

# Конспект по C++

Черепанов Валерий

7 июня 2016 г.

## 1 25 февраля 2016 г.

### 1.1 Универсальная сортировка

Сортировка с стиле C

```
{void nsort(void *array, size_t n, size_t elem_size, int (*fcmp)(void*, void*));}
```

Сортировка с стиле C++

```
// Interface
class Comparable {
    virtual int compare(const Coparable*) = 0 const;
}

// ** потому что мы не знаем размер наследников 'Comparable'.
void nsort(Comparable **c, size_t n);
```

Поддержка встроенных типов

```
template<typename T>
int nsort(T* array, size_t size) {
    if (array[i] < array[i+1])
        T t = array[i]; // swap ...
}

// OK
int a[100];
nsort(0, 100);

Worker a[100]; // Требуется оператор <, конструктор копий, оператор =
Worker **w; // Это работает?
```

```
template<typename T, size_t size>
class Array {
    T array[size];
```

```

    size_t getSize() {
        return size;
    }
}

```

```

Array<int, 100> a;
f(a); // Как выглядит сигнатура f?

```

## Проблема с инстанцированными шаблонами

```

template<class T, class Container>
class Stack {
    Container C;
    void push(const T&);
    T pop();
};

```

```

Stack<int, Vector<int>>;
Stack<int, list<int>>;
Stack<double, vector<int>>; // Пользователь ошибся
s.push(3.5);
double d = s.pop(); // Возвратит 3, но не факт, что будет хотя бы warning

```

Решение:

```

// vector - значение по умолчанию
template<class T, template <typename V> class Container=vector>
stack<int, list>
class Stack {
    Container<T> c;
    ...
};

```

## Алтернативная реализация для определенных типов

```

template<class T>
class MyArray {
    T *array;
};
template<>
class MyArray<bool> { // Используется только для bool
};

```

2 3 марта 2016 г.

### 2.1 Исключения

```

class BookEntry {
    Image *myImage;
}

```

```

char *name;
BookEntry() {
    name = new char[256];
    myImage = new myImage;
    myImage.load(...);
}
~BookEntry() {
    delete [] name;
    delete myImage;
}
}

```

Проблема — если конструктор получит exception, то деструктор не будет вызван.

Варианты решения:

1. RAII, то есть, например,

```
shared_ptr<Image>
```

2.

```

NetworkConnection {
    connect() {
        // работа с сетью, при ошибке Exception
    }
    ~NetworkConnection {
        logger log;
        log.print("Destructor NC");
    }
}

void main() {
    try {
        NetworkConnection nc;
        nc.connect(...);
    } catch (...) {

    }
}

```

Неприятная ситуация — допустим, было вызвано какое-то исключение, оно попало в деструктор, а в деструкторе `log.print` тоже бросил исключение. Тогда исходное исключение потеряется. Поэтому в C++ запрещено бросать исключение в деструкторе. При появлении оно сразу летит в `main` и кладет программу.

### 2.1.1 Гарантия исключений

**No except** просто не бросаем исключения.

**Base guarantee** даже при исключении класс остается валидным.

**Strong guarantee** если произошло исключение, то класс возвращается в исходное состояние. Обычно для этого делается копия изначального состояния класса.

## 2.2 STL

### 2.2.1 Конфликт заголовков

Некоторые заголовки C и C++ называются одинаково. Поэтому к названиям заголовков C в начале добавляется буква “с”.

### 2.2.2 Конфликт имен

Чтобы не засорять лишний раз scope можно объявлять классы внутри классов.

Развитие этой идеи — namespaces.

