Функторы и стирание типов

Практическое объектно-ориентированное программирование

23.10.2024

План

- λ - выражения и замыкания
- Проброс захвата и кортежи
- Стирание типов

Новый синтаксис функций

• Начиная с C++11 функции можно писать несколько странно:

```
auto foo() { return 2.0; } // → double foo()
auto foo() -> int { return 2.0; } // нет вывода типов
```

• Основная мотивация: простота вывода из типов аргументов.

```
auto foo(std::input_iterator auto inp) -> decltype(*inp) {
// .....
}
```

• Также этот синтаксис работает для λ - выражений.

```
auto adder = [](int x, int y) -> int  return x + y;
```

λ-выражение и closure

• Без захвата это объект замыкания с перегруженным приведением.

```
auto adder = [](int x, int y) -> int { return x + y; };
struct Closure {
  static int func(int x, int y) { return x + y; }
  using func_t = std::decay_t<decltype(func)>;
  operator func_t() const { return func; }
};
```

• Это означает что есть приведение типов.

```
int (*pf)(int x, int y) = adder; // implicit cast
```

Callables

- Что такое вызываемый объект (callable или функтор)?
  - Указатель на функцию.
  - Объект с приведением к указателю на функцию.
  - Объект с круглыми скобками.
- Мы ничего не забыли?
- Мы не написали ничего лишнего?

std::invoke

- Для вашего обобщённого кода он абстрагирует callables включая stateful.

```
auto psf = &S::foo;
auto psn = &S::n;
auto r1 = std::invoke(foo, 1);
auto r2 = std::invoke(psf, s, 1);
auto r3 = std::invoke(psf, &s, 1);
auto r4 = std::invoke(psn, s);
```

- Что бывает удивительно полезно

Проекция Лававея

• Благодаря магии invoke, лямбды иногда сводятся к проекторам.

```
template <typename Range, typename Callable>
void print_range(Range r, Callable c) {
  for (auto e : r)
    std::cout « std::invoke(c, e) « “ “;
  std::cout « std::endl;
}
std::vector<std::pair<int, int» v = {{1, 1}, {2, 2}, {3, 3}};
print_range(v, [](const std::pair<int, int> &p){ return p.second; });
print_range(v, &std::pair<int, int>::second);
```

Обсуждение

- Почти идеальная прозрачная оболочка для одного аргумента.
- Но это также значит что λ-выражение не (настоящий) callable?
- Не так страшно, приведение работает, делая из него callable там где нужно.
- Интересно что кложура не копируема (но copy-конструируема), а указатель на функцию копируем.

```
auto test = []{};
test = []{}; // FAIL
```

- Можно вызывать приведение типов руками.

```
auto test = +[]{};
test = []{}; // Ok, positive lambda hack
```

Захват аргументов

• При наличии захвата, обобщённое λ - выражение это структура с оператором вызова.

```
int a = 2, b = 3;
auto parm_adder = [a, b](int x, int y) {
  return x * a + y * b;
};
struct Closure {
  int a_, b_;
  Closure(int a, int b) : a_(a), b_(b) {}
  auto operator()(int x, int y) const { return x * a_ + y * b_; }
};
```

Захват по значению: mutable и нет

- При захвате по значению мы можем контролировать const-qual.

```
auto parametrized = [a] { a += 1; // ошибка
struct Closure {
  ....
  auto operator()() const { ..... }
```

- Используется ключевое слово mutable.

```
auto parametrized = [a]() mutable { a += 1; // ok
struct Closure {
  ....
  auto operator()() { ..... }
```

Захват по ссылке

- При захвате по ссылке мы всегда (даже из const метода) можем изменять то что под ссылкой.

```
auto parametrized = [&a](int x) { a += 1; // ok
struct Closure {
  int &a_;
  Closure(int a) : a_(a) {}
  auto operator()(int x) const { a += 1; ..... }
};
```

- Допустим вы хотите это заблокировать и захватить по константной ссылке.
- Что делать?

