Конспект №1.  Работа с аргументами командной строки в программах на языке C++.

# При запуске программы из командной строки, ей можно передавать дополнительные параметры в текстовом виде. В программе эти параметры из командной строки можно получить через аргументы функции main при использовании функции main в следующей форме:

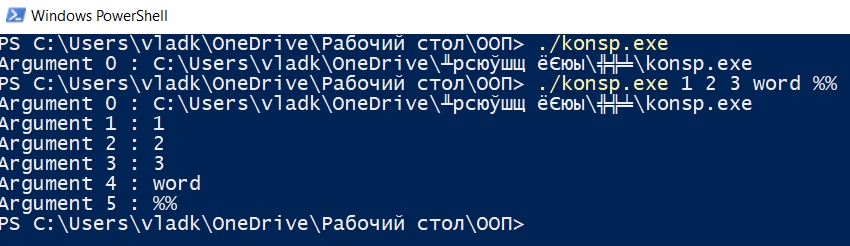


# argc – это целочисленный параметр, содержащий **счетчик** числа аргументов, переданных программе (argc = **arg**ument **c**ount, число аргументов).  argc всегда будет не меньше 1 потому, что первым аргументом всегда является имя самой программы. Каждый аргумент командной строки, предоставляемый пользователем, приведет к увеличению argc на 1.

# argv – это **место**, где хранятся фактические значения аргументов. (argv = **arg**ument **v**alues, значения аргументов, хотя на самом деле расшифровывается как «argument vectors»).

# Пример:





Источники:   
[Аргументы командной строки](https://radioprog.ru/post/1201)  
[Параметры командной строки в С++](https://code-live.ru/post/cpp-command-line-arguments/)

Отчёт №1. Как программировать в C++ через консоль. Компилятор GCC.

## Настроить консоль, найдя в поисковике windows «Изменение переменных среды текущего пользователя».

## Найти переменную Path и нажать на нее лкм 2 раза.

## Выбрать кнопку «Создать» и внести путь до компилятора g++, обычно это C:\MinGW\bin.

## Включить консоль Windows PowerShell и проверить, что g++ работает. Вместо ошибки на первом скриншоте должна выйти любая другая.

## 

## Создаем файл с расширением cpp. Пример: <название>.cpp.

## Пишем код.

## Открываем папку, в которой храним этот файл, нажав сочетание shift+пкм.

## В консоли прописываем: g++ <название>.cpp -o <название>.exe

## В этой же папке должен появится файл название.exe.

## Теперь можно запускать программу в консоли с помощью команды ./<название>.exe

##### Сделанная программа:



Как компилировать в командной строке несколько файлов и программировать с модулями?

# Модуль логически состоит из двух файлов:

# С исходным кодом (module.cpp). Включает в себя определения функций, а также определения глобальных переменных и констант (если они есть); в первой строчке обычно подключается заголовочный файл того же модуля: #include "module.h".

# Заголовочного (module.h). Включает в себя прототипы функций, определения констант, объявления глобальных переменных - но только для тех элементов модуля, о которых должны знать другие модули.

# Главный модуль программы обычно содержит только функцию main и не имеет заголовочного файла (поскольку функция main не используется в других модулях).

# Разделяем программу на модули. Функции переносим в модуль calc.cpp.

# определения функций в calc.h

# 

# главная программа остается в главном модуле main.cpp

# 

# В файле main.cpp нужно написать #include "calc.h" //модуль с вычислениями

# Команда в командной строке для того чтобы скомпилировать несколько файлов: g++ main.cpp calc.cpp –o main.exe

Конспект №2.  Значения аргументов функции по умолчанию.

# Аргумент по умолчанию – это такой аргумент функции, который программист может не указывать при вызове функции. Аргумент по умолчанию добавляется компилятором автоматически.

