Элементы функционального программирования в языке C++

Вадим Винник

Ведущий инженер-программист

vinnik.wadim@yandex.ru

C++ Russia
2023

Вадим Винник

VadimVi

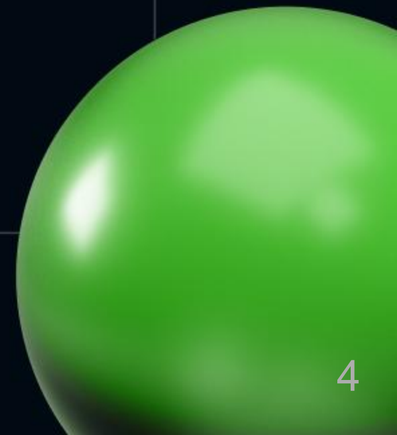
О себе

- C++
- Метапрограммирование
- Функциональный стиль
- Haskell
- Формальная семантика
- Практическая разработка
- Преподавание

Краткое содержание

- Что такое функциональное программирование
- Потребность в функциональном стиле
- Функции высшего порядка
- Указатели на функции как простейший вид
- Функциональные объекты в C++
- `std::function` и `std::bind`
- Рекурсия и неподвижные точки

Пролегомены



Функциональное программирование

- Функциональное программирование имеет столь же давнюю историю, как и привычное большинству программистов императивное.
  - Первый императивный язык высокого уровня FORTRAN — 1957.
  - Первый функциональный язык LISP — 1958.
- Функциональное программирование долгое время занимало узкую нишу:
  - научные исследования по теории программирования;
  - разработка трансляторов;
  - разработка систем с особыми требованиями (надёжность, корректность);
  - встроенные языки сложных программных систем (см. *десятое правило Гринспена*).
- Главный сдерживающий фактор — высокий порог вхождения.
  - В основе функциональных языков — λ-исчисление и теория категорий.
  - Чтобы вывести “Hello world” на языке Haskell, нужно применить *монаду*.
  - Ничего сложного: это моноид в категории эндофункторов.

Рост популярности функциональной парадигмы

- Сейчас резко возрастает интерес к функциональной парадигме.
- Функциональное программирование опередило своё время.
- Его преимущества отчётливо проявляются и выходят на первый план.
- Новые возможности аппаратуры: массовость многоядерных машин.
  - Нужны технологии, обеспечивающие распараллеливаемость алгоритмов…
  - … без усилий со стороны программиста
  - … и без потерь производительности на синхронизацию.
- Новые вызовы: возросшая сложность программных систем.
  - Нужны новые способы декомпозиции систем,
  - … сохраняющие контроль над её отдельными частями,
  - … позволяющие гибко приспособить язык к предметной области (см. EDSL),
  - … общегчающие создание гарантированно корректных систем.

Способы внедрения функциональной парадигмы

- Изучать и внедрять языки, специально предназначенные для функционального программирования.

- Понятия и методы, характерные для функционального программирования, импортировать в распространённые языки.

Основные черты функциональной парадигмы

- Чистые вычисления
  - Отсутствие изменяемого состояния.
  - Чёткое отделение чистых вычислений от ввода-вывода.
  - Референциальная прозрачность: возможность замены равного равным.
  - Программа есть математическая формула, только в необычной нотации.
  - Гарантированная воспроизводимость результатов запуска.
  - Чистые вычисления хорошо распараллеливаются.
  - Рекурсии вместо циклов.
- Работа с неизменяемыми объектами
  - Вместо изменения одного и того же объекта — вычисление нового объекта…
  - … или вычисление изменений, накладываемых на старый объект (см. декоратор).
  - Размен памяти и скорости *потока* на эффективное распараллеливание.
  - Пример: сохранение текста в редакторе одновременно с редактированием.

