Итераторы

Задача: линейный поиск в массиве

```
template <typename T>
??? find(T* array, std::size_t size, const T& value)
```

Предложите тип возвращаемого значения.

Задача: линейный поиск в массиве

```
template <typename T>
??? find(T* array, std::size_t size, const T& value)
```

Предложите тип возвращаемого значения.

- ✓ `T*` Указатель на найденный элемент
- ✓ `std::size_t/int` Индекс найденного элемента
- ✗ `T&` Ссылка на элемент
- ✗ `bool`

Индекс если элемент не найден

int: -1

std::size_t:

- static_cast<std::size_t>(-1)
- std::numeric_limits<std::size_t>::max()

Указатель

```
template <typename T>
T* find(T* array, std::size_t size, const T& value) {
    for (std::size_t i = 0; i < size; ++i) {
        if (array[i] == value) {
            return &array[i];
        }
    }
    return nullptr;
}
```

Задача: линейный поиск в списке

```
template <typename T>
??? find(Node<T> head, const T& value)
```

Предложите тип возвращаемого значения.

Задача: линейный поиск в списке

```
template <typename T>
??? find(Node<T> head, const T& value)
```

Предложите тип возвращаемого значения.

- ✓ `Node<T>*` Указатель на найденный элемент
- ✗ `std::size_t/int` Индекс найденного элемента
- ✗ `T&` Ссылка на элемент
- ✗ `bool`

Реализация find для списка

```
template <typename T>
Node<T>* find(Node<T>* head, const T& value) {
    for (Node<T>* cur = head; cur != nullptr;
                              cur = cur->next_) {
        if (cur->value_ == value) {
            return cur;
        }
    }
    return nullptr;
}
```


https://godbolt.org/z/E7efa9e4e

Наблюдения

- Обе функции find — это один и тот же алгоритм.
- Реализации отличаются внутренней структурой контейнера и способом его обхода.

=> Нам нужен способ реализовать обобщенный алгоритм, не зависящий от контейнера

=> Все проблемы проектирования решаются добавлением нового уровня косвенности. Кроме проблемы слишком большого количества уровней косвенности.

Обобщенный find

```
template <typename Iterator, typename T>
Iterator find(Iterator first, Iterator last,
              const T& value) {
  for (; first != last; ++first) {
    if (*first == value) {
      return first;
    }
  }
  return last;
}
```


algorithm/find

Обобщенный find

```
template <typename Iterator, typename T>
Iterator find(Iterator first, Iterator last,
              const T& value) {
  for (; first != last; ++first) {
    if (*first == value) {
      return first;
    }
  }
  return last;
}
```

Какие операции должен поддерживать Iterator?

Обобщенный find

```
template <typename Iterator, typename T>
Iterator find(Iterator first, Iterator last,
              const T& value) {
  for (; first != last; ++first) {
    if (*first == value) {
      return first;
    }
  }
  return last;
}
```

- Сравнение
- Инкремент
- Разыменование

Реализуем наивный итератор для списка

```
template <typename T>
class List {
 public:
  class Iterator { ... };
};
```

Итератор: сравнение

```
class Iterator {
 public:
  explicit Iterator(Node* current) :
    current_(current) {}

  bool operator==(const Iterator& other) const {
    return current_ == other.current_;
  }

 private:
  Node* current_;
};
```

Итератор: разыменование

```
class Iterator {
 public:
  T& operator*() { return current_->value_; }

  T* operator->() { return &current_->value_; }

  ...
};
```

Итератор: инкремент

```
class Iterator {
 public:
  Iterator& operator++() {
    current_ = current_->next_;
    return *this;
  }

  Iterator operator++(int) {
    auto old = *this;
    ++(*this);
    return old;
  }
  ...
};
```

Итератор: begin & end

```
template <typename T>
class List {
 public:

  Iterator begin() { return Iterator(head_); }

  Iterator end() { return Iterator(nullptr); }

};
```

Резюме

- Пока ничего нового
- И это работает. Точнее, делает вид, что работает.
- https://godbolt.org/z/Enzor7xco

STL-совместимый итератор

- Интерфейс
- Поведение

Интерфейс

```
template <typename T>
class Iterator {
 public:
  using difference_type = ...;
  using value_type = ...;
  using pointer = ...;
  using reference = ...;
  using iterator_category = ...;
  // + Операторы в зависимости от iterator_category
};
```


iterator/iterator_traits

Интерфейс: умолчания для простого случая

```
template <typename T>
class Iterator {
 public:
  using difference_type = std::ptrdiff_t;
  using value_type = T;
  using pointer = T*;
  using reference = T&;
  using iterator_category = ...;
  // + Операторы в зависимости от iterator_category
};
```

Категории итераторов

- Legacy**Input**Iterator
- Legacy**Output**Iterator
- Legacy**Forward**Iterator
- Legacy**Bidirectional**Iterator
- Legacy**RandomAccess**Iterator

Категории итераторов

Iterator category	Operations and storage requirement						
	write	read	increment		decrement	random access	contiguous storage
			without multiple passes	with multiple passes			
LegacyOutputIterator	Required		Required				
LegacyInputIterator (mutable if supports write operation)		Required	Required				
LegacyForwardIterator (also satisfies *LegacyInputIterator*)		Required	Required	Required			
LegacyBidirectionalIterator (also satisfies *LegacyForwardIterator*)		Required	Required	Required	Required		
LegacyRandomAccessIterator (also satisfies *LegacyBidirectionalIterator*)		Required	Required	Required	Required	Required	
LegacyContiguousIterator[1] (also satisfies *LegacyRandomAccessIterator*)		Required	Required	Required	Required	Required	Required

iterator

Теги категорий

struct **input_iterator_tag** { };

struct **output_iterator_tag** { };

struct **forward_iterator_tag** : input_iterator_tag { };

struct **bidirectional_iterator_tag** : forward_iterator_tag { };

struct **random_access_iterator_tag** : bidirectional_iterator_tag { };

iterator/iterator_tags

Например, так

```
template <typename T>
class Iterator {
 public:
  using difference_type = std::ptrdiff_t;
  using value_type = T;
  using pointer = T*;
  using reference = T&;
  using iterator_category = std::forward_iterator_tag;
  // + Операторы
};
```

