ИТЕРАТОРЫ

Обобщённый обход контейнеров. Собственные итераторы.
Инвалидация итераторов. Характеристики итераторов.

К. Владимиров, Syntacore, 2023
mail-to: konstantin.vladimirov@gmail.com

- Простые прикладные итераторы
- Преобразования и адаптеры
- Инвалидация
- Перемещающие итераторы

Первый пример: обход вектора

- Задача: пока функция `func` возвращает `true` применять её к элементам вектора.

```
template <typename F>
size_t traverse(vector<int>& v, F func) {
  size_t nelts = v.size();
  for (size_t i = 0; i != nelts; ++i)
    if (!func(v[i]))
      return i;
  return nelts;
}
```

- Видите ли вы проблемы в этом решении?

Обобщение обхода

- Задача: пока функция `func` возвращает `true` применять её к элементам произвольного контейнера.

```
template <typename Cont, typename F>
size_t traverse(Cont& cont, F func) {
  size_t nelts = cont.size();
  for (size_t i = 0; i != nelts; ++i)
    if (!func(cont[i]))
      return i;
  return nelts;
}
```

- Что если `Cont` это `std::list`?

Обобщение обхода

- Задача: пока функция `func` возвращает `true` применять её к элементам произвольного контейнера.

```
template <typename Cont, typename F>
size_t traverse(Cont& cont, F func) {
  size_t elts = 0
  for (auto it = cont.begin(); it != cont.end(); ++it, ++elts)
    if (!func(*it))
      break;
  return elts;
}
```

- Теперь подойдёт любой стандартный контейнер.

Range-based обход

- Концепция итератора может быть скрыта под капотом.

```
template <typename C, typename F>
size_t traverse (C&& cont, F func) {
  size_t nelts = 0;
  for (auto&& elt : cont)
    if (!(++nelts, func(elt))) // elt это *it
      break;
  return nelts;
}
```

- Тут очевидны две ответственности этого цикла.

Range-based обход

```
for (init-statement; range_declaration : range_initializer)
  loop_statement;
```

- Эквивалентно следующему

```
auto && __range = range_initializer;

auto __begin = begin(__range); // не обязательно std::begin

auto __end = end(__range);     // не обязательно std::end

for ( ; __begin != __end; ++__begin) {
  range_declaration = *__begin;
  loop_statement
}
```


https://godbolt.org/z/WsG59Y8Ex

Требования к range-based обходу

- Объект возвращаемый `std::begin()` должен поддерживать:
  - инкремент
  - разыменование
  - сравнение на неравенство

```
for ( ; __begin != __end; ++__begin) {
   range_declaration = *__begin;
```

- Эти требования входят в статический интерфейс (concept) прямого итератора
- Можно заметить, что всем этим требованиям отвечают обычные указатели
- Очень важно: итератор это не какой-то класс и не наследник какого-то класса, это что угодно с этим интерфейсом

Указатели как итераторы

- Например почему бы и не указатели внутрь встроенных массивов?

```
int marr[6] = {1, 2, 3, 4, 5, 6};

// ranged-base traverse работает!
for (auto elt : marr) {
```

- Хотя тут у нас появляется один интересный вопрос: а как работает `std::begin` что внезапно для массивов он не пытается вызвать `marr.begin()`?
- Интересно тут также следующее: вообще-то указатели умеют куда больше, чем просто разыменование, инкремент и сравнение

Свойства указателей

- Создание по умолчанию, копирование, копирующее присваивание.
- Разыменование как rvalue и доступ к полям по разыменованию.
- Разыменование как lvalue и присваивание значения элементу под ним.
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декремент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности.

Output итераторы

- Создание по умолчанию, копирование, копирующее присваивание.
- Разыменование как rvalue и доступ к полям по разыменованию.
- Разыменование как lvalue и присваивание значения элементу под ним.
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декремент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности.

Input итераторы

- Создание по умолчанию, копирование, копирующее присваивание.
- Разыменование как rvalue и доступ к полям по разыменованию.
- Разыменование как lvalue и присваивание значения элементу под ним
- Инкремент и постинкремент за $O(1)$
- Сравнимость на равенство и неравенство за $O(1)$
- Декремент и постдекремент за $O(1)$
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за $O(1)$
- Многократный проход по одной и той же последовательности.

Forward итераторы

- Создание по умолчанию, копирование, копирующее присваивание.
- Разыменование как rvalue и доступ к полям по разыменованию.
- Разыменование как lvalue и присваивание значения элементу под ним.
- Инкремент и постинкремент за `O(1)`
- Сравнимость на равенство и неравенство за `O(1)`
- Декремент и постдекремент за `O(1)`
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за `O(1)`
- Многократный проход по одной и той же последовательности.