Важнейшее понятие: функция высшего порядка

- “Обычная” функция — первого порядка:
  - принимает и возвращает “обычные” данные (числовые, строковые и т.д.).
- Функция высшего порядка:
  - обрабатывает обычные данные и функции (первого или высшего) порядка.
- Аргументы и\или возвращаемое значение функции высшего порядка — в свою очередь функции.
- Иначе говоря, функции обрабатываются как обычные объекты данных.
- В частности, функции можно хранить в контейнерах и создавать на лету.
- Позволяет делать программы более гибкими и простыми для понимания.
- Да, более простыми. Сложное понятие проще применять, …
  - … если оно адекватно отражает естественную логику предмета.

Пример: функция `std::for_each`

- Принимает на вход пару итераторов и операцию `f`.
- Обходит диапазон контейнера поэлементно.
- Применяет операцию `f` к каждому элементу.
- Конкретный способ обхода контейнера инкапсулирован в итераторах.
- Общее понятие обхода реализовано в функции `for_each`.
- Пользователю функции остаётся задать лишь вариативное:
  - операцию, которую нужно применить к каждому элементу.
- Единый и универсальный интерфейс для всех действий вида “над каждым элементом `x` из контейнера `s` выполнить операцию `f`”.
- Функция `for_each` — высшего порядка, т.к. имеет аргумент-функцию `f`.

Функции высшего порядка в традиционных языках

- Язык C: указатели на функции.
- Стандартные функции `lsearch`, `bsearch`, `qsort`.
- Функции обратного вызова (callback).
- Язык C++: функциональные объекты и λ-нотация.
- Алгоритмы обработки контейнеров в STL:
  - `std::transform`, `std::copy_if`, `std::find_if`, `std::accumulate` и множество других.
- Язык C#: делегаты и λ-нотация.
- События (event) и подписка обработчиков.
- Linq-расширения коллекций:
  - `Select`, `Where`, `Join`, `Aggregate`, `First`, `Any`.

Указатели на функции

- Простейший инструмент высшего порядка
- Доступно в языке C
- Создание обобщённых алгоритмов
- Передача состояния
- Функции обратного вызова

Указатели на данные и функции в языке C

```
int x;                  // переменная, содержащая числовое значение
int *p;                 // переменная-указатель на переменную-число
p = &x;                 // поставить указатель на переменную
*p = 0;                 // косвенный доступ к переменной x
int f(int x);           // обычная функция
int *g(int x);          // функция, возвращающая указатель
int (*fp)(int x); // переменная-указатель на функцию
fp = &f;                // поставить указатель на функцию f
fp = f;                 // то же самое, упрощённый синтаксис
x = (*fp)(0);           // косвенный вызов функции по указателю
x = fp(0);              // то же самое, упрощённый синтаксис
```

Функция высшего порядка на языке C

```
typedef void (*int_action_t)(int);

void int_for_each(int *b, int *e, int_action_t f) {
  while (b != e) // указатели в роли итераторов
    f(*b++);
}

void int_print(int x) {
  printf("%d\n", x);
}

int m[8] = { 1, 1, 2, 3, 5, 8, 13, 21 };
int_for_each(m, m+8, int_print)
```

Свёртка по бинарной операции, язык C

Недостаток алгоритма `int_for_each`: каждый элемент массива обрабатывается сам по себе, невозможно при обработке очередного элемента использовать результат обработки предыдущего.

```
typedef int (*int_binop_t)(int x, int y);

int int_fold(int *b, int *e, int_binop_t f, int a) {
  while (b != e)
    a = f(a, *b++);
  return a;
}
```

Примеры применения свёртки

```
int int_maxof(int x, int y)  { return x > y ? x : y; }
int int_plus (int x, int y)  { return x + y; }
int int_mult (int x, int y)  { return x * y; }

int m[] = { 1, 1, 2, 3, 5, 8, 13, 21 };

int max     = int_fold(m, m+8, int_maxof, INT_MIN);
int sum     = int_fold(m, m+8, int_plus,         0);
int product = int_fold(m, m+8, int_mult,         1);
```

Вычисляется $(\ldots((a_0 \lozenge x_0) \lozenge x_1) \ldots \lozenge x_{n-1})$ с пользовательской операцией $\lozenge$.

