

АЛГОРИТМЫ

Введение в алгоритмы стандартной библиотеки и обобщённое
программирование

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

➤ Функторы и состояние

- ❑ Алгоритмы

- ❑ No raw loops

- ❑ Case study: группы перестановок

Обсуждение: for each

- У нас есть метод `equal_range` в `unordered_multimap`, который возвращает диапазон пар ключ-значение с одинаковым ключом.

```
std::unordered_multimap<int, int> m;  
....  
auto [begin, end] = m.equal_range(i);
```

- Вы можете как-то проитерироваться и что-то сделать с каждым значением.

```
for (auto it = begin; it != end; ++it) f(*it); // 1  
std::for_each(begin, end, f);                // 2
```

- Какой из двух вариантов вы выберете и почему?

Обсуждение

- Выбирать всегда следует алгоритм стандартной библиотеки
- Аргумент от тела цикла

```
for (auto elt : cont) {  
    // позволяет неконтролируемо вставить массу кода  
}
```

- Аргумент от распараллеливания (C++17)

```
for_each(std::execution::par, cont.begin(), cont.end(), func);
```

- В общем случае абстракция циклов повышает и читаемость и эффективность
- Такие абстракции называются **абстракциями с отрицательной стоимостью**

Обсуждение

- Представим, что нам надо как-то использовать внешнее значение.
- Например пусть нам надо умножить value на некое число.

```
int mult = get_my_mult(); // очень долгая функция
for (auto it = begin; it != end; ++it) {
    it->second = it->second * mult;
}
```

- Кажется тут нет способа сделать вызов `std::for_each`?

```
std::for_each(std::execution::par, begin, end, [](auto &&elt){
    elt.second = elt.second * mult; // ошибка
});
```

Идея функтора

- Допустим мы написали класс с состоянием и оператором вызова.

```
struct Multer {  
    int m;  
    Multer(mult) : m(mult) {}  
    void operator()(auto &&elt) { elt.second = elt.second * m; }  
}
```

- Тогда это становится возможным.

```
std::for_each(std::execution::par, begin, end, Multer{mult});
```

- Но на каждый чих не наздравствуешься. Неудобно каждый раз писать функтор.

Hello, lambda world

- Давайте посмотрим на лямбды в их блеске и славе.

```
int main(int argc, const char **argv) {  
    return [argv] () -> int {  
        std::cout << "Hello from " << argv[0] << std::endl;  
        return 0;  
    } ();  
}
```

- Мы можем это упростить, выкинув пустое тело и выводимый тип.

```
auto hi = [argv] { std::cout << argv[0] << "\n"; return 0; };  
hi();
```

([] () { }) () ;

is now legal C++

Обсуждение

- λ -выражения это не функции

```
auto t = [z](auto x, auto y) { return x < y * z; };
```

- Это скорее классы с перегруженным operator()

```
struct __closure_type_for_t {  
    int __k;  
    auto operator()(auto x, auto y) const {  
        return x < y * __k;  
    }  
} t{z};
```

Убираем const

- Если мы хотим изменять захваченный по значению контекст мы должны сделать нашу лямбду в явном виде mutable

```
auto t = [z](auto x, auto y) mutable { z += 1; return x < y * z; };
```

- Обратите внимание, что z изменяется в пределах замыкания.

```
auto s = t;
```

```
t(1, 2); // z изменилось внутри t, но не внутри s
```

- Замыкания по умолчанию копируемые (если не захвачено ничего со стёртым ctor).
- Глобальные и статические переменные захватывать не надо, они доступны и так.

Виды захвата

- Захват по значению (по ссылке)

```
auto fval = [a, b](int x) { return a + b * x; };  
auto fvalm = [a, b](int x) mutable { a += b * x; return a; };  
auto fref = [&a, &b](int x) { a += b * x; return a; };
```

- Захват по ссылке всегда mutable и отслеживает состояние переменной.

```
a = 42;
```

```
fval(x); // тот же
```

```
fref(x); // использует новое a
```

- Разумеется можно смешивать: [&a, b, &c, d]

Виды захвата

- Захват всего контекста по значению (по ссылке)

```
auto faval = [=](int x) { return a + b*x; };  
auto faref = [&](int x) { a += b*x; return a; };
```

- Захват всего по значению и частично по ссылке и наоборот

```
auto favalb = [=, &b](int x) { return a + b*x; };  
auto farefa = [&, a](int x) { b += a*x; return b; };
```

- Захват с переименованием

```
auto freval = [la = a](int x) { return la + x; };  
auto freref = [&la = a](int x) { la += x; return la; };
```

Виды захвата

- Переименование позволяет захватить с перемещением.

```
std::ostream_iterator<int> os{std::cout, " "};  
std::vector v = {1, 2, 3};
```

```
auto out = [w = std::move(v), os] {  
    std::copy(w.begin(), w.end(), os);  
};
```

- Теперь вектор передан в состояние замыкания.
- Передача осуществляется в конструкторе замыкания, т.е. в момент создания функции, а не в момент её вызова.

Захват в теле класса

```
struct Foo {  
    int x;  
    void func() {  
        [x] mutable { x += 3; } (); // FAIL  
        [&x] { x += 3; } (); // FAIL  
  
        [=] { x_ += 3; } (); // OK  
        [&] { x_ += 3; } (); // OK  
  
        [this] { x_ += 3; } (); // OK  
    }  
};
```

- Это работает, поскольку полный захват захватывает `this`

Задача: локальный контекст

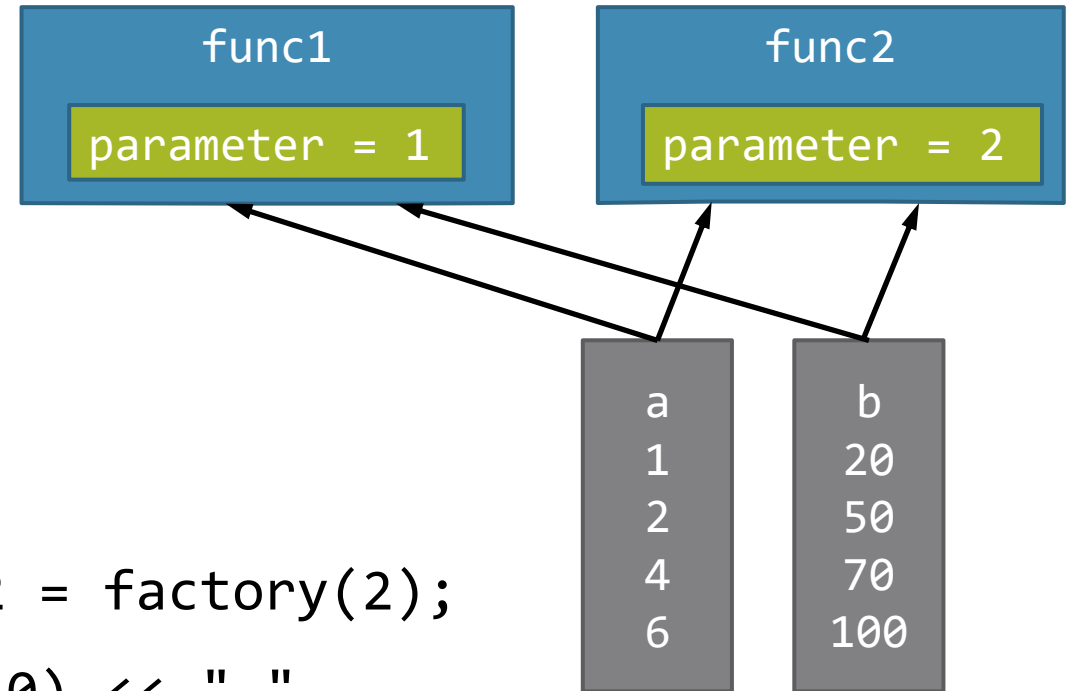
```
auto factory (int parameter) {  
    static int a = 0;  
    return [=] (int argument) {  
        static int b = 0;  
        a += parameter; b += argument;  
        return a + b;  
    }  
}
```

```
auto func1 = factory(1); auto func2 = factory(2);  
cout << func1(20) << " " << func1(30) << " "  
      << func2(20) << " " << func2(30) << endl;
```

