# Билеты к экзамену C++
# МКН, Современное программирование
# семестр II

Тамарин Вячеслав

June 18, 2020

# Contents

# Вопрос 1 Шаблоны

- решение в стиле C (`#define`)
- шаблонные классы
- шаблонные функции
- специализация шаблонов (частичные и полные; в т.ч. для функций)
- шаблонный параметр, не являющийся типом

## i Решение в стиле *C*

Пусть есть класс массива для целых чисел или умный указатель

```
class MyArray {
private:
    int *array;
};

class scoped_ptr {
private:
    GaussNumber *ptr;
}
```

Эти классы рассчитаны только для одного типа данных и для каждого типа придется вручную создавать новый тип.

Решить проблему можно с помощью `#define`. Классы для каждого нового типа будет генерировать препроцессор с помощью макросов.

MyArray.h

```
#define MyArray(TYPE) class MyArray_##TYPE {\
private: \
    TYPE *array; \
    size_t size; \
public: \
    TYPE get(size_t index) { \
        return array[index]; \
    } \
};
```

main.c

```
#include "MyArray.h"
MyArray(int);
MyArray(double);

int main() {
    MyArray_int a; // вместо MyArray(int) будет полный текст макроса
    MyArray_double b;
}
```

**Проблема:** Программист и компилятор видят разный исходный текст, разные сообщения од ошибках, препрепроцессор заменит любое подходящее слово на данный код.

## ii Шаблонные классы

MyArray.h

```
template <typename T>
class MyArray {
private:
    T *array;
    size_t size;
```

```
public:
    T& get(size_t index) {
        return array[i];
    }
};
```

Можно вынести определение методов за пределы объявления класса

MyArray.h

```
template<class T> // синоним template<typename T>
T& MyArray<T>::operator[] (size_t index) {
    return array[i];
}
```

main.cpp

```
#include "MyArray.h"
int main() {
    MyArray<int> a;
    MyArray<double> b;
    MyArray<MyArray<int>> c; // лучше не писать до c++11
}
```

**Особенности:**

1. Подстановку делает компилятор, а не препроцессор
2. Код шаблонного класса всегда в заголовочном файле
3. Иногда помещают в `MyArray_impl.h`
4. Увеличивается время компиляции
5. Методы шаблонного класса всегда `inline`

## iii Шаблонные функции

```
template <class T>
void swap(T &a, T &b) {
    T t(a);
    a = b;
    b = t;
}
int i = 10, j = 20;
swap<int>(i, j);
```

```
template <typename V>
void reverse(MyArray<V> &a) {
    V t;
    for (size_t i = 0; i < a.size()/2; ++i) {
        t = a.get(i);
        a.set(i, a.get(a.size() - i - 1);
        a.set(a.size() - i - 1, t);
    }
}
// Вызов
reverse<int>(a);
```

### Вывод шаблонных параметров

Компилятор может понять, какие аргументы у шаблона функции, если это однозначно определяется.

```
MyArray<int> a;
MyArray<double> b;
reverse(a);
reverse(b);
```

## iv Специализация

Идея: оптимизация для конкретного класса.

Общая версия

```
template<typename T>
class Array {
private:
    T *a;
public:
    Array (size_t size) {
        a = new T [size];
        ...
    }
};
```

Полная специализация

Для bool

```
template<>
class Array<bool> {
private:
    char *a;
public:
    Array (size_t size) {
        a = new char [(size-1)/8 + 2];
        ...
    }
};
```

Частичная специализация

Для массивов

```
template<class T>
class Array <Array<t>> {
    T **a;
};
```

## v Шаблонный параметр, не являющийся типом

```
template<size_t Size>
class Bitset {
private:
    char m[(Size-1)/8 + 1];
public:
    bool get(size_t index) { ... }
};

Bitset<128> b1;
```

# Вопрос 2 Исключения

- обработка ошибок в стиле C
- try/catch/throw
- исключения в конструкторах и деструкторах
- идиома RAII: использование и примеры классов
- гарантии исключений

Виды ошибок:

1. По вине программиста

Примеры

```
char *s = NULL;
size_t l = strlen(s);
Array a(-1);
```

Обработка этих ошибок

- Лучше выявить на стадии тестирования
- Если программа идеальна, не происходят
- Библиотека C такие ошибки не обрабатывает
- Библиотека C++ по-разному: `vector.at(i)`, `vector.operator[i]`
- Обрабатывать или нет — на усмотрение программиста

2. По вине окружения
   - Файл не существует
   - Сервер разорвал соединение
   - Вместо числа ввели букву

   Обработка ошибок
   - Могут происходить и при работе идеальной программы
   - Обязательно обрабатывать

## i Обработка ошибок в стиле C

Для обработки ошибок:

- Проверка на наличие ошибки в if
- Освобождение ресурсов

```
delete [] array;
close(f);
```

- Сообщить пользователю или вызывающей функции

```
FILE *f = fopen("a.txt", "r");
if (f == NULL) {
    printf("File a.txt not found\n");
}

if (f == NULL) {
    return -1;
}
```

- Предпринять действия по восстановлению (попробовать соединиться еще раз)

В стиле C информация об ошибке передается через возвращаемое значение и через глобальную переменную:

```c
FILE* fopen(...) {
    if (file not found) {
        errno = 666;
        return NULL;
    }
    if (permission denied) {
        errno = 777;
        return NULL;
    }
    ...
}
```

По возвращаемому значению не знаем причину ошибки, глобальная переменная хранит код ошибки, можно получить оттуда сообщение (`strerror(code)`) .

