Chapitre 8: Généricité

Introduction

Introduction

Par ma foi ! il y a plus de six mois que j'utilise la généricité sans que j'en susse rien, et je vous suis le plus obligé du monde de m'avoir appris cela.

Monsieur J., étudiant INF2

Introduction

Le concept de généricité ne vous est pas inconnu. Vous l'avez abordé en INF1 sous divers aspects

- La surcharge de fonctions qui permet de définir la même fonction pour des paramètres différents
- Les classes `string`, `wstring`, `u16string`, `u32string`, qui définissent similairement des chaines de caractères différentes (`char`, `wchar_t`, `char16_t`, `char32_t`)
- Les classes `vector` et `array` qui peuvent contenir divers types de données
- les fonctions d'`<algorithm>` qui s'appliquent à des conteneurs variés

Rappel – surcharge de fonction

- Plusieurs fonctions peuvent partager le même nom à condition que leurs profils – le nombre et l'ordre des types des paramètres – permettent au compilateur de déterminer laquelle appeler

- Mais si le code de ces fonctions est identique au type près, beaucoup de code se trouve dupliqué

```
int somme (int a, int b) {
    return a + b;
}

double somme (double a, double b) {
    return a + b;
}

int main() {
    cout << somme(10,20) << endl;
    cout << somme(1.0,1.5) << endl;
}
```

Exemple

- Pour éviter cette duplication, la fonction `somme` sera plutôt écrite de manière générique, valable pour tout type `T`

```
template <typename T>
T somme(T a, T b) {
    return a + b;
}
```

- Le code client devient

```
int main() {
    cout << somme<int>(10,20) << endl;
    cout << somme<double>(1.0,1.5) << endl;
}
```

mais on verra que l'on peut le simplifier

Généricité – principe général

- La syntaxe générale pour rendre une déclaration générique est

```
template < liste_de_paramètres > déclaration
```

- Ce qui permet de déclarer génériquement
  - une famille de fonctions, y compris des fonctions membres de classes
  - une famille de classes, y compris des classes imbriquées
  - un alias à une famille de types (C++11)
  - une famille de variables (C++14)

Fonctions génériques

Déclaration / définition

- La déclaration (définition) d'une fonction générique est précédée du mot réservé `template`, suivi des paramètres génériques formels placés entre `<>`, typiquement chacun précédé du mot réservé `typename`
- Les noms de types de la liste des paramètres génériques peuvent être utilisés comme tout autre type, i.e. dans la déclaration (paramètres ou type de retour) ou dans le corps de la définition (variables locales, cast, …)

```
// déclaration
template <typename T> void echanger(T& v1, T& v2);

// définition
template <typename T> void echanger(T& v1, T& v2) {
  T temp = v1; v1 = v2; v2 = temp;
}
```

typename / class

- Notons que pour des raisons de rétrocompatibilité, on peut aussi écrire `class` à la place de `typename`

```
// déclaration
template <class T> void echanger(T& v1, T& v2);

// définition
template <class T> void echanger(T& v1, T& v2) {
  T temp = v1; v1 = v2; v2 = temp;
}
```

mais c'est une pratique déconseillée

Instanciation

- La définition d'une fonction générique ne définit pas réellement une fonction, mais juste un moule devant être instancié. Compiler un fichier ne contenant que cette définition ne générerait aucun code.

- Pour que le compilateur génère du code, il faut instancier la fonction générique avec des types effectifs. Cela peut se faire explicitement

```
template void echanger<int> (int&, int&);
template void echanger<char> (char&, char&);
```

Instanciation

- L'instanciation peut aussi se faire implicitement, en appelant la fonction

```
int main()
{
  int a = 0, b = 1;
  echanger<int>(a,b); // instanciation et appel de
                      // echanger<int>(int&,int&)
}
```

Compilation séparée

La définition d'une fonction générique ne définissant pas réellement une fonction mais un moule, on a les possibilités suivantes en terme de compilation séparée

