**Методические рекомендации**

**по разработке с использованием IDE**

**Microsoft Visual Studio 2019 Community**

Оглавление

1. [Введение 3](#_Toc25616624)
2. [Установка Visual Studio 4](#_Toc25616625)
3. [Пользовательский интерфейс Visual Studio 5](#_Toc25616626)
4. [Теоретические основы 10](#_Toc25616627)
5. [Стандарт программирования 20](#_Toc25616628)
6. [Важные замечания об ознакомительных работах 21](#_Toc25616629)
7. [Ознакомительная работа 1.1 – Быстрый старт разработки на C++ 23](#_Toc25616630)
8. [Ознакомительная работа 1.2 – Внешняя система сборки CMake 26](#_Toc25616631)
9. [Ознакомительная работа 1.3 – Автоматизированное тестирование с Google Test 31](#_Toc25616632)
10. [Ознакомительная работа 1.4 – Контроль версий с помощью git 36](#_Toc25616633)
11. [Администрирование 1.1 - Настройка переменной окружения PATH 50](#_Toc25616634)
12. [Администрирование 1.2 – Генерация сценария нижележащей системы сборки с помощью CMake и ручная сборка по сценарию нижележащей системы сборки 51](#_Toc25616635)
13. [Администрирование 1.3 – Переименование вложенного проекта Visual Studio 53](#_Toc25616636)
14. [Администрирование 1.4 – Работа с несколькими проектами Visual Studio 54](#_Toc25616637)
15. [Администрирование 1.5 – Использование собранных ранее исполняемых модулей в решениях Visual Studio 56](#_Toc25616638)
16. [Администрирование 1.6 – Использование MSBuild вместо Ninja в качестве нижележащей системы сборки при работе с проектами CMake в Visual Studio 57](#_Toc25616639)
17. [Интеграция 1.1 - SFML (библиотека мультимедиа) – использование в проекте Visual C++ 59](#_Toc25616640)
18. [Интеграция 1.2 - nlohmann\_json – библиотека формата JSON для современного C++ - использование в проекте CMake 63](#_Toc25616641)
19. [Заключение 66](#_Toc25616642)

# Введение

Интегрированная среда разработки (**Integrated Development Environment, далее «среда разработки» или «IDE»**) – это программа или набор программ, используемых для разработки программного обеспечения.

Среда разработки взаимодействует с пользователем и настраиваемым набором внешних программ. Как правило, пользователь не имеет доступа к конкретным внешним программам, IDE вызывает их, передает в них пользовательский ввод и выводит результат их работы в свой пользовательский интерфейс самостоятельно (IDE является “**front end**” – интерфейсом, доступным пользователю для взаимодействия, внешние программы являются “**back end**” – компонентом, непосредственно обрабатывающим данные). При этом текстовый редактор обычно является интегрированным в пользовательский интерфейс и неотъемлемым компонентом среды разработки.

В данных рекомендациях рассматривается разработка на C++ с использованием Microsoft Visual Studio 2019 Community (**далее – Visual Studio**) в ОС Windows. В том числе приводятся необходимые составляющие **toolchain** (набора внешних программ, необходимых для разработки, также **toolset**) и рассматривается разработка с использованием toolchainпо умолчанию в Visual Studio, а также специального (custom) toolchain, включающего в себя дополнительную внешнюю систему сборки. Рассматривается разработка консольного приложения, статической библиотеки, динамической библиотеки, оконного приложения для Windows.

Рассматриваются встроенные средства сборки Visual Studio (интегрированная система сборки **MSBuild**), а также сторонняя система сборки **CMake**.

В приложении к рекомендациям приведены типовые сценарии системы сборки CMake для перечисленных модулей, а также типовая стратегия контроля версий с использованием системы контроля версий (**Version Control System, VCS**) **Git**.

# Установка Visual Studio

Загрузите Visual Studio Installer с [официального сайта Microsoft](https://www.microsoft.com), выбрав в числе продуктов Visual Studio 2019 Community.   
Visual Studio Installer позволяет объединять составляющие среды разработки в т. н. «рабочие нагрузки» (**workload**), например, рабочая нагрузка «Разработка классических приложений на C++» содержит средства, необходимые для разработки консольных программ без использования сторонних средств, например, C++ / CLI. Убедитесь, что будут установлены workload, соответствующие разработке на C++ и C# в консоли.

Также возможен ручной выбор набора для установки. Вне зависимости от выбора рабочей нагрузки или отдельных компонентов, убедитесь, что установлены:

* Средства CMake C++ для Windows;
* Расширение GitHub для Visual Studio;
* Git для Windows;
* Адаптер тестов для Google Test;
* Пакет SDK для универсальной CRT для Windows.

# Пользовательский интерфейс Visual Studio

**Начальный экран Visual Studio. Организация проектов и решений**

После установки запустите Visual Studio и ознакомьтесь с Начальным экраном (рисунок 1).

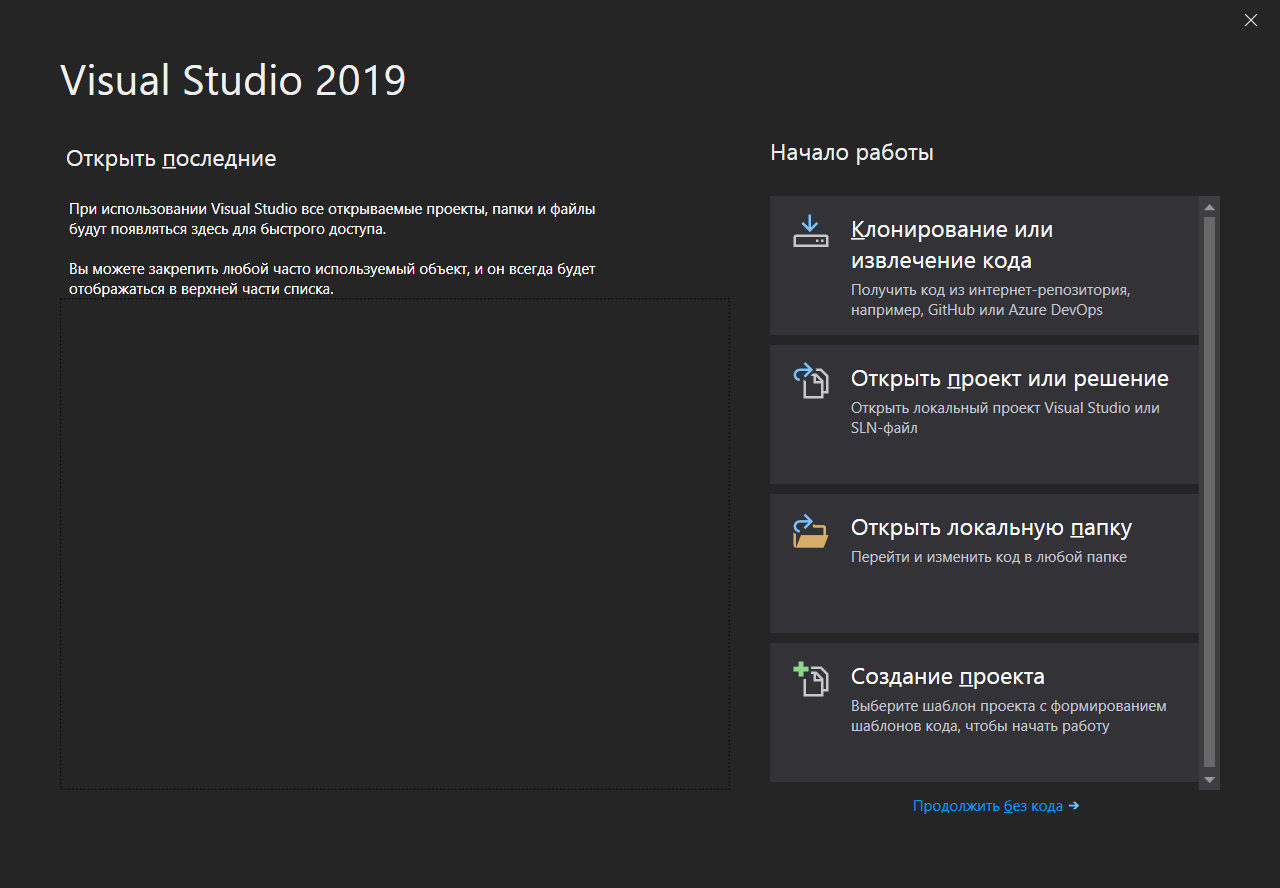


Рисунок 1 - Начальный экран

Проектом (**project**) в Visual Studio называется набор файлов, участвующих в сборке, представляющих одно приложение, одну библиотеку или один набор ресурсов, иначе говоря, один результат сборки.

Решением (**solution**) в Visual Studio называется один или несколько проектов, их общие настройки интерфейса среды разработки, их общие сценарии сборки, зависимости между ними и файлы, не связанные с конкретным проектом.

Одно решение может использоваться для представления одного программного продукта целиком, один проект – для представления конкретного исполняемого модуля.

В рамках лабораторных работ возможно сопоставление каждой дисциплине своего решения: так, Объектно-ориентированное программирование на C++ потребует стандартных средств разработки на C++ для всех проектов, а работы по Технологиям пространственного моделирования могут иметь общий toolchain для разработки на C# + Windows.Forms с использованием библиотеки OpenTK.

Работа в Visual Studio возможна после создания решения (а в нем – минимум одного проекта), клонирования существующего решения из локального (на диске) или удаленного (в сети) репозитория, открытия существующего решения или открытия произвольной папки на диске (как правило, содержащей сценарий поддерживаемой системы сборки, например CMake). Для репозиториев под управлением системы контроля версий Git возможна работа с версиями из интерфейса Visual Studio и соответствующего расширения от GitHub (GitHub для Visual Studio).

В дальнейшем будет рассмотрен процесс разработки по перечисленным сценариям.

**Настраиваемые окна и панели инструментов**

На начальном экране Visual Studio выберите «Продолжить без кода».