Решение: локальный контекст

```
auto factory (int parameter) {  
    static int a = 0;  
    return [=] (int argument) {  
        static int b = 0;  
        a += parameter; b += argument;  
        return a + b;  
    };  
}
```

```
auto func1 = factory(1); auto func2 = factory(2);  
cout << func1(20) << " " << func1(30) << " "  
      << func2(20) << " " << func2(30) << endl;
```



Единая типизация замыканий

```
auto t = [&x, &y] { return x + y; };
```

- Тут не вполне ясно что такое auto.
- Мало того, так нельзя делать в теле класса.

```
struct Foo {  
    auto t = [&x, &y] { return x + y; }; // ошибка  
};
```

- Мы понимаем, что это closure type но не можем записать его явно.
- В этом случае мы можем частично стереть тип.

Единая типизация замыканий

```
auto t = [&x, &y] { return x + y; }; // closure
```

```
function<int()> f = [&x, &y] { return x + y; }; // function
```

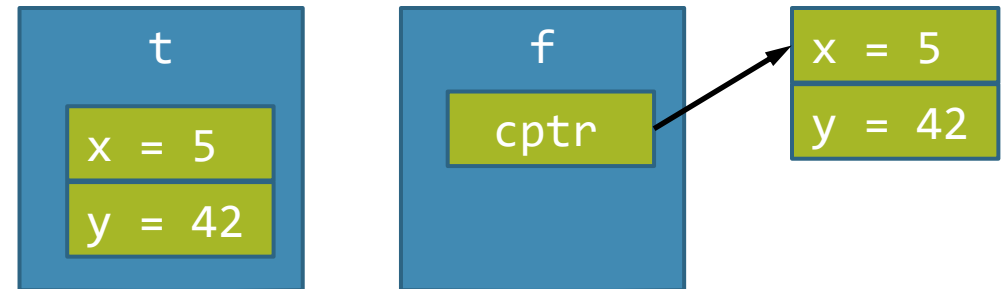
- `std::function<сигнатура>` это единый тип к которому **приводятся** все замыкания с данной сигнатурой.

- Ключевое слово "приводятся".

```
t = [] { return 42; } // FAIL
```

```
f = [] { return 42; } // ok
```

- Тип функции теряет информацию о захвате контекста. Значение имеет только сигнатура.



Снова захват в теле класса

```
using VVTy = std::function<void(void)>;  
struct Foo {  
    int x;  
  
    VVTy xplus1 = [&] { x += 3; }; // OK, but hmmm....  
    VVTy xplus2 = [this] { x += 3; }; // OK
```

- Это вдвойне интересно, так как такие лямбды-члены ведут себя как методы.

```
struct Foo f;  
f.xplus1(); // OK
```

Информация о конкретном типе

- Механизм `std::function` унифицирует типы замыканий
- Информация о реальном типе возвращается через `target_type`

```
int f(int);
```

```
function<int(int)> fn1 = f,  
                  fn2 = [](int a) {return -a;},  
                  fn3 = [x](int a) {return x - a;};  
cout << fn1.target_type().name() << endl  
      << fn2.target_type().name() << endl  
      << fn3.target_type().name() << endl;
```

Case study: finally

- Допустим у нас есть очень простой класс:

```
struct Finally {  
    std::function<void()> action_;  
    explicit Finally(std::function<void()> action):  
        action_(std::move(action)){}  
    ~Finally() { action_(); }  
};
```

- Теперь мы можем вот такие фокусы

```
FILE *f = fopen("myfile.dat", "r"); assert(f);  
Finally close_f([&f]{ fclose(f); });
```

Обсуждение: heap indirection

- Есть некоторая проблема с таким подходом к finally, а именно производительность. Гораздо эффективней иметь closure.

```
template <typename ActTy> inline auto Finally(ActTy fn) {  
    struct Finally_impl {  
        ActTy act;  
        explicit Finally_impl(ActTy action): act(std::move(action)){}  
        ~Finally() { act(); }  
    };  
    return Finally_impl(std::move(fn)); // bingo  
}
```

- Функция нужна для вывода типов (можно ли сделать deduction hint?)

Предостережение

- У вас может возникнуть соблазн сделать функтор с изменяемым состоянием.

```
auto [begin, end] = m.equal_range(i);
```

```
int max = 0;
```

```
std::for_each(begin, end, [&max](auto &&elt) {  
    int n = f(elt.second);  
    if (n > max) max = n;  
});
```

```
std::cout << "Answer is: " << max << std::endl;
```

- Это не ошибка, но это дурной тон. Чем опасен такой подход?

- ❑ Функторы и состояние

- Алгоритмы

- ❑ No raw loops

- ❑ Case study: группы перестановок

Алгоритмы

- Алгоритм стандартной библиотеки это **функция**, выполняющая действие над **интервалами**, заданными с помощью итераторов.

```
template<class InputIt, class OutputIt>  
OutputIt copy(InputIt fst, InputIt last, OutputIt res);
```

- Имя алгоритма может иметь суффиксы и префиксы
 - if (например copy_if) – выполнить действие при выполнении предиката
 - n (например copy_n) – выполнить действие ограниченное количество раз
 - copy (например reverse_copy) – разместить результат в новом контейнере
 - stable (например stable_partition) – алгоритм работает стабильно
- Вопрос должен ли copy_n существовать в языке – дискуссионный.

Несколько примеров copy_x

```
int arr[] = {2, 3, 5, 7, 11, 13, 17};  
std::vector<int> v(7);  
std::copy_n(arr, 7, v.begin());  
std::copy(v.begin(), v.end(),  
          std::ostream_iterator<int>(std::cout, "\n"));  
std::fill(v.begin(), v.end(), 0);  
std::copy_if(arr, arr + 7, v.begin(),  
             [](int i){ return (i % 3) == 1; });
```

- Контрольный вопрос: что на экране, что в векторе?

Обсуждение: специализации циклов

- Есть довольно серьёзное заблуждение, что все алгоритмы это `for_each`
- Общее правило: более специализированный алгоритм лучше менее специализированного.
- Алгоритм в этом смысле можно рассматривать как **специализацию цикла**.
 - Обычный `for` это что угодно, компилятор и библиотека ничего не знают и оптимизируют как угодно.
 - Но тот же сорту это именно копирование и ничто иное.
- Важнейший навык – выбирать правильные алгоритмы.
- Для этого надо видеть **паттерны в коде**.



Паттерны в коде: find_if

- Классический паттерн: for, внутри if, внутри break

```
for (auto&& elt : v)
    if (check(elt)) {
        action(elt);
        break;
    }
```

- Разумеется это std::find_if

```
auto it = std::find_if(v.begin(), v.end(),
                      [](auto &&elt) { return check(elt); });
if (it != v.end()) action(elt);
```

Иногда и `find_if` много

- Допустим вам принесли код, который использует `find_if`

```
if (std::find_if(v.begin(), v.end(), p) != v.end())  
    action();
```

- Его можно и даже нужно упростить далее

```
if (std::any_of(v.begin(), v.end(), p)) // есть ли элемент?  
    action();
```

- Аналогично `std::all_of` и `std::none_of`
- Мы всегда должны стремиться к выбору самого специализированного алгоритма из существующих.