- Placer la définition dans un fichier header sans fichier .cpp correspondant. Tout code qui inclut ce header peut l'instancier – typiquement implicitement – avec tous types d'argument. Il n'est pas nécessaire de déclarer explicitement la fonction, la définir suffit
- Placer la déclaration dans un fichier header et la définition dans un fichier .cpp, accompagnée des instanciations explicites de tous les types qui seront utilisables. Aucun autre type ne sera utilisable.
- Il était auparavant possible d'utiliser le mot-clé `export` pour séparer déclaration et définition sans devoir instancier explicitement tous les types, mais cette possibilité, mal supportée par les compilateurs, a disparu avec C++11.

extern

- L'instanciation implicite des templates rend la tâche du compilateur complexe dans un cadre de compilation séparée.
- Chaque instanciation implicite dans un fichier .cpp entraine la génération de code dans le fichier objet correspondant. Ce code, éventuellement dupliqué, doit ensuite être nettoyé par l'éditeur de liens.
- Pour simplifier et accélérer la compilation, il est possible d'indiquer au compilateur de ne pas instancier implicitement une instance dont on sait qu'elle est instanciée explicitement ailleurs. Pour cela, on utilise la syntaxe de l'instanciation explicite, précédée du mot clé `extern`. Par exemple…

```
extern template void echanger<int> (int&, int&);
extern template void echanger<char> (char&, char&);
```

Paramètres multiples

- Il peut y avoir plusieurs paramètres génériques, séparés dans la liste par des virgules

```
template <typename T, typename U>
void f(T v1, U v2) {
  ...
}
```

- Cette fonction s'instancie en spécifiant les types effectifs souhaités séparés par des virgules, par exemple explicitement

```
template void f<int, double>(int, double);
```

Déduction d'arguments

Pour les fonctions génériques (pas pour les classes), il n'est pas nécessaire de spécifier les types effectifs souhaités si ceux-ci peuvent êtres déduits du contexte. La fonction générique

```
template <typename T> void echanger(T& v1, T& v2);
```

peut également être instanciée explicitement ainsi

```
template void echanger<>(int&, int&); // <int> déduit
template void echanger(char&, char&); // <char> déduit
```

ou implicitement comme suit

```
int a = 0, b = 1;
echanger<>(a,b);
echanger(a,b); // instanciation et appel de
               // echanger<int>(int&,int&)
```

Déduction d'arguments

- Cette déduction n'est pas possible pour les arguments dont le type n'est pas utilisé comme paramètre de la fonction. Tout argument non déductible doit être spécifié explicitement dans l'instanciation

```
template <typename To, typename From> To convert(From val) {
  return (To) val;
}
int main () {
  double d = 0.5;
  int i = convert<int>(d);  // convert<int,double>(double);
  int i = convert<>(d);     // ne compile pas
  ...
```

- Dans l'exemple ci-dessus, seul le type `From` peut être déduit; le type `To`, lui, doit être fixé explicitement.

- Si l'ordre des types génériques avait été inversé dans notre exemple, il aurait fallu fixer explicitement à la fois `To` et `From` et écrire

```
int i = convert<double,int>(d);
```

Déduction d'arguments

Attention, avec la déduction des arguments génériques, c'est toujours le type exact qui est passé. Il n'y a donc pas de conversion implicite possible pour les arguments de la fonction

```
template <typename T> void f(T v1, T v2) { … }

int main() {
  int i1, i2; double d1, d2;
  f(i1,i2);           // f<int>(int,int)
  f(d1,d2);           // f<double>(double,double)
  f(i1,d1);           // erreur de compilation
  f<int>(i1,d1);      // f<int>(int,int) avec conversion
                      // de d1 en int
  f<double>(i1,d1);   // f<double>(double,double) avec
                      // conversion de i1 en double
```

Valeurs par défaut

- Enfin, on peut spécifier des valeurs par défaut pour les paramètres génériques

```
template <typename To = int, typename From = int>
To convert(From val) {
  return (To) val;
}
```

- Mais cette possibilité est peut utilisée pour les fonctions génériques car la déduction d'arguments prime sur cette valeur par défaut

```
double d;
convert(d);              // convert<int,double>
convert<int>(d);         // convert<int,double>
convert<int, int>(d);    // convert<int,int>
```