Bidirectional итераторы

- Создание по умолчанию, копирование, копирующее присваивание.
- Разыменование как rvalue и доступ к полям по разыменованию.
- Разыменование как lvalue и присваивание значения элементу под ним.
- Инкремент и постинкремент за `O(1)`
- Сравнимость на равенство и неравенство за `O(1)`
- Декремент и постдекремент за `O(1)`
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за `O(1)`
- Многократный проход по одной и той же последовательности.

Random-access итераторы

- Создание по умолчанию, копирование, копирующее присваивание.
- Разыменование как rvalue и доступ к полям по разыменованию.
- Разыменование как lvalue и присваивание значения элементу под ним.
- Инкремент и постинкремент за `O(1)`
- Сравнимость на равенство и неравенство за `O(1)`
- Декремент и постдекремент за `O(1)`
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за `O(1)`
- Многократный проход по одной и той же последовательности.

Итераторы: дело в асимптотике

- Инкремент и постинкремент за `O(1)` `// forward`
- Сложение с целым за `O(1)` `// random-access`
- Довольно очевидно, что для forward итератора в общем случае продвижение на произвольное расстояние это `O(N)`
- Есть функции, которые прячут это под капотом:

```
std::distance(Iter fst, Iter snd); // snd - fst, либо цикл
```

```
std::advance(Iter fst, int n); // fst + n, либо цикл
```

- Они делают это устраивая явную перегрузку по тегу категории.

Обсуждение

- Учитывая возможную плохую асимптотику distance, этот код может быть чуть хуже явного цикла.

```
template <typename C, typename F>
size_t traverse (C&& cont, F func) {
  auto it = std::find_if_not(cont.begin(), cont.end(), func);
  return std::distance(cont.begin(), it);
}
```

- Но может быть он чем-то лучше?

Обсуждение: используйте итераторы

- Этот пример лучше тем, что показывает реальное требование: не контейнер, а два итератора.

```
template <typename It, typename F>
size_t traverse (It start, It fin, F func) {
  auto it = std::find_if_not(start, fin, func);
  return std::distance(start, it);
}
```

- Есть ли в действительности разница по скорости?
- Да и внезапно она бывает просто огромная.

matrix_repro.cc

Quick bench results

Определение категории итераторов

- Используется класс характеристик.

```
typename iterator_traits<Iter>::iterator_category
```

- Возможные значения.
- `input_iterator_tag`
- `output_iterator_tag`
- `forward_iterator_tag: public input_iterator_tag`
- `bidirectional_iterator_tag: public forward_iterator_tag`
- `random_access_iterator_tag: public bidirectional_iterator_tag`

Перегрузка по тегу

- Например перегрузим вывод для тегов чтобы отлаживать наши программы.

```
ostream& operator << (ostream& out, random_access_iterator_tag) {
  out << "random access"; return out;
}

// .... и так далее для всех тегов ....

template <typename Iter> void print_iterator_type() {
  cout << iterator_traits<Iter>::iterator_category{} << endl;
}
```

- Теперь мы легко узнаем например категорию для деков.

```
print_iterator_type<typename deque<int>::iterator>();
```


categories.cc

Проверка категории

- Иногда мы хотим обложить перегрузку SFINAE проверкой.

```
template <typename It>
// имеет смысл только для input iterators
void foo(It first, It last)
```

- Поможет ли нам здесь `void_t`?

Интерлюдия: conditional_type

- Рассмотрим следующую sfinae-триаду

```
template <bool B, typename T, typename F>
struct conditional { using type = T; }

template <typename T, typename F>
struct conditional<false, T, F> { using type = F; }

template <bool B, typename T, typename F>
using conditional_t = typename conditional<B, T, F>::type;
```

- Она представляет собой условный тип.
- Можно ли сделать его невалидным для F?

Условный тип

- Рассмотрим следующую sfinae-триаду

```
template <bool B, typename T, typename F>
struct conditional { using type = T; }

template <typename T, typename F>
struct conditional<false, T, F> { using type = F; }

template <bool B, typename T, typename F>
using conditional_t = typename conditional<B, T, F>::type;
```

- Можно ли сделать его невалидным для F?
- Да, просто вычеркнем технически все упоминания `false-type`

ENABLE_IF

- Получившаяся триада `enable_if` является одной из самых полезных идиом в практическом SFINAE

```
template <bool B, typename T = void>
struct enable_if { using type = T; }

template <typename T = void>
struct enable_if<false, T> { }

template <bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

- Выкинув false, сделаем true примитивным, например void

Проверка категории

- Иногда мы хотим обложить перегрузку SFINAE проверкой