Обсуждение: и это частая проблема

- На самом деле это частая ситуация. Если вы видите как кто-то пишет:

```
auto [ita, itb] =  
    std::mismatch(a.begin(), a.end(), b.begin());  
  
if (ita != a.end() && itb != b.end())  
    action();
```

- То такому человеку надо вежливо объяснить, что он просто забыл про то, что существует более специализированный способ.

```
if (!std::equal(a.begin(), a.end(), b.begin()))  
    action();
```

Паттерны в коде: копирование

- Допустим мы хотим скопировать диапазон $[1, 6)$ начиная с позиции перед элементом 4.
- Все ли понимают почему обычный `std::copy` не подходит?
- Как бы вы выкрутились?



- В примере по ссылке обратите внимание что мы указываем конец. Как вы думаете что будет если мы укажем начало?

Паттерны в коде: transform

- То, что в функциональных языках называется map, в C++ это transform

```
vector<int> v = {2, 3, 5, 7, 11, 13};
```

- Неправильно: `for_each` тут пытается что-то делать над текущим элементом.

```
std::for_each(v.begin(), v.end(), [](auto& i) { i = -i; } );
```

- Обратите внимание: при использовании правильного алгоритма легко заменить выходной итератор.

```
auto negate = [](auto&& i) { return -i; }  
std::transform(v.begin(), v.end(), v.begin(), negate);
```



Обсуждение: функция как предикат

- Предположим хочется использовать функцию `std::toupper`. Как написать вызов алгоритма с этим предикатом?

```
std::string s = "hello";
```

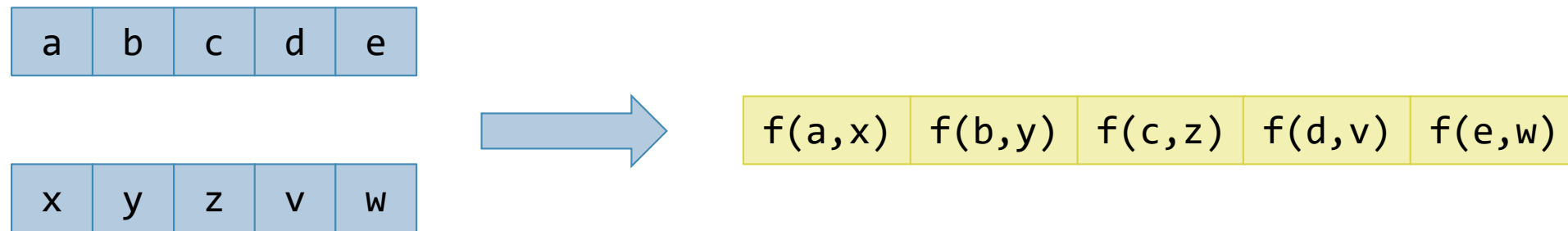
```
// увы, это не будет работать
```

```
std::transform(s.begin(), s.end(), s.begin(), std::toupper);
```

- Причина банальная: `std::toupper` это множество перегрузки. Компилятор не понимает какую из перегруженных функций взять.
- Как бы вы выкрутились?

Второй смысл transform

- У этого алгоритма есть форма, где он работает как своего рода zip.



- Дополнение к нему `std::transform_reduce` позволяет аккумуляровать бинарную операцию над ними.
- Интересно, что то же самое делает `std::inner_product`, но делает это **иначе**.

Отличия в гарантиях

- Есть такие пары функций, которые существуют с разными гарантиями на исполнение.
- Например `reduce` и `accumulate`
 - Обе берут бинарную операцию и считают результат, но
 - `accumulate` гарантированно сделает это in-order
 - `reduce` сделает это как угодно
- Ещё пример это `inclusive_scan` и `partial_sum`
- Также `transform_reduce` и `inner_product`
- Мы должны внимательно читать документацию чтобы осознавать такие вещи

Задача: кстати о чтении документации

- Как вы думаете, что такое `all_of`, `any_of` и `none_of` на пустых диапазонах?

```
std::vector<int> v; // пустой вектор
```

```
if (std::all_of(v.begin(), v.end(), p)) // ?  
    action_all();
```

```
if (std::any_of(v.begin(), v.end(), p)) // ?  
    action_any();
```

```
if (std::none_of(v.begin(), v.end(), p)) // ?  
    action_none();
```

Паттерны в коде: remove

- Как бы вы написали функцию remove?
- Идея функции: удалить из диапазона все значения val

```
template <typename Iter, typename T>  
Iter remove (Iter first, Iter last, const T& val) {  
    // что здесь?  
}
```

Некоторая засада с remove

- Как бы вы написали функцию remove?
- Идея функции: удалить из диапазона все значения val

```
template <typename Iter, typename T>  
Iter remove (Iter first, Iter last, const T& val) {  
    // что здесь?  
}
```

- Правильный ответ: **никак**. По итератору нечто можно удалить из контейнера только используя `Cont.erase(it)`

Идиома erase-remove

- Как же по настоящему работает remove?

```
std::vector<int> v = {1, 42, 2, 42, 3, 42, 4};
```

```
auto result = std::remove(v.begin(), v.end(), 42);  
v.erase(result, v.end());
```

- Или, группируя это в одну фразу:

```
v.erase(std::remove(v.begin(), v.end(), 42), v.end());
```

- Эта техника называется "идиома erase-remove"



Обсуждение: не только remove

- Среди алгоритмов не только remove "удаляет" элементы
- Например unique

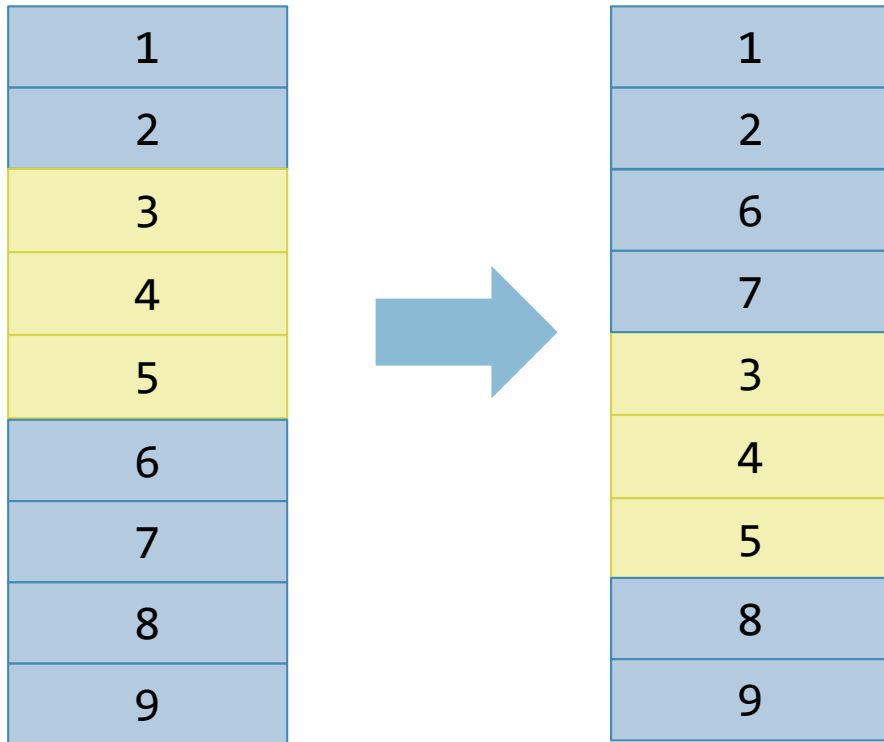
```
vector v = {1, 1, 2, 2, 3, 3, 4};
```

```
std::sort(v.begin(), v.end());
```

```
v.erase(std::unique(v.begin(), v.end()), v.end());
```

- Это тоже идиома erase-remove только без remove
- К счастью пока что в стандарте C++ только три таких алгоритма: remove, remove_if и unique. Но в пользовательском коде может попасться всякое.

