Метапрограммирование

Александр Смаль

CS центр
24 апреля 2018
Санкт-Петербург

Метапрограммирование

- Метапрограммированием называют создание программ, которые порождают другие программы.

Метапрограммирование

- Метапрограммированием называют создание программ, которые порождают другие программы.
- Шаблоны C++ можно рассматривать как функциональный язык для метапрограммирования.

Метапрограммирование

- Метапрограммированием называют создание программ, которые порождают другие программы.
- Шаблоны C++ можно рассматривать как функциональный язык для метапрограммирования.
- Метапрограммы C++ позволяют оперировать типами, шаблонами и значениями целочисленных типов.

Метапрограммирование

- Метапрограммированием называют создание программ, которые порождают другие программы.
- Шаблоны C++ можно рассматривать как функциональный язык для метапрограммирования.
- Метапрограммы C++ позволяют оперировать типами, шаблонами и значениями целочисленных типов.
- Метапрограммирование в C++ можно применять для широкого круга задач:
  - целочисленные compile-time вычисления,
  - compile-time проверка ошибок,
  - условная компиляция,
  - генеративное программирование,
  - …

Метапрограммирование

- Метапрограммированием называют создание программ, которые порождают другие программы.
- Шаблоны C++ можно рассматривать как функциональный язык для метапрограммирования.
- Метапрограммы C++ позволяют оперировать типами, шаблонами и значениями целочисленных типов.
- Метапрограммирование в C++ можно применять для широкого круга задач:
  - целочисленные compile-time вычисления,
  - compile-time проверка ошибок,
  - условная компиляция,
  - генеративное программирование,
  - …
- Для метапрограммирования существуют целые библиотеки, например, MPL из boost.

Метафункции

Метафункция — это шаблонный класс, который определяет имя типа `type` или целочисленную константу `value`.

- Аргументы метафункции — это аргументы шаблона.
- Возвращаемое значение — это `type` или `value`.

Метафункции могут возвращать типы:

```
template<typename T>
struct AddPointer
{
    using type = T *;
};
```

и значения целочисленных типов:

```
template<int N>
struct Square
{
    static int const value = N * N;
};
```

Вычисления в compile-time

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

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

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

(Это вычисление можно реализовать через `constexpr`.)

Определение списка

Шаблоны позволяют определять алгебраические типы данных.

```
// определяем список
template <typename ... Types>
struct TypeList;

// специализация по умолчанию
template <typename H, typename... T>
struct TypeList<H, T...>
{
    using Head = H;
    using Tail = TypeList<T...>;
};

// специализация для пустого списка
template <>
struct TypeList<> { };
```

Длина списка

```
// вычисление длины списка
template<typename TL>
struct Length
{
    static int const value = 1 +
        Length<typename TL::Tail>::value;
};

template<>
struct Length<TypeList<>>
{
    static int const value = 0;
};

int main()
{
    using TL = TypeList<double, float, int, char>;
    std::cout << Length<TL>::value << std::endl;
    return 0;
}
```

Операции со списком

```
// добавление элемента в начало списка
template<typename H, typename TL>
struct Cons;

template<typename H, typename... Types>
struct Cons<H, TypeList<Types...>>
{
    using type = TypeList<H, Types...>;
};

// конкатенация списков
template<typename TL1, typename TL2>
struct Concat;

template<typename... Ts1, typename... Ts2>
struct Concat<TypeList<Ts1...>, TypeList<Ts2...>>
{
    using type = TypeList<Ts1..., Ts2...>;
};
```

Вывод списка

```
// вывод списка в поток os
template<typename TL>
void printTypeList(std::ostream & os)
{
    os << typeid(typename TL::Head).name() << '\n';
    printTypeList<typename TL::Tail>(os);
};

// вывод пустого списка
template<>
void printTypeList<TypeList<>>(std::ostream & os) {}

int main()
{
    using TL = TypeList<double, float, int, char>;
    printTypeList<TL>(std::cout);
    return 0;
}
```

Генерация классов

```
struct A {
    void foo() {std::cout << "struct A\n";}
};
struct B {
    void foo() {std::cout << "struct B\n";}
};
struct C {
    void foo() {std::cout << "struct C\n";}
};

using Bases = TypeList<A, B, C>;
```

