C++ AVANCÉ

Programmation Template

Joel FALCOU

Techniques de Bases

Fonctions Génériques

Principes

- Fournir une syntaxe permettant d'exprimer l'universalité d'une fonction surchargée
- Limiter l'impact sur le code écrit, maintenu et généré

```
1  template<typename Type>
2  Type  minimum(Type  a, Type  b)
3  {
4    return a < b ? a : b;
5  }
```

Fonctions Génériques

Principes

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

// Type est deduit comme `int`
auto x = minimum(4, 5);

// Type est deduit comme `float`
auto y = minimum(4.6f, -0.5f);

// error: no matching function for call to 'minimum(double, int)'
auto z = minimum(4.6, 5);
```

Fonctions Génériques

Paramètres *Templates*

- Introduit par la notation `template< ... >`
- Contient au moins un paramètre de type
- Ces paramètres ont un identifiant unique introduit par le mot-clé `typename`

```
1  //       1st parametre template
2  //                 |   2nd parametre template
3  //                 |             |   3rd parametre template
4  //                 |             |             |
5  //                 v             v             v
6  template<typename Src, typename Dst, typename Size>
7  void copy(Src const& src, Dst& dst, Size qty)
8  {
9    for(Size i=0;i<qty;++i)
10     dst[i] = src[i];
11 }
```

Déduction du Type de Retour

Principes

- Le type de retour des fonctions génériques peut être complexe à exprimer voire impossible à déterminer par le développeur
- Le compilateur a toutes les informations nécessaires à sa détermination exacte
- On introduit une syntaxe pour laisser la main au compilateur pour cette tâche

```
1  template<typename T1, typename T2>
2  /* Que mettre ici ??? */ addition(T1 a, T2 b)
3  {
4    return a + b;
5  }
```

Déduction du Type de Retour

auto et decltype

```
template<typename T1, typename T2>
//  auto indique que le type de retour est donné plus tard
auto  addition(T1 a, T2 b)
      → decltype(a+b) // decltype calcule le type de retour
{
  return a + b;
}
```

Déduction du Type de Retour

auto

```
template<typename T1, typename T2>
// auto indique au compilateur de trouver tout seul le type de retour
auto  addition(T1 a, T2 b)
{
  // Le type est le type de la valeur renvoyé par return
  return a + b;
}
```

Déduction du Type de Retour

Retour parfait

```
template<typename T>
// decltype(auto) agit comme auto mais conserve les qualificateurs
decltype(auto) minimum(T const& a, T const& b)
{
  return (a<b) ? a : b;
}
```

Classes Génériques

Motivations

- Certaines structures de données sont paramétrables
- Ex: un tableau contenant un type arbitraire
- Les structures génériques permettent de transmettre cette information

Mise en place

- Syntaxe identique à celle des fonctions génériques
- Guide de déduction
- Spécialisation et spécialisation partielle

Classes Génériques

Du concret au générique

```
1  struct array_of_5reals
2  {
3    float data[5];
4
5    std::size_t size() const { return 5; }
6
7    float  operator[](std::ptrdiff_t i) const { return data[i]; }
8    float& operator[](std::ptrdiff_t i)       { return data[i]; }
9  };
10
11 array_of_5reals x = {1.2f,0.3f,5f,7.9f,8.88f};
```

Classes Génériques

Du concret au générique

```
// array_of_5 est une structure template
template<typename Type> struct array_of_5
{
  Type data[5];

  std::size_t size() const { return 5; }

  Type  operator[](std::ptrdiff_t i) const { return data[i]; }
  Type& operator[](std::ptrdiff_t i)       { return data[i]; }
};

// array_of_5<int> instancie array_of_5 pour T = int
array_of_5<int> x = {12,3,57,79,888};
```

Classes Génériques

Du concret au générique

