Итераторы

Проблема

Рассмотрим функцию `Print` для вывода содержимого произвольного контейнера:

```
template <class Container>
void Print(const Container& container) {
  for (size_t i = 0u, size = container.size(); i < size; ++i) {
    std::cout << container[i] << ' ';
  }
}
```

А если контейнер представляет из себя список?

```
// Просто фантазия, таких членов у std::list нет
template <class Container>
void Print(const Container& container) {
  for (auto* node = container.top(); node != nullptr; node = node->next) {
    std::cout << node->value << ' ';
  }
}
```

Проблема

Хочется иметь унифицированный способ обхода контейнеров и обращения к их элементам.

Итераторы

Итератор - объект с интерфейсом указателя, предоставляющий доступ к элементам контейнера и возможность их обхода.

Все контейнеры поддерживают методы `begin` (итератор на начало контейнера) и `end` (итератор на элемент *следующий за последним*).

Они имеют тип `std::vector<int>::iterator`, `std::list<std::string>::iterator` и т.п.

```
template <class Container>
void Print(const Container& container) {
  for (auto it = container.begin(), end = container.end(); it != end; ++it) {
    std::cout << *it << '\n';
  }
}
```

Итераторы

Чтобы абстрагироваться от понятия контейнера, большинство алгоритмов принимают непосредственно итераторы (полуоткрытый интервал) для работы с последовательностями.

```
template <class Iterator>
void Print(Iterator begin, Iterator end) {
  for (; begin != end; ++begin) {
    std::cout << *begin << '\n';
  }
}

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

std::sort(v.begin(), v.end());
Print(v.begin(), v.end());  // [1, 2, 3, 4]
```

Итераторы

С помощью итераторов можно изменять элементы, на которые они указывают (иногда).

```
template <class Iterator>
void ZeroAll(Iterator begin, Iterator end) {
  for (; begin != end; ++begin) {
    *begin = 0;
  }
}

std::vector<int> v{1, 2, 3};
ZeroAll(v.begin(), v.end());  // [0, 0 ,0]
```

А если вектор константный?

Константные итераторы

Константные итераторы позволяют читать значения, на которые указывают, но не позволяют изменять (при разыменовании возвращают константную ссылку).

```
template <class Iterator>
void ZeroAll(Iterator begin, Iterator end) {
  for (; begin != end; ++begin) {
    *begin = 0;  // CE: assignment of read-only location
  }
}

const std::vector<int> cv{1, 2, 3};
ZeroAll(cv.begin(), cv.end());
```

`begin()` и `end()` теперь возвращают `std::vector<T>::const_iterator`.
Для получения константных итераторов у неконстантных контейнеров можно использовать методы `cbegin()` и `cend()`.

Константные итераторы

В чем отличие `std::vector<T>::const_iterator` и `const std::vector<T>::iterator`?

Константные итераторы

В чем отличие `std::vector<T>::const_iterator` и `const std::vector<T>::iterator`?

В том же, в чем отличие `const T*` и `T* const`:

```
std::vector<int>::iterator it = v.begin();
++it; *it = 0;

std::vector<int>::const_iterator it = v.cbegin();
++it; // *it = 0; CE

const std::vector<int>::iterator it = v.begin();
*it = 0; // ++it; CE

const std::vector<int>::const_iterator it = v.cbegin();
// ++it; *it = 0; CE
```

Существует преобразование из `iterator` в `const_iterator`, но не наоборот.

Категории итераторов

Категории итераторов

Любой итератор обязан определять операции `++`, унарный `*`, `->`.

В зависимости от дополнительных поддерживаемых операций итераторы могут принадлежать следующим категориям:

- input iterator (итератор ввода)
- output iterator (итератор вывода)
- forward iterator (прямой итератор)
- bidirectional iterator (двунаправленный итератор)
- random access iterator (итератор произвольного доступа)
- contiguous iterator (непрерывный итератор) (C++20)

Input Iterator

Input iterator (итератор ввода) - итератор, предоставляющий доступ на чтение элементов (с помощью разыменования `*` или операции `->`).

