C++20: Contrats & Spécialisation

Définir des contraintes sur les templates
Spécialiser du code en fonction des types d’éléments

Quelques soucis avec les template (1)

Problème 1: **Détection des erreurs au moment de la compilation**:

```
template<typename T1, typename T2>
int add(T1 left, T2 right) {
      return left + right; }

int main() {
      add((int)1, std::string("ab"));
      return 0;}
```

Pas d'opération disponible

Quelques soucis avec les template (2)

Problème 2: **Définition du type résultat au moment de la compilation et non pas a priori.**

```
template<typename T1, typename T2>
T1 add(T1 left, T2 right) {
      return left + right;}

int main() {
     std::cout << (add(1.5, 1) == add(1,
      1.5)) << std::endl;
     return 0;
}
```

FALSE

Quelques soucis avec les template (3)

Problème 3: **Multiplication des redéfinitions**

```
class Complex
{...};

Complex operator +(int, const Complex&) {...}
Complex operator +(unsigned int, const Complex&) {...}
Complex operator +(long int, const Complex&) {...}
Complex operator +(long unsigned int, const Complex&)
{...}
Complex operator +(float, const Complex&) {...}
Complex operator +(double, const Complex&) {...}
Complex operator +(long double, const Complex&) {...}
```

La notion de contrat

- **Idée** :
  - Un contrat formel est défini entre le code appelant les services de l'objet et les services fournis par l'objet
  - Si le code de l'objet fournissant les services respecte l'ensemble des clauses du contrat imposée par le code appelant les services de l'objet, aucune erreur ne pourra se produire.

- **Principe :**
  - Le processus qui va utiliser les services d'un objet émet un ensemble de demandes (claims)
  - L'objet utilisé définit un ensemble de propriétés (responsibilities)
  - L'adéquation des propriétés de l'objet utilisé avec les demandes du processus forment le contrat

L‘implantation des contrats pour les arguments de Template

- Premières tentatives pour C++0X
- Font leurs apparitions avec C++20
- Deux notions:
  - `constraint`: définition de la contrainte ou des contraintes associées aux paramètres d‘une fonction, une classe, …
  - `concept:` ensemble d‘exigences nommées imposées sur un élément

Définition d'une contrainte

```
template<typename T1, typename T2>
int add(T1 left, T2 right) {
    return left + right; }
```

- Quelle contrainte pour que le code compile ?
  - Opérateur + doit être définit pour `T1` et `T2`.

Définition d‘une exigence

```
template<typename T1, typename T2>
concept addable = requires(T1 a, T2 b) {
    { a + b } -> std::convertible_to<int>;
};
```

Indique que l’addition d’une valeur de type `T1` dénommée `a` avec une valeur de type `T2` dénommée `b` retourne un type qui est convertible vers `int`.

Spécification de la contrainte

```
template<typename T1, typename T2>
concept addable = requires(T1 a, T2 b) {
    { a + b } -> std::convertible_to<int>;
};

template<typename T1, typename T2>
int add(T1 left, T2 right) requires addable<T1, T2> {
    return left + right;
}

int main() {
    add((int)1, std::string("ab"));
    add(1, 1.5);
    return 0;
}
```

Erreur désormais indiquée ici
comme étant une violation de contraintes

Comment fonctionne l‘évaluation de la contrainte

- Une contrainte est **un prédicat évalué** au **moment de la compilation**.

Pour chaque contrainte,

1. essaye de compiler le code exprimer dans la contrainte, si échec, le prédicat n‘est pas vérifié
2. vérifie éventuellement la contrainte sur le type résultat de la compilation ou la valeur retournée.

TD -- Partie 1

- Déterminer les contraintes.
- Définir des contraintes en utilisant des contraintes préétablies.

Spécialiser le code en fonction des paramètres de template

- Idée:
  - Les contraintes permettent de déterminer entre deux implantations laquelle est valide.
    - Le compilateur liste l‘ensemble des spécialisations possibles
    - Le compilateur teste si les exigences sont satisfaites, si ce n‘est pas le cas, passe à l‘exigence suivante.

Exemple de spécialisation par contraintes

```
#include<concepts>
#include<cmath>

template<typename T>
T add_digit(T value, int digit)
        requires std::integral<T> {
    return value * 10 + digit;
}

template<typename T>
T add_digit(T value, int digit)
        requires std::floating_point<T> {
    return std::floor(value) * 9 + digit + value;
}
```

Définir des contraintes complexes

```
template<typename T>
concept NotVoid = ! std::same_as<T, void>;

template<typename iteratorT>
concept ForwardIteratorOnPairs = requires
(iteratorT it)
{
    { it->first } -> NotVoid;
    { it->second } -> std::convertible_to<bool>;
} && std::forward_iterator<iteratorT>;
```

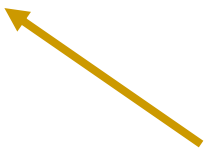
Ce qui nous donne:

```
std::cout << ForwardIteratorOnPairs<
    typename std::vector<int>::iterator> << std::endl;
      => false

std::cout << ForwardIteratorOnPairs<
    typename std::vector<std::pair<int,
       std::string>>::iterator> << std::endl;
      => false

std::cout << ForwardIteratorOnPairs<
    typename std::vector<std::pair<int,
       bool>>::iterator> << std::endl;
      => true
```

Obtenir le type résultat d'une opération

Problème 2: **Définition d'un résultat résultant d'une opération**

```
template<typename T1, typename T2>
T1 add(T1 left, T2 right) {
      return left + right;}

int main() {
     std::cout << (add(1.5, 1) == add(1,
      1.5)) << std::endl;
     return 0;
}
```



FALSE

Définir un type de résultat en fonction de l'évaluation d'une expression

```
template<typename T1, typename T2>
auto add(T1 left, T2 right) ->
decltype(left + right) {
        return left + right;
}

int main() {
     std::cout << (add(1.5, 1) == add(1,
      1.5)) << std::endl;
     return 0;
}
```

TRUE

Factoriser le code

```
class Complex
{
    template<std::floating_point floatT>
    Complex(floatT value): mRealPart((double)value),
        mImaginaryPart(0.0) {}
    template<std::integral intT>
    Complex(intT value): mRealPart((double)value),
        mImaginaryPart(0.0) {}

    template<typename floatOrIntT>
        requires std::integral<floatOrIntT> ||
                 std::floating_point<floatOrIntT>
    Complex& operator +=(floatOrIntT value) {
        mRealPart += (double)mRealPart;
        return *this;
    }
…
```

TD -- Partie 2

- Définir des contraintes pour caractériser un conteneur.
- Utiliser les contraintes pour spécialiser du code.