Не всегда хватает диапазона возвращаемых значений функции

```c
class Array {
    int *a;
public:
    // возвращает -1, когда индекс выходит за границу
    int get(size_t index);
};
int r1 = atoi("0");
int r2 = atoi("a"); // ?
```

Также код логики и обработка ошибок перемешаны

```c
r = fread(...);
if (r < ...) {
    // error
}
r = fseek(...);
if (r != 0) {
    // error
}
```

## ii try/catch/throw

Структура исключений

```c
class MyException {
private:
    char message[256];
    // filename, line, function name ...
public:
    const char* get();
};

double divide (int a, int b) {
    if (b == 0) {
        throw MyException("Division by zero");
    }
    return a/b;
}
```

Структура исключений

```c
try {
    x = divide(c, d);
}
catch (MyException& e) {
    std::cout << e.get(); // сообщаем пользователю
    // можем освободить ресурсы
    // throw e; проинформировать вызвавшую функцию
```

## iii Идиома RAII

Взятие ресурса должно «инкапсулировать» в класс, чтобы в случае исключения вызвался деструктор и освободил ресурс.

```
void f() {
    MyArray buffer(n);
    if ( ... ) throw MyException( ... );
}
```

```
void g() {
    autoptr p(new Person("Jenya", 36, true));
    divide(c, e); // может быть исключение
```

Если в divide произойдет исключение, объект еще не будет «достроен», поэтому деструктор не вызовется:

```
class PhoneBookItem {
    PhoneBookItem (const char *audio, const char *pic) {
        af = fopen(audio, "r");
        pf = fopen(pic, "r");
        divide(c, e); // исключение
        f();
    }
    ~PhoneBookItem() {
        fclose(af);
        fclose(pf);
    }
```

Поэтому нужно предусмотреть такую ситуацию и обернуть конструктор в try/catch:

```
class PhoneBookItem {
    PhoneBookItem (const char *audio, const char *pic) {
        try {
            af = fopen(audio, "r");
            pf = fopen(pic, "r");
            divide(c, e); // исключение
            f();
        }
        catch(MyException& e) {
            fclose(af);
            fclose(pf);
            throw e;
        }
    }
}
```

Исключения в деструкторе бросать нельзя, так как они могут подменить реальную причину ошибки. Если так происходит, программа аварийно завершается. В такой ситуации можно поступить так:

```
class PersonDatabase {
    ~PersonDatabase() {
        try {
            // брошена серверная ошибка
            networkLogger.log("Database is closed.");
        }
        catch (...) {} // поймать все
    }
};

f() {
    PersonDatabase db;
    if (...) throw MyException("Error: disk is full.")
}
```

### iv Гарантии

Гарантии:

1. обязательства функции (метода) с точки зрения работы с исключениями
2. документация для программиста, работающего с функцией (методом)

Виды гарантий:

**no throw guarantee** не бросает исключений вообще

```
void strlen(const char *s) {
    int count = 0;
    while (*s != 0) {
        s++; count++;
    }
    return count;
}
```

```
void f() {
    try {
        strlen(s);
        divide(a, b);
    }
    catch (...) { }
}
```

**basic guarantee** в случае возникновения исключения ресурсы не утекают

Если произойдет исключение, то память «течь» не будет, но измененные элементы array свои значения не восстановят:

```
class PersonDatabase {
    MyVector<Person> array;
    void process() {
        auto_ptr<Person> p(new Person());
        for (int i = 0; i < array.length; i++) {
            int a = divide(rand(), rand())
            // может быть исключение
            array[i]->setAge(a);
            std::cout << p;
        }
    }
};
```

**strong guarantee** переменные принимают те же значения, что были до возникновения ошибки

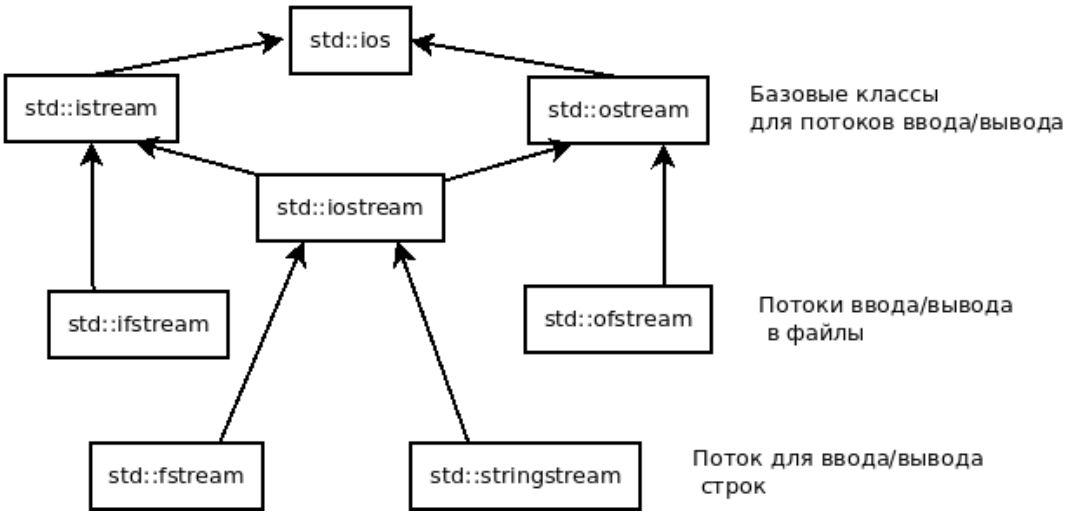
Идиома copy-and-swap

```
class PersonDatabase {
    MyVector<Person> array;
    void process() {
        auto_ptr<Person> p(new Person());
        MyVector<Person> copy(array);
        for (int i = 0; i < array.length; i++) {
            int a = divide(rand(), rand())
            // может быть исключение
            copy[i]->setAge(a);
        }
        array = copy;
    }
};
```