Spécialisation

Il est possible de redéfinir spécifiquement une fonction générique pour un argument générique donné en utilisant `template<>` suivi de la fonction où tous les types sont spécifiés

```
template<typename T> bool estDeTypeInt(T t) {
  return false;
}

template<> bool estDeTypeInt(int i) {
  return true;
}

int main () {
  cout << boolalpha
       << estDeTypeInt(1) << ' '          // true
       << estDeTypeInt('a') << ' '        // false
       << estDeTypeInt(1.) << endl ;      // false
```

Spécialisation

- Notons que, contrairement à ce que nous verrons plus tard pour les classes, il n'est pas possible de spécialiser partiellement une fonction à plusieurs arguments génériques

Spécialisation

Il est également possible de traiter différemment les références des références constantes, ou les valeurs des pointeurs. Mais il s'agit plutôt de surcharge que de spécialisation

```
template<typename T> bool estModifiable(const T& t)
{ return false; }

template<typename T> bool estModifiable(T& t)
{ return true; }

int main () {
  double d = 0.5;
  const int I = 1;
  estModifiable(I);        // false
  estModifiable(d);        // true
  estModifiable(0.5);      // false
```

Paramètres génériques non types

- Comme nous l'avons déjà vu avec la classe `std::array`, il est aussi possible d'avoir des paramètres génériques qui ne sont pas des types. Par exemple

```
template <int N> void incr(int& i) {
    i += N;
}

int main () {
  int i = 1;
  incr<10>(i);  // i vaut 11;
  …
```

- Attention, « `int N` » ressemble à une variable, mais il s'agit d'une valeur constante qui est déterminée à la compilation, pas à l'exécution

Paramètres génériques non types

Le type de ces paramètres peut être

- Un type intégral (`bool`, `char` … `int` … `unsigned long long`)
- Un type énuméré
- Un pointeur vers ou une référence à
  - une fonction
  - un objet alloué statiquement
  - un membre statique (objet ou fonction) d'une classe.

Dans tous les cas il faut que la valeur puisse en être déterminée à la compilation

Paramètres génériques non types

- Cela nous permet d'écrire une version assez particulière de « Hello, World! »

```
template <std::string & temp> void g() {
    temp += "World!";
}

std::string s; // variable globale dont on
               // peut déterminer une référence
               // à la compilation

int main ()
{
    s = "Hello, ";
    g<s>();
    cout << s << endl;
}
```

Classes génériques

Déclaration

- La déclaration d'une classe générique suit la syntaxe

```
template < liste_de_paramètres > déclaration
```

- Par exemple, pour rendre générique la classe CVector du chapitre précédent, il suffit de la réécrire ainsi

```
class CVector
{
  double x,y;
public:
  CVector() {};
  CVector(double a, double b)
    : x(a), y(b) {}
};
```

```
template <typename T>
class CVector
{
  T x, y;
public:
  CVector() {}
  CVector(T a, T b)
    : x(a), y(b) {}
};
```

Instanciation

- La classe générique doit être instanciée pour que le compilateur génère son code. Elle peut être instanciée explicitement

```
template class CVector<char>;
```

- Elle peut aussi être instanciée implicitement en l'utilisant dans le code

```
CVector<int> v(1,2);
CVector<double> w(1,2);
```

- Dans tous les cas il faut indiquer les paramètres génériques effectifs explicitement entre <>. Il n'y a jamais de déduction d'arguments pour les classes

- Enfin, comme pour les fonctions, on peut empêcher l'instanciation implicite avec le mot clé `extern`

```
extern template class CVector<int>;
```

Définition

- Pour pouvoir instancier une classe générique, il faut évidemment que ses fonctions membres soient définies. On peut le faire en ligne

```
template <typename T> class CVector {
  …
  T produitScalaire(const CVector<T>& cv) const {
        return x * cv.x + y * cv.y;
  }
  …
};
```

- Mais il est plus propre de séparer déclaration et définition pour les fonctions membres non triviales.

