

Билеты к экзамену С++  
МКН, Современное программирование  
семестр II

Тамарин Вячеслав

June 18, 2020

# Contents

Вопрос 1	Шаблоны . . . . .	2
i	Решение в стиле <i>C</i> . . . . .	2
ii	Шаблонные классы . . . . .	2
iii	Шаблонные функции . . . . .	3
iv	Специализация . . . . .	4
v	Шаблонный параметр, не являющийся типом . . . . .	4
Вопрос 2	Исключения . . . . .	5
i	Обработка ошибок в стиле <i>C</i> . . . . .	5
ii	try/catch/throw . . . . .	6
iii	Идиома RAII . . . . .	6
iv	Гарантии . . . . .	7
Вопрос 3	Ввод-вывод в <i>C++</i> . . . . .	9
i	Иерархия классов . . . . .	9
ii	Обработка ошибок . . . . .	10
iii	Формат вывода . . . . .	10
iv	Ввод-вывод пользовательских типов . . . . .	10
Вопрос 4	Приведение типов . . . . .	11
i	Приведение типов в <i>C</i> . . . . .	11
ii	Приведение типов в <i>C++</i> . . . . .	11
iii	Технология RTTI (Run-Time Type Information) . . . . .	12
Вопрос 5	Последовательные контейнеры . . . . .	13
i	string, vector, list . . . . .	13
ii	Итераторы . . . . .	14
Вопрос 6	Ассоциативные контейнеры . . . . .	16
i	set, multiset . . . . .	16
ii	map, multimap . . . . .	17
iii	unordered_map, unordered_set . . . . .	18
iv	Инваляция . . . . .	18
v	Собственный компаратор . . . . .	18
Вопрос 7	Алгоритмы . . . . .	20
i	Функторы . . . . .	20
ii	Алгоритмы . . . . .	20
Вопрос 8	Реализация итераторов и алгоритмов . . . . .	23
Вопрос 9	<i>C++11</i> . Разное. . . . .	25
i	Вывод типов . . . . .	25
ii	lambda . . . . .	25
iii	initializer_list . . . . .	26
iv	unique_ptr, shared_ptr . . . . .	27
Вопрос 10	move семантика . . . . .	28
i	lvalue и rvalue . . . . .	28
ii	move . . . . .	28

iii	std::move . . . . .	29
-----	---------------------	----

## 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

```
1 class MyArray {
2 private:
3     int *array;
4 };
5
6 class scoped_ptr {
7 private:
8     GaussNumber *ptr;
9 }
```

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
1 #define MyArray(TYPE) class MyArray_#TYPE {\
2 private: \
3     TYPE *array; \
4     size_t size; \
5 public: \
6     TYPE get(size_t index) { \
7         return array[index]; \
8     } \
9 };
```

```
main.c
1 #include "MyArray.h"
2 MyArray(int);
3 MyArray(double);
4
5 int main() {
6     MyArray_int a; // вместо MyArray(int) будет полный текст макроса
7     MyArray_double b;
8 }
```

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

### ii Шаблонные классы

```
MyArray.h
1 template <typename T>
2 class MyArray {
3 private:
4     T *array;
5     size_t size;
```

```

6 public:
7     T& get(size_t index) {
8         return array[i];
9     }
10 };

```

Можно вынести определение методов за пределы объявления класса

```

1 template<class T> // синоним template<typename T>
2 T& MyArray<T>::operator[] (size_t index) {
3     return array[i];
4 }

```

```

1 #include "MyArray.h"
2 int main() {
3     MyArray<int> a;
4     MyArray<double> b;
5     MyArray<MyArray<int>> c; // лучше не писать до c++11
6 }

```

### 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

```

1 template <class T>
2 void swap(T &a, T &b) {
3     T t(a);
4     a = b;
5     b = t;
6 }
7 int i = 10, j = 20;
8 swap<int>(i, j);

```

```

1 template <typename V>
2 void reverse(MyArray<V> &a) {
3     V t;
4     for (size_t i = 0; i < a.size()/2; ++i) {
5         t = a.get(i);
6         a.set(i, a.get(a.size() - i - 1);
7         a.set(a.size() - i - 1, t);
8     }
9 }
10 // Вызов
11 reverse<int>(a);

```

### Вывод шаблонных параметров

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