```
template <typename It>
using iterator_category_t =
        typename std::iterator_traits<It>::iterator_category;

template <typename It, typename T = std::enable_if_t<
    std::is_base_of_v<input_iterator_tag,
                      iterator_category_t<It>>>>
void foo(It first, It last)
```

- Все ли понимают, почему base of, а не same?

https://godbolt.org/z/Ma7Pn8MrY

Обсуждение

- Неплохой вектор с плохим итератором.

Case study: пишем свой итератор

- Постановка задачи: итерирование сразу по двум контейнерам

```
std::vector<int> keys = {1, 2, 3, 4};
std::vector<double> values = {4.0, 3.0, 2.0, 1.0};

for (auto &&both : make_zip_range(keys, values))
  std::cout << both.first << ", " << both.second << "; ";

// 1, 4.0; 2, 3.0; 3, 2.0; 4, 1.0
```

- Нужно придумать легковесную обёртку zip_range и возвращаемые ей итераторы (тип для них)

Пишем свой итератор: подготовка

- Создание zip_range очень просто

```
template<typename Keys, typename Values>
auto make_zip_range(Keys& K, Values &V) {
  return zip_range_t<Keys, Values>{K, V};
}
```

- И сам он очень прост, сложности только с типом итератора.
- Что должен внутри себя хранить zip range?

Пишем свой итератор: тело

- Тело тоже не представляет проблем

```
template<typename Keys, typename Values>
class zip_range_t {
  Keys &K_; Values &V_;

public:
 zip_iterator_t<KIter, VIter> begin() {
    return make_zip_iterator(std::begin(K_), std::begin(V_));
 }

// тут должно быть что-то
```

- Что вы будете писать дальше?

Пишем свой итератор: первые шаги

- В нашем итераторе нам нужно определить пять фундаментальных подтипов
  - **iterator_category** – категория нашего итератора
  - **difference_type** – тип для хранения разности итераторов
  - **value_type** – тип значений, по которым мы итерируемся
  - **reference** – тип ссылки на значения, по которым мы итерируемся
  - **pointer** – тип указателя на значения, по которым мы итерируемся
- Как вы думаете как мы их определим в нашем случае?

Простые вещи

- Некоторые вещи действительно просты

```
// вспомогательные using для value_type составных частей
using KeyType = typename iterator_traits<KeyIt>::value_type;
using ValueType = typename iterator_traits<ValueIt>::value_type;

// наше value это пара values
using value_type = std::pair<KeyType, ValueType>;
```

- К сожалению так нельзя определить тип pointer, потому что мы на самом деле не итерируемся по контейнеру пар
- Мы вернёмся к этому довольно скоро

Базовый интерфейс

- Нет никаких проблем чтобы попарно увеличивать и уменьшать итераторы

```
zip_iterator_t(KeyIt Kit, ValueIt Vit) : Kit_(Kit), Vit_(Vit) {}
zip_iterator_t &operator++() { ++Kit_; ++Vit_; return *this; }
zip_iterator_t &operator++(int) { тоже ничего сложного }
```

- Первая засада ждёт на операторе разыменования

```
using reference = std::pair<KeyType&, ValueType&>;
reference operator*() const { return {*Kit_, *Vit_}; }
```

- Будет ли это работать?

Всегда пользуйтесь traits

- Очевидно:

```
using reference = std::pair<KeyType&, ValueType&>;
```

- Это ошибка если в контейнере reference отличается от `value&`, например для `vector<bool>` и многих других
- Корректно:

```
using KeyRef = typename iterator_traits<KeyIt>::reference;
using ValueRef = typename iterator_traits<ValueIt>::reference;

using reference = std::pair<KeyRef, ValueRef>;

reference operator*() const { return {*Kit_, *Vit_}; // ok
```

Настоящая проблема: стрелочка

- Как вообще должен выглядеть оператор разыменования?

```
auto zit = make_zip_iterator(k.begin(), b.begin());
assert (k.front() == zit->first);

// zit->first drills down to (zit.operator->())->first
```

- Это должен быть аналог разыменованию и обращению к полю

```
pointer operator->() const { return some pointer; }
```

- Но что такое pointer? Простое решение не подходит

```
using pointer = std::pair<KeyPtr, ValuePtr>; // нет p->first
```

Некоторые дурацкие способы

- Можно продлить временному объекту жизнь сделав его статическим

```
using pointer = value_type*;

pointer operator->() const {
  static reference Ref;
  Ref = {*Kit_, *Vit_};
  return &Ref;
}
```

- Какие тут проблемы?
- Например рассмотрим `use(zit->first, (zit+1)->first)`

Изящное решение: прокси класс

- На помощь приходит прокси-класс