```
1  // array est une structure template
2  template<std::size_t N, typename Type> struct array
3  {
4    Type data[N];   // N est utilisable usable anywhere constants are expected
5
6    std::size_t size() const { return N; }
7
8    Type  operator[](std::ptrdiff_t i) const { return data[i]; }
9    Type& operator[](std::ptrdiff_t i)       { return data[i]; }
10 };
11
12 // array<3,int> instanciates array for Type = int and N = 3
13 array<3,int> x = {12,3,57};
```

Notion de Specialisation

Spécialisation de structures génériques

- Dans certains cas, le code contenu dans une structure générique peut vouloir varier en fonction d'un ou plusieurs paramètres templates afin de garantir la correction ou les performances du code en général.
- Il est possible de **spécialiser** un structure générique afin de s'assurer de la qualité du code généré.

Spécialisations totales ou partielles

- On parle de **spécialisation totale** si l'on fige la totalité des paramètres templates
- On parle de **spécialisation partielle** si l'on fige une partie des paramètres templates
- Lors de l'instantiation d'une structure générique présentant des spécialisations, le compilateur favorise la version **la plus spécialisée**

Notion de Specialisation

Code original

```
1  template<typename Type> struct label
2  {
3    label(Type v) : content_("value: '" + std::to_string(v) + "'") {}
4
5    std::string const& value() const { return content_; }
6
7    private:
8    std::string content_;
9  };
10
11 int main()
12 {
13   auto s = label<int>(4);
14   std::cout << s.value() << "\n";
15 }
```

Notion de Specialisation

Specialisation Totale

```
1  template<typename Type> struct label { /* old code */ };
2
3  template< > struct label<std::nullptr_t>
4  {
5    label(std::nullptr_t) : content_("'nulltpr'") {}
6    std::string const& value() const { return content_; }
7
8    private:
9    std::string content_;
10 };
```

Notion de Specialisation

Spécialisation Partielle

```
1  template<typename Type> struct label { /* old code */ };
2
3  template<typename T> struct label<T*>
4  {
5    label(T* v)
6    {
7      std::ostringstream str;
8      str << "address: " << (void*)(v);
9      content_ = str.str();
10   }
11
12   std::string const& value() const { return content_; }
13
14   private:
15   std::string content_;
16 };
```

Guides de Déduction

Challenge

- Les fonctions génériques déduisent le type de paramètres de manières implicites.
- Les classes génériques classes nécessite une sélection explicite.
- Le code devient souvent verbeux et répétitifs : `label<std::nullptr_t>(nullptr)`.

Solution

- Avant C++17: fonctions usines comme `std::make_pair`
- Aprés C++17: Les **Deduction Guide**

Guides de Déduction

Guides de Déduction Automatique

```
1  template<typename T> struct box
2  {
3    box(T  v) : value(v)  {}
4    box(T* p) : value(*p) {}
5    T value;
6  };
7
8  auto s = box(99);  // s est de type box<int>
9  auto t = box(&f);  // t est de type box<float>
```

Guides de Déduction

Guides de Déduction définis par l'Utilisateur

```cpp
template<typename T> struct box
{
  box(T  v) : value(v)  {}

  template<typename Iterator>
  box(Iterator b, Iterator e) : value(*(b + (e-b)/2)) {}

  T value;
};

// Deduction Guide customisé
template<typename Iterator>
box(Iterator b, Iterator e) -> box<typename std::iterator_traits<Iterator>::value_type>;

std::vector<int> v(10);
auto t = box(v.begin(), v.end());  // s est de box<int>
```

Templates Variadiques

Objectifs

- Certaines entités génériques nécessitent un nombre variables de paramètres templates.
- Exemples: `std::tuple`, tableaux multi-dimensionnel.
- Les **Template Parameters Pack** nous permettent de définir ces API.

```
1  template<typename... Args> void f(Args... args);
2
3  template<typename... Types> struct tuple {};
4  template<int... Sizes>      struct hull {};
```

Templates Variadiques - Cas récursif

```cpp
// Parametre Template
//         |
//         |       typename... est un variadic pack
//         |                   |
//         |                   |
//         v                   v
template<typename First, typename... Others>
auto total_size(First const&, Others const&... args)
{
  return sizeof(First) + total_size(args...);
}
```

Templates Variadiques - Cas récursif

```
1  template<typename First, typename... Others>
2  //          Parametre de la fonction
3  //                  |
4  //                  |       args est un parameter pack
5  //                  |                   |
6  //                  |                   |
7  //                  v                   v
8  auto total_size(First const&, Others const&... args)
9  {
10   return sizeof(First) + total_size(args...);
11 }
```

Templates Variadiques - Cas récursif

