ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

Департамент программной инженерии

Алгоритмы и структуры данных

Семинар №12. 2021-2022 учебный год

Нестеров Роман Александрович, *ДПИ ФКН и НУЛ ПОИС*

Бессмертный Александр Игоревич, *ДПИ ФКН*

Higher School of Economics

Faculty of Computer science


En mørk morgen kan bli en lys dag

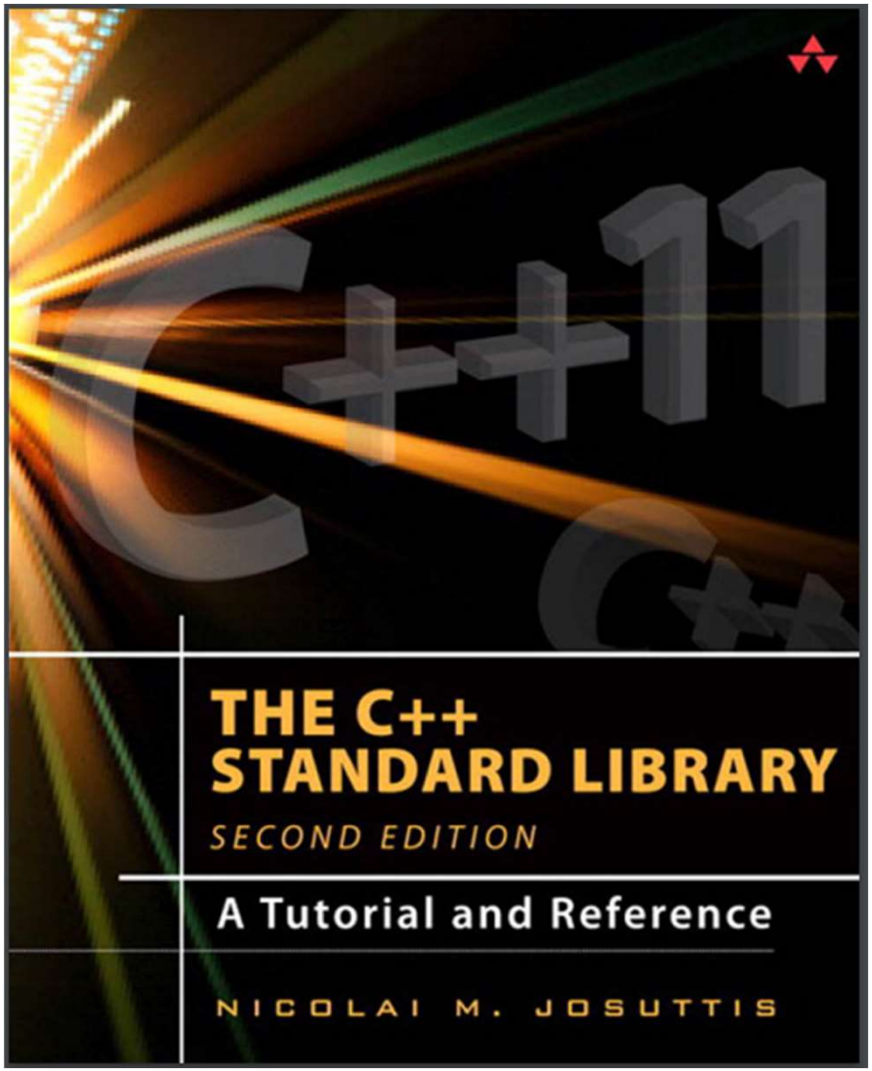
Где мы?

Специальные структуры данных			
Вероятностные структуры			
Хеширование, хеш-таблицы ✓		**Disjoint sets** ✓	
Сбалансированные деревья ✓		**B-деревья** ✓	
Рекурсивные структуры данных			
Список ✓		**Дерево** ✓	
Элементарные (линейные) структуры данных			
Массив ✓	**Строка** ✓	**Стек** ✓	**Очередь** ✓

План

- Стандартная библиотека шаблонов STL и ее основные компоненты
- Последовательные контейнеры
- Итераторы и алгоритмы
- Умные указатели

Стандартная библиотека шаблонов STL


C++11
THE C++
STANDARD LIBRARY
SECOND EDITION
A Tutorial and Reference
NICOLAI M. JOSUTTIS


СТАНДАРТНАЯ БИБЛИОТЕКА
C++
Справочное руководство
ВТОРОЕ ИЗДАНИЕ
НИКОЛАИ М. ДЖОСАТТИС

Историческая справка

- Первые идеи использования универсальных средств разработки – **1979 г., Александр Степанов**
- Библиотека шаблонов для языка ADA – **1987 г., Александр Степанов и Дэвид Массер**
- Представление основных идей STL – **1993 г.,** конференция по стандартизации C++

