

# Основы разработки на C++: красный пояс

Неделя 1

Макросы и шаблоны классов



# Оглавление

<b>Макросы и шаблоны классов</b>	<b>2</b>
1.1 Введение в макросы . . . . .	2
1.1.1 Введение в макросы . . . . .	2
1.1.2 Оператор # . . . . .	6
1.1.3 Макросы <code>--FILE--</code> и <code>--LINE--</code> . . . . .	7
1.1.4 Тёмная сторона макросов . . . . .	9
1.2 Шаблоны классов . . . . .	12
1.2.1 Введение в шаблоны классов . . . . .	12
1.2.2 Интеграция пользовательского класса в цикл <code>for</code> . . . . .	14
1.2.3 Разница между шаблоном и классом . . . . .	15
1.2.4 Вывод типов в шаблонах классов . . . . .	16
1.2.5 Автоматический вывод типа, возвращаемого функцией . . . . .	19

# Макросы и шаблоны классов

## 1.1. Введение в макросы

### 1.1.1. Введение в макросы

Первая тема нашего курса – введение в макросы. В курсе «Жёлтый пояс по C++» мы разработали unit test framework для создания юнит-тестов. Кроме того у нас была задача, которая называлась «Тестирование класса Rational», в которой нужно было написать набор юнит-тестов для класса Rational. Этот класс представлял собой рациональное число.

```
class Rational {
public:
    Rational() = default;
    Rational(int nn, int dd);

    int Numerator() const;
    int Denominator() const;

private:
    int n = 0;
    int d = 1;
};
```

Нам нужно было разработать набор юнит-тестов, которые проверяли, что этот класс реализован корректно. Рассмотрим программу, которая тестирует класс Rational с помощью unit test framework'a. В ней есть два теста: `TestDefaultConstructor()`, который проверяет, как работает конструктор по умолчанию в классе Rational, и `TestConstruction()`, в котором есть один тест, который проверяет, как ведет себя класс Rational, когда ему в конструктор передается числитель и знаменатель.

```
void TestDefaultConstructor() {
    const Rational defaultConstructed;
```

```
    AssertEqual(defaultConstrcuted.Numerator(), 0, "Default constructor  
denominator");  
    AssertEqual(defaultConstrcuted.Denominator(), 1, "Default constructor  
denominator");  
}  
  
void TestConstruction() {  
    const Rational r(3, 12);  
    AssertEqual(r.Numerator(), 1, "3/12 numerator");  
    AssertEqual(r.Denominator(), 4, "3/12 denominator");  
}
```

В функцию `main` передаётся объект класса `TestRunner`, запускается два теста с помощью метода `RunTest`, куда передаём тест и текстовое сообщение.

```
int main() {  
    TestRunner tr;  
    tr.RunTest(TestDefaultConstructor, "TestDefaultConstructor");  
    tr.RunTest(TestConstruction, "TestConstruction");  
    return 0;  
}
```

Строковые сообщения мы добавляли в `AssertEqual` для того, чтобы, когда наш `assert` срабатывает, понять, какой именно `assert` сработает. Намеренно допустим в классе `Rational` ошибку (например, вместо знаменателя будем возвращать числитель). С помощью сообщения об ошибке мы можем найти в нашем коде `assert`, который сработал. Нам помогло то, что мы сделали эти сообщения уникальными.

```
int Rational::Denominator() const {  
    return n;  
}  
// TestDefaultConstructor fail: Assertion failed: 0 != 1  
// TestConstruction fail: Assertion failed: 1 != 4 hint: 2 unit tests failed. Terminate
```

Посмотрим, как устроен вызов метода `RunTest` в классе `TestRunner`. Он принимает функцию, которая выполняет тестирование, и строчку, которая совпадает с названием функции. Эта строчка нужна, чтобы формировать сообщения “`TestDefaultConstructor fail`, `TestConstruction fail`”, то есть чтобы в консоль выводить имя теста, который либо прошёл, либо не прошёл. Это неудобно, потому что это дублирование кода.

Что хотим получить:

- Если срабатывает `AssertEqual(x, y)`, на экран выводится `Assertion failed: 2 != 3 hint: x != y. main.cpp:35;`
- Запускать тесты кодом `tr.RunTest(TestConstruction);`
- Если тест проходит, на экран выводится `TestConstruction OK.`

Для того, чтобы решить эту задачу, вспомним, что сборка проекта на C++ состоит из трёх стадий: препроцессинг, компиляция, компоновка. На стадии препроцессинга выполняются директивы `#include`. Содержимое подключаемых файлов копируется в тело компилируемого файла, после этого наступает стадия компиляции. Также на стадии препроцессинга происходит разворачивание макросов. Они объявляются с помощью ключевого слова `#define`.

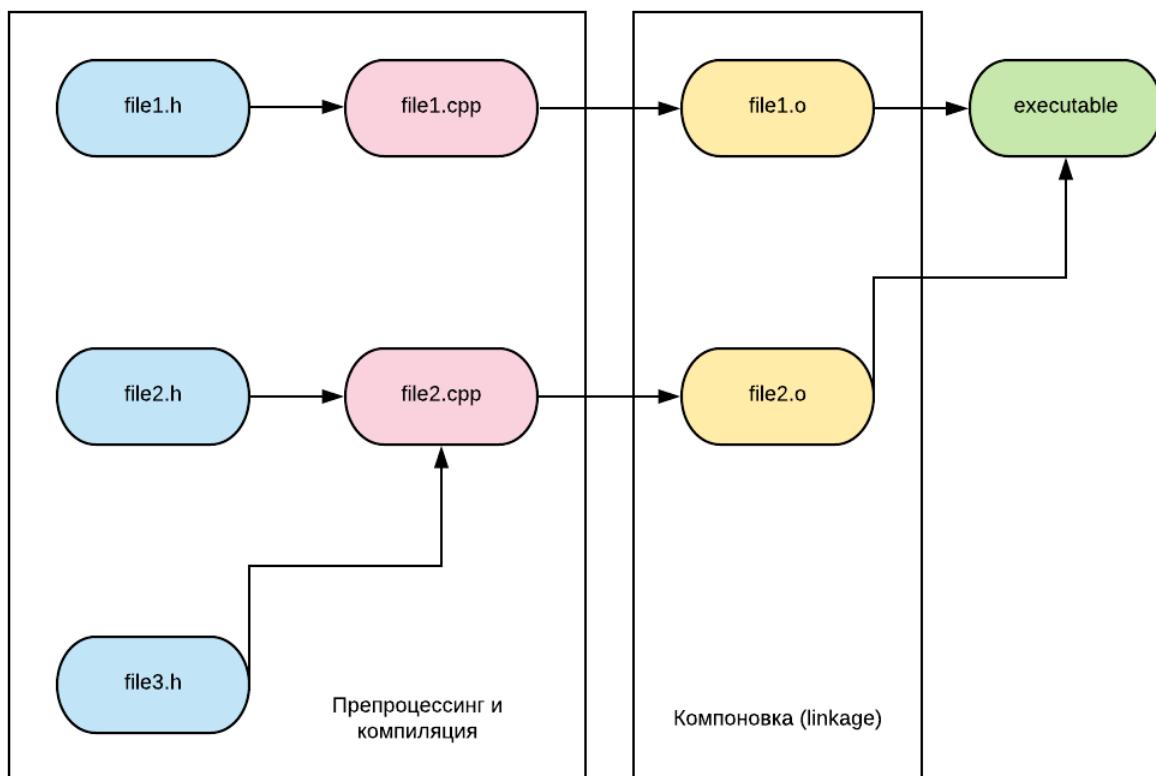


Рис. 1.1: Этапы сборки в C++

Для примера можно объявить макросы:

```
#define MY_MAIN int main()
#define FINISH return 0
```

Исправим функцию

```
int main() {
    return 0;
}
```

на

```
MY_MAIN {
    FINISH;
}
```

Объявленные макросы на этапе препроцессинга будут развёрнуты в свои определения.

Объявим макрос с тремя параметрами ASSERT\_EQUAL, который разворачивается в вызов шаблона AssertEqual с этими тремя параметрами:

```
#define ASSERT_EQUAL(x, y, m) \
    AssertEqual(x, y, m)
```

Можно заменить вызов шаблона на макрос, такая программа скомпилируется.

```
void TestDefaultConstructor() {
    const Rational defaultConstructed;
    ASSERT_EQUAL(defaultConstructed.Numerator(), 0, "Default constructor
denominator");
    ASSERT_EQUAL(defaultConstructed.Denominator(), 1, "Default constructor
denominator");
}
```

Мы с вами познакомились с макросами, узнали, что они создаются с помощью ключевого слова `#define`, и на этапе препроцессинга происходит текстовая замена имени макроса на его содержимое. При этом макросы могут не содержать параметров или содержать один или несколько параметров.

### 1.1.2. Оператор #

Нам хотелось бы избавиться от дублирования функции и её названия в коде (см. стр. 3).

Напишем макрос RUN\_TEST, который принимает на вход параметр `tr` – объект класса `TestRunner`, `func` – функция, содержащая тесты. Этот тест будет у объекта `tr` вызывать метод `RunTest` и передавать туда `func` и ещё раз `func`:

```
#define RUN_TEST(tr, func) \
    tr.RunTest(func, func)
```

Применим макрос в функции `main`.

```
int main() {
    TestRunner tr;
    RUN_TEST(tr, TestDefaultConstructor);
    RUN_TEST(tr, TestConstruction);
    return 0;
}
```

Такой код не скомпилируется, потому что наш макрос получает в качестве второго параметра функцию, хотя этот параметр должен быть строкой.

```
#define RUN_TEST(tr, func) \
    tr.RunTest(func, #func)
```

Поставим перед параметром `func` символ решётки. Такая программа работает. Теперь в качестве второго параметра передается имя функции, обёрнутое в кавычки, то есть строковый литерал. Таким образом, с помощью оператора `#` мы добились того, что когда мы вызываем юнит-тесты, мы можем не дублировать имя функции, а просто передавать эту функцию в макрос и препроцессор за нас в качестве второго параметра передаст имя этой функции.

Другой пример, когда оператор `#` в макросах бывает полезен. Иногда нам нужно что-то логировать, например, для отладки. Допустим, для отладки мы хотим выводить следующие значения в какой-нибудь поток:

```
int x = 4;
string t = "hello";
bool.isTrue = false;
```

Выведем их:

```
cerr << x << " " << t << " " << isTrue << endl;
// 4 hello 0
```

Можно использовать макросы и оператор #, чтобы сделать вывод более понятным:

```
#define AS_KV(x) #x << " = " << x
```

Перепишем код:

```
cerr << boolalpha;
cerr << AS_KV(x) << endl
    << AS_KV(t) << endl
    << AS_KV(isTrue) << endl;
// x = 4
// t = hello
// isTrue = false
```

С помощью макроса и оператора # мы сделали более читаемый отладочный вывод,

Итоги:

- Оператор # вставляет в код строковое представление параметра макроса;
- Макрос RUN\_TEST упрощает запуск тестов и избавляет от дублирования.

### 1.1.3. Макросы \_\_FILE\_\_ и \_\_LINE\_\_

Упростим использование шаблонов Assert и AssertEqual.

Для того, чтобы понять, какой assert сработал, нам достаточно знать название файла и номер строки в этом файле. Эту информацию мы можем получить автоматически. Для этого есть специальные макросы.

Используем макросы \_\_FILE\_\_ и \_\_LINE\_\_.

```
const string file = __FILE__;
const int line = __LINE__;
```

На стадии препроцессинга `__FILE__` раскроется в название файла (в нашем случае это будет `macro_intro.cpp`). `__LINE__` раскроется в номер строки, в котором был объявлен (в нашем случае 14, поскольку макрос был объявлен на 14 строке).

Воспользуемся этими макросами, чтобы сформировать уникальное сообщение для `assert`'а. Чтобы создавать многострочные макросы, нужно каждую строчку кроме последней (с закрывающейся скобкой) завершать нисходящим слэшом.

```
#define ASSERT_EQUAL(x, y) { \
    ostringstream os; \
    os << __FILE__ << ":" << __LINE__; \
    AssertEqual(x, y, os.str()); \
}
```

Теперь макрос стал от двух параметров, потому что сообщение генерируется автоматически. Воспользуемся им:

```
void TestDefaultConstructor() { \
    const Rational defaultConstructed; \
    ASSERT_EQUAL(defaultConstructed.Numerator(), 0); \
    ASSERT_EQUAL(defaultConstructed.Denominator(), 1); \
} \
// TestDefaultConstructor fail: Assertion failed: 0 != 1 hint: ..\src\macro_intro.cpp:19
```

В поле `hint` фреймворк выводит имя файла и строку, в котором вызван `assert`. Усовершенствуем выводимое сообщение:

```
os << #x << " != " << #y << ", " \
    << __FILE__ << ":" << __LINE__;
```

В поле `hint` получаем сообщение:

```
// hint: defaultConstructed.Denominator() != 1, ..\src\macro_intro.cpp:20
```

Осталось перенести макрос в файл `test_runner.h`. Добавим туда также макросы `ASSERT` и `RUN_TEST`.

```
#define ASSERT(x) { \
    ostringstream os; \
    os << #x << " is false, " \
        << __FILE__ << ":" << __LINE__; \
    AssertEqual(x, y, os.str()); \
}
```

#### 1.1.4. Тёмная сторона макросов

Макросы позволили сделать наш код короче и проще в использовании. Но вы могли слышать рекомендации, что макросы в C++ – это зло и что никогда нельзя использовать их в своих программах. Да, действительно, при чрезмерном их использовании могут возникать проблемы. Рассмотрим пример:

```
#define MAX (a, b) a > b ? a : b // находит максимум из двух своих аргументов

int main() {
    int x = 4;
    int y = 2;
    int z = MAX(x, y) + 5;
    cout << z;
}
// 4
```

Мы ожидаем, что на экран будет выведено 9, однако получаем 4. Посмотрим, во что раскрывается наш макрос.

```
int z = x > y ? x : y + 5;
```

Если  $x > y$ , то в переменную  $z$  записывается значение  $x$ , если это не так, то в  $z$  запишется  $y + 5$ . Чтобы макрос работал правильно, можно обернуть его в скобки.

```
#define MAX(a, b) (a > b ? a : b)
```

Однако в данном случае гораздо лучше использовать функцию `max` из библиотеки алгоритмов.

Рассмотрим более реальный пример. В стандартной библиотеке нет функции, которая возводит свой аргумент в квадрат. Реализуем макрос, учтём прошлые ошибки и сразу обернём его в скобки.

```
#define SQR(x) (x * x)
```

Реализуем следующий код:

```
int main() {
    int x = 3;
    int z = SQR(x + 1);
    cout << z;
}
```

```
// 7
```

В консоли мы ожидаем увидеть 16, но видим 7. Посмотрим вывод препроцессора, чтобы узнать, в какое выражение раскрылся макрос.

```
int z = (x + 1 * x + 1);
```

Ошибка можно быстро исправить, если обернуть x в скобки.

```
#define SQR(x) ((x) * (x))
```

Вместо этого макроса лучше написать шаблон.

```
template <typename T>
T Sqr(T x) {
    return x * x;
}
```

Такой шаблон прекрасно справляется с задачей возведения в квадрат, при этом мы можем не бояться забыть обернуть макрос и аргументы в скобки.

Напишем функцию LogAndReturn и передадим её в качестве параметра в макрос SQR. Мы ожидаем, что в консоли выведется x = 3, а потом выведется 9.

```
int LogAndReturn(int x) {
    cout << "x = " << x << endl;
    return x;
}

int main() {
    int z = SQR(LogAndReturn(3));
    cout << z;
}
// x = 3
// x = 3
// 9
```

Мы не ожидали, что функция LogAndReturn выполнится дважды. Посмотрим результаты препроцессирования.

```
int main() {
    int z = ((LogAndReturn(3)) * (LogAndReturn(3)));
```

```
    cout << z;  
}
```

Макрос выполнил прямую текстовую замену и вызов функции `LogAndReturn` добавился в код дважды. Это учебный пример. Если бы функция в реальном коде выполняла сложные, долгие вычисления, то мы на ровном месте могли бы получить просадку производительности из-за неудачного использования макроса.

Сохраним результат вызова функции в переменную `x` и передадим её в макрос:

```
int main() {  
    x = LogAndReturn(3);  
    int z = SQR(x);  
    cout << z;  
}  
  
// x = 3  
// 9
```

Всё будет работать нормально. Допустим, далее нам понадобится переменная `x`, увеличенная на 1. Ради экономии места увеличим её прямо на месте.

```
int main() {  
    x = LogAndReturn(3);  
    int z = SQR(x++);  
    cout << z;;  
}  
  
// x = 3  
// 12
```

Вместо ожидаемой 9 получили 12. В результатах препроцессирования видим:

```
int z = ((x++) * (x++));
```

Когда одну и ту же переменную мы изменяем несколько раз в одном и том же выражении, то результат не определён.

Если вместо макроса можно написать функцию или шаблон, то именно так и нужно сделать. Так вы защититесь от неожиданных ошибок. Следовательно, если макрос не использует `_FILE_`, `_LINE_` или оператор `#`, подумайте, можно ли обойтись без него. Если вы всё же пишете макрос, то старайтесь использовать каждый аргумент только один раз, максимально изолируйте аргументы с помощью скобок.

## 1.2. Шаблоны классов

В курсе «Жёлтый пояс по C++» мы изучили шаблоны функций. Они позволяют избежать дублирования кода в функциях, который отличаются типами своих аргументов или типом возвращаемого значения. В этом модуле мы изучим шаблоны классов. Они решают ту же самую задачу: позволяют избежать дублирования кода в классах, которые отличаются только типами своих полей, или типами параметров своих методов, или типами возвращаемых значений в методах.

### 1.2.1. Введение в шаблоны классов

Простейший пример шаблона класса – это пара. Забудем на время, что существует стандартная пара. Напишем соответствующую структуру.

```
struct PairOfStringAndInt {  
    string first;  
    int second;  
};
```

Воспользуемся этой парой.

```
int main() {  
    PairOfStringAndInt si;  
    si.first = "Hello";  
    si.second = 5;  
}
```

Потом у нас как-то проект развивается, мы пишем код дальше и понимаем, что нам нужна ещё пара, из логического значения и символа.

```
struct PairOfBoolAndChar {  
    bool first;  
    char second;  
};
```

Добавим в `main`:

```
int main() {  
    PairOfStringAndInt si;
```

```
    si.first = "Hello";
    si.second = 5;

    PairOfBoolAndChar bc;
    bc.first = true;
    bc.second = 'z';
}
```

Понятно, что каждый раз, когда у нас возникает необходимость в новых сочетаниях типов, нам приходится объявлять новую структуру. Мы видим, что классы `PairOfStringAndInt` и `PairOfBoolAndChar` структурно одинаковы, но отличаются типами своих полей. Вместо них мы можем написать шаблон класса.