Откроется главный экран среды разработки без активного проекта, где можно настроить предпочитаемый внешний вид, а затем перейти к управлению проектами (например, созданию, открытию или клонированию), как если бы соответствующий сценарий был выбран непосредственно на начальном экране.

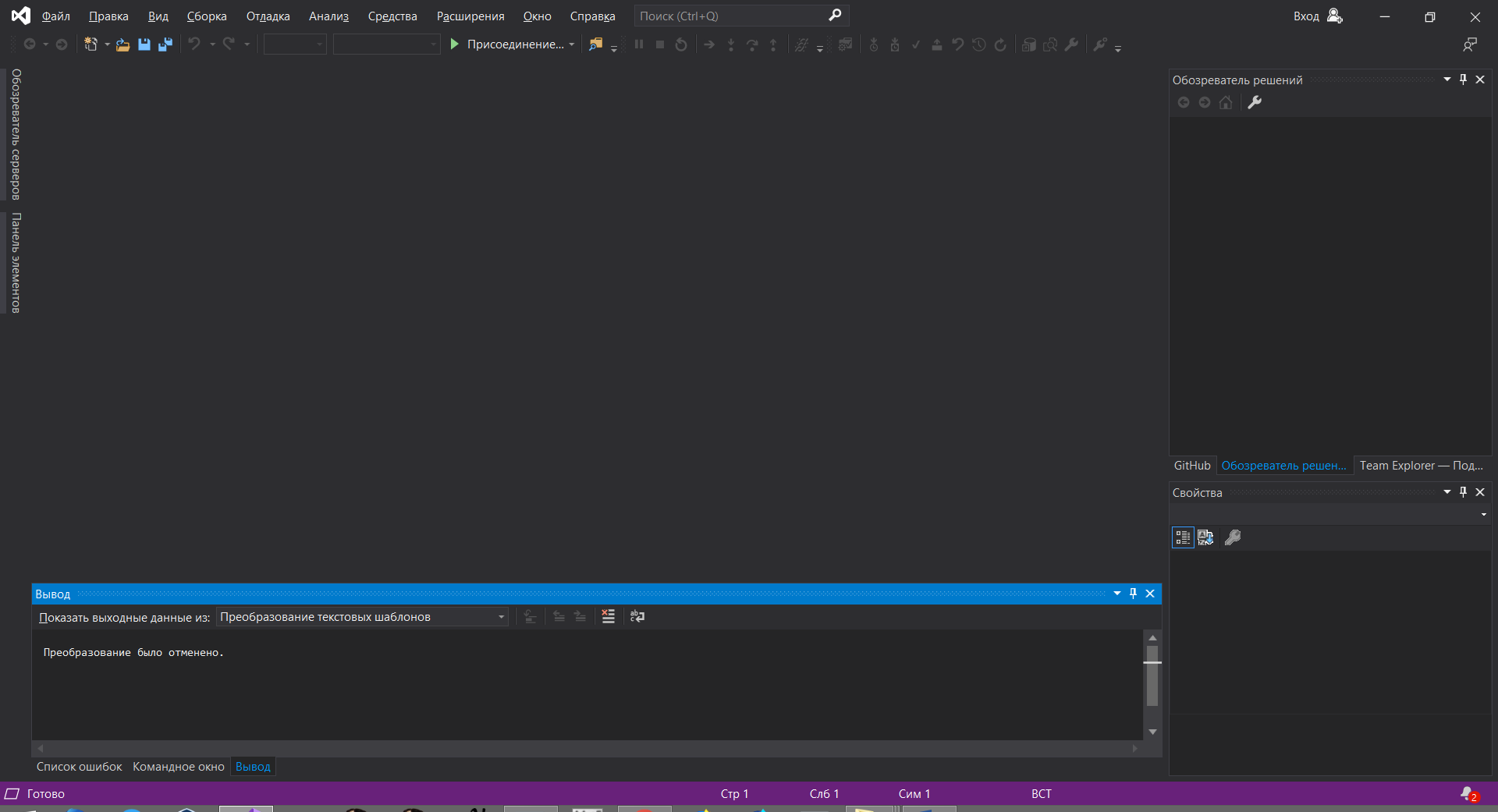


Рисунок 2 - Главный экран

**Настраиваемые окна**

Измените по своему усмотрению взаимное расположение настраиваемых окон (в примере на Рисунке 2 это Вывод, Обозреватель решений и Свойства).

Для этого перетащите окно, кликнув по его заголовку, в произвольную область для его открепления от главного экрана или на одну из позиций всплывающей подсказки, позволяющей разместить окно по границе экрана.

После закрепления окна в пределах главного экрана, возможно размещение других окон относительно него (например, окно Свойства выровнено по нижней границе окна Обозревателя решений).

Окна Список ошибок, Командное окно и Вывод на Рисунке 2 размещены в одной области главного экрана. Для операций с одним из них перетаскивайте его не за заголовок, содержащий название активного окна (на Рисунке 2 это Вывод), а за вкладку в нижней части с названием перетаскиваемого окна.

Возможно добавление в эту группу еще нескольких вкладок, для этого перетащите добавляемое окно на её область экрана так, чтобы подсветить эту область целиком.

Наконец, клик по пиктограммам канцелярской кнопки и “X” в правом верхнем углу окна позволяет настроить автоматическое скрытие окна (окно открывается и закрывается по клику на заголовок, который может изменить своё расположение) или закрыть его.

Закрытые окна можно восстановить из элемента меню Вид главного экрана и разместить заново.

Для управления файлами проекта (в том числе добавления файлов в проект и создания новых файлов) необходимо окно Обозреватель решений.

Окна Вывод и Список ошибок используются компилятором для вывода хода компиляции, возможных ошибок и предупреждений.

В обзоре средств контроля версий и отладки будут рассмотрены и другие.

Добавьте перечисленные окна на главный экран и перейдите к подзаголовку Панели инструментов.

**Панели инструментов**

**Возможно выборочное включение и отключение панелей инструментов – групп иконок, расположенных под строкой меню главного экрана.**

**Список панелей инструментов с переключаемым отображением (вкл – выкл) находится в Главный экран –> Вид -> Панели инструментов.**

**После включения панели она появляется под строкой меню. Для перетаскивания доступна левая сторона панели. При наведении курсора он приобретает форму четырехгранной звезды с лучами-стрелками.**

**Раскрывающийся список на правой стороне каждой панели содержит пункт Добавить или удалить кнопки для тонкой настройки.**

**Убедитесь, что на главном экране присутствуют панели инструментов Сборка, Отладка, Управление версиями и Стандартная.**

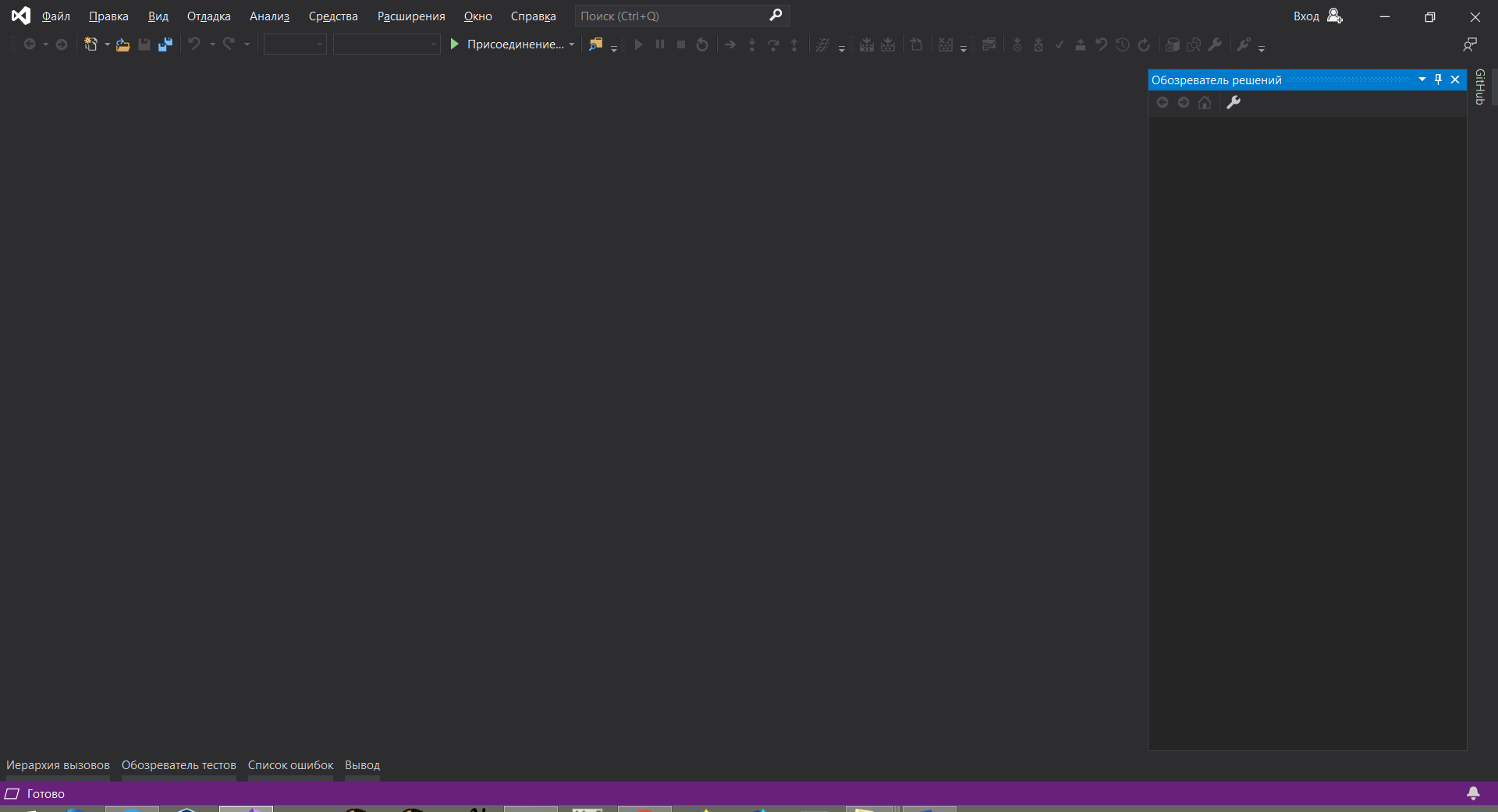


Рисунок 3 - Пример настройки окон и панелей инструментов на главном экране

В примере на Рисунке 3 окна Иерархия вызовов, Обозреватель тестов, Список ошибок и Вывод имеют объединенную рабочую область, закреплены и выровнены по нижней границе главного экрана, автоматическое скрытие включено.