Объекты итератора ввода могут быть проверены на равенство (`==`, `!=`).

Данные итераторы являются однопроходными, то есть можно пройти в одном направлении ровно 1 раз.

```
template <class InputIterator>
void Function(InputIterator begin) {
  auto copy_begin = begin;
  ++begin;       // Ok, copy_begin инвалидируется
  ++copy_begin;  // UB
}
```

Примеры: `std::istream_iterator`

Forward Iterator

Forward Iterator (прямой итератор) - input iterator, с возможностью создания по умолчанию (нулевой итератор) и многократным проходом по последовательности.

```
template <class ForwardIterator>
void Function(ForwardIterator begin, ForwardIterator end) {
  auto copy_begin = begin;
  for (; begin != end; ++begin) {
    std::cout << *begin << ' ';
  }
  for (; copy_begin != end; ++copy_begin) {
    std::cout << *copy_begin << ' ';
  }
}
```

Примеры: `std::forward_list<T>::iterator`, `std::unordered_set<T>::iterator`, `std::unordered_multimap<Key, Value>::iterator`, …

Bidirectional Iterator

Bidirectional iterator (двунаправленный итератор) - forward iterator, для которого дополнительно определены операции `--` (префиксная и постфиксная).

```
template <class BidirectionalIterator>
void Function(BidirectionalIterator begin, BidirectionalIterator end) {
  for (auto it = begin; it != end; ++it) {
    std::cout << *it << ' ';
  }
  for (auto it = end; it != begin;) {
    std::cout << *(--it) << ' ';
  }
}
```

Примеры: `std::list<T>::iterator`, `std::set<T>::iterator`, `std::multimap<Key, Value>::iterator`, ...

Random Access Iterator

Random access iterator (итератор произвольного доступа) - bidirectional iterator, для которого дополнительно определены арифметические операции (сложение с числом `+` `+=`, вычитание числа `-` `-=`, разность итераторов `-`), доступ по индексу `[]`, а также отношение порядка (`<` `>` `<=` `>=`)

```
template <class RndAccessIterator>
void Function(RndAccessIterator begin, RndAccessIterator end) {
  for (auto it = begin; it < end; it += 2) {
    std::cout << *it << ' ';
  }
  size_t size = end - begin;
  for (size_t i = 0; i < size; ++i) {
    std::cout << begin[i] << ' ';
  }
}
```

Примеры: `std::vector<T>::iterator`, `std::array<T, N>::iterator`, `std::deque<T>::iterator`

Contiguous Iterator (C++20)

Contiguous iterator (непрерывный итератор) - random access iterator, для которого выполнено свойство:

```
*(iterator + n) <=> *(&(*iterator) + n)
```

, то есть данные под итератором расположены непрерывно в памяти.