```

1 MyArray<int> a;
2 MyArray<double> b;
3 reverse(a);
4 reverse(b);

```

#### iv Специализация

Идея: оптимизация для конкретного класса.

Общая версия

```

1 template<typename T>
2 class Array {
3 private:
4     T *a;
5 public:
6     Array (size_t size) {
7         a = new T [size];
8         ...
9     }
10 };

```

Полная специализация

Для bool

```

1 template<>
2 class Array<bool> {
3 private:
4     char *a;
5 public:
6     Array (size_t size) {
7         a = new char [(size-1)/8 + 2];
8         ...
9     }
10 };

```

Частичная специализация

Для массивов

```

1 template<class T>
2 class Array <Array<t>> {
3     T **a;
4 };

```

#### v Шаблонный параметр, не являющийся типом

```

1 template<size_t Size>
2 class Bitset {
3 private:
4     char m[(Size-1)/8 + 1];
5 public:
6     bool get(size_t index) { ... }
7 };
8
9 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

```
1 char *s = NULL;
2 size_t l = strlen(s);
3 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]`
- Обработать 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

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

- Сообщить пользователю или вызывающей функции

```
1 FILE *f = fopen("a.txt", "r");
2 if (f == NULL) {
3     printf("File a.txt not found\n");
4 }
5
6 if (f == NULL) {
7     return -1;
8 }
```

- Предпринять действия по восстановлению (попробовать соединиться еще раз)

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

```

1 FILE* fopen(...) {
2     if (file not found) {
3         errno = 666;
4         return NULL;
5     }
6     if (permission denied) {
7         errno = 777;
8         return NULL;
9     }
10    ...
11 }

```

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

```

1 class Array {
2     int *a;
3 public:
4     // возвращает -1, когда индекс выходит за границу
5     int get(size_t index);
6 };
7 int r1 = atoi("0");
8 int r2 = atoi("a"); // ?

```

Также код логики и обработка ошибок перемешаны

```

1 r = fread(...);
2 if (r < ...) {
3     // error
4 }
5 r = fseek(...);
6 if (r != 0) {
7     // error
8 }

```

## ii try/catch/throw

Структура исключений

```

1 class MyException {
2 private:
3     char message[256];
4     // filename, line, function name ...
5 public:
6     const char* get();
7 };
8
9 double divide (int a, int b) {
10     if (b == 0) {
11         throw MyException("Division by zero");
12     }
13     return a/b;
14 }

```

Структура исключений

```

1 try {
2     x = divide(c, d);
3 }
4 catch (MyException& e) {
5     std::cout << e.get(); // сообщаем пользователю
6     // можем освободить ресурсы
7     // throw e; проинформировать вызвавшую функцию

```

## iii Идиома RAII

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



```

1 void f() {
2     MyArray buffer(n);
3     if ( ... ) throw MyException( ... );
4 }

```

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

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

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:

```

1 void g() {
2     autoPtr p(new Person("Jenya", 36, true));
3     divide(c, e); // может быть исключение

```

```

1 class PhoneBookItem {
2     PhoneBookItem (const char *audio, const char *pic) {
3         af = fopen(audio, "r");
4         pf = fopen(pic, "r");
5         divide(c, e); // исключение
6         f();
7     }
8     ~PhoneBookItem() {
9         fclose(af);
10        fclose(pf);
11    }

```

```

1 class PhoneBookItem {
2     PhoneBookItem (const char *audio, const char *pic) {
3         try {
4             af = fopen(audio, "r");
5             pf = fopen(pic, "r");
6             divide(c, e); // исключение
7             f();
8         }
9         catch(MyException& e) {
10            fclose(af);
11            fclose(pf);
12            throw e;
13        }
14    }
15 }

```