```
template <typename Reference> struct arrow_proxy {
  Reference R;
  Reference *operator->() { return &R; } // non const
};

using pointer = arrow_proxy<reference>;

pointer operator->() const { return pointer{{*Kit_, *Vit_}}; }
```

- Есть некие опасения в том что прокси провиснет, но нам он нужен чтобы пережить drill-down, а его явно переживёт

ziprange.hpp

Обсуждение

- Рассмотренный zip-range это типичный адаптер итератора
- Давайте поговорим о некоторых других

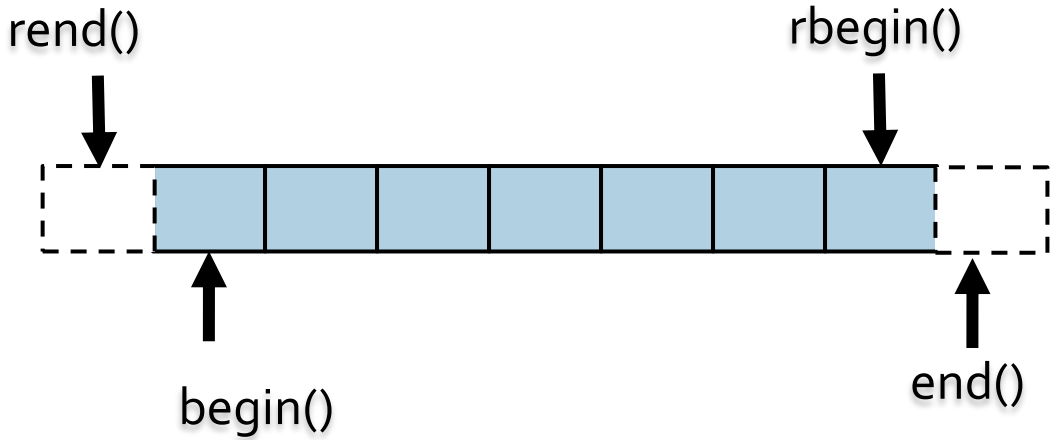
- Категории итераторов
  - Преобразования и адаптеры
- Инвалидация
- Перемещающие итераторы

Обсуждение

- Категории итераторов это не единственный признак, по которому они могут различаться
- Какие ещё признаки приходят на ум для различия итераторов внутри **одной и той же** категории, например bidirectional?

Направления и константность

- По направлению:
  - cont.begin()
  - cont.rbegin()
- Константные
  - cont.cbegin()
  - cont.crbegin()



Пример обратных итераторов

- Как получить вектор обратный данному?

```
vector<int> vecf = {1, 2, 3, 4, 5, 6};
```

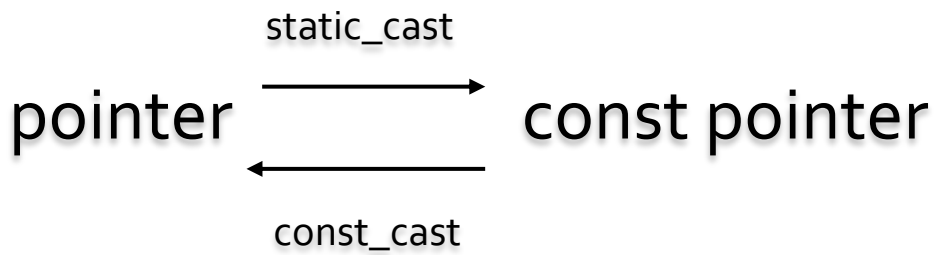
- Плохой вариант

