

# Programmation C++ Avancée

## Session 5 – Patrons de Fonction et de Classe

Joel Falcou   Guillaume Melquiond

Laboratoire de Recherche en Informatique

# Contexte

---

## Qu'est-ce qu'un template ?

- Modèle générique de code de fonction ou de classe
- Paramétrable par des types concrets ou des valeurs constantes
- Spécifier ces paramètres **instancie** le template et génère du code

## Pourquoi les templates ?

- Base de la généricité en C++
- Support pour le polymorphisme statique
- Aller au-delà des macros du préprocesseur

# Syntaxe

---

- Paramétrage introduit par `template<>`

- Chaque paramètre peut être :

- un **type** introduit par les mots-clés `class` ou `typename`

```
template<typename T> T maximum(T a, T b);
```

- une **valeur** de type booléen, entier ou pointeur

```
template<int Value> struct number;
```

- un **type template** déclaré *in situ*

```
template<template<class> class U, class T>  
U<T> make_container(std::size_t N, T const &value);
```

- Ces paramètres peuvent avoir une valeur par défaut

```
template<class T = int, int Value = 0> struct number;
```

# Fonction template

---

## Principes

- Une **fonction template** est un modèle de génération de fonction paramétrable
- Elle remplace avantageusement les macros en explicitant les types

```
template<typename T> T maximum(T a, T b)
{
    return a > b ? a : b;
}
```

- L'appel d'une **fonction template** s'effectue comme pour une fonction normale
- Le compilateur **infère** les paramètres à partir du type des arguments

```
int i,j,k;

k = maximum(i,j);
```

# Fonction template

---

## Exemple : la fonction swap

```
void swap(int &a, int &b)
{
    int tmp(a);
    a = b;
    b = tmp;
}
```

```
void swap(float &a, float &b)
{
    float tmp(a);
    a = b;
    b = tmp;
}
```

# Fonction template

---

## Exemple : la fonction swap

```
template<class T> void swap(T &a, T &b)
{
    T tmp(a);
    a = b;
    b = tmp;
}
```

# Fonction template

---

## Algorithme de résolution d'un appel template

Choix de la bonne fonction lors d'un appel :

1. Parmi les fonctions non-template :
  - 1.1 Une seule association exacte : **résolution terminée**
  - 1.2 Plusieurs associations exactes : **erreur**
  - 1.3 Aucune : **aller en 2**
2. Parmi les fonctions template :
  - 2.1 Une seule association exacte : **résolution terminée**
  - 2.2 Plusieurs associations exactes : **erreur**
  - 2.3 Aucune : **aller en 3**
3. Réexaminer les fonctions non-template de façon classique avec d'éventuelles conversions de type

# Fonction template

---

## Résolution d'un appel template

```
template<typename T> T Function1() { return T(); }
```

```
int x = Function1();           // KO
```

```
int x = Function1<int>(); // OK
```

```
template<typename T> void Function2(T x1, T x2) {}
```

```
int x1 = 5;
```

```
double x2 = 6.5;
```

```
Function2(x1, x2); // KO
```

```
Function2<double>(x1, x2); // OK
```

```
Function2(static_cast<double>(x1), x2); // OK
```



# Fonctions et inférence de type

---

## Impact sur le type de retour des fonctions

- auto et decltype simplifient l'écriture du prototype des fonctions
- Notion de *trailing return type*

```
template<typename T1, typename T2>
/* ?????? */
add(T1 const &a, T2 const &b)
{
    return a + b;
}
```

# Fonctions et inférence de type

---

## Impact sur le type de retour des fonctions

- auto et decltype simplifient l'écriture du prototype des fonctions
- Notion de *trailing return type*

```
// C++ 11
template<typename T1, typename T2>
auto add(T1 const &a, T2 const &b) -> decltype(a + b)
{
    return a + b;
}
```

