Подробнее про категории итераторов

В предыдущем видео мы обсудили категории итераторов довольно поверхностно, приведя лишь примеры категорий уже известных нам итераторов. Это помогает с помощью документации понять, для каких итераторов можно вызывать конкретный алгоритм, но требует довольно бессмысленного запоминания. В этом материале мы обсудим, на чём основано такое деление на категории и как угадывать требования алгоритмов, не обращаясь к документации.

Введение

Как известно, итератор в C++ — это абстракция: не какой-то конкретный тип или набор типов, не какой-то общий базовый класс, а всего лишь **набор требований** к типу: возможность итератор разыменовать (*it), увеличить (++it) и т. д. При этом в разных областях применения итераторов — во всевозможных алгоритмах и методах контейнеров — набор требований к итераторам отличается: какие-то алгоритмы требуют от итераторов большего, чем другие. Так и появляются *категории итераторов:* более детальные наборы требований к соответствующему типу.

Чтобы выделить самые базовые категории итераторов, рассмотрим простейший алгоритм — copy, копирующий значения из переданного диапазона в новое место:

```
1   template<typename InputIt, typename OutputIt>
2   OutputIt copy(InputIt first, InputIt last, OutputIt d_first);
```

Его реализация (представленная в том числе и в документации) настолько проста, что использованию этого алгоритма обычно предпочитают самописный цикл:

```
1   template<typename InputIt, typename OutputIt>
2   OutputIt copy(InputIt first, InputIt last,
3                 OutputIt d_first)
4   {
5       while (first != last) {
6           *d_first++ = *first++;
7       }
8       return d_first;
9   }
```

Запись *first++ эквивалентна *(first++): по сути это разыменование итератора с *последующим* его сдвигом.

Любители чуть менее тривиальных примеров могут с тем же успехом рассмотреть алгоритм copy_if или transform.

Поскольку функция шаблонная, она не указывает конкретный тип своих параметров, вместо этого выдвигая требования к ним через код: например, передать в неё числа не выйдет, поскольку их нельзя разыменовать с помощью *.

Уже на этом примере видны различия в использовании итераторов:

1. first и last типа InputIt — итераторы, задающие исходный диапазон. Используются для *итерирования* по этому диапазону и, соответственно, должны позволять сравнение на неравенство (first != last), разыменование и инкремент (*first++). Итераторы, удовлетворяющие этим требованиям, называются *входными* (input).
2. d_first типа OutputIt — итератор, в который копируются значения. Ему достаточно уметь лишь получать очередное значение с последующим инкрементом: *d_first++ = value. Итераторы, удовлетворяющие этим требованиям, называются *выходными* (output).

Однако видно, что есть итераторы, являющиеся одновременно и входными, и выходными: например, итераторы неконстантного вектора; с их помощью можно и читать значения, и записывать их. Такие итераторы, как правило, можно отнести и к более узким категориям.

С другой стороны, видно, что к любым итераторам можно выдвинуть самые базовые требования:

1. Возможность сделать ++it, имея в виду перемещение итератора к следующему элементу.
2. Возможность сделать *it, имея в виду обращение к значению по итератору (возможно, лишь для записи туда нового значения).
3. Помимо этого допускается объединить разыменование и инкремент в одно выражение с помощью постфиксного инкремента: *it++.
4. Наконец, есть некоторый набор стандратных технических требований к типам, например, возможность копировать итераторы. **Полный список** можно изучить здесь. Однако, несмотря на возможность итератор скопировать, не всегда копию итератора можно безопасно использовать. Подробнее об этом вы узнаете ниже в контексте свойства *многопроходности.*

Входные (input) итераторы

Итак, input-итераторы позволяют проитерироваться по набору значений, не изменяя их. Например, можно вывести эти значения через пробел:

```
for (; begin != end; ++begin) {
  cout << *begin << " ";
}
```

(Этот код, конечно, не скомпилируется, если для *it не определён оператор << вывода в поток: например, для пары или вектора.)