Окно Обозреватель решений выровнено по правой границе главного экрана, не скрыто и сейчас активно.

Окно Github также выровнено по правой границе, но скрыто и будет доступно по клику по заголовку.

Отображены панели инструментов Стандартная, Сборка, Система управления версиями и Отладка. Для панели Отладка добавлен элемент Начать отладку.

Настройка пользовательского интерфейса завершена.

# Теоретические основы

**\*\*\***

Сборкой (**build**) называется получение программы или программного модуля из входных файлов, в том числе файлов исходного кода (**source code files**, мн. ч. **sources**), предварительно скомпилированных файлов, мультимедиа ресурсов (звуков, изображений, видео).

Сборка осуществляется системой сборки (**build system**), а её ход описывает сгенерированный автоматически или написанный вручную сценарий сборки (**build script**).

**О трансляции, компиляции, интерпретации**

Трансляцией программы называется преобразование программы на одном языке программирования в программу на другом. В общем случае, трансляцией называется любое преобразование текста на одном языке в текст на другом.

Подмножество трансляторов, выполняющее преобразование в код, исполняемый непосредственно машиной, называется компиляторами (**compiler**), а выполняемая ими трансляция – компиляцией.

Подмножество трансляторов, выполняющее построчный анализ и немедленное выполнение кода, называется интерпретаторами (**interpreter**), а выполняемая ими трансляция – интерпретацией. Исходный код для интерпретации называется также сценарием (**script**).

**О компоновке и связывании**

Опишем разновидности процесса сопоставления сущности и её имени и реализации такового, в большинстве своем носящие в русскоязычной литературе название «связывание» или «компоновка», а также вспомогательные по отношению к ним понятия:

* **Name binding** – сопоставление имени (применительно к языку программирования C++ - идентификатора[[1]](#footnote-1)) с данными или кодом, в том числе:
  + **Static binding** (**early binding**, также «раннее связывание») – когда такое сопоставление выполнено до начала работы программы, то есть во время компоновки или компиляции;
  + **Dynamic binding (late binding**, также «позднее связывание») – когда такое сопоставление выполняется в ходе работы программы.
* **Symbol resolution** – сопоставление объявления (**declaration**) и определения (**definition**) сущности, то есть тех же данных и кода, в ходе сборки программы (в том числе в ходе трансляции или компоновки);
* **Runtime symbol resolution** – то же сопоставление, но в ходе работы программы;
* **Linking** («компоновка») – приведение промежуточного представления или образа программы к состоянию, в котором возможно её использование, в том числе:
  + **Static linking** (также просто **linking**) – процесс такового приведения, включающий в себя symbol resolution и выполняемый в ходе трансляции исходного кода, объектами данного процесса являются файлы исходного кода[[2]](#footnote-2), а результатом – исполняемый модуль;
  + **Dynamic linking** («динамическая компоновка») – процесс такового приведения, включающий в себя runtime symbol resolution и являющийся составной частью загрузки (**loading**) программы операционной системой, объектами данного процесса являются исполняемые модули, а результатом – исполняемые модули, все перекрестные ссылки в которых успешно сопоставлены данным и коду из другого модуля.
* **Linkage** – свойство сущностей языка C++, определяющее возможность symbol resolution с их участием, а также их использования, в том числе:
  + **No linkage** – возможно выполнить symbol resolution в пределах области видимости (**scope**), возможно использовать сущность в рамках этой же области, доступ к ней вне этой области невозможен, и её существование вне этой области не гарантируется;
  + **Internal linkage** – возможно выполнить symbol resolution в пределах некоей единицы трансляции (**translation unit**, см. далее), возможно использовать сущность в рамках этой же единицы трансляции; runtime symbol resolution не допускается;
  + **External linkage**[[3]](#footnote-3)– возможно выполнить symbol resolution и использовать сущность (обращаться к ней по имени, для данных – изменять значение и т.д.) за пределами одной единицы трансляции. Более того, возможен runtime symbol resolution определения этой сущности в рамках одного исполняемого модуля и её объявления в другом.

Здесь и далее к описанным процессам будут применяться только оригинальные названия, исключением является процесс static linking, для которого будет употребляться термин «компоновка».

Термин linkage подробно не рассматривается и приведен ввиду сходства с linking (и возможных неверных предположений о необходимости linkage там, где речь идет о linking).

**Ход трансляции исходного кода на C++**

Стандарт языка C++ [[4]](#footnote-4) определяет в том числе набор простых инструкций, или т. н. директив (определить константу, выполнить замену, сравнить константы, выполнить набор директив в зависимости от результата сравнения).

Этот набор называется языком препроцессора, выполнение инструкций – препроцессингом (**preprocessing** – предварительная обработка), а программа или программный модуль, выполняющий эту обработку – препроцессором (**preprocessor**).

В ходе трансляции[[5]](#footnote-5) исходного кода на C++ препроцессинг, удаляющий из исходного кода директивы препроцессора и опционально заменяющий их другими данными (например, результатом подстановки содержимого файла), осуществляется раньше трансляции остального кода, не являющегося директивами препроцессора.

Промежуточное представление[[6]](#footnote-6), полученное в результате трансляции остального кода, может вызывать функцию, которая была в рамках хода трансляции объявлена (**declared**), но не была определена (**defined**), или конкретизировать (**instantiate**) шаблон класса, который был предварительно объявлен (**forward-declared**), но не был определен, или вызывать методы класса, который был предварительно объявлен, но не определён, или каким-либо еще образом требовать сопоставления существующего и используемого идентификатора с существующим определением функции (**function definition**), определением шаблона класса (**template class definition**) и так далее.

Такое сопоставление, которое, как уже упоминалось, носит название разрешения символов (**symbol resolution**), завершается успешно, если больше нет неразрешенных символов, примеры которых приведены выше, и, наряду с трансляцией промежуточного представления кода в итоговый машинный код и снабжением этого кода некоторой дополнительной информацией, зависящей от целевой платформы[[7]](#footnote-7), завершает ход трансляции исходного кода в исполняемый модуль – образ программы, готовый к загрузке операционной системой для исполнения.

**Реализация трансляции исходного кода на C++ в ходе сборки**

Рассматривается ситуация, когда сборка выполняется несколькими программами, каждая из которых принимает данные текущего этапа трансляции, обрабатывает их и передает в следующую программу; такая схема получила название конвейера. Кроме очевидной программы или интерпретируемого сценария запуска всех программ в конвейере (обычно он является частью системы сборки), препроцессор, описанный ранее, является одной из этих программ.

Далее, результат работы препроцессора – исходный код, лишенный директив препроцессора – передается в компилятор, реализованный в виде отдельной программы. Результатом работы компилятора является упомянутое выше промежуточное представление, обычно в виде файла, называемое объектным файлом (**object file**). Объектный файл содержит в себе скомпилированный машинный код для целевой платформы, а также дополнительную информацию, о которой далее.

В ходе лексического анализа исходного кода компилятору требуется сопоставить идентификатор переменной, класса, функции, шаблона класса и т.д. уже определённой переменной и т.д., имеющей в том числе определенную область видимости (**scope**), время жизни (**storage duration**), **linkage** – тип связывания. Как правило, компилятор реализует это сопоставление, используя таблицу символов (**symbol table**), хранящую идентификаторы и соответствующие им параметры, а по завершении работы включает в объектный файл разрешенные (**resolved**) и неразрешенные (**unresolved**) символы (последние имеют по понятным причинам external linkage), образующие таблицу экспорта (**export table**) и таблицу импорта (**import table**) соответственно.[[8]](#footnote-8)

При этом они необязательно являются единственными включенными в объектный файл символами – в целях отладки к ним могут быть присоединены символы[[9]](#footnote-9), не входящие в эти две категории, т. н. отладочные символы, или **debugging symbols** (н-р переменные из заведомо недоступной пользователю библиотеки области видимости[[10]](#footnote-10)).

Целевая платформа (**target platform**), на которой будет выполняться скомпилированный модуль, и платформа (**host platform**), на которой выполняется сам компилятор, могут различаться. В таком случае компиляция называется кросс-компиляцией (**cross compiling**).

Кросс-компиляция позволяет решить «проблему курицы и яйца» для платформы: чтобы скомпилировать что-то на новой машине с новой платформой, нужно запустить на ней компилятор, код которого (предположим, что он также написан на C++) уже собран для данной платформы. Проблема решается запуском на старой платформе уже собранного для старой платформы компилятора, передачей ему собственного исходного кода и указанием новой платформы в качестве целевой. В данных рекомендациях разработка с использованием кросс-компиляции не рассматривается.

Как правило, за один вызов конвейера из двух программ – препроцессора и компилятора – транслируется в объектный файл только часть исходного кода (называемая единицей трансляции, **translation unit**), предназначенного для сборки.[[11]](#footnote-11)

Распространена организация единиц трансляции, при которой конвейер получает на вход несколько файлов исходного кода, а результатом их работы является такое же количество объектных файлов.

Объектные файлы содержат упоминаемые выше неразрешенные ссылки, и разрешением ссылок обычно занимается отдельная программа - компоновщик (или редактор связей, **link editor**, **linker**, «линкер»). Это разрешение заключается в сопоставлении каждого использования функции, класса, шаблона класса во всех имеющихся объектных файлах их определению, находящемуся в конкретном объектном файле.

Компоновщик вызывается после компиляции всех файлов исходного кода в объектные файлы. Компоновщик принимает объектные файлы и в случае успешного разрешения всех зависимостей результатом его работы является исполняемый модуль – образ программы, готовой к выполнению.

