ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

Департамент программной инженерии

Алгоритмы и структуры данных

Семинар №13. 2021-2022 учебный год

Нестеров Роман Александрович, *ДПИ ФКН и НУЛ ПОИС*

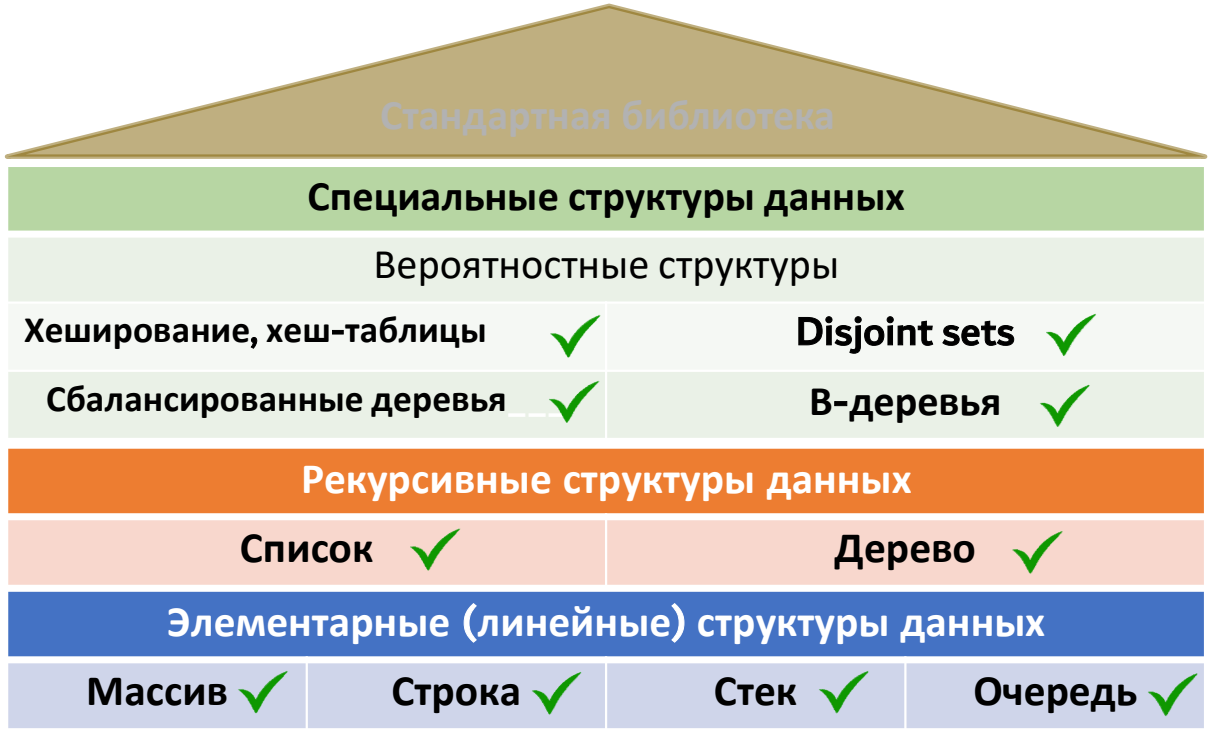
Бессмертный Александр Игоревич, *ДПИ ФКН*

Higher School of Economics

Faculty of Computer science

Error is a discipline through which we advance

Где мы?


Стандартная библиотека
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

- Ассоциативные упорядоченные и не очень контейнеры
- Адаптеры контейнеров, итераторов
- Умный указатель `unique_ptr`

Ассоциативные контейнеры STL

Бинарные деревья поиска

Контейнер	Реализация	Возможности
`set<T>` `set<T, …>`	Сбалансированное бинарное дерево поиска (красно-черное дерево)	• Хранение ключей, автоматически отсортированный в определенном порядке • Оптимизированы для быстрого поиска элементов (поддержка специальных функций) • Двунаправленные итераторы
`multiset<T>` `multiset<T, …>`		• Хранение автоматически сортируемых ключей с возможностью дубликатов
`map<Key, Val>` `map<Key, Val, …>`		• Хранение пар `(key, value)`, отсортированных по ключам
`multimap<Key, Val>` `multimap<Key, Val, …>`		• Хранение пар `(key, value)`, отсортированных по ключам с возможностью дубликатов ключей

Множества и мультимножества

Критерий сортировки ключей

По умолчанию, ключи сортируются с помощью сравнения “<“ (функциональный объект **`std::less<T>`**).