# Fonctions et inférence de type

---

## Impact sur le type de retour des fonctions

- auto et decltype simplifient l'écriture du prototype des fonctions
- Notion de *trailing return type*

```
// C++ 14
template<typename T1, typename T2>
auto add(T1 const &a, T2 const &b)
{
    return a + b;
}
```

# Templates variadiques

---

## Application aux fonctions

- Variante type-safe de l'ellipse ... du C
- Notion de *parameter pack*
- Déduction automatique des types

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

```
template<typename T0, typename... Ts>  
auto sum(T0 v0, Ts... vs)  
{  
    return v0 + sum(vs...);  
}
```

# Templates variadiques

---

## Application aux fonctions

- Variante type-safe de l'ellipse ... du C
- Notion de *parameter pack*
- Dédution automatique des types

```
int main()
{
    auto a = sum(1u, 2, 3., 4.5f);
    std::cout << a << "\n";
    return 0;
}
```

# Templates variadiques

---

## Application aux fonctions

- Variante type-safe de l'ellipse ... du C
- Notion de *parameter pack*
- Dédution automatique des types

```
template<typename... Ts>  
auto sum(Ts... vs)  
{  
    // C++ 17  
    return (0 + ... + vs);  
}
```

# Classe template

---

## Principes

- Une **classe template** est un modèle de génération de classe paramétrable
- Elle permet de gérer des variantes de classes sans polymorphisme dynamique
- Les paramètres template sont à spécifier explicitement
- Une classe template ne devient un type complet que lorsqu'elle est entièrement spécifiée

## Quelques détails

- Une classe non template peut avoir des méthodes template
- Une classe template peut avoir des méthodes template utilisant des paramètres supplémentaires
- Si A hérite de B, et que C est une classe template, il n'existe aucun lien implicite entre C<A> et C<B>

# Classe template

---

## Exemple : la classe `pair<T1,T2>`

```
template<class T1, class T2> struct pair
{
    pair() {}
    pair(T1 const &a, T2 const &b) : first_(a), second_(b) {}

    T1 first_;
    T2 second_;
};
```



# Spécialisation de template

---

## Spécialisation totale

Permet de spécifier un template entièrement pour un jeu de paramètres donnés

```
template<class T1, class T2> struct pair
{
    pair() {}
    pair(T1 const &a, T2 const &b) : first_(a), second_(b) {}

    T1 first_;
    T2 second_;
};

template<> struct pair<void,void>
{};
```

# Spécialisation de template

---

## Spécialisation partielle

Permet de spécifier certains paramètres du template afin de spécialiser une partie de son comportement

```
template<class T> struct add_ref  
{ typedef T &type; };
```

```
template<class T> struct add_ref<T&>  
{ typedef T &type; };
```

```
template<class T> struct add_ref<T const>  
{ typedef T const &type; };
```

```
template<class T> struct add_ref<T const&>  
{ typedef T const &type; };
```

Attention aux ambiguïtés !

# Spécialisation de template

---

Exemple : sélecteur conditionnel de type

Spécification

`if_<b,T1,T2>::type` s'évalue en T1 si b vaut true et T2 sinon

# Spécialisation de template

---

## Exemple : sélecteur conditionnel de type

```
template<bool Condition, typename T, typename F> class if_;  
  
template<typename T, typename F> struct if_<true, T, F>  
{  
    typedef T type;  
};  
  
template<typename T, typename F> struct if_<false, T, F>  
{  
    typedef F type;  
};
```

# Spécialisation de template

---

## Exemple : sélecteur conditionnel de type

```
int main()
{
    typename if_<true , int, void*>::type number(3);
    typename if_<false, int, void*>::type pointer(&number);

    typedef typename if_<(sizeof(void *) > sizeof(uint32_t))
        , uint64_t
        , uint32_t
        >::type    integral_ptr_t;

    integral_ptr_t ptr =
        reinterpret_cast<integral_ptr_t>(pointer);
}
```