Существует еще ряд задач, выполняемых в ходе преобразования объектных файлов в исполняемый модуль, и при этом не имеющих отношения к трансляции кода, в том числе: внедрение информации, зависящей от целевой платформы, о том, как следует загружать образ программы, описываемый исполняемым модулем, в память; какой именно машинный код в модуле получит управление после успешной загрузки модуля[[12]](#footnote-12), компоновка с какими внешними модулями необходима во время загрузки, и т.д.

Как правило, ими тоже занимается компоновщик, хотя это и сложилось исторически на распространенных платформах и зависит (как и само разделение процесса трансляции между препроцессором, компилятором и компоновщиком) от реализации таковых и от реализации процесса загрузки программ операционной системой.

Резюмируя рассматриваемый ход сборки исходного кода на C++:

* Система сборки запускает для каждого файла исходного кода последовательно препроцессор и компилятор (они могут быть реализованы в одной программе) для препроцессинга и компиляции исходного кода в объектный файл, а затем для результирующего набора объектных файлов однократно запускает компоновщик.
* Сборка считается успешной, если препроцессор без ошибок обработал директивы на языке препроцессора[[13]](#footnote-13), компилятор без ошибок скомпилировал исходный код, а компоновщик выполнил компоновку, в результате которой не осталось неразрешенных ссылок, и также выполнил некоторые другие зависящие от целевой платформы действия.
* Результат успешной сборки – исполняемый модуль.

В данных рекомендациях рассматриваются системы сборки MSBuild и CMake, а также набор препроцессор-компилятор-компоновщик, входящий в состав продукта Microsoft Visual C++. В дальнейшем соответствующий компилятор называется Microsoft Visual C++ Compiler. Препроцессор и компилятор в MSVC++ реализованы в виде одного исполняемого модуля (файл cl.exe), а компоновщик – отдельно (файл link.exe). Просмотр символов реализует утилита dumpbin.exe.[[14]](#footnote-14)

Следует отметить, что по отношению к Visual Studio компилятор (Microsoft Visual C++ Compiler) и система сборки, пусть даже входящие в стандартную рабочую нагрузку среды, являются внешними компонентами, и совокупность таковых – toolchain (упоминалось во Введении) – может быть изменена.

В нашем случае меняться будет только внешняя система сборки – с MSBuild на CMake. При использовании CMake, последняя выполняет роль front end, генерируя на основе сценария CMake сценарий сборки нижележащей системы **(underlying build system)**, с которым работает Visual Studio – по умолчанию это система сборки Ninja, хотя ей может выступать и MSBuild, При самостоятельной сборке будет использоваться система MSBuild.

**Трансляция в промежуточный код**

Возможно сначала транслировать исходный код программы в некий промежуточный код (**intermediate code**, исторически называемый **p-code, bytecode**, или **байткод**), и работать далее с этим кодом, по описываемым далее причинам предпочитаемым машинному.

В таком случае, нужна система из двух трансляторов, один из которых транслирует исходный код в промежуточный, а второй транслирует промежуточный код в машинный.

**Реализация трансляции в промежуточный код и его интерпретации во время выполнения программы (at runtime)**

Для описанной системы из двух трансляторов введем следующие ограничения: первый срабатывает однократно в процессе также однократной сборки. Второй интерпретирует промежуточный код, то есть построчно анализирует его и выполняет.

Второй транслятор является виртуальной машиной (**virtual machine, VM**), которая эмулирует компьютер, исполняющий промежуточный код.

Первый же транслятор является для этой виртуальной машины компилятором, поскольку результат его работы можно без изменений передать в виртуальную машину для выполнения. [[15]](#footnote-15)

Продуктом сборки программы в таком случае является промежуточный код, то есть только первый транслятор задействован в процессе сборки.

Результат сборки может быть запущен на любой платформе, на которой реализован второй транслятор (виртуальная машина, исполняющая промежуточный код). В отличие от описанной сборки программы на C++, когда программа должна быть транслирована в машинный код конкретной платформы, программа, транслированная в промежуточный код, может распространяться «как есть» и выполняться везде, где реализован второй транслятор (виртуальная машина).

Примером такой виртуальной машины является **CLR (Common Language Runtime)** – среда, исполняющая промежуточный код на языке **CIL (Common Intermediate Language**, он же **Microsoft Intermediate Language, MSIL**).

Спецификация **Common Language Infrastructure (CLI)** определяет среду CLR, язык CIL, набор классов BCL (Base Class Library) и некоторые другие элементы инфраструктуры разработки для виртуальной машины, выполняющей промежуточный код.

Инфраструктуру CLI реализует **Microsoft .NET Framework**. Microsoft .NET Framework и Microsoft Visual C# являются частью workload (набора для установки) и, очевидно, toolchain (набора средств разработки) для языка C#.

Результатом работы Microsoft Visual C# или любого другого совместимого с CLI продукта для другого языка в режиме компиляции в промежуточный код является **assembly** – исполняемый модуль, почти аналогичный таковому для операционной системы без промежуточной среды, при этом содержащий **assembly manifest** - метаинформацию (т.е. обобщенную для всех подобных модулей, присутствующую в каждом из них информацию) о внешних ресурсах, экспортируемых символах, текущей версии, необходимых для работы разрешениях, других assembly для работы, данных локализации и т.д. Следует отметить, что возможно отношение «один assembly – много файлов, один из которых assembly manifest», в то время как для классических исполняемых модулей метаинформация обязательно встроена в тот же исполняемый файл, и притом один, а для исполняемых модулей, требующих dynamic linking, не существует объединенного формата assembly manifest, и они требуют внешнего объявления имеющихся в них символов, известного во время компиляции.

В главе «Реализация трансляции исходного кода на C++ в ходе сборки» упоминались внешние модули, dynamic linking с которыми осуществляется во время загрузки программы операционной системой. Возможна аналогичная dynamiс linking с assembly, в том числе представляющими установленные в систему модули, находящиеся в %WINDIR%/assembly.

Исторически существовала только одна, и притом не имеющая открытого исходного кода, позволяющего анализ и разработку аналога, совокупность реализации среды CLI и программных продуктов, включающих в себя компиляторы в промежуточный код CIL, существовавшая только для платформ Windows (в последующем также их современных, 64-битных преемников). Это Microsoft .NET Framework и наборы компиляторов Microsoft Visual C#, Microsoft Visual Basic .NET, Microsoft Visual C++ с расширением C++ \ CLI, и средства для некоторых других языков.

В настоящее время существуют реализации инфраструктуры CLI с открытым исходным кодом, например, любительский Mono или .NET Core, реализованный и поддерживаемый Microsoft. Возможна (в данных рекомендациях не рассматривается) их самостоятельная сборка для целевой платформы и запуск на этой платформе программ, скомпилированных для CLR.

Резюмируя рассматриваемый ход сборки исходного кода на C#:

* Система сборки запускает для набора файлов препроцессор и компилятор в промежуточный код, результатом работы которых является набор assembly, имеющих всю необходимую информацию для работы в среде CLR.

# Стандарт программирования

В ознакомительных работах используется следующий стандарт программирования (**coding standard**):

* Стиль Олмана (открывающая и закрывающая скобка имеют равную величину отступа) для определения структур, классов, пространств имён (namespaces), объединений (unions), перечислений (enumerations) классов перечислений (enumeration classes), шаблонов классов (class templates), блоков условий и циклов (while, for, do .. while, if, if .. else);
* Стиль Кернигана и Ричи (K&R) для инициализаторов, требующих переноса строки, для списков инициализации членов класса и вызовов целевых конструкторов (target constructors) внутри делегирующих конструкторов (delegating constructors), для вызовов функций (в том числе шаблонных функций и методов класса);
* Все пользовательские классы находятся в пространстве имен mai или вложенных классах или пространствах имен;
* Директивы о включении заголовочных файлов пользователя располагаются перед таковыми о включении внешних библиотек, а таковые о включении внешних библиотек – перед директивами о включении стандартных заголовочных файлов. Предкомпилируемые заголовки по умолчанию не используются. Между этими тремя категориями директив интервал составляет 1 строку, между директивами и остальным кодом – 1 или 2 строки;
* Автоматические переменные именуются в camelCase, члены класса, шаблона класса с модификатором доступа public, сами классы, пространства имен и сущности в них именуются в PascalCase, макроконстанты в UPPER\_SNAKE\_CASE, константы, не являющиеся макроконстантами, имеют префикс k, переменные с глобальной областью видимости имеют префикс g, а члены класса или шаблона класса с модификатором доступа, отличным от public, имеют префикс m\_.
* В списке базовых классов, требующем переноса строки, underlying type, требующем переноса строки, список переносится целиком, и классы перечисляются с переносом строки после каждого из них.

Настроить автоматическое форматирование можно в Отладка -> Параметры -> Текстовый редактор -> C / C++ -> Форматирование.

# Важные замечания об ознакомительных работах

**Сохраните выполненные работы для шаблонов проектов**

Ход нижеследующих работ основывается на двух из них: некоторые работы используют классические решения Visual Studio со вложенными проектами, а некоторые – организацию проектов CMake (об этом далее). Завершенная работа «Быстрый старт разработки на C++» является шаблоном всех работ первого рода, а завершенная работа «Внешняя система сборки CMake» - шаблоном работ второго рода.

Обязательно сохраните для последующего использования каталоги выполненных работ «Быстрый старт разработки на C++», «Внешняя система сборки CMake». Позже будет рассмотрен способ быстрого развертывания оснастки на их основе (см. работу о контроле версий).

**Установите рекомендуемые параметры проектов**

Далее, для каждого из проектов внутри каждого из решений Visual Studio (проекты CMake не являются таковыми, см. работу «Внешняя система сборки CMake») установите параметры проекта.

Эти параметры принимает система сборки (в данном случае MSBuild) и передает соответствующие им опции во все остальные программы, участвующие в разработке (в т. ч. компилятор и компоновщик).