Поэлементное применение “наоборот”

```
typedef int (*int_unop_t)(int x);

int int_apply_chain(int_unop_t *b, int_unop_t *e, int x) {
  while (b != e) // b, e - указатели на указатели на функции
    x = (*b++)(x);
  return x;
}
```

- Вычисляется $f_{n-1}(...f_1(f_0(x))...)$.
- Цепочку можно формировать динамически из имеющихся функций.
- Компенсируется невозможность создавать функции динамически.
- Шаблоны “композит” и “цепочка обязанностей” без объектов и классов.

Передача состояния

- Между итерациями передавать объект данных,
  - хранящий промежуточное состояние.
- Обработка каждого очередного элемента изменяет это состояние.

```
typedef void (*int_stateful_op_t)(int x, void *s);

void int_apply_stateful(
  int *b, int *e, int_stateful_op_t f, void *s)
{
  while (b != e)
    f(*b++, s);
}
```

Пример вычисления с состоянием

```
typedef struct { int count; int sum; } int_average_state_t;

void int_count_and_sum(int x, void *p) {
  int_average_state_t *state = (int_average_state_t*) p;
  ++state->count;
  state->sum += x;
}

double int_average(int *b, int *e) {
  int_average_state_t state = {0, 0};
  apply_stateful(b, e, &int_count_and_sum, &state);
  return ((double) state.sum) / state.count;
}
```

Функции высшего порядка в стандартной библиотеке

```
int str_comparer(void const *p, void const *q) {
  char const *s = *(char const**) p;
  char const *t = *(char const**) q;
  return strcmp(s, t);
}

char *words[] = {"dog", "cat", "cow", "pig", "rat", "hen"};

qsort(words, 6, sizeof(char*), str_comparer);
```

Функции обратного вызова в языке C

- В англоязычной терминологии — callback.
- Применяются для гибко настраиваемой реакции программы на наступление определённых событий.
- Компонент программы воплощает обнаружение события, …
- … а для реакции на него вызывает пользовательскую функцию обратного вызова, …
- … указатель на которую получает при инициализации.
-
```
#define EBOOK_BTN_PGNEXT 10001 // аппаратный код кнопки
void on_next_page(void *state);
register_button(EBOOK_BTN_PGNEXT, on_next_page, state);
```

Разбор образа файловой системы

- Получение данных, реакция на директорию, файл — обратные вызовы.
- “Диалог” обработчиков с движком через аргументы и возврат значений.

```
void *traverse_file_system(
  read_status_t (*reader_f)(size_t, byte_t*, size_t, void*),
  dir_handling_status_t (*on_dir_enter)(char const*, void*),
  dir_handling_status_t (*on_dir_leave)(char const*, void*),
  file_handling_status_t (*on_file)(char const*, void*),
  error_handling_status_t (*on_parse_error)(void*),
  void *reader_state,
  void *parser_state);
```

Заключительные замечания по языку C

- Универсальные алгоритмы инкапсулируют способ обхода структуры данных, который может быть сложным.
- Пользователю остаётся передать указатели на свои функции для обработки элементов этой структуры.
- Сдвиг парадигмы: вместо того, чтобы функции только действовали, а данные только подвергались действиям, функции становятся данными!
- Правда, единственный вид “обработки” функций — вызов.
  - Создать “на лету” или изменить существующую функцию язык не позволяет.
- Вместе с указателями на функции обратного вызова приходится передавать состояние без контроля типов — по `void*`.

