академия больших данных

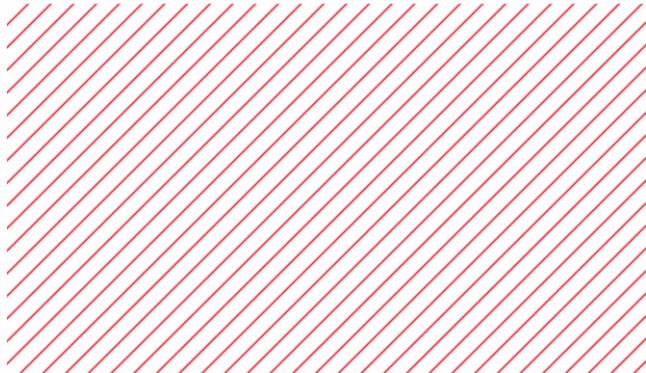
mail.ru group

Functional & Concepts

Лекция 10

Гуров Роман

Advanced C++



Мотивировка

Вспомним передачу своего компаратора в `std::sort`

```
struct Cmp {
    bool operator()(int a, int b) {
        return a > b;
    };
};
```

```
std::sort(vec.begin(), vec.end(), Cmp());
```

Создавать новую функцию или функтор для однострочного компаратора слишком долго и неудобно

Лямбда-функции

В C++11 ввели возможность объявлять анонимные функции

```
std::sort(vec.begin(), vec.end(),
          [](int a, int b) {return a > b;});
```

Данная конструкция называется замыканием

Лямбда-функции

В C++11 ввели возможность объявлять анонимные функции

```
std::sort(vec.begin(), vec.end(),
          [](int a, int b) {return a > b;});
```

Тело функции

Лямбда-функции

В C++11 ввели возможность объявлять анонимные функции

```
std::sort(vec.begin(), vec.end(),
          [](int a, int b) {return a > b;});
```

Принимаемые аргументы

Лямбда-функции

В C++11 ввели возможность объявлять анонимные функции

```
std::sort(vec.begin(), vec.end(),
          [](int a, int b) {return a > b;});
```

А это что?

Списки захвата

Пусть возникла необходимость отсортировать точки по удаленности от данной

```
Point c = {1., 2.};

std::sort(vec.begin(), vec.end(),
          [ ](Point a, Point b) {return distance(a, c) < distance(b, c);});
```

Нельзя использовать внутри лямбды переменную извне

Списки захвата

Пусть возникла необходимость отсортировать точки по удаленности от данной

```
Point c = {1., 2.};

std::sort(vec.begin(), vec.end(),
          [c](Point a, Point b) {return distance(a, c) < distance(b, c);});
```

Для этого нужно использовать список захвата

Списки захвата

Пусть возникла необходимость отсортировать точки по удаленности от данной

```
Point c = {1., 2.};

std::sort(vec.begin(), vec.end(),
          [c](Point a, Point b) {return distance(a, c) < distance(b, c);});
```

В данном случае происходит захват по значению

Списки захвата

Есть возможность захватывать переменные по ссылке

```
int num = 1;
```

```
[&num]() {num = 5;}
```

Специального синтаксиса для константной ссылки не предусмотрено

Естественно, можно захватить несколько локальных переменных:

```
[&ref, val, val2]() {num = 5;}
```

Особенность захвата по значению

В захвате по значению есть нюанс:

```
[num]() {num = 5;}
```

Захваченные по значению переменные нельзя изменять

Для изменяемости можно объявить всю лямбду **mutable**

```
[num]() mutable {num = 5;};   // OK
```

Но стоит помнить, что это значение будет
общим для всех вызовов функции (как **static**)

this в списке захвата

Внутри класса в лямбду можно захватить **this**

```
struct C {
    int x;

    void foo() {
        auto func = [this]() {std::cout << x;};
        // ...
    }
};
```

При этом внутри лямбда-функции поля класса будут доступны без явного использования **this**->, так же как внутри методов класса

Захват с присваиванием

А как сделать перемещающий захват?

В C++14 добавили возможность делать захват с присваиванием:

```
[obj_x2 = obj * 2]() {/* ... */}
```

При помощи такого синтаксиса можно явно вызвать перемещение

```
[obj = std::move(obj)]() {/* ... */}
```

И даже сделать захват по константной ссылке, если очень надо

```
[&obj = std::as_const(obj)]() {/* ... */}
```

Указание возвращаемого значения

По умолчанию возвращаемое значение определяется так, будто функция возвращает **auto**

Но его можно указать и явно:

```
[]() -> bool {return 1;};
```

```
[](int& a) -> int& {return a;}
```

Захват по умолчанию

Можно захватить сразу все локальные переменные

Захват всего по ссылке → `[&]() {/* ... */}`

Захват всего по значению → `[=]() {/* ... */}`

Но это считается плохой практикой и так делать не стоит

Захват по умолчанию

Рассмотрим такой класс:

```
struct C {
    int divisor;

    auto getFunction() {
        return [&](int n) {return n % divisor;};
    }
};
```

```
auto func = C{7}.getFunction();

std::cout << func(15);
```