# Вопрос 3 Ввод-вывод в C++

- иерархия классов
- методы/флаги/манипуляторы
- обработка ошибок
- ввод-вывод пользовательских типов

## i Иерархия классов




Глобальные переенные `cin` (istream), `cout` (ostream), `cerr` (ostream).

Стандартная библиотека содержит перегруженные операторы `operator »`, `operator «` для примитивных типов и строк.

```
std::ostream& operator << (std::ostream& os, int v) {
    // convert int to bytes, write bytes
    return os;
}
```

```
std::istream& operator >> (std::istream& is, int& v) {
    // read bytes, convert to int
    return is;
}
```

Такой код приводит к очистке буфера fflush потока, что замедляет вывод:

```
std::cout << x << std::endl;
```

Оператор читает строку до пробела, getline до конца строки:

```
std::ifstream ifstream if("in.txt");
std::string word;
if >> word;
std::string line;
getline(if, line);
```

Побайтовый ввод/вывод: read, write, seekg, tellg.

### ii Обработка ошибок

rdstate() — чем закончиласть последняя операция: eofbit, goodbit, failbit (считываем другим типом), badbit (не существует файла)

### iii Формат вывода

```
int x = 255;
std::cout.setf(std::ios::hex, std::ios::basefield);
std::cout << x; // после вывода флаг очистится
```

Через манипуляторы

```
ostream& operator << (ostream& (*pf)(ostream&));
```

### iv Ввод-вывод пользовательских типов

```
class Point {
private:
    int x;
    int y;
public:
    friend ostream& operator << (ostream& os, const Point& p);
    friend istream& operator << (istream& is, Point& p);
    // friend функция может иметь доступ к приватным членам
};

ostream& operator << (ostream& os, const Point& p) {
    os << p.x << " " << p.y << "\n";
    return os;
}
istream& operator << (istream& is, Point& p) {
    is >> p.x >> p.y;
    return is;
}
```

Так как ostream — базовый класс, можно использовать один и тот же оператор для вывода на экран, строку и файл: (cout, ofstream of("file"), stringstream ss).

# Вопрос 4 Приведение типов

- C-style cast, static_cast, const_cast, reinterpret_cast - поведение и преимущества
- RTTI и dynamic_cast

## i Приведение типов в C

```
void f(char *p);
int *pi = malloc(100 * sizeof(int));
f(pi); // неявное приведение типов

int a = 65535;
char b = a; // неявное приведение типов

int c = 3.5; // тоже неявное

int a = 5; int b = 6;
double c = a / (double)b; // явное, действительно хотим
```

Неявное приведение может вызвать warning, если указать -Wall -Werror, станет ошибкой.

## ii Приведение типов в C++

Явное приведение требуется для указателей (кроме к void* и приведения у базовому классу).

```
void *f();
int* pi = (int*) f();

// class B : public class A
void print(const A* p);
B b;
print(& b);
```

Приведение для классов

```
Class BigInt {
    BigInt(int a);  // from int
    operator int(); // to int

    BigInt(const Complex&); // from Complex
    operator Complex();     // to Complex
```

Явное приведение упрощает поиск в коде и более точно выражает намерение программиста:

- Разные cast'ы для разных случаев
- Компилятор делает более точную проверку

explicit запрещает неявное приведение.

### static_cast

Примитивные типы; классы, связанные с наследованием; приведение к void*; пользовательские преобразования BigInt → int

```
int a = 65535;
char b = static_cast<char>(a);

// class B: public class A
void f(B *b);
A *a = new B();
f(static_cast<B*>(a));
```

### reinterpret_cast

Указатели разных типов

```
void* f();
int* pi = reinterpret_cast<int*>(f());

char *pc =  ...; int *pi = ...;
pc = reinterpret_cast<char*> pi;
```

### const_cast

Добавление и удаление const

```
char const *p1 = "Hello";
char *p2 = const_cast<char*>(p1);
p2[0] = 'h'; // undefined behavior
```

## iii Технология RTTI (Run-Time Type Information)

- Оператор dynamic_cast осуществляет безопасное преобразование указателя на базовый класс в указатель на производный (ссылки).
- Оператор typeid возвращает фактический тип объекта для указателя (ссылки).