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};
```

Мы написали простейший шаблон класса «пара». Воспользуемся им.

```
int main() {
    Pair<string, int> si;
    si.first = "Hello";
    si.second = 5;

    Pair<bool, char> bc;
    bc.first = true;
    bc.second = 'z';
}
```

Важно отметить, что `Pair` – это шаблон класса, а `Pair<bool, char>` – это уже класс, самостоятельный тип. Таким образом, в этой программе мы создаем из шаблона класса два класса. Создание типа из шаблона класса называется **инстанцированием**.

### 1.2.2. Интеграция пользовательского класса в цикл for

Рассмотрим пример. У нас есть вектор целых чисел, по которому можно итерироваться, используя цикл `range-based for`. Если мы хотим проитерироваться по первым трем элементам вектора, то нам приходится использовать обычный цикл со счетчиком. В этом нет универсальности, к тому же если размер вектора меньше 3, то цикл может выйти за пределы вектора и программа будет вести себя не так, как мы ожидаем. Давайте напишем функцию `Head`.

```
int main() {
    vector<int> v = {1, 2, 3, 4, 5};
    for (int x : Head(v, 3)) {
        cout << x << " ";
    }
}
```

Эта запись означает, что с помощью удобного цикла `range-based for` мы хотим проитерироваться по первым трем элементам вектора. Кроме того, эта функция должна корректно работать, и когда в векторе меньше чем три элемента. Мы напишем шаблон функции в данном случае.

```
template <typename T>
vector<T> Head(vector<T>& v, size_t top) {
    return {
        v.begin(),
        next(v.begin(), min(top, v.size()))
    };
}
```

Функция принимает на вход вектор `v` по ссылке, параметр `top` задаёт размер префикса, по которому мы хотим проитерироваться. Функция возвращает два итератора: начало вектора и `begin`, который с помощью оператора `next` мы продвинули на минимум из значения параметра `top` и размера вектора.

Функция `Head` создает копию вектора. Это не практично. Более того, в текущей реализации функции мы не можем изменять элементы изначального вектора.

И нам нужен какой-то другой способ, который бы позволил нам из функции `Head` вернуть диапазон внутри исходного вектора, и с помощью этого диапазона обращаться к элементам самого вектора. Например, из функции `Head` вместо копии вектора возвращать лишь пару итераторов на вектор `v` и итерироваться в цикле по ним. Это возможно сделать как раз с помощью шаблонов классов, с которыми мы с вами познакомились в предыдущем уроке.

```
template <typename Iterator>
struct IteratorRange {
    Iterator first, last;
};
```

Применим этот шаблон класса внутри функции Head.

```
template <typename T>
IteratorRange<typename vector<T>::iterator> Head(vector<T>& v, size_t top) {
    return {
        v.begin(), next(v.begin(), min(top, v.size()))
    };
}
```

В текущем виде код не скомпилируется, поскольку у структуры `IteratorRange` нет методов `begin` и `end`. Давайте их добавим.

```
Iterator begin() const {
    return first;
}
Iterator end() const {
    return last;
}
```

Чтобы по объекту класса можно было проитерироваться с помощью цикла `for`, он должен иметь методы `begin()` и `end()`. Методы `begin()` и `end()` должны возвращать итераторы.

### 1.2.3. Разница между шаблоном и классом

Сам по себе `IteratorRange` не является классом, это шаблон класса. В него необходимо подставить конкретный тип, чтобы создать класс. `IteratorRange` у нас параметризован типом итератора, а чтобы создать из него класс, в него нужно подставить какой-то конкретный тип итератора, например, итератор вектора целых чисел.

```
IteratorRange<vector<int>::iterator>
```

Все стандартные контейнеры, которыми мы пользовались, например, `vector`, `map`, `set` являются шаблонами классов.

Допустим, мы хотим посчитать, сколько элементов у нас находится в диапазоне двух итераторов. Мы не можем оформить функцию вот так:

```
size_t RangeSize(IteratorRange r) {
    return r.end() - r.begin();
}
```

Дело в том, что параметр `r` должен иметь тип `IteratorRange` – не тип, это шаблон типа. Чтобы создать из этого шаблона тип, его нужно инстанцировать.

```
template <typename T>
size_t RangeSize(IteratorRange<T> r) {
    return r.end() - r.begin();
}
```

#### 1.2.4. Вывод типов в шаблонах классов

Допустим, мы хотим обратиться к суффиксу вектора – его второй половине.

```
IteratorRange<vector<int>::iterator> second_half {
    v.begin() + v.size() / 2, v.end()
};
```

Чтобы объявить переменную `second_half`, нам пришлось написать достаточно громоздкую конструкцию. Напишем так называемую **порождающую функцию**.

```
template <typename Iterator>
IteratorRange<Iterator> MakeRange(Iterator begin, Iterator end) {
    return IteratorRange<Iterator>(begin, end);
}
```

Теперь мы можем лаконично объявить переменную `second_half`.

```
auto second_half = MakeRange {
    v.begin() + v.size() / 2, v.end()
};
```

Порождающие функции позволяют возложить на компилятор выведение шаблонных типов при инстанцировании шаблонных классов. Они сокращают код и избавляют от необходимости много

печатать. Однако для каждого шаблона класса порождающую функцию приходится писать самостоятельно. Кроме того, из записи `auto full = MakeRange(t.begin(), t.end())` неочевиден тип переменной `full`. Необходимо отдельно проверить, что возвращает функция `MakeRange`.

Порождающие функции – это не единственный способ возложить на компилятор вывод шаблонных типов при инстанцировании шаблонов класса. Другой способ появился в стандарте C++17. Чтобы им воспользоваться, необходимо настроить вашу среду разработки в соответствии с этим стандартом.

Необходимо убедиться, что у вас стоит компилятор GCC версии не младше седьмой. В самой IDE необходимо убедиться, что проект собирается с использованием самого свежего стандарта.

Если в классе есть конструктор, позволяющий определить тип шаблона, компилятор выводит его сам. Рассмотрим пример:

```
template <typename T> struct Widget {
    Widget(T value);
};

Widget w_int(5);
```

У нас есть шаблон класса `Widget`, в котором есть конструктор, принимающий значение `value` типа `T`. Компилятор по этому конструктору может сам вывести тип. Мы можем объявить переменную `w_int`, проинициализировать её значением 5. При этом в качестве её типа мы просто пишем `Widget`. Компилятор берёт 5 и смотрит, какие конструкторы есть в шаблоне `Widget`. Видит конструктор, принимающий `value` типа `T`. Он понимает, что 5 имеет тип `int`, поэтому мы хотим инстанцировать шаблон типа `Widget` с помощью типа `int`, и он создаёт класс `Widget<int>`.

Рассмотрим другой пример:

```
pair<int, bool> p(t, true);
```

Из документации мы знаем, что у шаблона класса `pair` есть конструктор, который принимает параметры его шаблонных аргументов, поэтому компилятор может воспользоваться конструктором и вывести типа. Код мы можем переписать следующим образом:

```
pair p(5, true);
```

Код скомпилируется, поскольку по 5 и `true` компилятор поймёт, что нам нужно из шаблона `pair` создать класс `pair<int, bool>`.

Переделаем структуру `IteratorRange` в класс, добавим туда конструктор:

```
class IteratorRange {
private:
    Iterator first, last;

public:
    IteratorRange(Iterator f, Iterator l)
        : first(f)
        , last(l)
    {
    }
    Iterator begin() const {
        return first;
    }
    Iterator end() const {
        return last;
    }
}
```

Тогда пример с `second_half` мы можем переписать следующим образом:

```
IteratorRange second_half(
    v.begin() + v.size() / 2, v.end()
)
```

Компилятор видит, что мы создаём объект класса с помощью двух аргументов типа `vector<int>::iterator`, он понимает, благодаря конструктору, что мы должны инстанцировать шаблон `IteratorRange` с помощью типа `vector<int>::iterator`. Таким образом он создаёт объект класса `IteratorRange` от `vector<int>::iterator`.

Способ выводения типов с помощью конструктора обладает преимуществами:

- не всегда нужно писать дополнительный код;
- `IteratorRange full(t.begin(), t.end())` – проще понять, какой тип у `full`.

Из следующего кода может показаться, что `r_i` и `r_s` имеют один и тот же тип, потому что перед ними стоит `IteratorRange`.

```
vector<int> ints;
vector<string> strs;
IteratorRange r_i(begin(ints), end(ints));
IteratorRange r_s(begin(strs), end(strs));
```

По умолчанию при инстанцировании стоит явно указывать шаблонный тип. Если это не удобно, то используем способ вывода через конструктор, потому что в нём явно указано имя шаблона. Если по какой-то причине мы не можем использовать этот способ, то делаем порождающую функцию.

### 1.2.5. Автоматический вывод типа, возвращаемого функцией

Вернемся к функции `Head`. Сейчас она работает только для вектора. Перепишем её так, чтобы она позволяла итерироваться по префиксному произвольного контейнера.

```
template <typename Container>
IteratorRange<???> Head(Container v, size_t top) {
    return {
        v.begin(), next(v.begin(), min(top, v.size()))
    };
}
```

Возникает вопрос: что нам написать при инстанцировании шаблона `IteratorRange`? Можем написать `typename Container::iterator`. Воспользуемся функцией `Head` для вывода четырёх минимальных элементов множества:

```
set<int> nums = {5, 7, 12, 8, 10, 5, 6, 1};
for (int x : Head(nums, 4)) {
    cout << x << ' ';
}
// 1 5 6 7
```

Для `deque<int>` код также работает правильно, однако для `const deque<int>` код не скомпилируется. Дело в том, что у `const deque` метод `begin` возвращает `const_iterator`, который не разрешает изменять элементы вектора. Нам нужно уметь выбирать между константным итератором для константных объектов и неконстантным итератором для неконстантных объектов. Напишем `auto` в качестве типа возвращаемого значения функции `Head`. Таким образом мы укажем компилятору взять возвращаемый тип из команды `return`, то есть `IteratorRange`.

```
template <typename Container>
auto Head(Container v, size_t top) {
    return IteratorRange{
        v.begin(), next(v.begin(), min(top, v.size())))
    };
}
```

Такой код работает для константного `deque`. Кроме того, работает пример с модификацией вектора `v` с помощью функции `Head`. Теперь компилятор сам выводит тип итератора, с которым нужно инстанцировать `IteratorRange`.

По умолчанию следует явно указывать тип результата функции. Использовать `auto` в качестве типа результата функции стоит только если:

- тип результата громоздкий;
- тело функции очень короткое.

В противном случае может пострадать понятность кода.

# Основы разработки на C++: красный пояс

Неделя 2

Принципы оптимизации кода, сложность алгоритмов и эффективное  
использование ввода/вывода



# Оглавление

<b>Принципы оптимизации кода, сложность алгоритмов и эффективное использование ввода/вывода</b>	<b>2</b>
2.1 Принципы оптимизации кода . . . . .	2
2.1.1 Первое правило оптимизации кода . . . . .	2
2.1.2 Второе правило оптимизации кода . . . . .	2
2.1.3 Разработка своего профайлера . . . . .	4
2.1.4 Совершенствование своего профайлера . . . . .	6
2.2 Эффективное использование потоков ввода/вывода . . . . .	8
2.2.1 Буферизация в выходных потоках . . . . .	8
2.2.2 Когда нужно использовать endl, а когда – '\n' . . . . .	9
2.2.3 Связанность потоков . . . . .	10
2.3 Сложность алгоритмов . . . . .	12
2.3.1 Введение . . . . .	13
2.3.2 Оценка сложности . . . . .	15
2.3.3 Практические применения . . . . .	16
2.3.4 Амортизированная сложность . . . . .	20

# Принципы оптимизации кода, сложность алгоритмов и эффективное использование ввода/вывода

## 2.1. Принципы оптимизации кода

### 2.1.1. Первое правило оптимизации кода

**Избегайте преждевременной оптимизации.** Преждевременная оптимизация – это упор на производительность в ущерб простоте, дизайну, понятности, поддерживаемости и так далее. В соответствии с принципом Парето 80% работы программы тратится на исполнение 20% кода. Эти 20% и нужно оптимизировать, но для начала их нужно найти. Делать максимально быстрым весь код тяжело, время на это будет уходить впустую. Преждевременная оптимизация приводит к необоснованному усложнению кода, затруднению поддержки и замедлению разработки. Как правило код надо стараться сделать правильным, простым, а только потом быстрым. Это не значит, что о производительности вообще не надо думать.

### 2.1.2. Второе правило оптимизации кода

Код не может быть просто быстрым или медленным. Он может быть достаточно или не достаточно быстрым для задачи, которую он решает. Рассмотрим пример.

```
vector<string> GenerateBigVector() {
    vector<string> result;
    for (int i = 0; i < 28000; ++i) {
        result.insert(begin(result), to_string(i));
    }
    return result;
```

```
}

int main() {
    cout << GenerateBigVector().size() << endl;
    return 0;
}
```

Функция `GenerateBigVector` генерирует вектор из 28000 элементов, программа выводит размер этого вектора. Замерим время работы программы. Программа работает 4.337 секунды. Понятно, что для данной задачи это медленно.

Мы можем бороться с проблемой интуитивно. Создадим версию `GenerateBigVector`, которая будет принимать вектор по ссылке, тогда не возникнет лишнего копирования и предположительно программа станет работать быстрее. Сделаем вектор из 32000 элементов.

```
void GenerateBigVector(vector<string>& result) {
    for (int i = 0; i < 32000; ++i) {
        result.insert(begin(result), to_string(i));
    }
}
```

Запустим программу с такой версией функции `GenerateBigVector` и замерим время работы. Программа отработала за 5.638 секунды. Это говорит о том, что интуиция нас подвела. Таким образом, мы приходим ко второму правилу оптимизации, которое звучит просто: **замеряйте!**

Интуиция часто даёт неверные результаты, а измерения объективны, они позволяют найти то место программы, из-за которой она работает медленно.

Замерим, сколько работает программа, а также замерим, сколько работает цикл по формированию вектора.

```
vector<string> GenerateBigVector() {
    vector<string> result;
    auto start = steady_clock::now();
    for (int i = 0; i < 28000; ++i) {
        result.insert(begin(result), to_string(i));
    }
    auto finish = steady_clock::now();
    auto duration = finish - start;
    cerr << "Cycle: "
         << duration_cast<milliseconds>(duration).count()
```

```
    << endl;
    return result;
}

int main() {
    auto start = steady_clock::now();
    cout << GenerateBigVector().size() << endl;
    auto finish = steady_clock::now();
    auto duration = finish - start;
    cerr << "Total: "
        << duration_cast<milliseconds>(duration).count()
        << endl;
    return 0;
}
```

Замерив время, мы понимаем, что общее время работы программы и время формирования вектора совпадает. Большая часть времени уходит на работу в цикле.

```
result.insert(begin(result), to_string(i));
```

Здесь мы вставляем каждый следующий элемент в начало вектора. Чтобы вставить элемент в начало, нужно сдвинуть текущее содержимое вектора вправо. Изменим программу и будем вставлять каждый новый элемент не в начало вектора, а в конец.

```
result.push_back(to_string(i));
```

Такая программа отработает всего за 6 миллисекунд.

Прежде чем ускорять код, замерьте, сколько он работает. Если это недостаточно быстро, то начинайте искать узкие места. Интуиция не работает – замеряйте!

### 2.1.3. Разработка своего профайлера

На практике для нахождения медленно работающего места в программе используются специальные инструменты, которые называются профайлерами. Некоторые из них:

- gperftools (Linux, Mac OS);

- perf (Linux);
- VTune (Windows, Linux).

Мы сделаем свой простой профайлер и будем им пользоваться.

Замерять время работы с помощью способа из предыдущего пункта громоздко. Напишем специальный класс.

```
class LogDuration {
public:
    LogDuration()
        : start(steady_clock::now())
    {
    }

    ~LogDuration() {
        auto finish = steady_clock::now();
        auto dur = finish - start;
        cerr << duration_cast<milliseconds>(dur).count()
            << " ms" << endl;
    }
private:
    steady_clock::time_point start;
};
```

`start` – это момент начала измерений. В конструкторе мы записываем в это поле `steady_clock::now()`. В деструкторе мы запоминаем момент окончания работы. Вычисляем длительность работы (разность). Выводим длительность в миллисекундах.

Теперь для измерения длительности работы блока кода мы пишем перед ним `LogDuration input;` и обрамляем конструкцию в фигурные скобки:

```
{
    LogDuration input;
    for (int i = 0; i < element_count; ++i) {
        int x;
        cin >> x;
        elements.insert(x);
    }
}
```

Улучшим читаемость результатов работы программы.

```
class LogDuration {
public:
    explicit LogDuration(const string& msg = "") 
        : message(msg + ": ")
        , start(steady_clock::now())
    {}

    ~LogDuration() {
        auto finish = steady_clock::now();
        auto dur = finish - start;
        cerr << message
            << duration_cast<milliseconds>(dur).count()
            << " ms" << endl;
    }

private:
    string message;
    steady_clock::time_point start;
};
```

Теперь мы можем добавлять сообщение, например:

```
{
    LogDuration input("Input");
    for (int i = 0; i < element_count; ++i) {
        int x;
        cin >> x;
        elements.insert(x);
    }
}
// Input: 113 ms
```

## 2.1.4. Совершенствование своего профайлера

Посмотрим, как мы пользуемся классом LogDuration.

```
{  
    LogDuration input("Input");  
    ...  
}
```

Мы объявляем переменную с именем `input` и в качестве сообщения тоже используем "Input". Имена переменных нам не нужны, мы не обращаемся к ним в коде. Нам нужен способ, который позволит объявлять объекты класса `LogDuration`, но избавит от необходимости придумывать имя для переменных. Тут нам помогут макросы.