```
vector<int> vecb { vecf.end(), vecf.begin() };
```

- Хороший вариант

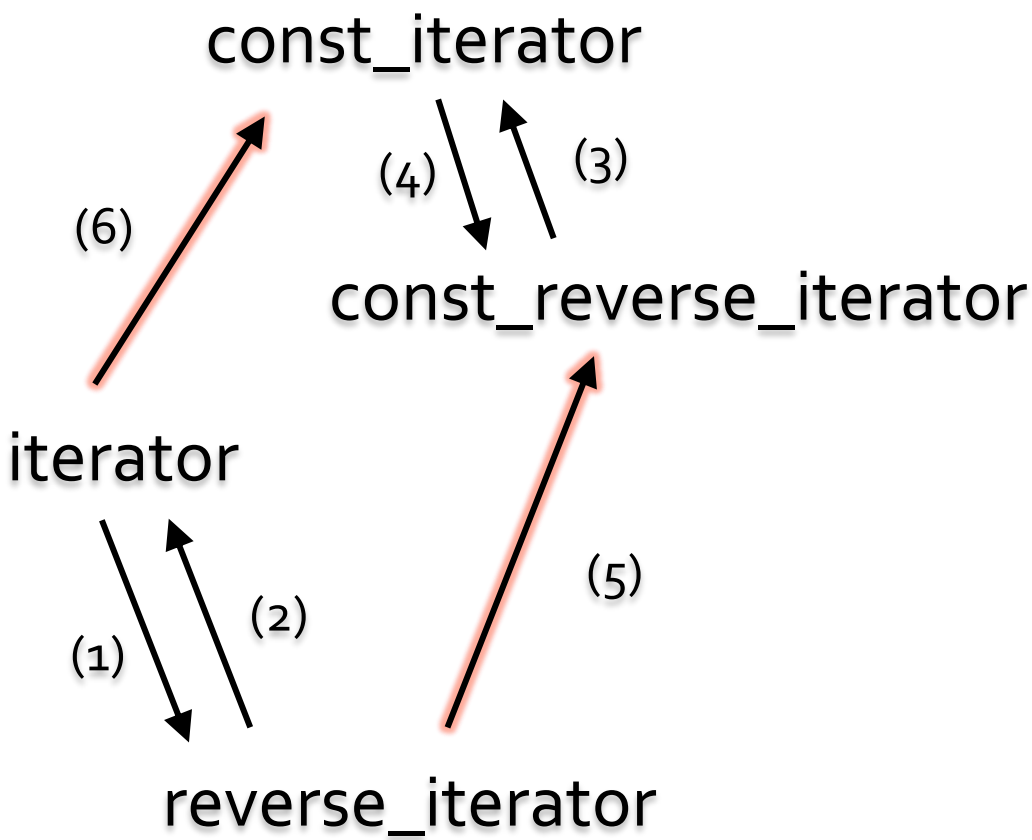
```
vector<int> vecb { vecf.rbegin(), vecf.rend() };
```

Преобразования указателей



- Она так проста потому что указатели ковариантны к константности
- Увы, итераторы инвариантны и могут не иметь вообще ничего общего
- Как будет выглядеть (видимо более сложная) диаграмма преобразований итераторов?

Диаграмма Майерса




```
(1,2,3,4) Обращение итератора
auto rit = make_reverse_iterator(it);
auto it = rit.base();
(5,6) Добавление константности
Cont::const_iterator cit = it;
Cont::const_reverse_iterator crit =
rit;
```

Предложение Майерса

- Актуальная проблема: `const_cast` для итераторов. То есть как привести `const_iterator` к обычному?
- Майерс предлагает использовать advance

```
Iter i(cont.begin());

std::advance(i, std::distance<decltype(ci)>(i, ci));
```

- Вопросы:
  - Зачем явно указан шаблонный параметр?
  - Проблемы с этим подходом?

Предложение Майерса

- Актуальная проблема: `const_cast` для итераторов. То есть как привести `const_iterator` к обычному?
- Майерс предлагает использовать advance

```
Iter i(cont.begin());

std::advance(i, std::distance<decltype(ci)>(i, ci));
```

- Явный шаблонный параметр, чтобы избежать неоднозначного вывода типов.
- Основная проблема: время O(N) для "неудачных" контейнеров, таких, как списки

meyers.cc

Трюк Хинанта

- Изящная юридическая казуистика из серии "не знаешь – не угадаешь"

```
template <typename Container, typename ConstIterator>
typename Container::iterator
remove_constness(Container& c, ConstIterator it) {
  return c.erase(it, it);
}
```

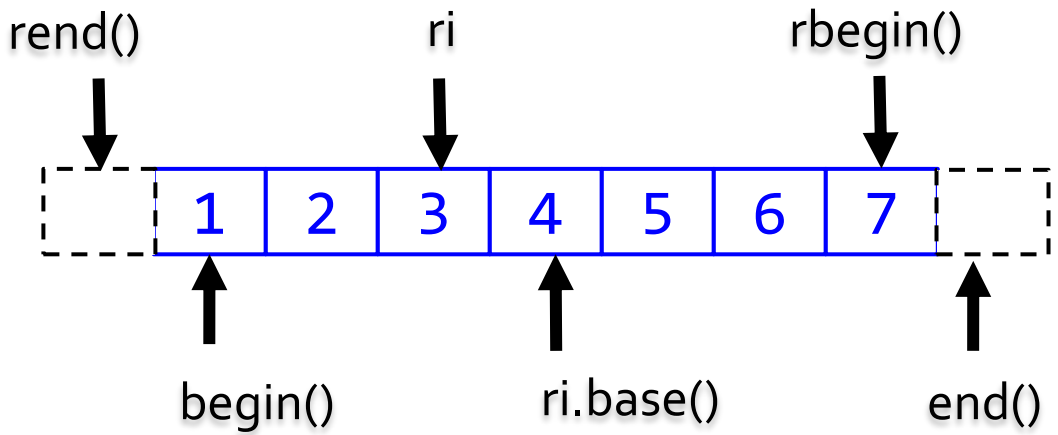
- Идея в том, что начиная с C++11, удаление пустого диапазона позволено, не делает ничего и возвращает iterator
- Это работает за O(1) но не работает для обратных итераторов и для строк

Переход к прямому итерированию