Критерий сортировки должен определять отношение строгого слабого порядка **(strict weak ordering)** на множестве ключей:

1. Если **`x<y`**, то **`!(y<x)`** | Если **`pred(x, y)`**, то **`!pred(x, y)`**
2. Если **`x<y && y<z`**, то **`x<z`** | Если **`pred(x, y) && pred(y, z)`**, то **`pred(x, z)`**
3. **`!(x<x)`** | **`!op(x, x)`**
4. Если **`a==b && b==c`**, то **`a==c`**

Критерий сортировки ключей

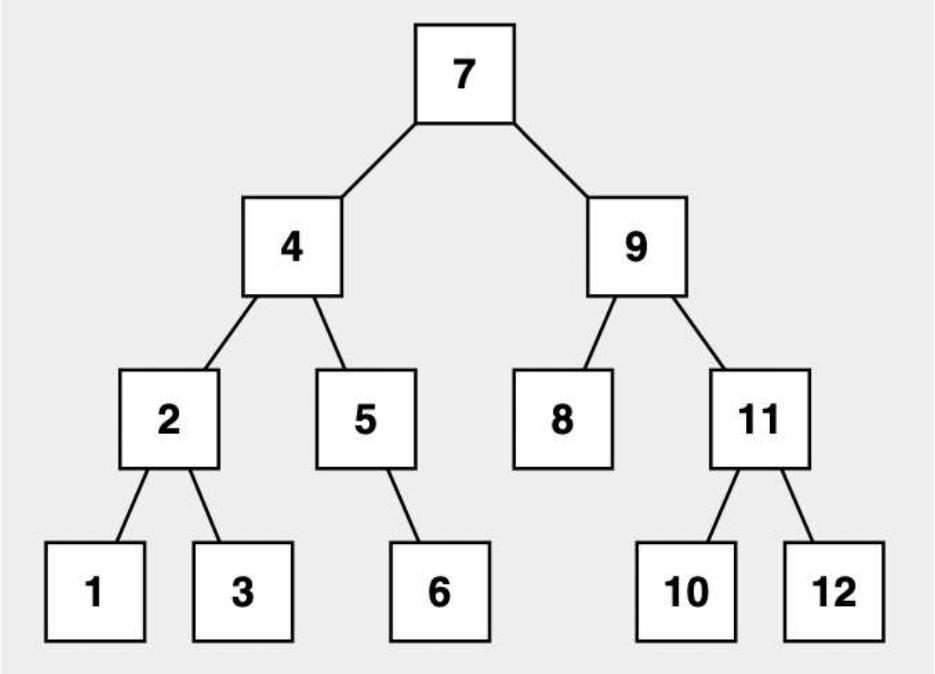
По умолчанию, ключи сортируются с помощью сравнения “**<**“ (функциональный объект **`std::less<T>`**).

Критерий сортировки должен определять отношение строгого слабого порядка **(strict weak ordering)** на множестве ключей:

1. Если **`x<y`**, то **`!(y<x)`** | Если **`pred(x, y)`**, то **`!pred(x, y)`**
2. Если **`x<y && y<z`**, то **`x<z`** | Если **`pred(x, y) && pred(y, z)`**, то **`pred(x, z)`**
3. **`!(x<x)`** | **`!op(x, x)`**
4. Если **`a==b && b==c`**, то **`a==c`**

Как выразить **`a==b`**, используя только **`< (op())`**?