**Замечание 1.** По очевидным причинам не стоит писать *using* в заголовочных файлах.

У ifstream есть 4 флага:

**good** Все ок.

**fail** Неправильный формат.

**bad** Что-то не то с файлом.

**eof** Конец файла

```
ifs.exception(std::ifstream::badbit | ...); // При выставлении флага вылетит эксерт
```

## 3 18 марта 2016 г.

### 3.1 Исправление некоторых неточностей

#### 3.1.1 reserve

Для вызова `vector.reserve` не требуется вызывать какие-либо конструкторы (даже по умолчанию). Там используется placement new, то есть просто выделяется кусок памяти с помощью new и sizeof, а потом пишется `new(address) Obj` где `address` — это выделенная память.

#### 3.1.2 string

Раньше string старались быть ленивыми и при создании старались сделать ссылку на уже созданное. Но плохо было то, что мы могли с легкостью получить линию при обращении с помощью [] (строка копировалась в новую память, чтобы ее можно было модифицировать), да к тому же мы могли инвалидировать итератор.

#### 3.1.3 Манипуляторы потоков

Манипуляторы вроде hex как правило просто выставляют необходимый флаг с помощью `setf`. Для их использования перегружен оператор <<, он принимает поток и указатель на функцию.

## 3.2 STL

### 3.2.1 Кое-что про map

Как уже говорилось, `map` — это по сути `set` пар. Оператор `[]` у `map` создает новый элемент (вызывая стандартный конструктор), если мы обратились по несуществующему ключу, потому что он не знает, хотим ли мы лишь получить значение по ключу или изменить его.

### 3.2.2 Кое-что про set

```
class Person {
    string name;
    int age;

    bool operator <(const Person& p) {
        return name < p.name;
    }
}
```

Если мы создадим `set<Person>`, то элементы будут отсортированы по полю `name`. Но мы можем захотеть сортировать по `age`. Как этого добиться? На помощь приходят функторы!

```
struct by_age {
    bool operator()(const Person& p1, const Person& p2) {
        return p1.age < p2.age;
    }
}
```

```
set<Person, by_age> s; // Используем так
```

Внутри это устроено примерно так:

```
template<typename T, class comparator>
class set {
    insert(...) {
        if (comparator()(n1, n2)) { ... } // Анонимный объект
    }
}
```

Если мы не передаем в `set` второй шаблонный параметр, то используется стандартный функтор `less`. Он выглядит примерно так:

```
template<typename T>
struct less {
    bool operator()(const T& t1, const T& t2) {
        return t1 < t2;
    }
}
```

**Замечание 2.** *multimap* и *multiset* — это тоже деревья, но обычно в узле хранится список элементов.

### 3.2.3 algorithm

Итераторы — обертки над указателями. Их идея заключается в том, что алгоритм может работать с разными структурами данных, поддерживающими одинаковые операции (от структуры требуется лишь предоставить итераторы и методы работы с ними).

Некоторые алгоритмы STL:

1. `swap(T& a, T& b), max(T& a, T& b)`
2. `count[_if](It a, It b, const T& x)`  
`x` — значение в `count`, функция или функтор в `count_if` (можно считать сложные функции или экономить время на сравнение).
3. `equal(It a, It b, It it) // Сравнивает [a, b) с [it, ...)`
4. `sort, min_element, nth_element, reverse, ...`

### 3.2.4 Чем хороши функторы?

Тем, что можно передавать параметры!

```
class Finder {
private:
    int what;
public:
    Finder(int w):what(w){}
    bool operator()(int n) {
        return n > what;
    }
};
// v - какой-то контейнер
find_if(begin(v), end(v), Finder(473));
```

### 3.2.5 Чем плохи итераторы?

У них слишком маленький базовый интерфейс (`++`, `--`, `*`, `->`), поэтому они “из коробки” не подходят для адекватной реализации многих алгоритмов (даже для бинарного поиска). Кроме того, мы не знаем тип объекта итератора, а он может нам понадобиться. Для решения этих проблем были придуманы `iterator_traits`.

Как часть решения первой проблемы алгоритмы обарачивают некоторые обращения к итераторам используя функции `distance` и `advance`.