Под эти довольно мягкие требования попадают итераторы всех контейнеров: векторов, множеств, словарей и др., в том числе константных.

Итого, требования к входным итераторам следующие:

- ++it: переход к следующему значению;
- *it: **получение значения** по итератору;
- it->field: обращение к полю значения, эквивалентно (*it).field;
- *(it++): получение текущего значения и переход к следующему;
- it1 != end: сравнение на неравенство, необходимое лишь для определения конца диапазона.

В этот список осознанно не попало довольно важное свойство, которым обладают все итераторы контейнеров, — **многопроходность.** Формально его можно определить так: тип итераторов является многопроходным, если по диапазону [begin, end) итераторов этого типа можно **проитерироваться более одного раза.** Таким образом, по диапазону из многопроходных итераторов можно пройтись несколько раз, а по диапазону из однопроходных — только один раз.

Чуть ниже мы увидим, как неожиданно могут вести себя однопроходные итераторы; а в разделе про forward-итераторы — зачем в алгоритмах может понадобиться многопроходность.

istream_iterator

Категории итераторов не возникали на пустом месте: каждая из них описывает некоторый набор существующих итераторов, не удовлетворяющих более жёстким требованиям. В частности, типичным примером входного итератора является **итератор чтения из потока** — istream_iterator из заголовочного файла <iterator>. Так выглядит пример заполнения вектора всеми строками из входного потока с помощью этого итератора:

```
3   vector<string> strings(start, end);
```

Разыменование этого итератора практически эквивалентно чтению очередного значения из потока (в последнем примере это cin). Поскольку итератор старается занимать минимально необходимый объём памяти, он не сохраняет предыдущие считанные значения и потому является однопроходным. Так, вектор more_strings всегда будет содержать только первое значение: все введённые строки уже сохранены в вектор strings, а сам итератор start помнит лишь значение, сохранённое в нём при инициализации, то есть первую строку из ввода. Попробуйте запустить этот код:

```
1   const istream_iterator<string> start(cin);
2   const istream_iterator<string> end;
3   vector<string> strings(start, end);
4   vector<string> more_strings(start, end);
5   // more_strings состоит из одного элемента — strings[0]
```

Поскольку такие итераторы не привязаны к конкретной ячейке в памяти и получают очередное значение из ввода, нельзя рассчитывать и на «разумное» сравнение итераторов на равенство:

```
1   const istream_iterator<string> start(cin);
2   auto start_copy = start;
3   ++start_copy;
4   cout << (start == start_copy) << endl;
```

Если ввести больше одной строки, каждый из итераторов будет помнить свою строку, но при этом они будут считаться равными. Во-первых, с некоторой точки зрения это логично: при итерировании с помощью любого из них будет получен один и тот же набор значений (если только не пытаться использовать оба итератора одновременно). Во-вторых, для простейших итераторов достаточно иметь возможность корректного сравнения лишь с конечным итератором. В роли конечного istream-итератора всегда выступает созданный с помощью конструктора по умолчанию: обычный итератор становится равным ему, как только данные в потоке иссякли.

Как видно из примера с вектором strings, однопроходность и другие ограничения итераторов не являются критичными: при наличии возможности использовать дополнительную память все значения из данного диапазона можно просто сложить в вектор (или любой другой контейнер).

Функции, работающие с input-итераторами

Несмотря на вышеперечисленные ограничения, для input-итераторов (как и для многих других) существуют функции, готовые с ними работать и не выдвигающие более жёстких требований. Один из самых простых примеров — использованный в предыдущих примерах конструктор вектора от двух итераторов: понятно, что для создания вектора по набору значений достаточно лишь один раз проитерироваться по этому набору. Но как для произвольной функции понять, достаточно ли передаваемым в неё итераторам быть входными или требования к ним более жёсткие?

Первый способ — заглянуть в документацию, как уже было показано в видео. Например, упомянутый конструктор вектора выглядит следующим образом:

```
3         const Allocator& alloc = Allocator());
```

Не обращайте внимания на аллокатор: здесь достаточно заметить, что тип итератора (шаблонный параметр функции) — InputIt.

