Обработка коллекций: единая суть и множество проявлений

Вадим Винник

Об авторе

- SolarWinds MSP.
- Technical lead developer.
- C++, C# for work.
- Haskell just for fun.
- Domains:
  - Network traffic filtering.
  - Data backup and recovery.
  - Business process simulation.
- Ph.D., lecturer in universities.
- vadym.vinnyk@solarwinds.com.
- vadim.vinnik@gmail.com.

Постановка общей проблемы

- Сегодняшнее программирование разделено на множество областей.
- Ответвившись от единого истока, эти области далеко разошлись.
- Существует ли вообще концептуально единое программирование?
- На чём держится единство программирования?
- В чём его общезначимая суть, инвариантная во всех подразделах?

Общий ответ: композиционность

- “Истина рождается как ересь, а умирает как предрассудок” (Т.Гексли).
- На конференциях теперь уже звучит как прописная истина.
- Программирование - композиционно.
- Состыковывание сущностей для получения сложных сущностей.
- Базовые сущности и способы их стыковки - очень различны.
- Неизменно одно - композиционная природа программных сущностей.
- Композиции составляют логическую структуру программирования.
- Сущности (объекты, классы, модули) - наполняют эту структуру.
- Композиции - первичны.
- Программные сущности - вторичны, определяются композициями.

Примеры композиций

- Последовательное выполнение, ветвление и цикл.
- Подпрограммы, модули и библиотеки.
- Перенаправление ввода-вывода в Unix shell
  - `cat *.txt | grep pattern | wc -l`
- DLL. COM. Assembly (.NET).
- Инкапсуляция - композиция данных и алгоритмов в одном объекте.
- Полиморфизм - композиция реализаций под общим интерфейсом.
- RESTful API - композиция клиентов и серверов.
- Интеграция компонентов контейнером IoC.
- Шаблоны ООП (примеры: стратегия, посетитель).

Отступление о структурализме

- Идея о первичности отношений, пришла из философского течения.
- Привычный подход: первичны вещи, над ними надстроены отношения.
- Противоположный подход: первичны отношения.
- Вещи - лишь узлы в ткани отношений.
- Отношения - форма существования вещей.
- Вещь существует лишь в той мере, в которой находится в определённых отношениях с другими вещами.
- Вещь в себе - всё равно что не существует.
- Существовать - значит соотноситься.
- Пример. Знание о мире - не сумма фактов, а целостная картина.
- Отдельные факты менее важны, чем структура в целом.

От структуралализма - к теории программирования

- Сущности индивидуализируются своими отношениями.
- В программировании отношения - это композиции.
- Всё, что можно с сущностями делать, определяется композициями;
  - Как строить новые;
  - Как использовать существующие;
  - Что о сущностях вообще можно знать.
- Каждая прикладная область программирования характеризуется своим набором композиций.
- Адекватно понять и описать прикладную область программирования - значит найти систему композиций, вскрывающих её суть.
- Если эта концептуальная задача решена, то кодирование - дело техники.

Уточнённая постановка задачи

- Показать, как эволюционируют системы композиций.
- На примере одного узкого, но важного семейства задач.
- Цель - не научить итерированию по коллекциям, а показать, что…
- Для одного класса задач существует много систем композиций.
- Некоторые из них скрывают или искажают суть предметной области.
- Некоторые - адекватно отражают эту суть.
- Пользуясь определённым методом, можно от первых перейти к вторым.

Обработка коллекций

- Элементы - однотипны.
  - Полиморфизм не запрещён…
  - но скрыт за единым для всех элементов интерфейсом.
- Элементы образуют линейную последовательность.
  - На уровне внутренней реализации структура данных может быть любой.
  - Однако на уровне интерфейса коллекция выглядит как последовательность.
- Последовательность - не обязательно конечна.
  - В ходе выполнения программы обрабатывается конечное число элементов.
  - Однако оно может не быть ограничено заранее.
  - Бесконечные списки характерны для ленивых языков, особенно - функциональных.
  - Коллекция не хранит элементы, а генерирует их по мере необходимости.
- Однократный поэлементный проход по коллекции.

Примеры

- Наибольший, наименьший элемент.
- Сумма, среднее арифметическое.
- Пропуск заданного числа элементов.
- Обрыв последовательности по условию.
- Фильтрация по условию.
- Преобразование значений (map, transform).