Итак, в Проект -> Свойства: [имя проекта] -> Свойства конфигурации должны быть установлены следующие значения параметров для всех дальнейших проектов Visual Studio:

* Общие -> Набор инструментов платформы (это и есть toolchain) – Visual Studio 2019 (…);
* Общие -> Стандарт языка C++ - Стандарт ISO C++17 (std:c++17);
* C / C++ -> Уровень предупреждений -> Уровень4 (/W4);
* C / C++ -> Уровень предупреждений -> Обрабатывать предупреждения как ошибки – Да (/WX).

**Одна Ознакомительная работа – одно решение Visual Studio или проект CMake (если не указано иного)**

Как сказано в главе о пользовательском интерфейсе, возможно организовать набор проектов внутри решения, соответствующих каждой работе.

При этом результирующим продуктом одной Ознакомительной работы, в отличие от типовых лабораторных работ, является не один, а несколько исполняемых модулей, так что имеет смысл рассматриваемая далее организация, когда одна Ознакомительная работа соответствует одному решению, и модули внутри неё – каждый своему проекту.

**«Проект Visual Studio», «Решение Visual Studio», «Проект MSBuild», «Проект CMake»**

Пользовательский интерфейс Visual Studio для начала работы предлагает создать проект, однако при создании проекта, не принадлежащего текущему решению, новое решение создается автоматически. Далее упоминание проекта Visual Studio иногда будет означать одновременно и проект, и решение, в котором он находится.

При создании проекта, если выбрать шаблон «Проект CMake» и не включать его в существующее решение Visual Studio, ни проект, ни решение Visual Studio созданы не будут (хотя проект CMake по-прежнему можно создать не сам по себе, а в рамках существующего решения, эта возможность в рамках данных рекомендаций не рассматривается).

Решение Visual Studio описывается файлами [название решения].sln (сокращение от SoLutioN) и [название решения].suo (сокращение от Solution User Options), а каждый проект Visual Studio (один или несколько) в рамках решения – файлом [название проекта].\*proj, например .vcxproj для Visual C++, .vbproj для Visual Basic, и т.д. Формат файлов .sln, описывающих решение, позволяет задать все принадлежащие решению проекты, а также наборы программ, предназначенные для работы с ними. Формат файлов .suo хранит некоторые настройки, заданные пользователем для данного решения, и не имеет значения для сборки.

О файлах .\*proj следует рассказать отдельно. Почти всегда они содержат структурированные данные в формате XML, задающие ход сборки для системы MSBuild. Условимся называть их сценариями MSBuild, хотя анализ XML и нельзя назвать интерпретацией сценария в приведенном ранее смысле. Таким образом, проекты и решения Visual Studio почти всегда являются также и проектами MSBuild, в противоположность проектам CMake.

# Ознакомительная работа 1.1 – Быстрый старт разработки на C++

После завершения настройки пользовательского интерфейса создайте новый проект, нажав на соответствующую плитку на Начальном экране или перейдя в Файл -> Создать -> Проект на Главном экране Visual Studio. Выберите название нового решения, в которое будет включен этот проект, и само название проекта.

Закончив создание, на Главном экране, где теперь открыто решение, а в нем проект, убедитесь, что установлены корректные значения параметров, описывающих необходимые настройки компиляции (см. Важные замечания об ознакомительных работах).

Убедившись, что окно Обозревателя решений присутствует на главном экране, добавьте в проект класс под именем Sorter (в окне Обозревателя Ctrl-Shift-A -> в модальном окне Добавить… выбрать Visual C++, затем Класс C++ и указать имя в соответствующем поле), а также файл исходного кода runtests.cpp (аналогично, но вместо добавления Класс C++ следует добавить Файл C++ (.cpp)).

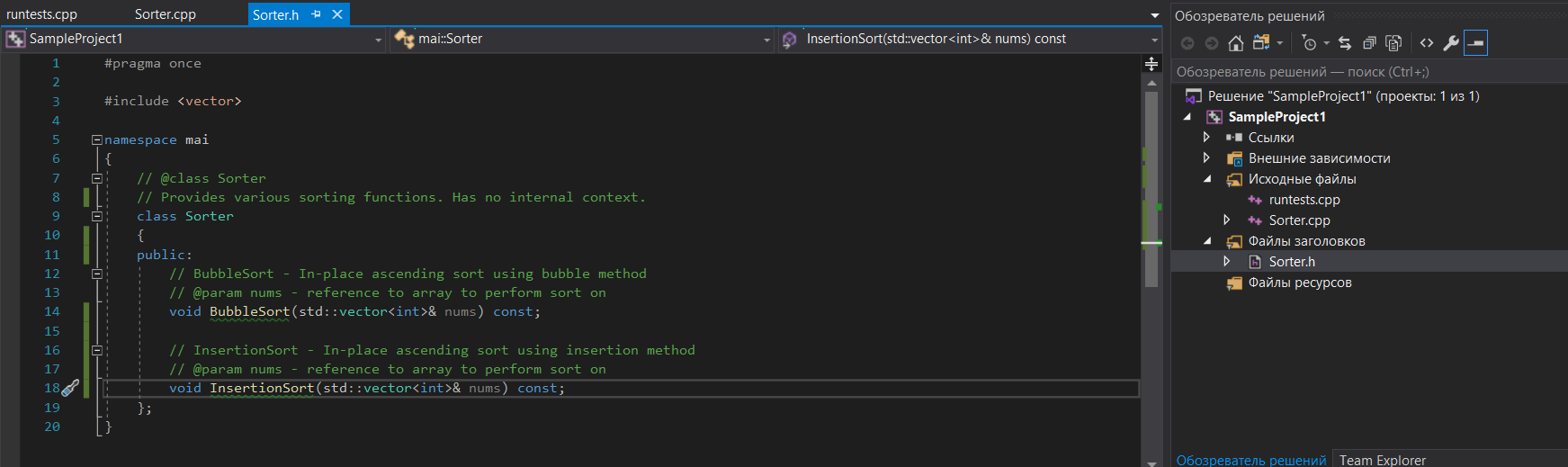


Рисунок 4 - Общий вид проекта и кода заголовочного файла класса Sorter

Далее для каждого из методов класса Sorter в контекстном меню, открываемом правым кликом по названию метода, выберите Быстрые действия и рефакторинг -> Создать определение … в Sorter.cpp.

Реализуйте методы, заполнив сгенерированное определение.

Установите следующие значения параметров проекта:

* Общие -> Тип конфигурации – Приложение (.exe);
* Компоновщик -> Система -> Подсистема – Консоль (/SUBSYSTEM:CONSOLE).

Теперь, когда класс Sorter реализован и проект сопоставлен исполняемому модулю (это консольное приложение), попробуйте выполнить сборку (ПКМ по названию проекта в дереве Обозревателя решений -> Собрать или в элементе Цель отладки на панели Стандартная ЛКМ по Локальный отладчик Windows, сборка произойдет автоматически перед запуском).

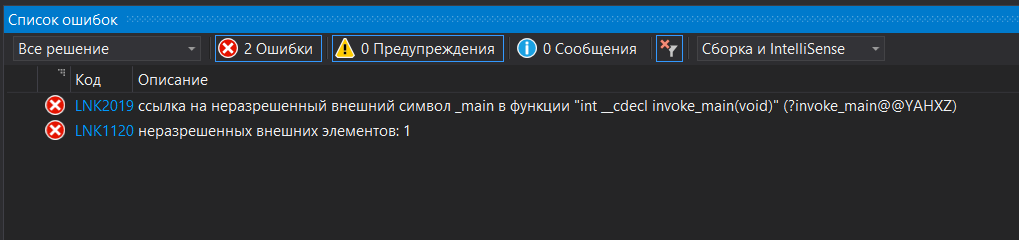


Рисунок 5 - Ошибка компоновки в ходе сборки приложения

Если класс был реализован корректно, компиляция пройдет успешно, а компоновка завершится с указанными на Рисунке 5 ошибками (ошибки компоновщика в формате вывода средств Visual C++ имеют префикс LNK).

Результирующий исполняемый модуль для выбранной конфигурации (Приложение (.exe)) не может быть скомпонован, поскольку в исходном коде, предоставленном пользователем (программистом), не определена функция main.[[16]](#footnote-16)

По-прежнему можно собрать имеющийся код, изменив Проект -> Свойства [имя проекта] -> Свойства конфигурации -> Общие -> Тип конфигурации -> Приложение (.exe) на Динамическая библиотека (.dll) или Статическая библиотека (.lib). В обоих случаях сборка завершится успешно, однако исполняемый модуль (называемый динамической или статической библиотекой) невозможно непосредственно выполнить[[17]](#footnote-17), поскольку он не имеет упомянутой в разделе «Реализация трансляции исходного кода на C++ в ходе сборки» entry point, то есть точки входа. В Ознакомительной работе, посвященной применению dynamic linking / static linking, будет рассмотрено использование таких модулей.

Верните исходный тип конфигурации, в файле исходного кода runtests.cpp определите функцию main, включающую в себя использование класса Sorter, и выполните сборку повторно, после чего запустите собранное приложение.

В ходе разработки может потребоваться привести в соответствие отладочную информацию с текущим проектом, очистив и пересобрав проект, открыть проект в файловом менеджере, просмотреть диаграмму классов и их связей, переименовать проект или настроить контроль версий; для этого удобно воспользоваться контекстным меню (ПКМ по названию проекта в окне Обозревателя решений).

В ходе работы был собран проект и запущено результирующее приложение. Убедитесь, что приложение работает корректно (оно должно на данном этапе сортировать последовательности) и переходите к следующей работе.