С точки зрения перемещения итераторы бывают:

1. Forward
2. Bidirectional
3. Random Access

## 4 25 марта 2016 г.

### 4.1 Итераторы внутри STL

```
template<class Iter>
void sort(Iter p, Iter q);

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

#### Проблемы

1. Знаем итератор, но не знаем, например, тип элементов вектора.
2. Не знаем, что умеет итератор (например, может ли он в random access?). Поэтому большинство операций с итератором обарачиваем в библиотечные функции

```
advance(Iter& it, int n);
distance(Iter& it1, Iter& it2);
```

#### Решения

Как решена проблемы в STL?

```
template<class T>
class vector {
    T *array;
    class Iterator {
        typedef value_type T; // Решение первой проблемы
        // В sort пишем typename Iter::value_type var;
        typedef iterator_category ra_iterator; // Решение второй проблемы
    };
};
```

Как делать “if” по типу? Перегрузкой!

```
template<class Iter>
void advance (Iter it, int n) {
    typename Iter::iterator_category ite;
    advance_impl(it, n, ite);
}

template<class Iter>
void advance_impl(Iter& it, int n, ra_iterator it) {
    it += n;
}

template <class Iter>
void advance_impl(Iter it, int n, int n, bidi_iterator it) {
    int i = 0;
```

```

    if (n > 0) {
        while(i < n) {
            ++it;
            ++i;
        }
    }
    if (n < 0) {
        while(i > n) {
            --it;
            --i;
        }
    }
}

```

Используется полиморфизм времени компиляции.

## 4.2 Iterator traits

Хотим делать примерно то же самое, но не для итераторов, а для указателей. Проблема:

```

template<class Iter>
void sort(Iter p, Iter q) {
    Iter::value_type;
}

```

Если вызовем sort от двух указателей, то получим compilation error.

Решение проблемы:

```

template<class Iter>
class iter_traits {
    typedef value_type Iter::value_type;
    typedef iterator_category Iter::iterator_category;
}
vector<int>;
typename iter_traits<vector<int>::iterator>::value_type a;

```

Кажется, мы ничего на самом деле не решили, а просто написали какую-то чушь. Но на самом деле это не так, нужно лишь воспользоваться специализацией шаблонов!

```

template<typename Iter*> // специализация для указателей
class iter_type {
    typedef value_type Iter;
};

Iter::iterator_category → iter_traits<Iter>::iterator_category

```



5 08 апреля 2016 г.

## 5.1 Преобразования типов

```
double d1 = (double)3/4;
char *d = new double[100];
char *cd = (char)d; // Хотим вызвать send, поэтому cast

// В C часто использовали void*, char*.
int send(char *c, size_t size);
void *malloc(...);
sort(void *a, size_t n);
```

### Новые возможности C++

#### const\_cast

Убираем и добавляем константность.

```
void old_c_func(char *s);
f(const char *s) {
    old_c_func(s); // Не скомпилируется

    char *c1 = const_cast<char*>(s); // Аналог (char *)s из C
    old_c_func(c1);
}
```

// Проблема: похожие константные и не константные методы

```
class Vector {
    T& operator[](int id) {
        return const_cast<const Vector&>(*this)[id]; // Или не так?
        % FIXME (may be wrong)
    }
    const T& operator[](int id) const {

    }
}
```

// Просто примеры с const

```
void foo(string &s); // Если напишем const string, то все будет ок
foo("English"); // Не скомпилируется. Проблема в том, что мы передаем ссылку на временный
```

#### static\_cast

Обычное приведение в духе C.

```
double d = static_cast<double>(3);
A *xa;
B *b = new B();
a = static_cast<A*>(b); // B к A
```

```
C *c = new c();
C *c1 = static_cast<C*>(a); // А (на самом деле В) к С. Компилятор ничего не понимает
```

### reinterpret\_cast

Говорит компилятору рассматривать что-то одного типа как что-то другого типа. Не генерирует какой-либо ассемблерный код.

### dynamic\_cast

Выполняется только в рантайме. Проверяет пристальнее, чем `static_cast`.