```
#define LOG_DURATION(message) \  
    LogDuration UNIQ_ID(__LINE__){message};
```

Макрос `UNIQ_ID` имеет следующее устройство (не будем останавливаться на том, как оно получилось):

```
#define UNIQ_ID_IMPL(lineno) _a_local_var_##lineno  
#define UNIQ_ID(lineno) UNIQ_ID_IMPL(lineno)
```

Просто поверьте на слово: конструкция `UNIQ_ID(__LINE__)` позволяет объявить уникальный идентификатор в пределах заданного .cpp-файла.

Теперь мы можем замерять время с помощью конструкции

```
{  
    LOG_DURATION("Input");  
    ...  
}
```

Наша программа работает. Ввод длится 102 миллисекунды, обработка запросов – 76, вся программа – 181. При этом мы добились желаемого удобства. Нам больше не нужно придумывать имя для объекта, мы просто пишем `LOG_DURATION` и сообщение и получаем в стандартный поток ошибок длительность работы соответствующего блока `count`. Это действительно удобно.

Допустим, мы захотим померить, сколько работает одна итерация нашего цикла. Мы просто пишем `LOG_DURATION("Iter " + to_string(i))`.

Остался один штрих, чтобы сделать класс `LogDuration` переиспользуемым. Давайте его вынесем в отдельный файл. Заведем заголовочный файл в нашем проекте. Назовем его `profile.h` и вынесем сюда класс `LogDuration` и необходимые `include`'ы. Это `#include <chrono>, using`

namespace std;; using namespace std::chrono;. И так как мы выводим в cerr, нам понадобится ещё <iostream>.

Подведём итоги. Мы смогли разработать класс LogDuration, который можно использовать для замера времени работы программы и отдельных её блоков, а также сделали специальный макрос LOG\_DURATION, который нас избавляет от придумывания ненужных идентификаторов и упрощает работу с этим профайлером.

## 2.2. Эффективное использование потоков ввода/вывода

В этом модуле мы посмотрим, какие проблемы могут возникнуть при использовании потоков ввода/вывода и как эти проблемы решаются.

### 2.2.1. Буферизация в выходных потоках

Файловый поток ofstream не сразу пишет выводимые в него данные в файл, а накапливает их в промежуточном буфере и сбрасывает его в файл, когда он наполнился. Поток вывода cout ведет себя точно так же. Манипулятор endl не только выводит перевод строки, но и сбрасывает буфер потока в файл.

Сравним производительность потоков. Будем измерять, за какое время выведутся в файл 15000 строк при использовании endl и при использовании '\n'. Будем использовать профайлер profile.h.

```
int main() {
{
    LOG_DURATION("endl");
    ofstream out("output.txt");
    for (int i = 0; i < 15000; ++i) {
        out << "London is the capital of Great Britain. "
             << "I am travelling down the river"
             << endl;
    }
}
{
    LOG_DURATION("'\\n'");
    ofstream out("output2.txt");
```

```
for (int i = 0; i < 15000; ++i) {
    out << "London is the capital of Great Britain. "
        << "I am travelling down the river"
        << '\n';
}
}
}

// endl: 137 ms
// '\n': 16 ms
```

Пусть теперь выводится в 10 раз больше строчек, снова выполним наш код, и увидим поразительную огромную разницу: `endl` – 530 миллисекунд, `'\n'` – 168 миллисекунд.

Тот факт, что `endl` сбрасывает буфер потока, имеет значительное влияние на производительность. Дело в том, что при использовании `endl` мы пишем в файл каждый раз, а при использовании `'\n'` – только когда буфер заполнился.

Таким образом, использование `endl` может приводить к снижению скорости вывода в файл или `cout`.

## 2.2.2. Когда нужно использовать `endl`, а когда – `'\n'`

Возникает вопрос: зачем использовать `endl`? Он нужен в ситуациях, когда нам не важна скорость вывода всей информации, а важно увидеть последнее выведенное сообщение, как только оно было выведено в выходной поток. Самый простой пример – отладка программы с использованием отладочного вывода в консоль. Использовать буферизованный вывод в данном случае неудобно, потому что вероятна ситуация, когда вы выводите отладочное сообщение, а на следующей команде программа падает. Таким образом, есть вероятность, что это сообщение не будет выведено в консоль, хотя оно может содержать важную информацию об ошибке.

Рассмотрим пример с классом `LogDuration`. Эта программа состоит из двух функций. Тела функций `CouldBeSlowOne` и `CouldBeSlowTwo` скрыты, потому что сейчас они не важны. Одна из них работает медленно, мы не знаем, какая. Чтобы узнать это, мы оборачиваем их в `LOG_DURATION`.

```
int main() {
{
    LOG_DURATION("One");
}
```

```
    CouldBeSlowOne();
}
{
    LOG_DURATION("Two");
    CouldBeSlowTwo();
}
}

// One: 9 ms
// Two: 7093 ms
```

Если мы реализуем `LOG_DURATION` с помощью '`\n`', то оба сообщения выводятся в конце работы программы, ждать придётся долго. Если же в реализации использовать `endl`, то первое сообщение выводится сразу после завершения работы первой функции. Это удобнее, мы можем не ждать выполнения второй функции.

Теперь давайте посмотрим на `TestRunner` из предыдущего курса (курс [«Основы разработки на C++: жёлтый пояс»](#)), давайте посмотрим на `class TestRunner` и его метод `RunTest`. `endl` здесь используется во всех сообщениях. Когда тест выполнился успешно, выводится `OK` и `endl`. И когда случилось падение теста, мы тоже выводим его имя, говорим, что он упал, сообщение из исключения и `endl`.

Если здесь заменить `endl` на символ перевода строки, то мы не будем видеть сразу результат выполнения тестов. Конечно, когда у нас простые юнит-тесты, они все отрабатывают очень быстро, и нам не важно, что эти сообщения буферизируются и потом выводятся целиком. Но если у нас есть юнит-тесты, которые занимают заметное время, то нам лучше сразу же получать информацию о том, прошли они или нет.

Перевод строки можно использовать, когда вам важна скорость вывода. Например, если вы пишете консольное приложение, которое превращает данные из стандартного ввода в какие-то другие данные и выводит их в стандартный вывод.

### 2.2.3. Связанность потоков

Итак, теперь мы с вами знаем, что замена `endl` на символ перевода строки может ускорять вывод программ на C++. Рассмотрим пример:

```
int main() {
    for (int i = 0; i < 100000; ++i) {
```

```
    int x;
    cin >> x;
    cout << x << endl;
}
}
```

В цикле 100000 раз из стандартного ввода считывается число и 100000 раз записывается в стандартный вывод. Логично ожидать, что при замене `endl` на '`\n`' программа заработает быстрее. Однако стоит проверить это и сделать измерения.

```
int main() {
{
    LOG_DURATION("endl");
    for (int i = 0; i < 100000; ++i) {
        int x;
        cin >> x;
        cout << x << endl;
    }
}
{
    LOG_DURATION("'\n'");
    for (int i = 0; i < 100000; ++i) {
        int x;
        cin >> x;
        cout << x << endl;
    }
}
// endl: 387 ms
// '\n': 373 ms
```

Реализация с '`\n`' работает немного быстрее, однако мы ожидали, что она отработает в разы быстрее реализации с `endl`. Рассмотрим другой пример:

```
cout << "Enter two integers: ";
for (;;) {
}
//
```

Данный код не выведет на экран ничего, потому что передаваемое сообщение попало в буфер и остаётся там на протяжении бесконечного цикла.

```
cout << "Enter two integers: ";
int x, y;
cin >> x >> y;
for (;;) {
}
// Enter two integers:
```

Такой код выводит `Enter two integers: .` Получается, произошёл сброс внутреннего буфера `cout`. Этот пример нам показывает, что по умолчанию поток `cout` связан с потоком `cin`. Это специально сделано для интерактивных приложений вроде нашего калькулятора. Этим же объясняется неожиданно медленная работа реализации с '`\n`' из прошлого примера. Мы можем разорвать связь между `cout` и `cin`:

```
cin.tie(nullptr);
cout << "Enter two integers: ";
int x, y;
cin >> x >> y;
for (;;) {
}
```

Что такое `nullptr`, будет рассказано далее в модуле «Модели памяти».

Ещё больше ускорить работу программ можно, если прописать строку:

```
ios_base::sync_with_stdio(false);
```

Что это за команда? Это выходит за рамки нашего курса. Важно помнить, что вызывать её надо до первой операции ввода/вывода.

## 2.3. Сложность алгоритмов

Мы научились замерять время работы кода. В частности, сравнивать два алгоритма по времени работы. Уже в предыдущем курсе возникали ситуации, когда один алгоритм заведомо эффективнее другого. Например, мы проходили контейнер `dequeue`. Чем он был лучше, чем вектор?

### 2.3.1. Введение

Рассмотрим пример. Попробуем вставлять элементы в начало вектора и в начало `deque`. Вспомним, что вставлять в начало вектора гораздо менее эффективно, чем вставлять в начало `deque`.

```
{
    LOG_DURATION("vector");
    vector<int> v;
    for (int i = 0; i < 100000; ++i) {
        v.insert(begin(v), i);
    }
}
{
    LOG_DURATION("deque");
    deque<int> v;
    for (int i = 0; i < 100000; ++i) {
        v.insert(begin(v), i);
    }
}
// vector: 1661 ms
// deque: 9 ms
```

Рассмотрим ещё один пример. Продемонстрируем, что метод `lower_bound` эффективнее, чем одноимённая функция на примере множества.

```
set<int> numbers;
for (int i = 0; i < 3000000; ++i) {
    numbers.insert(i);
}
const int x = 1000000;

{
    LOG_DURATION("global_lower_bound");
    cout << *lower_bound(begin(numbers), end(numbers), x);
}
{
    LOG_DURATION("lower_bound_method");
    cout << *numbers.lower_bound(x);
}
// global lower_bound: 944 ms
// lower_bound method: 0 ms
```

Рассмотрим третий пример. Сравним функции `lower_bound` и функции `find_if`. Функция `lower_bound` делает бинарный поиск, в то время как функция `find_if` запускает цикл `for`. `lower_bound` заведомо эффективнее, проверим это.

```
const int NUMBER_COUNT = 1000000;
const int NUMBER = 7654321;
const int QUERY_COUNT = 10;

int main() {
    vector<int> v;
    for (int i = 0; i < NUMBER_COUNT; ++i) {
        v.push_back(i * 10);
    }

    {
        LOG_DURATION("lower_bound");
        for (int i = 0; i < QUERY_COUNT; ++i) {
            lower_bound(begin(v), end(v), NUMBER);
        }
    }
    {
        LOG_DURATION("find_if");
        for (int i = 0; i < QUERY_COUNT; ++i) {
            find_if(begin(v), end(v)),
                [NUMBER](int y) { return y >= NUMBER});
        }
    }
    return 0;
}
// lower_bound: 0 ms
// find_if: 123 ms
```

`lower_bound` быстрее, об этом мы и так заранее знали.

Чтобы понять, будет ли эффективен алгоритм, не обязательно его программировать и замерять. Иногда можно заранее знать, какой алгоритм больше подходит.

### 2.3.2. Оценка сложности

Обозначим за  $N$  размер диапазона. Нам известно, что `lower_bound` работает за время порядка  $\log N$ , а `find_if` в худшем случае работает за  $N$ . Не будем учитывать множители при  $\log N$  и  $N$ , так как при достаточно большом  $N$ ,  $N$  будет больше  $\log N$  при любом константном множителе при них. В частности, будем считать, что логарифм двоичный.

Рассмотрим более сложный пример. В цикле пройдёмся по контейнеру `numbers_to_find` и для каждого элемента этого контейнера вызовем `lower_bound`. Поищем это число в другом контейнере. Затем выполним простую операцию, например, `x` умножим на 2.

```
for (int& x : numbers_to_find) {
    lower_bound(begin(v), end(v), x);
    x *= 2;
}
```

Внешний цикл делает столько итераций, каков размер контейнера `numbers_to_find`. Дальше в каждой из итераций мы запускаем `lower_bound`, который работает  $\log N$  времени, потом еще одна операция тратится на умножение `x` на 2. Получаем выражение для грубой оценки количества операций.

```
numbers_to_find.size() * (log(v.size()) + 1)
```

Это оценка для худшего случая. В реальности она может не достигаться.

Выполним программу для `NUMBER = 1` и `QUERY_COUNT = 1000000`.

```
// lower_bound: 701 ms
// find_if: 81 ms
```

`lower_bound` выполнял логарифм операций, в то время как `find_if` сходился практически сразу, выполнив всего несколько операций.

Обсудим сложность алгоритмов.

- **Константная (1):** арифметические операции, обращение к элементу вектора;
- **Логарифмическая ( $\log N$ ):** двоичный поиск (`lower_bound` и пр.), поиск в `set` или `map`;
- **Линейная ( $N$ ):** цикл `for` (с лёгкими итерациями), алгоритмы `find_if`, `min_element` и пр.;

- $N \log N$ : алгоритм `sort` (с лёгкими сравнениями).

Часто используют  $O$ -символику. Например,  $O(\log N)$  означает, что алгоритм выполняет не больше, чем  $\log N$  операций с точностью до константы. « $O$  большое» означает оценку сверху. В принципе она может не достигаться вообще никогда.

Как узнать сложность работы незнакомого алгоритма? Можно вычислить самостоятельно, если известны детали работы алгоритма. Также сложность можно посмотреть в документации.

### 2.3.3. Практические применения

Скорость алгоритмов стоит сравнивать по следующему несложному правилу:

$$1 < \log N < N < N \log N < N^2 < \dots$$

При сложении большее поглощает меньшее:

$$O(N) + O(\log N) = O(N)$$

$O(5N)$  и  $O(8N)$  надо сравнивать измерениями.

Как оценить, подходит ли алгоритм под заданные ограничения? Нужно рассматривать худший случай, брать сложность алгоритма и подставлять туда худшие значения входных данных. Так получается количество операций алгоритма. Далее нужно примерно прикинуть константу, умножить количество операций на неё. В результате получается грубая оценка количества элементарных операций. Далее можно прикинуть время работы алгоритма, если учесть, что на хороших процессорах за секунду можно успеть сделать миллиард простых операций.

Рассмотрим примеры.

**Задача «Синонимы»:**

Поступает три типа запросов.

- ADD word1 word2: добавить пару синонимов (`word1, word2`);

- COUNT word: узнать количество синонимов слова word;
- CHECK word1 word2: узнать, являются ли слова word1 и word2 синонимами.

В задаче приходит 70000 запросов, слова имеют длину не более 100 символов, решить задачу надо за 1 секунду.

```
int main() {
    int q;
    cin >> q;
    map<string, set<string>> synonyms; // для каждого слова запоминаем множество его
                                            // синонимов
    for (int i = 0; i < q; ++i) { // обрабатываем Q запросов (будет Q итераций)
        string operation_code;
        cin >> operation_code;
        if (operation_code == "ADD") {
            string first_word, second_word;
            cin >> first_word >> second_word; // если слова длины не более чем L, то
                                                // считываем их за L простейших операций
            synonyms[first_word].insert(second_word); // поиск в словаре конкретного
                                                // ключа работает за логарифм от
                                                // размера словаря. Он в худшем
                                                // случае равен количеству слов,
                                                // которые уже туда вставили.
                                                // Сложность равна количеству
                                                // запросов, умноженная на
                                                // стоимость сравнения двух строк
                                                // длины не более L.
                                                // итоговая сложность: L logQ
            synonyms[second_word].insert(first_word);
            // O(L) + O(L logQ) = O(L logQ)
        } else if (operation_code == "COUNT") {
            string word;
            cin >> word;
            cout << synonyms[word].size() << endl;
            // поиск в словаре занимает L logQ. Итоговая сложность: O(L logQ)
        } else if (operation_code == "CHECK") {
            string first_word, second_word;
            cin >> first_word >> second_word;
            if (synonyms[first_word].count(second_word) == 1) {
                cout << "YES" << endl;
            }
        }
    }
}
```

```

    } else {
        cout << "NO" << endl;
    }
    // считываем два слова, ищем их в словаре, потом во множестве
    // O(L logQ)
}
}

// обрабатываем Q запросов за L logQ
// суммарная сложность программы: O(Q L logQ)
return 0;
}

```

Задача «Имена и фамилии – 1»:

- `ChangeFirstName(year, first_name)`: добавить факт изменения имени на `first_name` в год `year`;
- `ChangeLastName(year, last_name)`: добавить факт изменения фамилии на `last_name` в год `year`;
- `GetFullName(year)`: получить имя и фамилию по состоянию на конец года `year`;

Были следующие ограничения: 100 запросов, слова длины 10, на выполнение дана одна секунда.

```

string FindNameByYear(const map<int, string>& names, int year) {
    string name; // изначально имя неизвестно
    for (const auto& item : names) { // O(Q) итераций
        if (item.first <= year) { // сравниваем два числа за O(1)
            name = item.second; // присвоить одну строку в другую стоит столько же
            // операций, как длина строки: O(L)
        } else {
            // иначе пора остановиться, так как эта запись и все последующие относятся к
            // будущему
            break;
        }
    }
    return name;
// O(QL) операций
}

```

```

class Person {
public:
    void ChangeFirstName(int year, const string& first_name) {
        first_names[year] = first_name; // ищем year как ключ в словаре и сохраняем
                                       // туда строчку
    } // O(logQ + L)
    void ChangeLastName(int year, const string& last_name) {
        last_names[year] = last_name;
    } // O(logQ + L)
    string GetFullName(int year) {
        const string first_name = FindNameByYear(first_names, year);
        const string last_name = FindNameByYear(last_names, year);

        if (first_name.empty() && last_name.empty()) {
            return "Incognito";
        } else if (first_name.empty()) {
            return last_name + " with unknown first name";
        } else if (last_name.empty()) {
            return first_name + " with unknown last name";
        } else {
            return first_name + " " + last_name;
        }
        // сложность O(L), которая тратится на сложение строк. Также тут вызывается
        // FindNameByYear. Итоговая сложность: O(L) + O(QL)
    }

private:
    map<int, string> first_names;
    map<int, string> last_names;
};

// итоговая сложность программы: O(Q * Q * L)

```

В задаче «Имена и фамилии – 4» оптимизирована функция `FindNameByYear`. В ней теперь вызывается метод `upper_bound`, которая работает за логарифм от размера контейнера, то есть за  $\log Q$ . Тогда суммарная сложность будет  $O(\log Q + L)$ . Метод `GetFullName` также отработает за  $O(\log Q + L)$ . Получаем суммарную сложность  $O(Q(\log Q + L))$ .

### 2.3.4. Амортизированная сложность

Рассмотрим задачу. К нам приходят события с временными метками. Нужно уметь добавлять событие с временной меткой и находить, сколько событий случилось за последние пять минут. Для решения задачи напишем класс:

```
class EventManager {  
public:  
    void Add(uint64_t time);  
    int Count(uint64_t time);  
}
```

Метод `Add` добавляет события, метод `Count` узнаёт количество событий за последние пять минут. Задачу можно решать с помощью очереди. Подключим `<queue>`.

```
private:  
    queue<uint64_t> events;
```

В очередь мы будем добавлять новые события и удалять старые:

```
void Add(uint64_t time) {  
    events.push(time);  
}  
int Count(uint64_t time) {  
    return events.size();  
}
```

Добавим в `private` метод `Adjust`, который принимает новые временные метки и удаляет старые.

```
void Adjust(uint64_t time) {  
    while (!events.empty() && events.front() <= time - 300) {  
        events.pop();  
    }  
}
```

Удалять старые события нужно и при добавлении в очередь, и при вычислении количества событий в ней.

