академия больших данных

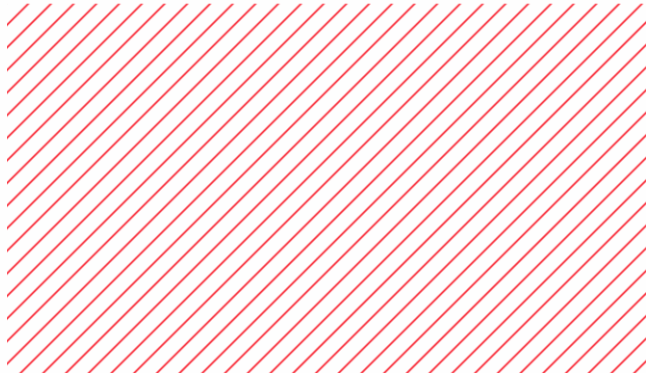
mail.ru group

Вывод типов и метапрограммирование

Лекция 8

Гуров Роман

Advanced C++



Проблема

Хотим записать в переменную значение итератора

```
template <class Iter>
void f(Iter iter) {
    typename std::iterator_traits<Iter>::value_type val = *iter;
```

Неудобно писать тип целиком, но ещё терпимо

Проблема

Попробуем просто проитерироваться по контейнеру:

```
for (std::unordered_map<MyType, std::string, std::hash<MyType>, MyTypeSpecialEqualityPred>::const_iterat
```

...

Range-based for

Предыдущий пример можно упростить синтаксисом из C++11

```
for (const std::pair<MyType, std::string>& x : map) {
```

Но и тут есть простор для ошибок:

value_type	std::pair<const Key, T>

Из-за забытого **`const`** объект будет неявно копироваться

Ключевое слово auto

Можно избежать явного указания типа
и позволить компилятору вывести его

```
template <class Iter>
void f(Iter iter) {
    auto val = *iter;
}
```

```
for (auto it = map.begin(); it != map.end(); ++it) {
```

```
for (const auto& x : map) {
```

Ключевое слово auto

auto можно также использовать для возвращаемого значения функции

```
auto f() {
    return 42;
}
```

```
template <class Iter>
auto& f(Iter iter) {
    return *iter;
}
```

Ключевое слово auto

Как определить, что подставится вместо **auto**?

```
const auto& = value;
```

Так же, как для шаблонных аргументов функции

```
template <typename T>
void f(const T& value);
```

std::initializer_list

В C++11 появилась возможность инициализировать
вектор перечислением элементов

```
std::vector<int> vec = {1, 2, 3};
```

Трактуется как
`std::initializer_list<int>`

Для этого реализуется отдельный конструктор:

```
vector(std::initializer_list<T> init);
```

`std::initializer_list` – легковесный объект, копирование которого
не вызывает копирования внутренних объектов

std::initializer_list

При помощи списков инициализации можно удобно работать с переменным числом аргументов, если они одного типа

```
template <typename... Args>
void h(const Args&... args) {
    for (auto& x : {args...}) {
        std::cout << x;
    }
}
```

std::initializer_list

Для списков инициализации поведение при выводе типов отличается

```
auto il = {1, 2, 3};      // OK


template <class T>
void g(T x);

g({1, 2, 3});             // not OK


template <class T>
void g(std::initializer_list<T> x);

g({1, 2, 3});             // OK
```

Ключевое слово auto

Что будет, если написать `auto&&`?

```
auto&& = value;
```

Так же, как и с аргументами функции, получится универсальная ссылка, способная принимать и rvalue, и lvalue

Ключевое слово decltype

decltype позволяет задать такой же тип, как у указанного объекта

```
std::vector<int> a = {1, 2};
decltype(a) b;
```

decltype обрабатывается на этапе компиляции

```
int a = 1;
decltype(a++) b;
a == 1;
```

Выражение используется только для вывода типа,
в программу оно не попадает

Ключевое слово decltype

decltype выводит тип по другим правилам:
отбрасывания ссылок не происходит, возвращается именно тот тип,
с которым был объявлен объект

```
int&& a = 1;
decltype(a) b = a;    // Ошибка компиляции
// ^ int&&
```

Выражения в decltype

Какой тип будет выведен, если передано выражение?