```
std::vector v {1, 2, 3, 4, 5, 6, 7};
auto ri = v.rbegin() + 4;
auto it = ri.base();
cout << *ri << " " << *it << endl; // что на экране?
```

Переход к прямому итерированию

```
std::vector v {1, 2, 3, 4, 5, 6, 7};
auto ri = v.rbegin() + 4;
auto it = ri.base();
cout << *ri << " " << *it << endl; // 3 4
```



Адаптация: обратный range-based обход

- Задача: сделать адаптер reverse_cont, такой, чтобы работал цикл:

`for (auto &&elt : vec)` – обойти в прямом порядке

`for (auto &&elt : reverse_cont(vec))` – обойти в обратном порядке

Реализация reverse_cont

```
template <typename T> struct reversion_wrapper {
  T& iterable;
};

template <typename T> auto begin(reversion_wrapper<T> w) {
  return rbegin(w.iterable);
}

template <typename T> auto end(reversion_wrapper<T> w) {
  return rend(w.iterable);
}

template <typename T>
reversion_wrapper<T> reverse_cont(T&& iterable) {
  return { iterable };
}
```

Обсуждение

- Это разительно отличается от полноценного zip range
- Тут мы по сути переиспользуем обычные итераторы, меняется только обёртка

Адаптация: inserters

- Преобразование записи во вставку

```
std::vector<int> vec;

// тяжёлый способ
std::back_insert_iterator<std::vector<int>> bins(vec);

// лёгкий способ, похожий на reverse_cont выше
auto bins = std::back_inserter(vec);

*bins = 1; // вставка элемента, как vec.push_back(1)
```

- Что должен делать инкремент `bins++`?

Адаптация: inserters

- Преобразование записи во вставку

```
std::vector<int> vec;
auto bins = std::back_inserter(vec);
bins = 1; // вставка элемента, как vec.push_back(1)
```

- Что должен делать инкремент `bins++`?
- Практически ничего.
- Более того, даже разыменование `*bins` ничего осмысленного не делает. Поэтому работает также как показано выше.

Виды адаптеров вставки для итераторов

- `std::back_inserter` для вставки в конец (предпочтительно)
- `std::front_inserter` для вставки в начало (можно попасть на асимптотику)
- `std::inserter` для вставки в произвольное место (шансы на так себе асимптотику сильно увеличиваются)

```
std::vector<int> v = {2, 3, 7, 11};

auto it = std::find(v.begin(), v.end(), 3);

auto insit = std::inserter(v, it);

insit = 5; // теперь v = {2, 3, 5, 7, 11}
```

Пример: кросс-копирование

```
template <typename InpIter, typename OutIter>
OutIter cross_copy(InpIter fst, InpIter lst, OutIter dst) {
  while (fst != lst) { *dst = *fst; ++fst; ++dst; }
  return dst;
}

std::list<int> lst = {1, 2, 3, 4, 5, 6};
std::vector<int> vec;
```

Задача: скопировать содержимое списка `lst` в вектор `vec`

Пример: кросс-копирование

```
template <typename InpIter, typename OutIter>
OutIter cross_copy(InpIter fst, InpIter lst, OutIter dst) {
  while (fst != lst) { *dst = *fst; ++fst; ++dst; }
  return dst;
}

std::list<int> lst = {1, 2, 3, 4, 5, 6};
std::vector<int> vec;

vec.resize(lst.size());
cross_copy(lst.begin(), lst.end(), vec.begin());
```

Пример: кросс-копирование

```
template <typename InpIter, typename OutIter>
OutIter cross_copy(InpIter fst, InpIter lst, OutIter dst) {
  while (fst != lst) { *dst = *fst; ++fst; ++dst; }
  return dst;
}

std::list<int> lst = {1, 2, 3, 4, 5, 6};
std::vector<int> vec;

cross_copy(lst.begin(), lst.end(), std::back_inserter(vec));
```

Пример: кросс-копирование

```
template <typename InpIter, typename OutIter>
OutIter cross_copy(InpIter fst, InpIter lst, OutIter dst) {
  while (fst != lst) { *dst = *fst; ++fst; ++dst; }
  return dst;
}

std::list<int> lst = {1, 2, 3, 4, 5, 6};
std::vector<int> vec;

cross_copy(lst.begin(), lst.end(), std::back_inserter(vec));

cross_copy(vec.begin(), vec.end(),
           std::ostream_iterator<int>(std::cout, "\n"));
```

Похожая система неспроста.
И там и там output итераторы

Простая задача: снова cross-copy

```
template <typename InpIter, typename OutIter>
OutIter cross_copy(InpIter fst, InpIter lst, OutIter dst) {
  while (fst != lst) { *dst = *fst; ++fst; ++dst; }
  return dst;
}

std::list<int> lst = {1, 2, 3, 4, 5, 6};
std::vector<int> vec = {10, 20, 30, 40, 50, 60};


cross_copy(lst.begin(), lst.end(),
           inserter(vec, vec.begin() + 3)); // что в vec?
```

Простая задача: снова cross-copy