### dynamic_cast

```
// class B: public class A;
// class C: public class A;
void f(B *b);

A *a = new C();
f(static_cast<B*>(a)); // при компиляции без ошибок, но undefined behavior во время работы

if (dynamic_cast<B*>(a) != 0) {
    f(static_cast<B*>(a));
}
```

Фактический тип

```
#include <typeinfo>
// class C: public class A
A *a = new C();
type_info ti = typeid(*a); // требуется ссылка
ti.name(); // "C"
```

- RTTI работает для классов с виртуальными функциями, информация о типе хранится в таблице виртуальных функций
- Чаще всего используется, когда нужно сделать костыль для существующего кода, который нельзя переделать.

```
class Shape {
    virtual draw() = 0;
};

draw_all(Shape* shapes, size_t n) {
    for (int i = 0; i < n; i++) {
        Shape *p = shapes[i];
        p->draw();
        if (dynamic_cast<AnimatedShape*>(p) != 0)
            p->animated_draw();
    }
}
```

# Вопрос 5 Последовательные контейнеры

- string, vector, list
- array
- внутреннее устройство и основные операции
- итераторы и их инвалидация

Требования к хранимым в контейнере объектам:

1. Корректно работает конструктор копий (copy-constructable)
2. Корректно работает оператор присваивания

Методы всех контейнеров:

1. Конструктор по умолчанию, конструктор копирования, оператор присваивания, деструктор
2. Операторы сравнения: `==, !=, >, >=, <, <=`
3. `size(), empty()`
4. `swap(obj2)`
5. `insert(), erase()`
6. `clear()`
7. `begin(), end()`

Особенности последовательных контейнеров:

1. Сохраняют порядок, в котором были добавлены элементы
2. Есть добавление в конец: `push_back`

## i string, vector, list

### vector

Динамический массив с автоматическим изменением размера при добавлении элементов.

```
std::vector<int> v;
```

1. Для уменьшения числа вызовов new при добавлении элементов выделяется с запасом
2. `size, capacity`
3. Сложность добавления в конец вектора и удаления из конца — $\mathcal{O}^*(1)$
4. Сложность добавления и удаления элемента из начала или середины — $\mathcal{O}(n)$
5. Сложность доступа к элементу по индексу — $\mathcal{O}(1)$

Методы:

1. `size()/resize()` — получение/изменение размера вектора
2. `capacity()/reserve()` — получение/изменение зарезервированной памяти
3. `push_back()/pop_back()` — добавление/удаление последнего элемента
4. `operator[]`, `at()` — получение элемента по индексу
5. `data()` — указатель на массив

```
#include <vector>

int main() {
        std::vector<int> v;
        v.push_back(10);
        v.pop_back();
```

```
        v[0] = 13;
        v.at(0) = 14;
        size_t size = v.size();
        bool is_empty = v.empty();
        v.clear();
        v.resize(10); // меняем размер на 10 элементов, работает, если есть конструктор по умолчанию
        v.resize(20, 5); // новые 10 элементов будут равны 5

        int *dst = new ...
        memcpy(dst, v.data(), v.size());

        v.reserve(100); // резервируем память на 100 элементов, размер  не меняется
        v.reserve(v.size() + 100);
        v.clear(); // изменит размер до 0, но вместимость не изменится
        vector<int>(v).swap(v); // уменьшить размерность вектора до реально используемых элементов
        return 0;
}
```

### list

Двусвязный список. Вставка и удаление в любом месте за $(O)(1)$. Нет обращения по индексу. Методы:

1. `size()`/`resize()` — получение/изменение размера вектора
2. `push_back()`/`pop_back()` — добавление/удаление последнего элемента
3. `push_front()`/`pop_front()` — добавление/удаление первого элемента
4. `merge()`/`splice()` — объединение/разделение

### string

Контейнер для хранения символьных последовательностей.

1. Метод `c_str()` для совместимости со старым кодом:

```
std::string res = "Hello";
printf("%s", res.c_str());
```

2. Алгоритмы `substr()`, `find()`, `...`
3. Поддержка преобразований типа с C-строками

```
f(const std::string& s);
f("Hello");
```

4. `append`, `operator+`, `operator+=`
5. `string = basic_string<char>`
6. `wstring = basic_string<wchar_t>` — для работы с длинными символами

## ii Итераторы

Объекты, которые синтаксически ведут себя как указатель (++, -, *, ->). Это универсальный способ перебора элементов контейнеров в STL. Итераторы реализованы как вложенные классы для контейнеров.

```
class vector {
        ...
        class iterator {
            operator ++() { T* ptr++; }
        }
```

```
}
vector::iterator it1;

class list {
    class iterator {
                operator ++() { Node<T> ptr = ptr->next; }
        }
}
list::iterator it2;
```

Все итераторы имеют функции, которые возвращают итераторы на первый и следующий за последним элемент.