```
A* a;
B *b = new B();
// Если кажется, что можно сделать исходя из иерархии, то ок
a = static_cast<A*>(b);
// Хранит указатель на массив виртуальных функций и понимает больше
C *c = dynamic_cast<C*>(a); // исключение

Vector<A*> v;
v.add(new B());
v.add(new C());
v.add(new D());
v.add(new E()); // Есть какой-то метод g, которого нет у А, хотим вызывать его вместо g
for(int i = 0; i < n; i++) {
    v[i]->f();
}

#include <typeinfo>
E *pe;
if(pe=dynamic_cast<E*>(v[i])) { // Проверим, Е ли это
    pe->g();
} else {
    v[i]->f();
}
```

`typeid` Выполняется в рантайме, действует как `dynamic_cast`

```
type_info ti = typeid(*c);
ti.name(); // "MyClass"
```

## 6 15 апреля 2016 г.

### 6.1 C++11

#### 6.1.1 Избавляемся от define

`nullptr` Раньше было так:

```

// Было
int *x = 0;
int *p1 = NULL; // #define NULL (void*)0;

void f(long l) { }
void f(char *s) { }

int main() {
    f(0L); // OK
    char s[] = "Hello";
    f(s); // OK
    f(NULL); // Ambiguity
    f(0);    // Ambiguity
}

```

В C++ эту проблему постарались решить с помощью нового ключевого слова `nullptr`. Такой “указатель” неявно кприводиться к указателю любого другого типа.

**static\_assert** Раньше был только обычный `assert`.

```

// Если boolvar = false, то runtime error.
assert(boolvar);

```

Теперь появился `static_assert`, который выполняется на стадии компиляции.

```

// Второй аргумент - сообщение об ошибке
static_assert(sizeof(int)>=4, "This program could not work");

template<typename T, size_t size>
class static_array {
    T array[size];
    static_array() {
        static_assert(size > 0, "Empty arrays are prohibited");
    }
}

static_array<int, 100> a; // OK
static_array<int, 0> b;   // Error

```

**Type alias** Раньше:

```

// Так можно
typedef map<string, list<int>> ml_t;
f(ml_t a);

```

```

// А вот так уже нет
typedef map<string, list<T>> ml_t;
f(ml_t<int> a);

```

C++11 спешить на помощь!

```

template<typename T>
using mt_t=map<string, list<T>>

```

**constexpr** Было:

```
size_t square(int x) a {  
    return x*x;  
}  
const int sz = 1;  
int a[sz]; // OK  
static_array<int, sz> sa; // OK  
int a[square(3)]; // Error  
static_array<int, square(3)> sa; // Тоже error
```

Решение:

```
// constexpr говорит, что функция при определенных  
// условиях может использоваться в константных выражениях  
constexpr size_t square(int x) a {  
    return x*x;  
}
```

### 6.1.2 Декораторы

**default** и **delete**

```
class T {  
    int v = 0; // Такая инициализация - тоже фишка C++11  
public:  
    // default говорит, что в любом случае нужно создать метод по умолчанию  
    T() = default;  
    // delete говорит, что не стоит создавать метод по-умолчанию  
    T& operator= (const T&) = delete;  
    T(const T&) = delete;  
}
```

**override** Проблема:

```
class Base {  
public:  
    virtual void f(int) const;  
    virtual int g() const;  
    void n(int) const;  
}  
  
class Derived:public Base {  
    void f(int) const;  
    virtual int g(int); // Забыли const  
    void n(int) const;  
}  
  
const Base *b = new Derived();
```

```

b->f()
b->g(); // Base::g(), так как сигнатуры не совпадают
b->n();

```

Решение — **override**. Это слово говорит компилятору, что функция что-то перекрывает. “Под капотом” компилятор просто проверяет, есть ли такая виртуальная функция в базовом классе.