Паттерны в коде: перемещение группы



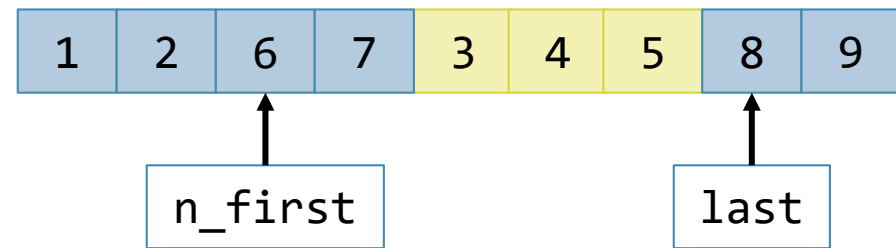
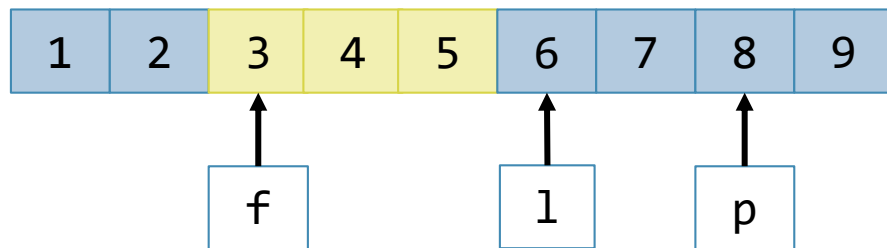
- Три параметра:
 - начало группы **f**, ***f == 3**
 - конец группы **l**, ***l == 6**
 - позиция **p** куда группа должна быть перемещена, ***p == 8**
- Как бы вы написали такое перемещение?

Внезапно rotate

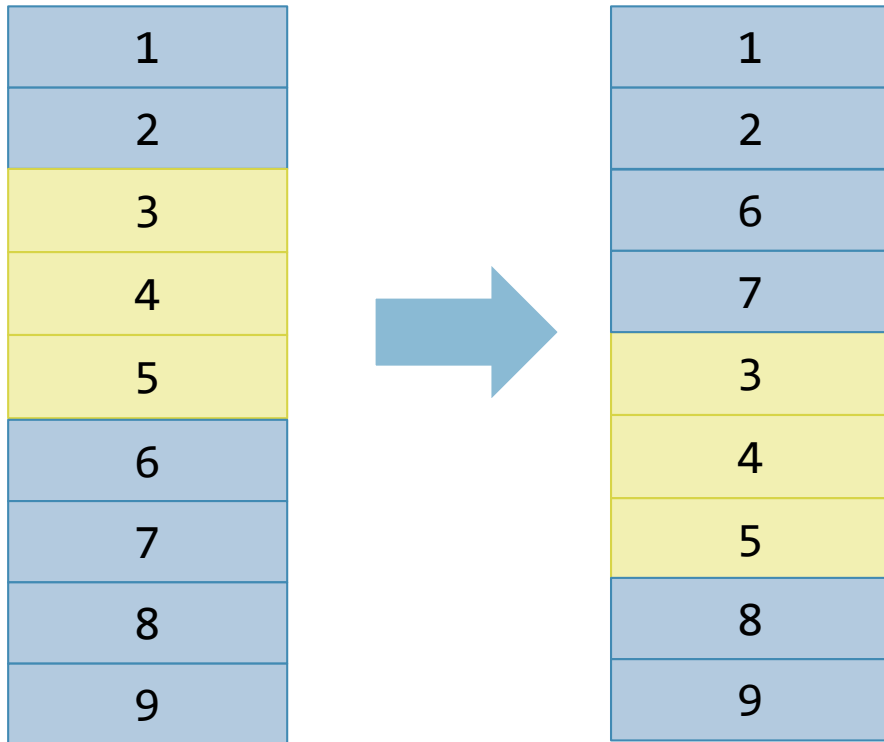
- rotate работает следующим образом:

```
void rotate(FwIt first, FwIt n_first, FwIt last );
```

- Диапазон от `first` до `last` проворачивается так, чтобы первым элементом стал `n_first`
- Ментальная модель `rotate(f, l, p)` это перенос группы `[f, l)` в позицию перед `p`.



Групповое перемещение это rotate



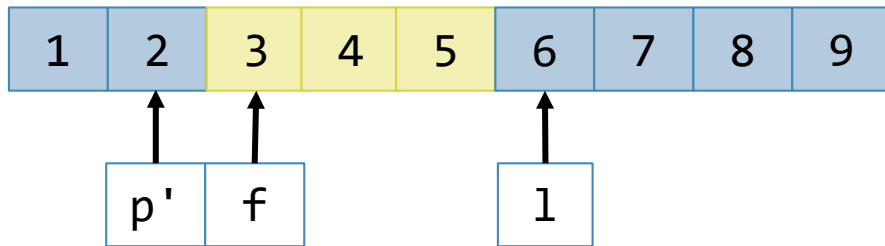
- Три параметра:
 - начало группы **f**, ***f == 3**
 - конец группы **l**, ***l == 6**
 - позиция **p** куда группа должна быть перемещена, ***p == 8**
- `rotate(f, l, p)` это перенос группы `[f, l)` в позицию перед `p`.
- Итак, вы бы написали `rotate`?

Небольшая проблема

- rotate работает следующим образом:

```
void rotate(FwIt first, FwIt n_first, FwIt last );
```

- Но что, если позиция куда нужно переместить лежит выше f?
- Тогда `rotate(f, 1, p')` не будет работать, так как `[f, p')` не образует диапазон

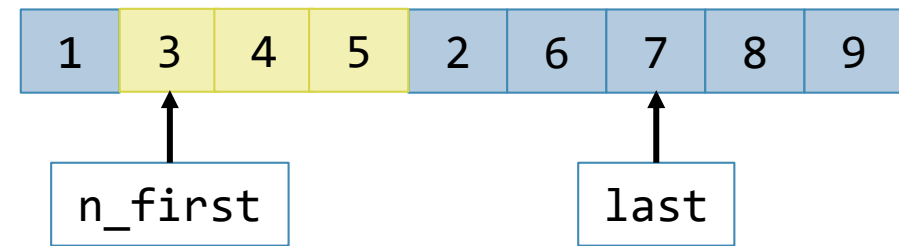
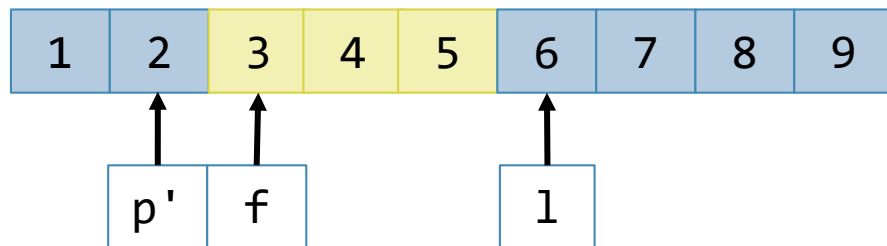