```cpp
template<typename First, typename... Others>
auto total_size(First const&, Others const&... args)
{
  // La notation ... déroule le contenu d'une expression
  // contenant un variadic pack ou un parameter pack pour
  // chacun des éléments qu'ils contiennent en les séparant
  // par une virgule syntaxique
  return sizeof(First) + total_size(args...);
}
```

Templates Variadiques - Cas récursif

```cpp
#include <iostream>

template<typename Type> auto total_size(Type const&) { return sizeof(Type); }

template<typename First, typename... Others>
auto total_size(First const&, Others const&... args)
{
  return sizeof(First) + total_size(args...);
}

int main()
{
  std::cout << total_size(1, 'z', 3.);
}
```

Templates Variadiques - Cas non-récursif

```cpp
#include <iostream>

template<typename... Others>
auto total_size(Others const&...)
{
  std::size_t size[] = { sizeof(Others)... };
  std::size_t r = 0;

  for(auto s : size)
    r += s;

  return r;
}

int main()
{
  std::cout << total_size(1, 'z', 3.);
}
```

Templates Variadiques - Cas non-récursif

```cpp
#include <iostream>

template<typename... Others>
auto total_size(Others const&...)
{
  // En C++17, ... s'utilise avec n'importe-quel opérateur binaire
  return (sizeof(Others) + ... + 0ULL);
}

int main()
{
  std::cout << total_size(1, 'z', 3.);
}
```

Le Problème du *Perfect Forwarding*

Contexte

- Certaines fonctions sont utilisées comme trampoline vers une ou plusieurs autres fonctions
- Elles doivent pouvoir passer n'importe quel catégorie de paramètres aux fonctions sous-jacentes
- Comment gérer les 3^N cas du à la gestion de N paramètres pouvant être des lvalue, des lvalue immuables ou des rvalue ?

```
1  // Ne prend pas de rvalue ou de const lvalue reference
2  template<typename Type, typename... Args>
3  auto instanciate(Args&... args) { return Type(args...); }
4
5  // Ne prend pas de refrence de lvalue
6  template<typename Type, typename... Args>
7  auto instanciate(Args const&... args) { return Type(args...); }
```

Solution: Les *Forwarding References*

- Si `T` est un paramètre template, alors `T&&` est une **référence universelle**
- Ce type peut matcher des lvalue, des lvalue constante, des rvalue sans erreurs
- La fonction `std::forward` permet de transférer proprement ces référence universelles à d'autres fonctions sans perte d'informations

```
1  template<typename Type, typename... Args>
2  // /!\ Args&& est une forwarding reference pas une rvalue-reference vers Args
3  auto instanciate(Args &&... args)
4  {
5    return Type( std::forward<Args>(args)... );
6  }
```

Méta-programmation

Contexte

Programmation Générative

- La programmation classique décrit du code dont les entrées et les sorties sont des données.
- La programmation générative décrit du code dont les entrées et les sorties sont des fragments de code: type, expression, programme.
- Elle introduit la notion d'**automatisation** dans le processus d'écriture de code.

La Méta-programmation

- Un méta-programme est un programme manipulant un programme.
- La méta-programmation est une des techniques fondamentales de la programmation générative.
- Elle n'est pas spécifique à C++, nous permettant de nous inspirer d'autres langages dans nos constructions.

Les *Traits*

Objectifs

- Fournir un protocole standard pour classifier les types
- Fournir un protocole standard pour générer des types
- Permettre de généraliser du code template

Exemple

- Détection de qualificateur; `&`, `*`, `const`
- Classification par propriétés fondamentales
- Génération de types sécurisée

Les *Traits*

Principe général

- Un traits = une structure template fournissant un type interne ou valeur constante
- Ce membre interne renvoie un type ou une valeur constante répondant à une question précise
- Accessible via `#include <type_traits>`

```
1  template<typename T, typename U>
2  auto f(T t, U u)
3  {
4    if( std::is_same<T,U>::value )  return t+u;
5    else                            return typename std::common_type<T,U>::type{t};
6  }
```

Les *Traits*

Principe général

- Un traits = une structure template fournissant un type interne ou valeur constante
- Ce membre interne renvoie un type ou une valeur constante répondant à une question précise
- Accessible via `#include <type_traits>`

```
1  template<typename T> auto f(T t)
2  {
3    // Arrete la compilation si la condition est fausse
4    static_assert ( !std::is_pointer<T>::value
5                  , "Cette fonction ne fonctionne pas avec des pointeurs"
6                  );
7
8    return &t;
9  }
```

Stratégie d'implémentation

Spécialisation partielle