Définition séparée

- On déclare la méthode dans la déclaration de la classe

```
template <typename T> class CVector {
  …
  T produitScalaire(const CVector<T>& cv) const;
  …
};
```

- Pour la définir en dehors de cette déclaration, il faut préciser la classe à laquelle elle appartient avec la syntaxe suivante

```
template <typename T>
T CVector<T>::produitScalaire(const CVector<T>& cv) const {
    return x * cv.x + y * cv.y;
}
```

Opérateurs

- Ces considérations s'appliquent également aux opérateurs qui sont des méthodes comme les autres

```
template <typename T> class CVector {
  …
  CVector<T> operator + (const CVector<T>& cv) const;
  …
};
```

```
template <typename T>
CVector<T> CVector<T>::operator + (const CVector<T>& cv) const
{
    CVector<T> temp;
    temp.x = x + cv.x;
    temp.y = y + cv.y;
    return temp;
}
```

Compilation séparée

- Comme pour les fonctions génériques, la définition d'une classe générique ne génère de code que si elle est instanciée.
- Elle ne peut donc pas être compilée telle quelle pour donner du code objet.
- La définition d'une classe générique doit plutôt être vue comme une déclaration, qu'il faut entièrement inclure dans un fichier header.

CVector.h

```
#ifndef CVECTOR_H
#define CVECTOR_H

template <typename T>
class CVector {
  T x, y;
public:
  CVector() {}
  CVector(T a, T b): x(a), y(b) {}
  T produitScalaire
    (const CVector<T>& cv) const;
  CVector<T> operator +
    (const CVector<T>& cv) const;
};

template <typename T>
T CVector<T>::produitScalaire
(const CVector<T>& cv) const {
  return x * cv.x + y * cv.y;
}

template <typename T>
CVector<T> CVector<T>::operator +
(const CVector<T>& cv) const {
  CVector<T> temp;
  temp.x = x + cv.x;
  temp.y = y + cv.y;
  return temp;
}
#endif
```

Compilation séparée

- Une solution couramment utilisée consiste à couper ce fichier header en deux pour séparer déclaration et définitions, l'un incluant l'autre

CVector.h

```
#ifndef CVECTOR_H
#define CVECTOR_H

template <typename T>
class CVector {
  T x, y;
public:
  CVector<T>() {}
  CVector<T>(T a, T b)
    : x(a), y(b) {}
  T produitScalaire
    (const Cvector<T>& cv) const;
  CVector<T> operator +
    (const CVector<T>& cv) const;
};

#include "CVectorImpl.h"
#endif
```

CVectorImpl.h

```
#ifndef CVECTORIMPL_H
#define CVECTORIMPL_H

template <typename T>
T CVector<T>::produitScalaire
(const CVector<T>& cv) const {
    return x * cv.x + y * cv.y;
}

template <typename T>
CVector<T> CVector<T>::operator +
(const CVector<T>& cv) const {
    CVector<T> temp;
    temp.x = x + cv.x;
    temp.y = y + cv.y;
    return temp;
}
#endif
```

Méthodes génériques

- Comme les autres fonctions, les fonctions membres d'une classe peuvent également être génériques. Par ex.

```
template <typename T> class CVector {
  …
  template <typename U> CVector<U> convert();  …
};
```

- Attention à choisir des noms différents pour les paramètres génériques de la classe et de la méthode.

- La définition s'écrit alors

```
template <typename T> template <typename U>
CVector<U> CVector<T>::convert() {
    CVector<U> u( (U)x, (U)y );
    return u;
}
```

Fonctions amies génériques

- Pour pouvoir afficher nos objets de type CVector<T>, il faut surcharger `operator` << de manière générique

```
// déclaration avancée de CVector
template <typename T> class CVector;

// déclaration + définition d'operator <<
template <typename T>
ostream& operator << (ostream& os, const CVector<T>& cv) {
  os << cv.x << ' ' << cv.y;
  return os;
}

template <typename T> class CVector {
  // amitié entre CVector<T> et l'opérateur << générique avec
  // le paramètre générique effectif T
 friend ostream& operator << <T>(ostream& os, const CVector<T>& cv);
  …
}
```

Fonctions amies génériques

- Notons que les déclarations d'amitié peuvent s'écrire de plusieurs manières

```
friend ostream& operator << <T>(ostream& os, const CVector<T>& cv);
friend ostream& operator << <T>(ostream& os, const CVector& cv);
friend ostream& operator << <>(ostream& os, const CVector<T>& cv);
friend ostream& operator << <>(ostream& os, const CVector& cv);
```

- Mais oublier <…> avant la parenthèse ne compile pas.

