Программирование на языке C++
Лекция 11

Метапрограммирование: основы

Александр Смаль

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;
}
```