C++ 메타 프로그래밍과 constexpr

김경진
Astersoft
Microsoft MVP(Visual C++)

Agenda

메타 프로그래밍

템플릿 메타 프로그래밍

constexpr

1

메타 프로그래밍

메타 프로그래밍

meta-

<‘초월한’>

<‘~의 범주를 넘어서는’>

메타 프로그래밍

- metaphysics

  일반적 물리학 범주를 넘어서는 학문

  → 형이상학

- metaprogramming

  일반적 프로그래밍의 범주를 넘어서는 프로그래밍

  → 기존 프로그래밍과는 다른 무언가를 의미

메타 프로그래밍

일반적 프로그래밍	**사용자의 데이터를** 처리하고 가공하는 프로그램을 만드는 것
메타 프로그래밍	**프로그램을 데이터로** 처리하고 가공하는 프로그램을 만드는 것

예) C++ 컴파일러, YACC 파서 생성기

C++ 메타 프로그래밍

일반적 프로그래밍	**런타임에** 수행할 작업을 프로그래밍 하는 것
메타 프로그래밍	**컴파일 타임에** 수행할 작업을 프로그래밍 하는 것

Why C++ Metaprogramming?

- 런타임에 수행할 작업을 컴파일 타임에 미리 수행하여 상수화
  → 컴파일 시간은 다소 늘어나지만 런타임 퍼포먼스는 증가

- 계산 결과가 언어와 더 밀접하게 상호작용 가능
  → 계산 결과를 배열의 크기와 같은 상수에 사용 가능

2

템플릿 메타프로그래밍

템플릿 메타프로그래밍의 탄생

- 1994년 Erwin Unruh의 우연한 발견
- 템플릿을 이용한 미완의 소수 계산 코드 구현

템플릿 메타프로그래밍의 탄생

```
Error "primes.cpp",L16/C63(#416): prim | Type `enum{}' can't be converted to txpe `D<2>' ("primes.cpp",L2/C25).
Error "primes.cpp",L11/C25(#416): prim | Type `enum{}' can't be converted to txpe `D<3>' ("primes.cpp",L2/C25).
Error "primes.cpp",L11/C25(#416): prim | Type `enum{}' can't be converted to txpe `D<5>' ("primes.cpp",L2/C25).
Error "primes.cpp",L11/C25(#416): prim | Type `enum{}' can't be converted to txpe `D<7>' ("primes.cpp",L2/C25).
Error "primes.cpp",L11/C25(#416): prim | Type `enum{}' can't be converted to txpe `D<11>' ("primes.cpp",L2/C25).
Error "primes.cpp",L11/C25(#416): prim | Type `enum{}' can't be converted to txpe `D<13>' ("primes.cpp",L2/C25).
Error "primes.cpp",L11/C25(#416): prim | Type `enum{}' can't be converted to txpe `D<17>' ("primes.cpp",L2/C25).
Error "primes.cpp",L11/C25(#416): prim | Type `enum{}' can't be converted to txpe `D<19>' ("primes.cpp",L2/C25).
Error "primes.cpp",L11/C25(#416): prim | Type `enum{}' can't be converted to txpe `D<23>' ("primes.cpp",L2/C25).
Error "primes.cpp",L11/C25(#416): prim | Type `enum{}' can't be converted to txpe `D<29>' ("primes.cpp",L2/C25).
```

- 런타임이 아닌 컴파일 타임에 무언가를 계산할 수 있다는 가능성을 보여줌

타입 다루기

- 템플릿은 특정 타입에 의존하지 않고 재사용성을 높이기 위한 목적으로 만들어짐

- 템플릿 인자: 타입

  → template <typemame T>

타입 다루기

Demo

is_same_type

```
template <typename T1, typename T2>
struct is_same_type
{
    enum { value = false };
};

template <typename T>
struct is_same_type<T, T>
{
    enum { value = true };
};

int main()
{
    cout << is_same_type<int, double>::value << endl;
    cout << is_same_type<int, int>::value << endl;
}
```

```
0
1
```