Spécialisation

- On peut spécialiser certaines méthodes ou au contraire toute la classe générique en la définissant une nouvelle fois en spécifiant les arguments effectifs.
- Par ex., spécialiser le produit scalaire pour le type bool

```
template <typename T>   // cas général
T CVector<T>::produitScalaire(const CVector<T>& cv) const
{
    return x * cv.x + y * cv.y;
}
```

```
template <>             // spécialisation
bool CVector<bool>::produitScalaire(const CVector& cv) const
{
    return (x and cv.x) or (y and cv.y);
}
```

Spécialisation

- On peut également spécialiser toute la classe générique.

```
template <> class CVector<bool> {
  // réécriture de toute la classe pour
  // le type bool
};
```

- Cela nécessite de réécrire l'entièreté de la classe pour un argument particulier.

Spécialisation partielle

- Pour les classes à plusieurs paramètres génériques, il est possible de ne les spécialiser que partiellement. Il reste des paramètres génériques…

```
template<class T1, class T2, int I>
class A {};                     // template primaire

template<class T, int I>
class A<T, T*, I> {};           // #1: spécialisation partielle.
                                // T2 est un pointeur vers T1
template<class T, class T2, int I>
class A<T*, T2, I> {};          // #2: spécialisation partielle.
                                // T1 est un pointeur
template<class T>
class A<int, T*, 5> {};         // #3: spécialisation partielle.
                                // T1 est entier int, I vaut 5,
                                // et T2 est un pointeur
template<class X, class T, int I>
class A<X, T*, I> {};           // #4: spécialisation partielle.
                                // T2 est un pointeur
```

Paramètres template template

- Considérons le problème suivant. Vous disposez de conteneurs génériques `Liste` et `Tableau`

```
template <typename T> class Liste { … };
template <typename T> class Tableau { … };
```

- Vous utilisez `Tableau` pour mettre en œuvre un troisième conteneur – `Pile`

```
template <typename T> class Pile {
  Tableau<T> data;
  …
};
```

- Mais vous voudriez une solution plus générique qui permette d'avoir `Liste` ou `Tableau` en paramètre générique de `Pile`

Paramètres template template

- Une solution – celle choisie par la STL – consiste à ajouter un paramètre générique pour le type du conteneur.

```
template <typename T, typename CONTENEUR> class Pile {
  CONTENEUR data;
  …
};
```

- Pour utiliser cette classe, on doit déclarer les piles ainsi

```
Pile<int, Tableau<int>> p1;
Pile<double, Liste<double>> p2;
```

- Ce qui n'est pas entièrement satisfaisant. On aimerait pouvoir écrire

```
Pile<int, Tableau> p1;
Pile<double, Liste> p2;
```

Paramètres template template

- Pour cela, il faut modifier la déclaration de Pile en utilisant un nouveau type de paramètre générique – le template template

```
template <typename T, template<typename> class CONTENEUR>
class Pile {
  CONTENEUR<T> data;
  …
};
```

- Au prix d'une déclaration plus complexe de Pile
  - son utilisation est simplifiée
  - le risque de se tromper en mélangeant les types (e.g. `Pile<double, Tableau<int>> p1;`) disparaît
- Attention, c'est le seul cas où il faut utiliser le mot-clé `class` et pas `typename`. Cette restriction disparaîtra avec C++17

Alias de types génériques (C++11)

Alias

- C++11 introduit les alias de types génériques. Ils s'écrivent

```
template < template-parameter-list >
using identifier = type-id ;
```

- Cela permet par exemple d'écrire

```
template <typename T> using Tab10 = array<T,10>;
template <typename T> using ptr = T*;
```

Que l'on utilise ainsi

```
ptr<int> pi;            // int* pi;
Tab10<double> tab;      // array<double,10> tab;
```

Alias

- On est évidemment tenté de simplifier des types complexes comme

```
template <typename T>
using rIter = vector<T>::reverse_iterator;
```

mais … `Error: Missing 'typename' prior to dependent type name 'vector<T>::reverse_iterator'`