Константность пропагируется

- Захват по ссылке константных объектов даёт константную ссылку.

```
const int a = 5;
auto l = [&a](int x) { a += x; return a; };
```

- Если нужно захватить по константной ссылке неконстантный объект, нам нужно переименование.

```
int a = 5;
auto l = [???](int x) { a += x; return a; };
```

Захват с переименованием

- В современном C++ хороший тон это явное переименование.

```
int a = 1;
auto lmd = [&ra = a, va = a] { return va + ra; };
```

- В частности это даёт возможность move-захвата.

```
std::vector b = {1, 2, 3};
auto lmd2 = [vb = std::move(b)] { // &vb тут провалится
  return vb;
};
```

- Обратите внимание: поле vb внутри замыкания это value

Продолжаем идеальный проброс

- В прошлый раз мы остановились на таком варианте.

```
template<typename Fun, typename... Args>
decltype(auto) transparent(Fun fun, Args&&... args) {
  return fun(std::forward<Args>(args)...);
}
```

- Теперь должно быть очевидно, что это не всегда сработает.

```
auto a = std::make_unique<int>(1);
auto lmd = [uptr = std::move(a)](int x) { return *uptr + x; };
transparent(lmd, 3); // oops
```

Последняя прозрачная оболочка

- Следующая итерация ещё более идеальна.

```
template<typename Fun, typename... Args>
decltype(auto) transparent(Fun&& fun, Args&&... args) {
  return std::forward<Fun>(fun)(std::forward<Args>(args)...);
}
```

- Этот логический шаг для многих труден.
- Сложно уйти от идеи функции к идее вызываемого объекта.

Захват с вариабельными пачками

- Начиная с C++20 у нас есть полноценные пачки в списке захвата.

```
// ниже предполагаем Args ... args
auto lm1 = [args...] { return sizeof...(args); };
auto lm2 = [...xs = args] { return sizeof...(xs); };
auto lm3 = [&...xs = args] { return sizeof...(xs); };
auto lm4 = [...xs = std::move(args)] {
  return sizeof...(xs);
};
```

Пример: каррирование

- Основная идея: частичная подстановка аргументов

```
auto add = [](auto x, auto y) { return x + y; }
auto add4 = curry(add, 4);
assert(add4(11) == 15);
```

Захват всего

• Для захвата пачки тут лучше захватывать всё по значению.

```
template <typename Function, typename... Arguments>
auto curry(Function function, Arguments... args) {
  return [=](auto... rest) {
    return function(args..., rest...);
  }
}
```

• Это можно использовать для настоящего каррирования.

```
auto fam = [](auto x, auto y, auto z)  return x * (y + z); ;
auto fam3 = curry(fam, 3);
```

Главное правило захвата

- Захватывается только локальный нестатический контекст.

```
int g = 1;
int foo (int b) {
  int x = 2;
  static int a = 3;
  if (b == 4) {
    int y = 5;
    auto lam = [=] { return x + y + a + b + g; };
// здесь изменения x, y, b уже не изменят результат
// зато изменения a и g  изменят
    std::cout « lam() « std::endl;
```

Вернемся к списку захвата

- Можем ли мы организовать проброс списка захвата?

```
auto foo = []<typename T>(T&& a) {
  return [a = std::forward<T>(a)]() mutable { return ++a; };
};
```

- Давайте попробуем.

```
int x = 1;
auto f1 = foo(1); // rvalue мы ожидаем захвата по значению
EXPECT_EQ(f1(), 2);
auto f2 = foo(x); // lvalue мы ожидаем захвата по ссылке
EXPECT_EQ(f2(), 2);
EXPECT_EQ(x, 2);
```

Причина проблем

- Давайте поиграем в компилятор.

```
[a = std::forward<T>(a)]() mutable { return ++a; };
template <typename T>
struct Closure {
  T a_;
  Closure(T&& a) : a_(std::forward<T>(a)) {}
  auto operator()() { return ++a_; }
};
```

- Да, конечно, это всё равно захват по значению.

Что тут можно сделать?

- Мы должны где-то сохранить либо value либо ref и пробросить это by-value.

```
template <typename T> auto fwd_capture(T&& x) {
struct CapT { T value; };
  return CapT{std::forward<T>(x)};
}
auto foo = []<typename T>(T&& a) {
   return [a = fwd_capture(a)]() mutable { return ++a.value; };
};
```

- Что если мы теперь хотим пробросить целую пачку?