```

1 class PersonDatabase {
2     ~PersonDatabase() {
3         try {
4             // брошена серверная ошибка
5             networkLogger.log("Database is closed.");
6         }
7         catch (...) {} // поймать все
8     }
9 };
10
11 f() {
12     PersonDatabase db;
13     if (...) throw MyException("Error: disk is full.")
14 }

```

#### 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

```

1 void strlen(const char *s) {
2     int count = 0;
3     while (*s != 0) {
4         s++; count++;
5     }
6     return count;
7 }

```

```

1 void f() {
2     try {
3         strlen(s);
4         divide(a, b);
5     }
6     catch (...) { }
7 }

```

**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:

```

1 class PersonDatabase {
2     MyVector<Person> array;
3     void process() {
4         auto_ptr<Person> p(new Person());
5         for (int i = 0; i < array.length; i++) {
6             int a = divide(rand(), rand())
7             // может быть исключение
8             array[i]->setAge(a);
9             std::cout << p;
10        }
11    }
12 };

```

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

Идиомы copy-and-swap

```

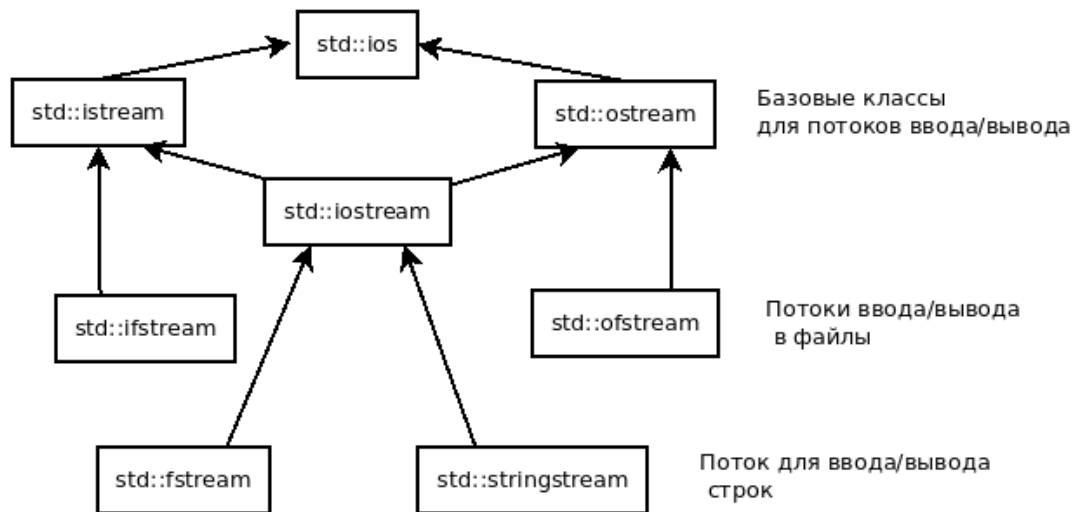
1 class PersonDatabase {
2     MyVector<Person> array;
3     void process() {
4         auto_ptr<Person> p(new Person());
5         MyVector<Person> copy(array);
6         for (int i = 0; i < array.length; i++) {
7             int a = divide(rand(), rand())
8             // может быть исключение
9             copy[i]->setAge(a);
10        }
11        array = copy;
12    }
13 };

```

## Вопрос 3 Ввод-вывод в C++

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

### i Иерархия классов



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

Стандартная библиотека содержит перегруженные операторы `operator >>`, `operator <<` для примитивных типов и строк.

```
1 std::ostream& operator << (std::ostream& os, int v) {
2     // convert int to bytes, write bytes
3     return os;
4 }
```

```
1 std::istream& operator >> (std::istream& is, int& v) {
2     // read bytes, convert to int
3     return is;
4 }
```

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

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

Оператор читает строку до пробела, `getline` до конца строки:

```
1 std::ifstream ifstream if("in.txt");
2 std::string word;
3 if >> word;
4 std::string line;
5 getline(if, line);
```

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

## ii Обработка ошибок

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

## iii Формат вывода

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

Через манипуляторы

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

## iv Ввод-вывод пользовательских типов

