ППОИС

Часть 1

Структура стандартной библиотеки классов

Библиотека STL

Standard Template Library

Содержание

Назначение и альтернативы

Библиотека классов, необходимых для построение приложения общего назначения.

Альтернативы:

- boost
- клоны stl

Стандартная библиотека C++

1. Ввод-вывод
2. Многопоточность
3. Регулярные выражения
4. Библиотека C
5. Библиотека шаблонов STL
6. Прочее (дата и время, обработка ошибок и т.д.)

std::pair

Тип, позволяющий упаковать два значения в один объект

```
#include <utility>

auto p1 = std::pair<int, double>(1, 2.0);

auto p2 = std::make_pair(1, 2.0);

auto x = p1.first; // x == 1

auto y = p1.second; // y == 2.0
```

tuple

Тип, позволяющий упаковать несколько значений в один объект

```
#include <tuple>

auto t = std::make_tuple(1, 2.0, “abc”);

int a = std::get<0>(t);

double b = std::get<1>(t);

std::string c = std::get<2>(t);
```

tie

tie, как и make_tuple, создает tuple, но не объектов, а ссылок

```
class MyClass {
        int x_;
        std::string y_;
        double z_;
        bool operator<(const MyClass& o) const {
                return std::tie(x_, y_, z_) < std::tie(o.x_, o.y_, o.z_);
        }
};
```

Библиотека STL

1. Контейнеры (containers) - хранение набора объектов
2. Итераторы (iterators) - средства для доступа к итерируемому объекту
3. Алгоритмы (algorithms) - типовые операции с данными
4. Адаптеры (adaptors) - обеспечивают возможность использовать функции и методы, предназначенные для одного типа контейнера, с другими типами контейнеров
5. Функциональные объекты (functors) - объекты, которые могут быть вызваны как функции

Контейнеры

1. Последовательные (sequence)
2. Ассоциативные (associative)
3. Неупорядоченные ассоциативные (unordered associative)
4. Контейнеры-адаптеры (container adaptors)

std::array

Вставка - нет

Удаление - нет

Поиск - O(N)

Доступ - O(1)

```
std::array<int, 3> a = {1, 2, 3};

auto x = a[2];

a[2] = x * 2;
```

std::vector

Динамический массив, при добавлении элементов может изменять свой размер

Вставка - O(N)

Удаление - O(N)

Поиск - O(N)

Доступ - O(1)

Итераторы

Позволяют реализовывать универсальные алгоритмы

1. Ввода (input)
2. Однонаправленные (forward)
3. Двунаправленные (bidirectional)
4. Произвольного доступа (random access)
5. Вывода (output)

Операции

1. Input
    a. p++
    b. x = *p
2. Output
    a. p++
    b. *p = x
    c. p == q
3. Forward == Input + Output
4. Bidirectional = Forward + p- -
5. Random access = Bidirectional +
    a. p = p + n
    b. p = p - n

std::deque

Интерфейс повторяет интерфейс std::vector, отличие в хранении в памяти.

vector хранит данные в одном непрерывном массиве

deque хранит данные в связанных блоках по n элементов

[ ] [ ] [ ] [ ] [ ] [ ]

[ ] [ ] [ ] [ ] [ ] [ ]

std::forward_list

Связный список, элементы которого хранятся произвольно в памяти

Вставка - O(1)

Удаление - O(1)

Поиск - O(N)

Доступ - O(N)

std::list

Отличие от односвязного - возможность переходить в любом направлении

Ассоциативные контейнеры

Позволяют хранить пары вида ключ=значение и поддерживают операции добавления пары, а также удаления пары по ключу

1. Элементы отсортированы по ключу
2. Элементы не отсортированы по ключу

Элементы отсортированы по ключу

1. set<Key, Compare, Allocator>
2. map<Key, T, Compare, Allocator>
3. multiset<Key, Compare, Allocator>
4. multimap<Key, T, Compare, Allocator>

Вставка: O(logN)

Поиск: O(logN)