Основные компоненты

Принципы Generic Programming

- **Переиспользование**

  Пишем меньше, делаем больше
- **Гибкость**

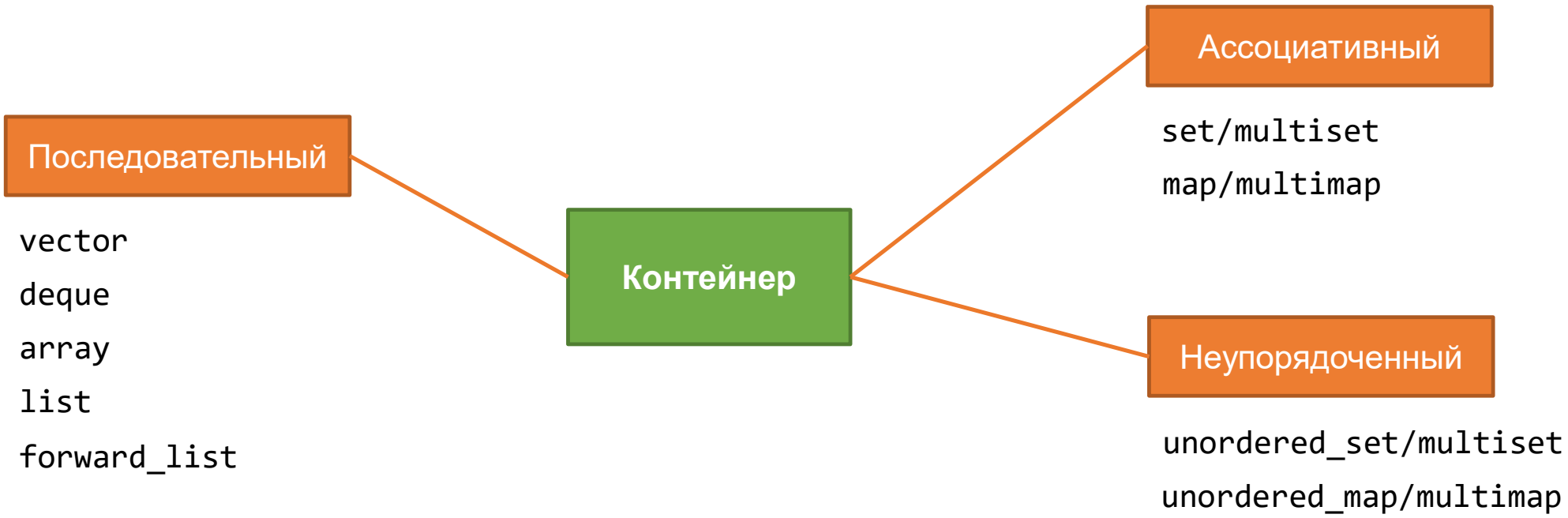
  Итераторы отделяют контейнеры от алгоритмов
- **Эффективность работы**

Контейнер

- Является абстрактным типом данных (АТД)
- Параметризуется типом содержащихся элементов

- В каждом контейнере объявлены различные «свойства» (*traits*): `iterator, const_iterator, value_type` и др.
- Фабричные методы для создания итераторов: `begin()/end()` и `rbegin()/rend()`

Основные типы контейнеров



Основные возможности контейнера

Инициализация

- Конструктор по умолчанию
- Конструктор копирования
- Конструктор перемещения
- Конструктор, принимающий список инициализаторов
- Инициализация из стандартного потока ввода (с помощью итератора)
- …

Присваивание и `swap()`

- Присваивание контейнеров влечет за собой **поэлементное копирование**
- Обмен содержимого контейнеров также может быть выполнен с помощью функции `swap`:
  `a.swap(b)` или `swap(a, b)`

Размер контейнера

- Проверка пустоты `empty()` (`a.begin() == a.end()`)
- Текущее количество элементов `size()`
- Максимальное количество элементов `max_size()`

Сравнение

Стандартные операторы сравнения ==, !=, <, <=, >, >=

1. Сравниваемые контейнеры должны быть одного типа
2. Два контейнера равны, если их элементы равны и расположены в одном и том же порядке
3. Сравнение “<“ выполняется посредством лексикографического сравнения

Доступ к элементам

Все контейнеры предоставляют интерфейс итератора

- Цикл for по диапазону

```
for (auto& elem : container) {. . .}
```

- Цикл for с итераторами

```
for (auto pos = c.begin(); pos != c.end(); ++pos) {. . .}
```

Доступ к элементам

Все контейнеры предоставляют интерфейс итератора

- Цикл for по диапазону

```
for (auto& elem : container) {. . .}
```

- Цикл for с итераторами

```
for (auto pos = c.begin(); pos != c.end(); ++pos) {. . .}
```

В результате модификации контейнера (вставка, удаление, очистка clear()) итераторы могут быть денонсированы (invalidated).

Последовательные контейнеры

Массив или связный список

Контейнер	Реализация	Возможности
`vector<T>`	Динамический массив	• Произвольный доступ • Вставка, удаление элементов в конец *и необходимую позицию* • Изменение размера и др.
`deque<T>`	Динамический массив	• Вставка и удаление в начало или конец *и необходимую позицию*
`array<T>`	Фиксированный массив	
`list<T>`	Двусвязный список	• Нет произвольного доступа • Специальные методы склейки и слияния списков
`forward_list<T>`	Односвязный список	

Избегаем перераспределения памяти `vector`

В случае достижения емкости вектора происходит перераспределение памяти и удвоение его емкости.

