

Les modèles

Introduction au C++ et à la programmation objet

E. Courcelle

CALMIP, UMS 3669

Décembre 2019

Les modèles pour quoi faire ?

Ce que nous savons faire:

- Exprimer les relations d'agrégation
- Exprimer la relation "Est une sorte de"

Ce dont nous avons encore besoin:

- Savoir exprimer la relation "Est une liste de"
 - Cela nous permettra d'implémenter des "conteneurs d'objets" avec une structure définie indépendante du type d'objets que l'on met dedans

Les modèles vont nous le permettre

Un modèle de fonction

La fonction générique ci-dessous renvoie le plus "petit" de ses deux arguments:

```
template <typename T> min(const T& a, const T& b)
{
    if (a > b) return b;
    else return a;
}
```

Le code suivant pourra être compilé:

```
int i=4, j=6;
int m = min(i, j);

float x=3.14, y=5.65;
float z = min(x, y);

string s1="houlala", s2="aieaieaie";
string s = min(s1, s2);
```

Mais le code suivant **ne compilera pas**:

```
complexe c1=0, c2=1;
complexe c = min(c1, c2);
```

Trois fonctions différentes ont été générées, mais ça coince sur la quatrième.

Un modèle de classe

Une nouvelle définition des complexes, qui peut reposer sur différents types de nombres:

```
template<typename NUM=float> class complexe {
public:
    complexe(NUM x=0, NUM y=0);
    complexe(const complexe<NUM> & );
    ~complexe();
    operator NUM();
    complexe<NUM> & operator=(const complexe<NUM> &);
    complexe<NUM> & operator+=(const complexe<NUM> &);
    NUM get_r() const { return r;};
    NUM get_i() const { return i;};
    void set_r(NUM x) { r=x; m_flg=false;};
    void set_i(NUM x) { i=x; m_flg=false;};
    NUM get_m() const;
    static void set_debug() { deb_flg=true;};
    static void clr_debug() { deb_flg=false;};

private:
    NUM r;
    NUM i;
    mutable bool m_flg;
    mutable NUM m;
    static bool deb_flg;
};
```

Un modèle de classe (2)

Définition d'une fonction-membre en-dehors de la classe:

```
template <typename NUM> void complexe<NUM>::_calc_module() const {  
    m=sqrt(r*r+i*i);  
};
```

Il s'agit d'une fonction modèle, membre de la classe `complexe<NUM>`

Instantiation du modèle

On peut alors utiliser notre classe de la manière suivante:

```
main() {  
    complexe◇ F;           // Utilisation de la valeur par défaut: float  
    complexe<double> D;  
}
```

Ou, sans doute plus lisible grâce à l'instruction C typedef:

```
typedef complexe◇ complexe_float;  
typedef complexe<double> complexe_double;  
  
main() {  
    complexe_float F;  
    complexe_double D;  
}
```

Paramètres de modèles

Deux types de paramètres sont utilisables dans les modèles:

- typename suivi d'un **nom de type** (ou de classe)
- int (ou un autre type entier) suivi d'un nombre, qui indique par exemple une dimension

La classe tableau réécrite:

```
template<size_t TAILLE> class tableau {  
    private:  
        int buffer[TAILLE];  
  
    public:  
        tableau() {};  
};  
  
template<size_t TAILLE> tableau<TAILLE>::tableau() {  
    for (size_t i=0; i<TAILLE; ++i) {  
        buffer[i] = 0;  
    };  
};  
  
main() {  
    tableau<100> T1,T2;  
    T1 = T2;  
}
```

- Pas besoin d'allocation mémoire ni de copie (pas de trio infernal, le compilateur fait ça très bien)
- On a besoin d'un constructeur sinon rien n'est initialisé
- **ATTENTION ! La taille du tableau est fixée lors de la compilation**

Paramètres de modèles (2)

La classe Tableau avec deux paramètres de modèles:

```
template<size_t TAILLE, typename T> class tableau {  
    private:  
        T buffer[TAILLE];  
  
    public:  
        tableau(){};  
};  
  
template<size_t TAILLE, typename T> tableau<TAILLE,T>::tableau() {  
    for (size_t i=0; i<TAILLE; ++i) {  
        buffer[i] = 0;  
    };  
};  
  
main() {  
    tableau<100,float> T1,T2;  
    T1 = T2;  
}
```

Spécialisation

Dans ce qui suit, on suppose que les objets mis dans le tableau sont **pourvus d'un opérateur inférieur**.

Ecrivons la fonctions maxVal, qui va rechercher la valeur la plus grande du tableau:

```
T template<size_t TAILLE, typename T> tableau<TAILLE>::maxVal() {  
    T maxVal = "La valeur la plus grande possible pour T";  
    for (size_t i=0; i<TAILLE; ++i) {  
        if (buffer[i] < maxVal) maxVal=buffer[i];  
    };  
    return maxVal;  
};
```

Comment exprimer "La valeur la plus grande possible pour T" ?

Si T est un type numérique, on peut utiliser les defines trouvés dans limits.h:

- SHRT_MIN SHRT_MAX pour des short
- INT_MIN, INT_MAX pour des int
- ...

... sauf que ça ne va pas bien s'insérer dans le modèle !

On passe donc par une classe de la bibliothèque standard:

```
T template<size_t TAILLE, typename T> tableau<TAILLE>::maxVal() {  
    T maxVal = std::numeric_limits<T>::max();  
    for (size_t i=0; i<TAILLE; ++i) {  
        if (buffer[i] < maxVal) maxVal=buffer[i];  
    };  
    return maxVal;  
};
```

Spécialisation (2)

Et pour une classe écrite par moi (maClasse), je **spécialise** le modèle:

```
template< struct std::numeric_limits<maClasse> {  
    static maClasse min() { return maClasse(...); };  
    static maClasse max() { return maClasse(...); };  
};
```

Voir le fichier d'en-têtes `limits`

On peut spécialiser complètement ou partiellement les modèles:

```
template<typename T1, typename T2> class Machin {  
    ...  
}  
  
// Les deux parametres ont le meme type  
template<typename T> class Machin<T,T> {  
    ...  
}  
  
// Le second type est double  
template<typename T1> class Machin<T1,double> {  
    ...  
}  
  
// T1 et T2 sont doubles  
template<> class Machin<double,double> {  
    ...  
}  
  
// On travaille avec des pointeurs  
template<typename T1, typename T2> class Machin<T1 *, T2 *> {  
    ...  
}
```

ATTENTION !

- Tout est fixé (dimensions, types, ...) lors de la compilation
- Le modèle permet d'éviter le copié-collé grâce à la programmation générique
- Les codes des modèles sont écrits **dans les .hpp**, car le compilateur a besoin du code lors de la compilation

LE PLUS

- Tout est fixé (dimensions, types, ...) lors de la compilation:
 - le compilateur peut faire tous les tests de cohérence
- Le code généré peut être très rapide (pas d'allocation mémoire, ...)
- Une bibliothèque reposant sur des templates sera très simple à installer (juste copier des fichiers, qui sont tous des .hpp)

LE MOINS

- Tout est fixé (dimensions, types, ...) lors de la compilation:
 - le code offrira moins de souplesse (dimensions)
- La compilation est plus lente

De nombreuses bibliothèques reposent sur les templates:

- La stl: Standard Template Library
- Beaucoup de bibliothèques intégrées à boost
- La bibliothèque de calcul matriciel eigen

<http://eigen.tuxfamily.org> est une bibliothèque de templates

Ces bibliothèques utilisent des techniques de "Template metaprogramming"

cf. https://en.wikipedia.org/wiki/Template_metaprogramming

(à suivre)