type_traits

Primary Type	Composite Type	Type Properties	Type Relationships
is_null_pointer	is_fundamental	is_const	is_same
is_integral	is_arithmetic	is_volatile	is_base_of
is_floating_point	is_scalar	is_trivial	is_convertible
is_array	is_object	is_trivially_copyable	is_invocable
is_enum	is_compound	is_standard_layout	is_nothrow_invocable
is_union	is_reference	is_pod	
is_function	is_member_pointer	is_literal_type	
is_pointer		is_empty	
is_lvalue_reference		is_polymorphic	
…		…	

type_traits 활용 예

```
template <typename T>
void foo(const T& data)
{
    static_assert(std::is_integral<T>::value,
        "Template argument must be a integral type.");
}

int main()
{
    foo(1.5);  // Compile Error
}
```

```
error C2338: Template argument must be a integral type.
```

데이터 다루기

- 템플릿 인자에는 타입뿐만 아니라 데이터 역시 사용이 가능함

- 템플릿 인자: 데이터

  → template <int val>

데이터 다루기

Demo

add / substract

```
template <int left, int right>
struct add
{
    enum { value = left + right };
};

template <int left, int right>
struct substract
{
    enum { value = left - right };
};

int main()
{
    cout << add<10, 20>::value << endl;
    cout << substract<10, 20>::value << endl;
}
```

```
30
-10
```

factorial

```
template <int val>
struct factorial
{
    enum { value = val * factorial<val - 1>::value };
};

template <>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
    cout << factorial<10>::value << endl;
}
```

```
3628800
```

fibonacci_number

```
template <int val>
struct finonacci_number
{
    enum { value = finonacci_number<val - 1>::value + finonacci_number<val - 2>::value };
};

template <>
struct finonacci_number<0>
{
    enum { value = 0 };
};

template <>
struct finonacci_number<1>
{
    enum { value = 1 };
};

int main()
{
    cout << finonacci_number<10>::value << endl;
}
```

```
55
```