```
template <typename InpIter, typename OutIter>
OutIter cross_copy(InpIter fst, InpIter lst, OutIter dst) {
  while (fst != lst) { *dst = *fst; ++fst; ++dst; }
  return dst;
}

std::list<int> lst = {1, 2, 3, 4, 5, 6};
std::vector<int> vec = {10, 20, 30, 40, 50, 60};


cross_copy(lst.begin(), lst.end(),
           std::inserter(vec, vec.begin() + 3));
// vec == { 10, 20, 30, 1, 2, 3, 4, 5, 6, 40, 50, 60 }
```

Обсуждение

- Рассмотрим этот пример ещё раз

```
std::vector<int> vec = {10, 20, 30, 40, 50, 60};
auto i5 = vec.begin() + 5;
cross_copy(lst.begin(), lst.end(),
           std::inserter(vec, vec.begin() + 3));

// vec == { 10, 20, 30, 1, 2, 3, 4, 5, 6, 40, 50, 60 }

*i5 = 42;
```

- А теперь что в векторе?
- Кажется, мы тут наступили на нечто, не слишком приятное

- Категории итераторов
- Преобразования и адаптеры
- Инвалидация
- Перемещающие итераторы

Валидность итераторов

- Валидный итератор
  - конформно поддерживает все операции для своей категории итераторов
- Валидный диапазон
  - состоит из двух валидных итераторов
  - второй итератор достижим из первого

Задача: валиден ли диапазон?

```
std::istream_iterator<string> beg(ifstream("in.txt")), end;

cross_copy(beg, end,
           std::ostream_iterator<string>(ofstream("out.txt")));
```

Задача: валиден ли диапазон?

```
std::istream_iterator<string> beg(ifstream("in.txt")), end;

cross_copy(beg, end,
           std::ostream_iterator<string>(ofstream("out.txt")));
```

- Нет, здесь "висячий" итератор, который не валиден
- Итератор может быть невалиден по ряду причин:
  - Он не инициализирован (кроме сингулярных итераторов потоков)
  - Он подвис, т.е. ссылается на объект с истекшим сроком жизни
  - Он указывает за пределы диапазона
  - Он инвалидирован операциями над контейнером
  - Это использованный итератор ввода

Сингулярные итераторы

- Уже рассматривались ранее и иногда в них нет ничего плохого

```
std::istream_iterator<string> beg(ifstream("in.txt")), end;
```

- Здесь итератор end сингулярен, но вполне валиден.
- Но ниже ситуация хуже:

```
std::list<string>::iterator lstit;
cross_copy(vec.begin(), vec.end(), lstit);
```

- Здесь сингулярный итератор не валиден и случается UB
- Аналог – неинициализированный указатель.

Итераторы за границами диапазона

Два основных типа:

Итераторы, показывающие на end() или rend(), которые можно использовать но нельзя разыменовать (past-the-end)

```
std::list<int> lst;
auto past_end = lst.begin();

std::cout << *past_end << std::endl; // fail

lst.insert (past_end, 1); // ok
```

Итераторы, показывающие далеко после конца или раньше начала, с которыми нельзя делать ничего (out-of-range).

Инвалидация итераторов

- Итератор может быть инвалидирован операциями над контейнером

```
std::vector<int> v = {11, 7, 5, 3, 2};
auto vit = std::advance(v.begin(), 3);
v.clear();
std::cout << *vit << std::endl; // fail
```

- Здесь итератор может инвалидирован и куда он указывает не определено. К сожалению, если итератор реализован как указатель, он будет куда-то указывать и ошибка может быть "тихой".

Инвалидация итераторов

- Самое плохое, что она может быть плавающей. Простой пример:

```
std::vector<int> v = {2, 3, 5, 7, 11};
auto vit = std::advance(v.begin(), 3);
v.push_back(13);
std::cout << *vit << std::endl; // ok???
```

- Здесь итератор может быть инвалидирован, если ёмкость вектора закончилось и случился `realloc`. А может и не быть.

Правила базовой инвалидации

- Вставка
  - `vector` – сохраняются все итераторы до точки вставки кроме случаев перевыделения, кгда все инвалидированы
  - `deque` – все итераторы всегда инвалидированы
  - `list` – все итераторы сохраняются
- Удаление
  - `vector` – инвалидируются все итераторы после точки удаления, сохраняются до.
  - `deque` – инвалидируются все итераторы при удалении из середины. При удалении из начала или конца, все итераторы сохраняются (кроме итертаора на удаляемый).
  - `list` – сохраняются все итераторы кроме итератора на удаляемый элемент

Использованные итераторы

- Эта проблема касается в основном одноразовых итераторов, таких как input-итераторы