- Le compilateur n'est pas capable de déterminer que `reverse_iterator` ci-dessus est un type et pas une variable. Il faut le lui indiquer avec le mot-clé `typename`

```
template <typename T>
using rIter = typename vector<T>::reverse_iterator;
```

Variables génériques (C++14)

Variables génériques

- C++ 14 introduit la possibilité de déclarer des variables génériques. L'exemple classique est

```
template <typename T>
const T pi = T(3.14159265358979323385);
```

- On instancie et utilise par exemple cette variable ainsi

```
template <typename T>
T circular_area(T r) {
   return pi<T> * r * r;
                // instanciation implicite de pi<T>
}
```

- Cela permet d'éviter des conversions de types inutiles si nous avions défini pi de type `double` par exemple

Variables génériques

- Avant C++14, la technique classique pour obtenir le même effet est celle utilisée par la librairie boost.
- `boost/math/constants/constants.hpp` définit des fonctions génériques telles que

```
namespace boost{ namespace math{ namespace constants{
  template <class T> T pi();
  ...
}}}
```

- Qui s'utilisent par exemple ainsi

```
template<class T>
T circular_area(T r) {
    return boost::math::constants::pi<T>() * r * r;
}
```

Concepts (STL)

Contrat implicite

- Ecrire une fonction générique telle que

```
template <typename T> T square(T a) {
  return a * a;
}
```

```
Invalid operands to binary
expression ('const char *' and
'const char *')
```

implique un contrat implicite avec l'utilisateur de cette fonction. Il doit l'instancier avec un type pour lequel l'opérateur * existe. Par exemple le type `int`

```
cout << square(5) << endl; // affiche 25
```

- Par contre, le code suivant ne compile pas

```
cout << square("Hello") << endl;
```

Contrat implicite

- Notons que l'erreur de compilation est localisée dans la fonction générique, pas là où l'erreur est réellement commise: à l'instanciation de `square` avec un type non compatible.
- Il y a pire... Reprenons la version minimaliste de notre classe `CVector`

```
class CVector {
  int x, y;
public:
  CVector(int x, int y) : x(x), y(y) {};
};
```

- Et stockons des objets de ce type dans un `std::vector`.

```
vector<CVector> v;
```

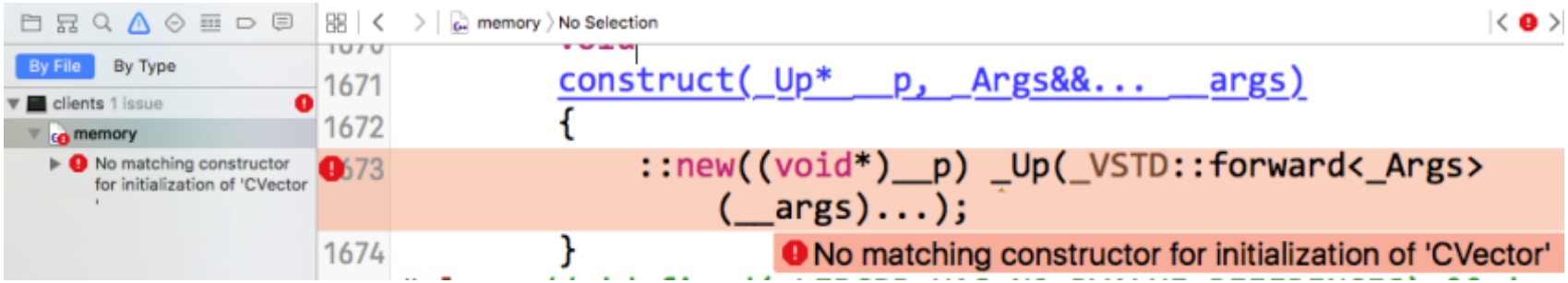
- Jusque-là tout va bien…

Contrat implicite

- Essayons de redimensionner ce `vector`

```
v.resize(2);
```

- Nous obtenons une erreur de compilation à la ligne 1673 du fichier `<memory>` de la librairie standard



- Pourquoi ???