Решение

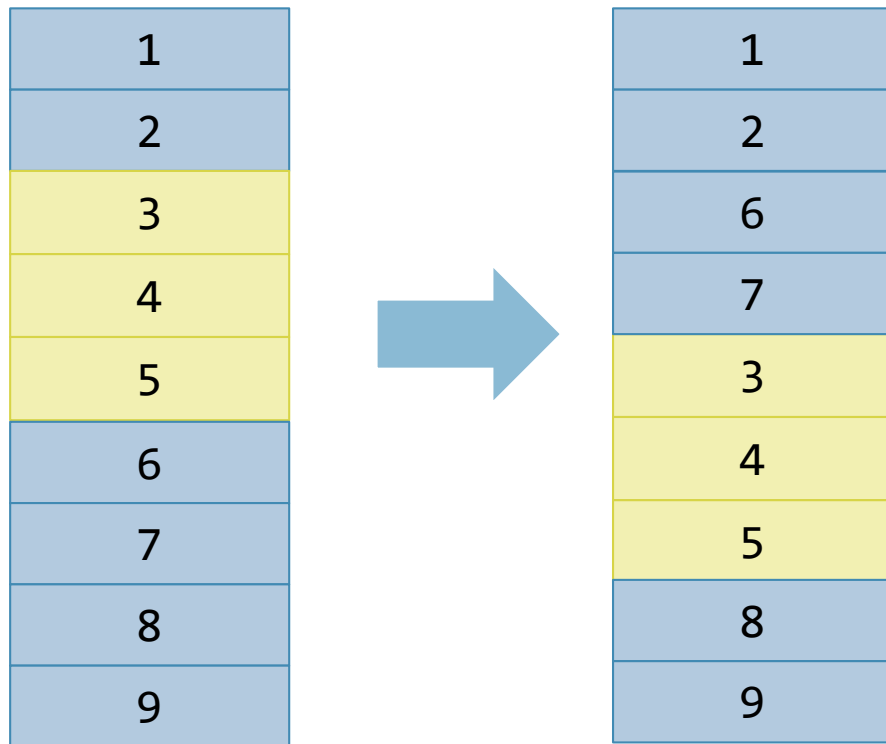
- rotate работает следующим образом:

```
void rotate(FwIt first, FwIt n_first, FwIt last );
```

- Но что, если позиция куда нужно переместить лежит выше f?
- Тогда `rotate(f, 1, p')` не будет работать, так как `[f, p')` не образует диапазон
- Зато будет работать `rotate(p', f, 1)`



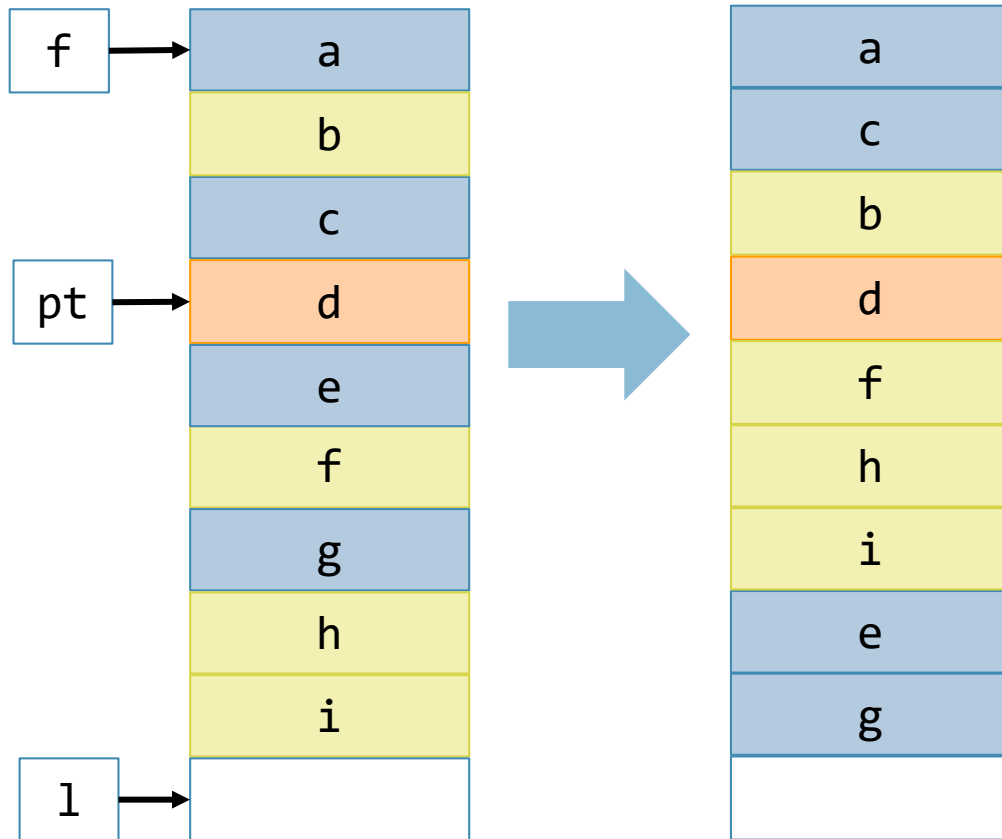
Групповое перемещение элементов



- Три параметра:
 - начало группы **f**, где угодно
 - конец группы **l**, где угодно
 - позиция **p** куда группа должна быть перемещена, где угодно
- Итого мы хотим сделать так

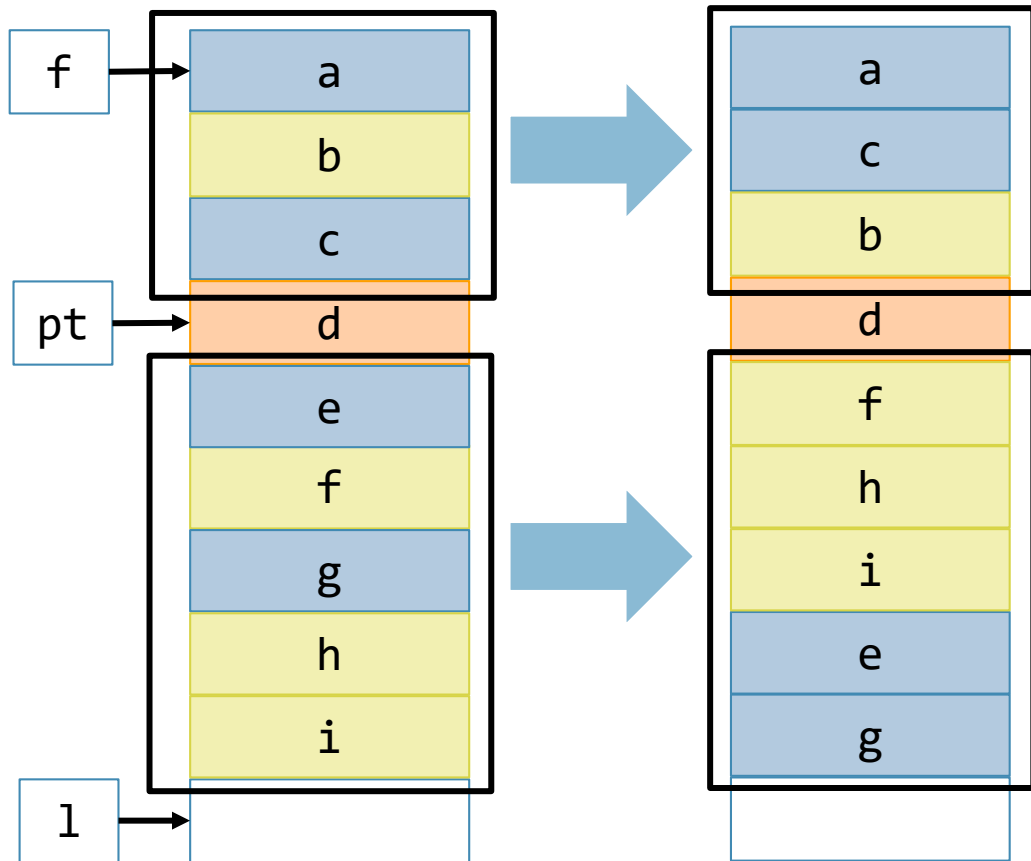
```
if (p < f) rotate(p, f, l);  
else if (l < p) rotate(f, l, p);
```
- Обсуждение: что мы хотим сделать при неправильной позиции?