```
template<typename T>
struct is_pointer : std::false_type {};

template<typename T>
struct is_pointer<T*> : std::true_type {};
```

Spécialisation totale

```
template<typename T>
struct is_void : std::false_type {};

template<>
struct is_void<void> : std::true_type {};
```

Stratégie d'implémentation

Surcharge de fonction

- Utilisation des propriétés de `decltype` pour forcer le choix entre deux surcharge de fonction
- `decltype` est ensuite réutilisé pour calculer le type de retour de la fonction sélectionnée

```
1  template<typename T> struct is_streamable
2  {
3    template<typename U>
4    static auto test(int) → decltype ( std::cout << std::declval<U>()
5                                     , std::true_type{}
6                                     );
7
8    template<typename> static std::false_type test(...);
9
10   static const bool value = decltype(test<T>(0))::value;
11 };
```

Stratégie d'implémentation

Détection par `decltype`

- Utilisation des propriétés de `decltype` pour générer un type invalide si une expression est invalide
- `std::void_t` permet de détecter ces appels incorrects et de générer `void` sinon

```
1  template<typename T, typename Enable = void>
2  struct is_streamable : std::false_type
3  {};
4
5  template<typename T>
6  struct  is_streamable<T, std::void_t<decltype(std::cout << std::declval<T>())>
7        : std::true_type
8  {};
```

Concepts

Challenge

- La syntaxe des *Traits* est complexe y compris dans des cas simples.
- Certaines propriétés ne peuvent pas être exprimées comme un motif de type.
- Utilisation complexe de la surcharge de fonction.

Solution

- Proposer une syntaxe permettant d'exprimer simplement la notion d' **interface syntaxique**.
- Intégration dans le processus de spécification de composants génériques.
- Support de la notion de raffinement.

Quels types de Contraintes ?

- Contraintes booléennes.
- Contraintes `requires`.

Contraintes booléennes

```
1  template<typename T>
2  concept small_type = sizeof(T) < 4;
3
4  bool bs = small_type<char>;     // is true
5  bool bl = small_type<double>;   // is false
```

Concepts

Contraintes par pré-requis

```
template<typename T>
concept arithmetic = requires(T a, T b)
{
  { a + b };
  { a - b };
  { a * b };
  { a / b };
};
```

Concepts

Contraintes par pré-requis

```
template<typename T>
concept arithmetic = requires(T a, T b)
{
  { a + b } -> std::same_as<T>;
  { a - b } -> std::same_as<T>;
  { a * b } -> std::same_as<T>;
  { a / b } -> std::same_as<T>;
};
```

Concepts

Raffinement et subsumption

```
1  template<typename T> concept term = requires(T a, T b)
2  {
3    { a + b } -> std::same_as<T>;
4    { a - b } -> std::same_as<T>;
5  };
6
7  template<typename T> concept factor = requires(T a, T b)
8  {
9    { a * b } -> std::same_as<T>;
10   { a / b } -> std::same_as<T>;
11 };
12
13 template<typename T>
14 concept arithmetic = term<T> && factor<T>;
```

Cas d'Usage : Contraintes de fonctions génériques

```
template<arithmetic T> auto compute(T a, T b)
{
  return (a+b) / (a-b);
}

template<term T> auto compute(T a, T b)
{
  return -(a+b);
}

template<factor T> auto compute(T a, T b)
{
  return (a*a*a*a)/(b*b);
}
```

Cas d'Usage : Contraintes de fonctions génériques

```
auto compute(arithmetic auto a, arithmetic auto b)
{
  return (a+b) / (a-b);
}

auto compute(term auto a, term auto b)
{
  return -(a+b);
}

auto compute(factor auto a, factor auto b)
{
  return (a*a*a*a)/(b*b);
}
```

Calculs à la Compilation

Calcul à la Compilation

Implémentation par *template*

```cpp
#include <iostream>
#include <array>

template<int N> struct factorial
{
  static const int value = N * factorial<N-1>::value;
};

template<> struct factorial<0> { static const int value = 1; };

int main()
{
  std::array<int, factorial<7>::value> x;
  std::cout << x.size() << "\n";
}
```

Calcul à la Compilation

Fonction `constexpr`