Contrat implicite

- Pour comprendre cette erreur, relisons le prototype de la méthode `resize`

public member function

std::vector::resize

C++98 | C++11

```
void resize (size_type n, value_type val = value_type());
```

- Il y a un 2ème paramètre de type `value_type`. C'est en fait un `typedef` du paramètre générique `T` de `vector`

Member types

C++98 | C++11

member type	definition
value_type	The first template parameter (T)

- `resize` avec le 2ème paramètre non spécifié utilise donc le constructeur par défaut de `T`. Et ce constructeur n'existe pas pour notre classe `CVector`. Nous n'avons pas respecté le contrat implicite de `std::vector`

Contrat explicite

- En C++, il n'y a malheureusement pas moyen d'expliciter ce contrat, ce qui permettrait au compilateur de générer une erreur au bon endroit, i.e. à l'instanciation avec un type non compatible.
- Par contre, Java et C# offrent cette possibilité. Par exemple, en C# on peut écrire

```
public class maClasse<T> where T : IComparable, new()
{
  // T doit mettre en œuvre l'interface IComparable -
  // donc offrir une méthode CompareTo(T) – et être
  // constructible sans paramètre
}
```

Concepts

- Le comité C++11 a envisagé d'introduire un mécanisme similaire sous le nom de concepts, mais l'idée a été abandonnée en juillet 2009.
- Une version préliminaire – non standardisée – a été mise en œuvre dans ConceptGCC et sera disponible dans GCC à partir de la version 6.
- En attendant … la seule manière d'expliciter ce contrat est de le faire dans la documentation

Un dernier exemple: `std::sort`

- Considérons un dernier exemple. On veut stocker des valeurs dans un `std::vector` puis les trier avec la fonction `std::sort` d'`<algorithm>`

```
vector<int> v { 0, 4, 2 };
sort(v.begin(), v.end());
```

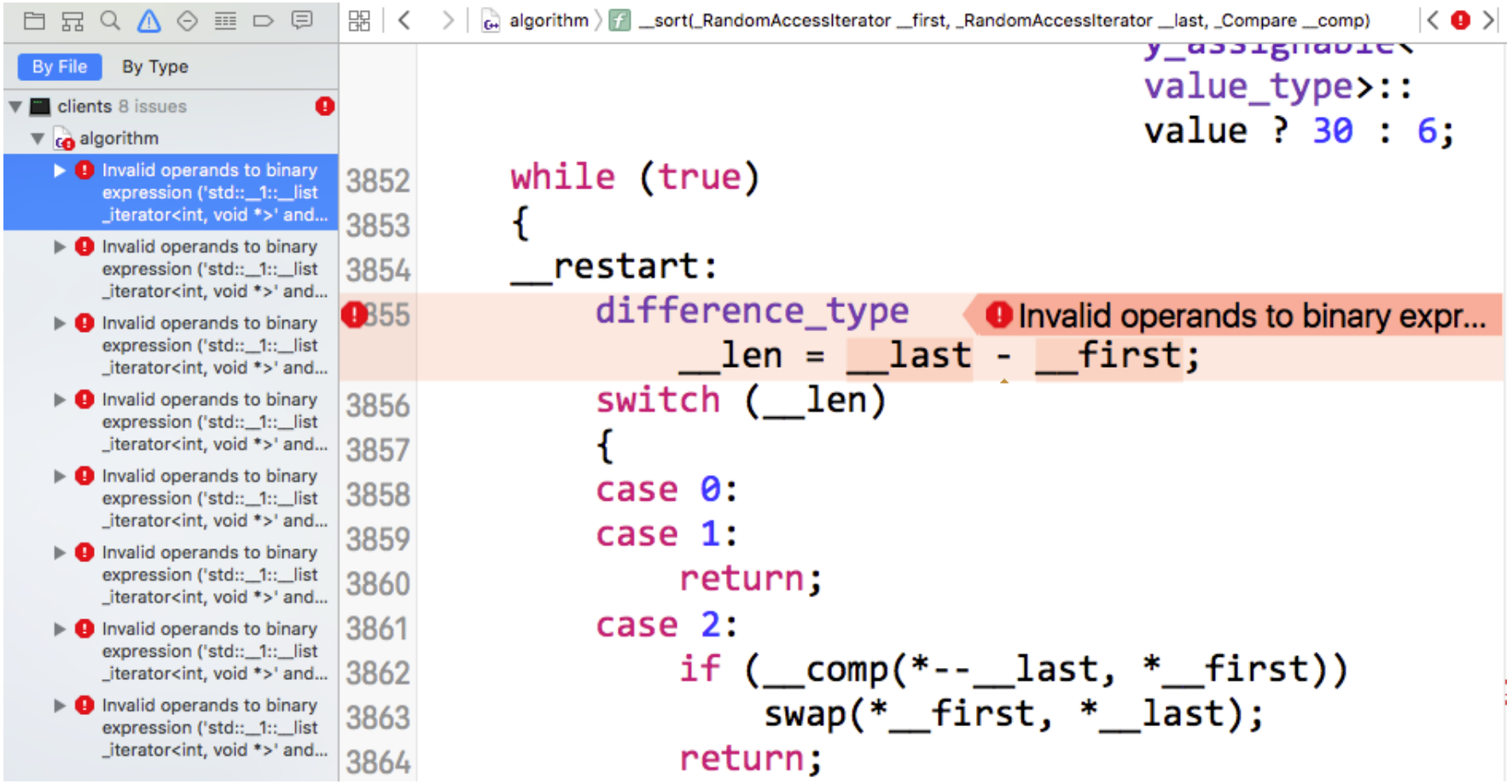
- On change d'avis et on veut plutôt utiliser une `std::list`

```
list<int> li { 0, 4, 2 };
sort(li.begin(), li.end());
```