Второй способ — опираясь на своё представление о реализации этой функции, оценить, что именно требуется от итераторов. Рассмотрим, например, уже известные вам алгоритмы count и count_if, вычисляющие количество элементов в заданном наборе, удовлетворяющих некоторому условию. Понятно, что их можно реализовать предельно просто, с помощью лишь одного прохода по данному диапазону. Эта догадка подтверждается документацией: как заветным InputIt в заголовках функций, так и секцией «Possible implementation», в которой приведена возможная реализация этой функции и ссылки на её настоящую реализацию в стандартных библиотеках.

Иногда построение подобных догадок требует более углублённого знания алгоритмов и структур данных. Так, например, может показаться неочевидным, что функция merge, сливающая два отсортированных диапазона в один, требует на входе лишь входные итераторы. И всё же всегда можно заглянуть в документацию или даже просто попробовать скомпилировать нужный код, а параллельно развивать интуицию и знание алгоритмов. Но имейте в виду: не всегда одна лишь компилируемость функции для некоторых итераторов означает, что она будет корректно работать. Соответствующий пример будет рассмотрен ниже.

Полный список требований к input-итераторам находится здесь.

Выходные (output) итераторы

Напомним, как использовался выходной итератор d_first в реализации функции copy:

```
4   {
5       while (first != last) {
6           *d_first++ = *first++;
7       }
8       return d_first;
9   }
```

Отсюда видно, что единственное требование к выходному итератору — уметь принимать очередное значение с помощью конструкции *d_first++ = value. Понятно, что итераторы неконстантных (изменяемых) контейнеров умеют это без каких-либо доработок: *d_first возвращает ссылку на элемент, с помощью которой можно записать новое значение, а оператор ++ действительно перемещает итератор на следующую ячейку памяти.

Но итераторы хороши своим уровнем абстракции: для того, чтобы складывать куда-то значения, необязательно хранить адрес какой-то ячейки в памяти; достаточно лишь реализовать конструкцию *it++ = value. Это позволяет реализовать уже известные вам back_insert_iterator и insert_iterator (конструируемые с помощью функций back_inserter и inserter), а также ostream_iterator — итератор вывода в поток.

Но как это реализовано и можно ли использовать разыменование и инкремент выходного итератора в отрыве от конструкции *it++ = value? Ответ на это легко найти в документации:

- инкремент итератора не делает *ничего;*
- разыменование итератора возвращает ссылку *на сам итератор;*
- зато реализовано присваивание итератору сохраняемого значения, здесь как раз и сосредоточена основная логика.

Таким образом, для подобных итераторов запись *it++ = value эквивалентна it = value, но при этом первый вариант универсален и работает для всех выходных итераторов, включая, например, итераторы вектора. Не используйте второй вариант: вас не поймут.

Отдельно отметим, что **выходными не являются** итераторы множеств (set<T>), словарей (map<K, V>) и константных контейнеров (const vector<T>). Это объясняется очень легко: для этих итераторов ссылка *it является константной и потому не допускает запись.

В предыдущих видео мы уже рассматривали алгоритмы, возвращающие набор значений с помощью выходных итераторов. Обратите внимание, что эти алгоритмы для возврата одного набора принимают один выходной итератор, а не диапазон; поэтому от выходного итератора не требуется возможность сравнения на неравенство. С другой стороны, передаваемый в функцию выходной итератор должен гарантированно уметь принять все переданные в него значения.

Полный список требований к output-итераторам находится здесь.

Прямые (forward) итераторы

Категория forward-итераторов приближает рассмотренные выше абстракции к итераторам настоящих контейнеров.

Рассмотрим forward_list (односвязный список) — линейный контейнер, который позволяет проитерироваться по элементам в прямом порядке, но не в обратном и не перескакивая через несколько элементов. Объясняется это тем, что каждый элемент односвязного списка хранится в памяти независимо и «знает» лишь, где находится следующий за ним. Понятно, что итераторы этого контейнера являются входными, а в случае его неконстантности ещё и выходными. Но какими дополнительными преимуществами обладают итераторы списка?