```

class Derived:public Base {
    void f(int) const override; // OK
    virtual int g(int) override; // Error
    void n(int) const override; // OK
}

```

## Конструкторы

```

class MyNumber {
    MyNumber(char* s): A(atoi(s)) { } // Вызываем один конструктор из другого
    MyNumber(int i) { };
}

```

```

MyNumber("64");
MyNumber(64);

```

## 7 22 апреля 2016 г.

### 7.0.1 auto и decltype

```

vector<int>::iterator it = vec.begin();
auto it = vec.begin(); // автоматический вывод типа

```

```

int a;
int& ref = a;
auto b = ref; // int, & снимается
auto& c = ref; // int

```

```

template<typename T, typename V>
void multiply(vector<T>& a, vector<T>& b) {
    auto tmp = a[i]*b[i]; // Error. Не знаем, какой будет тип
}

```

```

template<class T, class V>
decltype(T()*V()) multiply(const T& a, const V& b) {
    decltype(a*b) product_type; // Такой же тип, как у operator*(T, V);
    product_type tmp = a*b;
    return tmp;
}

```

```
template<class T, class V>
// Просто auto нас не поймет (хотя в C++14 может)
auto multiply(const T& a, const V& b) -> decltype(a*b) {
    return a*b;
}
```

## 7.1 move семантика

Используется, если класс(структура) имеет указатели. В таком случае, если мы хотим “переместить” объект, то можно лишь переместить их, а не копировать что-либо. В C++ все выражения делятся на lvalue и rvalue. Грубо говоря, lvalue — то, что можно писать слева от “=”, а rvalue — все остальное.

```
template<class T>
// Будут выполнены лишние действия в виде копирований (например, из a в tmp)
void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

Решение: конструктор и оператор присваивания.

```
// %FIXME UNDONE
```

## 8 15 апреля 2016 г.

### 8.0.1 Lambda функции

```
find_if(Iterator first,
        Iterator last,
        Functor operator);
```

Предположим, что мы хотим найти первое положительное значение в контейнере. Пишем так:

```
struct PositiveFinder {
    bool operator()(int v) { // Для простоты int
        return v > 0;
    }
};
```

```
vector<int> v(100);
auto f = find_if(begin(v), end(v), PositiveFinder());
```

Что здесь плохо? Громоздкий синтаксис, для каждой цели нужно писать свой функтор (ну или хотя бы функцию). Кроме того, это засоряет namespace.

В C++11 предложено такое решение с использованием лямбд (анонимных классов).

```

auto f = find_if(begin(v), end(v), [](int n) {return n > 0});
// Можно и с параметрами
f = find_if(begin(v), end(v), [threshold](int n){return n > threshold});
// Варианты
[=] // Все локальные текущего scope передаются по значению
[&] // Тоже самое, но по ссылке
[=var1, &var2] // Одно по значению, одно по ссылке (= можно не писать)

```

Захват переменных ([]) называется замыканием (closure).

## 8.1 variadic templates(и functions)

Это есть даже в C.

```

// Все делается написанием одной-единственной функции
printf("%s, %d", s, d);
printf("%d", d);

// Здесь определены макросы va_start, va_arg, va_end
#include <stdarg.h>

int min(int n, ...) { // Произвольное число аргументов
    va_list args; // Список аргументов
    va_start(args, n); // Аргументы начинаются после n
    int m = 0, m = INF;
    while(i++ < n) { // n - число аргументов
        int arg = va_arg(args, int); // Извлекаем очередной аргумент типа int
        m = min1(m, arg);
    }
    // Если аргументы выделяются с помощью malloc, то надо очистить память
    va_end(args);
    return m;
}

