Prog. avancée
Master 1

Programmation avancée

C++ TEMPLATEs

Sylvain Théry

thery@unistra.fr

ICube

UNIVERSITÉ DE STRASBOURG

Concept

Paramétrer une fonction ou une classe par une valeur constante ou un type (simple, classe).

Pas de surcout à l'exécution (une version compilée par instantiation uniquement si elle est utilisée)

Objectifs

Eviter la réécriture de code

Optimiser le code (pas de polymorphisme)

Fonctions: exemple max

But: une fonction max qui renvoit le maximum de 2 valeurs.

```
int max(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
}
```

```
float max(float a, float b)
{
    if (a>b)
        return a;
    else
        return b;
}
```

return ((a > b) ? a : b);

return ((a > b) ? a : b);

Fonctions: exemple max

Une seule version paramétrée par le type:

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

Remarque:

- le type T doit implémenter l'opérateur **>** sinon problème de compilation à l'instanciation.
- a & b de même type: max(3.1f,2.3) → erreur

Fonctions: exemple max

Lors de l'utilisation de la fonction template, le paramètre peut-être déduit automatiquement par le compilateur.

```
float x = max(3.5f, 7.4f);
```

```
double y = max(3.5, 7.4);
```

Mais on peut aussi le forcer explicitement

```
float z = max<float>(3.5, 7.4);
```

Classes: exemple Pile

But: Ecrire une classe Pile générique qui puisse s'instancier en Pile de int, Pile de float, etc...

Attention: Tous les objets dans la pile on le même type. Très différent d'une classe Pile qui puisse manipuler des objets de type différents !

Classes: exemple Pile


```
template <typename T>
class Pile
{
    T * data;
    …
    void push(const T& v);
    T top();
    ...
}
```

Syntaxe & organisation

La déclaration **et** l'implémentation doivent être incluses simultanément (header).

Trois solutions:

- *implémentations dans les déclarations*
- les déclarations suivis des implémentations dans le même fichier .h
- les déclarations dans le fichier .h, les implémentations dans un fichier .hpp inclu à la fin du fichier .h

Attention les prototypes de la déclaration et de l'implémentation doivent correspondre !

non vérifiable par le compilateur → pb au link !

Test

Il est d'autant plus important de tester le code template car celui-ci n'est pas forcément compilé pour toutes les possibilités d'instanciation.

Exemple: max<char*> (voir spécialisation)

Plusieurs niveaux:
- ça compile
- ça s'exécute
- ça fait ce que ça doit faire

Paramétrage

On peut paramétrer une fonction ou une classe template par:

- un type (template <typename T>)
- une constante numérique entière (template <int N>
- un booléen
- une combinaison des trois précédentes possibilités.

Ex: template <typename T, int W, int H>

Remarque paramètres

Il peut être parfois intéressant de faire des accesseurs (constant) aux paramètres template numériques.

Dans du code template ou utilisant du polymorphisme il peut-être utile de pouvoir retrouver le paramètre template à l'utilisation.

De même on peut définir un type local avec *using* pour les paramètres non numériques

Paramètres par défaut

Les paramètres de classe template peuvent avoir une valeur par défaut (même règle que pour les appels de fonctions).

Pour les fonctions template c'est uniquement supporté à partir de la norme C++11

Instanciation explicite (spécialisation)

On peut explicitement spécialiser un template pour un paramètre donné:

- pour optimiser une version
- pour palier à l'absence d'un opérateur utilisé dans la version générique.
- parce que la version générique donne pour certains paramètres un résultat faux
- pour terminer une récursivité !

Instanciation explicite (spécialisation)

Exemple : pour comparer des chaînes de caractères (C) avec notre fonction max if faut la spécialiser:

```
template <>
char* max<char*>(char* ch1, char* ch2)
{
    if (strcmp(ch1,ch2)>0)
        return ch1;
    return ch2;
}
```

Sinon on compare les adresses !

spécialisation vs surcharge

SPECIALISATION

```
template <>
char* max<char*>(char* ch1, char* ch2)
{
      if (strcmp(ch1,ch2)>0)
            return ch1;
      return ch2;
}
```

Force la signature à être cohérente avec la version générique

SURCHARGE

```
char* max(char* ch1, char* ch2)
{
      if (strcmp(ch1,ch2)>0)
            return ch1;
      return ch2;
}
```

La signature peut être différente

Si les deux sont définies, la surcharge l'emporte, mais on peut forcer l'accès à la version spécialisée

Spécialisation partielle

On peut spécialiser une méthode d'une classe template (tous les paramètres fixés)

On peut fixer un sous ensemble des paramètres d'une classe template (mais pas d'une fonction):

il faut alors redéfinir toute la classe

Spécialisation partielle


```
template <typename T, int N>
class A
{
….
     T fonction(int j) { return m_x[ N-j ];}
};

template <>
int A<int,0>::fonction(int j) { return m_x[ 0 ];}
```

Spécialisation partielle


```
template <typename T1, int N, typename T2>
class A
{
…
};

template <typename T1, typename T2>
class A<T1,0,T2>
{
…
};

template <typename T>
class A<int,0,T>
{
…
};
```

template & static

On peut bien-sur déclarer un membre *static* dans une classe template. Il ne faut pas oublier de l'initialiser à l'extérieur de la classe (mais ici dans le .h ou .hpp)