- Вызов `reserve()`, который сразу гарантирует заданную емкость массива

```
std::vector<int> v;
v.reserve(150);
```

- Передача количества элементов в конструктор

```
std::vector<T> v(15);
```

Контейнер `vector<bool>`

- Специальная реализация динамического массива, где под каждый элемент отводится **1 бит**.
- Фактически, представляет собой *битовое поле* динамического размера.

Контейнер `vector<bool>`

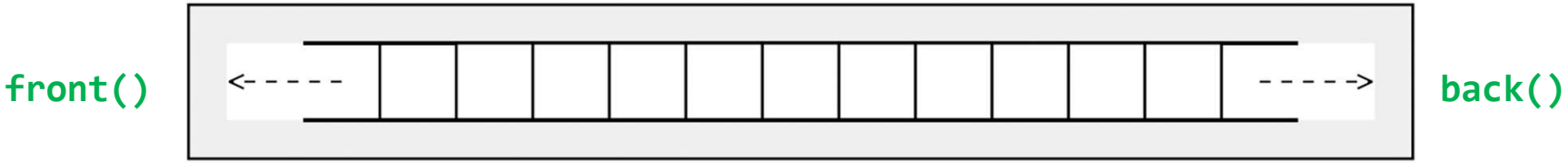
- Специальная реализация динамического массива, где под каждый элемент отводится **1 бит**.
- Фактически, представляет собой *битовое поле* динамического размера.

Дополнительные операции:

```
container.flip();
container[i].flip();
```

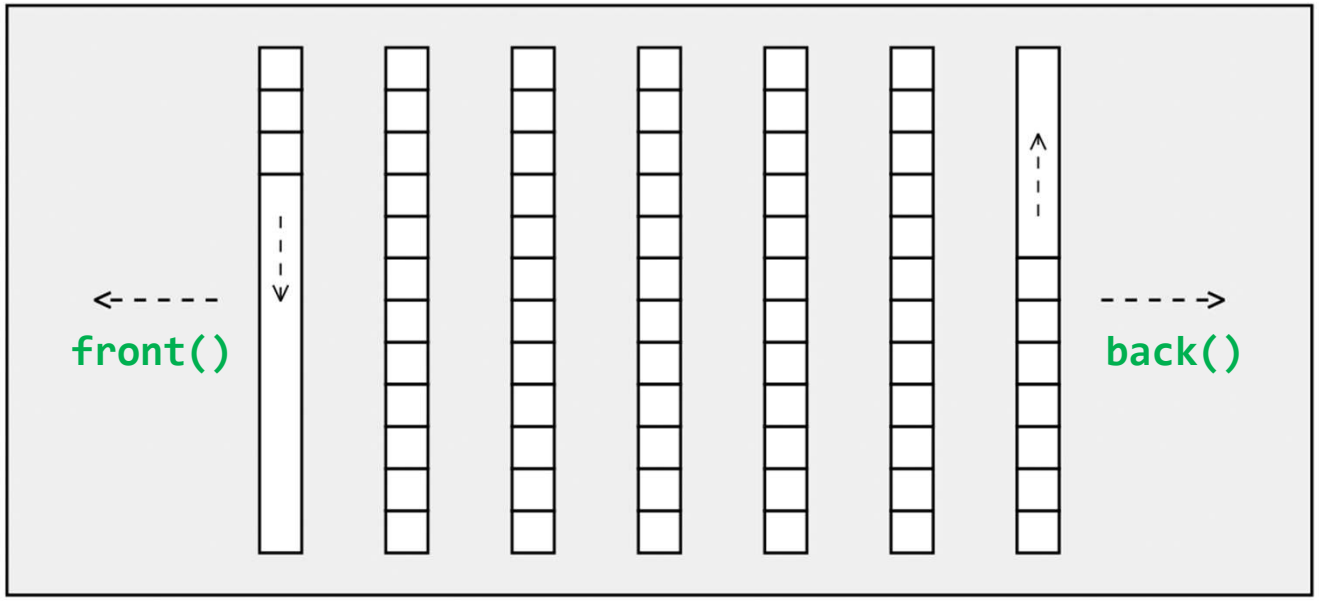
Контейнер deque

Динамический массив, открытый с обоих концов



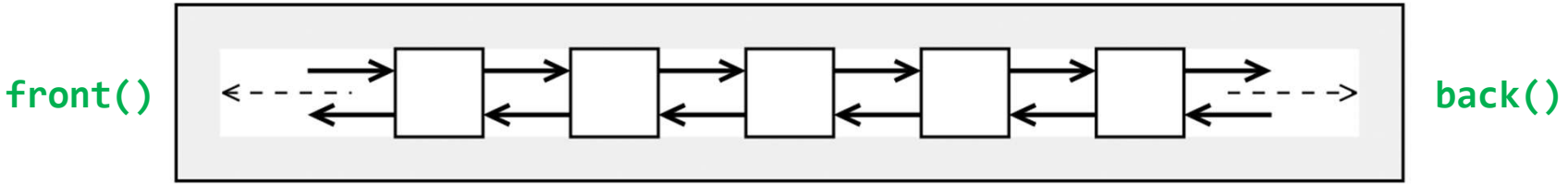
Контейнер deque

Внутренняя организация позволяет экономить при перераспределении памяти



Контейнер `list`

Стандартный двусвязный список с поддержкой специальных операций склейки (splice) и слияния (merge)