Удаление: O(logN)

Доступ:O(logN)

Элементы не отсортированы по ключу

1. unordered_set<Key, Hash, KeyEqual, Allocator>
2. unordered_map<Key, T, Hash, KeyEqual, Allocator>
3. unordered_multiset<Key, Hash, KeyEqual, Allocator>
4. unordered_multimap<Key, T, Hash, KeyEqual, Allocator>

Вставка: O(1) или O(N)

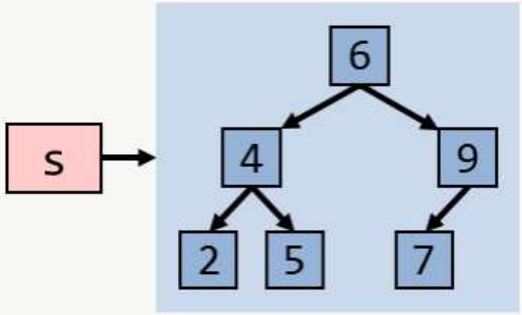
Поиск: O(1) или O(N)

Удаление: O(1) или O(N)

Доступ:O(1) или O(N)

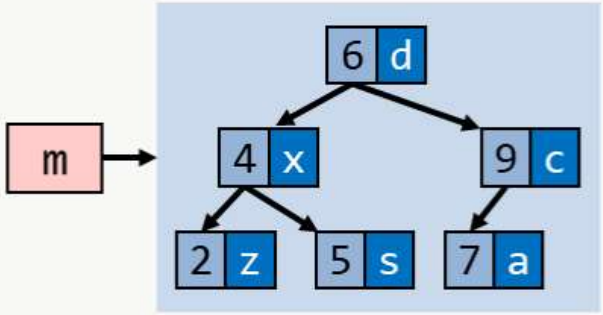
set<*Key*>

multiset<*K*>



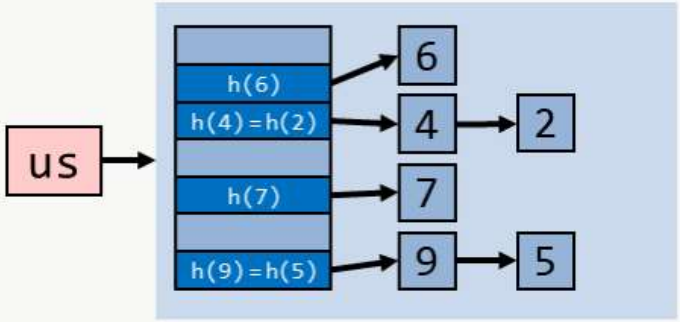
map<*Key*,*Value*>

multimap<*K*,*V*>



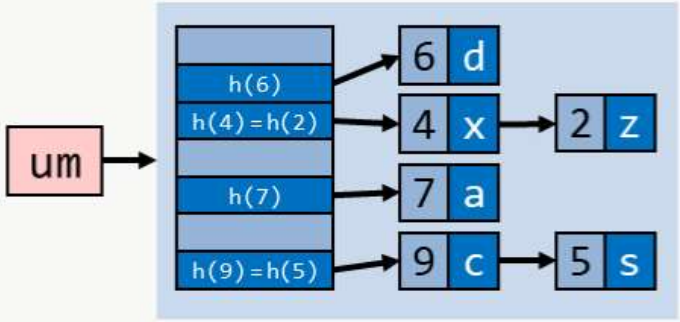
unordered_set<*Key*>

unordered_multiset<*Key*>



unordered_map<*Key*,*Value*>

unordered_multimap<*Key*,*Value*>



Контейнеры-адаптеры

Являются обертками над другими контейнерами

1. stack<T, Container = std::deque<T>> (LIFO)
2. queue<T, Container> (FIFO)
3. priority_queue<T, Container, Compare> - можно за O(1) извлечь элемент, удовлетворяющий условию

stack

```
#include <stack>

std::stack<int> s;

s.push(3);

s.push(5);

int x = s.top(); // 5

s.pop();

int y = s.top(); // 3
```

Библиотека алгоритмов STL

1. Не изменяющие последовательные алгоритмы
2. Изменяющие последовательные алгоритмы
3. Алгоритмы сортировки
4. Бинарные алгоритмы поиска
5. Алгоритмы слияния
6. Кучи
7. Операции отношений

adjacent_find

Возвращает итератор, указывающий на первую пару по условию. Если такой пары нет - то итератор - end.