Эксперимент

```
template <typename T, typename IteratorTag>
class Array {
 public:
  class Iterator {
   public:
    using iterator_category = IteratorTag;
    // Далее все операции для RandomAccess
  };
  ...
};
```

Эксперимент

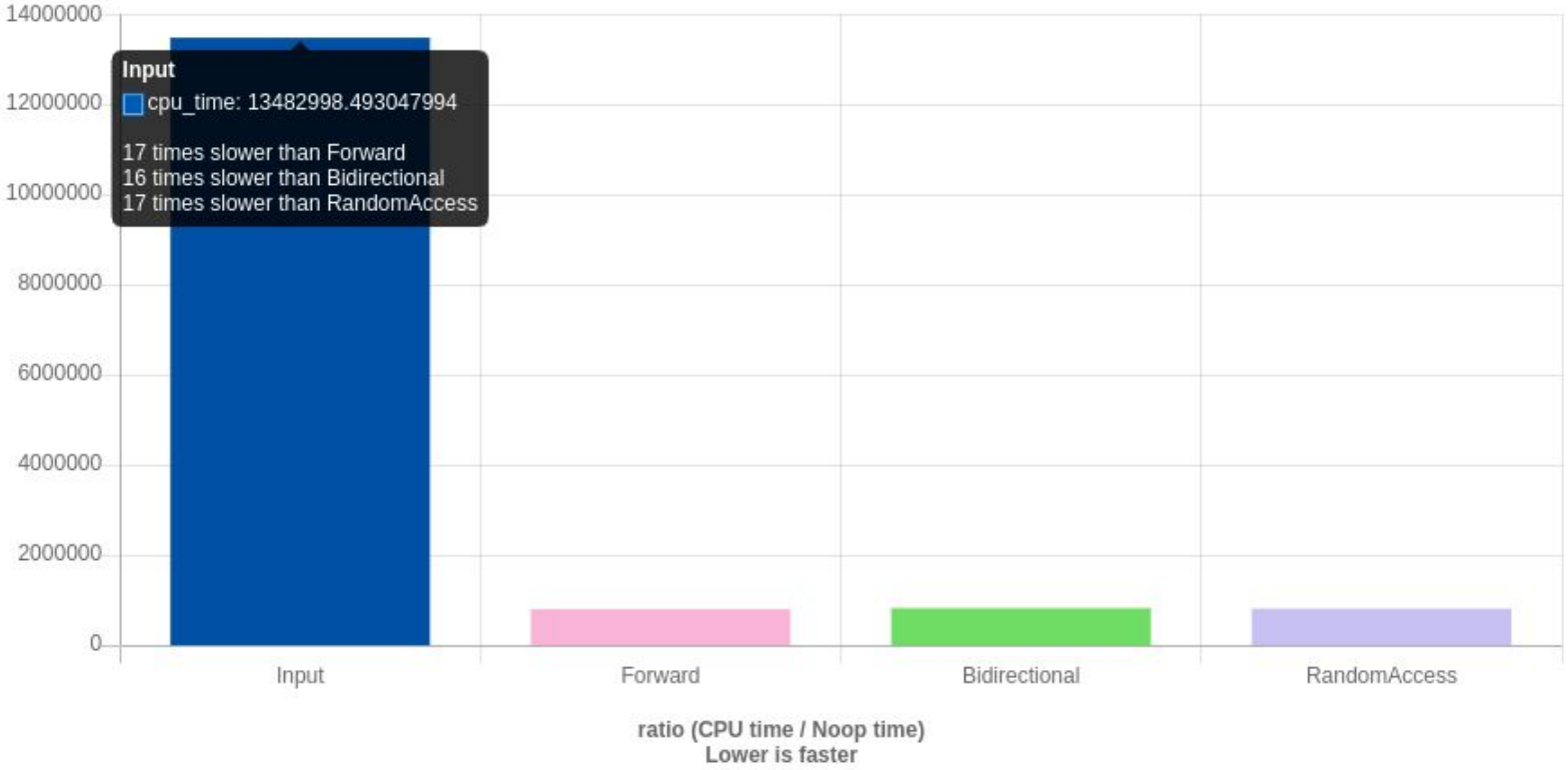
```
template <typename T, typename IteratorTag>
class Array { ... };

template <typename T>
using ArrayInput = Array<T, std::input_iterator_tag>;

template <typename T>
using ArrayRandomAccess
    = Array<T, std::random_access_iterator_tag>;
```

Эксперимент

```
auto arr = create_array<ArrayInput<int>>();
std::vector<int> v(arr.begin(), arr.end());
```

vs

```
auto arr = create_array<ArrayRandomAccess<int>>();
std::vector<int> v(arr.begin(), arr.end());
```



https://quick-bench.com/q/Q6vi67Sh1oZtq_nozgdBJgXEH5M

Как это работает

https://godbolt.org/z/bozETK6xb

BitIterator

https://godbolt.org/z/qM7KbGfoG

Вопросы?