##### Общая форма объявления функции, которая содержит аргументы по умолчанию:

#### **returned\_type** FunName(**type1** v1 = val1, **type2** v2 = val2, ..., **typeN** vN = valN)

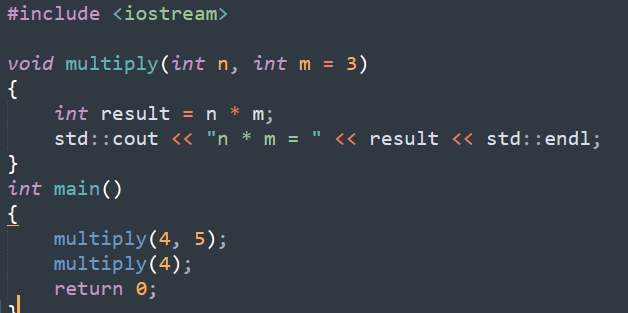
#### {

#### // ...

#### }

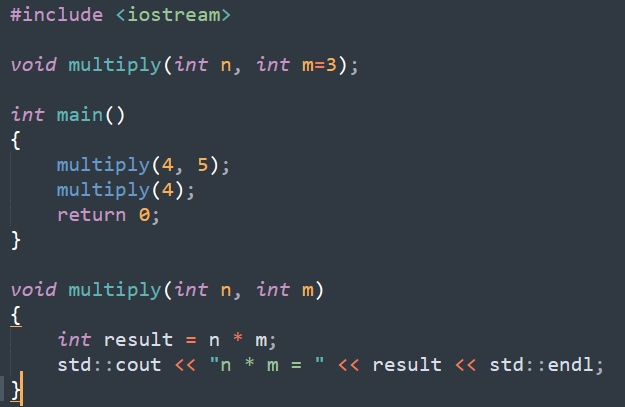
# returned\_type – тип возвращаемый функцией;FunName – имя функции;type1, type2, …, typeN  типы переменных v1, v2, …, vN;val2, …, valN – значения, которое присваиваются по умолчанию переменным *v2*, …, *vN*. В этом случае типы и присваиваемые значения должны быть совместимыми.

##### Пример программы:



# Для установки значения по умолчанию параметру присваивается значение int m = 3. И если для второго параметра не будет передано значение, то он будет использовать значение по умолчанию.

При объявлении прототипа подобной функции он тоже может содержать значение по умолчанию для параметра. И в этом случае мы можем не определять в функции значение по умолчанию для параметра - оно будет браться из прототипа:



###### [Функции. Аргументы по умолчанию в функциях](https://www.bestprog.net/ru/2018/07/30/functions-arguments-by-default-in-functions_ru/)

###### [Параметры функции](https://metanit.com/cpp/tutorial/3.2.php)

Конспект №3. Variadic arguments. Эллипсис.

# Функции, использующие эллипсис, выглядят следующим образом:

#### тип\_возврата имя\_функции(список\_аргументов, ...)

# список\_аргументов — это один или несколько обычных параметров функции. Обратите внимание, функции, которые используют эллипсис, должны иметь по крайней мере один параметр, который не является эллипсисом.

# **Эллипсис** (англ. «ellipsis»), который представлен в виде многоточия … в языке C++, всегда должен быть последним параметром в функции. О нем можно думать, как о [**массиве**](https://ravesli.com/urok-74-massivy-chast-1/), который содержит любые другие параметры, кроме тех, которые указаны в список\_аргументов.

# Рассмотрим пример с использованием эллипсиса. Предположим, что нам нужно написать функцию, которая вычисляет среднее арифметическое переданных аргументов:

#include <iostream>

#include <cstdarg> // требуется для использования эллипсиса

// Эллипсис должен быть последним параметром.

// Переменная count - это количество переданных аргументов

double findAverage(int count, ...)

{

    double sum = 0;

    // Мы получаем доступ к эллипсису через va\_list, поэтому объявляем переменную этого типа

    va\_list list;

    // Инициализируем va\_list, используя va\_start. Первый параметр - это список, который нужно инициализировать.

    // Второй параметр - это последний параметр, который не является эллипсисом

    va\_start(list, count);

    // Перебираем каждый из аргументов эллипсиса

    for (int arg=0; arg < count; ++arg)

         // Используем va\_arg для получения параметров из эллипсиса.

         // Первый параметр - это va\_list, который мы используем.