```

Как это устроено внутри? Может быть много вариантов, один из них такой:

```

typedef unsigned char* va_list;
// va_start делает примерно это:
args = &n + sizeof(n);
// va_arg делает примерно это: %FIXME подумать
T var = T(*args);
args += sizeof(T);

```

В C++ развили эту идею.

```

struct Fact {
    static const int val = n*Fact<n-1>::val;
};

```

```

template<>
struct Fact<0> {
    static const int val = 1;
};

int main() {
    int v = Fact<5>::val; // Вычисляется во время компиляции
}

template<typename T, typename ...Args>

T min(T n, Args ...rest) {
    return min(n, min(rest...));
}

template<typename T>
T min(T a, T b) {
    if (a < b) return a;
    return b;
}

```

### 8.1.1 Многопоточное программирование

Допустим, мы хотим перемножить матрицы. Ясно, что левую часть полученной матрицы можно считать независимо от правой. Можно использовать стандартный подход “разделяй и властвуй”.

Но если у нас есть один процессор, то мы не получим выигрыша, а только проигрыш на рекурсию и переключение между процессами.

Как происходит переключение? В операционной системе есть так называемый scheduler, который следит за всем этим. У каждого процесса контекст, то есть всякие указатели на исполняемый в данный момент участок кода и так далее. При переключении процесса нужно записать контекст в память/регистры, поэтому это требует времени. Такая многозадачность (со скедулером) называется вытесняемой.

Есть другие виды многозадачности, например такой: В сами потоки записаны команды на передачу управления. Плюс в том, что меньше накладных расходов, а минус в том, что если в программе баг, то может повиснуть вся система.

Многопоточность не всегда возможна в рамках одной программы, потому что некоторые алгоритмы требуют последовательное выполнение. Но при этом она необходима на уровне операционной системы для поддержки многозадачности.

Обычно кроме потоков, которые что-то считают есть еще и управляемый поток, который делает join, то есть ждет, пока доработают все потоки и тогда уже выводит результат.

**Атомарность** Запустим каждую функцию в своем потоке.

```

SomeClass n = 0;
void f1() { // Поток t1
    while(true)
        n.change();
}

```



```

}
void f2() { // Поток t2
    while(true)
        cout << n;
}

```

Какая здесь могут быть проблемы?

Во время операции t1 Поток t2 может попробовать вывести n в то время как над ним выполняется операция `increment`. Таким образом, может произойти попытка вывода объекта n в неконсистентном состоянии.

Кэширование Разберем в следующий раз.

Есть такой код:

```

// В первом потоке
for(int i = 0; i < 10000; i++) {
    x++;
}

// Во втором потоке
while(true) {
    cout << x;
}

```

Что должно получиться на выходе из первого потока?

```

; 10000 раз
load
add
store

```

Компилятор может все оптимизировать до:

```

load
add ; 10000 раз
store

```

Понятно, что полученное поведение может быть неожиданным.

Еще одна подобная проблема:

```

int quit = 1;
while(quit) {

}

```

Компилятор может заменить цикл на `while(true)`. Для решения подобных проблем еще ключевое слово `volatile`, которое говорит компилятору, что переменная может быть изменена извне и с ней нужно быть поаккуратнее.

## 8.2 Потоки в C++

```
void hello() {
    cout << "Hello from" << this_thread.get_id() << '\n';
}

int main() {
    vector<thread> threads;
    for (int i = 0; i < 5; i++) {
        threads.push_back(thread(hello));
    }
}
```

На самом деле в программе не 5, а 6 потоков (один для `main`), но программа окончит работу именно после завершения потока `main`. Чтобы это починить пишем примерно так:

```
for(auto& thread: threads) {
    thread.join(); // Блокирует оновной поток, до тех пор пока не закончится данный
}
```

**Thread-safe** Данный термин означает, что что-либо можно без проблем использовать в многопоточном окружении. При это он не означает атомарность. Например, пример с `hello` выше может смешивать все 5 строк.