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

auto it = std::adjacent_find(v.begin(), v.end());

    if (it != v.end()) {

        std::cout << "First occurrence of two consecutive elements: " << *it << std::endl;

    } else {

        std::cout << "No two consecutive equal elements found" << std::endl;

    }
```

all_of

Проверяет, что все элементы последовательности удовлетворяют предикату.

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

if (std::all_of(v.begin(), v.end(), [](int x) {return x < 5;}));

    std::cout << “all elements are < 5”;
```

any_of
none_of

equal

Проверяет, что две последовательности идентичны.

```
bool isPalindrome(const std::string& s)
{
	auto mid = s.begin() + s.size() / 2;
	return std::equal(s.begin(), mid, s.rbegin());
}
```

1. **find**
2. **find_if**
3. **find_if_not**
4. **find_end**
5. **find_first_of** - вхождение из второй последовательности в первой последовательности
6. **for_each** - вызов функции с каждым элементом последовательности

Модифицирующие

1. **copy**
2. **copy_backward**
3. **move**
4. **move_backward**
5. **fill**
6. **fill_n**
7. **generate**
8. **generate_n**
9. **remove**
10. **remove_if**
11. **shuffle**
12. **unique**

Алгоритмы сортировки

1. **is_sorted**
2. **sort**
3. **partial_sort**
4. **stable_sort** - сохраняет места для одинаковых объектов

Алгоритмы бинарного поиска

1. **lower_bound**
2. **upper_bound**
3. **equal_range**

Функторы

```
struct MultiplyBy {
    int factor;
    MultiplyBy(int factor) : factor(factor) {}
    int operator()(int x) const {
        return x * factor;
    }
};
```

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
MultiplyBy multiplyByFive(5);
```

```
    std::transform(numbers.begin(), numbers.end(), numbers.begin(), multiplyByFive);
```

Умные указатели

- std::unique_ptr
- std::shared_ptr
- std::weak_ptr

std::unique_ptr

Полностью владеет переданным ему объектом, а не делится «владением» еще с другими классами

std::make_unique() - упрощение создания

```
std::unique_ptr<Fraction> f1 = std::make_unique<Fraction>(7, 9);
std::cout << *f1 << '\n';

auto f2 = std::make_unique<Fraction[]>(5);
std::cout << f2[0] << '\n';
```

Пример возвращения из функции

```
std::unique_ptr<Item> createItem() {
    return std::make_unique<Item>();
}

int main() {
    std::unique_ptr<Item> ptr = createItem();
    // Делаем что-либо
    return 0;
}
```

Пример передачи в функцию

```
void takeOwnership(std::unique_ptr<Item> item) {
    if (item) {
        std::cout << *item;
    }
} // Item уничтожается здесь

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

    // takeOwnership(ptr); // это не скомпилируется. Мы должны использовать семантику перемещения
    takeOwnership(std::move(ptr)); // используем семантику перемещения

    std::cout << "Ending program\n";

    return 0;
}
```

std::move

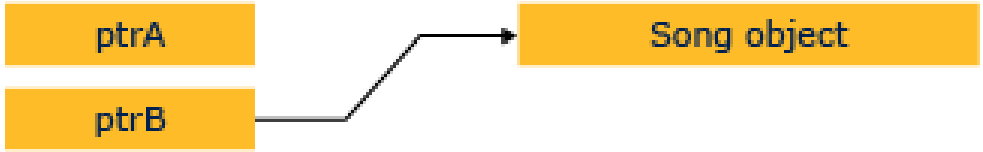
Конвертирует передаваемый аргумент в r-value