Для vector и deque реализована арифметика, как для указателй:

```
int i = *(v.begin() + 5);
```

Проход по контейнеру:

```
list<int> l;
list<int>::iterator it = l.begin();
for (; it != l.end(); ++it) cout << *it;
```

Итератор, не позволяющий менять данные, на которые он указывает:

```
list<int>::const_iterator cit = l.begin();
```

Удаление элемента, на который указываем

```
it = v.erase(it);
```

Вставка элемента на данную позицию

```
it = v.insert(it, 5);
```

### Инвалидация итераторов

После того, как произвели операцию с контейнером, итератор может указывать в неверное место.

```
std::vector<int> v;
it = v.erase(it); // возвращаем следующий элемент
std::vector<int>::iterator itb = v.begin();
v.push_back(3);
v.push_back(5);
```

Поставить перед каждым четным элементом вектора 0:

```
std::vector<int> v;
for (std::vector<int>:: iterator i = v.begin(); i != v.end(); ++i) {
    if (*i % 2 == 0) {
            i = v.insert(i, 0);
                ++i;
```

# Вопрос 6 Ассоциативные контейнеры

- set, multiset, map, multimap
- unordered_set, unordered_map
- внутреннее устройство и основные операции
- итераторы и их инвалидация

Особенности:

1. Переупорядочивают элементы для быстрого поискка $(O)(\log n)$
2. Возможные реализации: дерево поиска $\mathcal{O}(\log n)$
3. Требуют отношение порядка `operator<()`
4. Нет произвольного доступа по индексу

Общие методы:

1. Конструктор по умолчанию, конструктор копирования, оператор присваивания, деструктор
2. Операторы сравнения: `==, !=, >, >=, <, <=`
3. `size(), empty()`
4. `swap(obj2)`
5. `clear()`
6. `begin(), end()`
7. `erase` по ключу
8. `insert`/`insert` с подсказкой (итератор)
9. `count` число элементов с заданным ключом
10. `find` поиск точного совпадения (возвращает итератор)
11. `lower_bound, upper_bound, equal_range`

## i set, multiset

В set все элементы уникальны, в multiset допустимы дубликаты. Реализованы с помощью BST с ключами в вершине.

```
#include <set>

std::set<int> s;
s.insert(10); s.insert(20); s.insert(10); // s.size() = 2

std::multiset<int> ms;
ms.insert(10); ms.insert(20); ms.insert(20); // ms.size() = 3

std::cout << s.count(20) << " " << ms.count(20) << std::endl;
```

Порядок не сохраняется.

```
std::set<int> s;
s.insert(10);
s.insert(6);
s.insert(20);
s.insert(5);
s.insert(1);

for (std::set<int>::iterator it = s.begin(); it != s.end(); ++i)
        std::cout << *it << " ";
} // выведет 1 5 6 10 20
```

Присваивание запрещено.

```
std::set<int> s;
s.insert(10);
std::set<int>::iterator it = s.find(10);
*it = 5; // запрещено
```

## ii map, multimap

В map элементы уникальны, в multimap — нет. Реализация с помощью BST, в вершине $(key, value)$
Особый метод: `operator[]`.

### Вспомогательный класс pair

```
template<class F, class S>
struct pair {
        F first;
        S second;
        ... // конструкторы
};

template<class F, class S>
pair<F, S> make_pair(F const& f, S const& s);

template<class Key, class T, ...> class map {
        ...
        typedef pair<const Key, T> value_type;
};
```

```
#include <map>

std::map<string, int> phonebook;
phonebook.insert(std::pair<string, int> ("Mary", 2128506));
phonebook.insert(std::make_pair("Alex", 9286385); // вывод типов параметров
phonebook.insert(std::make_pair("Bob", 2128506);

std::map<std::string, int>::iterator it = phonebook.find("John");
if (it != phonebook.end())
    std::cout << "Jonh`s p/n is " << it->second << std::endl;

for (it = phonebook.begin(); it != phonebook.end(); ++it)
    std::cout << it->first << ": " << it->second << "\n";
```

`operator[]` Работает только с неконстантными map. Требует наличие конструктора по умолчанию у T.

```
T & operator[](Key const& k) {
    iterator i = find(k);
        if (i == end())
            i = insert(k, T());
    return i->second;
}
```

Требует $\mathcal{O}(\log n)$ времени.

Пример

```
m["vasya"] = 10;
m["petya"] = 20;
int p = m["kola"]; // если такого ключа нет, будет создан новый элемент со значением 0
```

### iii unordered_map, unordered_set

```
template<class Key, class Hash = std::hash<Key>,
                 class KeyEqual = std::equal_to<Key>,
                 class Allocator = std::allocator<Key>>
        class unordered_set;
```

Реализовано как хэштаблица из каманов и списков. Операции find знимают $(O)(1)$, но, если все оказались в одном кармане, то $\mathcal{O}(n)$, insert аналогично, если вызывает рехэширование).

Для своего типа нужно реализовать hash:

```
class Point {
private:
    int x, y;
private:
    bool operator == (const Point& rhs) const {
                return x == rhs.x && y == rhs.y;
        }
};

namespace std {
    template<>
        struct hash<Point> {
            size_t operator() (Point const &p) const {
                    return (std::hash<int>()(p.getX()) * 53 + std::hash<int>()(p.getY())) * 239;
                }
        };
}

std::unordered_set<Point> points;
```

### iv Инвалидация

```
std::map<string, int> m;
std::map<string, int>::iterator it = m.begin();
for (; it != m.end;)
    if (it->second == 0)
            m.erase(++it);
        else
            ++it;
```

В C++11

```
for (; it != m.end(); )
    if (it->second == 0)
            it = m.erase(it);
    else
            ++it;
```

### v Собственный компаратор

```
struct Person {
    string name;
        string surname;
};
bool operator < (Person const& a, Person const& b) {
    return a.name < b.name || (a.name == b.name && a.surname < b.surname);
```

```
}
std::set<Person> s1; // уникальные по name и surname

struct PersonComp {
        bool operator () (Person const& a, Person const& b) const {
            return a.surname < b.surname;
        }
};
std::set<Person, PersonComp> s2; // только по surname
```