Внутренняя организация и особенности



Нельзя напрямую изменить значение ключа.

- Не предоставляется интерфейс для прямого доступа
- Доступ через итератор не позволяет изменять ключи

Вставка ключа

В случае вставки в обычное множество возвращается пара значений **<iterator, bool>**

```
std::set<double> c;
...
if (c.insert(3.67).second) {
    std::cout << "3.67 inserted" << std::endl;
}
else {
    std::cout << "3.67 already exists" << std::endl;
}
```

Специальные функции поиска

Функции поиска (такие же есть среди алгоритмов), оптимизированные для поиска в множестве

Функция	Назначение
`set.count(T value)`	Подсчет количества ключей со значением value
`set.find(T value)`	Поиск вхождения ключа со значением value
`set.lower_bound(T value)`	Поиск первого ключа, который больше или равен value
`set.upper_bound(T value)`	Поиск первого ключа, который больше value
`Set.equal_range(T value)`	Поиск диапазон с ключами, равными value

Специальные функции поиска

Функции поиска (такие же есть среди алгоритмов), оптимизированные для поиска в множестве

```
std::multiset<int> setTest;

setTest.insert(5); setTest.insert(4);
setTest.insert(3); setTest.insert(7);
setTest.insert(7); setTest.insert(7);
setTest.insert(8); setTest.insert(8);

std::cout << *setTest.lower_bound(5) << " ";
std::cout << *setTest.upper_bound(5) << " ";
auto bounds = setTest.equal_range(7);

std::cout << *bounds.first << " " << *bounds.second << "\n";

setTest.erase(bounds.first, bounds.second);

for (int elem : setTest) { std::cout << elem << " "; }
```

Создание мультимножества из массива и вывод

Перенос идеологии итераторов на обычные фиксированные массивы и вывод в поток с помощью итератора вывода.

```
const int N = 10;

int a[N] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};
int b[N] = {4, 4, 2, 4, 2, 4, 0, 1, 5, 5};

std::multiset<int> A(a, a + N);
std::multiset<int> B(b, b + N);

std::cout << "Set A: ";
std::copy(A.begin(), A.end(), std::ostream_iterator<int>(std::cout, " "));
std::cout << std::endl;
```