```
void Add(uint64_t time) {  
    Adjust(time);  
    events.push(time);  
}
```

```
int Count(uint64_t time) {
    Adjust(time);
    return events.size();
}
```

Попробуем оценить сложность каждого запроса.

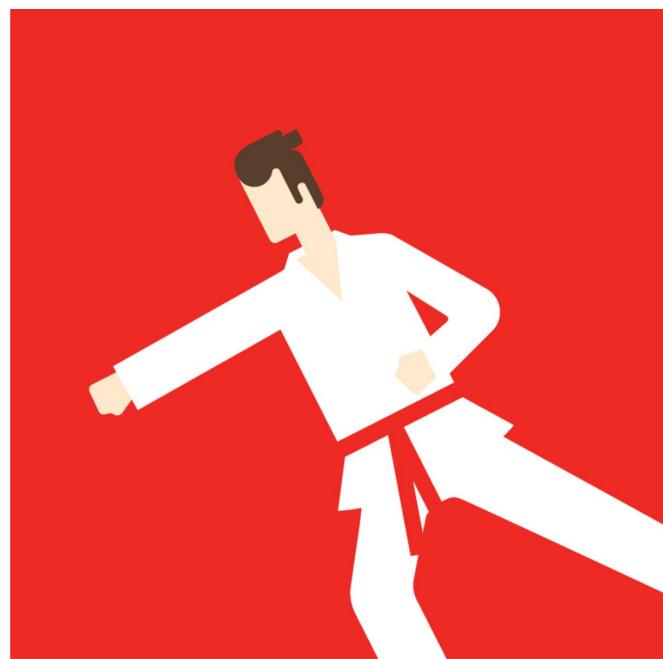
```
class EventManager {
public:
    void Add(uint64_t time) { // O(Q)
        Adjust(time);
        events.push(time); // O(1)
    }
    int Count(uint64_t time) { // O(Q)
        Adjust(time);
        return events.size(); // O(1)
    }
private:
    queue<uint64_t> events;
    void Adjust(uint64_t time) { // O(Q)
        while (!events.empty() && events.front() <= time - 300) { // в худшем
            // случае O(Q)
            events.pop(); // O(1)
        }
    }
}
```

Получается, суммарная сложность будет квадратичной? Дело в том, что это верхняя оценка и она достигаться не будет. Каждый конкретный вызов `Adjust`, `Add` или `Count` может работать долго. Суммарно все эти события будут работать за  $O(Q)$ , а не за  $O(Q * Q)$ . В таком случае говорят, что сложность `Add`, `Count`, `Adjust` не  $O(Q)$ , а amortized  $O(1)$ , то есть в среднем каждый метод работает как  $O(1)$ .

# Основы разработки на C++: красный пояс

Неделя 3

Модель памяти в C++



# Оглавление

<b>Модель памяти в C++</b>	<b>2</b>
3.1    Модель памяти . . . . .	2
3.1.1    Введение в модель памяти: стек . . . . .	2
3.1.2    Введение в модель памяти: куча . . . . .	4
3.1.3    Оператор <code>new</code> . . . . .	5
3.1.4    Оператор <code>delete</code> . . . . .	6
3.1.5 <code>new</code> и <code>delete</code> для объектов классового типа . . . . .	8
3.1.6    Операторы <code>new[]</code> и <code>delete[]</code> . . . . .	9
3.1.7    Введение в арифметику указателей . . . . .	10
3.1.8    Добавляем в вектор <code>begin</code> и <code>end</code> . . . . .	13
3.1.9    Константный указатель и указатель на константу . . . . .	15
3.1.10    Итоги раздела . . . . .	17

# Модель памяти в C++

## 3.1. Модель памяти

В этом разделе мы изучим, как эффективно использовать основные типы языка C++. Для этого необходимо изучить, как они устроены внутри.

### 3.1.1. Введение в модель памяти: стек

Рассмотрим пример:

```
void second() {
    int s_a = 3;
    double s_d = 2.0;
}

void first() {
    int f_a = 2;
    char f_c = 'a';
    second();
}

int main() {
    int a = 1;
    char c = 'r';
    first();
    second();
    a = 2;
    c = 'q';
}
```

Стек	
main()	
	int a
	char c

У каждой функции есть локальные переменные. Они хранятся в специальной области оперативной памяти, которая называется стеком.

Стек	
first()	main()
	int a
	char c
	int f_a
	char f_c
	int s_a
	double s_d

Когда в программе запускается очередная функция, то на стеке резервируется блок памяти, достаточный для хранения локальных переменных. Кроме того, в этом блоке хранится служебная информация, такая как адрес возврата. Такая область памяти называется стековым фреймом функции. Когда функция `main` запускает функцию `first`, то на стеке ниже функции `main` резервируется фрейм для функции `first`. То же самое происходит, когда функция `first` вызывает функцию `second`.

Стек	
main()	
	int a
	char c
second()	int s_a
	int s_a
	double s_d
second()	double s_d

Когда функция `second` завершает свою работу, то вершина стека перемещается на фрейм предыдущей функции. При этом фрейм отработавшей функции просто остаётся на стеке. Выйдем из функцию `first` в функцию `main`, запустим функцию `second` из `main`.

Стековый фрейм перетирает данные, которые были в стеке от предыдущих вызовов. Выйдем из функции `second` и выйдем из функции `main`. Когда выходим из программы, то вершина стека поднимается до самого верха.

### 3.1.2. Введение в модель памяти: куча

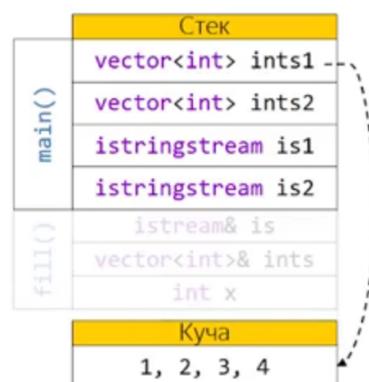
Рассмотрим пример

```
void fill(istream& is, vector<int>& ints) {
    int x;
    while (is >> x) {
        ints.push_back(x);
    }
}

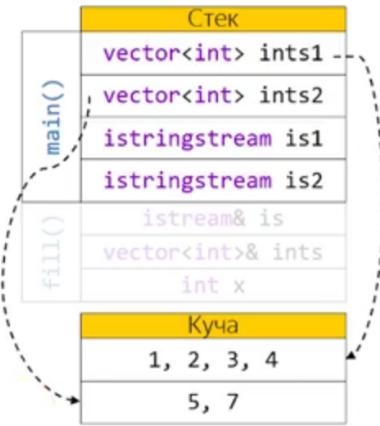
int main() {
    vector<int> ints1, ints2;
    istringstream is1("1 2 3 4");
    fill(is1, ints1);
    istringstream is2("5 7");
    fill(is2, ints2);
}
```

В функции `main` объявлены два вектора целых чисел, также объявлены два потока ввода из строк. Кроме того, есть функция `fill`, она читает из потока числа и помещает их в вектор. В результате работы программы в векторе `ints1` будут храниться цифры 1, 2, 3, 4; в векторе `ints2` будут храниться цифры 5, 7. Предположим, что числа, которые функция `fill` разместит в вектор, располагаются ниже её стекового фрейма. Это не так, потому что при повторном запуске `fill`, она перезапишет эти числа. Получается, в данном случае нам не подходит стек, потому что объекты, создаваемые функцией `fill`, должны жить дольше, чем функция `fill`. На стеке память постоянно будет перезаписываться вызовами новых функций.

Для хранения элементов вектора нам подходит источник памяти, который называется «куча». Во время работы программы может в любой момент обращаться в кучу и выделять в ней блок памяти произвольного размера или освобождать ранее выделенный. Посмотрим, как программа из примера использует кучу для хранения элементов вектора.



После завершения работы функции `fill` в куче будет выделен блок памяти, достаточный для хранения четырех целых чисел. При этом вектор `ints1` будет ссылаться на эту область памяти в куче и будет иметь доступ к своим элементам.



Когда `fill` отработает во второй раз, то в куче будет выделен отдельный блок памяти, достаточный для хранения двух целых чисел.

Подведём итог. В модели памяти C++ есть два источника памяти: стек и куча. В стеке размещаются локальные переменные функций. В куче размещаются объекты, которые должны жить дольше, чем создавшая их функция.

### 3.1.3. Оператор `new`

Возможность выделения памяти в куче доступна любому программисту. Чтобы выделить в куче память для хранения одного значения типа `int` нужно объявить локальную переменную типа `int*`.

```
int* pInt = new int;
```

`new` – это специальный оператор, который выполняет выделение памяти из кучи. Оператор `int*` подсказывает, для хранения какого типа нам нужна память. При этом `pInt` – это указатель на значение типа `int`. Сама переменная `pInt` будет храниться в стеке, но она будет указывать на область памяти в куче.



Указатель – это адрес в памяти. Память в C++ представляется как линейный массив байтов. Указатель – это индекс в этом массиве.

Как обратиться к значению, на которое указывает указатель? Синтаксис при работе с указателями такой же, как при работе с итераторами.

Рассмотрим другой пример.

```
string* s = new string;
*s = "Hello";
cout << *s << ' ' << s->size() << endl;
// Hello 5
```

Оператор `*`, примененный к указателю, возвращает ссылку на объект в куче.

```
string* s = new string;
*s = "Hello";
string& ref_to_s = *s;
ref_to_s += ", world";
cout << *s << endl;
// Hello, world
```

Инициализация объектов, выделенных на куче, выполняется аналогично инициализации объектов, создаваемых на стеке при объявлении локальных переменных.

### 3.1.4. Оператор `delete`

Рассмотрим пример. У нас есть вектор целых чисел `v`, в отдельной области видимости объявлен итератор `iter`, в который записан результат поиска пятерки в нашем векторе `v`.

```
vector<int> v = {1, 2, 3, 4, 5};
{
    auto iter = find(begin(v), end(v), 5);
}
```

Что станет с пятеркой после выхода `iter` из области видимости? Ничего не произойдет, потому что итератор просто указывает позицию в контейнере и никак не управляет значением, на которое он указывает. Сам итератор разрушился, а значение в векторе никуда не делось, оно продолжает там храниться и мы можем им пользоваться.

Рассмотрим другой пример. Мы в отдельной области видимости выделяем в куче память для хранения целого числа и инициализируем её пятеркой, указатель на эту память мы записываем

в переменную `pFive`. Что станет с пятёркой в куче после выхода `pFive` из области видимости? Ничего – пятёрка останется в куче, и произойдёт утечка памяти, то есть память, выделенная на куче, будет числиться за нашим процессом и у нас не будет возможности обратиться к этой памяти и освободить её, потому что мы потеряли указатель на эту память, он вышел за пределы области видимости.

Напишем программу, которая будет считывать со входа число `n` и будет считать сумму `n` случайных 64-битных чисел.

```
int main() {
    int n;
    cin >> n;

    mt19937_64 random_gen; // генератор случайных чисел
    uint64_t sum = 0;
    for (int i = 0; i < n; ++i) {
        uint64_t x = random_gen();
        sum += x;
    }
    cout << sum;
}
// на вход подаём 10
// на выходе 4762160605604824475
```

Программа работает нормально. Теперь будем выделять в куче память для хранения очередного случайного числа.

```
for (int i = 0; i < n; ++i) {
    auto x = new uint64_t;
    *x = random_gen();
    sum += *x;
}
```

Такая программа также работает, но происходит утечка памяти. Это может стать проблемой при запуске программы на больших `n`. Бороться с этим можно при помощи оператора `delete`. Он освобождает память, выделенную оператором `new`.

```
for (int i = 0; i < n; ++i) {
    auto x = new uint64_t;
    *x = random_gen();
    sum += *x;
}
```

```
    delete x;  
}
```

Программа работает, при этом количество выделенной памяти не растёт.

### 3.1.5. new и delete для объектов классового типа

Продемонстрируем, что при вызове оператора `new` для объектов классового типа оператор `new` не только выделяет необходимую память в куче, но и вызывает конструктор.

```
struct Widget {  
    Widget() {  
        cout << "constructor" << endl;  
    }  
};  
  
int main() {  
    new Widget;  
}  
// constructor
```

Когда для этого объекта вызывается деструктор? Напишем его.

```
~Widget() {  
    cout << "destructor" << endl;  
}
```

При выполнении программы в консоль вывелось только `constructor`. Деструктор не был вызван. Деструктор вызывает оператор `delete`.

```
int main() {  
    Widget* w = new Widget;  
    delete w;  
}  
// constructor  
// destructor
```

Мы убедились, что оператор `delete` не только освобождает место в памяти, но и вызывает деструктор для соответствующего объекта в куче.

### 3.1.6. Операторы `new[]` и `delete[]`

Напишем свой собственный `vector`. Начнём с простого интерфейса:

```
template <typename T>
class SimpleVector {
public:
    explicit SimpleVector (size_t size);
    ~SimpleVector();

private:
    T* data;
};

int main() {
    SimpleVector<int> sv(5);
}
```

Объявим шаблон класса `SimpleVector`. В интерфейсе нашего класса пока будут только конструктор и деструктор. Конструктор принимает количество элементов в нашем векторе. Также у нас будет приватное поле `data`, которая будет указателем на тип `T`.

Реализуем конструктор. Нам в куче нужно выделить `size` объектов. Пока мы умеем создавать только один объект. Несколько объектов можно создать с помощью оператора `new []`. Он создает блок памяти для хранения необходимого количества объектов.

```
explicit SimpleVector (size_t size){
    data = new T[size];
}
```

Сейчас в нашей программе есть утечка памяти. Освобождать её следует в деструкторе.

```
~SimpleVector() {
    delete [] data;
}
```

Если вместо `delete []` пропишем просто `delete`, то программа будет работать некорректно.

### 3.1.7. Введение в арифметику указателей

Добавим в наш вектор возможность обращаться к отдельным элементам. У нас есть указатель `data` на первый элемент вектора. Указатели на другие элементы получаются очень просто: нужно добавить целое число, например, `data + 1` указывает на второй элемент вектора.



Можем написать оператор доступа по индексу.

```
T& operator[] (size_t index) {
    return *(data + index);
}
```

Теперь можно заполнять вектор числами и выводить их на экран.

```
int main() {
    SimpleVector<int> sv(5);
    for (int i = 0; i < 5; ++i) {
        sv[i] = 5 - i;
    }
    for (int i = 0; i < 5; ++i) {
        cout << sv[i] << ' ';
    }
}

// 5 4 3 2 1
```

В нашей программе мы выделили память на 5 элементов. Но если, например, попытаться вывести на экран `sv[12]`, то код скомпилируется и может даже отработать:

```
SimpleVector<int> sv(5);
cout << sv[12] << endl;
// 7827296
```

Язык C++ не контролирует доступ к данным, которые мы осуществляем через указатель. Сейчас мы прочитали элемент, который лежит за границами нашего вектора.

```
SimpleVector<string> sv(5);
cout << sv[12] << endl;
```

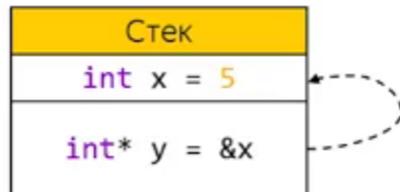
Такая программа упадёт, потому что мы обращаемся к случайному участку памяти, в котором не выполняются инварианты класса `string`.

К указателям можно не только прибавлять целые числа, но и вычитать их.



Локальные переменные хранятся на стеке. Стек – это область оперативной памяти, поэтому у его элементов тоже есть адрес. Его можно получить с помощью оператора `&`:

```
int main() {
    int x = 5;
    int* y = &x;
}
```



Переменная `y` хранит в себе адрес переменной `x`.

```
int main() {
    int x = 5;
    int* y = &x;
    *y = 7;
    cout << x << endl;
```

```
}

// 7

int main() {
    int a = 43;
    int b = 71;
    int c = 89;
    cout << *(&b - 1) << ' ' << *(&b + 1);
}
// 43 89

void f() {
    int a = 43;
    int b = 71;
}

int main() {
    int c = 89;
    for (int i = 0; i < 20; ++i){
        f();
        int x = *(&c - i);
        cout << i << ' ' << x << endl;
    }
}
// 0 89
// ...
// 14 43
// 15 71
// ...
```

Программа выводит 20 строк. В строках 14 и 15 записаны числа 43 и 71. Они совпадают с переменными **a** и **b** нашей программы. Если мы возьмем, например, **a** = 434 и **b** = 711, то в строках 14 и 15 окажутся уже 434 и 711 соответственно.

В функции **main** мы запустили функцию **f**. Она запустилась, выделила на стеке фрейм и разместила в нем свои переменные. Мы не знаем точно, где на стеке эти переменные лежат, пробуем разные смещения относительно своей переменной **c**, пытаемся найти стековый фрейм функции **f**. Мы установили, что в четырнадцати **int**'ах от переменной **c** лежит переменная **a**. Мы убедились, что после завершения функции с её стековым фреймом ничего не происходит.

```
int main() {
```

```
int c = 89;
int* d = &c;
int* e = d + 1;
cout << d << endl
    << e << endl;
}
// 0x62fe3c
// 0x62fe40
```

В консоль вывелись два шестнадцатеричных числа. Если мы вычтем одно из другого, то получим 4. Если запустить программу с

```
int* e = d + 3;
```

то получаем 12. Этот пример иллюстрирует то, что при прибавлению к указателю числа целочисленное значение, которое хранится в указателе, изменяется на число, умноженное на размер типа, на который указывает наш указатель.

Перепишем оператор [], используя менее громоздкий синтаксис.

```
T& operator[] (size_t index) {
    return data[index];
}
```

### 3.1.8. Добавляем в вектор begin и end

Добавим в наш вектор методы `begin` и `end`, чтобы по нему можно было итерироваться в цикле. Цикл `range-based for` разворачивается в следующую конструкцию:

```
SimpleVector<int> v(5);
for (auto i = v.begin(); i != v.end(); ++i) {

}
```

Требования к итераторам в `range-based for`:

- `begin()` указывает на первый элемент;

- `end()` указывает на последний элемент;
- увеличение итератора на один переводит его на следующий элемент.

Всем этим требованиям удовлетворяют указатели на элементы нашего вектора. В качестве `begin` можно использовать указатель на первый элемент нашего вектора, в качестве `end` – указатель на последний элемент. Соответственно, в качестве типа возвращаемого значения для операторов `begin()` и `end()` можем использовать указатель на элемент типа `T`.

В результате класс `SimpleVector` будет выглядеть следующим образом.