Храним пачку в std::tuple

- Кортеж это специальный тип чтобы связывать разнородные аргументы.

```
template <typename ... T> auto fwd_capture(T&& ... x) {
  return std::tuple<T...>(std::forward<T>(x)...);
}
auto foo = []<typename ... T>(T&& ... a) {
  return [a = fwd_capture(a...)]() mutable {
    return ++std::get<0>(a);
  };
};
```

- Можно ли тут улучшить?

Создание tuple

- Есть пять способов создать кортеж

1. конструктор
2. tuple<VTypes...> make_tuple(Types&&...)
3. tuple<CTypes...> tuple_cat(Tuples&&...)
4. tuple<Types&...> tie(Types&...)
5. tuple<T&&...> forward_as_tuple(T&&...)

Простые способы

- Создание конструктором

tuple<int, double, int> t1(1, 2.0, 3);

- Создание через make_tuple

auto t2 = make_tuple(4, 5.0, 6);

- Объединение

auto t3 = tuple_cat (t1, t2, make_pair(7, 8.0));

assert (t3 == make_tuple(1, 2.0, 3, 4, 5.0, 6, 7, 8.0));

Связывание

- Любые две переменные могут быть связаны в кортеж.

```
int a = 5; double b = 1.0; auto t = std::tie(a, b);
```

- Точно так же кортеж развязывается в переменные.

```
std::tuple<int, int> foo();
int a; int b; std::tie(a, b) = foo();
```

- Есть упрощающий синтаксис

```
auto [a, b] = foo();
```

Связывание в деталях

• Связывание для кортежа может быть и ссылкой (в т. ч. правой).

```
auto [xv, yv, zv] = make_tuple("a 1.0, 4); // ok, zv is int
auto&& [xrv, yrv] = make_tuple(2, 3); // ok, xrv is int&&
```

• Биндинги оказались настолько удобны, что их распространили на массивы.

```
int a[2] = {1,2};
auto& [xr, yr] = a; // изменение xr изменит a[0]
```

• А также на любые классы, у которых открыты все нестатические элементы.

```
struct {int x; double y} b = {1, 2.0};
const auto& [xcr, ycr] = b; // изменение xcr невозможно
```

Связывание для собственных классов

• Чтобы поддержать связывание для собственного класса, необходимо определить специализации для всего трёх функций

```
class Config {
  int x; double y; std::string z;
  // открытые члены, включая геттеры вида get_x(), get_y() и get_z()
};
template<> struct tuple_size<Config>: integral_constant<size_t, 3>
{};
template<> decltype(auto) get<0>(Config& c) { return c.get_x(); }
template<> struct tuple_element<0, Config> { using type = int; };
```

• Теперь будет работать (нужно добавить get для остальных).

```
auto [id, value, name] = get_config();
```

Проброс в кортеж

• Мы можем также использовать forward_as_tuple.

```
auto foo = []<typename ... T>(T&& ... a) {
  return [a = std::forward_as_tuple(a...)]() mutable {
    return ++std::get<0>(a);
  };
};
```

• Внезапно оказывается что нам и для одного аргумента не надо было ничего придумывать.

Применение кортежа как аргументов

- Мы можем пробрасывать кортеж как аргумент в std::apply

```
auto add = [](auto x, auto y) { return x + y; };
auto x = std::apply(add, std::pair(1, 2));
```

- В принципе можно и самому реализовать это через std::invoke и распаковку.
- std::apply для tuple может вместе с лямбдами играть роль std::for_each

```
std::apply([&fn](auto&& ... xs) (fn(xs), ...);, t);
```

- И да, я забыл форвардинг, добавьте его самостоятельно.

Стирание через std::any

- Позволяет сделать из C++ практически Python.

```
std::any a = 1;
a = 3.14;
assert(a.has_value());
std::cout « std::any_cast<double>(a) « std::endl;
a.reset();
assert(!a.has_value());
a = Heavy(100); // работает, но лишнее перемещение
```

- Размещение делается с помощью make_any.

```
auto h = std::make_any<Heavy>(100); // так лучше
```

Варианты

• Вариант это type-discriminated union между типами из ограниченного набора.

std::variant<int, float> v = 12;

int i = std::get<int>(v);

if (std::holds_alternative<float>(v)) { .....

• Можно сохранить вектор вариантов.

std::vector<std::variant<int, float, string> > vec = {10, 1.5, “hello“};

• Как обойти этот вектор?