Понятно, что при захвате по ссылке после уничтожения объекта поле станет недоступным и вызов лямбды будет undefined behavior

Захват по умолчанию

Но с захватом по значению то все должно быть хорошо?

```
struct C {
    int divisor;

    auto getFunction() {
        return [=](int n) {return n % divisor;};
    }
};
```

```
auto func = C{7}.getFunction();

std::cout << func(15);
```

Оказывается, что нет

Поля класса не являются локальными переменными вообще, а видно их в лямбде только потому, что захватывается **this**

Поэтому, лучше просто явно перечислять все, что требуется захватить

auto в лямбда-функциях

Лямбды позволяют использовать ключевое слово
auto в своих параметрах

```
std::sort(vec.begin(), vec.end(),
          [](const auto& a, const auto& b) {return a > b;});
```

В таком случае `operator()` у лямбды будет шаблонным

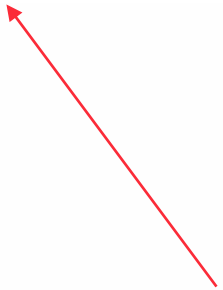
Также, можно использовать в возвращаемом значении
decltype от аргументов

```
[](auto& a) -> decltype(a) {return a;}
```

Прямая передача

А что делать, если принимаем `auto&&` и хотим передать дальше?

```
[](auto&& a) {g(std::forward<T>(a));}
```



Что написать сюда?
В лямбде нет шаблонного параметра

Прямая передача

А что делать, если принимаем **auto**&& и хотим передать дальше?

```
[](auto&& a) {g(std::forward<T>(a));}
```

В этом поможет **decltype**

```
[](auto&& a) {g(std::forward<decltype(a)>(a));}
```

std::function

При передаче функции в другую функцию приходилось писать подобные конструкции:

```
template <class Func>
void f(Func function) {
    function();
}
```

Но это никак не ограничивает передаваемый тип, а C-style указатели на функции не работают с функторами

std::function

Для этого существует класс `std::function`, позволяющий хранить и вызывать сущности, у которых есть `operator()`

```
void print(int n) {
    std::cout << n << std::endl;
}
```

```
std::function<void(int)> func_print = print;
func_print(-2);
```

В него можно передать и лямбда-функцию

```
std::function<void()> func_lambda_print = [] { print(100); };
func_lambda_print();
```

std::function

Чтобы принимать шаблонные параметры в таком виде, можно сделать специализацию

```
template <class Unused>
struct function;

template <class ReturnType, class... Args>
struct function<ReturnType (Args...)> {
    // ...
};
```

std::function

Для реализации полезно знать о способе хранения функций

```
struct callable_base {
    virtual int operator()(double d) = 0;
};

template <typename F>
struct callable : callable_base {
    F functor;
    explicit callable(F functor) : functor(functor) {}
    int operator()(double d) override { return functor(d); }
};

template <class Func>
void accept_function(Func f) {
    std::unique_ptr<callable_base> f_ptr = std::make_unique<callable<Func>>(f);
    // ...
}
```

Таким методом можно хранить любые объекты, не привязываясь к их изначальному типу

std::bind

Позволяет сделать вокруг функции обёртку, фиксирующую некоторые аргументы

```
using namespace std::placeholders;

auto f = [](auto a, auto b, auto c) {return a + b * c;};

int n = 3;
auto f1 = std::bind(f, n, 2, _1);
```

```
f1(5);
// Вызовется f(3, 2, 5)
```

_1 – так называемый placeholder, на место которого подставится аргумент функции, которую вернёт `bind`

Placeholder'ы работают и для вложенных `bind`'ов

```
auto g = [](auto a) {return a * a;};

auto f2 = std::bind(f, _2, std::bind(g, _2), _1);
```

```
f2(4, 3);
// Вызовется f(3, g(3), 4)
```

std::invoke

Позволяет универсальным образом вызывать функции

```
auto bar(int v1, int v2) {
    std::cout << v1 << ' ' << v2;
}
```

```
std::invoke(bar, 1, 2);
```

Для указателей на метод класса семантика вызова отличается от обычной

```
class Foo {
    int m_v {1};
public:
    void foo(int v1) {
        std::cout << m_v << ' ' << v1;
    }
};
```

```
Foo foo;
auto m_ptr = &Foo::foo;
(foo.*m_ptr)(2);
```

Но `std::invoke` позволяет работать с ними через круглые скобки

```
std::invoke(&Foo::foo, foo, 2);
std::invoke(&Foo::foo, &foo, 2);
```

std::invoke_result

Позволяет получить возвращаемый тип invoke

```
struct Foo {
    auto operator()(int v1, int v2) {
        return v1 + v2;
    }
};
```

```
std::invoke_result_t<Foo, int, int> a;
std::invoke_result_t<decltype(&Foo::operator()), Foo, int, int> b;
```

Важно то, что при невозможности сделать invoke с такими аргументами,
у `invoke_result` просто не будет определен `::type`,
ошибки компиляции при этом не произойдет

