

Programmation C++ Avancée

Session 7 – Méta-programmation

Joel Falcou Guillaume Melquiond

Laboratoire de Recherche en Informatique

Les templates en C++11/14

Nouvelles fonctionnalités

- La fin du cauchemar des >>
- Les *Traits*
- Assertion à la compilation
- Notion d'expression constante
- Maîtrise de la SFINAE

Traits

Objectifs

- Introspection limitée sur les propriétés des types
- Génération de nouveaux types
- Outils pour la spécialisation avancée
- Notion de **méta-fonction** : fonction manipulant et retournant un type

Traits

Introspection

- Classification des types selon leur sémantique
- Vérification d'existence d'une interface donnée
- Récupération d'informations structurelles

Exemple

```
#include <type_traits>

int main()
{
    std::cout << std::is_same<float,int>::value << '\n';
    std::cout << std::is_convertible<float,int>::value << '\n';
    std::cout << std::is_base_of<std::istream,std::ifstream>::value << '\n';
    std::cout << std::is_class<std::vector<int>>::value << '\n';
    std::cout << std::is_constructible<std::string,char*>::value << '\n';
    std::cout << std::is_polymorphic<std::istream>::value << '\n';
    std::cout << std::is_pointer<void*>::value << '\n';
}
```

Traits

Générateur de type

- Manipulation sûre des qualificateurs
- Création de types vérifiant certaines propriétés

Exemple

```
#include <memory>
#include <type_traits>

int main()
{
    int i;
    std::add_pointer<int>::type pi = &i;
    std::add_rvalue_reference<int>::type rri = std::forward<int>(i);
}
```

Traits – Application

```
#include <cstring>
#include <type_traits>

template<bool B> using bool_ = std::integral_constant<bool,B>;

template<typename T> void copy(T &dst, T const &src, bool_<true> const &)
{
    std::memcpy(&dst, &src, sizeof(T));
}

template<typename T> void copy(T &dst, T const &src, bool_<false> const &)
{
    dst = src;
}

template<typename T> void copy(T &dst, T const &src)
{
    typename std::is_trivially_copyable<T>::type select;
    copy(dst, src, select);
}
```

static_assert

Objectifs

- assert vérifie l'état logique d'un programme à l'exécution
- Comment vérifier l'état logique à la compilation ?
- Émission de messages d'erreur spécifiques
- Interaction avec les *Traits*

Exemple

```
#include <type_traits>

template<typename T> T factorial(T n)
{
    static_assert(std::is_integral<T>::value, "factorial requires integral parameter");
    return n < 2 ? 1 : n * factorial(n - 1);
}
```

Expressions constantes

Objectifs

- Simplifier le développement de méta-fonctions numériques
- Syntaxe homogène aux fonctions classiques
- Utilisable dans les contextes requérant une constante

Exemples

```
#include <iostream>

int factorial(int n) { return n < 2 ? 1 : n * factorial(n - 1); }
template<int N> void display() { std::cout << N << '\n'; }

int main()
{
    std::cout << factorial(8) << '\n';
    display<factorial(5)>();
    // error: call to non-constexpr function 'int factorial(int)'
}
```


Expressions constantes

Objectifs

- Simplifier le développement de méta-fonctions numériques
- Syntaxe homogène aux fonctions classiques
- Utilisable dans les contextes requérant une constante

Exemples

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

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

unsigned n = factorial<5>::value;
```

Expressions constantes

Objectifs

- Simplifier le développement de méta-fonctions numériques
- Syntaxe homogène aux fonctions classiques
- Utilisable dans les contextes requérant une constante

Exemples

```
#include <iostream>

constexpr int factorial(int n) { return n < 2 ? 1 : n * factorial(n - 1); }
template<int N> void display() { std::cout << N << '\n'; }

int main() {
    display<factorial(5)>();

    int x;
    std::cin >> x;
    std::cout << factorial(x) << '\n';
}
```

Expressions constantes

Objectifs

- Simplifier le développement de méta-fonctions numériques
- Syntaxe homogène aux fonctions classiques
- Utilisable dans les contextes requérant une constante

Exemples