1. *Многопроходность:* поскольку за итератором стоят реально хранящиеся в памяти элементы, ничто не мешает проитерироваться по ним несколько раз.
2. *Итератор разыменовывается в ссылку,* а не во что-то ненастоящее, как было с back_insert_iterator. (Впрочем, существуют forward-итераторы, для которых operator* возвращает результат по значению.)
3. *Разумное сравнение:* благодаря тому, что каждый итератор (кроме, возможно, end) указывает на реальное место в памяти, нет проблем в определении корректного сравнения на равенство и неравенство для любых итераторов *одного контейнера.*

Это и есть достаточно полный список требований, отличающих прямой итератор от входного. При этом в случае неконстантности ссылки из п. 2 итератор оказывается ещё и выходным.

Функции, работающие с forward-итераторами

Рассмотренный выше список требований может показаться довольно искусственным: неужели есть алгоритмы, которые принимают forward-итераторы, но не готовы работать с входными? Оказывается, что да, и дело в первую очередь в многопроходности.

В качестве примера можно привести следующие алгоритмы, не изменяющие переданный диапазон: find_first_of, find_end, search, adjacent_find. Сигнатура простейшего варианта функции find_first_of выглядит следующим образом:

```
template<typename InputIt, typename ForwardIt>
InputIt find_first_of(InputIt first, InputIt last,
                      ForwardIt s_first, ForwardIt s_last);
```

В результате возвращается первый из элементов [first, last), равный одному из элементов [s_first, s_last). Понятно, что самый простой способ это реализовать — для каждого элемента первого диапазона сравнить его с каждым элементов второго. И действительно, код этой функции по сути эквивалентен такому:

```
return find_if(
    first, last,
    [s_first, s_last](const auto& value) {
      return find(s_first, s_last, value) != s_last;
    }
);
```

Такой код не требует многопроходности от первого диапазона, но для второго она необходима, так как find для него запускается несколько раз.

Похожая история имеет место для алгоритма search: search(first, last, s_first, s_last) ищёт первое вхождение [s_first, s_last) в [first, last), примеряя один полуинтервал ко второму во всевозможных позициях; соответственно, многопроходность требуется от обоих диапазонов. find_end, в свою очередь, решает очень похожую задачу и с большой вероятностью использует search

adjacent_find

С adjacent_find дело обстоит несколько сложнее:

```
template<typename ForwardIt>
ForwardIt adjacent_find(ForwardIt first, ForwardIt last);
```

Этот алгоритм ищет совпадающие соседние элементы. Неужели это нельзя сделать за один проход и ослабить требования на параметры до input-итераторов? Чтобы разобраться, рассмотрим возможную реализацию, которая приводится в документации:

```cpp
template<typename ForwardIt>
ForwardIt adjacent_find(ForwardIt first, ForwardIt last) {
    if (first == last) {
        return last;
    }
    ForwardIt next = first;
    ++next;
    for (; next != last; ++next, ++first) {
        if (*first == *next) {
            return first;
        }
    }
    return last;
}
```

Видите ли вы, где этот код требует многопроходности?

По сути он дважды итерируется по диапазону: сначала с помощью next, а сразу за ним — first. Найдите тест, на котором эта функция будет некорректно работать для istream-итераторов. Проверьте, работает ли на этом тесте adjacent_find из стандартной библиотеки.

Перейдём на некоторое время в более формальную плоскость и разберёмся, в каких именно требованиях выражается многопроходность. Их можно найти в статье про ForwardIterator, в разделе «Multipass guarantee». Какие из этих свойств действительно необходимы для корректной работы последнего примера?

