant la valeur maximale de deux valeurs partageant un type commun et comparables et retourner le résultat avec le type commun ;
- Écrire une fonction calculant la valeur réelle d'un quotient en imposant que le numérateur doit être un entier et le dénominateur un entier non signé.

Définir ses propres concepts

Principe

- Utilisation des templates et des mots clefs `concept` et `requires` ;
- Peut utiliser les paramètres templates ou toute variable visible au niveau de la définition ;
- Les expressions données dans le concept doivent être compilables pour un type donné.

Exemple simple

```
template<typename K> concept Sommable = requires( T a, T b ) {
        a+b; // L'expression a+b doit être valide à la compilation...
};
```

Remarques

Les expressions dans le concept doivent être syntaxiquement corrects.

Concept composé

Possibilité de mettre plusieurs contraintes dans un concept

Exemples simples

```
template<typename K> concept AnneauStructure = requires(K a, K b) {
    {a+b} -> std::convertible_to<K>;
    {a-b} -> std::convertible_to<K>;
    {a*b} -> std::convertible_to<K>;
    {-a } -> std::convertible_to<K>;
};
```

Remarques

- Déclaration retour attendu d'une expression $\Rightarrow$ entourer l'expression par deux accolades ;
- Pas possible déclarer un type comme retour d'expression. Il faut passer par un concept !

Combinaison logique de concepts

Possibilité de combiner des concepts pour adapter la contrainte ou créer un nouveau concept.

Exemple

```
template<typename Container> concept Iterable = requires(Container c) {
    {c.begin()} -> std::convertible_to<typename Container::const_iterator>;
    {c.end()  } -> std::convertible_to<typename Container::const_iterator>; };

template<typename K> requires std::floating_point<K> ||
                            ( std::invocable<K> &&
                              std::floating_point<typename std::invoke_result<K>::type>)
void fill_container( Iterable auto& container, K&& value ) {
    if constexpr (std::invocable<K>) for (auto& coef : container ) coef = value();
    else for (auto& coef : container ) coef = value;       }

fill_container(tableau1, 3.14);
fill_container(tableau2, [counter=0]() mutable { ++counter; return 0.5*counter; } );
```

Exemple de concept avec combinaison logique

Exemple

```
template<typename K,typename... ArgsType> concept ReelPeutEtreProcedural = std::floating_point<K> ||
      (std::regular_invocable<K,ArgsType...> &&
       std::floating_point<typename std::invoke_result<K,ArgsType...>::type>);

template<typename... ArgsType>
void fill_container_v2( Iterable auto& container, ReelPeutEtreProcedural<ArgsType...> auto&& value,
                        ArgsType const& ... args )
{
    if constexpr (std::invocable<decltype(value),decltype(args)...>)
        for (auto& coef : container ) coef = value(args...);
    else
        for (auto& coef : container ) coef = value;
}
fill_container_v2(tableau2, [counter=0](int m) mutable { ++counter; return double(counter*m); },
                  rand()%257 );
```

Exercice sur les concepts

- Définir un concept définissant un corps (au sens algébrique) : On reprend le concept d'anneau et on vérifie de plus si il existe un opérateur division (`/`) ou une méthode `inverse` associé au type ;
- Tester le concept en écrivant une fonction générique résolvant une équation linéaire ;
- Tester la fonction avec des réels, des entiers et des fractions rationnelles (dont les sources sont données).