Примитивные решения: массив и файл

```
int sum_array(
  int const* p,
  size_t n
) {
  size_t i;
  s = 0;
  for (i = 0; i != n; ++i)
    s += p[i];
  return s;
}
```

```
int sum_file(int fd)
{
  int s = 0;
  unsigned char x;

  while (1 == read(fd, &x, 1))
    s += x;

  return s;
}
```

Примитивные решения: список

```
typedef struct node_t_
{
  int value;
  struct node_t_ *next;
} node_t;
```

```
int sum_list(node_t const* p)
{
  int s = 0;
  while (NULL != p) {
    s += p->value;
    p = p->next;
  }

  return s;
}
```

Примитивные решения

- Простота. Все использованные конструкции и понятия известны начинающим программистам.
- Концептуально - эти алгоритмы делают одно и то же: вычисляют сумму линейной последовательности чисел.
- Реализация выглядит совершенно по-разному.
- За различиями проявлений не просматривается единство сути.
- Сильная зависимость от реализации.
- Много кода нужно переписать, если меняется структура данных.
- Алгоритм плохо подходит для повторного использования.

Универсальное решение

```
void traverse(
  (*is_end)(void*),
  (*shift)(void*),
  (*action)(
      void* src,
      void* state),
  void *src,
  void *state
)
```

```
{
  while (!is_end(src))
  {
    action(src, state);
    shift(src);
  }
}
```

Универсальное решение: список

```
typedef struct list_t_
{
  node_t *curr;
} list_t;

int lst_end(void *p)
{
  list_t *t = (list_t*)p;
  return NULL == t->curr;
}
```

```
void lst_shift(void *p) {
  list_t *t = (list_t*)p;
  t->curr = t->curr->next;
}

void lst_sum(void *p, void *q)
{
  list_t *t = (list_t*) p;
  int *s = (int*) q;
  *s += t->curr->value;
}
```

Универсальное решение: список

```
node_t c = { 30, NULL };
node_t b = { 20, &c };
node_t a = { 10, &b };

list_t list = { &a };
int s1 = 0;
```

```
traverse(
    lst_end,
    lst_shift,
    lst_sum,
    &list,
    &s1);
```

Универсальное решение: массив

```
typedef struct arr_t_ {
  int *data;
  size_t len;
  size_t idx;
} arr_t;

int arr_end(void *p) {
  arr_t *t = (arr_t*) p;
  return t->idx == t->len;
}
```

```
void arr_shift(void *p) {
  arr_t *t = (arr_t*) p;
  ++ t->idx;
}

void arr_sum(void *p, void *q)
{
  arr_t *t = (arr_t*) p;
  int *s = (int*) q;
  *s += t->data[t->idx];
}
```

Универсальное решение: массив

```
int m[4] = { 3, 4, 5, 6 };

arr_t arr = {
  m, /* data */
  4, /* len  */
  0  /* idx  */
};

int s2 = 0;
```

```
traverse(
    arr_end,
    arr_shift,
    arr_sum,
    &arr,
    &s2);
```

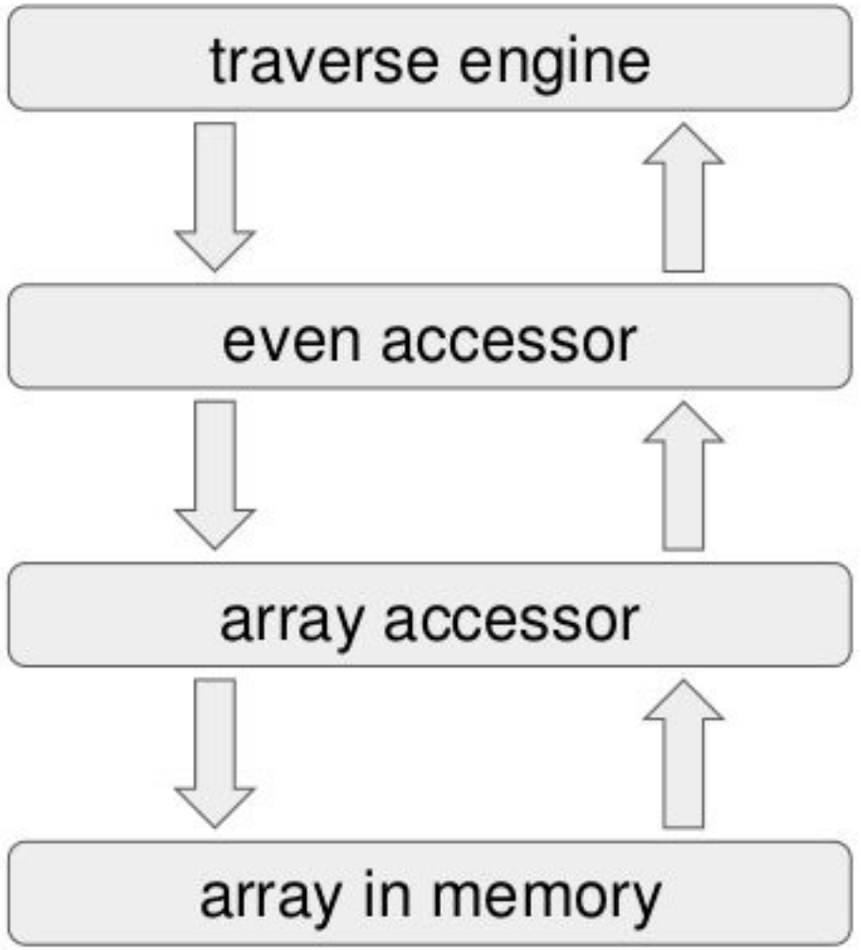
Композиция адаптеров. Виртуальная коллекция

```
typedef struct even_t {
  void *src;
  int (*is_end)(void*);
  void (*shift)(void*);
  void (*act)(void*, void*);
} even_t;

int even_end(void *p) {
  even_t *t = (even_t*) p;
  return t->is_end(t->src);
}
```

```
void even_shift(void *p) {
  even_t *t = (even_t*) p;
  t->shift(t->src);
  if (!t->is_end(t->src))
    t->shift(t->src);
}
void even_act
(void *p, void *q) {
  even_t *t = (even_t*) p;
  t->act(t->src, q);
}
```