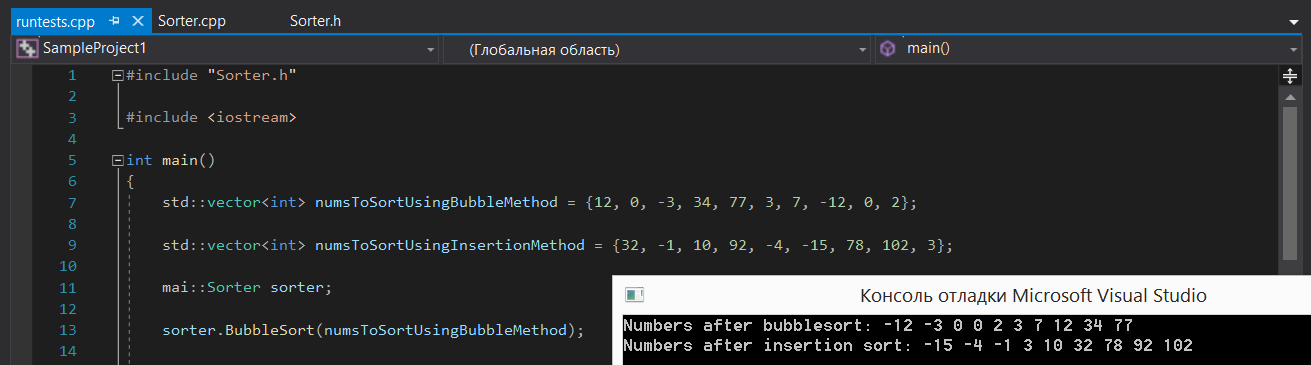


Рисунок 6 - Возможная реализация runtests.cpp и корректная работа класса Sorter

# Ознакомительная работа 1.2 – Внешняя система сборки CMake

В настоящее время (осень 2019) не представляется возможным конвертировать проект Visual Studio в проект CMake без внешних инструментов, несовместимых с Visual Studio рассматриваемой версии, а создание проекта CMake внутри существующего решения Visual Studio приводит к генерации избыточного числа сценариев CMake для всей иерархии проектов и корневого решения.

Поэтому код предыдущей работы будет импортирован в новую вручную, перемещением в файловом менеджере (см. предпоследний абзац предыдущей работы об удобном способе вызвать этот менеджер).

Создайте для текущей Ознакомительной работы новый проект Sorter (и включающее его в себя новое решение IntroductionToCMake), выбрав шаблон Проект CMake (Начальный экран -> Создание проекта -> «cmake» в строку поиска -> Проект CMake или Главный экран -> Файл -> Создать -> Проект… -> «cmake» в строку поиска -> Проект CMake).

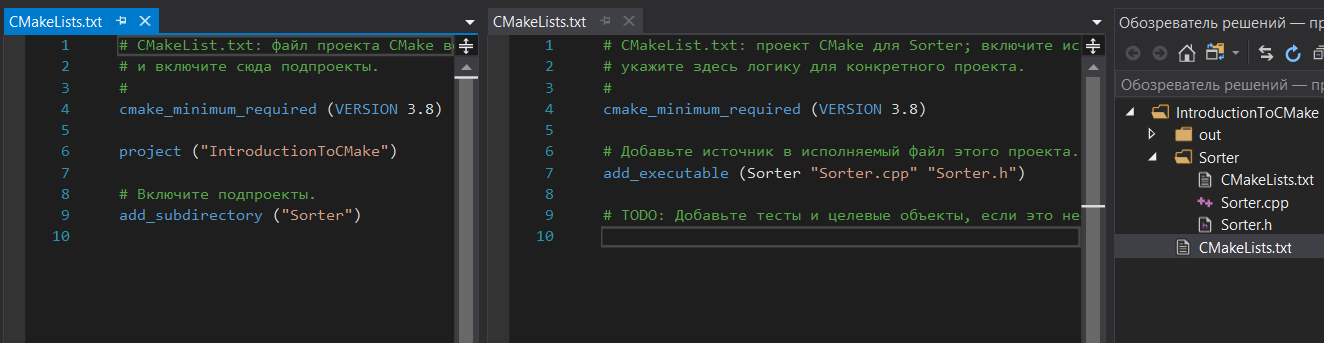


Рисунок 7 - Автогенерируемые сценарии CMake для решения и вложенного проекта

**О сущностях CMake**

**CMake target** – это приложение (**CMake executable**), библиотека (**CMake library**) или какой-то другой, неспециализированный результат сборки (**CMake custom target**). Ближайший аналог в сущностях Visual Studio (строго говоря, сущностях MSBuild) – проект.

**CMake project** – это именованное и логически объединенное множество CMake target’ов. Отношение CMake target к CMake project похоже на отношение проекта к решению (множество проектов принадлежат одному решению). Ближайший аналог в сущностях MSBuild – решение (!).

**CMakeLists.txt[[18]](#footnote-18)** – это файл, от которого в файловой системе начинается структура каталогов проекта (каталог, в котором находится CMakeLists.txt – корневой) и который описывает CMake project, принадлежащие ему CMake targets, файлы исходного кода и ресурсы для сборки, и так далее. CMakeLists.txt может указывать на вложенный каталог, имеющий свой CMakeLists.txt, описывающий другой CMake project, и так далее. Ближайший аналог для MSBuild – сценарии с расширением \*proj, описывающие проекты, и документы .sln, описывающие решения.

На Рисунке 7 – пример такой организации, когда решению Visual Studio и проекту, вложенному в это решение, соответствуют каждому свой CMakeLists.txt.  
В конечном итоге, CMakeLists.txt с наименьшим уровнем вложенности позволяет собрать не только CMake target'ы в составе собственного CMake project, но также CMake target’ы в составе CMake project’ов, заданных во всех CMakeLists.txt, расположенных на один или несколько уровней ниже (во вложенных каталогах).

**CMake generator** – это транслятор CMakeLists.txt в представление нижележащей системы сборки (сценарий MSBuild, makefile[[19]](#footnote-19), сценарий Ninja и т.д.), которая будет выполнять непосредственно сборку. Как было сказано ранее, CMake – это front end, позволяющий делегировать сборку многим другим системам сборки. Аналога для MSBuild не существует, поскольку MSBuild является конечной нижележащей системой сборки, а не делегирует сборку кому-то еще.

**CMake cache** – это результат работы CMake, включающий в себя сценарий нижележащей системы сборки. Аналога для MSBuild не существует по причине отсутствия нижележащей системы сборки (см. CMake generator).

Результатом запуска CMake является CMake cache. CMake не требует генерировать CMake cache повторно, если не были внесены изменения в список файлов, относящихся к CMake target (например, удален или добавлен новый файл; изменение содержимого не требует повторного запуска) или каким-то иным образом не был нарушен ход сборки.

Для запуска CMake, генерации CMake cache и т.о. получения Visual Studio актуальных данных о сборке выполните Проект -> Создавать кэш для [имя проекта]. Для создания CMake cache для конкретного CMakeLists.txt откройте на Главном экране соответствующий CMakeLists.txt и в контекстном меню (ПКМ по заголовку вкладки с файлом на Главном экране) выберите «Сохранить CMakeLists.txt». Все вышеупомянутые действия будут выполнены автоматически. [[20]](#footnote-20)

Visual Studio не работает с CMake напрямую. CMake генерирует сценарий Ninja **(не сценарий MSBuild!),** который запускается при сборке того или иного CMake target, входящего в состав текущего CMake project. Ninja входит в состав средств CMake для Windows в Visual Studio.

Также может потребоваться удаление предыдущего CMake cache, для этого воспользуйтесь Проект -> Кэш CMake (только для [конфигурация]) -> Удалить кэш.

Создайте во вложенном каталоге Sorter, представляющем проект Sorter, каталоги для заголовочных файлов и файлов исходного кода – “include” и “src” соответственно. Удалите автоматически сгенерированные Sorter.h и Sorter.cpp, а вместо них поместите в include и src Sorter.h и Sorter.cpp из работы «Быстрый старт разработки на C++». В src создайте подкаталог tests и переместите в него runtests.cpp из упомянутой работы (это можно сделать перетаскиванием файлов из предыдущей работы в Обозреватель решений).

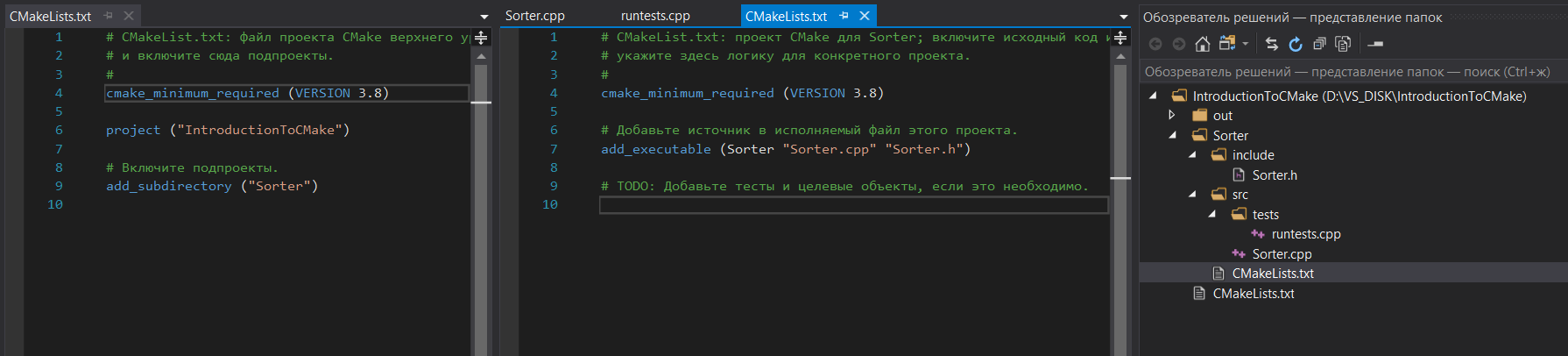


Рисунок 8 - Итоговая структура проекта Sorter в составе решения IntroductionToCMake

Теперь генерирование корректного CMake cache (и, соответственно, сборка нижележащей системой исполняемого модуля) невозможно, поскольку описанный процедурой add\_executable CMake target содержит несуществующие файлы. Попробуйте (см. выше инструкцию для создания кэша) запустить CMake для любого из CMakeLists.txt и убедитесь в возникновении цепочки ошибок, включающей в себя «Cannot find source file …».