Exemple:

```
template<typename T>
int Pile<T>::s_xxx = 0;
```

template & friend

On peut déclarer une fonction friend dans une classe template (operator << par exemple)

Il faut la déclarer comme une fonction template:

```
template <typename T>
friend std::ostream& operator<<(std::ostream& out, const A<T>& a) ;
```

template récursif

On peut utiliser la récursivité (sur les fonctions et les classes) :

```
template <int N>
void rec_fonc()
{
    …
    rec_fonc<N-1>()
}
```

Attention à la condition d'arrêt (spécialisation):

```
template <>
void rec_fonc<0>()
{
    …
}
```

Pour le fun: Fibonacci


```
template < int N>
class Fibonacci
{
public:
	static const int Val = Fibonacci<N-2>::Val + Fibonacci<N-1>::Val;
};
```

```
template <>
class Fibonacci<0>
{
public:
	static const int Val = 1;
};
```

```
template <>
class Fibonacci<1>
{
public:
	static const int Val = 1;
};
```

Appel: `int f = Fibonacci<7>::Val;`

Pas de caculs à l'exécution, calculs fait par le compilateur !

template variadic

fonction/classe template à nombre de paramètres variable.

Permet des fonctions à nombre de paramètres variable typés

```
template <typename ...Ts>          ...Ts déclaration des types
void fonc(Ts… ps)                  Ts... utilisation des types
{
      fonc2(ps…);                  ps… utilisation des param
```

Nombre de types dans Ts: sizeof...(Ts)

Comment extraire les types de Ts : récursivité de spécialisation

```
template <typename T1, typename ...Ts>
void fonc(const T1& p1, Ts...ps) {
            fonc(ps…);

void fonc() {
```

fonction template variadic


```
int somme()
{
return 0;
}

template <typename... Ts>
int somme(int a, Ts... r)
{
    return a+somme(r...);
}
```

```
template <typename T>
T tsomme(T v)
{
    return v;
}

template <typename T, typename... Ts>
auto tsomme(T a, Ts... r)
{
    return a+tsomme(r...);
}
```

auto: ici C++14 !!!

classe template variadic


```
template <int... Ns>
struct Somme
{};
```

déclaration de la classe générique

```
template <int N1>
struct Somme<N1>
{
    static constexpr int value = N1;
};
```

spécialisation pour la condition d'arrêt

```
template <int N1, int... Ns>
struct Somme<N1,Ns...>
{
    static constexpr int value = N1 + Somme<Ns...>::value;
};
```

spécialisation pour la récurence

SFINAE

Substitution Failure Is Not An Error

Code template compilé sous certaines conditions

Exemple:

```
template <typename T>
auto fonction(T p1)
-> typename std::enable_if<std::is_floating_point<T>::value, void>::type
{
      std::cout << “reel “ << p1 << std::endl;
}
```

std::is_floating_point<T>::value -> vrai si T est float ou double

std::enable_if< B, T >::type → T si B est vrai, sinon *failure*

Tout est évalué lors de la compilation.

SFINAE


```
template <typename T>
auto fonction(T p1)
-> typename std::enable_if<std::is_floating_point<T>::value, void>::type
{
      std::cout << “reel “ << p1 << std::endl;
}

template <typename T>
auto fonction(T p1)
-> typename std::enable_if<std::is_floating_point<T>::value, void>::type
{
      std::cout << “entier “ << p1 << std::endl;
}
```

Trait C++

Une classe de Trait est une classe générique qui va donne des informations sur son paramètre.

Exemple: **std::is_signed** (#include<type_traits>)

Utilisation: std::is_signed<float>::value

⇩

constexpr bool

Convention : 1 info → value ou type

depuis C++14 : std::is_signed<float>()

Trait C++

Exemple de trait plus complexe:

std::numeric_limits (#include<limits>)

Utilisation:

std::numeric_limits<float>::is_signed
std::numeric_limits<float>::min()
std::numeric_limits<float>::max()
…

Peut contenir:

- des constantes
- des fonctions (constexpr ou pas)
- des types

Trait C++

Ecrire sa propre classe de trait:

Une classe template avec un paramètre T (type) qui fournie

- des constantes
  - booléennes:

```
static constexpr bool is_ok = true;
```

  - numériques:

```
static constexpr int size = 3;
```

- des types:

```
using Scalar = T::Scalar
```

Trait C++

Principe de la spécialisation de template:

classe générique ⟹ les valeurs par défaut

Pour chaque paramètre on écrit une spécialisation.

Lors de la définition d'un nouveaut type, il suffit d'ajouter la spécialisation du trait (dans le .h du type)

Trait C++

Ajouter nouveau type aux paramètres d'un trait existant

Exemple avec std::is_signed

```
namespace std
{
    template<>
    class is_signed<Vec3f>
    {
    public:
        static constexpr bool value = true;
    };
}
```

Trait C++ exemple


```
template<typename T>
struct other
{
	using type = T;
};
```

```
template<>
struct other<int>
{
	using type = float;
};
```

```
template<>
struct other<float>
{
	using type = int;
};
```

```
template <typename T>
typename other<T>::type pingpong(T v)
{
	return other<T>::type(v);
}

int main()
{
	auto x = pingpong(3.0f);
	auto y = pingpong(x);

	std::cout << typeid(x).name() << std::endl;
	std::cout << typeid(y).name() << std::endl;
```