Паттерны в коде: gather



- Ментальная модель такая: по щелчку мыши на панели d мы собираем по некоему предикату панели.
- После чего не меняя их относительного порядка собираем их около d.
- Как бы вы написали этот алгоритм?
`gather(It f, It l, It pt, F pred);`
- Подсказка: здесь есть очень красивое и почти очевидное решение.

Внезапно stable_partition



```
gather(f, l, pt, p) ::=  
    stable_partition(f, pt, not(p));  
    stable_partition(pt, l, p);
```

- Это просто открывающая глаза штука.
- Было замечено Шоном Парентом (Principal Scientist в Adobe) на ревью у Маршала Клоу (libc++ maintainer в Qualcomm).
- К вопросу о пользе code review даже у людей такого уровня.

Общий обзор

- **Не модифицирующие**
 - all_of, any_of, none_of
 - for_each (n)
 - find (if), count (if)
 - search, find_end, find_first_of
 - mismatch
 - adjacent_find
 - min (element), max (element)
 - clamp
 - equal, lexicographical_compare
- **Слияния и кучи**
 - merge, inplace_merge
 - is_heap (until)
 - make_heap, push_heap, etc....
 - set_union, set_intersection, etc....
- **Модифицирующие**
 - copy (if | n | backward)
 - move (backward)
 - swap, swap_ranges
 - iter_swap
 - transform
 - replace (copy)(if)
 - fill (n), generate (n)
 - reverse
 - rotate (copy)
 - shuffle, sample
 - shift_left (shift_right)
- **Структурные**
 - remove (copy)(if)
 - unique (copy)
- **Сортировка и поиск**
 - partition, stable_partition, partition_point
 - sort, partial_sort, stable_sort
 - nth_element
 - lower_bound, upper_bound, equal_range
 - binary_search
- **Численные**
 - accumulate, reduce, transform_reduce
 - iota
 - adjacent_difference
 - partial_sum, inclusive_scan, exclusive_scan
 - transform_(inclusive | exclusive)_scan
 - inner_product
 - is_permutation
 - next_permutation, prev_permutation

- ❑ Функторы и состояние

- ❑ Алгоритмы

- No raw loops

- ❑ Case study: группы перестановок

Программа Шона Парента

- Если вы видите необходимость написать в вашей программе цикл, посмотрите есть ли возможность использовать стандартный алгоритм.
- Если нет стандартного, придумайте свой, предложите в стандартную библиотеку, опубликуйте статью, прославьтесь.
- В пределах программы Шона Парента, лучшее, что вы можете сделать для кода это **убрать вспомогательные циклы**.
- Вспомогательный цикл внутри функции это любой цикл, который делает нечто, не полностью совпадающее с основным предназначением функции.

Пример вспомогательных циклов

- Следующий фрагмент кода это часть реального кода Chrome OS

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel, int fixed_index) {  
    // check if panel has moved to the other side or another panel  
    const int center_x = fixed_panel->cur_panel_center();  
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {  
        Panel *panel = expanded_panels_[i].get();  
        if (center_x <= panel->cur_panel_center() || i == expanded_panels_.size() - 1) {  
            if (panel != fixed_panel) {  
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];  
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);  
                if (i < expanded_panels_.size())  
                    expanded_panels_.insert(expanded_panels_.begin() + i, ref);  
            }  
            else  
                expanded_panels_.push_back(ref);  
        }  
        break;  
    }  
    // .... далее ещё много кода в этой функции ....  
}
```

Пример вспомогательных циклов

- Он содержит **бессмысленный комментарий**, **странные проверки** и он **похоже квадратичный**.

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel, int fixed_index) {  
    // check if panel has moved to the other side or another panel  
    const int center_x = fixed_panel->cur_panel_center();  
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {  
        Panel *panel = expanded_panels_[i].get();  
        if (center_x <= panel->cur_panel_center() || i == expanded_panels_.size() - 1) {  
            if (panel != fixed_panel) {  
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];  
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);  
                if (i < expanded_panels_.size())  
                    expanded_panels_.insert(expanded_panels_.begin() + i, ref);  
            }  
            else  
                expanded_panels_.push_back(ref);  
        }  
        break;  
    }  
    // .... далее ещё много кода в этой функции ....  
}
```

Пример вспомогательных циклов

- Первый шаг упрощения: push_back не лучше чем insert в конец контейнера

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel, int fixed_index) {  
    // check if panel has moved to the other side or another panel  
    const int center_x = fixed_panel->cur_panel_center();  
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {  
        Panel *panel = expanded_panels_[i].get();  
        if (center_x <= panel->cur_panel_center() || i == expanded_panels_.size() - 1) {  
            if (panel != fixed_panel) {  
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];  
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);  
                expanded_panels_.insert(expanded_panels_.begin() + i, ref);  
            }  
            break;  
        }  
    }  
}  
// .... далее ещё много кода в этой функции ....
```

Пример вспомогательных циклов

- Второй шаг упрощения: условие которое выполняется единожды можно вынести вниз из цикла

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel, int fixed_index) {  
    const int center_x = fixed_panel->cur_panel_center();  
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {  
        Panel *panel = expanded_panels_[i].get();  
        if (center_x <= panel->cur_panel_center() || i == expanded_panels_.size() - 1)  
            break;  
    }  
}
```

```
// Fix this code: panel is panel found above
```

```
if (panel != fixed_panel) {  
    ref_ptr<Panel> ref = expanded_panels_[fixed_index];  
    expanded_panels_.erase(expanded_panels_.begin() + fixed_index);  
    expanded_panels_.insert(expanded_panels_.begin() + i, ref);  
}
```

.... далее ещё много кода в этой функции

Пример вспомогательных циклов

- Третий шаг: убираем бессмысленную проверку

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel, int fixed_index) {  
    const int center_x = fixed_panel->cur_panel_center();  
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {  
        Panel *panel = expanded_panels_[i].get();  
        if (center_x <= panel->cur_panel_center())  
            break;  
    }
```

```
// Fix this code: panel is panel found above  
if (panel != fixed_panel) {  
    ref_ptr<Panel> ref = expanded_panels_[fixed_index];  
    expanded_panels_.erase(expanded_panels_.begin() + fixed_index);  
    expanded_panels_.insert(expanded_panels_.begin() + i, ref);  
}
```

- Теперь **подсвеченное** это стандартный алгоритм. Кто узнает какой?

Пример вспомогательных циклов

- Разумеется это `find_if` (что делает четвёртый шаг)

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel, int fixed_index) {
    const int center_x = fixed_panel->cur_panel_center();
    auto p = std::find_if(begin(expanded_panels_), end(expanded_panels_),
        [&](const ref_ptr<Panel> &e) { return center_x <= e->cur_panel_center(); });

    // Fix this code: panel is panel found above
    if (panel != fixed_panel) {
        ref_ptr<Panel> ref = expanded_panels_[fixed_index];
        expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
        expanded_panels_.insert(p, ref);
    }
}
```

.... далее ещё много кода в этой функции

- Код уже выглядит гораздо лучше, но мешанина с `erase` и `insert` внизу всё ещё цепляет глаз. Кто-нибудь предложит исправление?