Функциональные объекты

- Моделирование функций объектами классов
- Преодоление недостатков, присущих указателям
- Стандартные шаблоны-обёртки над операциями
- Стандартные функции высшего порядка

Функциональные объекты в C++

- В ООП данные и действия объединяются в объекте.
- Объект — и пассивное хранилище данных состояния, и активный преобразователь своего состояния.
- C++ позволяет перегрузку операции () применения функции к аргументу.
- Тем самым, объект может вести себя так, *как если бы* был функцией.
- Новое средство не сразу было воспринято сообществом.
- *Наверное, вы и не подозревали, что этот оператор тоже можно перегрузить. Если у вас часто возникает непреодолимое желание превратить объект в функцию, возможно, ваша психика нестабильна и вам стоит серьезно подумать над сменой рода занятий*
  (Дж. Элджер, 1998)

Пример функционального объекта

```
class adder {
public:
  explicit adder(int a): m_a(a) {}
  int operator()(int x) const { return m_a + x; }
private:
  int const m_a;
};

adder const plus_two(2);
int const y = plus_two(10); // 12
```

Стандартные функциональные объекты

Стандартная библиотека содержит шаблоны классов-обёрток для операций:

- Арифметических: `plus`, `minus`, `multiplies`, `divides`, `modulus`, `negate`.
- Сравнений: `equal_to`, `not_equal_to`, `greater`, `less`, ...
- Логических: `logical_and`, `logical_or`, `logical_not`.
- Побитовых: `bit_and`, `bit_or`, `bit_xor`, `bit_not`.

```
template <class T = void>
struct multiplies {
  constexpr T operator()(const T &lhs, const T &rhs) const
    { return lhs * rhs; }
};
```

Использование обёрток над операциями

- Сами по себе шаблоны классов-обёрток над операциями были бы бесполезны.
  - В выражениях удобнее писать знаки операций: +, <=, && и др.
- Удобны для передачи операций в функции высшего порядка.

```
std::sort(
    names.begin(), names.end(), std::greater<std::string>{});

auto const product = std::accumulate(
    nums.cbegin(), nums.cend(), 1.0, std::multiplies<double>{});
```

Особенности функциональных объектов

- В отличие от указателей на функции, хранят состояние внутри себя.
- Отпадает необходимость в небезопасных преобразованиях `void*`.
- Состояние может быть как неизменяемым, так и изменяемым.
- Соответственно, `operator()` может быть константным или нет.
- В функциях стандартной библиотеки передаются по значению.
- Следовательно, необходим конструктор копии (copy constructible).
- Рекомендуется делать легковесными.
- Не могут преобразовываться к типу указателя на функцию.
- Поэтому несовместимы с традиционным для языка C механизмом…
  - … если не прибегнуть к некоторым ухищрениям.

Адаптер функционального объекта (упрощённый)

```
template <typename F, typename... A>
decltype(std::declval<F>()(std::declval<A>()...))
fptr_adaptor(A... args, void *p) {
  auto &f = *static_cast<F*>(p);
  return f(args...);
}
```

Превращает функциональный объект в функцию, принимающую состояние через нетипизированный указатель. Функция приводит указатель к типу функционального объекта и применяет его к остальным аргументам.

Функциональный объект для адаптера

```
class pcstring_cmp {
  std::locale m_locale;

public:
  pcstring_cmp(std::string const& name) : m_locale(name) {}

  int operator()(void const* p, void const* q) const {
    auto const& s = **static_cast<std::string* const*>(p);
    auto const& t = **static_cast<std::string* const*>(q);
    return m_locale(s, t) ? -1 : m_locale(t, s) ? +1 : 0;
  }
};
```

Применение адаптера

```
std::string *items[] = { new std::string("one")... };

pcstring_cmp comparer{"en_US.UTF-8"};

qsort_r(
  items,
  count,
  sizeof(items[0]),
  &fptr_adaptor<pcstring_cmp, void const*, void const*>,
  &comparer);
```