```
1 class Point {
2 private:
3     int x;
4     int y;
5 public:
6     friend ostream& operator << (ostream& os, const Point& p);
7     friend istream& operator << (istream& is, Point& p);
8     // friend функция может иметь доступ к приватным членам
9 };
10
11 ostream& operator << (ostream& os, const Point& p) {
12     os << p.x << " " << p.y << "\n";
13     return os;
14 }
15 istream& operator << (istream& is, Point& p) {
16     is >> p.x >> p.y;
17     return is;
18 }
```

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

```
1 void f(char *p);
2 int *pi = malloc(100 * sizeof(int));
3 f(pi); // неявное приведение типов
4
5 int a = 65535;
6 char b = a; // неявное приведение типов
7
8 int c = 3.5; // тоже неявное
9
10 int a = 5; int b = 6;
11 double c = a / (double)b; // явное, действительно шотим
```

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).

```
1 void *f();
2 int* pi = (int*) f();
3
4 // class B : public class A
5 void print(const A* p);
6 B b;
7 print(&b);
```

#### Приведение для классов

```
1 Class BigInt {
2     BigInt(int a); // from int
3     operator int(); // to int
4
5     BigInt(const Complex&); // from Complex
6     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

```
1 int a = 65535;
2 char b = static_cast<char>(a);
3
4 // class B: public class A
5 void f(B *b);
6 A *a = new B();
7 f(static_cast<B*>(a));
```

## reinterpret\_cast

Указатели разных типов

```
1 void* f();
2 int* pi = reinterpret_cast<int*>(f());
3
4 char *pc = ...; int *pi = ...;
5 pc = reinterpret_cast<char*> pi;
```

## const\_cast

Добавление и удаление const

```
1 char const *p1 = "Hello";
2 char *p2 = const_cast<char*>(p1);
3 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

```
1 // class B: public class A;
2 // class C: public class A;
3 void f(B *b);
4
5 A *a = new C();
6 f(static_cast<B*>(a)); // при компиляции без ошибок, но undefined behavior во время работы
7
8 if (dynamic_cast<B*>(a) != 0) {
9     f(static_cast<B*>(a));
10 }
```

Фактический тип

```
1 #include <typeinfo>
2 // class C: public class A
3 A *a = new C();
4 type_info ti = typeid(*a); // требуется ссылка
5 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.

```
1 class Shape {
2     virtual draw() = 0;
3 };
4
5 draw_all(Shape* shapes, size_t n) {
6     for (int i = 0; i < n; i++) {
7         Shape *p = shapes[i];
8         p->draw();
9         if (dynamic_cast<AnimatedShape*>(p) != 0)
10             p->animated_draw();
11     }
12 }
```

## 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.

```
1 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

```
1 #include <vector>
2
3 int main() {
4     std::vector<int> v;
5     v.push_back(10);
6     v.pop_back();
7 }
```

```

7      v[0] = 13;
8      v.at(0) = 14;
9      size_t size = v.size();
10     bool is_empty = v.empty();
11     v.clear();
12     v.resize(10); // меняем размер на 10 элементов, работает, если есть конструктор по умолчанию
13     v.resize(20, 5); // новые 10 элементов будут равны 5
14
15     int *dst = new ...
16     memcpy(dst, v.data(), v.size());
17
18     v.reserve(100); // резервируем память на 100 элементов, размер не меняется
19     v.reserve(v.size() + 100);
20     v.clear(); // изменил размер до 0, но вместимость не изменится
21     vector<int>(v).swap(v); // уменьшить размерность вектора до реально используемых элементов
22     return 0;
23 }

```

## 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:

```

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

```

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

```

1  f(const std::string& s);
2  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.

```

1  class vector {
2      ...
3      class iterator {
4          operator ++() { T* ptr++; }
5      }

```



```

6 }
7 vector::iterator it1;
8
9 class list {
10     class iterator {
11         operator ++() { Node<T> ptr = ptr->next; }
12     }
13 }
14 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 указателей:

```

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

```

Проход по контейнеру:

```

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

```

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

```

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

```

Удаление элемента, на который указываем

```

1 it = v.erase(it);

```

Вставка элемента на данную позицию

```

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

```

## Инвалидация итераторов

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