1. Корректность сравнения на == любых итераторов. В последнем примере **не нужна:** сравнение происходит только с last.
2. Безопасность записи нового значения. **Не актуально,** так как adjacent_find не изменяет элементы.
3. Устойчивость *it к увеличению копии it. **Нужна,** и для istream-итератора выполняется благодаря сохранению текущего элемента.
4. Выполнение ++it1 == ++it2 для равных итераторов it1 и it2. С одной стороны, как уже было отмечено в п. 1, сравнение итераторов в этом коде не очень нужно, с другой — явно **ожидается,** что при увеличении обоих итераторов first и next с отставанием в 1 шаг будет получаться одинаковое значение. Таким образом, в сочетании с п. 1 получаем свойство, которое **нужно и не выполняется** для istream-итераторов.

Можно ли реализовать adjacent_find без одновременного прохода двумя итераторами? Да, и такой код можно увидеть в реализации этого алгоритма в стандартной библиотеке g++ (для краткости приведена версия без использования предиката):

```
template<typename ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last) {
  if (first == last) {
    return last;
  }
  ForwardIterator next = first;
  while (++next != last) {
    if (*first == *next) {
      return first;
    }
    first = next;
  }
  return last;
}
```

Такой код уже **успешно работает с istream-итераторами.** Но тогда почему не ослабить требования на параметры adjacent_find? Неужели istream-итератор в чём-то лучше стандартного input-итератора?

Да, и это свойство неоднократно обсуждалось выше. По стандарту от input-итератора не требуется сохранение текущего значения. Более формально, при увеличении итератора его копии не обязаны оставаться разыменуемыми. Это свойство требуется при разыменовании итератора first, но выполняется для input-итератора как раз благодаря тому, что он помнит текущее значение и «не теряет» его при копировании и сдвиге.

Можно ли реализовать adjacent_find так, чтобы он действительно мог работать с любыми input-итераторами? Да, и для этого достаточно вместо first между соседними итерациями сохранять само предыдущее значение — *first — в отдельную переменную. Правда, так не стоит делать для многопроходных итераторов, поэтому понадобится отдельный вариант функции.

Дополнения

Ещё одно место в документации, где можно легко встретить forward-итераторы — **параллельные версии** алгоритмов, работающих с input-итераторами, например, count:

```
template<typename ExecutionPolicy, typename ForwardIt, typename T>
int count(
    ExecutionPolicy&& policy,
    ForwardIt first, ForwardIt last,
    const T& value
);
```

Параллельность будет подробно обсуждаться в следующем курсе, но суть здесь следующая: подсчёт количества элементов в диапазоне распараллеливается на разные потоки, для этого исходный диапазон разбивается на порции, которые будут независимо этими потоками обрабатываться — так и возникает необходимость в многопроходности.

Наконец, как уже обсуждалось в предыдущем видео, нужно иметь в виду, что некоторые алгоритмы, принимающие forward-итераторы, в отличие от рассмотренных выше, изменяют или переставляют элементы. Простые примеры:

- remove переставляет элементы местами, откладывая лишние;
- replace присваивает некоторым элементам новое значение.

Понятно, что такие алгоритмы нельзя вызвать для константных итераторов. Обычно по описанию функции легко понять, собирается ли она менять переданные элементы.

Итак, мы подробно обсудили отличия forward-итераторов от более широких категорий. В первую очередь, это многопроходность, в которой периодически возникает необходимость. Также forward-итераторы — самые простые, позволяющие одновременно и читать нижележащие значения, и изменять их.

Полный список требований к forward-итераторам находится здесь.

Двунаправленные (bidirectional) итераторы

Как легко понять по названию, с помощью двунаправленных итераторов можно итерироваться как вперёд, так и назад — с помощью --it или it--. Остальные свойства эти итераторы наследуют от прямых: таким образом, **любой bidirectional-итератор является forward-итератором.** Более того, поскольку любой forward-итератор является входным, то и любой двунаправленный им является.

Двунаправленными являются итераторы всех контейнеров, кроме рассмотренного выше и редко используемого forward_list. При этом итераторы множества и словаря являются двунаправленными и не принадлежат более узким категориям, то есть не удовлетворяют более жёстким требованиям, рассмотренным ниже.