```
class SimpleVector {
public:

    explicit SimpleVector (size_t size) {
        data = new T[size];
        end_ = data + size;
    }

    ~SimpleVector() {
        delete[] data;
    }

    T& operator[] (size_t index) {
        return data[index];
    }

    T* begin() { return data; }
    T* end()   { return end_; }

private:
    T* data;
    T* end_;
};
```

Теперь напишем функцию `print` для нашего вектора, которая будет печатать его на консоль.

```
template <typename T>
void Print(const SimpleVector<T>& v) {
    for (const auto& x : v) {
        cout << x << ' ';
```

```
    }  
}
```

Такая функция работать не будет из-за проблем с константностью. Вектор `v` передается по константной ссылке, поэтому и вызывать мы можем только константные методы объекта `v`. Методы `begin` и `end` константными не являются. Сделаем их константными:

```
T* begin() const { return data; }  
T* end() const { return end_; }
```

В текущей функции `Print` мы можем случайно поменять наш вектор, например, прописав строку  
`*i = 42;`

Теперь вызвав функцию `Print` два раза, мы увидим на экране:

```
// 5 3 4 -1  
// 42 42 42 42
```

Чтобы избежать этой проблемы, в методах `begin()` и `end()` нам нужно возвращать указатели на константы:

```
const T* begin() const { return data; }  
const T* end() const { return end_; }
```

Теперь компилятор будет запрещать изменять вектор, принимаемый по константной ссылке.

Есть еще одна проблема. Теперь мы не можем через возвращаемые итераторы менять наш вектор, например, его сортировать. Нам нужно завести две пары `begin` и `end`: константную и неконстантную.

```
T* begin() { return data; }  
T* end() { return end_; }  
  
const T* begin() const { return data; }  
const T* end() const { return end_; }
```

### 3.1.9. Константный указатель и указатель на константу

У константности указателей есть двойственность.



С одной стороны у нас есть указатель, с другой есть объект, на который он указывает. Соответственно мы можем менять сам указатель, а можем этот объект. Когда переменная объявлена как `T* ptr`, то это просто указатель и мы можем менять как объект, так и сам указатель.

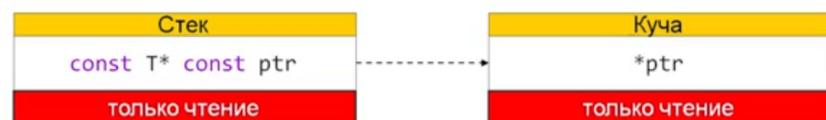


Если же перед типом указателя мы пишем ключевое слово `const`, то получаем указатель на константу: мы можем изменять сам этот указатель, но объект мы менять не можем.

Если мы хотим менять объект, но не хотим менять сам указатель, мы можем объявить константный указатель.



Ключевое слово `const` у него идет после `*`. Тогда мы можем менять объект, на который указывает указатель, но не можем менять сам указатель. Если же мы хотим защититься от любых изменений данных, мы можем объявить константный указатель на константу:



### 3.1.10. Итоги раздела

#### Сравнение стека и кучи

	Стек	Куча
Создание объекта	<code>string s;</code>	<code>string* s = new string;</code>
Уничтожение объекта	Автоматически при выходе из области видимости	Вручную: <code>delete s;</code>
Применяется	Для локальных переменных	Для объектов, которые живут дольше, чем создавшая их функция

#### Недостатки использования `new/delete`

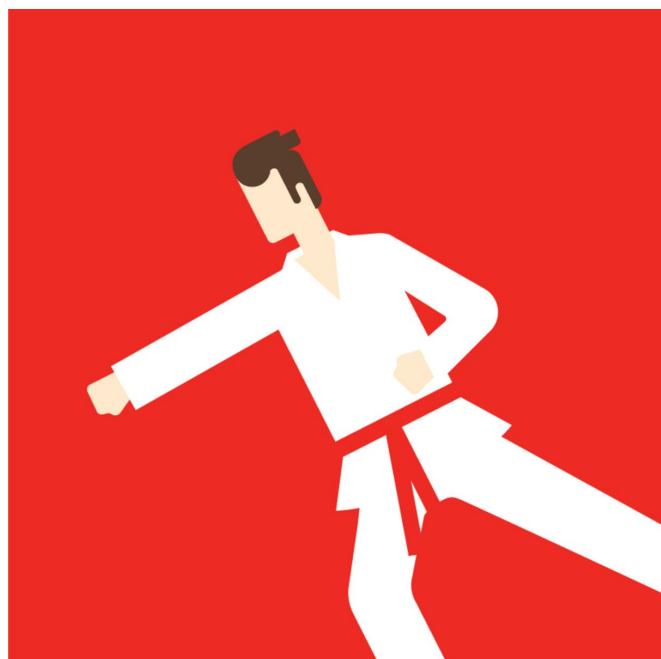
- можно забыть вызывать `delete`;
- можно перепутать `delete` и `delete[]`.

Всё, что связано с операторами `new`, `delete` и арифметикой указателей было рассказано, чтобы потом рассказать про внутреннее устройство контейнеров языка C++. В современном языке C++ нет ни одной причины применять `new` и `delete` в прикладном коде.

# Основы разработки на C++: красный пояс

Неделя 4

Эффективное использование линейных контейнеров



# Оглавление

<b>Эффективное использование линейных контейнеров</b>	<b>2</b>
4.1 Эффективное использование линейных контейнеров . . . . .	2
4.1.1 Эффективное использование вектора . . . . .	2
4.1.2 Инвалидация ссылок . . . . .	5
4.1.3 Эффективное использование дека . . . . .	7
4.1.4 Инвалидация итераторов . . . . .	9
4.1.5 Контейнер <code>list</code> . . . . .	10
4.1.6 Контейнер <code>array</code> . . . . .	14
4.1.7 Класс <code>string_view</code> . . . . .	16

# Эффективное использование линейных контейнеров

## 4.1. Эффективное использование линейных контейнеров

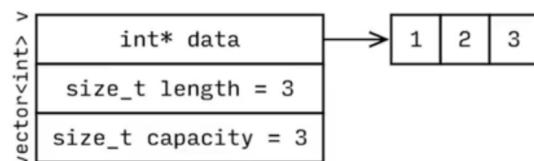
Мы приступаем к исследованию последовательных или линейных контейнеров. Так называют контейнеры, которые сохраняют порядок вставляемых в них элементов. Типичный пример – вектор. Пример непоследовательного контейнера – множество.

### 4.1.1. Эффективное использование вектора

Создадим вектор из трёх чисел:

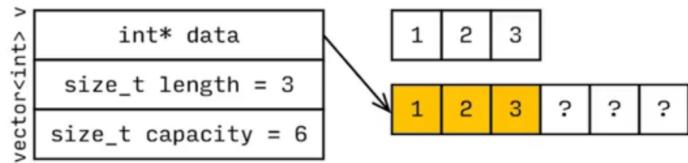
```
vector<int> v = {1, 2, 3};
```

На стеке у этого вектора будет указатель на кучу, также на стеке будет храниться длина этого вектора, а также `capacity` – количество памяти, доступное данному вектору в куче.



Что будет, если мы захотим добавить четвёрку в вектор? Нам нужен блок памяти, в который мы сможем положить 1, 2, 3 и 4. Соответственно, как и в задаче `SimpleVector`, стандартный вектор выделяет в два раза больше памяти, чем у него было до этого.

Он выделяет блок под 6 `int`'ов, чтобы туда можно было положить четыре числа. Теперь вектор присваивает себе блок памяти из шести элементов, про старый забывает, в новый копирует



три `int`'а, которые у него были, и теперь туда же может положить четверку, предварительно освободив старый блок памяти. Теперь длина – 4, `capacity` – 6.

Напишем функцию `LogVectorParams`, которая будет для вектора выводить его параметры: длину и его `capacity`.

```
void LogVectorParams (const vector<int>& v) {
    cout << "Length = " << v.size() << ", "
        "capacity = " << v.capacity() << "\n";
}
```

Вызовем функцию после создания вектора и после добавления четырёх единиц.

```
int main() {
    vector<int> v = {1, 2, 3};
    LogVectorParams(v);
    v.push_back(4);
    LogVectorParams(v);

    return 0;
}
// Length = 3, capacity = 3
// Length = 4, capacity = 6
```

Всё, как мы указали. Мы утверждаем, что элементы вектора хранятся подряд. Более того, в `SimpleVector`'е мы оперировали указателями и там был указатель на данные в куче. Оказывается, стандартный вектор тоже может отдать нам этот указатель.

У стандартного вектора есть метод `data`, который возвращает указатель на те данные, которые у него лежат в куче. Вызовем этот метод, сохраним его в указателе. Мы хотим посмотреть на данные, которые лежат по указателю `data` и в следующих ячейках. Вектор константный, поэтому `data` возвращает константный указатель, поэтому переменная `data` должна иметь тип `const int*`.

```
int* data = v.data();
```

```
for (size_t i = 0; i < v.capacity(); ++i) {
    cout << *(data + i) << " ";
}
cout << "\n";

// Length = 3, capacity = 3
// 1 2 3
// Length = 6, capacity = 6
// 1 2 3 4 78063648 0
```

С помощью указателей мы посмотрели на содержимое вектора: там лежат 1, 2, 3, 4 и два числа, которые когда-то лежали в куче. Вектор их никак не инициализировал, в эту память вектор потом сможет добавить новые числа.

Если мы захотим очистить память от этих двух чисел, то можем использовать `shrink_to_fit()`. Добавим его в программу и посмотрим, что получится.

```
v.shrink_to_fit();
LogVectorParams(v);
// Length = 4, capacity = 4
// 1 2 3 4
```

Рассмотрим пример. Допустим, мы хотим сложить в вектор какие-то числа, заранее зная, сколько этих чисел будет. Подадим на вход число 10000000.

```
int main() {
    int size;
    cin >> size;

    { LOG_DURATION("push_back");
        vector<int> v;
        for (int i = 0; i < size; ++i) {
            v.push_back(i);
        }
    }

    return 0;
}
// push_back: 427 ms
```

Казалось бы, за секунды мы должны успевать больше операций. Такая серия `push_back`'ов неэф-

фективна за счёт того, что при `push_back`'е вектор при необходимости перевыделяет память и копирует данные из старой памяти в новую. В данном случае мы знаем, что вектор будет иметь размер `size`. Пусть вектор заранее выделит себе блок из `size` элементов.

```
{ LOG_DURATION("push_back");
    vector<int> v;
    v.reserve(size);
    for (int i = 0; i < size; ++i) {
        v.push_back(i);
    }
}
// push_back: 288 ms
```

#### 4.1.2. Инвалидация ссылок

У нас есть вектор из трех элементов, что будет, если перед добавлением четвертого элемента сохранить ссылку на первый элемент?

```
int main() {
    vector<int> v = {1, 2, 3};
    int& first = v[0];
    cout << first << "\n";
    v.push_back(4);
    cout << first << "\n";

    return 0;
}
// 1
// 45628208
```

До `push_back`'а там лежала 1, после `push_back`'а мы знаем, что вектор переместил данные в другое место, а старые данные освободил. Теперь по ссылке лежит мусорное число. Ссылка перестала быть валидной – инвалидировалась.

В материале «Класс `StringSet`» представлена реализация контейнера, в который можно добавлять строки с заданным приоритетом с помощью метода `Add`, а также находить последнюю добавленную строку и строку с наибольшим приоритетом.

В предложенной реализации строки складываются в вектор, также они складываются в `set` с

помощью структур, которые называются `StringItem`. В каждой структуре лежит сама строка вместе с приоритетом. Также есть оператор сравнения, который сравнивает `StringItem`'ы по приоритету. При добавлении строчки в контейнер мы кладем её в вектор и в `set`. В методе `FindLast()` мы берем последний элемент вектора, в методе `FindBest()` мы берём последний элемент `set`'а.

Создадим контейнер `strings`, кладём туда строчку "upper" с высоким приоритетом, затем строчку "lower" с низким приоритетом. Метод `FindLast()` должен вывести последнюю добавленную строчку, то есть "lower", а метод `FindBest()` должен вывести строчку с наивысшим приоритетом, то есть "upper".

```
int main() {
    StringSet strings;
    strings.Add("upper", 10);
    strings.Add("lower", 0);
    cout << strings.FindLast() << "\n";
    cout << strings.FindBest() << "\n";
    return 0;
}
// lower
// upper
```

Является ли эффективным складывать одну и ту же строчку и в вектор и в множество? Хочется этого избежать и складывать строчки в один контейнер. Можно в один контейнер сложить строчки, а в другой складывать ссылки на строчки в том контейнере. Во множество теперь будем класть не саму строчку, а ссылку на неё. В методе `Add` в `sorted_data` вставляем теперь ссылку на строчку, которую мы добавили в вектор.

```
sorted_data.Insert(StringItem{data.back(), priority});
```

В результате работы программы на экран выводится только `lower`. `upper` не выводится, потому что произошла инвалидация.

Чтобы решить проблему можно использовать контейнер, который не обладает таким недостатком, как вектор, например, `deque`.

Что делать, если важно, чтобы оставался вектор, а не `deque`? Тогда придется отказаться от ссылок. Например, вместо ссылок можно хранить индексы элементов в векторе.

### 4.1.3. Эффективное использование дека

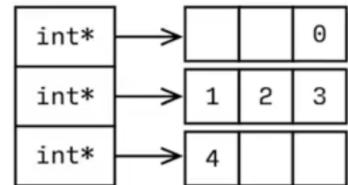
Дек можно получить, отказавшись от требования хранить все элементы подряд в едином куске памяти. Выделим кусок памяти и положим туда три int'a.

```
deque<int> d = {1, 2, 3};
```

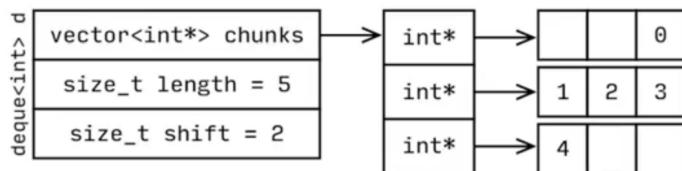
		0
1	2	3
4		

Если мы хотим добавить туда четверку, то не можем добавить её туда сразу после тройки, потому что выделили блок памяти под три int'a. Но мы можем выделить ещё один блок памяти на три int'a и положить в первую ячейку четвёрку. Аналогично, если мы хотим добавить в начало ноль, то нужно выделить блок памяти на три int'a и в последний элемент, чтобы соблюсти некий разумный порядок, положить ноль.

Нам нужно уметь обращаться к элементу по его номеру. Необходимо эти блоки памяти – они называются чанками – как-то проиндексировать. Сохраним вектор указателей на эти чанки, на эти блоки памяти.



В самом деке нужно хранить вектор указателей на чанки, количество элементов, также понадобится хранить так называемый сдвиг. Сдвиг того самого нуля от начала его чанка, то есть, сдвиг первого элемента дека от начала его блока памяти.



Как теперь найти второй элемент? Прибавляем сдвиг, с учетом сдвига необходим четвертый элемент. Размер блока памяти – 3. Делим 4 на 3, получаем 1, идем в первый чанк памяти и там идем в элемент  $4 \% 3$ , то есть берем остаток от деления и получаем, что в первом блоке памяти нам нужен первый элемент.

В итоге мы получили быструю вставку в начало и неинвалидацию ссылок при вставке в начало или в конец. Однако теперь мы тяжелее получаем элемент по индексу. Итерироваться тоже непросто, потому что нужно перепрыгивать между чанками.

Пусть нам нужно вставить набор чисел в наш контейнер, при этом мы заранее не знаем, сколько будет чисел – то есть не можем вызывать `reserve`.

```
int main() {
    const int SIZE = 1000000;

    vector<int> v;
    { LOG_DURATION("vector");
        for (int i = 0; i < SIZE; ++i) {
            v.push_back(i);
        }
    }

    deque<int> d;
    { LOG_DURATION("deque");
        for (int i = 0; i < SIZE; ++i) {
            d.push_back(i);
        }
    }

    return 0;
}
// vector: 61 ms
// deque: 52 ms
```

Особой разницы нет. Что, если элементов 5000000?

```
// vector: 398 ms
// deque: 168 ms
```

Это говорит о том, что нельзя заранее сказать, что будет полезнее: вектор или дек. Нужно измерять.

Теперь попробуем отсортировать полученные вектор и дек.

```
{ LOG_DURATION("sort vector");
    sort(rbegin(v), rend(v));
}
{ LOG_DURATION("sort deque");
    sort(rbegin(d), rend(d));
}
// sort vector: 5468 ms
```

```
// sort deque: 10851 ms
```

Мы быстрее заполнили дек, но сортировали его гораздо дольше, чем вектор.

Вектор хорош быстрыми обращениями к элементам. Итерирование по вектору тоже быстрое. Дек хорош тем, что можно быстро сделать серию `push_back`'ов, если заранее не известен размер. Кроме того, можно быстро вставлять в начало, при этом не инвалидируя ссылки на элементы дека.

#### 4.1.4. Инвалидация итераторов

Создадим вектор из одного элемента и сохраним итератор на это число. Затем вставим 2000 чисел в вектор. Посмотрим, что будет лежать по этому итератору.

```
int main() {
    vector<int> numbers = {1};
    auto it = begin(numbers);
    cout << *it << "\n";

    for (int i = 0; i < 2000; ++i) {
        numbers.push_back(i);
    }

    cout << *it << "\n";

    return 0;
}
// 1
// 78067936
```

Итераторы вектора по сути являются указателями. Соответственно при вставке в вектор они инвалидируются. Что будет для дека?

```
// 1
// 1
```

Итератор не пострадал. Попробуем с помощью этого итератора посмотреть на последний элемент дека.

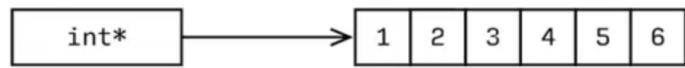
```
cout << *it << " " << *(it + numbers.size() - 1) << "\n";
```

Такая программа падает. Дело в том, что итератор был получен до добавления в дек элементов. Этот итератор знает только про устройство старого дека. Итераторы содержат дополнительную логику, позволяющую итерироваться по контейнеру. Дек не может сохранять итераторы валидными.

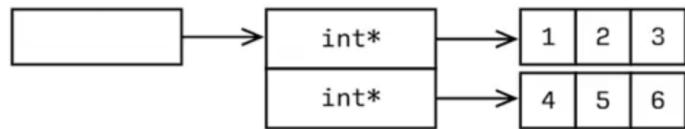
#### 4.1.5. Контейнер list

Мы узнали, что есть различные стратегии выделения памяти под линейные контейнеры: вектор выделяет единый блок памяти под все свои элементы, а дек выделяет память чанками.