Интересные детали

- Лямбда-выражения
- Передача функций через тип-параметр шаблона
- Шаблон класса std::function
- Частичное связывание аргументов функции

Лямбда-выражения

```
auto f = [a] (int x) -> int { return a * x + x % a; }
```

- “Синтаксический сахар” для функциональных объектов.

```
class lambda_t {
public:
  explicit lambda_t (int val): m_val(val) {}
  int operator()(int x) const { return m_val * x + x % m_val; }
private:
  int const m_val;
};

lambda_t f { a };
```

Особенности лямбда-выражений

```
auto f = [a] (int x) -> int { return a * x + x % a; }
```

- Переменные из области видимости могут захватываться в замыкание по значению (с копированием) или по ссылке.
- Лямбда-выражение *с пустым замыканием* можно преобразовывать к типу указателя на функцию.
- Тип возвращаемого значения можно явно не указывать: оставить вывод типа компилятору.
- (C++20) можно явно указывать список параметров шаблона.
- Множество других интересных деталей, лежащих в стороне от сегодняшней темы.

Способ передачи аргументов-функций в STL

```
template <class InputIt, class UnaryPred>
constexpr bool all_of(InputIt first, InputIt last, UnaryPred p);
```

- Шаблон функции параметризован по типу предиката.
- Это даёт компилятору возможность глубоко оптимизировать код.

```
bool is_non0(int x) { return x != 0; }

struct non0_t {
  bool operator()(int x) const { return x != 0; }
};

v1 = std::all_of(b, e, &is_non0); // UnaryPred == bool(*)(int)
v2 = std::all_of(b, e, non0_t{}); // UnaryPred == non0_t
```

Шаблон класса `std::function`

```
template <typename R>
class function; // намеренно оставлено без определения

template <typename R, typename... Args> // специализация
class function<R(Args...)> {...};

std::function<bool(int)> p;
p = &is_non0;                       // указатель на функцию
p = non0_t {};                      // объект класса
p = [](int x) { return x > 0; }     // лямбда

v = std::all_of(b, e, p); // UnaryPred == std::function<bool(int)>
```

Шаблон класса `std::function`

- Универсальная полиморфная обёртка для вызываемых сущностей:
  - обычных функций (посредством указателей на них),
  - функциональных объектов,
  - лямбда-выражений,
  - bind-выражений (см. далее),
  - членов-функций,
  - членов-данных.
- В реализации используется стирание типов (type erasure).
  - Объект `std::function<...>` не помнит фактический тип завёрнутой в него сущности.
  - Потеря производительности по сравнению с вызовом функции по указателю.
  - Такова плата за универсальность и гибкость.

Частичное применение функции в математике

- Пусть дана функция f от трёх аргументов: $f(x, y, z)$.
- “Обычная” трактовка: выражения вида $f(\mathrm{a})$ не имеют смысла.
- “Необычная” трактовка: фиксируется значение первого аргумента.
- Значение выражения — остаточная функция $f(y, z)$ двух аргументов.
- Соответственно, значение выражений вида $f(\mathrm{a}, \mathrm{b})$ — функция $f(z)$.
- Математический аппарат:
  - каррирование (по имени Хаскелла Карри);
  - λ-исчисление;
  - декартово-замкнутые категории.
- Применение в программировании:
  - Языки Haskell, LISP и мн. др.

Bind-выражения: каррирование в C++

- Пусть нужно подсчитать (count_if) положительные элементы контейнера.
- Второй аргумент `std::greater<int>{}` фиксирован и равен 0.
- Нужно превратить сравнение в функцию одного оставшегося аргумента.

```
auto const is_positive = std::bind(std::greater<int>{}, _1, 0);
auto const n = std::count_if(v.cbegin(), v.cend(), is_positive);
```

- Функция `bind` строит новую функцию, единственный аргумент которой подаётся в объект-функцию сравнения первым.
- Вторым аргументом в объект-функцию сравнения подаётся 0.
- Объекты _1, _2, … замещают аргументы созданной функции.