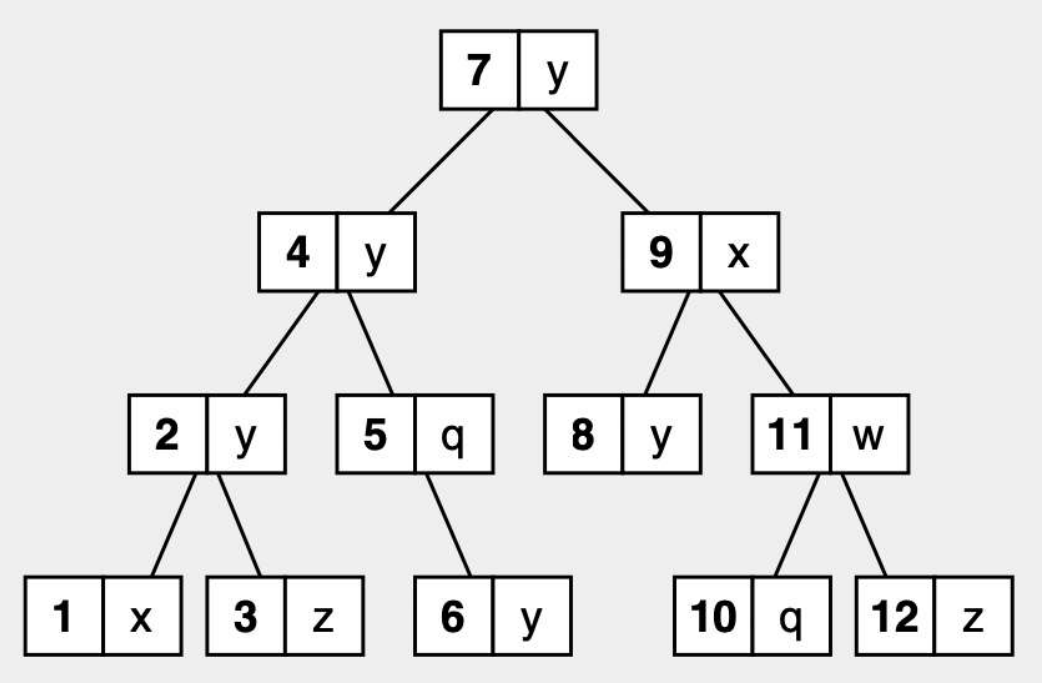
Пересечение, объединение и разность

Стандартные операции над множествами

```
const int N = 10;
int a[N] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};
int b[N] = {4, 4, 2, 4, 2, 4, 0, 1, 5, 5};

std::multiset<int> A(a, a + N); std::multiset<int> B(b, b + N);

std::set_union(A.begin(), A.end(), B.begin(), B.end(),
               std::ostream_iterator<int>(std::cout, " "));

std::set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                      std::ostream_iterator<int>(std::cout, " "));

std::multiset<int> C;
std::set_difference(A.begin(), A.end(), B.begin(), B.end(),
                    std::inserter(C, C.end()));
