Семестр 2. Лекция 10. Метапрограммирование. SFINAE. enable_if.

Евгений Линский

19 Мая 2017

Метапрограммирование

- Ранее мы, в основном, рассматривали обобщенное программирование (generic) в C++ — алгоритмы, написанные в стиле “типы будут заданы позже”.
- Метапрограммирование в C++ — это алгоритмы (операции), выполняемые во время компиляции программы.
  1. Обычная функция вычисляет значение в run-time
  2. Метафункция во время компиляции
     - вычисляет значение/константу (constexpr) или тип
     - выбирает тот или иной вариант алгоритма (ifdef)
     - генерирует алгоритм (variadic templates)

Метапрограммирование. Пример 1.

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

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

std::cout << Factorial<5>::value << std::endl;
```

Метапрограммирование. Пример 2.

```
float euclidean_baseline(int n, float* x, float* y){
  for(int i = 0; i < n; ++i){
    float num = x[i] - y[i];
    result += num * num;
  }
  ...
}
float euclidean_intrinsic(int n, float* x, float* y){
  __m128 euclidean = _mm_setzero_ps();
  for (; n>3; n-=4) {
     __m128 a = _mm_loadu_ps(x);
     __m128 b = _mm_loadu_ps(y);
     __m128 a_minus_b = _mm_sub_ps(a,b);
     __m128 a_minus_b_sq = _mm_mul_ps(a_minus_b, a_minus_b)
    euclidean = _mm_add_ps(euclidean, a_minus_b_sq);
    x+=4;
    y+=4;
  }
  ...
}
```

Метапрограммирование. Пример 2.

Если процессор поддерживает расширение команд SSE...

```
float euclidean(int dim, float* x, float* y){
  float
    (*euclidean)(int, float*, float*) = euclidean_baseline;

#ifdef __SSE__
  euclidean = euclidean_intrinsic;
#endif
  return euclidean(dim, x, y);
}
```

Прием SFINAE.

- Коротко: “Substitution Failure Is Not An Error”
- Длинно: “При выборе одной из перегрузок шаблонной функции варианты, вызывающие синтаксическую ошибку при подстановке шаблонного параметра, не вызывают ошибку компиляции, а исключаются из списка кандидатов на наиболее подходящую перегрузку”.

Синтаксическая ошибка при подстановке шаблонного параметра:

```
template <typename T>
void show(typename T::iterator x, typename T::iterator y){
    for (; x != y; ++x) cout << *x << endl;
}
show<int>(16, 18);
```

Компилятор скажет:

```
error: no matching function for call to 'show(int, int)'
note: candidate is:
note: template void show(typename T::iterator,
                         typename T::iterator)
```

Прием SFINAE.

```
template <typename T>
void show(typename T::iterator x, typename T::iterator y){
    for (; x != y; ++x) cout << *x << endl;
}
template <typename T>
void show(T a, T b)
{
    cout << a << "; " << b << endl;
}
show<int>(16, 18);
```

Компилятор скажет: OK.

Применение SFINAE.

- Хочу проверить, есть ли у типа итератор или нет
- Потом на основе этой проверки выбрать тот или другой вариант алгоритма (будет дальше)

```
template <typename T>
struct has_iterator {
    template <typename U>
    static char test(typename U::iterator* x);

    template <typename U>
    static long test(U* x);

    static bool value = sizeof(test<T>(0)) == 1;
};
//value = sizeof(test<int>(0)) == 1
std::cout << has_iterator<int>::value << std::endl;
std::cout << has_iterator<std::vector<int>>::value;
```

NB: выбирается первая, подходящая перегрузка.

Отступление про typename.

```
static char test(U::iterator* x);
```

Как трактовать *U::iterator* x*?

1. Указатель типа *U::iterator**?
2. Умножение статической переменной *U::iterator* на *x*?

static char test(typename U::iterator x);* уточняет, что это вариант 1.

enable_if (C++11).

enable_if: включает (enable) в "исходник" код функции, только если выполняется условие (if).

```
template <typename T>
typename enable_if<!has_iterator<T>::value, void>::type
  show(const T& x) {
    cout << x << endl;
}

template <typename T>
typename enable_if<has_iterator<T>::value, void>::type
  show(const T& x) {
    for (auto& i : x)
        cout << i << endl;
}
```

enable_if (реализация).

vector<string> s = {today, "is, "Friday"};
show(s);

- Вариант без итератора не будет включен в “исходник” => его не будет и в бинарнике.
- Если условие в enable_if верно (true), то ::type будет равен void
- Если условия неверно, то будет синтаксическая ошибка и это вариант функции будет исключен (SFINAE).

Реализация enable_if:

```
template<bool B, class T>
struct enable_if {};

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

enable_if (замена ifdef).

```
struct is_64_bit {
    static const bool value = sizeof(void*) == 8;
};

template<typename T = void>
typename std::enable_if<is_64_bit::value, T>::type
my_memcpy(void* target, const void* source, size_t n)
{
    std::cout << "64 bit memcpy" << std::endl;
}

template<typename T = void>
typename std::enable_if<!is_64_bit::value, T>::type
my_memcpy(void* target, const void* source, size_t n)
{
    std::cout << "32 bit memcpy" << std::endl;
}
```

NB: SFINAE работает только для шаблонных функций, поэтому мы искусственно вводит шаблонный параметр.

type_traits.

1. #include <type_traits>
2. Много метафункций в стиле нашего has_iterator (называются traits), которые можно использовать в enable_if (например: is_integral, is_pointer, is_copy_assignable)
   - traits — "выводит" (deduce) информацию о типе (часто реализованы с помощью специализации шаблонов для каждого типа)
   - Без traits можно сделать версию функцию f() для каждого примитивного типа (или один общий случай и варианты для нескольких типов с помощью специализации шаблонов)
   - С traits и enable_if можно сделать одну версию для целых типов, другую для вещественных типов