Вопрос к тесту:

```
// Первый способ
thread t(hello);
v.push_back(hello);

// Второй способ
v.push_back(t(hello));
```

Чем они отличаются и стоит ли использовать `std::move`?

Какие неприятности могут возникать с `shared` переменными.

```
volatile int x;

// Thread 1
while(true) {
    x++;
}

// Thread 2
while(true) {
    if (x%2==0)
        cout << x;
}
```

Автор ожидал увидеть на экране только четные числа, но видит все подряд. Это называется состояние гонки.

```
struct Counter {
    volatile int value;
    Counter(): value(0) {}
    void increment() {
        ++value;
    }
};

Counter ctr;
for (int i = 0; i < 5; i++) {
    threads.push_back(
        thread([&ctr]() { for (int i = 0; i < 100; i++) { ctr.increment(); }}}));
}
// Join
cout << ctr.value;
```

Автор ожидал увидеть 500, но он снова ошибся. Что могло пойти не так? Два потока могли параллельно загрузить переменную с одним и тем же значением в какой-то регистр, сделать там инкремент и загрузить. Тогда в результате нескольких инкрементов мы увидим изменение значения переменной лишь на единицу. Подобные проблемы решаются семафорами, один из видов которых — мьютекс мы сейчас рассмотрим.

```
#include <mutex>
mutex mut;
mut.lock();
// Какие-то действия
mut.unlock();
```

Если один поток зашел находится внутри мьютекса, то другой туда зайти не может и ему придется подождать.

```
void decrement() {
    mut.lock();
    if (value == 0)
        throw exception(" ");
    --value;
    mut.unlock();
}
```

Проблема в том, что из-за исключения не будет вызван деструктор. Для решения этой проблемы есть `lock_guard`

```
lock_guard<mutex> l(mut);
// Код decrement без mut.*;
```

```

class Integer {
    int i;
    mutex mut;
    void mul() {
        lock_guard<mutex> l(mut);
        i *= 3;
    }
    void div() {
        lock_guard<mutex> l(mut);
        i /= div;
    }
    void both() {
        lock_guard<mutex> l(mut); // Deadlock!
        mul();
        div();
    }
}

```

Взяли мьютекс и тут же снова хотим его взять. Не классический дедлок, обычно два потока ждут друг друга, а не один ждет сам себя. Существует `recursive_mutex` который можно брать несколько раз и в такой ситуации все будет ок.

## 9 13 мая

### 9.1 `timed_mutex`

Еще один тип мьютексов — `timed_mutex`. Он используется, когда есть функция (в данном примере `f`), которая работает с внешними данными.

```

while(true) {
    // Ждем 100 миллисекунд, вдруг мьютекс освободится?
    if(mutex.try_lock_for(timeout) {
        f();
        mutex.unlock();
    } else {
        g(); // f() выполнять нельзя, займемся чем-нибудь другим
    }
}

```

### 9.2 Пример приложения с потоками

Типы взаимодействия между потоками:

**readers/writers** Потоки могут писать, читать, либо делать и то, и другое.

**producers/consumers** Одни потоки производят задания и кладут их в очередь, другие их выполняют.

```

my_queue q;
mutex m;
void push(int i) {
    flag = false;
    while(flag != true) {
        m.lock();
        if (!q.full()) {
            flag = true;
        }
        m.unlock();
    }
    lock_guard l(m);
    q.push(i);
}

void pop() {
    // Тут проверка, что в очереди что-то есть,
    // По аналогии с проверкой на полноту в push
    lock_guard l(m);
    q.pop();
}

int main() {
    vector<consumer> cs;
    vector<producer> ps;
    for(consumer &c: cs) {
        c.init(q); c.start()
    }
    for(producer &p: ps) {
        p.init(q); p.start()
    }
}

```

Получилось не очень удобно. В следующей лекции будут рассмотрены conditional variables, которые позволяют решить это удобно.