is_prime

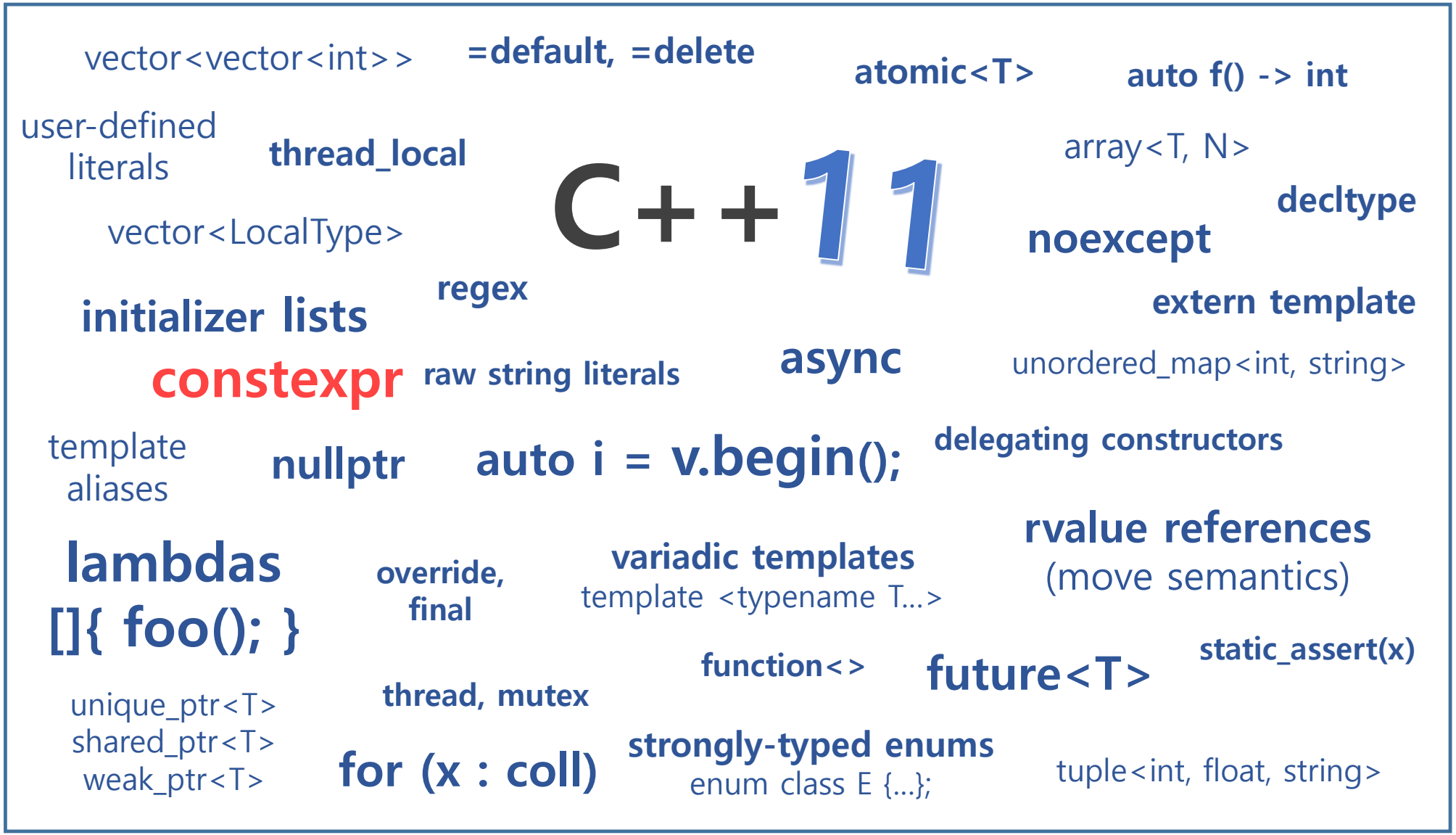
```
struct false_type
{
    typedef false_type type;
    enum { value = 0 };
};

struct true_type
{
    typedef true_type type;
    enum { value = 1 };
};

template<bool condition, class T, class U>
struct if_
{
    typedef U type;
};

template <class T, class U>
struct if_ < true, T, U >
{
    typedef T type;
};
```

```
template<size_t N, size_t c>
struct is_prime_impl {
    typedef typename if_<(c*c > N),
    true_type,
    typename if_ < (N % c == 0),
    false_type,
    is_prime_impl<N, c + 1> > ::type > ::type type;
    enum { value = type::value };
};

template<size_t N>
struct is_prime {
    enum { value = is_prime_impl<N, 2>::type::value };
};

template <>
struct is_prime <0> {
    enum { value = 0 };
};

template <>
struct is_prime <1> {
    enum { value = 0 };
};
```

템플릿 메타프로그래밍의 한계

- 컴파일 타임 계산만 가능
- 가독성이 매우 낮음

constexpr


vector<vector<int>>
=default, =delete
atomic<T>
auto f() -> int
user-defined literals
thread_local
C++11
array<T, N>
vector<LocalType>
decltype
noexcept
regex
initializer lists
extern template
constexpr
raw string literals
async
unordered_map<int, string>
template aliases
nullptr
auto i = v.begin();
delegating constructors
lambdas
[]{ foo(); }
override, final
variadic templates
template <typename T...>
rvalue references
(move semantics)
function<>
future<T>
static_assert(x)
unique_ptr<T>
shared_ptr<T>
weak_ptr<T>
thread, mutex
for (x : coll)
strongly-typed enums
enum class E {...};
tuple<int, float, string>

constexpr specifier

- const, static 과 용법이 같은 한정자 (변수, 함수에 사용)
- '컴파일 타임에 값을 도출하겠다' 라는 의미를 부여
- C++ 메타 프로그래밍을 문법 차원에서 지원

constexpr 변수

- ‘변수의 값을 컴파일 타임에 결정하여 상수화 하겠다’ 라는 의미
- 반드시 상수 식으로 초기화 되어야 함

```
constexpr int n = 0;                    // OK
constexpr int m = std::time(NULL);      // error C2127
```

constexpr 함수

- ‘함수 파라미터에 상수식이 전달될 경우, 함수 내용을 컴파일 타임에 처리하겠다’ 라는 의미

```
constexpr int square(int x) {
    return x * x;
}
```