# Вопрос 7 Алгоритмы

- функторы
- обзор алгоритмов с примерами (swap, iter_swap, sort, find, copy, unique, remove_if, lower_bound)
- erase-remove idiom
- реализация алгоритмов через итераторы

## i Функторы

Функторы — классы, объекты котороых похожи на функцию и перегружен оператор (). Используется, когда нужно описать алгоритму, что нужно сделать, при этом можтет иметь поля.

```
struct sum_sq {
    int operator() (int a, int b) const {
            return a * a + b * b;
        }
};
sum_sq f;
int res = f(3, 4);
```

сравниваем с данным числом

```
struct cmp {
    int value;
        cmp(int v): value (v) {}

        bool operator () (int a) const {
            return a < value;
        }
};

cmp f(2);
bool b = f(13);
```

Такой функтор (с bool) называют предикатом.

Подсчет суммы

```
struct accum {
    int acc;
        accum() : acc(0) {}

        void operator () (int a) {
            acc += a;
        }
};

accum f;
f(13); f(16);
```

На функциях можно сделать аналогично черес статические переменные.

## ii Алгоритмы

### Микро-алгоритмы

- `swap(T &a, T&b)`
- `iter_swap(It p, It q)`
  меняет местами значения, на которые указывают итераторы
- `max(const T &a, const T &b)`
- `min(const T &a, const T &b)`

- `size_t count(It p, It q, const T& x)`
- `size_t count_if(It p, It q, Pr pred)`

пример использования предикаторов

```
struct Person {
    int age;
        string name;
        string city;
        Person (...) {...}
};
struct by_city {
    bool operator () (const Person &p1, const Person &p2) const {
                return p1.city < p2.city;
        }
};

Person p1(30, "V", "Msc"); Person p2(15, "K", "Spb");
by_city f;
std::max<Person, by_city> (p1, p2, f);
std::max(p1, p2, by_city()); // анонимная переменная
```

### Алгоритмы типа find

- `find(It p, It q, const T &x)`
  возвращает итератор на первое вхождение элемента x в последовательнось между p и q
- `find_if(It p, It q, Pr pred)`
  возвращает итератор на первое вхождение элемента, для которого pred вернул true
- `max_element(It p, It q)`
- `min_element(It p, It q)`
- `equal(It p, It q, Itr i)`
  Сравнивает две последовательности, первая p-q, вторя должна быть той же длины. Если короче, то undefined behaviour.
- `pair<It, Itr>mismach(It p, It q, Itr i)`
  Возвращает пару итераторов, указывающую на первое несовпадение.
- `F for_each(It p, It q, F func)`
  Применяет func для всех элементов, возвращает функтор.

```
std::vector<int> v = {1, 2, 5, 3, 4, 5, 5, 2};
std::vector<int>::iterator two = find(v.begin(), v.end(), 2);
```

### Модифицирующие алгоритмы

- `fill(It p, It q, const T &x)`
- `generate(It p, It q, F gen)`
  Заполняет последовательность значениями, сгенерированными функтором gen
- `copy(It p, It q, Itr out)`
- `reverse(It p, It q)`
- `sort(It p, It q)`
- `transform(It p, It q, Itr out, F func)`
  К каждому элементу применяем функтор func и записывают в последовательность, начинающуюся с итератора out

```
std::vector<int> v = {5, 2, 8, 9, 4, 1};
std::sort(v.begin(), v.end());

```

```
std::vector<int> v1 = {5, 3, 3, 4, 1, 5, 5, 4};
std::vector<int>::iterator last = std::unique(v1.begin(), v1.end());
v1.erase(last, v1.end());
```

### erase-remove idiom

Сначала поставим все нужные элементы в начало, а в конце оставим мусор, вернем итератор на начало мусора. Далее можно его удалить через erase.

```
std::vector<int> v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
v.erase(std::remove(std::begin(v), std::end(v), 5), std::end(v));
```

### Реализация алгоритмов через итераторы

Это удобно тем, что после одной реализации можно использовать для различных типов и структур данных, имеющих итераторы.

```
template<class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last, OutputIt d_first) {
    while (first != last)
            *d_first++ = *first++;
    return d_first;
}

vector<int> v;
list<int> l;
copy(v.begin(), v.end(), l.begin());

template<class InputIt, class OutputIt, class UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last, OutputIt d_first, UnaryPredicate pred) {
    while (first != last) {
            if (pred(*first))
                        *d_first++ = *first;
                first++;
        }
    return d_first;
}
copy_if(v.begin(), v.end(), l.begin(), divide_by(8));
```

## Вопрос 8 Реализация итераторов и алгоритмов

- реализация собственного итератора
- `value_type, iterator_category`
- `std::advance, std::distance` (зачем и реализация)

Для того, чтобы реализовать алгоритм, нужно будет получить тип элемента, на который указывает итератор. Для этого у итератора нужно добавить поле, хранящее этот тип.

```
template<class T>
class vector {
    T *array;
        class iterator {
            typedef T value_type;
        };
};
```

```
template<class It>
void q_sort(It p, It q) {
    It::value_type pivot = *p;
}
```

Также есть `iterator::pointer`, `iterator::reference`, `iterator::iterator_category`.