```

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

```

Поставить перед каждым четным элементом вектора 0:

```

1 std::vector<int> v;
2 for (std::vector<int>::iterator i = v.begin(); i != v.end(); ++i) {
3     if (*i % 2 == 0) {
4         i = v.insert(i, 0);
5         ++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 поиска ( $O(\log n)$ )
2. Возможные реализации: дерево поиска  $O(\log n)$
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.

```
1 #include <set>
2
3 std::set<int> s;
4 s.insert(10); s.insert(20); s.insert(10); // s.size() = 2
5
6 std::multiset<int> ms;
7 ms.insert(10); ms.insert(20); ms.insert(20); // ms.size() = 3
8
9 std::cout << s.count(20) << " " << ms.count(20) << std::endl;
```

Порядок не сохраняется.

```
1 std::set<int> s;
2 s.insert(10);
3 s.insert(6);
4 s.insert(20);
5 s.insert(5);
6 s.insert(1);
7
8 for (std::set<int>::iterator it = s.begin(); it != s.end(); ++i)
9     std::cout << *it << " ";
10 } // выведет 1 5 6 10 20
```

Присваивание запрещено.

```
1 std::set<int> s;  
2 s.insert(10);  
3 std::set<int>::iterator it = s.find(10);  
4 *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

```
1 template<class F, class S>  
2 struct pair {  
3     F first;  
4     S second;  
5     ... // конструкторы  
6 };  
7  
8 template<class F, class S>  
9 pair<F, S> make_pair(F const& f, S const& s);  
10  
11 template<class Key, class T, ...> class map {  
12     ...  
13     typedef pair<const Key, T> value_type;  
14 };
```

```
1 #include <map>  
2  
3 std::map<string, int> phonebook;  
4 phonebook.insert(std::pair<string, int> ("Mary", 2128506));  
5 phonebook.insert(std::make_pair("Alex", 9286385)); // вывод типов параметров  
6 phonebook.insert(std::make_pair("Bob", 2128506));  
7  
8 std::map<std::string, int>::iterator it = phonebook.find("John");  
9 if (it != phonebook.end())  
10     std::cout << "Jonh`s p/n is " << it->second << std::endl;  
11  
12 for (it = phonebook.begin(); it != phonebook.end(); ++it)  
13     std::cout << it->first << ": " << it->second << "\n";
```

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

```
1 T & operator[](Key const& k) {  
2     iterator i = find(k);  
3     if (i == end())  
4         i = insert(k, T());  
5     return i->second;  
6 }
```

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

Пример

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

### iii unordered\_map, unordered\_set

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

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

Для своего типа нужно реализовать hash:

```
1 class Point {
2 private:
3     int x, y;
4 private:
5     bool operator == (const Point& rhs) const {
6         return x == rhs.x && y == rhs.y;
7     }
8 };
9
10 namespace std {
11     template<>
12     struct hash<Point> {
13         size_t operator() (Point const &p) const {
14             return (std::hash<int>()(p.getX()) * 53 + std::hash<int>()(p.getY())) * 239;
15         }
16     };
17 }
18
19 std::unordered_set<Point> points;
```

### iv Инвалидация

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

### В C++11

```
1 for (; it != m.end(); )
2     if (it->second == 0)
3         it = m.erase(it);
4     else
5         ++it;
```

### v Собственный компаратор

```
1 struct Person {
2     string name;
3     string surname;
4 };
5 bool operator < (Person const& a, Person const& b) {
6     return a.name < b.name || (a.name == b.name && a.surname < b.surname);
```

```
7 }  
8 std::set<Person> s1; // уникальные по name и surname  
9  
10 struct PersonComp {  
11     bool operator () (Person const& a, Person const& b) const {  
12         return a.surname < b.surname;  
13     }  
14 };  
15 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 которых похожи на функцию и перегружен оператор (). Используется, когда нужно описать алгоритму, что нужно сделать, при этом может иметь поля.

```
1 struct sum_sq {
2     int operator() (int a, int b) const {
3         return a * a + b * b;
4     }
5 };
6 sum_sq f;
7 int res = f(3, 4);
```

сравниваем с данным числом

```
1 struct cmp {
2     int value;
3     cmp(int v): value (v) {}
4
5     bool operator () (int a) const {
6         return a < value;
7     }
8 };
9
10 cmp f(2);
11 bool b = f(13);
```

Такой функтор (с bool) называют предикатом.