Пример вспомогательных циклов

- Шаг пятый: rotate

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel, int fixed_index) {  
    const int center_x = fixed_panel->cur_panel_center();  
    auto p = std::find_if(begin(expanded_panels_), end(expanded_panels_),  
        [&](const ref_ptr<Panel> &e) {  
            return center_x <= e->cur_panel_center(); }));  
  
    // Fix this code: panel is panel found above  
    if (panel != fixed_panel) {  
        auto f = std::begin(expanded_panels_) + fixed_index;  
        std::rotate(p, f, f + 1);  
    }  
}
```

.... далее ещё много кода в этой функции

- Ещё идеи? Может наконец сделаем код рабочим, заменив panel на *p?

Пример вспомогательных циклов

- Шаг шестой: проверка вообще не нужна, rotate хорошо отрабатывает на пустом диапазоне

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel, int fixed_index) {  
    const int center_x = fixed_panel->cur_panel_center();  
    auto p = std::find_if(begin(expanded_panels_), end(expanded_panels_),  
        [&](const ref_ptr<Panel> &e) {  
            return center_x <= e->cur_panel_center();  
        });  
    auto f = begin(expanded_panels_) + fixed_index;  
    std::rotate(p, f, f + 1);  
}
```

.... далее ещё много кода в этой функции

- Это невероятно лучше, чем то, что было. И это эффективнее.
- Но это может стать ещё эффективнее, если мы уверены, что исходные панели отсортированы по индексу.

Сортировки и недосортировки

- Задача: получить **первые N по величине** элементов контейнера `cont` всё равно в каком порядке. После этого вывести их на экран.
 1. `sort(cont.begin(), cont.end());`
 2. `partial_sort(cont.begin(), cont.begin() + N, cont.end());`
 3. `nth_element(cont.begin(), cont.begin() + N, cont.end());`
- Все три алгоритма решают задачу. Но понятно, что третий вариант требует для этого меньше времени.

Обсуждение

- У нас есть заведомо сортированные контейнеры, такие как `std::set` и есть возможность создать сортированный диапазон на последовательном контейнере.
- Можем ли мы это как-то использовать?

Бинарный поиск

- Неявное ограничение всего семейства таких алгоритмов: они работают только на отсортированных интервалах

```
vector<int> v = {81, 9, 54, 36, 27, 63, 18, 72, 45};
```

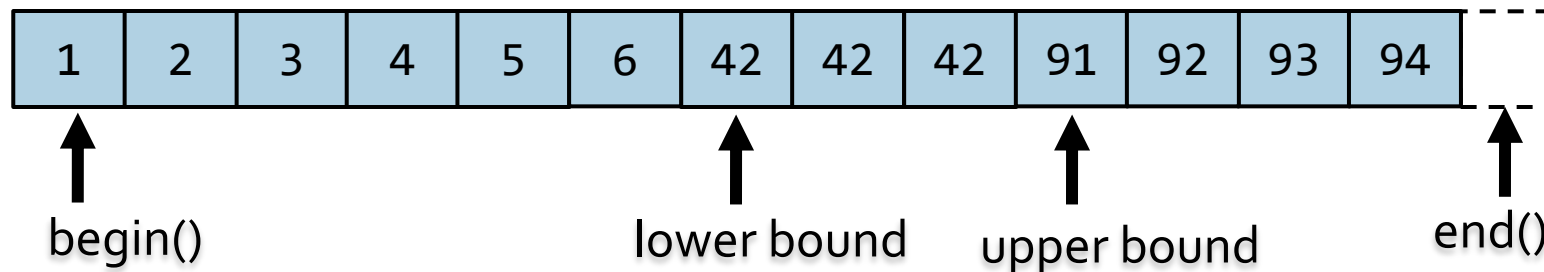
```
sort(v.begin(), v.end());
```

```
if (binary_search(v.begin(), v.end(), 37)) {  
    // сюда мы не попадём?  
}
```

- Надо добавить: мы туда **в лучшем случае** не попадём.

Поисковые алгоритмы

- **binary_search** – есть элемент или его нет
- **lower_bound** – где мог бы быть элемент, если бы он был (слева)
- **upper_bound** – где мог бы быть элемент, если бы он был (справа)
- **equal_range** – есть ли элемент и если да, то где
- Сложность каждого из них логарифмическая.



Вернёмся к панелям

- Вам предлагают использовать изначальную сортированность панелей в следующем коде...

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel, int fixed_index) {
    // check if panel has moved to the other side or another panel
    const int center_x = fixed_panel->cur_panel_center();
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {
        Panel *panel = expanded_panels_[i].get();
        if (center_x <= panel->cur_panel_center() ||
            i == expanded_panels_.size() - 1) {
            if (panel != fixed_panel) {
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
                if (i < expanded_panels_.size()) {
                    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
                } else {
                    expanded_panels_.push_back(ref)
                }
            } break;
        }
    }
} // .... далее ещё много кода в этой функции ....
```


Немного более человечно

- Ну ладно, допустим вот в этом коде (оптимистичней, правда?)

```
const int center_x = fixed_panel->cur_panel_center();  
auto p = find_if(begin(expanded_panels_), end(expanded_panels_),  
                [&](auto *e) {  
                    return center_x <= e->cur_panel_center();  
                }));
```

```
auto f = begin(expanded_panels_) + fixed_index;  
rotate(p, f, f + 1);
```

- Кажется, если панели изначально сортированы, то find_if делает чересчур много...

Внезапное переобувание

- Ну ладно, допустим вот в этом коде (оптимистичней, правда?)

```
const int center_x = fixed_panel->cur_panel_center();
auto p = lower_bound(begin(expanded_panels_), end(expanded_panels_),
                    center_x, [](auto *e, int x) {
                        return e->cur_panel_center() < x;
                    }));
```