Ожидаемо есть и алгоритмы, на эту двунаправленность полагающиеся: как переставляющий элементы reverse, так и просто читающий (и записывающий) элементы в обратном порядке copy_backward. В документации такие итераторы коротко обозначаются BidirIt.

Полный список требований к bidirectional-итераторам находится здесь.

Итераторы произвольного доступа (random access)

Наконец, мы перешли к категории итераторов с максимальным набором требований — из тех, что могут быть необходимы для вызова алгоритма. Итераторы произвольного доступа умеют всё то же, что и итераторы вектора. А именно:

1. Являются двунаправленными.
2. Позволяют сдвинуть итератор на целочисленное количество позиций:

- it + n, n + it, it - n (результат — итератор);
- it += n, it -= n (результат — ссылка на итератор);
- it[n] (эквивалентно *(it + n)).

3. Позволяют вычитать итераторы и сравнивать их: it1 - it2, it1 < it2, it1 <= it2, it1 > it2, it1 >= it2.

Этим требованиям удовлетворяют итераторы вектора и дека (про него вы узнаете в следующем блоке), но не множества и словаря.

Может возникнуть вопрос: почему бы не реализовать операцию it + n для любых итераторов, не являющихся random access, с помощью n-кратного инкремента? Дело в том, что C++ старается не обманывать разработчика и требует эффективной реализации от операций, выглядящих как встроенные. В данном случае прибавление числа к итератору *должно иметь константную сложность*. Для обобщённых операций с итераторами есть следующие функции из заголовочного файла <iterator>:

- next замещает it + 1 или it + n;
- prev замещает it - 1 или it - n;
- advance замещает it += n;
- distance замещает it2 - it1.

Сложность этих функций линейна (по n или результату distance), но для итераторов произвольного доступа константная, потому что в этом случае реализация осуществляется с помощью +, - и +=.

Итератор произвольного доступа не обязан ссылаться на конкретные ячейки в памяти. Например, можно реализовать **считающий итератор,** поочерёдно возвращающий значения целочисленного типа: 1, 2, 3 и т. д. Такой итератор реализован в библиотеке ranges C++20: см. статью про iota_view, секцию std::ranges::iota_view::iterator. В частности, видно, что категория этого итератора зависит от возможностей типа W, по которому выполняется перебор. Подробнее библиотека ranges будет рассмотрена в 4-м курсе специализации.

Функции, работающие с random access итераторами

Напомним, что в силу наследования свойств рассмотренные выше функции, принимающие входные, прямые и двунаправленные итераторы, можно вызвать для итераторов произвольного доступа. Функции, принимающие выходные итераторы, можно вызвать только для неконстантных итераторов.

При этом есть функции, принимающие лишь итераторы произвольного доступа, то есть требующие сравнения итераторов или быстрого перемещения на несколько ячеек. Простой пример такой функции — shuffle: произвольный доступ в ней необходим для выбора случайного элемента из диапазона с помощью быстрого обращения по случайному индексу. Слово «random» неслучайно присутствует и в названии категории итераторов, и в старом названии функции shuffle, однако путать эти его смыслы всё же не стоит: итераторы с *произвольным* доступом позволяют за константную сложность обращаться по *произвольному* индексу, но напрямую со случайностью не связаны.

Таких функций, которые вовсе не скомпилируются для двунаправленных итераторов, немного. Чаще вы встретите ситуацию, которую удобно обсудить на примере алгоритма lower_bound. В реализации этого алгоритма необходимо обращаться к *середине диапазона итераторов*. Если сделать это с помощью +, алгоритм будет работать только для итераторов произвольного доступа — но быстро. Если же использовать advance, как и сделано в секции «Possible implementation», алгоритм будет работать и для forward-итераторов, пусть и медленнее.

Соответственно, в секции «Сложность» для lower_bound гарантируется следующее: *функция выполняет O(log(last - first)) сравнений, однако для не random access итераторов количество инкрементов итераторов будет линейным.*

Полный список требований к итераторам произвольного доступа находится здесь.