Если мы знаем, какие операции поддерживает итератор, можно написать более эффективный код. Возможные категории:

**Random access iterator (RA)** Поддерживает ++, --, +=. Используется в `std::vector, std::deque, std::array` (последний C++11).

**Bidirectioinal iterator (BiDi)** Поддерживает только ++, --. Это более слабый итератор. Используется в остальных контейнерах.

**Forward iterator (Fwd)** Поддерживает только ++. Можно использовать в вводе/выводе.

RA

```
template <class T>
class vector {
    T *array;
        class iterator {
            typedef ra_tag iterator_category; // пустая структура для хранения типа
        };
};
```

BiDi

```
template <class T>
class list {
        Node *head;
        class iterator {
            typedef bidi_tag iterator_category; // пустая структура для хранения типа
        };
};
```

Теперь нужно использовать знание о том, что итератор поддерживает больше операций. Для этого используются следующие функции:

`std::advance(it, n)` Продвигает итератор на $n$ позиций вперед. Для RA использует +=, для BiDi ++.

Реализация

```
template <class Iterator>
Iterator advance (Iterator it, int amount) {
    typedef iterator::iterator_category tag;
        advance(it, amount, tag());
}

template <class RAIterator>
RAIterator advance (RAIterator it, int amount, ra_tag t) {
    return it + amount;
}

template <class BiDiIterator>
BiDiIterator advance (BiDiIterator it, int amount, bidi_tag t) {
    for (; amount; --amount;) ++it;
        return it;
}
```

`std::distance(it1, it2)` Возвращает расстояние между итераторами.

```
template<class It>
typename std::iterator_traits<It>::difference_type
    distance(It first, It last) {
    return detail::do_distance(first, last,
                               typename std::iterator_traits<It>::iterator_category());
}

template<class It>
typename std::iterator_traits<It>::difference_type
    do_distance(It first, It last, std::input_iterator_tag) {
    std::iterator_traits<It>::difference_type result = 0;
    while (first != last) {
        ++first;
        ++result;
    }
    return result;
}

template<class It>
typename std::iterator_traits<It>::difference_type
    do_distance(It first, It last, std::random_access_iterator_tag) {
    return last - first;
}
```

# Вопрос 9 C++11. Разное.

- `lambda` и захваты
- `auto, decltype`
- `initializer_list`
- `shared_ptr, unique_ptr` (использование)

## i Вывод типов

```
auto x = expression; // тип x выводится по expression
decltype(expression) y; // тип x будет такой же, как у expression
```

Для чего это нужно? Пусть мы хотим пройти по вектору строк константным итератором в обратном порядке.

```
for (std::vector<std::string>::const_iterator i = v.cbegin(); i != v.cend(); ++i) {
    std::vector<std::string>::const_reverse_iterator j = i;
        ...
}
```

Получаем очень очень длинный и тяжелочитабельный код. На C++11 можно переписать короче

```
for (auto i = v.cbegin(); i != v.cend(); ++i) { // v.cbegin() - константный итератор
    decltype(v.crbegin()) j = i;
        ...
}
```

Еще отличия auto и dectype:

```
const std::vector<int> v(1);
auto a = v[0]; // int, так как нужен тип, которому можно присвоить значение
decltype(v[0]) b = 1 // const int& (как и у v[0])
```

decltype в typedef

```
typedef decltype(v[0]) new_type;
```

Уточнение auto

```
string s = "hello";
auto& s1 = s; // s1 - string&;
const auto& c = v[0]; // c - const int&
```

Но есть побочный эффект:

```
std::map <KeyClass, ValueClass> m;
auto I = m.find(something); // все знают, что find вернет iterator

MyClass myObj;
auto ret = myObj.findRecord(something);
// большинству придется сходить посмотреть, что возвращает findRecord
```

## ii lambda

```
std::vector<int> v = {50, -10, 20, -30};
// до C++ нужен компаратор
struct abs_cmpr {
    bool operator () (int a, int b) const {
            return abs(a) < abs(b);
```

```
        }
};
std::sort(v.begin(), v.end(), abs_cmpr());

// C++11 есть lambda функции для упрощения таких задач
std::sort(v.begin(), v.end(), [](int a, int b) { return abs(a) < abs(b); });
```

### Структура lambda функций

```
[список захвата](параметры){сама функция}
```

Захвачены могут быть локальные переменные из области видимости. Типы захвата:

1. ничего `[]`
2. все по ссылке `[&]`
3. все по значению `[=]`
4. смешанные `[x,&y]`, `[&,x]`, `[=,&z]`

**begin, end** Сделаны для реализации алгоритмов как для итераторов, так и для массивов.

```
template<class T>
void foo(T& v) {
    auto i = begin(v);
        auto e = end(v);
        for (; i != e; i++) {}
}

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

// параметр N выводится компилятором
int a[4] = {0, 1, 2, 3};
auto e = end(a);
```

С помощью этого реализован `for(x : )`

```
for (auto x: v) cout << x << " ";
for (auto& x: v) x++;
```

## iii initializer_list

Инициализация по списку элементов для своих объектов.

```
std::vector<std::string> v = {"AA", "BB", "CC"};
std::vector<std::string> v ({"AA", "BB", "CC"});
std::vector<std::string> v {"AA", "BB", "CC"};
```