Генерация классов

```
struct A {
    void foo() {std::cout << "struct A\n";}
};
struct B {
    void foo() {std::cout << "struct B\n";}
};
struct C {
    void foo() {std::cout << "struct C\n";}
};

using Bases = TypeList<A, B, C>;
```

```
template<typename TL>
struct inherit;

template<typename... Types>
struct inherit<TypeList<Types...>> : Types... {};

struct D : inherit<Bases> { };
```

Генерация классов

```
struct D : inherit<Bases>
{
    void foo() { foo_impl<Bases>(); }

    template<typename L> void foo_impl();
};
template<typename L>
inline void D::foo_impl()
{
    // приводим this к указателю на базу из списка
    static_cast<typename L::Head *>(this)->foo();

    // рекурсивный вызов для хвоста списка
    foo_impl<typename L::Tail>();
}
template<>
inline void D::foo_impl<TypeList<>>()
{
}
```

Как определить наличие метода?

```
struct A {  void foo() { std::cout << "struct A\n"; } };
struct B {  }; // нет метода foo()
struct C {  void foo() { std::cout << "struct C\n"; } };
```

```
template<typename L>
inline void D::foo_impl()
{
    // приводим this к указателю на базу из списка
    static_cast<typename L::Head *>(this)->foo();

    // рекурсивный вызов для хвоста списка
    foo_impl<typename L::Tail>();
}
```

Как проверить наличие родственных связей?

```
typedef char YES;
struct NO { YES m[2]; };

template<class B, class D>
struct is_base_of
{
    static YES test(B * );
    static NO  test(...);

    static bool const value =
        sizeof(YES) == sizeof(test((D *)0));
};

template<class D>
struct is_base_of<D, D>
{
    static bool const value = false;
};
```

SFINAE

SFINAE = Substitution Failure Is Not An Error.
Ошибка при подстановке шаблонных параметров не является сама по себе ошибкой.

```
// ожидает, что у типа T определён
// вложенный тип value_type
template<class T>
void foo(typename T::value_type * v);

// работает с любым типом
template<class T>
void foo(T t);
```

```
// при инстанциировании первой перегрузки
// происходит ошибка (у int нет value_type),
// но это не приводит к ошибке компиляции
foo<int>(0);
```

Используем SFINAE

```
template<class T>
struct is_foo_defined
{
    // обёртка, которая позволит проверить
    // наличие метода foo с заданой сигнатурой
    template<class Z, void (Z::*)() = &Z::foo>
    struct wrapper {};

    template<class C>
    static YES check(wrapper<C> * p);

    template<class C>
    static NO  check(...);

    static bool const value =
        sizeof(YES) == sizeof(check<T>(0));
};
```

Проверяем наличие метода

```
template<bool b>
struct Bool2Type          { using type = YES; };

template<>
struct Bool2Type<false> { using type = NO;  };
```

```
template<class L>
void foo_impl()
{
    using Head = typename L::Head;

    constexpr bool has_foo =
            is_foo_defined<Head>::value;

    using CALL =
        typename Bool2Type<has_foo>::type;

    call_foo<Head>(CALL());
    foo_impl<typename L::Tail>();
}
```

Проверяем наличие метода (продолжение)

```
struct D : inherit<Bases>
{
    // ... foo, foo_impl

    template<class Base>
    void call_foo(YES)
    {
        static_cast<Base *>(this)->foo();
    }

    template<class Base>
    void call_foo(NO) { }
};
```

std::enable_if

```
// <type_traits>
namespace std {
    template<bool B, class T = void>
    struct enable_if {};

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

```
template<class T>
typename std::enable_if<
    std::is_integral<T>::value, T>::type
    div2(T t) { return t >> 1; }

template<class T>
typename std::enable_if<
    std::is_floating_point<T>::value, T>::type
    div2(T t) { return t / 2.0; }
```

std::enable_if

```
template<class T>
T div2(T t, typename std::enable_if<
        std::is_integral<T>::value, T>::type * = 0)
{ return t >> 1; }

template<class T, class E = typename std::enable_if<
    std::is_floating_point<T>::value, T>::type>
T div2(T t)
{ return t / 2.0; }
```

```
template<class T, class E = void>
class A;

template<class T>
class A<T, typename std::enable_if<
           std::is_integral<T>::value>::type>
{};
```