Вид выражения	**decltype**
lvalue	T&
xvalue	T&&
prvalue	T

```
int&& a = 1;
decltype(a++) b = a;
//       ^ int
```

```
int&& a = 1;
decltype((a)) b = a;
//       ^ int&
```

decltype и тернарный оператор

В тернарном операторе могут быть записаны выражения разных типов
Как на этапе компиляции определить тип?

```
decltype(a == b ? 10 : 20.5) c = 0;
```

На самом деле, тип возвращаемого значения всегда один и определяется списком правил

decltype и ссылки

На **decltype** можно навешивать амперсанды
по правилу сворачивания ссылок

```
int&& a = 1;
decltype(++a)&& b = a;
// ^ int&
```

Проверка выведенного типа

Как узнать, какой тип подставился на место выражения?
Можно вызвать намеренную ошибку компиляции и посмотреть

```
template <typename T>
struct C {
    C() = delete;
};

int&& a = 1;
C<decltype(a)> c;
```

```
error: use of deleted function 'C<T>::C() [with T = int&&]'
```

decltype(auto)

```
template <class T>
                getByIndex(T& cont, size_t i) {
    return cont[i];
}
```

Какой возвращаемый тип указать у такой функции?

decltype(auto)

```
template <class T>
auto& getByIndex(T& cont, size_t i) {
    return cont[i];
}
```

auto& не сработает для std::vector<**bool**>,
так как он возвращает rvalue

decltype(auto)

```
template <class T>
decltype(cont[i]) getByIndex(T& cont, size_t i) {
    return cont[i];
}
```

Хотим сделать так, но `cont[i]` недоступен
на момент объявления функции

decltype(auto)

```
template <class T>
decltype(auto) getByIndex(T& cont, size_t i) {
    return cont[i];
}
```

Выход – decltype(auto)

Compile-time вычисления

Попробуем реализовать compile-time **if**

```
template <bool b>
void If() {}

template <>
void If<true>() {f();};
```

```
If<x == 5>();
```

Пример

Можно посчитать числа Фибоначчи на этапе компиляции

```
template <int N>
struct Fibonacci {
    static const int value = Fibonacci<N - 2>::value + Fibonacci<N - 1>::value;
};

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

template <>
struct Fibonacci<1> {
    static const int value = 1;
};
```

Ключевое слово constexpr

Ключевое слово **`constexpr`** требует того, чтобы выражение было посчитано на этапе компиляции

```
constexpr int x = 5;
If<x == 5>();
```

Ключевое слово constexpr

constexpr применим и к функциям

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

constexpr не запрещает вызывать функцию
от не-**constexpr** параметров

```
fibonacci(10);   // compile-time
fibonacci(x);    // run-time
```

if constexpr

В C++17 добавили полноценный compile-time **if**

```
constexpr int x = 5;
if constexpr (x == 5) {
    f();
} else {
    g();
}
```

type_traits

Ранее уже упоминались некоторые функции из заголовочного файла `type_traits`:

`std::remove_reference`
`std::remove_const`

std::is_const

Позволяет узнать, является ли данный тип const

```
template <class T>
struct is_const {
    static constexpr bool value = false;
};

template <class T>
struct is_const<const T> {
    static constexpr bool value = true;
};
```

```
const int a = 1;
if (is_const<decltype(a)>::value) {}
```

std::true_type, std::false_type

Значения **true** и **false**, оформленные в виде типов

```
struct true_type {
    static constexpr bool value = true;
};

struct false_type {
    static constexpr bool value = false;
};
```

```
template <class T>
struct is_const : std::false_type {};

template <class T>
struct is_const<const T> : std::true_type {};
```

std::is_same

Как проверить на этапе компиляции равенство двух типов?

```
template <class U, class V>
struct is_same : std::false_type {};

template <class U>
struct is_same<U, U> : std::true_type {};
```

```
if (is_same<decltype(a), decltype(b)>::value) {}
```

std::conjunction

Логическое И для типов

```
template <class U, class V>
struct conjunction : false_type {};
template <>
struct conjunction<true_type, true_type> : true_type {};
```

Подобная реализация не сработает в таком случае:

```
if (conjunction<true_type, is_const<decltype(a)>>::value) {}
```

Можно исправить так:

```
template <class U, class V>
struct conjunction {
    static constexpr bool value = U::value && V::value;
};
```

std::conditional

Тернарный оператор для типов

```
template <bool Cond, class U, class V>
struct conditional {
    typedef U type;
};

template <class U, class V>
struct conditional<false, U, V> {
    typedef V type;
};
```

```
conditional<(q > 20), long long, int>::type x;
```

std::conjunction

`conjunction` в `std` поддерживает переменное число аргументов

```
template <class U1, class... Args>
struct conjunction<U1, Args...>
: conditional<U1::value, conjunction<Args...>, U1>::type {};

template <class U1>
struct conjunction<U1> : U1 {};
```

std::rank

Позволяет узнать размерность массива

```
template <class T>
struct rank {
    static constexpr size_t value = 0;
};
template <class T, size_t Size>
struct rank<T[Size]> {
    static constexpr size_t value = rank<T>::value + 1;
};
template <class T>
struct rank<T[]> {
    static constexpr size_t value = rank<T>::value + 1;
};
```

```
rank<int[4][6]>::value == 2
```