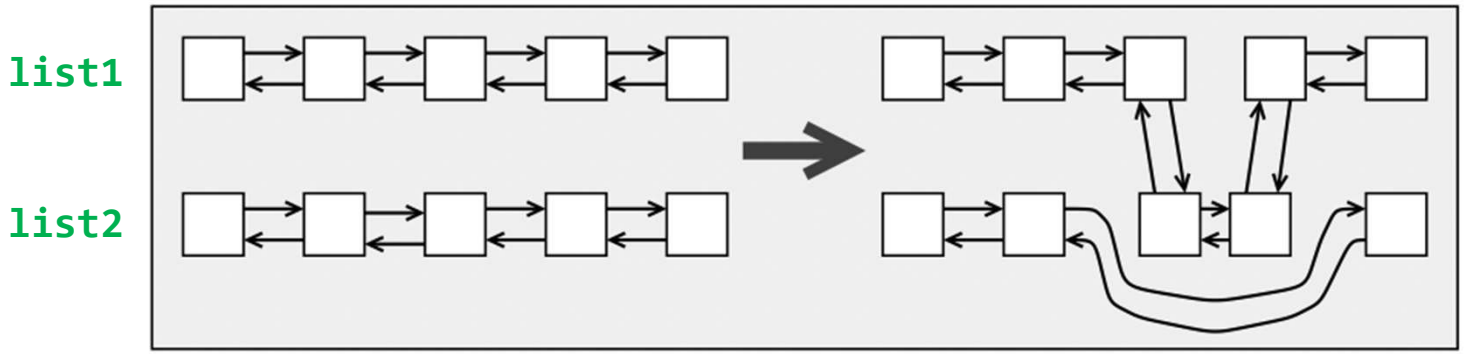
Контейнер `list`

Стандартный двусвязный список с поддержкой специальных операций склейки (splice) и слияния (merge)

Метод	Результат
`list.unique()`	Удаление дубликатов
`list.splice(pos, list2)` и его вариации с диапазонами	Перемещает все элементы `list2` в `list` перед итератором pos
`list.sort()`	Сортирует список (сравнение <)
`list.merge(list2)`	Слияние `list2` с `list`; результат записывается в `list` в отсортированном виде
`list.reverse()`	Переворот списка

Контейнер `list`

Стандартный двусвязный список с поддержкой специальных операций склейки (splice) и слияния (merge)




```
list1.splice(l1_p, list2, l2_p1, l2_p2)
```