```cpp
#include <iostream>
#include <array>

constexpr int factorial(int n)
{
  int r = 1;
  for(int i=1;i⩽n;i++) r *= i;
  return r;
}

int main()
{
  std::array<int, factorial(7)> x;
  std::cout << x.size() << "\n";
}
```

Réplication de code

Objectifs

- Dans notre optique d'automatisation, répéter du code de manière contrôlée est une opération de base.
- Besoin d'exprimer la notion de fragment de code.
- Besoin de pouvoir gérer la répétition.

Mise en place

- Fragment de code = Fonction anonyme
- Répétition = templates variadiques
- Contrôle = séquence statique d'indexage

Réplication de code

Exemple: `static_for`

```
1  // Déroule le remplissage du tableau a
2  std::array<int,10> a;
3
4  static_for<0,10>( [&](auto index) { a[index] = 1+index;} )
```

Réplication de code

Exemple: `static_for`

```
template<int Begin, int End, typename Func>
void static_for(Func f)
{
  // Il faut répliquer f(i) avec i allant de Begin à End exclu
}
```

Réplication de code

Exemple: `static_for`

- `std::make_index_sequence` permet de générer une liste variadique d'entier.
- Cette liste d'entier se capture avec `std::index_sequence`.

```
1  template<int... I, typename Func>
2  void static_for(Func f, std::index_sequence<I...>)
3  {
4    // Que faire ici
5  }
6
7  template<int Begin, int End, typename Func>
8  void static_for(Func f)
9  {
10   static_for(f, std::make_index_sequence<End-Begin>{});
11 }
```

Réplication de code

Exemple: `static_for`

- La présence d'un variadic nous permet d'utiliser `...` dans une *fold expression*.
- L'opérateur binaire utilisé est l'opérateur de séquencement: `operator,`.

```
1  template<int... I, typename Func>
2  void static_for(Func f, std::index_sequence<I...>)
3  {
4    (f(I), ...);
5  }
6
7  template<int Begin, int End, typename Func>
8  void static_for(Func f)
9  {
10   static_for(f, std::make_index_sequence<End-Begin>{});
11 }
```

Génération conditionnelle de code

Objectifs

- L'étape suivante est la génération de code sous condition.
- Choisir ou faire disparaître du code via une condition statique.
- Répétition + Choix = Meta-langage Turing Complet.

Outils

- SFINAE
- *Tag Dispatch*
- `if constexpr`

Génération conditionnelle de code

Génération conditionnelle de code : SFINAE

- Si une fonction template échoue à être définie, elle est éliminée.
- `std::enable_if` permet de contrôler cet échec.
- **Rendu plus ou moins obsolète par `requires`**

```
1  #include <type_traits>
2
3  template<typename T>
4  std::enable_if<std::is_trivially_copyable_v<T>> copy(T const* src, T* dst, int n)
5  {
6    std::memcpy(dst,src,sizeof(T)*n);
7  }
8
9  template<typename T>
10 std::enable_if<!std::is_trivially_copyable_v<T>> copy(T const* src, T* dst, int n)
11 {
12   for(int i = 0;i<n;++i) dst[i] = src[i];
13 }
```

Génération conditionnelle de code : Tag

```cpp
#include <type_traits>

template<typename T> void copy(T const* src, T* dst, int n, std::true_type)
{
  std::memcpy(dst,src,sizeof(T)*n);
}

template<typename T> void copy(T const* src, T* dst, int n, std::false_type)
{
  for(int i = 0;i<n;++i) dst[i] = src[i];
}

template<typename T> void copy(T const* src, T* dst, int n)
{
  copy(src,dst,n, std::is_trivially_copyable_t<T>{});
}
```

Génération conditionnelle de code

Génération conditionnelle de code : if constexpr

- `if constexpr` rend caduque une branche de code à la compilation
- Rapide à compiler.
- Ressemble à du code *runtime*.

```
#include <type_traits>

template<typename T> void copy(T const* src, T* dst, int n, std::true_type)
{
  if constexpr(std::is_trivially_copyable_v<T>)
  {
    std::memcpy(dst,src,sizeof(T)*n);
  }
  else
  {
    for(int i = 0;i<n;++i) dst[i] = src[i];
  }
}
```

Merci pour votre Attention