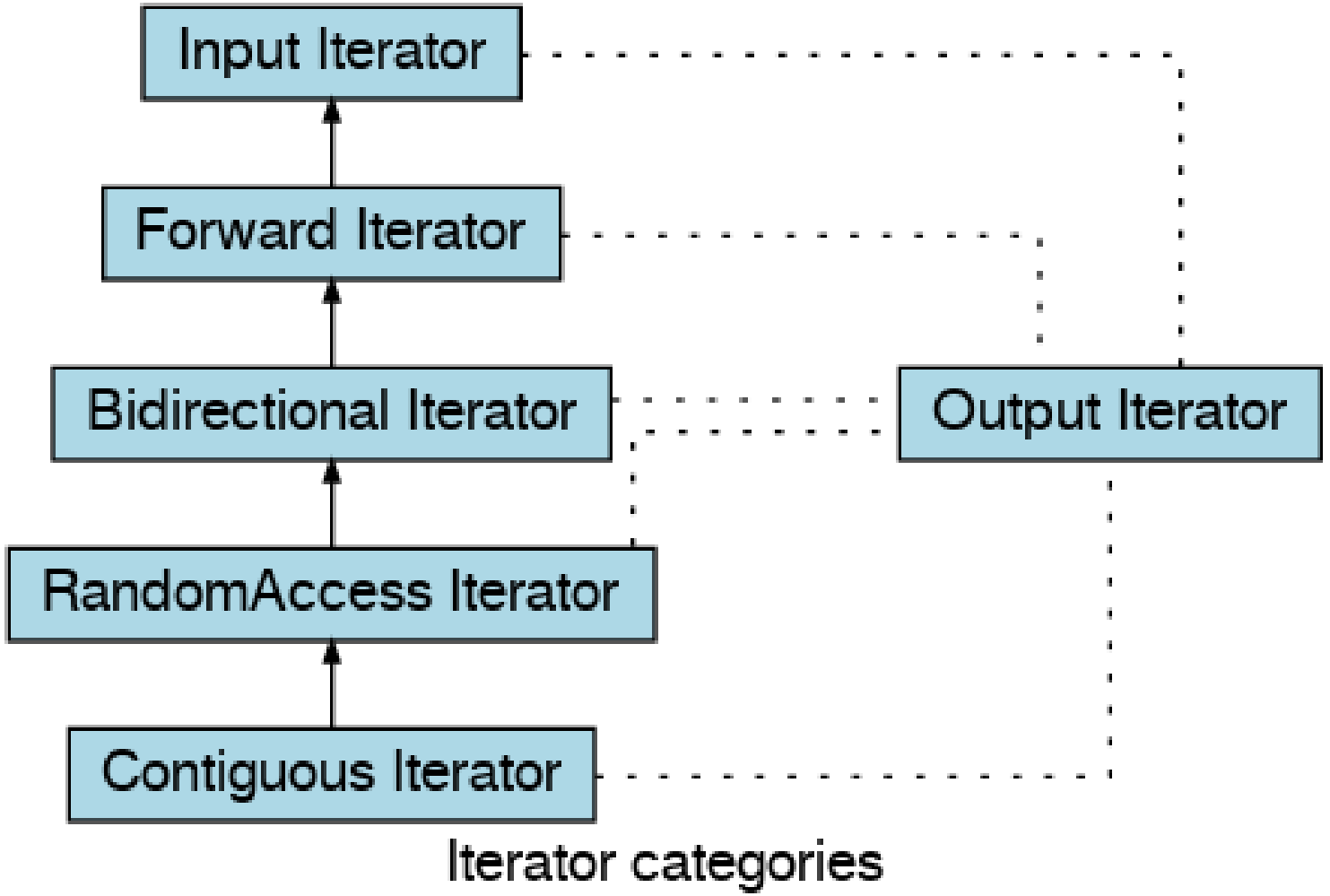
Примеры: `std::vector<T>::iterator`, `std::array<T, N>::iterator`

Output iterator

Любой из указанных выше итераторов дополнительно может принадлежать категории **output iterator (итератор вывода)**, если результату разыменования можно присвоить значение.

```
OutputIterator iterator = ...;
*iterator = 0;
```

Отношения между категориями



Обобщенные функции для работы с итераторами

Обобщенные функции для работы с итераторами

Допустим, в процессе разработки алгоритма понадобилось продвинуть итератор на 10 шагов вперед.