- Cela ne devrait pas poser de problème, `std::list` fournissant aussi des méthodes `.begin()` et `.end()`

Un dernier exemple: `std::sort`

- En fait, si, cela pose problème …



Un dernier exemple: `std::sort`

- En lisant bien la documentation, on aurait pu éviter cette erreur. En effet, la fonction `std::sort` a le prototype

function template

std::sort

default (1)
```
template <class RandomAccessIterator>
  void sort (RandomAccessIterator first, RandomAccessIterator last);
```

- Notons qu'elle aurait pu s'écrire

```
template <class T> void sort(T first, T second);
```

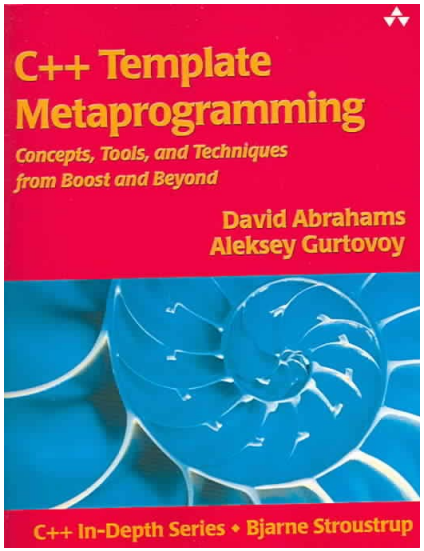
sans que cela change quoi que ce soit pour le compilateur. Le nom `RandomAccessIterator` documente le fait que l'itérateur reçu en paramètre doit mettre en œuvre des opérateurs tels que +, -, <, >=, …

- La lecture de la documentation de `std::list::begin()` nous apprend que ce n'est pas le cas de l'itérateur qu'elle fournit

Template Metaprogramming

Si vous n'en avez pas encore assez…

- Une utilisation avancée des templates permet notamment de répartir le temps de calcul entre le compilateur et l'exécution en pré-calculant des valeurs, des tests, en déroulant des boucles, … C'est le template metaprogramming (TMP)
- En fait, il n'y a pas de limite théorique à ce que l'on peut calculer avec le TMP. La compilation de templates C++ est Turing-complète
- Nous nous contenterons d'un exemple



Factorielle avec TMP

- Considérons la fonction factorielle récursive

```
unsigned int factorielle(unsigned int n) {
    return (n == 0) ? 1 : n * factorielle(n - 1);
}
```

- En TMP, on la réécrit avec une structure générique spécialisée pour le cas trivial `N == 0`.

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

template <> struct Factorielle<0> {
    enum { valeur = 1 };
};
```

- Utiliser `Factorielle<10>::valeur` dans le code instancie toutes les structures pour `0 <= N <= 10` et calcule toutes les `valeur` à la compilation. L'exécution est instantanée.