Перегрузка функций по категориям итераторов

Выше мы неоднократно заявляли о возможности реализовать некоторую функцию разными способами в зависимости от категории переданных в неё итераторов. До C++20 стандартный способ сделать это — использовать теги итераторов. Рассмотрим простейший пример из документации:

```cpp
template<typename BDIter>
void alg(BDIter it1, BDIter it2, std::bidirectional_iterator_tag) {
  std::cout << "alg() called for bidirectional iterator\n";
}

template<typename RAIter>
void alg(RAIter it1, RAIter it2, std::random_access_iterator_tag) {
  std::cout << "alg() called for random-access iterator\n";
}

template<typename Iter>
void alg(Iter first, Iter last) {
  alg(first, last,
      typename std::iterator_traits<Iter>::iterator_category());
}

int main() {
  std::vector<int> v;
  alg(v.begin(), v.end());
  // выведет «alg() called for random-access iterator»

  std::list<int> l;
  alg(l.begin(), l.end());
  // выведет «alg() called for bidirectional iterator»
}
```

Обсудим эту схему по пунктам:

1. В main вызывается функция alg от двух итераторов: вектора или списка. Есть ровно одна версия шаблона alg с 2 параметрами, так что вызывается именно она.
2. В функции alg с 2 параметрами определяется тег итератора. typename std::iterator_traits<Iter>::iterator_category — это тип для этого тега: std::random_access_iterator_tag для итератора вектора и std::bidirectional_iterator_tag для итератора списка. То же выражение с пустыми круглыми скобками в конце — значение по умолчанию этого типа.
3. Есть две версии шаблонной функции alg с 3 параметрами: первые два из них шаблонные (и потому в текущем виде не помогают выбрать конкретную версию функции), а третий имеет конкретный тип: это std::random_access_iterator_tag или std::bidirectional_iterator_tag. Сам третий параметр не используется, поэтому его название в заголовке функции не указано. Именно тип 3-го параметра помогает компилятору выбрать конкретную версию функции alg в зависимости от категории исходных итераторов.
4. В зависимости от категории итератора выполняется конкретная логика. В данном случае всего лишь выводятся разные сообщения.

Рассмотрим теперь реальный пример такой перегрузки — [реализацию алгоритма equal в стандартной библиотеке libc++](#).

Эта версия equal поэлементно сравнивает два диапазона итераторов [first1, last1) и [first2, last2) с помощью бинарного предиката p:

```
template<typename InputIt1, typename InputIt2, class BinaryPredicate>
bool equal(InputIt1 first1, InputIt1 last1,
           InputIt2 first2, InputIt2 last2,
           BinaryPredicate p);
```

Понятно, что если эти диапазоны имеют разную длину, их даже не надо сравнивать: можно сразу вернуть false. Но быстро вычислить длину можно только для итераторов произвольного доступа, так что хочется реализовать такую оптимизацию именно для них. Это так и сделано: в перегрузке по random_access_iterator_tag вызывается distance и в случае неравенства длин возвращается false; затем вызывается версия функции equal без 4-го аргумента — конца второго диапазона.

```
template <class _BinaryPredicate, class _RandomAccessIterator1, class _RandomAccessIterato
inline _LIBCPP_INLINE_VISIBILITY _LIBCPP_CONSTEXPR_AFTER_CXX17
bool
__equal(_RandomAccessIterator1 __first1, _RandomAccessIterator1 __last1,
        _RandomAccessIterator2 __first2, _RandomAccessIterator2 __last2, _BinaryPredicate
      random_access_iterator_tag, random_access_iterator_tag )
{
    if ( _VSTD::distance(__first1, __last1) != _VSTD::distance(__first2, __last2))
        return false;
    return _VSTD::equal<_RandomAccessIterator1, _RandomAccessIterator2,
                        typename add_lvalue_reference<_BinaryPredicate>::type>
                       (__first1, __last1, __first2, __pred );
}
```

Непрерывные (contiguous) итераторы

Казалось бы, неужели можно придумать более жёсткие требования к итераторам, которые ещё и будут актуальны для каких-либо алгоритмов? Можно. Однако, подобно свойству многопроходности, это новое свойство нельзя проверить на этапе компиляции.