Если итератор random access, то стоит написать:

```
iterator += n;  // O(1)
```

Иначе:

```
for (int i = 0; i < 10; ++i) { ++iterator; }
```

Обобщенные функции для работы с итераторами

Аналогичная проблема возникает с подсчетом расстояния между итераторами:

Если итератор random access, то стоит написать:

```
end - begin;  // O(1)
```

Иначе:

```
auto it = begin; size_t distance = 0;
while (it != end) { ++it; ++distance; }
```

Обобщенные функции для работы с итераторами

В качестве решения предлается использовать функции:

- `std::advance`

```
std::advance(iterator, n);  // продвигает iterator на n шагов вперед
```

- `std::next`

```
auto next = std::next(iterator);  // возвращает итератор на следующий элемент
auto next5 = std::next(iterator, 5);  // возвращает итератор на 5 шагов вперед
```

- `std::prev` (аналогичен `std::next`, но шагает назад)
- `std::distance`

```
auto distance = std::distance(begin, end);  // расстояние между begin и end
```

Инвалидация итераторов

Инвалидация итераторов

Итератор, указывающий на недействительные данные или *потенциально* являющийся таковым, является невалидным. Такой итератор не может быть разыменован и использован (приводит к undefined behaviour).

Классический пример - расширение буфера в `std::vector`:

```
std::vector<int> v{1, 2, 3};
auto iterator = v.begin() + 1;
for (int i = 0; i < 100; ++i) {  // наверняка произойдет перевыделение
    v.push_back(i);
}
*iterator = 0;  // undefined behaviour
```

Инвалидация итераторов

https://en.cppreference.com/w/cpp/container (iterator invalidation)

TL;DR:

1. При чтении данных итераторы никогда не инвалидируются.
2. Итераторы инвалидируются всякий раз, когда происходит перевыделение буфера или перехеширование.
3. Итератор на удаленный элемент всегда инвалидируется.
4. Для `std::vector` и `std::deque`: все итераторы на элементы после только что вставленного или удаленного инвалидируются.

Range-based for

Range-based for

В C++11 появился следующий способа обхода контейнеров:

```
std::vector<int> v{1, 2, 3, 4};
for (int x : v) {
  std::cout << x << ' ';
}
```

```
std::vector<int> v{1, 2, 3, 4};
for (int& x : v) {
  x = 0;
}
```

```
std::map<int, std::string> m{{1, "one"}, {2, "two"}, {3, "three"}};
for (const auto& item : m) {
  std::cout << item->first << ": " << item->second << '\n';
}
```

Range-based for: как это работает

Цикл вида

```
for (<type> value : container) {
  // ...
}
```

Эквивалентен следующему коду (он подставляется неявно компилятором):

```
for (auto it = container.begin(), end = container.end(); it != end; ++it) {
  <type> value = *it;
  // ...
}
```

Range-based for

Работает ли range-based for с массивами?

```
auto array = new int[4]{1, 2, 3, 4};
for (auto x : array) {  // ???
    // ...
}
```

```
int array[] = {1, 2, 3, 4};
for (auto x : array) {  // ???
    // ...
}
```

Range-based for

Работает ли range-based for с массивами?

```
auto array = new int[4]{1, 2, 3, 4};
for (auto x : array) {  // CE
  // ...
}
```

```
int array[] = {1, 2, 3, 4};
for (auto x : array) {  // Ok
  // ...
}
```

По указателю невозможно узнать размер массива. Но по массиву можно узнать его размер с помощью `sizeof`.

Range-based for: `std::begin`, `std::end`

У C-массивов нет методов `.begin()` и `.end()`. Но можно реализовать внешние функции `std::begin(container)` и `std::end(container)`, которые вызывают `begin` и `end`, если они есть. Если их нет, то работает специализация определенная для C-массивов.

```
for (auto it = std::begin(container), end = std::end(container); it != end; ++it) {
  <type> value = *it;
    // ...
}
```

```
template <class T, size_t N>
T* begin(T (&array)[N]) {
  return array;
}
template <class T, size_t N>
T* end(T (&array)[N]) {
  return array + N;
}
```

Range-based for: собственные контейнеры

Чтобы range-based for работал с пользовательскими контейнерами необходимо

- Либо реализовать методы `.begin()`, `.end()`, возвращающие итераторы
- Либо реализовать внешние функции `begin(container)`, `end(container)`

В зависимости от того, что будет найдено компилятором, будет использован первый или второй вариант.