         // Второй параметр - это ожидаемый тип параметров

         sum += va\_arg(list, int);

    // Выполняем очистку va\_list, когда уже сделали всё необходимое

    va\_end(list);

    return sum / count;

}

int main()

{

    std::cout << findAverage(4, 1, 2, 3, 4) << '\n';

    std::cout << findAverage(5, 1, 2, 3, 4, 5) << '\n';

}

# В теле функции, использующей переменные аргументы, доступ к значениям этих аргументов можно получить с помощью [библиотеки <cstdarg>](https://en.cppreference.com/w/cpp/utility/variadic):

|  |  |  |
| --- | --- | --- |
| Определено в заголовке [<cstdarg>](https://en.cppreference.com/w/cpp/header/cstdarg) | | |
| [**va\_start**](https://en.cppreference.com/w/cpp/utility/variadic/va_start) | позволяет получить доступ к переменным аргументам функции (макрос функции) |
| [**va\_arg**](https://en.cppreference.com/w/cpp/utility/variadic/va_arg) | обращается к следующему аргументу вариадной функции (макрос функции) |
| [**va\_copy**](https://en.cppreference.com/w/cpp/utility/variadic/va_copy)  (C++11) | создает копию переменных аргументов функции (макрос функции ) |
| [**va\_end**](https://en.cppreference.com/w/cpp/utility/variadic/va_end) | завершает обход аргументов вариадной функции (макрос функции) |
| [**va\_list**](https://en.cppreference.com/w/cpp/utility/variadic/va_list) | содержит информацию, необходимую va\_start, va\_arg, va\_end и va\_copy (typedef) |

###### [Variadic arguments](https://en.cppreference.com/w/cpp/language/variadic_arguments)

###### [Эллипсис](https://ravesli.com/urok-111-ellipsis-pochemu-ego-ne-sleduet-ispolzovat/)

Конспект №5. Обработка исключительных ситуаций.

# При работе программ возникают так называемые *исключительные ситуации*, когда дальнейшее нормальное выполнение приложения становится невозможным. Причиной исключительных ситуаций могут быть как ошибки в программе, так и неправильные действия пользователя, неверные данные и т.д.

# Для обработки таких ситуаций используются:

# - защищенные блоки (try-блоки) и перехваты исключений (catch-блоки); - инициализация исключений (инструкция throw).

# **try**: позволяет определить блок кода, который будет проверяться на наличие ошибок во время его выполнения;

# **throw**: нужен для создания и отображения исключений и используется для перечисления ошибок, которые генерирует функция, но не может самостоятельно обрабатывать исключения;

#### catch — блок кода, который выполняется при возникновении определенного Catch-блок имеет один из следующих форматов: catch (тип) {обработчик ошибочной ситуации} catch (тип идентификатор) {обработчик ошибочной ситуации} catch (…) {обработчик ошибочной ситуации}

# **std :: exception**

# Все исключения в языке C++ описываются типом exception, который определен в заголовочном файле <exception>. И при обработке исключений мы также можем использовать данный класс.

# С помощью метода what() в блоке catch можно получить переданное сообщение и, к примеру, вывести его на консоль.

несколько производных типов исключений, которые могут использоваться при различных ситуациях. Основные из них:

* **runtime\_error**: общий тип исключений, которые возникают во время выполнения
* **range\_error**: исключение, которое возникает, когда полученный результат превосходит допустимый диапазон
* **overflow\_error**: исключение, которое возникает, если полученный результат превышает допустимый диапазон
* **underflow\_error**: исключение, которое возникает, если полученный в вычислениях результат имеет недопустимые отрицательное значение (выход за нижнюю допустимую границу значений)
* **logic\_error**: исключение, которое возникает при наличии логических ошбок к коде программы
* **domain\_error**: исключение, которое возникает, если для некоторого значения, передаваемого в функцию, не определено результата
* **invalid\_argument**: исключение, которое возникает при передаче в функцию некорректного аргумента
* **length\_error**: исключение, которое возникает при попытке создать объект большего размера, чем допустим для данного типа
* **out\_of\_range**: исключение, которое возникает при попытке доступа к элементам вне допустимого диапазона

# Пример программы:



Спецификация noexcept.