Пример из реального приложения

Пусть имеется библиотечная функция

```
int listen_tcp(int port, std::function<int(packet_t const*)> f);
```

Для каждого пакета, поступившего на заданный порт, вызывается пользовательская функция-обработчик `f` с единственным аргументом.

Пусть в программе есть обработчик пакетов с тремя аргументами

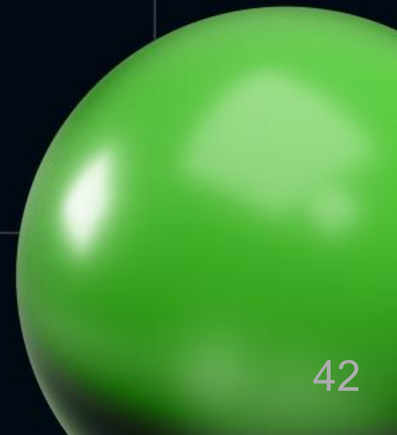
```
int handle_packet(config_t const*, cache_t*, packet_t const*);
```

Тогда её можно согласовать с `listen_tcp`, связав лишние аргументы:

```
listen_tcp(8307, std::bind(&handle_packet, pconfig, pcache, _1));
```

Обработка диапазонов

- Последовательность целых чисел
- Фильтрация по предикату
- Подсчёт числа элементов
- Поэлементное преобразование



Пример

```
auto primes = std::views::iota(1)
  | std::ranges::views::filter([](auto x) {
        return 2 == std::ranges::count_if(
            std::views::iota(1, x+1),
            [x](auto k) { return x % k == 0; }
        );
      }
    )
  | std::views::take(100)
  | std::views::transform([](auto x) { return x * x; });
```

Пример

```
auto primes = std::views::iota(1)
  | std::ranges::views::fil
        return 2 == std::ranges::count_if(
            std::views::iota(1, x+1),
            [x](auto k) { return x % k == 0; }
        );
      }
    )
  | std::views::take(100)
  | std::views::transform([](auto x) { return x * x; });
```

Взять *бесконечную* последовательность чисел начиная с 1

Пример

```
auto primes = std::views::iota(1)
  | std::ranges::views::filter([](auto x) {
        return 2 == std::ranges::count_if(
            std::views::iota(1, x+1),
            [x](auto k) { return x % k == 0; }
        );
      }
    )
  | std::views::take(100)
  | std::views::transform([](auto x) { return x * x; });
```

Отобрать из них те (x), что удовлетворяют критерию

Пример

```
auto primes = std::views::iota(1)
  | std::ranges::views::filter([](auto x
        return 2 == std::ranges::count_if(
            std::views::iota(1, x+1),
            [x](auto k) { return x % k == 0; }
        );
      }
    )
  | std::views::take(100)
  | std::views::transform([](auto x) { return x * x; });
```

Должно существовать ровно два элемента…

Пример

```
auto primes = std::views::iota(1)
  | std::ranges::views::filter([](auto x) {
        return 2 == std::ranges::count_if(
            std::views::iota(1, x+1),
            [x](auto k) { return x % k == 0; }
        );
      }
    )
  | std::views::take(100)
  | std::views::transform([](auto x) { return x * x; });
```

…в диапазоне от 1 до x (включительно)…

Пример

```
auto primes = std::views::iota(1)
  | std::ranges::views::filter([](auto x) {
        return 2 == std::ranges::count_if(
            std::views::iota(1, x+1),
            [x](auto k) { return x % k == 0; }
        );
      }
    )
  | std::views::take(100)
  | std::views::transform([](auto x) { return x * x; });
```

…на которые x делится без остатка

Пример

```
auto primes = std::views::iota(1)
  | std::ranges::views::filter([](auto x) {
        return 2 == std::ranges::count_if(
            std::views::iota(1, x+1),
            [x](auto k) { return x % k == 0; }
        );
      }
    )
  | std::views::take(100)
  | std::views::transform([](auto x) { return x * x; });
```