Композиция доступа. Виртуальная коллекция

```
even_t even = {
  &arr,
  arr_end,
  arr_shift,
  arr_sum
};

int s3 = 0;
```



Анализ

- Обход различных коллекций выглядит единообразно.
- Различия структур данных инкапсулированы в адаптерах доступа.
- Замена структуры данных требует лишь замены адаптеров.
- Композиция адаптеров позволяет гибко строить виртуальные коллекции.
- Шаблоны проектирования на языке без прямой поддержки ООП.
- Шаблоны “Итератор”, “Стратегия” и “Декоратор”.
- Более сложный для понимания код.
- Концептуально связанные детали разнесены по разным функциям.
- Простота использования - ценой сложной подготовки к использованию.

Контейнеры, итераторы и алгоритмы в STL

- Внутренняя реализация сложна, но смотреть внутрь необходимости нет.
- Можно доверять корректности реализации.
- Единый интерфейс для различных структур данных (список, массив…).
- Единая абстракция: последовательность задана парой итераторов.
- Универсальный алгоритм обхода коллекции: `std::for_each`.
- Специализированные алгоритмы: `count`, `all_of`, `accumulate`, `copy`.
- Итератор “ходит” лишь по контейнерному объекту.
- Не хватает виртуальных коллекций (фильтр, преобразователь).

Декоратор для STL своими руками

- Последовательность элементов типа T задана парой итераторов [a, b).
- Задано отношение порядка “<” над объектами типа T.
- Задан унарный предикат p над объектами типа T.
- Найти итератор, указывающий на элемент диапазона [a, b), наименьший (в смысле отношения “<”) среди тех, что удовлетворяют условию p…
- … или итератор b, если ни один элемент из [a, b) не удовлетворяет p.
- Концептуально:
  - Построить виртуальную последовательность, наложив фильтр по условию p на [a, b).
  - В полученной последовательности найти наименьший элемент.
- Пример: в наборе чисел {131, 17, 200, 507, 64, 7, 92, 108} найти наименьшее среди нечётных.

Идея решения

- Построить новое отношение порядка “$\prec$” с такими свойствами:
  - Если $p(x) \wedge p(y)$, то $x \prec y \Leftrightarrow x < y$.
  - Если $\neg p(x) \wedge \neg p(y)$, то $x \prec y \Leftrightarrow x < y$.
  - Если $p(x) \wedge \neg p(y)$, то $x \prec y$.
  - Если $\neg p(x) \wedge p(y)$, то $y \prec x$.
- Все объекты типа T, удовлетворяющие предикату p, считаются меньшими (в смысле “$\prec$”), чем не удовлетворяющие предикату.
- В множестве объектов, удовлетворяющих p, сохраняется порядок “<”.
- Остаётся найти наименьший относительно порядка “$\prec$”.
- В терминах программирования: отношение “<” обернуть декоратором.

```
struct trait_min {
  static bool select(bool x, bool y) {
    return x;
  }
};

// also trait_max - returns y

template <
    class Pr, // predicate to filter
by
    class Re, // ordering relation
    class Tr> // trait_(min|max)
class cond_compare {
private:
  Pr pred_;
  Re rel_;
```

```
public:
  cond_compare(Pr pred, Re rel)
      : pred_(pred), rel_(rel)
  {}

  template <class T>
  bool operator()(
    T const& x, T const& y
  ) const {
    bool b1 = pred_(x);
    bool b2 = pred_(y);
    return b1 == b2
      ? rel_(x, y)
      : Tr::select(b1, b2);
    }
};
```

Анализ

```
cond_compare<Pr, Re, trait_min> c(pred, compare);
auto i = std::min_element(first, last, c);
```

- Прост в использовании и несложен в реализации.
- Согласуется с композиционным стилем.
- Может сочетаться с другими декораторами.
- Альтернатива - декоратор итератора, пропускающий элементы.
- Отсутствует в STL.
- Перекрывается возможностями Boost.