```
ifstream file("in.txt");
istream_iterator<string> beg(file), fin;
cross_copy(beg, fin, ostream_iterator<string>(cout)); // ok
vector<string> vec(beg, fin); // fail
```

Обсуждение

- Поскольку итераторы – более общая концепция по сравнению с указателями, возможных проблем с ними тоже больше
- Но решают они больше проблем, чем создают и в обобщённом коде альтернативы им нет
- Давайте поговорим о техниках, улучшающих использование итераторов в обобщённом коде

- Категории итераторов
- Преобразования и адаптеры
- Инвалидация
- Перемещающие итераторы

Библиотека std::filesystem

- Абстрагирует такие понятия как "директория", "расширение", "путь" и прочие отсутствовавшие в стандарте до 2017-го
- Пример задачи
- Вам дают некий путь (например к исходному файлу) в качестве argv[1]
- Вам нужно сохранить его в рабочей папке вашей программы и добавить расширение

```
int main(int argc, char **argv) {
  // assume argv[1] = ../../my/folder/f42.in
  // then we need to create file f42.out in current folder
```

Решение: создаём путь

- Самым полезным объектом является path

```
std::filesystem::path p(argv[1]);
```

- Теперь с этим путём можно развлекаться

```
auto oldfn = p.filename(); // → std::filesystem::path
auto newname = oldfn.replace_extension(".out").string();
```

- Позволяет рационально решать задачи для которых раньше требовали странные хаки и платформенно-зависимые решения
- Абстрагирует особенно Windows и Unix-based систем

Итератор по содержимому директории

- Имея путь мы можем сделать итерацию по содержимому папки

```
std::filesystem::directory_iterator start{path}, fin;
```

- Обратите внимание итератор конца сингулярный (как для потоков ввода)

```
std::vector entries(start, fin); // → vector<directory_entry>
```

- Далее у каждого из entries[i] можно в свою очередь получить path и т. д.
- Тут интересно что `directory_iterator` предоставляет не только интерфейс итератора, но и интерфейс диапазона

directory_iterator

Итератор по содержимому директории

- Имея путь мы можем сделать итерацию по содержимому папки

```
std::filesystem::directory_iterator content{path};
```

- Интересно что тут это не итератор, это обёртка

```
std::vector entries(content.begin(), content.end());
```

- Далее у каждого из entries[i] можно в свою очередь получить path и т. д.
- Предположим, что нужно написать программу, которая брала бы рабочую папку и искала всё содержимое в её подпапках первого уровня

Обход первого уровня: первый вариант

- Первый вариант довольно прост

```
std::vector<directory_entry> contents(directory_entry d);

auto files_in_subdirs() {
  std::filesystem::directory_iterator start{"."};
  std::vector<std::filesystem::directory_entry> res;
  for (auto &&item : start)
    if (item.is_directory()) {
      auto ents = contents(item);
      res.insert(res.end(), ents.begin(), ents.end());
    }
  return res;
}
```

Обход первого уровня: проблема

- Но в первом варианте явно есть ненужное копирование

```
if (item.is_directory()) {
  auto ents = contents(item);
  res.insert(res.end(), ents.begin(), ents.end()); // copy
} // тут ents умрёт
```

- Мы можем его улучшить, сделав move-итераторы

```
res.insert(res.end(),
  std::make_move_iterator(ents.begin()),
  std::make_move_iterator(ents.end())); // move
```

- Эти итераторы работают вполне предсказуемо, заменяя копирование перемещением

Некая двойственность

- Мы можем сделать перемещение через итераторы к стандартному алгоритму

```
std::copy(std::make_move_iterator(src.begin()),
          std::make_move_iterator(src.end()),
          dst.begin());
```

- Или через специализированный алгоритм

```
std::move(src.begin(), src.end(), dst.begin());
```

- Обратите внимание на прекрасную перегрузку move
- Как бы вы предпочли сделать? Как бы вы учли move-итераторы в вашей SFINAE реализации для почти-std::copy?

Литература

- Information technology – Programming languages – C++, ISO/IEC 14882, 2017
- Bjarne Stroustrup – The C++ Programming Language (4th Edition)
- Scott Meyers – Effective STL, 50 specific ways to improve your use of the standard template library, 2001
- Scott Meyers – Effective Modern C++, O'Reilly, 2014
- Davide Vandevoorde, Nicolai M. Josuttis – C++ Templates. The Complete Guide, 2nd edition, Addison-Wesley Professional, 2017
- Casey Carter – Iterator Haiku, CppCon, 2016
- Patrick Niedzelski – Building and Extending the Iterator Hierarchy in a Modern, Multicore World, CppCon, 2016