```
vector<int> v = {1, 2, 3, 4, 5, 6};
```

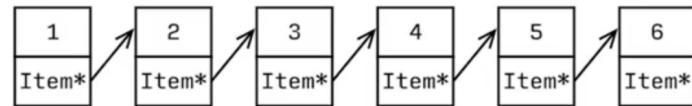


```
deque<int> d = {1, 2, 3, 4, 5, 6};
```

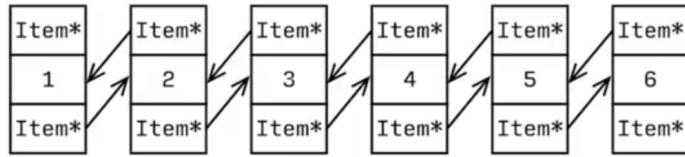


Что, если каждому элементу выделять свой кусок памяти? Такая стратегия используется в контейнере `list`.

Если каждому элементу контейнера соответствует свой блок памяти, то по такому контейнеру даже проитерироваться нельзя. Можно научить каждый элемент ходить к следующему, то есть рядом с ним положить указатель на следующий элемент.



Теперь мы можем проитерироваться по списку из начала в конец. Иногда удобнее проитерироваться в обратную сторону, поэтому стоит положить указатель на обратный элемент.



Теперь мы можем быстро удалять элементы из середины списка. Если мы хотим удалить элемент 4, то мы находим его, удаляем и перевешиваем указатели.

```
list<int> numbers = {1, 2, 3, 4, 5, 6};
auto it = find(begin(numbers),
               end(numbers), 4); // O(N)
numbers.erase(it); // O(1)
```

Все указатели и итераторы, кроме указателя на 4, останутся валидными. Однако при таком устройстве списка мы не можем быстро обратиться к элементу по его номеру.

Рассмотрим пример. Будем складывать их в контейнер с помощью метода `Add`. Удалять элементы будем с помощью метода `Remove`, который будет принимать условие, по которому нужно удалять элементы. Он будет шаблонным и будет принимать функциональный объект.

```
class NumbersOnVector {
public:
    void Add(int x) {
        data.push_back(x);
    }

    template <typename Predicate>
    void Remove(Predicate predicate){
        data.erase(
            remove_if(begin(data), end(data), predicate),
            end(data));
    }
private:
    vector<int> data;
};
```

Давайте реализуем этот контейнер с помощью списка.

```
class NumbersOnList {
public:
    void Add(int x) {
```

```
    data.push_back(x);
};

template <typename Predicate>
void Remove(Predicate predicate){
    data.remove_if(predicate);
}
private:
    list<int> data;
};
```

Напишем бенчмарк, чтобы сравнить два контейнера. Заполним контейнеры миллионом элементов, затем будем удалять их с помощью метода `Remove`: в начале элементы, остаток от деления на 10 у которых равен нулю, затем единице и так далее. В итоге за 10 вызовов метода `Remove` мы удалим все числа.

```
const int SIZE = 1000000;
const int REMOVAL_COUNT = 10;

int main() {
{ LOG_DURATION("vector");
    NumbersOnVector number;
    for (int i = 0; i < SIZE; ++i) {
        numbers.Add(i);
    }
    for (int i = 0; i < REMOVAL_COUNT; ++i) {
        numbers.Remove([i](int x) { return x % REMOVAL_COUNT == i; });
    }
}

{ LOG_DURATION("list");
    NumbersOnList number;
    for (int i = 0; i < SIZE; ++i) {
        numbers.Add(i);
    }
    for (int i = 0; i < REMOVAL_COUNT; ++i) {
        numbers.Remove([i](int x) { return x % REMOVAL_COUNT == i; });
    }
}
return 0;
}
```

```
// vector: 224 ms
// list: 433 ms
```

Мы удаляли по много элементов за раз. Попробуем удалять понемногу. Пусть будет SIZE = 10000 элементов и мы их попробуем удалить за REMOVAL\_COUNT = 1000 шагов.

```
// vector: 154 ms
// list: 109 ms
```

Теперь большую роль играет тот факт, что из списка можно быстро удалить из середины. Получается, от конфигурации входных параметров зависит, что эффективнее: вектор или список.

На том же примере продемонстрируем, что при удалении элементов из списка у нас неинвалидируются ссылки, указатели и даже итераторы. Дополним класс NumbersOnList, который будет находить последний элемент, удовлетворяющий некоторому условию.

```
auto FindLast(Predicate predicate) {
    return find_if(rbegin(data), rend(data), predicate);
}
```

Перед серией удалений из списка мы хотим найти последний элемент, который делится на REMOVAL\_COUNT. Оставим в контейнере те элементы, которые делятся на REMOVAL\_COUNT, при удалении мы будем итерироваться, начиная с единицы. В конце посмотрим на элемент, который мы нашли перед серией удалений.

```
{ LOG_DURATION("list");
    NumbersOnList number;
    for (int i = 0; i < SIZE; ++i) {
        numbers.Add(i);
    }
    auto it = numbers.FindLast(
        [] (int x) { return x % REMOVAL_COUNT == 0; });
    for (int i = 1; i < REMOVAL_COUNT; ++i) {
        numbers.Remove([i] (int x) { return x % REMOVAL_COUNT == i; });
    }
    cout << *it << "\n";
}
// list: 133 ms
// 9000
```

Итератор указывает на верный элемент. Проитерируемся по нему до начала списка.

```
while() (*it != 0) {
    cout << *it << " ";
    ++it;
}
// 9000 8000 7000 6000 5000 4000 3000 2000 1000
```

Проитерироваться получилось. Мы вывели весь список целиком с помощью итератора, который создали еще тогда, когда мы из списка не поудаляли много элементов. Это доказывает, что списки страхуют нас от инвалидации.

#### 4.1.6. Контейнер array

Пусть нам необходимо вызывать функцию COUNT раз. Функция всегда возвращает пять чисел.

```
vector<int> BuildVector(int i) {
    return {i, i + 1, i + 2, i + 3, i + 4};
}

const int COUNT = 1000000;

int main() {
    LOG_DURATION("vector");
    for (int i = 0; i < COUNT; ++i) {
        auto numbers = BuildVector()
    }
}

return 0;
}
// vector: 260 ms
```

Программа будет работать быстрее, если каждый раз выделять память не в куче, а в стеке. Будем использовать кортеж, он хранит свои данные на стеке.

```
tuple<int, int, int, int, int> BuildTuple(int i) {
    return make_tuple(i, i + 1, i + 2, i + 3, i + 4);
}

int main() {
```

```
{ LOG_DURATION("tuple");
    for (int i = 0; i < COUNT; ++i) {
        auto numbers = BuildTuple()
    }
}
return 0;
}
// tuple: 118 ms
```

Программа работает быстрее, но использование `tuple` влечет неудобства: по нему нельзя нормально проитерироваться, нельзя обратиться к элементу с использованием квадратных скобок. Необходим аналог контейнера `vector`, но на стеке. Можем попросить выделить на стеке пять `int`'ов, используя тип `array`.

```
array<int, 5> BuildArray(int i) {
    return {i, i + 1, i + 2, i + 3, i + 4};
}
```

Фрейм функции должен занимать фиксированное количество байт, поэтому прямо в массиве нужно указать, что необходимо 5 `int`'ов. 5 в данном случае – это шаблонный параметр класса. До этого мы сталкивались с классами, у которых шаблонный параметр это тип.

```
{ LOG_DURATION("array");
    for (int i = 0; i < COUNT; ++i) {
        auto numbers = BuildArray()
    }
}
// array: 9 ms
```

Почему массив настолько эффективнее, чем кортеж? Дело в том, что мы компилируем без оптимизаций. Если компиляция происходит без оптимизации, код может быть очень не эффективен. Если скомпилировать код, используя оптимизацию, то результат работы программы для разных контейнеров будет таким:

```
// vector: 127 ms
// tuple: 0 ms
// array: 0 ms
```

Между `array` и `tuple` в данном случае нет разницы.

Сложность алгоритма с использованием каждого контейнера одинакова. Разница во временах

работы программ вытекает из разницы в константе, в накладных расходах, требуемых для каждого контейнера.

Продемонстрируем, что данные массива хранятся на стеке.

```
int main() {
    int x = 111111;
    array<int, 10> numbers;
    numbers.fill(8);
    int y = 222222;

    for (int* p = &y; p <= &x; ++p) {
        cout << *p << " ";
    }
    cout << "\n";

    return 0;
}
// 222222 8 8 8 8 8 8 8 8 123 0 111111
```

В начале идет переменная у, затем идёт десять восьмёрок, затем некоторая служебная информация, и переменная x. Действительно, всё лежит на стеке.

#### 4.1.7. Класс `string_view`

Класс `string` похож на `vector`: он позволяет делать `push_back` и `reserve`. Однако семантика строки иногда отличается от семантики вектора.

Рассмотрим как пример задачу из второго курса, где надо было по пробелам строку разбивать на слова.

```
vector<string> SplitIntoWords(const string& str) {
    vector<string> result;

    auto str_begin = begin(str);
    const auto str_end = end(str);

    while (true) {
        auto it = find(str_begin, str_end, ' ');
        if (it == str_end)
```

```
result.push_back(string(str_begin, it));

    if (it == str_end) {
        break;
    } else {
        str_begin = it + 1;
    }
}

return result;
}
```

Функция итерируется по строке, на каждом шаге ищет пробел, создает строчку от текущего итератора до этого пробела, добавляет её в наш набор строк,двигается после этого на позицию, следующую за пробелом. Получается набор строк, разделённых пробелами, то есть слов. У этого кода есть очевидный недостаток. Можно было бы просто сказать, где в этой строке находятся слова, например, с первого до третьего элемента, с пятого до седьмого и так далее. Вместо этого мы стали создавать новые строки, выделять под них память, складывать эти строки в вектор.

Эта функция может возвращать не вектор, а некоторую ссылку на диапазон символов в строке. Для этого в стандарте C++17 доступен класс `string_view` – это ссылка на где-то хранящийся диапазон символов.

Новая функция будет возвращать `string_view`, принимать строчку `s`, внутри себя будет со-хранять её в `string_view`. Далее появляется проблема: `string_view` не дружит с итераторами. `string_view` ссылается на подряд идущий диапазон символов. Он работает с позициями в стро-ках, а не с итераторами. Мы начинаем не с итератора, а с нулевой позиции.

```
size_t pos = 0;
```

Аналогом итератора, указывающего за конец, является позиция за концом, её можно проинициализировать константой `strnpos` – это большое число, по сути означает несуществующую позицию.

```
const size_t pos_end = strnpos;
```

Будем искать пробел в `string_view str`, начиная с позиции `pos`.

```
size_t space = str.find(' ', pos);
```

Теперь нужно положить в результат подстроку текущего `string_view`, начиная от позиции `pos` и до пробела. Есть метод `substr`, который возвращает `string_view` на подстроку. Нужно начать подстроку с позиции `pos`, второй элемент – длина этой подстроки. Также нужно учесть случай, когда пробел не нашелся и `space` равно `npos`.

```
result.push_back(
    space == pos_end
    ? str.substr(pos)
    : str.substr(pos, space - pos));
```

Добавив очередное слово, нужно проверить, если пробел уже не нашелся, то все слова найдены.

```
if (space == pos_end) {
    break;
} else {
    pos = space + 1;
}
```

Итоговый код:

```
vector<string_view> SplitIntoWordsView(const string& s) {
    string_view str = s;

    vector<string_view> result;

    size_t pos = 0;
    const size_t pos_end = strnpos;

    while (true) {
        size_t space = str.find(' ', pos);
        result.push_back(
            space == pos_end
            ? str.substr(pos)
            : str.substr(pos, space - pos));

        if (space == str_end) {
            break;
        } else {
            pos = space + 1;
        }
    }
}
```

```
    return result;
}
```

Проверим этот код, сравнив его эффективность со старой функцией `SplitIntoWords`. Для этого заготовлена функция `GenerateText`, генерирующая текст из 10000000 букв "а", разбивающая его пробелами на слова через каждые сто символов.

```
int main() {
    const string text = GenerateText;
    { LOG_DURATION("string");
        const auto words = SplitIntoWords(text);
        cout << words[0] << "\n";
    }

    { LOG_DURATION("string_view");
        const auto words = SplitIntoWordsView(text);
        cout << words[0] << "\n";
    }
}

// *слово, состоящее из ста букв "а"*
// string: 188 ms
// *слово, состоящее из ста букв "а"*
// string_view: 22 ms
```

Сделаем код более компактным. Не будем держать единый `string_view` и ходить по нему позицией, а будем двигать начало `string_view`, когда обрабатываем сколько-то слов. В каждый момент `string_view` будет указывать на диапазон ещё необработанных символов. Если `string_view` указывает на ещё необработанные символы, то можно просто искать первый пробел:

```
size_t space = str.find(" ");
```

Подстроку будем брать не от `pos`, а от нулевой позиции. Если хотим взять подстроку от нулевой позиции до какой-то другой, возможно несуществующей, то можно брать субстроку от нуля до `space`.

```
result.push_back(str.substr(0, space));
```

Если пробел нашелся, то `space` – это его позиция, а также расстояние до него. Если пробел не нашелся, то вызываем `substr` от нуля до `pos` и это будет вся строка.

Теперь заканчиваем цикл. Если пробел не нашелся, то есть его позиция `pos`, то надо закончить

цикл, иначе надо откусить от начала `string_view` уже обработанный кусок длины `space + 1` с помощью специального метода `remove_prefix`.

```
if (space == strnpos) {
    break;
} else {
    str.remove_prefix(space + 1);
}
```

Итоговый код:

```
vector<string_view> SplitIntoWordsView(const string& s) {
    string_view str = s;

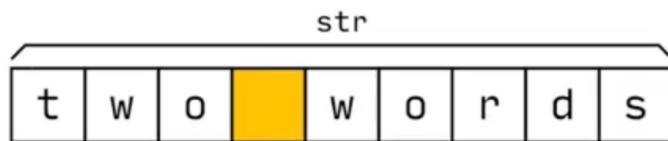
    vector<string_view> result;

    while(true) {
        size_t space = str.find(' ');
        result.push_back(str.substr(0, space));

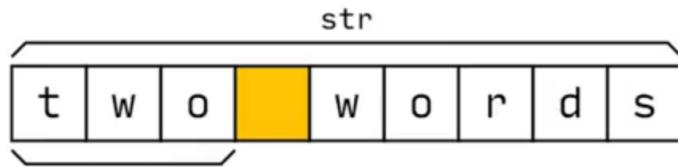
        if (space == strnpos) {
            break;
        } else {
            str.remove_prefix(space + 1);
        }
    }

    return result;
}
```

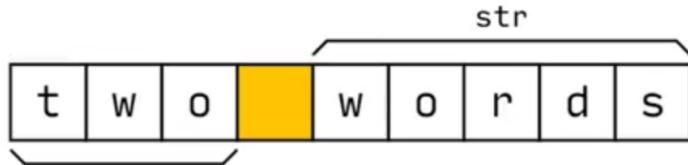
Разберём, как этот код работает. Пусть на вход подается строка "two words".



Мы заходим в цикл, ищем пробел, он находится, его позиция равна трём. Далее вызывается метод `substr` от параметров 0, 3. Он выдаёт строку, которая начинается с нулевой позиции и имеет длину три. Это как раз первое слово.



После этого идёт проверка, не пора ли закончить цикл, откусывается префикс длины 4.



Далее переходим во вторую итерацию цикла, ищем пробел, он не находится. Пробела нет, `space` равно `npos`. Вызывается `substr` от нуля и `npos`, тем самым к ответу добавляется весь текущий `string_view`. Условие `space == str.npos` выполняется, цикл можно закончить.

Вернёмся к первому варианту `SplitIntoWordsView`. Что, если бы мы не создавали `string_view` по строке, а работали бы со строкой, назвав её `str`, вызывая методы `find` и `substr?` Такой код отработает, но не всё однозначно. Если разбить на слова более короткую строчку, например "a b", то в результате получим:

```
// a
// ?
```

Первый вариант функции, который мы использовали ещё во втором курсе, выводит букву `a`. А второй вариант, в котором мы работали со строкой, а не `string_view`, возвращает теперь какой-то символ. Проблема в том, что метод `substr` для строки создаёт новую строчку. Созданные временные объекты могут уничтожиться, если их никуда не сохранить. Получается, мы создали новую строку, сохранили в вектор `string_view` указатель на данные этой строки и тут же эта строка уничтожилась, потому что была временным объектом, то есть указатель `string_view` указывает в никуда. Если `string_view` ссылается на какой-то диапазон символов, следите, чтобы он не уничтожился.

Как сделать правильно и более компактно? Можно сразу в функции принять `string_view` по значению:

```
vector<string_view> SplitIntoWordsView(string_view str) {
    ...
}
```

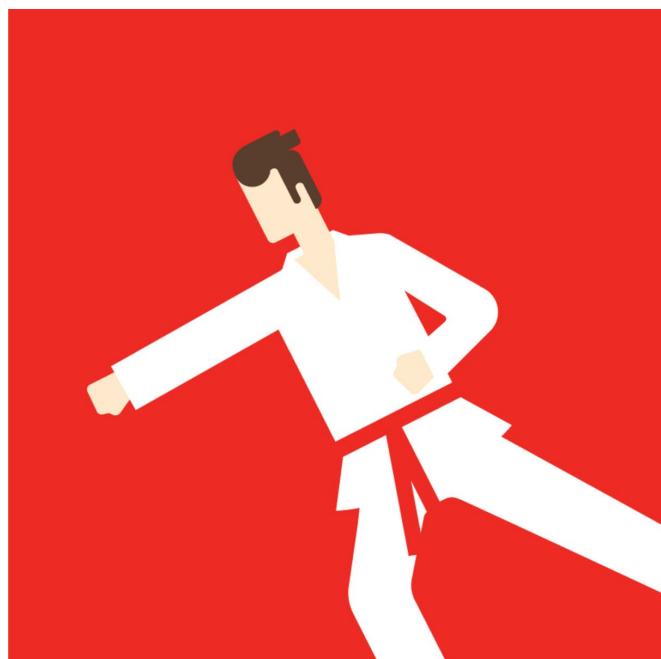
}

Если вы хотите в функции работать со `string_view`, удобно принять в неё `string_view`, потому что вы не получите проблем с методом `substr`, более того, в функцию можно передавать как строку, так и `string_view`. Этот вариант универсальнее.

# Основы разработки на C++: красный пояс

Неделя 5

Move-семантика и базовая многопоточность



# Оглавление

<b>Move-семантика и базовая многопоточность</b>	<b>3</b>
5.1 Move-семантика . . . . .	3
5.1.1 Перемещение временных объектов. . . . .	3
5.1.2 Перемещение в других ситуациях . . . . .	6
5.1.3 Функция <code>move</code> . . . . .	8
5.1.4 Когда перемещение не помогает . . . . .	11
5.1.5 Конструктор копирования и оператор присваивания . . . . .	13
5.1.6 Конструктор перемещения и перемещающий оператор присваивания . . . . .	15
5.1.7 Передача параметра по значению . . . . .	18
5.1.8 Move-итераторы . . . . .	18
5.1.9 Некопируемые типы . . . . .	20
5.1.10 NRVO и copy elision . . . . .	21
5.1.11 Опасности <code>return</code> . . . . .	22
5.2 Базовая многопоточность . . . . .	23
5.2.1 <code>async</code> и <code>future</code> . . . . .	23

5.2.2	Задача генерации и суммирования матрицы . . . . .	24
5.2.3	Особенности шаблона <code>future</code> . . . . .	26
5.2.4	Состояние гонки . . . . .	27
5.2.5	<code>mutex</code> и <code>lock_guard</code> . . . . .	29
5.2.6	<code>&lt;execution&gt;</code> , которого нет . . . . .	31

# Move-семантика и базовая многопоточность

## 5.1. Move-семантика

### 5.1.1. Перемещение временных объектов.

Рассмотрим задачу. Пусть есть функция, которая возвращает тяжёлую строку:

```
string MakeString() {
    return string(100000000, 'a');
}
```

Пусть необходимо добавить эту строчку в некоторый вектор строк.