```
template<class T>
void swap(T& x, T& y)
{
  T tmp { std::move(x) }; // вызывает конструктор перемещения
  x = std::move(y); // вызывает оператор присваивания перемещением
  y = std::move(tmp); // вызывает оператор присваивания перемещением
}
```

Пример

```
auto ptrA = make_unique<Song>(L"Diana Krall", L"The Look of Love");
```


```
auto ptrB = std::move(ptrA);
```



Ошибки

```
Item* item = new Item;
std::unique_ptr<Item> item1(item);
std::unique_ptr<Item> item2(item);
```

```
Item* item = new Item;
std::unique_ptr<Item> item1(item);
delete item;
```

std::shared_ptr

- Умный указатель, основанный на подсчете ссылок
- Обеспечивает совместное владение объектом

`std::make_shared()`

```
{
    auto ptr1 = std::make_shared<Item>();
    {
        auto ptr2 = ptr1;
    }
}
```

Структура

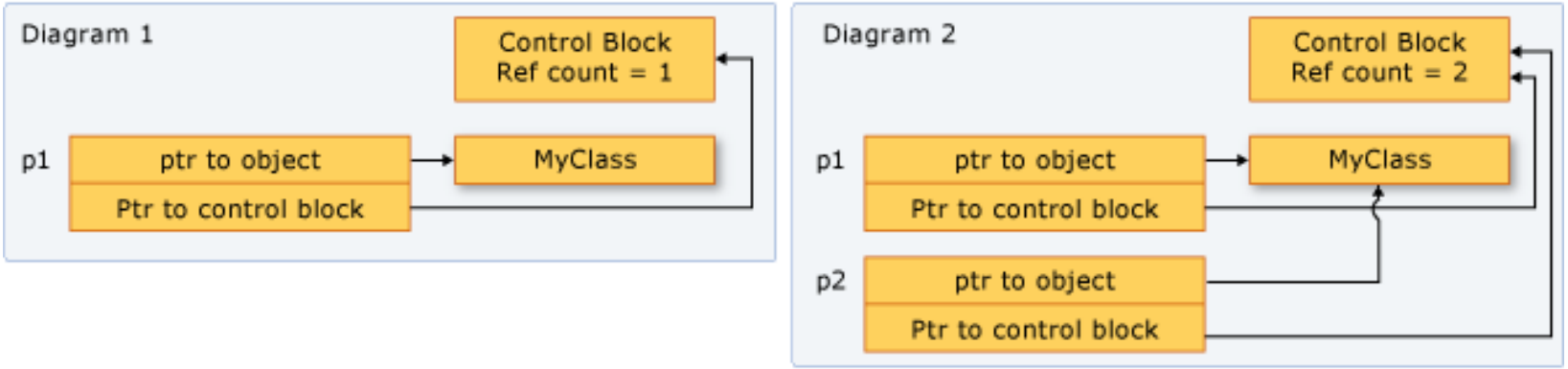


Diagram 1
Control Block
Ref count = 1
p1
ptr to object
Ptr to control block
MyClass
Diagram 2
Control Block
Ref count = 2
p1
ptr to object
Ptr to control block
MyClass
p2
ptr to object
Ptr to control block

Пример

```
{
    Item *item = new Item;
    std::shared_ptr<Item> ptr1(item);
    {
            std::shared_ptr<Item> ptr2(ptr1);
    }
}
```

```
{
        Item *item = new Item;
        std::shared_ptr<Item> ptr1(item);
        {
                std::shared_ptr<Item> ptr2(item);
        }
}
```

std::weak_ptr

- Слабая ссылка на объект
  - Не влияет на время жизни
  - Автоматически обнуляется после удаления объекта
- Пока объект жив, позволяет получить shared_ptr на него
- Решает проблему циклических ссылок и висячих указателей

- Владельцем не считается
- В подсчете ссылок не участвует
- lock() для получения shared_ptr