Концептуальные недостатки STL

- В самодостаточные сущности оформлены лишь контейнеры (список, вектор и др.).
- Диапазон моделируется не объектом, а парой итераторов.
- Для виртуальных коллекций нужны решения ad hoc.
- Понятие итератора одно, а роли - две.
- “Ошибки возникают по двум причинам: одну вещь называют разными именами или одним именем называют разные вещи”.
- Итератор указывает на элемент или на границу между элементами?
- Подробности - в докладе А.Шёдля [https://youtu.be/vrCtS6FDay8]:

Boost.Range. Пример - вместо тысячи слов

```
void f(std::vector<int> const& v) {
  auto w = v
    | boost::adaptors::reversed
    | boost::adaptors::uniqued
    | boost::adaptors::filtered([](int x){ return x%10 > 5; })
    | boost::adaptors::transformed([](int x){ return x*x; });

  boost::copy(w, std::ostream_iterator<int>(std::cout));
}
```

Свёртка: примеры

- Сумма последовательности чисел $s = \sum\{a_1,\ldots,a_n\}$.
- Рекуррентное определение:
  - $s_0 = 0$;
  - $s_{k+1} = s_k + a_{k+1}$;
  - $s = s_n$.
- Пример: $s_3 = s_2+a_3 = (s_1+a_2)+a_3 =(((s_0+a_1))+a_2)+a_3 =(((0+a_1))+a_2)+a_3$.
- Это - левая свёртка по операции “+” с начальным значением 0.
- Правая свёртка - симметрично.
- Например: $a_1+(a_2+(a_3+0)))$.

Свёртка, общее определение

- $f : U \times T \to U$.
- $a \in U$.
- (foldl f a) : $[T] \to U$.
- (foldl f a) [] = a.
- (foldl f a) (t:ts) = (foldl f (f a t)) ts.

- $g : T \times U \to U$.
- $a \in U$.
- (foldr g a) : $[T] \to U$.
- (foldr g a) [] = a.
- (foldr g a) (t:ts) = g t (foldr g a) ts.

- Функция высшего порядка: аргумент и значение суть функции.
- Математически - совершенно симметричны.
- Одинаково ли удобны для программной реализации через рекурсию?

Сведение различных алгоритмов к свёрткам

- Наибольший элемент последовательности целых:
  - свёртка по функции `std::max<int>`
  - с начальным значением `std::numeric_limits<int>::min()`.
- Удовлетворяют ли все элементы типа T предикату p:
  - свёртка по `[p](bool b, T x) { return b && p(x); }`,
  - начальное значением `true`.
- Переворачивание контейнера:
  - свёртка по `[](std::list<T> x, T y) { x.push_front(y); return x; }`,
  - начальное значение `std::list<T>()`.
- Применение последовательности функций, $f_n(...f_2(f_1(x))...)$:
  - свёртка по `[](T x, std::function<T(T)> f) { return f(x); }`,
  - начальное значение - x.

Поддержка свёрток в C++

- STL: `std::accumulate(begin, end, initval, func)`.
- Для правой свёртки - использовать `rbegin` и `rend`.
- Boost: последовательность задана объектом, а не парой итераторов:
- Не хватает - досрочного прерывания (аналог `break`).
- Прямая поддержка в C++17 для вариадических шаблонов! Пример:

```
template<typename ...Args> auto sum(Args ...args)
{
    return (args + ... + 0);
}
```

Что дальше?

- Обратная операция - развёртка, unfoldr.
  - Поддержка в C++ не обнаружена, зато есть в Haskell.
  - Тип: `(b → Maybe (a, b)) → b → [a]`,
  - Если (f x) == Nothing, то (unfoldr f x) == [],
  - Иначе (f x) == Just (u, y), тогда (unfoldr f x) == u : (unfoldr f y).
- Свёртки\развёртки естественно обобщаются для деревьев и др.
- Обобщение свёртки - катаморфизм.
- Обобщение развёртки - анаморфизм.
- Развёртка и затем свёртка? Гиломорфизм!
- Широко применяются в функциональном программировании.
- Раз уж C++ приобретает черты функционального языка...

Итоги

- Для любой области прикладных задач существует множество решений.
- Решение - это набор структур данных, алгоритмов, функций, классов.
- Решение должно отражать логику, внутренне присущую задачам.
- Если задачу приходится подгонять под инструмент - инструмент плох.
- Качество набора инструментов определяется их композиционностью.
- Композиционная обработка коллекций - в Boost.
- Свёрточно-развёрточный стиль представляет интерес.
- В C++ пока поддерживается слабо.
- Ждём нововведений?

}