Итераторы в STL

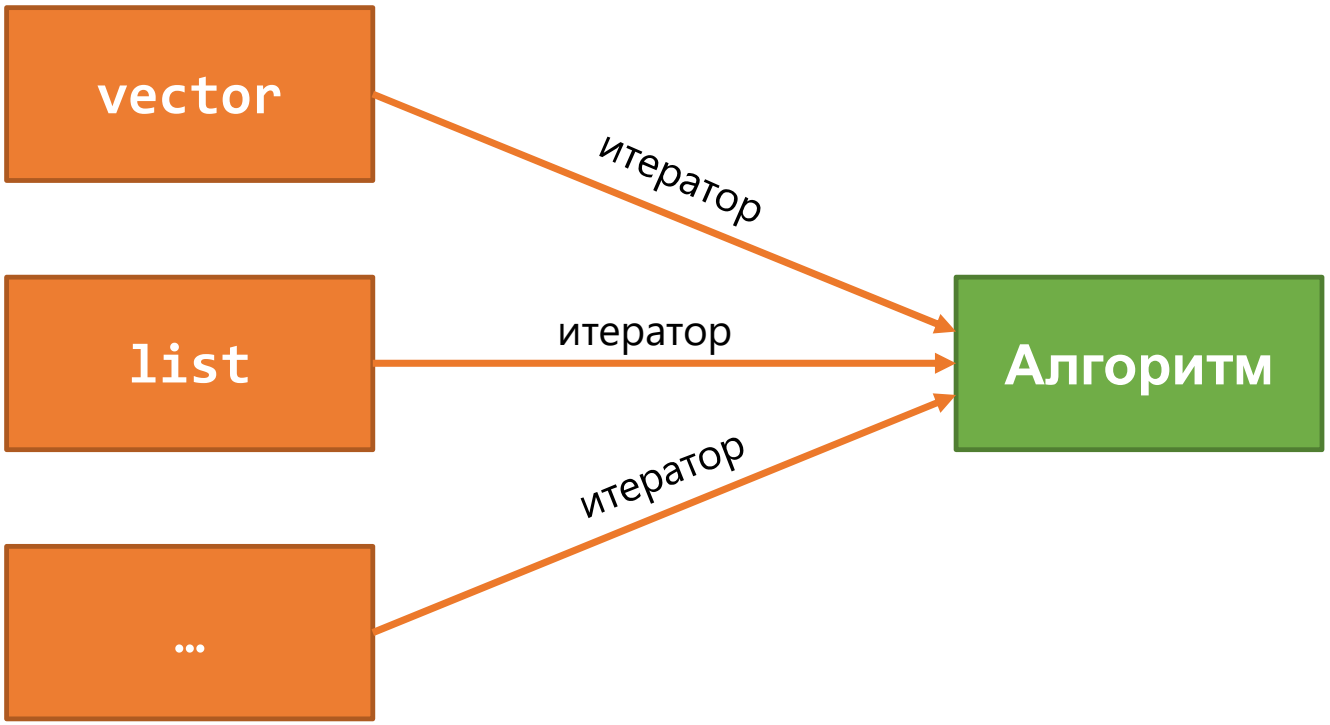
Итераторы

Реализация одноименного паттерна:

- Организация последовательного доступа к элементам контейнера без раскрытия его внутренней организации
- Инкапсуляция процесса последовательного перебора

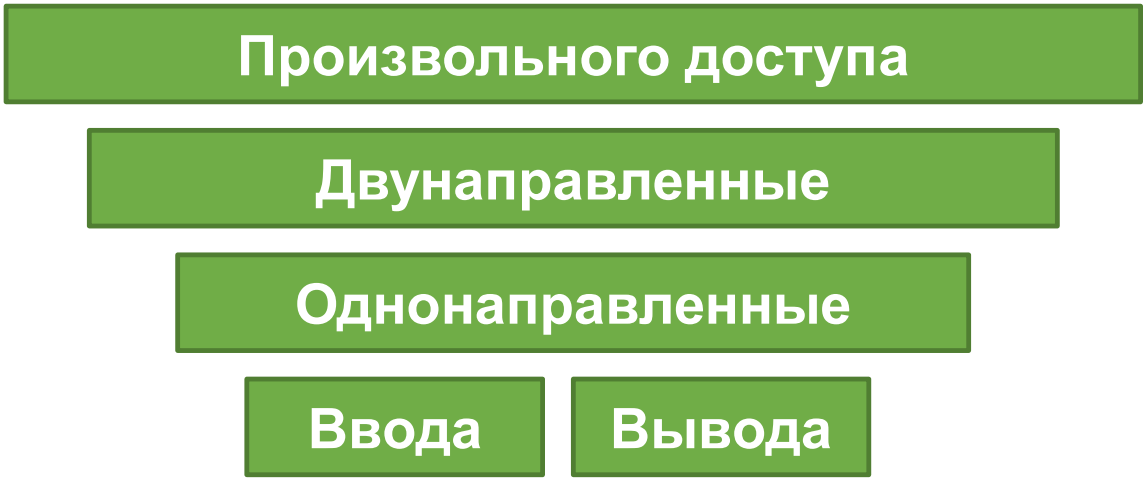
- Обобщение указателей – итераторы суть объекты, которые указывают на другие объекты

Мост между алгоритмами и контейнерами



Категории итераторов

Образуют **иерархию** в зависимости от доступных возможностей



Итератор ввода

Применяется для чтения элементов из входной последовательности

- Копирование и оператор присваивания
- Операторы == и !=
- Оператор разыменовывания *
- Префиксный и постфиксный инкремент ++a, a++

Итератор ввода

Применяется для чтения элементов из входной последовательности

```
std::vector<int> v;
for (istream_iterator<int> i = std::cin;
     i != istream_iterator<int> ();
     ++i) {
        v.push_back (*i);
}
```

Итератор произвольного доступа

Предоставляют алгоритму произвольный доступ к содержимому контейнера, который **его поддерживает**

- Арифметическое присваивание += и -=
- Сложение и вычитание (с учетом симметрии аргументов)
- Операторы сравнения (по порядку)
- Оператор индексации [ ]

Итератор произвольного доступа

Предоставляют алгоритму произвольный доступ к содержимому контейнера, который **его поддерживает**

```
std::vector<int> v(1, 1);
v.push_back(7); v.push_back(19); v.push_back(24);

std::vector<int>::iterator i = v.begin();
std::vector<int>::iterator j = i + 2;
i += 3;

(j > i) ? std::cout << "j > i" : std::cout << "not (j > i)";
```

Алгоритмы STL

Общие сведения

Алгоритмы работают с итераторами, а не с самими контейнерами

- Каждый контейнер предоставляет обычный итератор и константный итератор (например, **`list`** предоставляет двунаправленный, а **`vector`** – произвольного доступа)
- Шаблоны обеспечивают *типобезопасность* на этапе компиляции для комбинации контейнера, итераторов и алгоритмов