## 10 20 мая

### 10.1 Conditional variables

Код с прошлой лекции на новый лад.

```

int get() {
    m.lock();
    while(q.empty()) {
        m.unlock();
        m.lock();
    }
}

```

```

    }
    int s = q.pop();
    m.unlock();
    return s;
}

void put(int s) {
    m.lock();
    while(q.size() == capacity()) {
        m.unlock();
        m.lock();
    }
    q.push(s);
    m.unlock();
}

```

Проблема данного кода в том, что ждать разблокировки таким образом (в цикле) неэффективно. Кроме того, он слегка громоздок. На помощь приходят conditional variables.

```

void put(int s) {
    // Совмещает возможности mutex и lock_guard
    unique_lock<mutex> l(m);
    // Сним, пока не разбудят, проверяем условие
    // Если нет, то спим дальше, иначе работаем
    not_full.wait(l, [this]{return g.size() != capacity});
    q.push();
    // Разбудить один consumer
    not_empty.notify_one();
}

int get() {
    unique_lock<mutex> l(m);
    not_empty.wait(l, [this] { return !g.empty() });
    int s = q.pop();
    not_full.notify_one();
}

```

wait внутри устроен примерно так:

```

while(!condition) {
    wait(l);
}

```

## 10.2 Initializer list

Можно инициализировать большинство структур данных с помощью фигурных скобок. Для того, чтобы добавить поддержку в свой класс, делаем так:

```
vector::vector(initializer_list<T>& l) {
initializer_list<T>::iterator it = l.begin();
    T t = *it;
    // Тут создаем объект
}
```

Компилятор сначала создает массив в статической памяти, затем вызывает метод.

### 10.3 unordered\_set, unordered\_map

```
template<>
struct hash<PhoneEntry> {
    size_t operator()(const PhoneEntry& o) {
        return hash<int>()(num) + hash<string>()(s);
    }
};

class PhoneEntry {
    string s;
    int num;
    bool operator==(const PhoneEntry& o);
    friend class std::hash<PhoneEntry>;
};
```

### 10.4 Множественное наследование

Суть состоит в том, что можно наследоваться от нескольких классов одновременно.

Внимание! Существует распространенное мнение, что данную возможность вообще никогда не стоит использовать.

Существует ромбовидное наследование. В этом случае объекты не всегда можно неявно привести к типу какого-либо предка. В таких неоднозначных местах нужно использовать `static_cast`, либо **виртуальное наследование**.

```
struct A {
    int foo() { return 1; }
};

class B: public virtual A {};
class C : public virtual A {};
class D : public B, public C {};

int main () {
    D d;
    cout << d.foo();
}
```

Если убрать ключевое слово `virtual`, то метод `foo()` не может быть определён однозначно и в результате не будет доступен как объект класса `D` и код не скомпилируется.

11 27 мая

## 11.1 SFINAE

```
vector<int> v = {1, 2, 3, 4};
int n;
show(a); // show(int a); если нет итератора
show(v); // show(vector<T> i); если есть итератор

template<typename T>
struct has_iterator {
    template<typename U>
    static char test(typename U::iterator* x);
    template<typename U>
    static long test(U* x);
    static const value=sizeof(test<T>(0)==1); // test???
};

bool b = has_iterator<int>::value;
```

Не вызывает ошибку, если подстановка шаблонного параметра происходит хотя бы в 1 методе

```
#include <type_traits>

template<class T>
typename enable_if<!has_iterator<T>::value, void>::type;
show (const T& x) {
    cout << x << endl;
}

template<class T>
typename enable_if<has_iterator<T>::value, void>::type
show(const T& x) {
    for(auto& i: x) {
        cout << i << endl;
    }
}
```

Реализация enable\_if:

```
template<bool, typename T>
struct enable_if {
};
template<typename T>
struct enable_if<true, T> {
    typedef T type;
};
```