```

Отображения

Внутренняя организация отображений



Параметризуются типом ключей, значений и критерием сортировки.

- Изменять значение ключа напрямую нельзя.
- Можно изменять значения

Вставка в `std::map`

```
std::map<std::string, double> map;
```

- Список инициализаторов
  `map.insert({“testKey”, 459.78})`
- С помощью `value_type`
  `map.insert(std::map<std::string, double>::value_type(“testKey”, 459.78)`
- Передача пары
  `map.insert(std::pair<const std::string, double>("testKey", 459.78))`
- С помощью `make_pair`
  `map.insert(std::make_pair("testKey", 459.78))`

Вставка в std::map

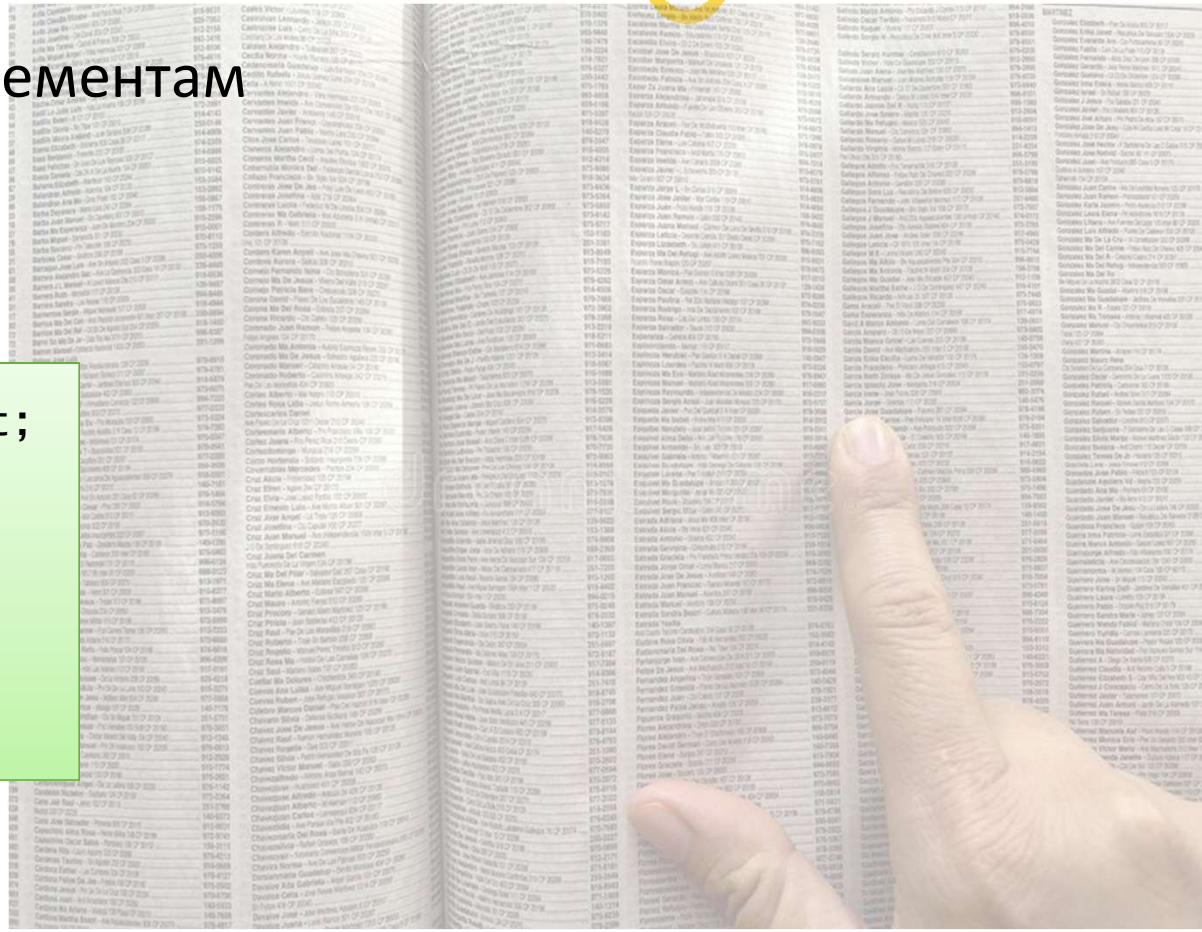
```
std::map<std::string, double> map;
```