```
#include <stdexcept>

constexpr int factorial(int n) {
    if (n < 0) throw std::out_of_range("");
    else return n < 2 ? 1 : n * factorial(n - 1);
}

int main() {
    constexpr int f = factorial(5); // OK
    constexpr int g = factorial(-1);
    // error: expression <throw-expression> is not a constant-expression
}
```

Maîtrise de la SFINAE

Qu'est-ce que la SFINAE ?

- Lors de la résolution de la surcharge de fonctions, il se peut qu'une surcharge instancie une fonction template
- L'instanciation du type de la fonction peut échouer
- Au lieu d'émettre une erreur, la surcharge est ignorée
- SFINAE = Substitution Failure Is Not An Error

Intérêts

- Contrôle fin de la surcharge de fonctions ou de classes templates
- Interaction avec les *Traits*
- `std::enable_if`

Maîtrise de la SFINAE

```
#include <tuple>
#include <iostream>

template <size_t n, typename... T>
typename std::enable_if<(n >= sizeof...(T)), void>::type
print_tuple(std::ostream&, const std::tuple<T...>&) {}

template <size_t n, typename... T>
typename std::enable_if<(n < sizeof...(T)), void>::type
print_tuple(std::ostream& os, const std::tuple<T...>& tup) {
    if (n != 0) os << ", ";
    os << std::get<n>(tup);
    print_tuple<n+1>(os, tup);
}

template <typename... T>
std::ostream& operator<<(std::ostream& os, const std::tuple<T...>& tup) {
    os << '[';
    print_tuple<0>(os, tup);
    return os << ']';
}

int main() {
    std::cout << std::make_tuple(0, 0.5, "abc") << '\n';
}
```

Tag Dispatching

Limitation de la SFINAE

- Les conditions d'exclusion doivent être disjointes
- Difficile à étendre hors des fonctions
- La compilation est en $O(N)$ avec le nombre de cas

Tag Dispatching

- Catégorisation des types selon des familles de propriétés
- Facile à étendre : une famille = un type
- La sélection se fait via la surcharge normale

Tag Dispatching – `std::advance`

```
namespace std
{
    struct input_iterator_tag {};
    struct bidirectional_iterator_tag : input_iterator_tag {};
    struct random_access_iterator_tag
        : bidirectional_iterator_tag {};
}
```

Tag Dispatching – std::advance

```
namespace std
{
    namespace detail
    {
        template <typename InputIterator, typename Distance>
        void advance_dispatch( InputIterator &i
                               , Distance n
                               , input_iterator_tag const &
                               )
        {
            assert(n >= 0);
            while (n-- > 0) ++i;
        }
    }
}
```


Tag Dispatching – `std::advance`

```
namespace std
{
    namespace detail
    {
        template <typename BidirectionalIterator, typename Distance>
        void advance_dispatch( BidirectionalIterator &i
                               , Distance n
                               , bidirectional_iterator_tag const &
                               )
        {
            if (n >= 0)
                while (n-->0) ++i;
            else
                while (n++<0) --i;
        }
    }
}
```

Tag Dispatching – std::advance

```
namespace std
{
    namespace detail
    {
        template <typename RandomAccessIterator, typename Distance>
        void advance_dispatch( RandomAccessIterator &i
                               , Distance n
                               , random_access_iterator_tag const &
                               )
        {
            i += n;
        }
    }
}
```

Tag Dispatching – `std::advance`

```
namespace std
{
    template <typename InputIterator, typename Distance>
    void advance(InputIterator &i, Distance n)
    {
        typename iterator_traits<InputIterator>::iterator_category category;
        detail::advance_dispatch(i, n, category);
    }
}
```