```
int main() {
    vector<string> strings;
    string heavy_string = MakeString();
    strings.push_back(heavy_string);

    return 0;
}
```

Мы хотим положить результат вызова функции `MakeString()` в вектор. Это можно сделать, не используя переменную `heavy_string`.

```
int main() {
    vector<string> strings;
    strings.push_back(MakeString());

    return 0;
}
```

Замерим время работы каждой реализации, используя макрос LOG\_DURATION.

```
// with variable: 299 ms
// without variable: 168 ms
```

Второй вариант работает быстрее, потому что временный объект – результат вызова функции – не сохраняется в переменную.

Исследуем скорость работы алгоритмов, в которых в вектор добавляется временный объект, созданный с помощью конструктора строки.

```
vector<string> strings;
string heavy_string = string(100000000, 'a');
strings.push_back(heavy_string);

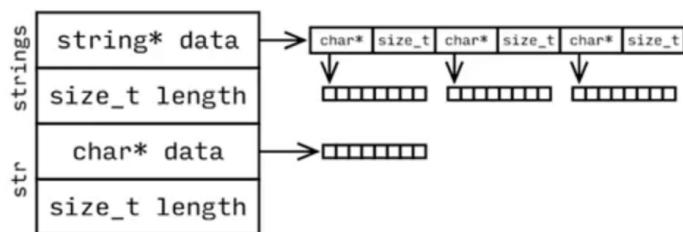
vector<string> strings;
strings.push_back(string(100000000, 'a'));

// ctor: with variable: 201 ms
// ctor: without variable: 122 ms
```

Без использования промежуточной переменной код работает быстрее. Во втором варианте в push\_back передаётся временный объект, он забирает данные этого объекта, не копируя. Подробнее разберёмся, как это работает.

Рассмотрим процесс добавления в вектор переменной. Вспомним, как хранятся данные вектора и строки в памяти.

```
vector<string> strings(3);
string str(100'000'000, 'a');
strings.push_back(str);
```

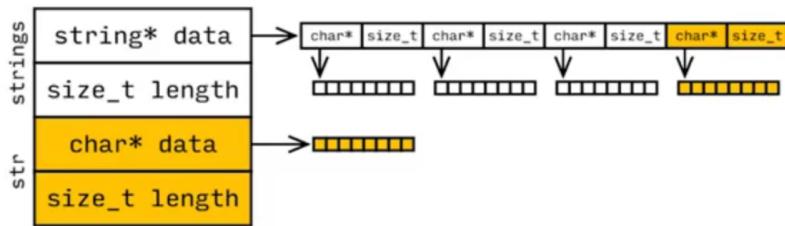


Локальная переменная strings, которая является вектором, и локальная переменная str со строкой хранятся на стеке. Вектор представляется указателем на свои данные в куче и длиной.

Кроме того, есть локальная переменная со строкой, которая представляет собой указатель на данные в куче и длину. У вектора в куче хранятся строки подряд, где каждая строка – указатель на свои данные и длину.

`push_back`, вызванный от локальной переменной, должен скопировать данные этой строки.

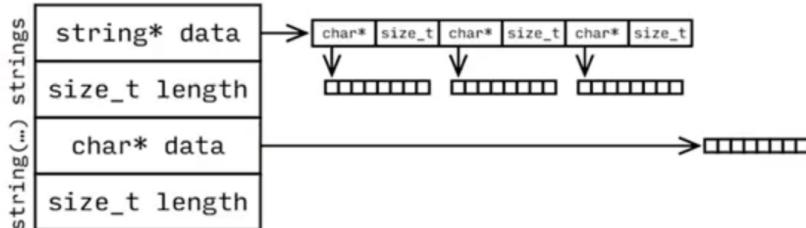
```
vector<string> strings(3);
string str(100'000'000, 'a');
strings.push_back(str);
```



Выделяем память под восемь символов, в вектор кладём указатель на них.

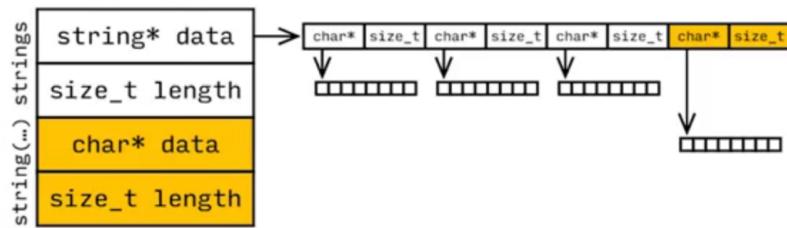
Если мы хотим добавить в вектор временный объект, то его данные в куче, на которые он ссылается, никому не будут нужны. Если их никто не заберёт, то они просто уничтожатся, потому что объект временный.

```
vector<string> strings(3);
strings.push_back(string(100'000'000, 'a'));
```



`push_back` имеет право эти данные не копировать, а просто свой указатель, который добавится в данные вектора, направить на эти данные, а у той самой локальной временной строки эти данные отобразить.

```
vector<string> strings(3);
strings.push_back(string(100'000'000, 'a'));
```



### 5.1.2. Перемещение в других ситуациях

Разберём ещё один класс ситуаций, когда у нас не происходит копирование данных временного объекта. В языке C++ везде, где может происходить копирование, имеет место и перемещение. Вспомним, где может происходить копирование объектов. Самый простой случай – это присваивание.

```
string target_string = "old_value";
string source_string = MakeString();
target_string = source_string;
```

В таком варианте здесь будет происходить копирование. Здесь тоже можно убрать промежуточную переменную `source_string` и сэкономить ресурсы на копировании.

```
string target_string = "old_value";
target_string = MakeString();
```

Замерим время работы программы с присваиванием в промежуточную переменную и без промежуточной переменной.

```
// assignment, with variable: 243 ms
// assignment, without variable: 119 ms
```

При использовании метода `set::insert` тоже можно обойтись без промежуточной переменной.

```
// set::insert, with variable
set<string> strings;
string heavy_string = MakeString();
strings.insert(heavy_string);
```

```
// set::insert, without variable
set<string> strings;
strings.insert(MakeString());

// set::insert, with variable: 217 ms
// set::insert, without variable: 100 ms
```

Теперь рассмотрим пример со словарём. Промежуточную переменную создаём и для ключа, и для значения.

```
map<string, string> strings;
string key = MakeString();
string value = MakeString();
strings[key] = value;
// map::operator[], with variables: 474 ms
```

Избавимся от промежуточной переменной для значения.

```
map<string, string> strings;
string key = MakeString();
strings[key] = MakeString();
// map::operator[], with variable for key: 305 ms
```

Теперь избавимся и от промежуточной переменной для ключа.

```
map<string, string> strings;
strings[MakeString()] = MakeString();
// map::operator[], without variables: 210 ms
```

Напишем функцию MakeVector(), создающую вектор, и попробуем этот вектор куда-нибудь положить.

```
vector<int> MakeVector() {
    return vector<int>(100000000, 0);
}
```

Попробуем положить такой вектор во множество.

```
// set::insert for vector, with variable
set<vector<int>> vectors;
vector<int> heavy_vector = MakeVector();
vectors.insert(heavy_vector);
```

```
// set::insert for vector, without variable
set<vector<int>> vectors;
vectors.insert(MakeVector());

// set::insert for vector, with variable: 1062 ms
// set::insert for vector, without variable: 386 ms
```

Если у вас есть некоторый временный объект, старайтесь не потерять это важное свойство – временность. Так компилятор сэкономит на копировании.

### 5.1.3. Функция `move`

Есть ситуации, когда есть невременный объект, но про него известно, что там, где он создан, он больше не понадобится.

Есть задача – считать набор строк из потока ввода и эти строки положить в вектор. Это делает функция `ReadStrings()`.

```
vector<string> ReadStrings(istream& stream) {
    vector<string> strings;
    string s;
    while (stream >> s) {
        strings.push_back(s);
    }
    return strings;
}
```

После того как мы кладем строчку `s` в вектор, она нам больше не понадобится. Хочется, чтобы метод `push_back` вел бы себя в этом случае как с временным объектом. Мы хотим изменить семантику объекта.

Есть функция `move`, которая может изменить семантику даже постоянного объекта. Чтобы использовать функцию `move`, нужно подключить модуль `utility`. Цикл теперь выглядит так:

```
while (stream >> s) {
    strings.push_back(move(s));
}
```

Проверим работу функции:

```
int main() {
    for (const string& s : ReadStrings(cin)) {
        cout << s << "\n";
    }
    return 0;
}
// ввод: Red belt C++
// вывод:
// Red
// belt
// C++
```

Разницу между реализациями с move и без можно определить не только по времени работы кода, но и по содержанию переменной s. Посмотрим, что лежит в переменной s до push\_back'a и после него. Также посмотрим, что лежит в конце вектора.

```
while (stream >> s) {
    cout << "s = " << s << "\n";
    strings.push_back(s);
    cout << "s = " << s <<
        ", strings.back() = " << strings.back() << "\n";
}
// ввод: a b c
// вывод:
// s = a
// s = a, strings.back() = a
// s = b
```

Буква a скопировалась. Если добавить вызов move в метод push\_back:

```
// ввод: a b c
// вывод:
// s = a
// s = , strings.back() = a
// s = b
```

Метод push\_back забрал данные из строки s.

Покажем, что код ускоряется. Для этого напишем функцию, которая умеет как перемещать, так и не перемещать.

```
vector<string> ReadStrings(istream& stream, bool use_move) {
    vector<string> strings;
    string s;
    while (stream >> s) {
        if (use_move) {
            strings.push_back(move(s));
        } else {
            strings.push_back(s);
        }
    }
    return strings;
}
```

Воспользуемся функцией `GenerateText()`. Она генерирует текст из миллиарда символов и разбивает его пробелами через каждые десять миллионов символов.

```
int main() {
    const string text = GenerateText();
    { LOG_DURATION("without move");
        istringstream stream(text);
        ReadStrings(stream, false);
    }
    { LOG_DURATION("with move");
        istringstream stream(text);
        ReadStrings(stream, true);
    }
}
// without move: 26359 ms
// with move: 17586 ms
```

Получили существенное ускорение за счет избавления от лишнего копирования.

Этим примером область применения функции `move` не ограничивается. Бывают ситуации, когда результат вызова функции разбиения строки на слова должен жить дольше чем исходная строка. Рассмотрим следующую реализацию функции `SplitIntoWords`:

```
vector<string> SplitIntoWords(const string& text) {
    vector<string> words;
    string current_word;
    for (const char c : text) {
        if (c == ' ') {
```

```

        words.push_back(current_word); // вызываем push_back от переменной, которая нам
        // дальше не нужна
        // логично будет обернуть её в move
        current_word.clear();
    } else {
        current_word.push_back(c);
    }
}
words.push_back(current_word);
return words;
}

```

#### 5.1.4. Когда перемещение не помогает

Если у объекта нет данных в куче, а основные данные на стеке, то данные придётся копировать. Массив хранит свои данные на стеке. Продемонстрируем на его примере, что перемещение не помогает ускорить работу с ним.

```

const int SIZE = 10000;

array<int, SIZE> MakeArray() {
    array<int, SIZE> a;
    a.fill(8);
    return a;
}

```

Создадим вектор массивов и положим массив в вектор десять тысяч раз с использованием промежуточной переменной и без использования промежуточной переменной.

```

int main() {
{ LOG_DURATION("with variable");
    vector<array<int, SIZE>> arrays;
    for (int i = 0; i < 10000; ++i) {
        auto heavy_array = MakeArray();
        arrays.push_back(heavy_array);
    }
}
{ LOG_DURATION("without variable");
    vector<array<int, SIZE>> arrays;

```

```
    for (int i = 0; i < 10000; ++i) {
        arrays.push_back(MakeArray());
    }
}

return 0;
}
// with variable: 1191 ms
// without variable: 1148 ms
```

Ускорения не произошло. Рассмотрим другой пример.

```
string MakeString() {
    return "C++";
}

int main() {
    vector<string> strings;
    string s = MakeString();
    cout << s << "\n";
    strings.push_back(s);
    cout << s << "\n";

    return 0;
}
// C++
// C++
```

Теперь сделаем переменную `s` константной:

```
const string s = MakeString();

// C++
// C++
```

Обернём строчку `s` в `move`:

```
strings.push_back(move(s));

// C++
// C++
```

Перемещения строки не произошло. Стока константная, перемещение из неё не сработает.

Вызов `move` для константного объекта бесполезен. Следите за константностью перемещаемого объекта.

### 5.1.5. Конструктор копирования и оператор присваивания

Обсудим, что происходит в следующих ситуациях:

```
vector<int> source = /* ... */;
vector<int> target = source;

vector<int> source2 = /* ... */;
target = source2;
```

Зачем это необходимо:

- Научиться перемещать собственные классы;
- Разговаривать с разработчиками на одном языке;
- Читать документацию и понимать, где может происходить перемещение;
- Развить интуицию относительно `move`-семантики.

В следующих случаях вызывается конструктор копирования (copy constructor):

```
vector<int> source = /* ... */;
vector<int> target = source;
vector<int> target2(source);
```

В следующих случаях вызывается оператор копирующего присваивания (copy assignment operator):

```
vector<int> source = /* ... */;
vector<int> target = /* ... */;
target = source;
target.operator=(source);
```

Познакомимся поближе с этими методами на примере класса `Logger`, который будет логировать вызовы этих методов.

```
class Logger {
public:
    Logger() { cout << "Default ctor\n"; } // конструктор по умолчанию
    Logger(const Logger&) { cout << "Copy ctor"; } // конструктор копирования
    void operator=(const Logger&) { cout << "Copy assignment\n"; } // оператор
    // присваивания
};

int main() {
    Logger source; // вызывается конструктор по умолчанию
    Logger target = source; // вызывается конструктор копирования

    return 0;
}
// Default ctor
// Copy ctor

vector<Logger> loggers;
loggers.push_back(target);
// Copy ctor

source = target; // вызывается оператор присваивания
// Copy assignment
```

Для собственных типов при необходимости компилятор сам генерирует конструктор копирования и оператор присваивания, которые просто копируют все поля.

Для типов, которые самостоятельно управляют памятью, нужно самостоятельно реализовывать копирование и присваивание.

Напишем конструктор копирования для класса `SimpleVector`.

Добавим конструктор от константной ссылки на `SimpleVector`.

```
SimpleVector(const SimpleVector& other);
```

Реализуем этот конструктор:

```
template <typename T>
```

```
SimpleVector<T>::SimpleVector(const SimpleVector<T>& other)
: data(new T[other.capacity]),
 size(other.size),
 capacity(other.capacity)
{
 copy(other.begin(), other.end(), begin());
}
```

### 5.1.6. Конструктор перемещения и перемещающий оператор присваивания

В следующих случаях вызывается конструктор перемещения (move constructor):

```
vector<int> source = /* ... */;
vector<int> target = move(source);
vector<int> target2(move(target));

vector<vector<int>> vectors;
vectors.push_back(vector<int>(5));
```

В следующем случае инициализация временным объектом оптимизируется без вызова конструктора перемещения. Этот случай особый, его обсудим позже.

```
vector<int> MakeVector();

vector<int> target = MakeVector();
```

В следующих случаях вызывается оператор перемещающего присваивания (move assignment operator):

```
vector<int> source = /* ... */;
vector<int> target = /* ... */;
target = move(source);
target = vector<int>(5);
```

Если у вас есть собственный класс с конструктором копирования, но без конструктора перемещения, то компилятор делает перемещение эквивалентным копированию. Рассмотрим пример с классом `SimpleVector`. Скажем компилятору самостоятельно сгенерировать конструктор перемещения.

```
// rvalue reference
SimpleVector(const SimpleVector&& other) = default;
```

`rvalue reference` ведёт себя как обыкновенная ссылка, но позволяет принимать временные объекты.

```
int main() {
    SimpleVector<int> source(1);
    SimpleVector<int> target = move(source);
    cout << source.Size() << " " << target.Size() << endl;
    return 0;
}
// 1 1
```

Такой код падает в конце, на деструкторах. Придётся самостоятельно реализовывать конструктор перемещения.

```
template <typename T>
SimpleVector<T>::SimpleVector(SimpleVector<T>&& other)
    : data(other.data),
    size(other.size),
    capacity(other.capacity)
{
    other.data = nullptr;
    other.size = other.capacity = 0;
}

// 0 1
```

Теперь у объекта, из которого мы перемещали, размер 0, а у объекта, в который мы перемещали – размер 1.

Реализуем оператор перемещающего присваивания.

```
void operator=(SimpleVector&& other);
```

Он будет реализован примерно как конструктор перемещения.

```
template <typename T>
void SimpleVector<T>::operator=(SimpleVector<T>&& other) {
    delete[] data;
    data = other.data;
    size = other.size;
    capacity = other.capacity;

    other.data = nullptr;
    other.size = other.capacity = 0;
}
```

Проверим конструктор перемещения.

```
int main() {
    SimpleVector<int> source(1);
    SimpleVector<int> target(1);
    target = move(source);
    cout << source.Size() << " " << target.Size() << endl;
    return 0;
}
// 0 1
```

Перегрузка по rvalue-ссылке – способ отличить временный объект от постоянного.