# Он указывает, является ли набор возможных исключений, которые могут экранировать функцию, пуст. Эта спецификация исключений была разработана для предоставления сводной информации о том, какие исключения могут быть выброшены из функции, но на практике было установлено, что это проблематично. Одна спецификация динамического исключения, которая оказалась несколько полезной, является безусловной throw() спецификацией. Например, объявление функции:

# void MyFunction(int i) throw();

# сообщает компилятору, что функция не создает исключений. Однако в [/std:c++14](https://learn.microsoft.com/ru-ru/cpp/build/reference/std-specify-language-standard-version?view=msvc-170) режиме это может привести к неопределенному поведению, если функция создает исключение. Поэтому рекомендуется использовать [noexcept](https://learn.microsoft.com/ru-ru/cpp/cpp/noexcept-cpp?view=msvc-170) оператор вместо приведенного выше:

# void MyFunction(int i) noexcept;

Пример: перехват системного исключения "деление на ноль"  
int x = 0;  
try {  
std::cout <<2/x; //Здесь произойдет выброс исключения  
}  
catch (...) {  
std::cout << "Division by zero" << std::endl;

<https://metanit.com/cpp/tutorial/6.1.php> обработка исключений

<https://en.cppreference.com/w/cpp/error/exception>

<https://learn.microsoft.com/ru-ru/cpp/cpp/try-throw-and-catch-statements-cpp?source=recommendations&view=msvc-170>

https://habr.com/ru/sandbox/28877/

Конспект №6. Стиль кода (правила оформления).

Общие рекомендации  
1. Допускаются любые нарушения рекомендаций, если это улучшает читаемость.   
2. Правила могут быть нарушены, если против них есть персональные возражения.

Соглашения об именовании

# Имена, представляющие типы, должны быть обязательно написаны в смешанном регистре, начиная с верхнего.

Line, SavingsAccount

Имена переменных должны быть записаны в смешанном регистре, начиная с нижнего.

line, savingsAccount

Общая практика в сообществе разработчиков C++. Позволяет легко отличать переменные от типов, предотвращает потенциальные коллизии имён, например: Line line;  
  
Именованные константы (включая значения перечислений) должны быть записаны в верхнем регистре с нижним подчёркиванием в качестве разделителя.

MAX\_ITERATIONS, COLOR\_RED, PI

# Общая практика в сообществе разработчиков C++. Использование таких констант должно быть сведено к минимуму. В большинстве случаев реализация значения в виде метода — лучшее решение:

int getMaxIterations() // НЕЛЬЗЯ: MAX\_ITERATIONS = 25

{

return 25;

}

Названия методов и функций должны быть глаголами, быть записанными в смешанном регистре и начинаться с нижнего.

getName(), computeTotalWidth()

Названия пространств имён следует записывать в нижнем регистре.

model::analyzer, io::iomanager, common::math::geometry

Следует называть имена типов в шаблонах одной заглавной буквой.

template<class T> ...

template<class C, class D> ...

Аббревиатуры и сокращения в именах должны записываться в нижнем регистре.

exportHtmlSource(); // НЕЛЬЗЯ: exportHTMLSource();

openDvdPlayer(); // НЕЛЬЗЯ: openDVDPlayer();

Настраиваемым переменным следует давать то же имя, что и у их типа.

void setTopic(Topic\* topic) // НЕЛЬЗЯ: void setTopic(Topic\* value)

// НЕЛЬЗЯ: void setTopic(Topic\* aTopic)

// НЕЛЬЗЯ: void setTopic(Topic\* t)

void connect(Database\* database) // НЕЛЬЗЯ: void connect(Database\* db)

// НЕЛЬЗЯ: void connect (Database\* oracleDB)

Все имена следует записывать по-английски.

fileName; // НЕ РЕКОМЕНДУЕТСЯ: imyaFayla

Переменные, имеющие большую область видимости, следует называть длинными именами, имеющие небольшую область видимости — короткими.  
Имена временных переменных, использующихся для хранения временных значений или индексов, лучше всего делать короткими. Программист, читающий такие переменные, должен иметь возможность предположить, что их значения не используются за пределами нескольких строк кода. Обычно это переменные i, j, k, l, m, n (для целых), а также c и d (для символов).  
  
Симметричные имена должны использоваться для соответствующих операций.

get/set, add/remove, create/destroy, start/stop, insert/delete,

increment/decrement, old/new, begin/end, first/last, up/down, min/max,

next/previous, old/new, open/close, show/hide, suspend/resume, и т. д.

Уменьшайте сложность за счёт симметрии.  
  
Следует избегать сокращений в именах.

computeAverage(); // НЕЛЬЗЯ: compAvg();

Следует избегать дополнительного именования указателей.

Line\* line; // НЕ РЕКОМЕНДУЕТСЯ: Line\* pLine;

// НЕ РЕКОМЕНДУЕТСЯ: Line\* linePtr;

Функциям (методам, возвращающим какие-либо значения) следует давать имена в зависимости от того, что они возвращают, а процедурам — в зависимости от того, что они выполняют (методы void).  
  