Подсчет суммы

```
1 struct accum {
2     int acc;
3     accum() : acc(0) {}
4
5     void operator () (int a) {
6         acc += a;
7     }
8 };
9
10 accum f;
11 f(13); f(16);
```

На функциях можно сделать аналогично через 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

```

1 struct Person {
2     int age;
3     string name;
4     string city;
5     Person (...) {...}
6 };
7 struct by_city {
8     bool operator () (const Person &p1, const Person &p2) const {
9         return p1.city < p2.city;
10    }
11 };
12
13 Person p1(30, "V", "Msc"); Person p2(15, "K", "Spb");
14 by_city f;
15 std::max<Person, by_city> (p1, p2, f);
16 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 последовательность 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, вторая должна быть той же длины. Если короче, то undefined behaviour.
- `pair<It, Itr>mismatch(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.

```

1 std::vector<int> v = {1, 2, 5, 3, 4, 5, 5, 2};
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 записываем в последовательность, начинающуюся с итератора out

```

1 std::vector<int> v = {5, 2, 8, 9, 4, 1};
2 std::sort(v.begin(), v.end());
3

```

```

4 std::vector<int> v1 = {5, 3, 3, 4, 1, 5, 5, 4};
5 std::vector<int>::iterator last = std::unique(v1.begin(), v1.end());
6 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.

```

1 std::vector<int> v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
2 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.

```

1 template<class InputIt, class OutputIt>
2 OutputIt copy(InputIt first, InputIt last, OutputIt d_first) {
3     while (first != last)
4         *d_first++ = *first++;
5     return d_first;
6 }
7
8 vector<int> v;
9 list<int> l;
10 copy(v.begin(), v.end(), l.begin());
11
12 template<class InputIt, class OutputIt, class UnaryPredicate>
13 OutputIt copy_if(InputIt first, InputIt last, OutputIt d_first, UnaryPredicate pred) {
14     while (first != last) {
15         if (pred(*first))
16             *d_first++ = *first;
17         first++;
18     }
19     return d_first;
20 }
21 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.

```
1 template<class T>
2 class vector {
3     T *array;
4     class iterator {
5         typedef T value_type;
6     };
7 };
```

```
1 template<class It>
2 void q_sort(It p, It q) {
3     It::value_type pivot = *p;
4 }
```

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).

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

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

RA

```
1 template <class T>
2 class vector {
3     T *array;
4     class iterator {
5         typedef ra_tag iterator_category; // пустая структура для хранения типа
6     };
7 };
```

BiDi

```
1 template <class T>
2 class list {
3     Node *head;
4     class iterator {
5         typedef bidi_tag iterator_category; // пустая структура для хранения типа
6     };
7 };
```

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

```
1 template <class Iterator>
2 Iterator advance (Iterator it, int amount) {
3     typedef iterator::iterator_category tag;
4     advance(it, amount, tag());
5 }
6
7 template <class RAIterator>
8 RAIterator advance (RAIterator it, int amount, ra_tag t) {
9     return it + amount;
10 }
11
12 template <class BiDiIterator>
13 BiDiIterator advance (BiDiIterator it, int amount, bidi_tag t) {
14     for (; amount; --amount;) ++it;
15     return it;
16 }
```

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

```
1 template<class It>
2 typename std::iterator_traits<It>::difference_type
3 distance(It first, It last) {
4     return detail::do_distance(first, last,
5                               typename std::iterator_traits<It>::iterator_category());
6 }
7
8 template<class It>
9 typename std::iterator_traits<It>::difference_type
10 do_distance(It first, It last, std::input_iterator_tag) {
11     std::iterator_traits<It>::difference_type result = 0;
12     while (first != last) {
13         ++first;
14         ++result;
15     }
16     return result;
17 }
18
19 template<class It>
20 typename std::iterator_traits<It>::difference_type
21 do_distance(It first, It last, std::random_access_iterator_tag) {
22     return last - first;
23 }
```

## Вопрос 9 C++11. Разное.

- lambda и захваты
- auto, decltype
- initializer\_list
- shared\_ptr, unique\_ptr (использование)

### i Вывод типов