бесконечная последовательность всех простых чисел

Пример

```
auto primes = std::views::iota(1)
  | std::ranges::views::filter([](auto x) {
        return 2 == std::ranges::count_if(
            std::views::iota(1, x+1),
            [x](auto k) { return x % k == 0; }
        );
      }
    )
  | std::views::take(100)
  | std::views::transform([](auto x) { return x * x; });
```

взять из неё первые 100 элементов

Пример

```
auto primes = std::views::iota(1)
  | std::ranges::views::filter([](auto x) {
        return 2 == std::ranges::count_if(
            std::views::iota(1, x+1),
            [x](auto k) { return x % k == 0; }
        );
      }
    )
  | std::views::take(100)
  | std::views::transform([](auto x) { return x * x; });
```

каждое число возвести в квадрат

Пример

```
auto primes = std::views::iota(1)
  | std::ranges::views::filter([](auto x) {
        return 2 == std::ranges::count_if(
            std::views::iota(1, x+1),
            [x](auto k) { return x % k == 0; }
        );
      }
    )
  | std::views::take(100)
  | std::views::transform([](auto x) { return x * x; });
```

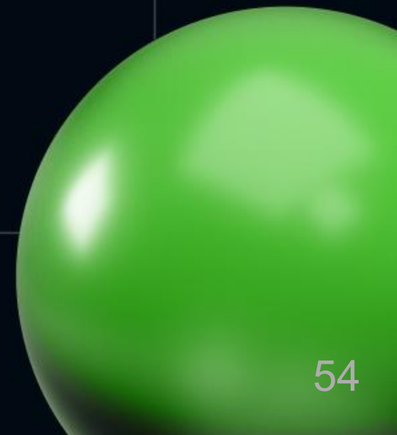
квадраты первых ста простых чисел

Замечания о бесконечных диапазонах

- Обработка бесконечных структур данных (списков, деревьев) очень характерна для функционального программирования.
- Конечно, бесконечный объём данных не хранится в памяти.
- Бесконечная структура данных виртуальна.
- В памяти располагается лишь алгоритм, позволяющий вычислить очередной элемент…
  - если в нём возникнет потребность.
- Наличие `take` предотвращает бесконечный цикл и переполнение памяти.

Кунсткамера

- Из чистой любви к искусству
- Рекурсивные лямбда-выражения
- Комбинатор неподвижной точки
- Оказывается, и так тоже можно



Может ли лямбда-выражение быть рекурсивным?

Наивная попытка

```
auto f = [&f](int k) {
  return k == 0 ? 1.0 : k * f(k-1);
};
```

- Переменная f объявлена с ключевым словом auto.
- Её тип должен вычислить компилятор исходя из инициализатора.
- Пока инициализатор не будет обработан до конца, тип f неизвестен.
- Но инициализатор зависит от самой переменной f.

Может ли лямбда-выражение быть рекурсивным?

Вторая попытка

```
std::function<double(int)> f = [f](int k) {
  return k == 0 ? 1.0 : k * f(k-1);
};
```

- Переменная f объявлена с конкретным типом.
- Код компилируется, но не работает.
- Переменная f захватывается по значению.
- В правой части переменная f уже известна компилятору…
- …Но еще не проинициализирована.

Может ли лямбда-выражение быть рекурсивным?

Третья и последняя попытка

```
std::function<double(int)> f = [&f](int k) {
  return k == 0 ? 1.0 : k * f(k-1);
};
```

- Переменная f захватывается по ссылке.
- Проинициализированная f и та f, которая используется в замыкании, суть одна и та же переменная.
- Компилируется и работает.

Чего не хватает предыдущему решению?