Заголовочным файлам C++ следует давать расширение .h (предпочтительно) либо .hpp. Файлы исходных кодов могут иметь расширения .c++ (рекомендуется), .C, .cc либо .cpp.

# Заголовочные файлы объявляют интерфейс, файлы исходного кода его реализовывают. Если программисту необходимо найти реализацию, он должен быть уверен, что найдёт её именно в файле исходного кода.

Типы

Локальные типы, используемые в одном файле, должны быть объявлены только в нём..  
  
Переменные

# - Переменные никогда не должны иметь двойной смысл. - Следует избегать использования глобальных переменных. - Символ указателя или ссылки в языке C++ следует ставить сразу после имени типа, а не с именем переменной.

float\* x; // НЕ РЕКОМЕНДУЕТСЯ: float \*x;

int& y; // НЕ РЕКОМЕНДУЕТСЯ: int &y;

# - Переменные следует объявлять в как можно меньшей области видимости. Это упрощает контроль над действием переменной и сторонними эффектами.

Циклы  
- Нельзя включать в конструкцию for() выражения, не относящиеся к управлению циклом.

sum = 0; // НЕЛЬЗЯ: for (i = 0, sum = 0; i < 100; i++)

for (i = 0; i < 100; i++) sum += value[i];

sum += value[i];

# - Переменные, относящиеся к циклу, следует инициализировать непосредственно перед ним.

isDone = false; // НЕ РЕКОМЕНДУЕТСЯ: bool isDone = false;

while (!isDone) { // :

: // while (!isDone) {

} // :

// }

- Можно избегать циклов do-while.  
Такие циклы хуже читаемы, поскольку условие описано после тела. Читающему придётся просмотреть весь цикл, чтобы понять его работу.  
- Следует избегать использования break и continue в циклах.  
Такие выражения следует использовать только тогда, когда они повышают читаемость.   
- Для бесконечных циклов следует использовать форму while (true) .

while (true) {

:

}

for (;;) { // НЕТ!

:

}

while (1) { // НЕТ!

:

}

Условные выражения  
- Строго избегайте сложных уловных выражений. Вместо этого вводите булевы переменные.

bool isFinished = (elementNo < 0) || (elementNo > maxElement);

bool isRepeatedEntry = elementNo == lastElement;

if (isFinished || isRepeatedEntry) {

:

}

// NOT:

if ((elementNo < 0) || (elementNo > maxElement)||

elementNo == lastElement) {

:

}

# - Ожидаемую часть следует располагать в части if, исключение — в части else. - Условие следует размещать в отдельной строке.

if (isDone) // НЕ РЕКОМЕНДУЕТСЯ: if (isDone) doCleanup();

doCleanup();

- Следует избегать «магических» чисел в коде. Числа, отличные от 0 или 1, следует объявлять как именованные константы.  
- Константы с плавающей точкой следует записывать с десятичной точкой и с указанием по крайней мере одной цифры после запятой.

double total = 0.0; // НЕ РЕКОМЕНДУЕТСЯ: double total = 0;

double speed = 3.0e8; // НЕ РЕКОМЕНДУЕТСЯ: double speed = 3e8;

double sum;

:

sum = (a + b) \* 10.0;

 Оформление

# - Основной отступ следует делать в два пробела (2-4).

for (i = 0; i < nElements; i++)

a[i] = 0;

# - Блоки кода следует оформлять так, как показано в примере 1 (рекомендуется) или в примере 2, но ни в коем случае не так, как показано в примере 3. Оформление функций и классов должно следовать примеру 2.

while (!done) {

doSomething();

done = moreToDo();

}

while (!done)

{

doSomething();

done = moreToDo();

}

while (!done)

{

doSomething();

done = moreToDo();

}

Пример 3 использует лишние отступы, что мешает ясному отображению логической структуры кода.  
- Определения методов следует оформлять следующим образом:

void someMethod()

{

...