Рефлексия в C++

В C++ можно попытаться реализовать руками подобие рефлексии из более высокоуровневых языков

Данная функция возвращает количество полей у структуры

```
template <size_t I>
struct ubiq_constructor {
    template <typename Type>
    constexpr operator Type&() const noexcept;
};


template <class T, size_t I0, size_t ... I>
constexpr auto fields_count(const std::index_sequence<I0, I...>&)
-> decltype(T{ ubiq_constructor<I0>(), ubiq_constructor<I>()...}, size_t()) {
    return sizeof...(I) + 1;
}

template <class T, size_t ... I>
constexpr auto fields_count(const std::index_sequence<I...>&) {
    return fields_count<T>(std::make_index_sequence<sizeof...(I) - 1>());
}

template <class T>
constexpr size_t fields_count() {
    return fields_count<T>(std::make_index_sequence<sizeof(T)>());
}
```

Концепты

В C++20 появилась новая сущность, объявляемая ключевым словом **concept**

```
template <typename T>
concept NothrowDefaultConstructible = noexcept(T{});
```

concept задает compile-time булевский предикат над шаблонными параметрами

```
template <class T, class U>
concept Derived = std::is_base_of_v<U, T>;
```

Концепты

Шаблонные параметры могут быть и нетиповыми

```
template<int I>
concept Even = I % 2 == 0;
```

Можно использовать их и вперемешку

```
template<typename T, size_t MaxSize>
concept SmallerThan = sizeof(T) < MaxSize;
```

Использование концептов

Но чем вообще концепт тогда отличается от **`constexpr bool`**?

Концепты можно использовать везде, где нужен bool на этапе компиляции

```
static_assert(SmallerThan<int, 6>);
```

```
template<typename T>
void f() noexcept(NothrowDefaultConstructible<T>) {
    T t;
    // ...
}
```

Использование концептов

Также, концепты можно использовать для ограничения шаблонного параметра

```
template <class T>
concept DefaultConstructible = std::is_default_constructible_v<T>;
```

```
template <DefaultConstructible T>
struct S {
    T value{};
};
```

Использование концептов

Возможно и частичное применение концептов

```
template<SmallerThan<5> T>
void f(T a) {

}
```

```
f(5);     // OK
f(5ll);   // Compilation Error
```

Предыдущую запись можно упростить с помощью **auto**

```
void f(SmallerThan<5> auto a) {

}
```

Использование концептов

Концепт можно применить даже перед auto для возвращаемого значения

```
template<typename T>
concept SmallerThanPointer = SmallerThan<T, sizeof(void *)>;

template<typename T>
SmallerThanPointer auto get_handle(T &object) {
    if constexpr (SmallerThanPointer<T>)
        return object;
    else
        return &object;
}
```

Использование концептов

В случае неудовлетворения условию концепта, компилятор выдаст гораздо более читаемую ошибку

```
std::list<int> l = {3, -1, 10};
std::sort(l.begin(), l.end());
//  error: cannot call std::sort with std::_List_iterator<int>
//  note:  concept RandomAccessIterator<std::_List_iterator<int>> was not satisfied
```

Использование концептов

Концепты не могут ссылаться на себя и их нельзя ограничить

```
template<typename T>
concept V = V<T*>; // Ошибка: рекурсивный концепт

template<class T>
concept C1 = true;
template<C1 T>
concept Error1 = true; // Ошибка: C1 T ограничивает концепт
```

requires

В концептах можно задавать более сложные условия при помощи конструкции **requires**

```
template<typename T, typename... Args>
concept Constructible = requires(Args... args) { T{args...}; };

template<typename T>
concept Comparable = requires(const T& a, const T& b) {
    {a < b} -> std::convertible_to<bool>;
};
```

requires

В requires можно указывать требования и на существование типа

```
template<typename T>
concept C = requires { typename T::inner; }
```

requires

Теперь попробуем ограничить функцию sort

```
template<typename Iterator>
concept RandomAccessIterator = BidirectionalIterator<Iterator> && requires /* ... */;

template<typename It>
concept Sortable = RandomAccessIterator<It> && Comparable<ValueType<It>>;

template<Sortable Iter>
void sort(Iter first, Iter last) {/* ... */}
```

requires

requires можно использовать также и для ограничения

```
template<typename It>
concept Sortable = RandomAccessIterator<It> && Comparable<ValueType<It>>;

template<typename Iter> requires Sortable<Iter>
void sort(Iter first, Iter last) {/* ... */}
```

requires

Теперь концепт `Sortable` можно и не объявлять

```
template<typename Iter> requires
  RandomAccessIterator<It> && Comparable<ValueType<It>>
void sort(Iter first, Iter last) {/* ... */}
```

requires

```
template<typename T>
concept Addable = requires (T x) { x + x; };

template<typename T> requires Addable<T>
T add(T a, T b) { return a + b; }

template<typename T>
requires requires (T x) { x + x; }
T add(T a, T b) { return a + b; }
```