- Лямбда-выражения удобно использовать именно там, где не хочется давать функции имя.
- Приведённое решение построено исключительно на том, что лямбда получает собственное имя через переменную и использует её же.
- Новая задача: сделать полностью анонимную рекурсивную функцию, которая не знает своего имени.
- Разве это возможно? Рекурсивная функция должна вызывать себя, но как ей это сделать, не зная собственного имени?
- Возможно с помощью комбинатора неподвижной точки.

Понятие неподвижной точки в общем виде

- Пусть дана функция $f : X \to X$.
- Рассмотрим корень уравнения $f(x) = x$.
- Если множество X называть *пространством*,
- …а его элементы x — *точками*,
- … то f есть *преобразование пространства*,
- … которое может оставлять некоторые точки неподвижными.
- Корень уравнения $f(x) = x$ называют *неподвижной точкой* функции f.
- У монотонного отображения *полной решётки* на себя всегда есть хотя бы одна неподвижная точка.
  - Следствие теоремы Кнастера-Тарского.

Неподвижные точки в программировании

- Цикл `while b do S od` — это неподвижная точка преобразования `F(X) = if b then S; X fi`
- Рекурсивный тип данных (например, список) есть неподвижная точка некоторого преобразования типов.
- Любая рекурсивная функция есть неподвижная точка некоторой функции высшего порядка (т.е. преобразования функций).
- Пример на языке Haskell (функция возведения числа 2 в степень x):

```
pow2Helper f = \x -> if x == 0 then 1 else 2 * (f (x-1))
pow2 = fix pow2Helper
```

  - Здесь `fix` — комбинатор неподвижной точки из стандартной библиотеки.
  - `fix f = f (fix f)`

Программирование через неподвижные точки

```
pow2Helper f = \x -> if x == 0 then 1 else 2 * (f (x-1))
pow2 = fix pow2Helper
```

- Вспомогательная функция имеет два аргумента:
  - в свою очередь функцию f (из числа в число),
  - число x.
- Преобразует f в другую функцию того же типа (из числа в число).
- Если в качестве f подать “несовершенную” функцию возведения 2 в степень x, грубое приближение к искомой функции, то…
- Результатом преобразования станет улучшенная версия этой функции.
- Предполагается, что f работает правильно для аргументов, меньших x.

Комбинатор неподвижной точки на C++

```
template <typename R, typename A>
std::function<R(A)>
fixed(std::function<R(std::function<R(A)>, A)> h)
{
  return [h](A x) { return h(fixed(h), x); };
}
```

- Универсальный движок рекурсии.
- Инкапсулирует в себе рекурсивность как таковую.
- Пользователю достаточно изобрести лишь нерекурсивную функцию h.

Пример применения

```
std::transform(
    xs.cbegin(),
    xs.cend(),
    std::back_inserter(ys),
    fixed<double, int>(
        [](std::function<double(int)> f, int k)
        {
            return k == 0 ? 1.0 : k * f(k - 1);
        }
    )
);
```

Итоги

- Даже языки, не предназначавшиеся создателями для функционального стиля программирования, позволяют создавать программы в функциональном стиле.
- Элементы функционального стиля могут хорошо сочетаться с императивной основой языка.
- Основной инструмент — объект данных, ведущий себя как функция, хоть и не являющийся функцией в собственном смысле слова.
- Смежные вопросы, требующие отдельного рассмотрения:
  - неизменяемые (immutable) объекты, чистые вычисления, функциональные структуры данных, реактивное программирование.

Практические итоги

- Меньше циклов for с явным итерированием по контейнерам.
- Больше std::accumulate, transform, find_if и их параллельных версий.
- Меньше изменяемых состояний.
- Больше данных в виде бесконечных последовательностей.
- Больше обработчиков для таких данных:
  - поэлементное преобразование,
  - фильтрация по условию,
  - агрегирование.
- Более гибкие, легко распараллеливаемые приложения.

} // благодарю за внимание