Типы алгоритмов

- Поиск и сортировка элементов в контейнере
- Численные расчеты – НОК, НОД, частичные суммы и пр.
- Алгоритмы, не меняющие содержимое контейнера
- Алгоритмы, меняющие содержимое контейнера
- Поиск минимума и максимума

std::adjacent_find()

```
std::vector<int> v1{0, 1, 2, 3, 40, 40, 41, 41, 5};
auto i1 = std::adjacent_find(v1.begin(), v1.end());

if (i1 == v1.end()) {
   std::cout << "No matching adjacent elements\n";
} else {
   std::cout << "The first adjacent pair is at "
          << std::distance(v1.begin(), i1) << ", *i1 = "
          << *i1 << '\n';
}

auto i2 = std::adjacent_find(v1.begin(), v1.end(), std::greater<int>());
if (i2 == v1.end()) {
   std::cout << "The entire vector is in ascending order\n";
} else {
   std::cout << "The last element in the non-decreasing subsequence is at "
          << std::distance(v1.begin(), i2) << ", *i2 = " << *i2 << '\n';
}
```

std::replace()

Замещает все вхождения некоторого значения в заданном итераторами диапазоне

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(1);

std::replace(v.begin (), v.end (), 1, 99);
```

std::remove_if()

Удаляет в заданном итераторами диапазоне элементы контейнера, для которых некоторый предикат истинен

```
#include <iostream>
#include <algorithm>

struct is_odd {
   bool operator () (int i) { return (i % 2)==1; }
};

int main () {
   int myints[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
   int *pbegin = myints;
   int *pend = myints + sizeof(myints) / sizeof (*myints);
   pend = std::remove_if(pbegin, pend, is_odd());
   return 0;
}
```

Умный указатель

Проблемы с указателями

```
void func() {
    T *ptr = new T;

    int x;
    std::cin >> x;

    if (x > 3) return;

    delete ptr;
}

int main() {
    func();
}
```

Проблемы с указателями

```
void func() {
    T *ptr = new T;

    int x;
    std::cin >> x;

    if (x > 3) return;

    delete ptr;
}

int main() {
    func();
}
```

```
void func() {
    T *ptr = new T;

    int x;
    std::cin >> x;

    ptr = new T;

    delete ptr;
}

int main() {
    func();
}
```

Проблемы с указателями

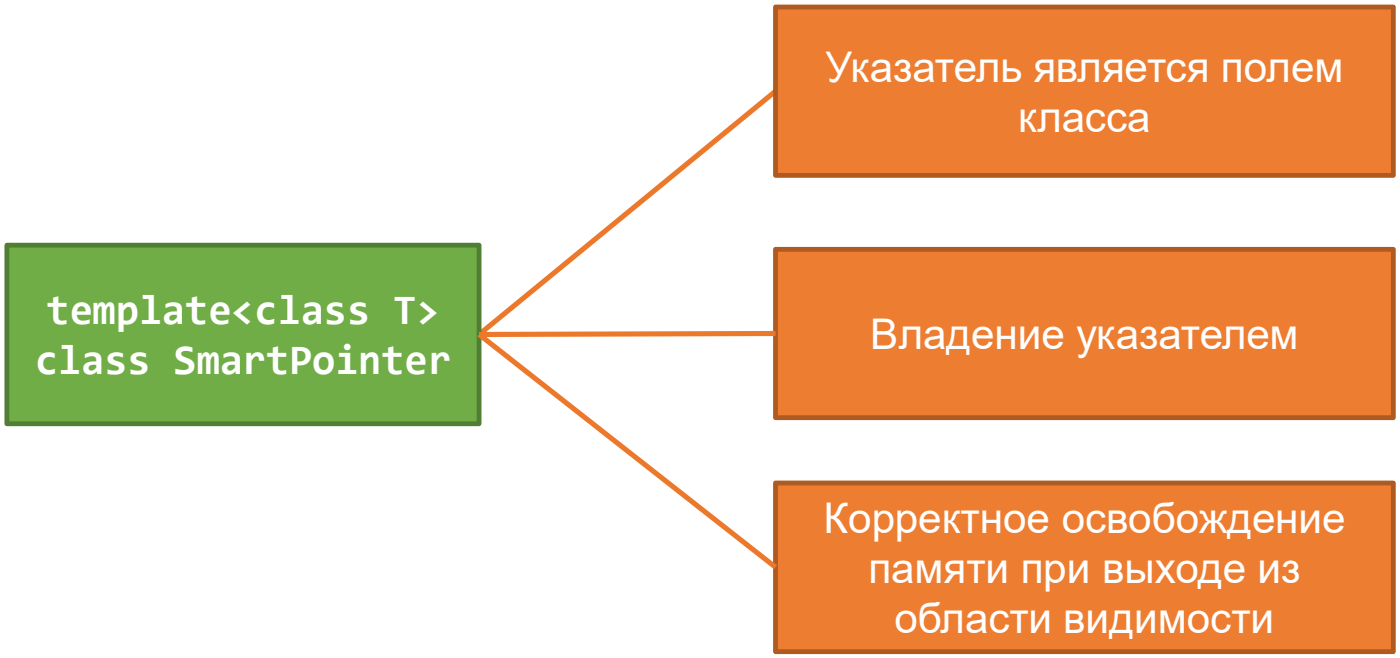
- Утечки памяти – потеря доступа к указателю
- Разыменовывание **`nullptr`**
- Попытка доступа к неинициализированной области динамической памяти
- Повторное удаление уже удаленного объекта

Проблемы с указателями

- Утечки памяти – потеря доступа к указателю
- Разыменовывание **`nullptr`**
- Попытка доступа к неинициализированной области динамической памяти
- Повторное удаление уже удаленного объекта

Использование класса для управления указателями и освобождения памяти

Класс для управления указателем



Класс для управления указателем