}

# - Конструкцию if-else следует оформлять следующим образом:

if (condition) {

statements;

}

if (condition) {

statements;

}

else {

statements;

}

if (condition) {

statements;

}

else if (condition) {

statements;

}

else {

statements;

}

# Следствие из правила, указанного выше. Причём написание *else* на той же строке, где стоит закрывающая фигурная скобка первого блока, не является запрещённым:

if (condition) {

statements;

} else {

statements;

}

# - Цикл for следует оформлять следующим образом:

for (initialization; condition; update) {

statements;

}

# - Цикл while следует оформлять следующим образом:

while (condition) {

statements;

}

# - Конструкцию switch следует оформлять следующим образом:

switch (condition) {

case ABC :

statements;

// Отсутствует "break"

case DEF :

statements;

break;

case XYZ :

statements;

break;

default :

statements;

break;

}

# - Конструкцию try-catch следует оформлять следующим образом:

try {

statements;

}

catch (Exception& exception) {

statements;

}

# - Если конструкция if-else содержит только одно выражение в теле, фигурные скобки можно опускать.

if (condition)

statement;

while (condition)

statement;

for (initialization; condition; update)

statement;

Пробелы

# — Операторы следует отбивать пробелами. — После зарезервированных ключевых слов языка C++ следует ставить пробел. — После запятых следует ставить пробелы. — Двоеточия следует отбивать пробелами. — После точек с запятой в цикле *for* следует ставить пробелы.

a = (b + c) \* d; // НЕ РЕКОМЕНДУЕТСЯ: a=(b+c)\*d

while (true) // НЕ РЕКОМЕНДУЕТСЯ: while(true)

{

...

doSomething(a, b, c, d); // НЕ РЕКОМЕНДУЕТСЯ: doSomething(a,b,c,d);

case 100 : // НЕ РЕКОМЕНДУЕТСЯ: case 100:

for (i = 0; i < 10; i++) { // НЕ РЕКОМЕНДУЕТСЯ: for(i=0;i<10;i++){

...

# - Используйте выравнивание везде, где это улучшает читаемость.

Комментарии

- Следует делать как можно меньше комментариев, делая код самодокументируемым путём выбора правильных имён и создания ясной логической структуры.  
- Все комментарии следует писать на английском.  
- Используйте // для всех комментариев, включая многострочные.   
После // следует ставить пробел, а сам комментарий следует начинать писать с большой буквы завершать точкой.  
- Комментарии следует располагать так, чтобы они относились к тому, что они описывают.

while (true) { // НЕ РЕКОМЕНДУЕТСЯ: while (true) {

// Do something // Do something

something(); something();

} }

Это делается с тем, чтобы избежать ситуацию, когда комментарии нарушают логическую структуру программы.  
  
[Рекомендации по написанию кода в C++](https://habr.com/ru/post/172091/)

Конспект №7. Синонимы типов с помощью using и typedef.

# typedef (сокращенно от «type definition», «определение типа») – это ключевое слово, которое создает псевдоним для существующего типа данных. Например:

# typedef double distance\_t; // определяем distance\_t как псевдоним для типа double

# По соглашению имена typedef объявляются с использованием суффикса "\_t". Это помогает указать, что идентификатор представляет собой тип, а не переменную или функцию, а также помогает предотвратить конфликты имен с другими типами идентификаторов.

# После определения имя typedef можно использовать везде, где требуется тип. Например, мы можем создать переменную с именем typedef в качестве типа:

# distance\_t milesToDestination{ 3.4 }; // определяет переменную типа double

# Обратите внимание, что typedef не определяет новый тип. Скорее, он просто создает новый идентификатор (псевдоним) для существующего типа. typedef можно использовать как замену везде, где можно использовать обычный тип.

Вместо typedef можно использовать следующий псевдоним типа:

# using distance\_t = double; // определяем distance\_t как псевдоним для типа double

# Псевдонимы типов using функционально эквивалентны определениям typedef, но имеют преимущество в более удобном синтаксисе определения.

# typedef int (\*fcn\_t)(double, char); // fcn\_t трудно найти

# using fcn\_t = int(\*)(double, char); // fcn\_t легче найти

# Если имя какого-то пространства имен кажется нам слишком длинным, можно ввести для него короткий синоним:

# namespace новое\_имя = старое\_имя;

###### [Псевдонимы типов](https://radioprog.ru/post/1131)

Auto, анонимные функции, защита от повторно включения (модули), #progmaonce (MSVC)

Дополните отчёт о компиляции: как использовать компилятор MSVC?

Изучите дорожную карту изучения C++: <https://salmer.github.io/CppDeveloperRoadmap/Russian/>