```
1 auto x = expression; // тип x выводится по expression
2 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.

```
1 for (std::vector<std::string>::const_iterator i = v.cbegin(); i != v.cend(); ++i) {
2     std::vector<std::string>::const_reverse_iterator j = i;
3     ...
4 }
```

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

```
1 for (auto i = v.cbegin(); i != v.cend(); ++i) { // v.cbegin() - константный итератор
2     decltype(v.cbegin()) j = i;
3     ...
4 }
```

Еще отличия auto и decltype:

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

```
1 typedef decltype(v[0]) new_type; // decltype в typedef
```

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

Но есть побочный эффект:

```
1 std::map<KeyClass, ValueClass> m;
2 auto I = m.find(something); // все знают, что find вернет iterator
3
4 MyClass myObj;
5 auto ret = myObj.findRecord(something);
6 // большинству придется сходиться посмотреть, что возвращает findRecord
```

### ii lambda

```
1 std::vector<int> v = {50, -10, 20, -30};
2 // до C++ нужен компаратор
3 struct abs_cmptr {
4     bool operator () (int a, int b) const {
5         return abs(a) < abs(b);
6     }
7 }
```

```

6         }
7     };
8     std::sort(v.begin(), v.end(), abs_cmp);
9
10    // C++11 есть lambda функции для упрощения таких задач
11    std::sort(v.begin(), v.end(), [](int a, int b) { return abs(a) < abs(b); });

```

## Структура lambda функций

```

1  [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.

```

1  template<class T>
2  void foo(T& v) {
3      auto i = begin(v);
4      auto e = end(v);
5      for (; i != e; i++) {}
6  }
7
8  template<class T, size_t N>
9  T* end(T (&a)[N]) { return a + N; }
10
11  // параметр N выводится компилятором
12  int a[4] = {0, 1, 2, 3};
13  auto e = end(a);

```

С помощью этого реализован `for(x : )`

```

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

```

## iii initializer\_list

Инициализация по списку элементов для своих объектов.

```

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

```

По такому коду генерируется `initializer_list`

```

1  template <typename T>
2  class vector {
3  public:
4      vector (std::initializer_list<T> l) {
5          const T *array = l.begin();
6          T value = *(array++);
7          ...
8          l.size();
9      }
10 };

```

#### iv unique\_ptr, shared\_ptr

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

```
1 struct Foo {
2     ...
3     void bar() {std::cout << "Foo::bar\n";}
4 };
5
6 void f(const Foo&) {std::cout << "f(const Foo&)\n";}
7
8 int main() {
9     std::unique_ptr<Foo> p1(new Foo);
10    if (p1) p1->bar();
11
12    // теперь только p2 будет владеть Foo
13    std::unique_ptr<Foo> p2(std::move(p1));
14    f(*p2);
15
16    p1 = std::move(p2); // можем вернуть p1
17    if (p1) p1->bar();
18 }
```

## 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

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

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

```
1 int square(int x) { return x*x; }
2 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.

```
1 class X {
2     int *array;
3     X(size_t size);
4     ~X();
5     X(const X&);
6     X& operator = (const X&);
7 };
8
9 X foo();
10 X x;
11 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.

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

```

6         X& operator = (const X& x);
7
8         X(X&& x) { std::swap(this->array, x.array); } // X&& - rvalue reference, это move constructor
9         X& operator = (X&& x) { this->array = std::move(x.array) } // это move assignment
10    };
11
12    X x1(100);
13    X x2(x1); // вызываем X(const X& x), так как x1 - lvalue
14    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:

```

1 template<class T>
2 void swap (T& a, T& b) {
3     T tmp(std::move(a));
4     a = std::move(b);
5     b = std::move(tmp);
6 }

```