```
template<class T>
class SmartPtr {
private:
    T *ptr_;

public:
    SmartPtr(T *ptr = nullptr) {
        ptr_=ptr;
    }

    ~SmartPtr() {
        delete ptr_;
    }
}
```

Класс для управления указателем

```
template<class T>
class SmartPtr {
private:
    T *ptr_;

public:
    SmartPtr(T *ptr = nullptr) {
        ptr_=ptr;
    }

    ~SmartPtr() {
        delete ptr_;
    }

    T& operator*() { return *ptr_; }
    T* operator->() { return ptr_; }
}
```

Класс для управления указателем

```
template<class T>
class SmartPtr {
private:
    T *ptr_;

public:
    SmartPtr(T *ptr = nullptr) {
        ptr_=ptr;
    }

    ~SmartPtr() {
        delete ptr_;
    }

    T& operator*() { return *ptr_; }
    T* operator->() { return ptr_; }
}
```

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }
};
```

```
void func() {
    SmartPtr<Test> ptr(new Test);
    int x;
    std::cin >> x;

    if (x > 3) return;
}
```

Класс для управления указателем

Проблемы при создании одного умного указателя на основе другого (**копирование**)

```
template<class T>
class SmartPtr {
private:
    T *ptr_;

public:
    SmartPtr(T *ptr = nullptr) {...}

    ~SmartPtr() {...}

    T& operator*() { return *ptr_; }
    T* operator->() { return ptr_; }
}
```

```
class Test {
public:
    Test() {. . .}
    ~Test() {. . .}
};
```

```
void func() {
    SmartPtr<Test> ptr(new Test);
    SmartPtr<Test> ptr2(ptr);
    SmartPtr<Test> ptr3;
    ptr3 = ptr2;
}
```

Класс для управления указателем

Проблемы при создании одного умного указателя на основе другого (**копирование**)

```
void func() {
    SmartPtr<Test> ptr(new Test);
    SmartPtr<Test> ptr2(ptr);
    SmartPtr<Test> ptr3;
    ptr3 = ptr2;
}
```

Класс для управления указателем

Передаем *владение* указателем из источника в объект назначения (**перемещение**)

```
template<class T>
class SmartPtr {
private:
    T *ptr_;

public:
    SmartPtr(T *ptr = nullptr) {...}

    ~SmartPtr() {...}

    T& operator*() { return *ptr_; }
    T* operator->() { return ptr_; }
}
```

```
SmartPtr(SmartPtr& p) {
    ptr_ = p.ptr_;
    p.ptr_ = nullptr;
}
```

```
SmartPtr& operator=(SmartPtr& p) {
    if (&a = this) return *this;

    delete ptr_;
    ptr_ = p.ptr_;
    p.ptr_ = nullptr;

    return *this;
}
```

Класс для управления указателем

Передаем *владение* указателем из источника в объект назначения (**перемещение**)

```
void func() {
    SmartPtr<Test> ptr(new Test);
    SmartPtr<Test> ptr2(ptr);
    SmartPtr<Test> ptr3;
    ptr3 = ptr2;
}
```

Класс `std::auto_ptr`

- Первая попытка создания умного указателя (C++98)
- Реализация перемещения посредством копирования, что приводит к проблемам передачи по значению
- Освобождение памяти только оператором **`delete`**

Класс `std::auto_ptr`

- Первая попытка создания умного указателя (C++98)
- Реализация перемещения посредством копирования, что приводит к проблемам передачи по значению
- Освобождение памяти только оператором **`delete`**

Строгое определение семантики перемещения (C++11) и нормальные умные указатели

Умный указатель `std::unique_ptr`

- Замена `std::auto_ptr`
- **Единолично** владеет переданным ему динамически выделенным объектом
- Заголовочный файл `<memory>`
- Семантика копирования по умолчанию **отключена**

Умный указатель `std::unique_ptr`

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }
};
```

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<Test> ptr(new Test);

    return 0;
}
```