По такому коду генерируется initializer_list

```
template <typename T>
class vector {
public:
        vector (std::initialaizer_list<T> l) {
            const T *array = l.begin();
                T value = *(array++);
                ...
                l.size();
        }
};
```

### iv unique_ptr, shared_ptr

Первое используется, если нужна ровно одна ссылка, второе же требует больше ресурсов и времени из-за постоянных вычислений.

```
struct Foo {
    ...
        void bar() {std::cout << "Foo::bar\n";}
};

void f(const Foo&) {std::cout << "f(const Foo&)\n";}

int main() {
        std::unique_ptr<Foo> p1(new Foo);
        if (p1) p1->bar();

        // теперь только p2 будет владеть Foo
        std::unique_ptr<Foo> p2(std::move(p1));
        f(*p2);

        p1 = std::move(p2); // можем вернуть p1
        if (p1) p1->bar();
}
```

# Вопрос 10 move семантика

- rvalue и lvalue
- rvalue references
- move constructor, move assignment
- std::move

## i lvalue и rvalue

**lvalue** Может быть и в левой, и в правой части присваивания (переменная).

- продолжает существовать за пределами выражения, где было использовано
- имеет имя
- можно взять адрес

**rvalue** Выражение, которое может быть только в правой части присваивания.

- не существует за пределами выражения, где было использовано
- временное значение (temporary value)

### Примеры

```
int a = 42;
int b = 43;
int c = a * b; // a * b - rvalue
```

```
vector<Person> people;
people.push_back(Person("Name", 36); // Person("Name", 36) - rvalue
```

```
int square(int x) { return x*x; }
int sq square(10); // square(10) - rvalue
```

## ii move

Используется, если внутри класса храним какие-то ресурсы: указатель на динамически выделенную память, файлы, ... Оптимизируем работу с временными выражениями rvalue. Пусть есть класс, обладающий описанными свойствами.

```
class X {
    int *array;
        X(size_t size);
        ~X();
        X(const X&);
        X& operator = (const X&);
};

X foo();
X x;
x = foo();
```

Что происходит во время присваивания?

1. копируем ресурс из temporary (rvalue)
2. освобождаем ресурс в x (delete, fclose, ...)
3. присваиваем в x скопированный ресурс
4. освобождаем ресурс в temporary

Можем обработать эффективно.

```
class X {
    int *array = NULL;
        X(size_t size);
        ~X();
        X(const X& x); // X& - lvalue reference
```

```
        X& operator = (const X& x);

        X(X&& x) { std::swap(this->array, x.array); } // X&& - rvalue reference, это move constructor
        X& operator = (X&& x) { this->array = std::move(x.array) } // это move assignment
};

X x1(100);
X x2(x1); // вызываем X(const X& x), так как x1 - lvalue
X x3(X(100)); // вызываем X(X&& x), так как X(100) - rvalue
```

### iii std::move

Говорим, что мы хотим для этой переменной использовать move конструктор:

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

# Вопрос 11 Метапрограммирование - I

- решение в стиле C (`#define, #ifdef`)
- `constexpr, static_assert`
- SFINAE, реализация предиката для типа (`has_iterator, is_integral`)

## i решение в стиле C

Можно написать условие через `#ifdef`, там указать `#error`, если условие выполнится, и мы попадем на `#error`, тогда возникнет ошибка предпроцессора.

## ii constexpr

Это модификатор функции, говорящий о том, что для явного аргумента можно вычислить во время компиляции и заменить на результат. На такую функцию накладывается несколько ограничений: (только C++11) однострочная, не должна вызывать функцию не constexpr, нет try/catch, не virtual, ...

```
constexpr unsigned fibonacci (unsigned i) {
        return (i <= 1u) ? i : (fibonacci(i-1) + fibonacci(i-2));
}
int array[fibonacci(3)];
```

Вариант без constexpr - вычисление на этапе компиляции

```
template<int N>
struct Factorial { // переход
    static int value N * Factorial<N-1>::value;
};

template<>
struct Factorial<0> { // база
    static int value = 1;
};

std::cout << Factorial<5>::value << std::endl;
```

## iii static_assert

Осуществляет проверку условия во время компиляции:

```
static_assert(sizeof(unsigned int) * 8 == 32, "16 bit CPU is not supported");
```

## iv template alias

Аналог typedef для шаблонов. Для этого есть новое слово using.

```
template<typename T>
using fast_vector = std::vector<T, big_pool_allocator<T>>;
fast_vector<int> v;
```

## v SFINAE

Используется, чтобы понять есть ли у класса метод с определенными параметрами во время компиляции.

```cpp
class Foo {
public:
    void do_smth(int);
        Foo() = delete;
};

class Bar {
};

template<typename T>
class Has_func {
public:
        template<typename U>
        static decltype(std::declval<U>().do_smth(1000)) detect(const U&);
        // declval нужен, чтобы при отсутствии конструктора по умолчанию у Foo, все компилировалось
        static float detect(...); // принимает любой аргумент

        static constexpr bool flag = std::is_same<void, decltype(detect(std::declval<T>()))>::value;
};

// std::is_same<void, int>::value // сравниваем типы на равенство

int main() {
    if (Has_func<Foo>::flag)
            std::cout << "method exists" << std::endl;
        else
            std::cout << "method does not exist" << std::endl;
        return 0;
}
```