- 전달되는 파라미터에 따라 컴파일타임, 런타임 처리가 결정됨

```
int n;
std::cin >> n;

square(10);  // 컴파일 타임 처리
square(n);   // 런타임 처리
```

constexpr 함수 제한 조건

- 함수 내에서는 하나의 표현식만 사용할 수 있으며, 반드시 리터럴 타입을 반환해야 함

  * 리터럴 타입: 정수, 부동소수, 열거형, 포인터, 참조 등의 타입을 지칭

```
constexpr LiteralType func() { return expression; }
```

constexpr 함수로 변환

- if / else 구문 → 삼항 연산자 (x > y ? x : y)
- for / while 루프 → 재귀 호출
- 변수 선언 → 파라미터 전달

constexpr을 이용한 메타 프로그래밍

Demo

factorial

```
constexpr int factorial(int n)
{
    return n == 0 ? 1 : n * factorial(n - 1);
}

int main()
{
    int n;
    std::cin >> n;

    constexpr int c_result = factorial(10);
    int r_result = factorial(n);
}
```

fibonacci_number

```
constexpr int fibonacci_number(int n)
{
    return n <= 1 ? n : fibonacci_number(n - 1) + fibonacci_number(n - 2);
}

int main()
{
    int n;
    std::cin >> n;

    constexpr int c_result = fibonacci_number(10);
    int r_result = fibonacci_number(n);
}
```

is_prime

```
constexpr bool is_prime(int n, int i = 2)
{
    return n <= 1 ? false :
        i * i > n ? true :
        n % i == 0 ? false :
        is_prime(n, i + 1);
}

int main()
{
    int n;
    std::cin >> n;

    constexpr bool c_result = is_prime(13);
    bool r_result = is_prime(n);
}
```

컴파일 타임 문자열 해시

```
constexpr unsigned int hash_code(const char* str)
{
    return str[0] ? static_cast<unsigned int>(str[0]) +
        0xEDB8832Full * hash_code(str + 1) : 8603;
}
```

컴파일 타임 문자열 해시

```
constexpr unsigned int hash_code(const char* str)
{
    return str[0] ? static_cast<unsigned int>(str[0]) +
        0xEDB8832Full * hash_code(str + 1) : 8603;
}

void foo(const char* name)
{
    switch (hash_code(name))
    {
        case hash_code("Kim"):
        break;

        case hash_code("Lee"):
        break;
    }
}
```

C++14 constexpr 제한 조건 완화

- 변수 선언 가능(static, thread_local 제외)
- if / switch 분기문 사용 가능
- range-based for 루프를 포함한 모든 반복문 사용 가능

is_prime C++14 ver.

```cpp
constexpr bool is_prime(int n)
{
    if (n <= 1)
        return false;

    for (int i = 2; i * i <= n; ++i)
    {
        if (n % i == 0)
            return false;
    }
    return true;
}
```

constexpr 관련 라이브러리

- Sprout C++ Libraries (Bolero MURAKAMI)
  http://bolero-murakami.github.io/Sprout/

- CEL constexpr Library (@sscrisk)
  https://github.com/sscrisk/CEL---ConstExpr-Library

Thank you!

참고 자료

- C++ Template Metaprogramming (David Abrahams, Aleksey Gurtovoy)
- http://blogs.embarcadero.com/jtembarcadero/2012/11/12/my-top-5-c11-language-and-library-features-countdown/
- http://en.cppreference.com/w/cpp/language/constexpr
- http://en.cppreference.com/w/cpp/language/constexpr
- http://blog.smartbear.com/c-plus-plus/using-constexpr-to-improve-security-performance-and-encapsulation-in-c/
- http://www.codeproject.com/Articles/417719/Constants-and-Constant-Expressions-in-Cplusplus
- http://cpptruths.blogspot.kr/2011/07/want-speed-use-constexpr-meta.html
- http://enki-tech.blogspot.kr/2012/09/c11-compile-time-calculator-with.html
- http://en.wikipedia.org/wiki/C%2B%2B14
- https://www.slideshare.net/emBO_Conference/programming-at-compile-time