Умный указатель `std::unique_ptr`

Корректная реализация семантики перемещения

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }
};
```

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<Test> ptr(new Test);
    std::unique_ptr<Test> ptr2;

    ptr2 = ptr;

    return 0;
}
```

Умный указатель `std::unique_ptr`

Корректная реализация семантики перемещения

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }
};
```

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<Test> ptr(new Test);
    std::unique_ptr<Test> ptr2;

    ptr2 = ptr;

    return 0;
}
```

Умный указатель `std::unique_ptr`

Корректная реализация семантики перемещения

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }
};
```

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<Test> ptr(new Test);
    std::unique_ptr<Test> ptr2;

    ptr2 = std::move(ptr);

    return 0;
}
```

Умный указатель `std::unique_ptr`

Доступны операторы ***** и **->**, а также проверка владения посредством преобразования к значению типа **bool**

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }

    // перегрузка вывода в поток
};
```

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<Test> ptr(new Test);

    if (ptr) {
        std::cout << *ptr;
    }

    return 0;
}
```

Умный указатель `std::unique_ptr`

Функция **`std::make_unique`** для создания умного указателя

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }

    // перегрузка вывода в поток
};
```

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<Test> ptr(new Test);
    std::cout << *ptr;

    auto ptr2 =
        std::make_unique<Test[]>(7);
    std::cout << *ptr2;

    return 0;
}
```

Умный указатель `std::unique_ptr`

Функция **`std::make_unique`** для создания умного указателя

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }

    // перегрузка вывода в поток
};
```

```
#include <iostream>
#include <memory>

int func1() { throw 0; }
void func2(. . .) { . . . }

int main() {
    func2(std::unique_ptr<Test>(new Test),
          func1());

    return 0;
}
```

Умный указатель `std::unique_ptr`

Функция **`std::make_unique`** для создания умного указателя

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }

    // перегрузка вывода в поток
};
```

```
#include <iostream>
#include <memory>

int func1() { throw 0; }
void func2(. . .) { . . . }

int main() {
    func2(std::unique_ptr<Test>(new Test),
          func1());

    return 0;
}
```

Умный указатель `std::unique_ptr`

Функция **`std::make_unique`** для создания умного указателя

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }

    // перегрузка вывода в поток
};
```

```
#include <iostream>
#include <memory>

int func1() { throw 0; }
void func2(. . .) { . . . }

int main() {
    func2(std::make_unique<Test>(),
          func1());

    return 0;
}
```

Умный указатель `std::unique_ptr`

Возврат умного указателя из функции выполняется **по значению**

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }

    // перегрузка вывода в поток
};
```

```
#include <iostream>
#include <memory>

std::unique_ptr<Test> func1() {
    return std::make_unique<Test>();
}

int main() {
    std::unique_ptr<Test> p = func1();

    return 0;
}
```

Умный указатель `std::unique_ptr`

Передача умного указателя в функцию **по значению** ведет к передаче права владения

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }

    // перегрузка вывода в поток
};
```

```
#include <iostream>
#include <memory>

void func1(std::unique_ptr<Test> p) {
    if (p) std::cout << *p;
}

int main() {
    auto ptr = std::make_unique<Test>();

    func1(std::move(ptr));

    return 0;
}
```

Умный указатель `std::unique_ptr`

Передача объекта («необработанного» указателя) из умного указателя в функцию **по адресу** не передает право владения

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }

    // перегрузка вывода в поток
};
```

```
#include <iostream>
#include <memory>

void func1(Test *t) {
    if (t) std::cout << *t;
}

int main() {
    auto ptr = std::make_unique<Test>();

    func1(ptr.get());

    return 0;
}
```

Умный указатель `std::unique_ptr`

Распространенные ошибки – владение одним и тем же объектом и ручное удаления объекта владения.

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }

    // перегрузка вывода в поток
};
```

```
Test *object = new Test;
std::unique_ptr<Test> ptr1(object);
std::unique_ptr<Test> ptr2(object);
```

```
Test *object = new Test;
std::unique_ptr<Test> ptr1(object);
delete object;
```

Умный указатель `std::unique_ptr`

Распространенные ошибки – владение одним и тем же объектом и ручное удаления объекта владения.

```
class Test {
public:
    Test() {
        std::cout << ”Created”;
    }
    ~Test() {
        std::cout << “Destroyed”;
    }

    // перегрузка вывода в поток
};
```

```
Test *object = new Test;
std::unique_ptr<Test> ptr1(object);
std::unique_ptr<Test> ptr2(object);
```

```
Test *object = new Test;
std::unique_ptr<Test> ptr1(object);
delete ptr;
```

`std::make_unique<…>` предотвращает такие ситуации

Умный указатель `std::unique_ptr`

- Владение и управление динамически выделенным объектом (освобождение памяти в случае выхода из области видимости)
- Совместим с динамическими массивами
- Использование в качестве члена класса (композиция)

Другие умные указатели

- **`std::shared_ptr`** допускает возможность совместного владения одним динамически выделенным объектом
- **`std::weak_ptr`** имеет доступ к объекту, но не считается его владельцем