- Список инициализаторов
  ```
  map.insert({“testKey”, 459.78})
  ```
- С помощью value_type
  ```
  map.insert(decltype(map)::value_type(“testKey”, 459.78)
  ```
- Передача пары (ключ, значение)
  ```
  map.insert(std::pair<const std::string, double>("testKey", 459.78))
  ```
- С помощью make_pair
  ```
  map.insert(std::make_pair("testKey", 459.78))
  ```

`std::map` как ассоциативный массив

Возможность **прямого доступа** к элементам отображения по их ключам

```
std::map<std::string,double> mapTest;

mapTest["key1"] = 1.45;
mapTest["key2"] = 1.56;
mapTest["key3"] = 3.49;

std::cout << mapTest["key4"];
```



Обработка std::map с помощью std::for_each

```
typedef std::map<std::string, int> MyMap;

struct print {
    void operator () (const MyMap::value_type &p) {
        std::cout << p.first << " " << p.second;
    }
};

MyMap myMap;

myMap["key1"]++; myMap["key2"]++; myMap["key1"]++;
myMap["test"]++; myMap["xxx"]++; myMap["zzz"]++;

std::for_each(myMap.begin(), myMap.end(), print());
```

Неупорядоченные контейнеры

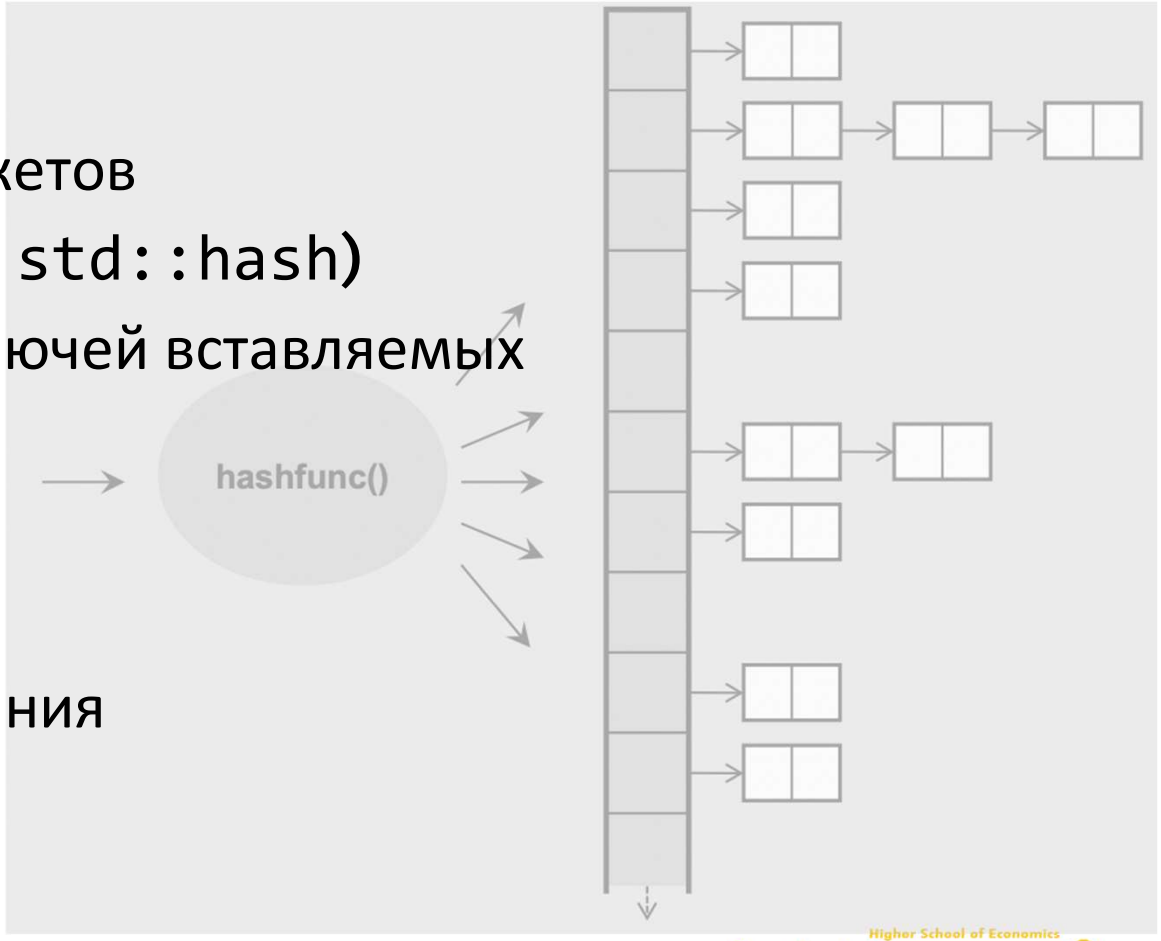
Хеш-таблицы

Ключ определяет **расположение** элемента в контейнере.

Контейнер	Реализация	Возможности
`unordered_set<T>` `unirdered_set<T, …>`	Хеш-таблица с **цепочками** для разрешения коллизий	• Хранение хешированных ключей • «Как минимум» однонаправленный итератор
`unordered_map<Key, Value>` `unirdered_map<Key, Value, …>`		• Хранение пар (ключ, значение), хешированных по ключу
`unordered_multiset<…>` `uniordered_multimap<…>`		

Параметры работы неупорядоченных контейнеров

- Минимальное количество бакетов
- Хеш-функция (по умолчанию, `std::hash`)
- Критерий эквивалентности ключей вставляемых объектов
- Максимальный коэффициент заполненности
- Форсирование перехеширования



Конфигурация неупорядоченных контейнеров

Функция	Назначение
`unordered_map.bucket_count()`	Текущее количество бакетов
`unordered_map.max_bucket_count()`	Максимальное количество доступных бакетов
`unordered_map.load_factor()`	Текущий коэффициент заполненности
`unordered_map.max_load_factor()`	Текущий максимальный коэффициент заполненности
`unordered_map.max_load_factor(v)`	Установка максимального коэффициента заполненности
`c.rehash(bnum)`	Перехеширование по желаемому количеству **бакетов**
`c.reserve(num)`	Перехеширование по желаемому количеству **записей** таблицы