Удалите из CMakeLists.txt проекта Sorter автоматически сгенерированный CMake target соответствующий приложению, удалив строку add\_executable(..) (или закомментировав символом # в начале строки), и повторите процедуру. Теперь результирующий сценарий Ninja не содержит объектов для сборки, о чем говорит пустой диапазон объектов на панели инструментов Стандартная (хотя если до этого хотя бы одна попытка сборки закончилась успешно и вывод не был удален, по-прежнему возможен запуск собранных ранее объектов).

Теперь преобразуйте проект Sorter так, чтобы он представлял самостоятельный программный продукт – библиотеку Sorter и тесты, проверяющие ее корректность (пока полуавтоматические).

Объявите об использовании стандарта C++17 следующим фрагментом кода на языке сценариев CMake, расположив его сразу после объявления минимальной версии проекта:

set(CMAKE\_CXX\_STANDARD 17)

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

set(CMAKE\_CXX\_EXTENSIONS OFF)

Далее, объявите переменную, хранящую список файлов исходного кода библиотеки Sorter. Это необязательно (можно перечислить файлы непосредственно при объявлении target), но удобно.

set(Sorter-sources src/Sorter.cpp)

Объявите CMake target (а именно CMake library), являющийся библиотекой, и явно укажите тип компоновки (и т.о. определите реализацию исполняемого модуля – статическая библиотека – являющуюся результатом успешной сборки, указав спецификатор STATIC[[21]](#footnote-21)):

add\_library(Sorter STATIC ${Sorter-sources})

Разрешите этому target доступ к заголовочным файлам проекта, находящимся в каталоге include, притом разрешите этот доступ как самому target, так и всем его пользователям, выбрав модификатор доступа PUBLIC:

target\_include\_directories(Sorter PUBLIC include)

Основная часть работы завершена, теперь объявите CMake target (конкретно CMake executable), соответствующий исполняемому модулю тестов:

add\_executable(Sorter-runtests src/runtests.cpp)

И, наконец, задайте компоновку модуля тестов с модулем библиотеки. Разрешать доступ к заголовочным файлам проекта отдельно не требуется, поскольку такой доступ есть у модуля библиотеки, и модуль тестов его наследует.

target\_link\_libraries(Sorter-runtests Sorter)

Сохраните завершенный CMakeLists.txt проекта Sorter, и, если CMake cache не был создан автоматически, выполните необходимые манипуляции для его создания, выберите доступный теперь на панели инструментов Стандартная объект сборки – модуль тестов – соберите и запустите его. Работа завершена.

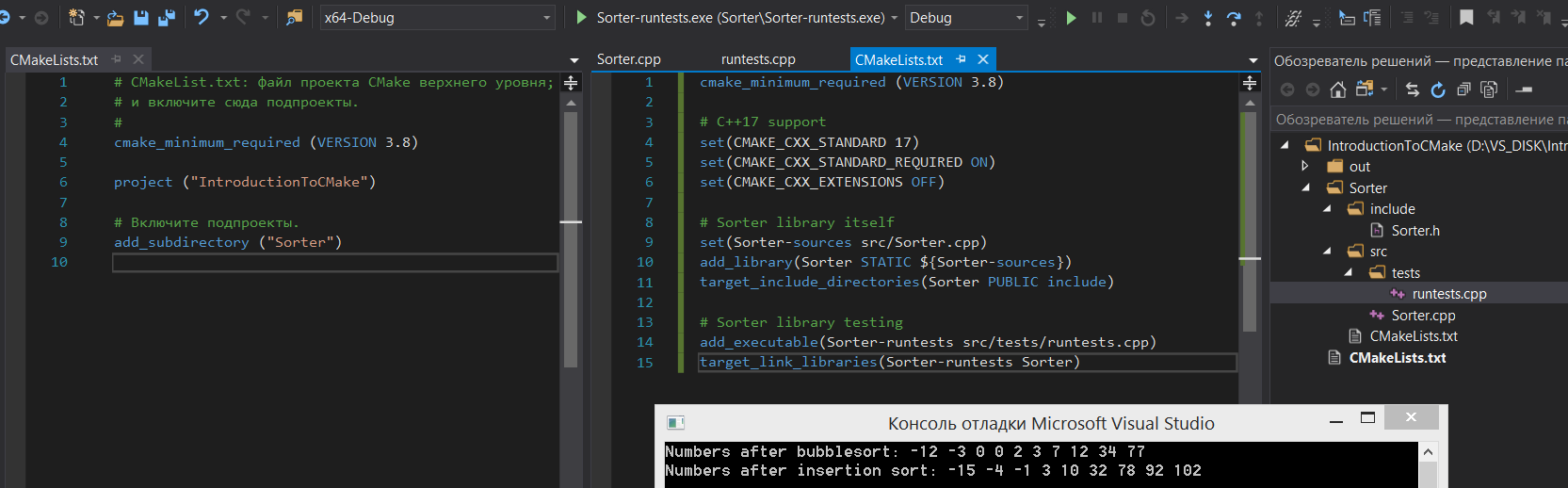


Рисунок 9 - Итоговый вид и результат работы модуля тестов в составе CMake проекта Sorter

Организация CMake targets, подобная таковой для классического проекта Visual Studio в предыдущей работе, тривиальна (отсутствуют инструкции компоновки, только один target – CMake executable, которому переданы Sorter.cpp и runtests.cpp) и рассматриваться отдельно не будет.

Перейдите к следующей работе для ознакомления с библиотекой автоматического тестирования googletest, её установкой, компоновкой и использованием.

# Ознакомительная работа 1.3 – Автоматизированное тестирование с Google Test

Вторая часть этой работы будет основана на предыдущей работе **(«Ознакомительная работа 1.1 - Внешняя система сборки CMake»)**, поэтому имеет смысл скопировать предыдущий проект, и продолжить с места завершения. Далее (в соответствующей работе) будет рассмотрен другой подход к шаблонам проектов, предусматривающий возможность клонирования текущего состояния / восстановления предыдущего (см. работу о контроле версий).

В числе исполняемых модулей для каждой из предыдущих работ было приложение, позволяющее проверить корректность реализации (в данном случае сортировки массива), вручную сравнив вводную последовательность и результат сортировки, проверив, что одна из них является перестановкой другой, и что для результата выполняется условие расположения в порядке невозрастания / неубывания.

Для тщательного тестирования данного аспекта реализации Sorter в рамках сжатых сроков имеет смысл перейти к автоматизированному тестированию.

Библиотека автоматизированного тестирования Google Test (далее сокращенно **googletest** или gtest) может быть интегрирована в проект CMake средствами компоновки и конфигурации CMake, или в решение Visual Studio адаптером MSBuild для Google Test (см. Установка Visual Studio), или в решение Visual Studio, будучи организована как проект googletest, от которого зависим проект тестовой оснастки. Рассмотрим все три варианта.

**Проект Visual Studio, декорированный тестовой оснасткой Google Test (вариант 1)**

Воспользуйтесь шаблоном проекта «Google Test» (на Главном экране Файл -> Создать -> Проект.. -> Google Test, доступен после установки соответствующего компонента, см. Введение).

После выбора вида компоновки проекта с googletest (рекомендуется статическая) можно сразу же собрать и запустить проект, несмотря даже на отсутствие main. Googletest – это не один исполняемый модуль, а два[[22]](#footnote-22): модуль gtest предоставляет возможности для тестирования, а gtest\_main тривиальную реализацию main по умолчанию.[[23]](#footnote-23) Для декорирования хода работы тестовой оснастки main можно реализовать самостоятельно, остальное Visual Studio сделает автоматически.

На этом первая часть работы закончена; тестирование Sorter будет подробно освещено в следующих.

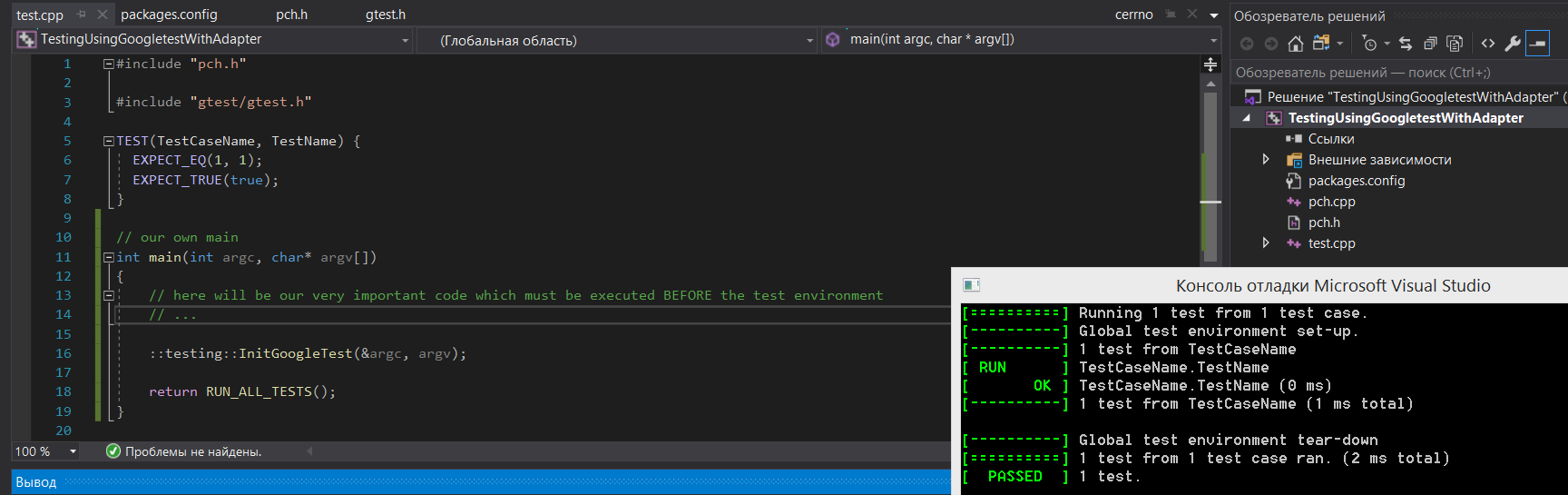


Рисунок 10 - Собственная реализация main, инициализирующей Google Test и запускающей тесты

**Сборка Google Test, её установка и компоновка с проектом CMake (вариант 2)**

Рассмотрим сборку googletest как обыкновенного CMake target, компонуемого с пользовательскими CMake targets.