```
auto f = begin(expanded_panels_) + fixed_index;
rotate(p, f, f + 1);
```

- Это очень важное наблюдение: переход к алгоритмам позволяет делать такие изменения "в одну строчку"

Обсуждение

- Алгоритм `find` затрачивает $O(N)$
- Алгоритм `equal_range` затрачивает $O(\log(N))$ но требует отсортированного интервала
- Проверка сортированности интервала выполняется через `is_sorted`, который работает за $O(N)$
- Есть ли способы как-то гарантировать сортированность?

- ❑ Функторы и состояние

- ❑ Алгоритмы

- ❑ No raw loops

- Case study: группы перестановок

Циклические перестановки

- Мы можем кодировать перестановки любых объектов как циклические перестановки.
- Простейший цикл это $(1\ 2)$ означает $1 \rightarrow 2, 2 \rightarrow 1$
- $(2\ 3\ 1)$ означает $2 \rightarrow 3, 3 \rightarrow 1, 1 \rightarrow 2$
- Очевидно, что $(2\ 3\ 1) == (1\ 2\ 3) == (3\ 1\ 2)$

Циклические перестановки

- Мы можем кодировать перестановки любых объектов как циклические перестановки.
- Простейший цикл это $(1\ 2)$ означает $1 \rightarrow 2, 2 \rightarrow 1$
- $(2\ 3\ 1)$ означает $2 \rightarrow 3, 3 \rightarrow 1, 1 \rightarrow 2$
- Очевидно, что $(2\ 3\ 1) == (1\ 2\ 3) == (3\ 1\ 2)$
- **Нормальной формой** называется цикл с минимальным элементом впереди
- Нормальная форма для $(4\ 2\ 1\ 3)$?
- Есть ли разница между $(4\ 2\ 1\ 3)$ и " $(4\ 2),$ а потом $(1\ 3)$ "?

Циклические перестановки

- Мы можем кодировать перестановки любых объектов как циклические перестановки.
- Простейший цикл это $(1\ 2)$ означает $1 \rightarrow 2, 2 \rightarrow 1$
- $(2\ 3\ 1)$ означает $2 \rightarrow 3, 3 \rightarrow 1, 1 \rightarrow 2$
- Очевидно, что $(2\ 3\ 1) == (1\ 2\ 3) == (3\ 1\ 2)$
- **Нормальной формой** называется цикл с минимальным элементом впереди
- Нормальная форма для $(4\ 2\ 1\ 3)$ это $(1\ 3\ 4\ 2)$
- Да, рассмотрим $[1\ 2\ 3\ 4]$. $(4\ 2\ 1\ 3)$ изменяет её в $[3\ 1\ 4\ 2]$, тогда как $(4\ 2)(1\ 3)$ в $[3\ 4\ 1\ 2]$

От перестановок к циклической записи

- Обычная перестановка: пишется в два столбика

1 2 3 4 5 6 7 8 9 – исходный

9 2 3 1 7 6 8 5 4 – результирующий

- Ей соответствует циклическая форма

$(1\ 9\ 4)(3)(5\ 7\ 8)(6)$

- Необходимо написать функцию:

$[9\ 2\ 3\ 1\ 7\ 6\ 8\ 5\ 4] \rightarrow (1\ 9\ 4)(2)(3)(5\ 7\ 8)(6)$

- Для начала: какая сигнатура должна быть у этой функции?

От перестановок к циклической записи

- Предлагаемая сигнатура

```
// creates an array of loops from permutation given by table  
// say: a, g, [d, c, e, g, b, f, a]  
// gives: [(a, d, g), (b, c, e), (f)]
```

```
template <typename T>  
void create_loops(T start, T fin, const vector<T>& table,  
                 vector<PermLoop<T>>& out);
```

- Ваша критика?

Области определения

- Кодирование области определения как (T start, T fin) крайне неуместно
- Вместо этого можно взять класс вроде такого

```
template <typename T, T start_, T fin_> struct Idom {  
    T val_;  
    Idom(T val) : val_(val) {} // range check possible  
    operator T() const { return val_; }  
    using type = T;  
    static const T start = start_;  
    static const T fin = fin_;  
};
```

От перестановок к циклической записи

- Предлагаемая сигнатура

```
// creates an array of loops from permutation given by table  
// say: [d, c, e, g, b, f, a]  
// gives: [(a, d, g), (b, c, e), (f)]
```

```
template <typename T>  
void create_loops(const vector<T>& table, vector<PermLoop<T>>& out);
```

- Теперь параметр это domain, предполагаем, что есть `T::start`, `T::fin`
- Ещё критика?

От перестановок к циклической записи

- Предлагаемая сигнатура

```
// creates an array of loops from permutation given by table  
// say: [d, c, e, g, b, f, a]  
// gives: [(a, d, g), (b, c, e), (f)]
```

```
template <typename RandIt, typename OutIt>  
void create_loops(RandIt tbeg, RandIt tend, OutIt lbeg);
```

- Теперь функция это обобщённый алгоритм

Применение перестановок

- Перестановка может быть применена
- Применим (1 2) к числу 1, получаем 2

```
template <typename T> T PermLoop<T>::apply(T x) const {  
    auto it = std::find(loop_.begin(), loop_.end(), x);  
    if (it == loop_.end()) return x;  
    auto nxt = std::next(it);  
    if (nxt == loop_.end()) return *loop_.begin();  
    return *nxt;  
}
```

- Применим (1 3) к [1 2 3 4 5 6], имеем [3 2 1 4 5 6]
- Какую сигнатуру должен иметь метод PermLoop<T>::apply для таблицы?

Применение перестановок

- Перестановка может быть применена
- Применим (1 2) к числу 1, получаем 2

```
template <typename T> T PermLoop<T>::apply(T x) const;
```

- Применим (1 3) к [1 2 3 4 5 6], имеем [3 2 1 4 5 6]
- Какую сигнатуру должен иметь метод PermLoop<T>::apply для таблицы?

```
template <typename T>  
template <typename RandIt>  
void PermLoop<T>::apply(RandIt tbeg, RandIt tend) const;
```

Перемножение перестановок

- Перестановки можно комбинировать не только когда они независимы, но и когда они содержат общие элементы
- Пример: $(1\ 2)(2\ 3)$
- Это означает $1 \rightarrow 2$, $2 \rightarrow 1$ и далее $2 \rightarrow 3$, $3 \rightarrow 2$
- Следовательно $1 \rightarrow 3$, $2 \rightarrow 1$, $3 \rightarrow 2$
- В циклической записи: $(1\ 2)(2\ 3) = (1\ 3\ 2)$
- Более сложный пример: $(1\ 3\ 2)(1\ 2\ 4)(1\ 4\ 3\ 2) = (1\ 2)(3)(4)$
- Какую сигнатуру должна иметь функция перемножения (упрощения заданного массива) перестановок?

Перемножение перестановок

- Неправильный, но соблазнительный вариант

```
template <typename T>  
void simplify_loops(vector<PermLoop<T>> &input);
```

- Правильный вариант (STL-way)

```
template <typename RandIt, typename OutIt>  
void simplify_loops(RandIt tbeg, RandIt tend, OutIt lbeg);
```


Перемножение перестановок

- Можно немного помедитировать над реализацией

```
template <typename RandIt, typename OutIt>
void simplify_loops(RandIt tbeg, RandIt tend, OutIt lbeg) {
    using T = std::decay_t<decltype(*tbeg)>::value_type;
    std::vector<T> table(T::fin - T::start + 1, T::start);
    std::iota(table.begin(), table.end(), T::start);
    for (auto loopit = std::make_reverse_iterator(tend),
          loopend = std::make_reverse_iterator(tbeg);
         loopit != loopend;
         ++loopit)
        loopit->apply(table.begin(), table.end());
    create_loops(table.begin(), table.end(), lbeg);
}
```

Обсуждение

- Это STL-подобные алгоритмы в максимально далёкой от STL предметной области. Тем не менее видно, как основные концепции упорядочивают и улучшают код

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, Addison-Wesley, 2001
- Sean Parent – C++ Seasoning, GoingNative'2013
- Sean Parent – Better Code: Data Structures, CppCon'2015
- Marshall Clow – STL Algorithms - why you should use them, and how to write your own, CppCon'2016
- Jonathan Boccara – 105 STL Algorithms in Less Than an Hour, CppCon'2018
- Ben Dean – Constructing Generic Algorithms: Principles and Practice - Ben Deane, CppCon'2020
- Sean Parent – Warning: `std::find()` is Broken!, CppCon'2021