Programmation C++ Avancée

Session 5 – Patrons de Fonctions et de Classes

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 abstraite ou des valeurs constantes
- Spécifier ces paramètres **instantie** le `template` et génére du code

Pourquoi les templates en C++ ?

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

Principe de bases

Syntaxe

- Paramètrage introduit par : `template<>`
- Chaque paramètres peut être :
  - **un type** introduit par le mot clé `class` ou `typename`

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

  - **une valeur** de type entière

```
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 un 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 étant type-safe

```
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 **infére** des arguments le type des paramètres.

```
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)
{
  int tmp(a);
  a = b;
  b = tmp;
}
```

Fonction template

Algorithme de résolution d'un appel `template`

Quand le compilateur détecte un appel de fonction il suit l'algorithme suivant pour associer la bonne fonction.

1. Dans les fonctions non templates :
   - 1.1 Une seule association exacte : résolution terminée
   - 1.2 Plusieurs association exacte : erreur
   - 1.3 Aucune : Allez en 2
2. Dans les fonctions templates :
   - 2.1 Une seule association exacte : résolution terminée
   - 2.2 Plusieurs association exacte : erreur
   - 2.3 Aucune : Allez en 3
3. Réexaminer les non templates de façons classiques avec d'éventuelles conversions de type.

Fonction template

Cas particulier - Retour template

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

int x = Fonction();       // KO
int x = Fonction<int>(); // OK

template <typename T> void Fonction2( T x1, T x2 ) {}

int x1 = 5;
double x2 = 6.5;

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

Templates variadiques

Applications aux fonctions

- Remplacement type-safe du `...` du C
- Notion de *paramaters pack*
- Déduction automatique des types

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

Templates variadiques

Applications aux fonctions

- Remplacement type-safe du `...` du C
- Notion de *paramaters pack*
- Déduction automatique des types

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

Templates variadiques

Applications aux fonctions

- Remplacement type-safe du `...` du C
- Notion de *paramaters pack*
- Déduction automatique des types

```
int main()
{
  auto a = sum(1u,2,3.,4.5f);

  std::cout << a << "\n";

  return 0;
}
```

Fonctions et inférence de type

Impact sur le 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 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 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;
}
```

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`[s] 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 une ou plusieurs méthodes `template`[s]
- Une classe `template` peut avoir une ou plusieurs méthodes `template`[s] paramétrés sur un autre jeu de type abstrait
- 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_;
};
```

Atelier Pratique

La classe `fixed_array<T,N>`

- `fixed_array<T,N>` représente un tableau de `N` éléments de type `T`
- Proposez une implantation simple fournissant les opérations classiques sur les tableaux

Indices

- `T` et `N` permettent de reconstruire un `T tab[N]`
- Que deviennent `size`, `empty` etc ...

Spécialisation de template

Cas d'utilisation

- Quid d'un `fixed_array<T,0>` ?
- Specifier des comportement differents en fonctions du paramètrage
- Version statique du patron de conception "Strategy" ou "State"

Mise en œuvre

- Spécialisation totale
- Spécialisation partielle

Spécialisation de template

Spécialisation totale

Permet de spécifier un `template` entièrement pour un jeu de paramètre donné.

```
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 certain arguments 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 ambiguités !

Spécialisation de template

Selecteur conditionnel de type

Spécifications

`if_<Bool,T1,T2>::type` s'évalue en `T1` si `Bool == true` et `T12` sinon.

Spécialisation de template

Selecteur 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

Selecteur conditionnel de type

```
int main()
{
   typename if_<true, int, void*>::result number(3);
   typename if_<false, int, void*>::result 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);
}
```