Стратегии перехеширования

- **Полная перестройка** в результате выполнения вставки или удаления записи из хеш-таблицы
- **Постепенное** изменение количества бакетов, доступных для записи в хеш-таблицу

Стратегии перехеширования

- **Полная перестройка** в результате выполнения вставки или удаления записи из хеш-таблицы
- **Постепенное** изменение количества бакетов, доступных для записи в хеш-таблицу

Перехеширование происходит только в результате вставки, `rehash`, `reserve` или полного очищения таблицы.

Доступ к бакетам

Работа с внутренним содержимым хеш-таблицы

Функция	Назначение
`unordered_map.bucket(value)`	Индекса бакета, где содержится значение value
`unordered_map.begin(id)`	Однонаправленный итератор начала бакета с индексом id
`unordered_map.end(id)`	Однонаправленный итератор за концом бакета с индексом id
`unordered_map.cbegin(id)` `unordered_map.cend(id)`	Константные итераторы

Адаптеры

Что можно адаптировать?

Реализация одноименного паттерна:
конвертация одного интерфейса в другой

- Адаптеры контейнеров
  `stack`, **`queue`**, **`priority_queue`**
- Адаптеры итераторов
  `reverse_iterator`, **`back_inserter`**, **`front_inserter`**

Адаптер `std::priority_queue`

Вставка происходит, как в обычную очередь, а извлекаются элементы с максимальным приоритетом

```
struct Place {
    unsigned int dist;
    std::string dest;

    Place(std::string dt, size_t ds) {
        . . .
    }
    bool operator<(const Place &right) {
        return dist < right.dist;
    }
};
```

```
std::priority_queue<Place> pque;
pque.push(Place("Moscow", 1500));
pque.push(Place("Saint P", 1678));
pque.push(Place("Novgorod", 1479));

pque.pop();
pque.pop();
```

Адаптер `std::back_inserter`

Применяется к контейнерам, которые поддерживают метод вставка **push_back**

```
std::vector<int> v;
auto in_begin = std::istream_iterator<int>(std::cin);
auto in_end = std::istream_iterator<int>();
std::copy(in_begin, in_end, std::back_inserter(v));
```

Умный указатель

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 укзатели

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 object;
```

`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

Перемещение

Категории выражений

`l-value`

- имя переменной, функции, элемента данных
- вызов функции, перегруженного оператора
- преинкременет, предекремент
- оператор *

`r-value`

- литералы `122`, `true`, `nullptr`
- временные значения `a + 3`
- анонимные объекты `Object(…)`
- оператор &

Ссылка r-value &&

Ссылки на временные объекты

- Увеличение продолжительности жизни объекта до продолжительности жизни самой ссылки
- В случае неконстантных ссылок **r-value** появляется возможность менять значения временных объектов

```
class Object { . . . };
. . .;
Object &&rRef = Object(. . .);
std::cout << rRef;
```

```
int &&rRef = 14;
rRef = 25;

std::cout << rRef;
```

Ссылка r-value &&

Использование при перегрузке функций

```
void function(const int& param) {
    std::cout << “lvalue overload”;
}
```

```
void function(const int&& param) {
    std::cout << “rvalue overload”;
}
```

```
int main() {
    int variable = -19456;

    function(variable);
    function(-19456);
}
```

Копирование 1

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

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

    ~SmartPtr() {...}

    SmartPtr(SmartPtr& p) {
        ptr_ = new T;
        *ptr_ = *p.ptr_;
    }

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

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

    delete ptr_;
    ptr_ = new T;
    *ptr_ = *p.ptr_;

    return *this;
}
```

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

Копирование 2

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

```
SmartPtr<Test> create() {
    SmartPtr<Test> obj(new Test);

    return obj;
}
```

```
int main() {
    SmartPtr<Test> object;
    object = create();

    return 0;
}
```

Копирование 3

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

```
SmartPtr<Test> create() {
    SmartPtr<Test> obj(new Test);

    return obj;
}
```

```
int main() {
    SmartPtr<Test> object;
    object = create();

    return 0;
}
```

Сколько произойдет копирований?