Это свойство — непрерывность хранения в памяти соответствующих итератору данных. Мы пока не изучали указатели, но если они вам уже знакомы, взгляните на более формальное определение: it[n] *эквивалентно* (&*it)[n]. Знание о том, что данные хранятся в памяти непрерывно, в некоторых случаях позволяет работать с ними более эффективно.

Непрерывными являются итераторы векторов и строк, а также сырые указатели и итераторы массива. Итераторы дека, изучаемого далее в этой неделе, являются random access, но не являются непрерывными. Подробнее про массивы, деки и особенности хранения данных в них вы узнаете в разделе «Линейные контейнеры» 3-го курса.

Понятие contiguous-итератора введено в язык в C++17, однако вместе с ним не появился contiguous_iterator_tag, позволяющий реализовать перегрузку из предыдущего раздела. Так пришлось поступить из-за нежелания сломать обратную совместимость: до C++17 было написано большое количество кода, полагающегося на то, что iterator_category для итераторов векторов и строк — это в точности random_access_iterator_tag. Тем не менее, в C++20 появляется более нативная возможность перегрузки функций по категориям итераторов (в том числе и для непрерывных) — **концепты.**

При этом аналог этой перегрузки реализовывался и до последних стандартов — но исключительно для сырых указателей. Например, алгоритм fill можно реализовать эффективнее для непрерывного диапазона элементов-байтов — с помощью функции memset, унаследованной из языка C. В стандартной библиотеке libstdc++ это сделано так:

```
1   // Specialization: for char types we can use memset.
2   template<typename _Tp>
3     inline typename
4     __gnu_cxx::__enable_if<__is_byte<_Tp>::__value, void>::__type
5     __fill_a(_Tp* __first, _Tp* __last, const _Tp& __c)
6     {
7       const _Tp __tmp = __c;
8       if (const size_t __len = __last - __first)
9         __builtin_memset(__first, static_cast<unsigned char>(__tmp), __len);
10    }
```

Мы не будем углябляться в детали этого кода, но в конечном счёте его смысл следующий: для простых типов (эквивалентных unsigned char) и для итераторов, являющихся указателями, вызывается memset. Эта функция не вызовется, если итератор является *обёрткой* над указателем, как это иногда случается с итераторами вектора.

Итого, в полной мере использовать понятие непрерывного итератора можно будет лишь в C++20 с помощью концептов, которые будут затронуты позже в нашей специализации.

Полный список требований к contiguous-итераторам находится здесь.

vector<bool>::iterator

Выше мы утаили неудобную правду про итераторы контейнера vector<bool>: оказывается, они не являются не то что непрерывными, а даже forward!

Дело в том, что vector<bool> устроен так, что каждый элемент занимает в нём не один байт (как одна переменная типа bool), а один бит. С одной стороны, это в 8 раз эффективнее по памяти, с другой — **не существует ссылки на бит.** Что же тогда возвращает *it для таких итераторов?

Результат этого выражения — так называемый *прокси-объект:* он ведёт себя как ссылка, при этом ссылкой не являясь. Он умеет неявно преобразоваться в bool, ему можно присвоить bool, но это не bool&. Таким образом, нарушается формальное требование к forward-итераторам: *it должно возвращать именно ссылку на элемент.

Тем не менее, по правилам хорошего тона итераторы vector<bool> часто поддерживаются наравне с итераторами других векторов, и формальное требование про ссылку оказывается избыточным.

В одной из задач этого блока вам понадобится получить по типу итератора It (или переменной it) тип нижележащего значения. Те, кто знакомы с decltype, могут предложить использовать для этого выражение decltype(*it). Но из-за этих же нестандартных итераторов *it не будет ссылкой на bool, и поэтому лучше использовать typename It::value_type. Самый же универсальный способ, работающий в том числе и для сырых указателей, — typename iterator_traits<It>::value_type.