Первый вариант: явный std::visit

- Мы можем внутри std::visit расписать каждый случай.

```
for (auto& v: vec) {
  std::visit([](auto&& arg) {
    using T = std::decay_t<decltype(arg)>;
    if constexpr(std::is_same_v<T, int>) {
      std::cout « arg % 5;
    }
    else if constexpr(std::is_same_v<T, float>) {
      std::cout « std::round(arg) - std::round(arg / 5) * 5;
    }
  }, v);
```

- Можно ли сделать лучше?

Задача: "перегрузка"лямбд

• Обычные функции могут быть перегруженными, но с лямбдами простой пример не работает.

```
auto f = [](int i) { печатаем "forint"};
auto f = [](double d) { печатаем "fordbl"}; // Увы
```

Можно ли сымитировать "перегрузку"?

```
auto f = make_overload([](int i) { печатаем "forint"},
[](double d) { печатаем "fordbl"});
f(3); // forint
f(3.0); // fordbl
```

Реализация "перегрузки"

- Первый вариант.

```
template <typename... F>
struct overload : F... {
  overload(F... f) : F(f)... {}
};
template <typename... F>
auto make_overload(F... f) { return overload<F...>(f...); }
```

- Как вы думаете, почему он не работает?

Менее наивный подход

```
template<typename F, typename... Fs>
struct overload : F, overload<Fs...> {
  using F::operator();
  using overload<Fs...>::operator();
  overload(F&& f, Fs&&... fs): F(std::forward<F>(f)),
    overload<Fs...>(std::forward<Fs>(fs)...) {}
};

template<typename F> struct overload {
  using F::operator();
  overload(F&& f) : F(std::forward<F>(f)) {}
};
```

- Как вы думаете, почему он не работает?

Одно замечание к хвосту "рекурсии"

```
template<typename F, typename... Fs>
struct overload : F, overload<Fs...>;
```

- Правильный подход: закрыть специализацией.

```
template<typename F>
struct overload<F> : F {
  using F::operator();
  overload(F&& f) : F(move(f)) {}
};
```

- Дело в том, что в C++ нет перегрузки классов

Все больше и больше раскрытий

- Начиная с C++17 работает красивый вариант.

```
template <typename... F>
struct overload : F... {
  using F::operator()...; // ok для C++17
  overload(F... f) : F(f)...
};
template <typename... F>
auto make_overload(F... f) {
  return overload<F...>(f...);
}
```

- На самом деле можно даже ещё проще.

Больше красоты

• Нам не нужен make_overload так как начиная с C++17 есть deduction hints.

```
template <typename... F>
struct overload : F... { using F::operator()...; };
template<class... Ts> overload(Ts...) -> overload<Ts...>;
```

• Теперь использование ещё симпатичней

```
auto f = overload {
  [](int i) { cout « “forint“ « endl; },
  [](double d) { cout « “fordouble“ « endl; }
};
```

Второй вариант: перегрузка

- Либо использовать перегрузку лямбд.

```
std::vector<std::variant<int, float, string» vec;
for (auto& v: vec) {
  std::visit(overloaded {
    [](int arg) { std::cout « arg % 5; },
    [](float arg) { действие для double }
  },v);
```

- Это даёт почти паттерн матчинг!

Экзотический вариант: обход пар

- Можно обходить несколько вариантов сразу.

```
std::variant<int, float> v1, v2;
std::visit(overloaded {
    [](int, int) { cout « "2 ints«< endl; },
    [](auto, auto) { cout « "int + float«< endl; },
    [](float, float) { cout « "2 floats«< endl; },
  }, v1, v2);
```

- Я не нашёл достойных применений этой технике.

Поищем рекурсию

• До сих пор мы для лямбд искали возможности которые есть у функций.

```
auto factorial = [&] (int i)
  return (i == 1) ? 1 : i * factorial(i - 1);
;
```

• Как вы думаете по каким причинам это не будет работать?

Стирание типов через std::function

• Нам нужно частично стереть тип, чтобы иметь возможность его не знать заранее.

```
std::function<int (int)> factorial = [&] (int i) {
  return (i == 1) ? 1 : i * factorial(i - 1);
};
```

• Что такое std::function?

• Как и std::any это красиво обёрнутый void*. Настоящий тип при этом сохраняется.

```
std::cout « factorial.target_type().name() « std::endl;
```