Перемещение 1

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

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

    ~SmartPtr() {...}

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

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

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

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

    return *this;
}
```

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

Перемещение 2

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

```
SmartPtr<Test> create() {
    SmartPtr<Test> obj(new Test);

    return obj;
}
```

```
int main() {
    SmartPtr<Test> object;
    object = create();

    return 0;
}
```

Сколько произойдет копирований?

Семантика перемещения: в целом

- При конструировании объектов или присваивании с аргументом **`l-value`** выполняется **копирование**
- При конструировании объектов или присваивании с аргументом **`r-value`** выполняется **перемещение**

- Изменение поведения функций в зависимости от категории аргументов
- Перевод перемещенных объектов в определенное состояние (например, **`nullptr`**)

- Отключение копирования при наличии перемещения

Семантика перемещения и l-value

Обмен значений двух аргументов

```
template<class T>
void swap(T &x, T &y) {
  T tmp { x };
  x = y;
  y = tmp;
}
```

Семантика перемещения и l-value

Обмен значений двух аргументов

`std::move` конвертирует `l-value` в ссылку `r-value`

```
template<class T>
void swap(T &x, T &y) {
  T tmp { x };
  x = y;
  y = tmp;
}
```

```
template<class T>
void swap(T &x, T &y) {
  T tmp { std::move(x) };
  x = std::move(y);
  y = std::move(tmp);
}
```

Пару слов об идеальной передаче

Вставка элементов в контейнер

- Реализация методов вставки поддерживает семантику перемещения (принимают ссылки `r-value` **T&&**)
- Метод вставки `emplace(…)` не конструирует временные объекты, а передает (`forward`) полученные аргументы в конструктор объекта

Вставка объектов в `std::vector`

```
class MyClass {
    int i_; double d_;
public:
    MyClass() {. . .}

    MyClass(int x, double y) {
        std::cout << "constructed\n";
        . . .
    }

    MyClass(const MyClass& other) {
        std::cout << "copied\n";
        . . .
    }

    MyClass(MyClass&& other) {
        std::cout << "moved\n";
        . . .
    }
};
```

```
int main() {
    std::vector<MyClass> vectorTest;

    std::cout << "inserting via push_back(T&&)\n";
    vectorTest.push_back(MyClass(4, 5.67));
    std::cout << "inserting via emplace_back(T&&)\n";
    vectorTest.emplace_back(4, 5.67);

    return 0;
}
```

```
inserting via push_back(T&&)
constructed
moved
inserting via emplace_back(T&&)
constructed
copied
```

Вставка объектов в std::vector

```
class MyClass {
    int i_; double d_;
public:
    MyClass() {. . .}

    MyClass(int x, double y) {
        std::cout << "constructed\n";
        . . .
    }

    MyClass(const MyClass& other) {
        std::cout << "copied\n";
        . . .
    }

    MyClass(MyClass&& other) {
        std::cout << "moved\n";
        . . .
    }
};
```

```
int main() {
    std::vector<MyClass> vectorTest;
    vectorTest.reserve(15);

    std::cout << "inserting via push_back(T&&)\n";
    vectorTest.push_back(MyClass(4, 5.67));
    std::cout << "inserting via emplace_back(T&&)\n";
    vectorTest.emplace_back(4, 5.67);

    return 0;
}
```

```
inserting via push_back(T&&)
constructed
moved
inserting via emplace_back(T&&)
constructed
```

Во время собеседования:
- Алгоритм Дейкстры
- Развертывание односвязного списка
- Бинарные деревья, паттерны

Первый день на работе:
- Фон корзинке поменяй
- Кнопочке углы закругли
- Вместо ссылки иконку сделай