```
target = source;
// ↑ вызывается operator=(const vector<int>&
target = vector<int>(5);
// ↑ вызывается operator=(vector<int>&&)
target = move(source);
// ↑ вызывается operator=(vector<int>&&)
```

При необходимости компилятор сам генерирует конструктор перемещения и оператор перемещающего присваивания, которые просто перемещают все поля. Если класс не управляет памятью самостоятельно, то перемещение для него будет просто работать.

### 5.1.7. Передача параметра по значению

Оптимизируем методы `ChangeFirstName` и `ChangeLastName` из класса `Person`, который реализовался в задаче «Имена и фамилии-4»

```
void ChangeFirstName(int year, const string& first_name) {
    first_names[year] = first_name;
}
void ChangeLastName(int year, const string& last_name) {
    last_names[year] = last_name;
}
```

Необходимо принимать строки по значению, а не по ссылке. Внутри функции будем перемещать строку внутрь словаря.

```
void ChangeFirstName(int year, string first_name) {
    first_names[year] = move(first_name);
}
void ChangeLastName(int year, string last_name) {
    last_names[year] = move(last_name);
}
```

Возможны два случая. Если мы вызываем `ChangeFirstName` от временного объекта, то он проинициализирует переменную `first_name`, поскольку объект временный, то для него вызовется конструктор перемещения. Затем мы снова вызовем перемещение, переместим `first_name` внутрь контейнера. Будет два перемещения.

Если мы вызываем этот метод не от временного объекта, тогда этот объект скопируется в аргумент функции, затем случится перемещение этого объекта внутрь контейнера.

Итого, приняв параметр функции по значению, мы можем сделать его универсальным, вызывать как от временных объектов, так и от постоянных.

### 5.1.8. Move-итераторы

Рассмотрим конструкцию, позволяющую сделать использование `move`-семантики более простым.

Рассмотрим задачу `SplitIntoSentences`, в которой нужно было написать функцию, принимающую набор токенов, разбивающую их на предложения. Рассмотрим саму функцию `SplitIntoSentences`:

```

template <typename Token>
vector<Sentence<Token>> SplitIntoSentences(
    vector<Token>> tokens) {
    vector<Sentence<Token>> sentences;
    auto tokens_begin = begin(tokens);
    while (tokens_begin != tokens_end) {
        const auto sentence_end =
            FindSentenceEnd(tokens_begin, tokens_end);
        Sentence<Token> sentence;
        for (; tokens_begin != sentence_end; ++tokens_begin) {
            sentence.push_back(move(*tokens_begin));
        }
        sentences.push_back(move(sentence));
    }
    return sentences;
}

```

В ней мы заводим итератор `tokens_begin` и идём этим итератором по токенам, находим конец очередного предложения. Берем текущий токен, идем итератором от текущего токена до конца предложения, все эти токены вставляем в текущее предложение. Затем это предложение вставляем в вектор предложений.

Рассмотрим работу цикла `for`. Он перебирает токены в их диапазоне, каждый из них вставляет в вектор.

Подключим модуль `iterator`. После этого станет доступна специальная функция, которую мы вызовем от итераторов. Она называется `make_move_iterator`. Такая функция возвращает обёртку над итератором, которая, если мы хотим скопировать объект, перемещает его. Функция `make_move_iterator` меняет семантику итератора, чтобы при обращении к нему данные перемещались бы, а не копировались.

Используя `make_move_iterator`, можем заменить этот кусок...

```

Sentence<Token> sentence;
for (; tokens_begin != sentence_end; ++tokens_begin) {
    sentence.push_back(move(*tokens_begin));
}
sentences.push_back(move(sentence));

```

...на такой:

```
sentences.push_back(Sentence<Token>(
    make_move_iterator(tokens_begin),
    make_move_iterator(sentence_end)
));
```

Осталось в конце менять итератор `tokens_begin`:

```
tokens_begin = sentence_end;
```

Рассмотрим задачу «Считалка Иосифа». Рассмотрим следующий цикл:

```
for (uint32_t i = 0; i < range_size; ++i, ++range_begin) {
    *range_begin = move(permutation[i]);
}
```

Без цикла это можно записать так:

```
copy(
    make_move_iterator(begin(permutation)), make_move_iterator(end(permutation)),
    range_begin);
);
```

Алгоритм `move` будет перемещать данные, а не копировать. Это избавит от необходимости вызывать `make_move_iterator`.

```
move(begin(permutation), end(permutation), range_begin);
```

### 5.1.9. Некопируемые типы

Продемонстрируем, как можно сделать тип, который не будет уметь копировать на примере `Logger`'а. Нужно написать `delete` вместо тела конструктора копирования:

```
Logger(const Logger&) = delete;
```

В языке есть такие типы, которые копировать бессмысленно, например, потоки ввода и вывода. Посмотрим, как с ними обращаться на примере вектора потоков.

```
vector<ofstream> streams;
streams.reserve(5);
```

```
for (int i = 0; i < 5; ++i) {
    ofstream stream(to_string(i) + ".txt");
    stream << "File #" << i << "\n";
    streams.push_back(move(stream));
}
for (auto& stream : streams) {
    stream << "Vector is ready!" << endl;
}
```

Откроем, например, файл «0.txt». Там написано:

```
// File #0
// Vector is ready!
```

Получилось работать с файловыми потоками, несмотря на то, что это некопируемые объекты.

### 5.1.10. NRVO и copy elision

Можно оптимизировать копирование объектов, у которых данные только на стеке. Продемонстрируем это на примере Logger'a. Пусть у нас есть функция MakeLogger():

```
Logger MakeLogger() {
    return Logger(); // temporary -> returned temporary
}
```

Временным объектом Logger() мы инициализируем тот промежуточный объект, который должен вернуться из функции. В функции main мы из временного объекта, который вернулся из функции, инициализируем переменную:

```
int main() {
    Logger logger = MakeLogger(); // temporary -> variable

    return 0;
}
// Default ctor
```

Поскольку в обоих случаях новый объект инициализируется временным объектом, происходит перемещение.

В результате работы кода вызывался только конструктор по умолчанию, конструкторы копирования и перемещения не вызвались. Дело в том, что в некоторых случаях компилятор умеет оптимизировать перемещение. Например:

- при возвращении из функции временного объекта;
- при инициализации нового объекта временным объектом.

Такая оптимизация называется copy elision.

При возвращении локальной переменной из функции перемещение и копирование опускаются. Такая оптимизация называется named return value optimization.

### 5.1.11. Опасности `return`

Рассмотрим два случая, когда `return` оптимизируется не так хорошо, как в случаях из предыдущего пункта.

```
pair<ifstream, ofstream> MakeStreams(const string& prefix) {
    ifstream input(prefix + ".in");
    ofstream output(prefix + ".out");
    return {input, output};
}
```

Код не компилируется, компилятор не может составить пару потоков. `input` и `output` передаются в конструктор пары, затем созданная с помощью этого конструктора пара должна проинициализировать возвращаемый временный объект. Поскольку эта пара тоже временная, то инициализация временного объекта, возвращаемого из функции этой парой, происходит безболезненно, но передача переменных `input` и `output` в конструктор пары происходит по обычным правилам языка C++. Чтобы решить проблему, следует обернуть `input` и `output` в `move`.

Если функция должна вернуть некоторый объект, например поток ввода:

```
ifstream MakeInputStream(const string& prefix) {
}
```

Внутри функции получили пару объектов, вернуть ходим только один её элемент.

```
ifstream MakeInputStream(const string& prefix) {
    auto streams = MakeStreams(prefix);
    return streams.first;
}
```

`streams.first` не является временным объектом. Также это выражение не является названием локальной переменной – это поле локальной переменной.

Проблема решается, если обернуть `streams` в move.

## 5.2. Базовая многопоточность

### 5.2.1. `async` и `future`

Напишем функцию, которая будет суммировать элементы двух векторов. В однопоточной синхронной версии наша функция будет выглядеть так:

```
int SumToVectors(const vector<int>& one,
                  const vector<int>& two) {
    return accumulate(begin(one), end(one), 0)
        + accumulate(begin(two), end(two), 0);
}
```

В начале мы находим сумму элементов одного вектора, потом сумму элементов другого вектора. Мы могли бы один вектор суммировать асинхронно, другой вектор суммировать одновременно с первым, потом сложить результаты. Понадобится заголовочный файл `future`:

```
#include <future>
```

Вызываем функцию `async`, она запускает асинхронную операцию. В данном случае возвращается результат суммирования вектора `one`.

```
future<int> f = async([] {
    return accumulate(begin(one), end(one), 0);
});
```

Далее в переменную `result` присваиваем сумму элементов второго вектора.

```
int result = accumulate(begin(two), end(two), 0);
```

Далее возвращаем `result` плюс то, что возвращает `async`. `async` возвращает `future`.

```
return result + f.get();
```

Такой код не компилируется, потому что не захвачена переменная `one`.

```
future<int> f = async([&one] {
    return accumulate(begin(one), end(one), 0);
});
```

Когда мы пишем `one` в квадратных скобках лямбды, то происходит копирование внутри лямбда-функции. Чтобы он не копировался, его следует передавать по ссылке.

### 5.2.2. Задача генерации и суммирования матрицы

У нас есть матрица:

```
vector<vector<int>> matrix;
```

Она генерируется с помощью функции `GenerateSingleThread`:

```
vector<vector<int>> GenerateSingleThread(size_t size) {
    vector<vector<int>> result(size);
    GenerateSingleThread(result, 0, size);
    return result;
}
```

Функция вызывает другую функцию `GenerateSingleThread`, которая является шаблоном.

```
template <typename ContainerOfVectors>
void GenerateSingleThread(
    ContainerOfVectors& result,
    size_t first_row,
    size_t column_size
) {
    for (auto& row : result) {
        row.reserve(column_size);
    }
}
```

```
    for (size_t column = 0; column < column_size; ++column) {
        row.push_back(first_row ^ column);
    }
    ++first_row;
}
}
```

Далее матрица обрабатывается с помощью функции SumSingleThread.

Запустим программу:

```
int main() {
    LOG_DURATION("Total");
    const size_t matrix_size = 7000;

    vector<vector<int>> matrix;
    {
        LOG_DURATION("Single thread generation");
        matrix = GenerateSingleThread(matrix_size);
    }
    {
        LOG_DURATION("Single thread sum");
        cout << SumSingleThread(matrix) << endl;
    }

}
// Single thread generation: 1254 ms
// Single thread sum: 375 ms
// 195928050144
// Total: 1651 ms
```

Мы хотим ускорить программу. Попробуем генерировать матрицу многопоточно. Напишем функцию GenerateMultiThread. Она будет принимать page\_size – желаемый размер страницы, который будет передаваться потоку.

```
vector<vector<int>> GenerateMultiThread(
    size_t size, size_t page_size
) {
    vector<vector<int>> result(size);
    return result;
}
```

Мы хотим разбивать вектор `result` на несколько частей. Здесь подходит шаблон `Paginator`.

```
vector<vector<int>> GenerateMultiThread(
    size_t size, size_t page_size
) {
    vector<vector<int>> result(size);
    vector<future<void>> futures;
    size_t first_row = 0;
    for (auto page : Paginate(result, page_size)) {
        futures.push_back(
            async([page, first_row, size] {
                GenerateSingleThread(page, first_row, size);
            })
        );
        first_row += page_size;
    }

    return result;
}
```

Теперь в `main()` будем вызывать многопоточный генератор.

```
{
    LOG_DURATION("Multi thread generation");
    matrix = GenerateMultiThread(matrix_size, 2000);
}

// Single thread generation: 1229 ms
// Multi thread generation: 611 ms
// Single thread sum: 365 ms
// 195928050144
// Total: 2345 ms
```

Многопоточная генерация матрицы оказалась в два раза быстрее, чем однопоточная.

### 5.2.3. Особенности шаблона `future`

Обратим внимание на вектор `futures` из функции `GenerateMultiThread`. Мы его объявили, сложили в него результаты вызова `async` и больше его не вызывали. Сохраняя результаты в вектор,

мы откладываем вызов их деструктора, в котором вызывается `get()`. Если результат вызова функции `async` не сохранить в переменную, то программа может выполняться последовательно.

#### 5.2.4. Состояние гонки

Разработаем класс `Account`:

- Он представляет собой банковский счёт;
- Не должен позволять тратить больше денег, чем есть на счету;
- Не должен допускать, чтобы баланс счёта стал отрицательным.

```
struct Account {  
    int balance = 0;  
  
    bool Spend(int value) {  
        if (value <= balance) {  
            balance -= value;  
            return true;  
        }  
        return false;  
    };  
};
```

Напишем функцию, которая будет пытаться 100000 раз потратить один рубль. Она возвращает количество потраченных денег.

```
int SpendMoney (Account& account) {  
    int total_spent = 0;  
    for (int i = 0; i < 100000; ++i) {  
        if (account.Spend(1)) {  
            ++total_spent;  
        }  
    }  
    return total_spent;  
}
```

```
int main() {
    Account my_account(100000);

    cout << "Total spent: " << SpendMoney(my_account)
        << "Balance: " << my_account.balance << endl;
}

// Total spent: 100000
// Balance: 0
```

Пусть теперь несколько людей тратят деньги с семейного счёта асинхронно.

```
int main() {
    Account family_account(100000);

    auto husband = async(SpendMoney, ref(family_account));
    auto wife = async(SpendMoney, ref(family_account));
    auto son = async(SpendMoney, ref(family_account));
    auto daughter = async(SpendMoney, ref(family_account));

    int spent = husband.get() + wife.get() + son.get()
        + daughter.get();

    cout << "Total spent: " << spent << endl
        << "Balance: " << family_account.balance << endl;
}

// Total spent: 140573
// Balance: 0
```

Семья потратила больше денег, чем изначально лежало на счёте.

Если несколько потоков обращаются к одной и той же переменной, целостность данных может быть нарушена. Класс `Account` поддерживал инвариант: баланс никогда не становится отрицательным, мы не можем потратить больше, чем есть на счету. Этот инвариант был нарушен при одновременном обращении к данным. В этом заключается гонка данных. Чтобы её избежать, необходимо выполнять синхронизацию доступа к данным из нескольких потоков.

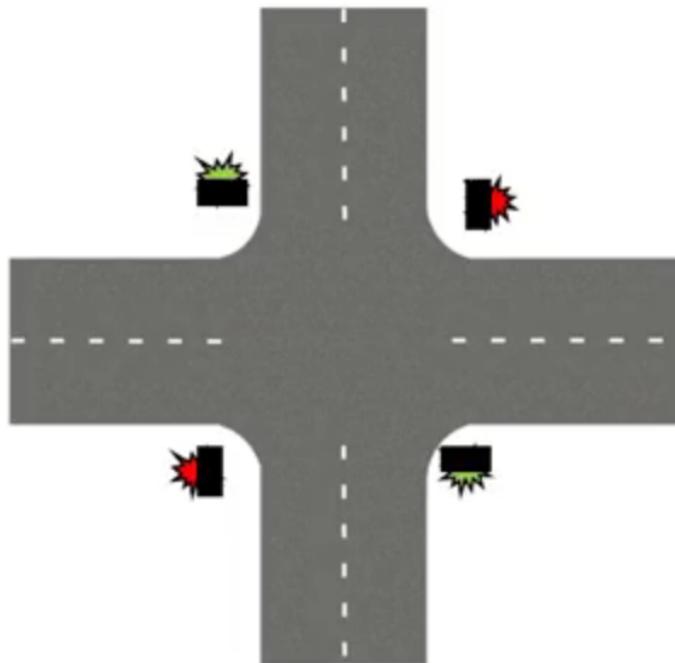
### 5.2.5. mutex и lock\_guard

Добавим в наш класс Account vector<int> history, который будет сохранять все транзакции.

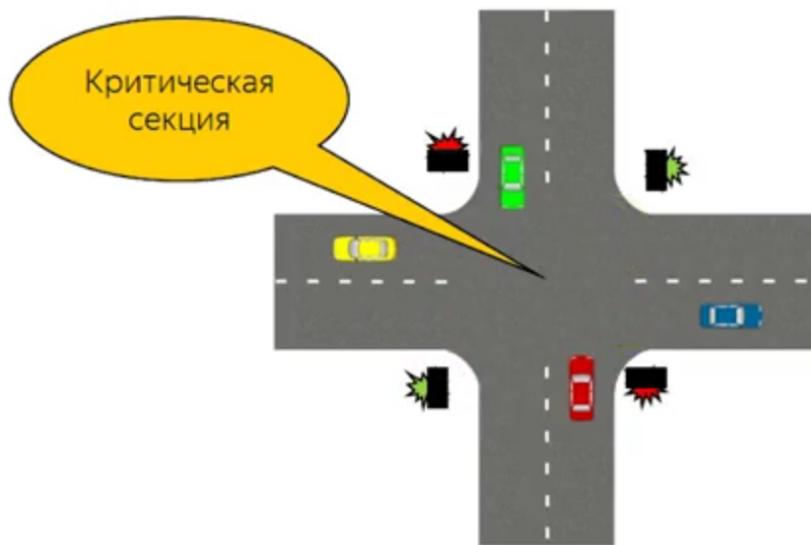
```
struct Account {
    int balance = 0;
    vector<int> history;

    bool Spend(int value) {
        if (value <= balance) {
            balance -= value;
            history.push_back(value);
            return true;
        }
        return false;
    };
};
```

Текущий код нужно адаптировать для работы нескольких потоков, используя Mutex (MUTual EXclusion). Простым примером мьютекса является перекрёсток.



Мьютекс защищает критическую секцию. В программировании это тот участок кода, который в любой момент времени может выполнять не более одного потока.



Применим мьютекс в нашей программе. Подключим заголовочный файл `mutex`. В классе `Account` объявим поле

```
mutex m;
```

Критической секцией в нашем коде является метод `Spend`. Его нужно защитить мьютексом.

```
bool Spend(int value) {
    lock_guard<mutex> g(m);
    if (value <= balance) {
        balance -= value;
        history.push_back(value);
        return true;
    }
    return false;
};
```

Теперь программа работает правильно:

```
// Total spent: 100000
// Balance: 0
```

### 5.2.6. <execution>, которого нет

Вернёмся к примеру с генерацией и суммированием элементов матрицы. Изменим реализацию функции `GenerateSingleThread`. Заменим цикл `for` на алгоритм `for_each`. По сути он делает то же самое.

```
void GenerateSingleThread(
    ContainerOfVectors& result,
    size_t first_row,
    size_t column_size
) {
    for_each (
        begin(result),
        end(result),
        [&first_row, column_size] (vector<int>& row) {
            row.reverse(column_size);
            for (size_t column = 0; column < column_size; ++column) {
                row.push_back(first_row ^ column);
            }
            ++first_row;
        }
    );
}
```

В стандарте C++17 были введены параллельные версии стандартных алгоритмов. Чтобы получить параллельную версию алгоритма `for_each` достаточно подключить заголовочный файл `execution` и в качестве первого параметра в функции указать `execution::par`.

```
for_each (execution::par, ...)
```

Ни один из ведущих компиляторов на момент записи видео (апрель 2018) не реализовал поддержку параллельных версий алгоритмов. Сейчас этим воспользоваться нельзя.