Начните работу с изучения инструкций по сборке проекта googletest (<https://github.com/google/googletest/blob/master/googletest/README.md>).

В разделе «Build with CMake : Incorporating Into An Existing CMake project» описывается встраивание googletest в существующий проект CMake. Такое встраивание возможно осуществить несколькими способами (см. упомянутый раздел), среди которых предпочтительным на данный момент является загрузка и сборка googletest средствами CMake во время генерации CMake cache корневого проекта (последний пункт среди перечисленных в разделе).

Выполните действия, описанные в «Build with CMake : Incorporating Into An Existing CMake project», с некоторыми изменениями:

* Сценарий CMake, описывающий загрузку googletest, назовите не CMakeLists.txt.in, а googletest.cmake.in[[24]](#footnote-24);
* Во фрагменте, предназначенном для вставки в существующий сценарий CMake, пример компоновки googletest с существующим target удалите, а сам фрагмент вставьте в конец корневого CMakeLists.txt;
* В CMakeLists.txt вложенного CMake project (в работе «Внешняя система сборки CMake» это тот CMakeLists.txt, в котором объявлен Sorter) укажите необходимость компоновки исполняемого модуля тестов с googletest (CMake target для googletest называется gtest, CMake target, поддерживающий запуск тестов без пользовательской функции main, называется gtest\_main).

В процессе работы возможны проблемы (в виде ошибок CMake / нижележащей системы сборки), связанные с присутствием результата прошлой сборки или генерации CMake cache. Как известно из работы «Внешняя система сборки CMake», для их решения может потребоваться удаление и пересоздание кэша (описано в ней же) или пересборка всех имеющихся объектов сборки (Сборка -> Перестроить всё).

Изучив примеры в googletest samples (https://github.com/google/googletest/tree/master/googletest/samples), реализуйте набор тестов, использующих в том числе assertions (набор макросов ASSERT\_\*), и переходите к следующему варианту выполнения работы.

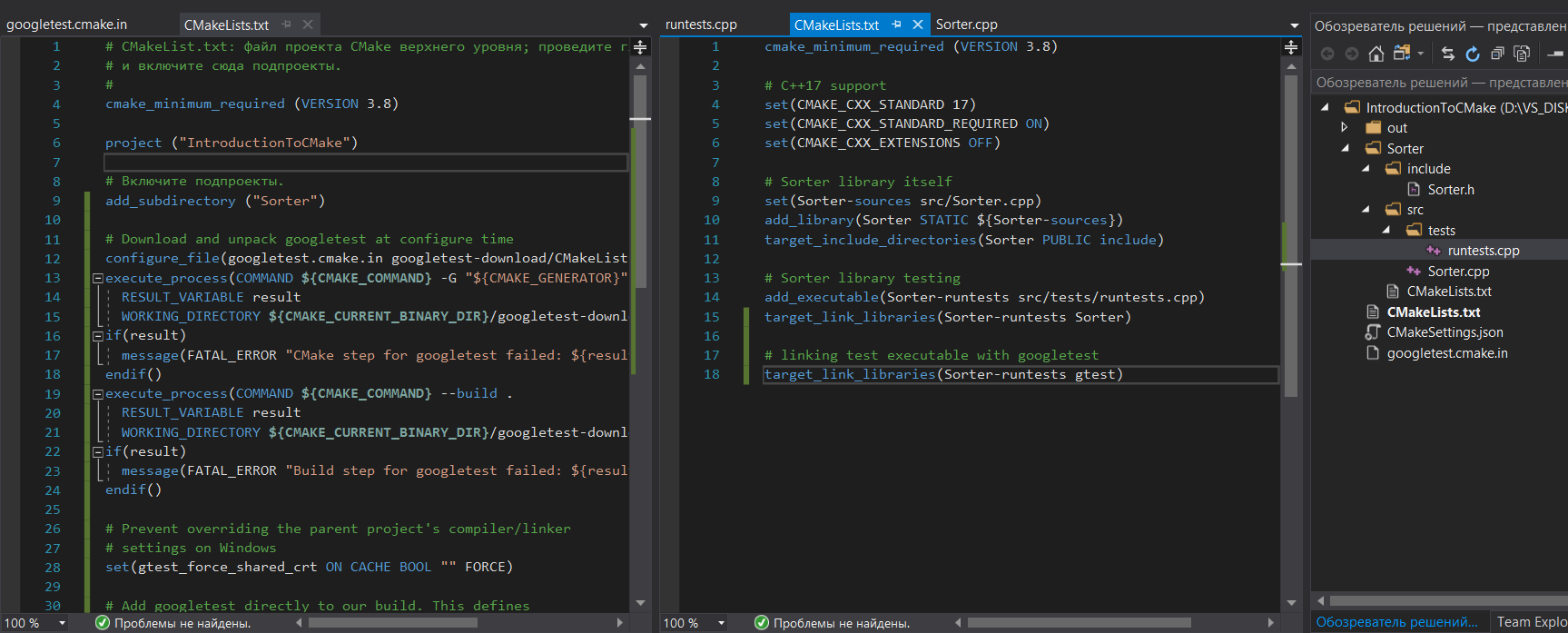


Рисунок 11 - Окончательная структура проекта,включая средства тестирования

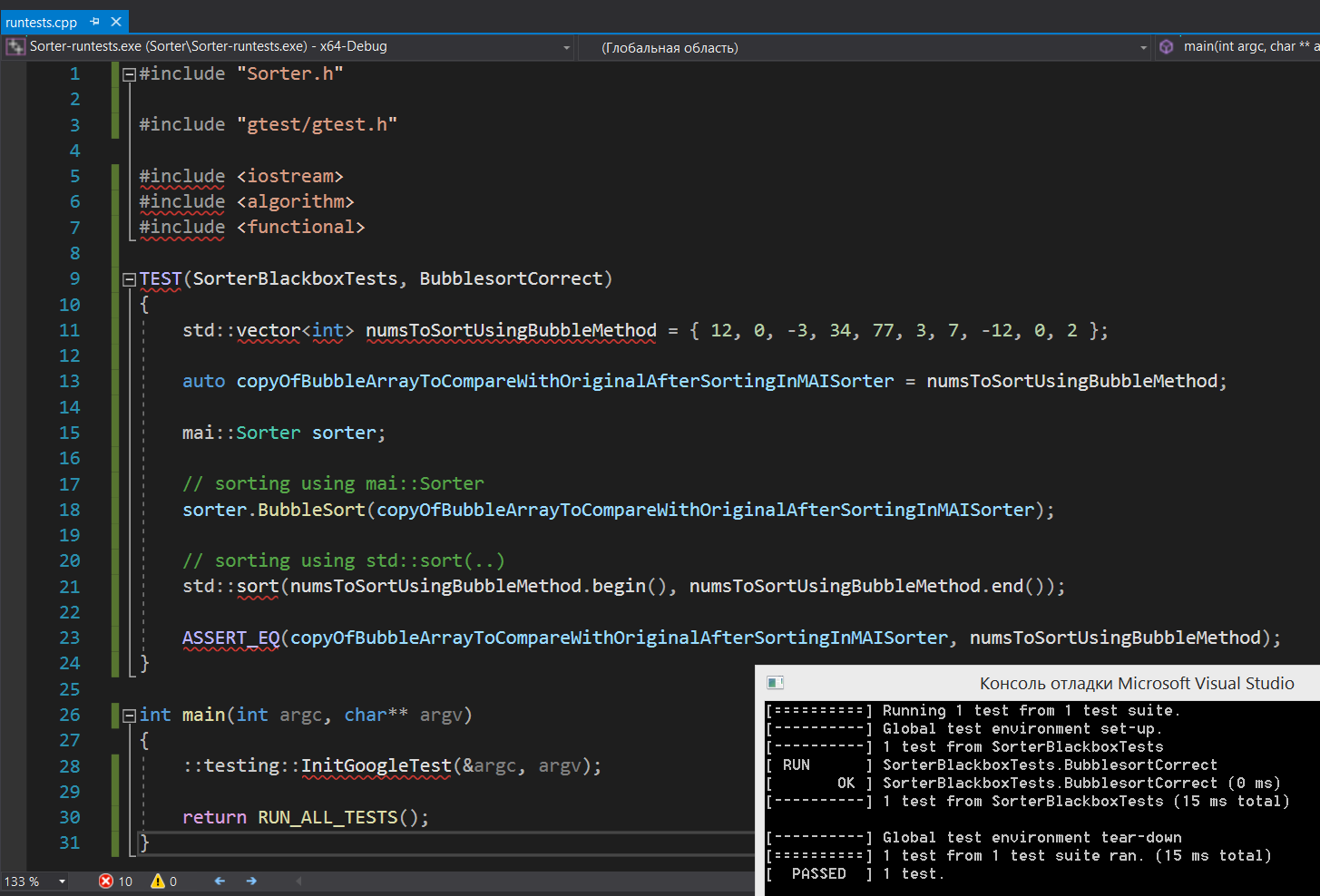
- 

Рисунок 12 - Пример набора тестов из одного теста mai::Sorter::BubbleSort(..) и результат выполнения такового

**Интеграция googletest в существующее решение Visual Studio (вариант 3)**

Перед началом работы настройте PATH согласно соответствующему разделу.

Воспользуйтесь результатом работы **«Ознакомительная работа 1.1 - Быстрый старт разработки на C++»,** далее назовем его шаблонным решением.

Создайте новое решение, а в нем проекты, представляющие исполняемый модуль и объект тестирования (на этот раз, в отличие от шаблонного решения, это будут два разных проекта, а не один), затем перенесите в эти проекты файлы шаблонного решения.

Или создайте в шаблонном решении новый проект и перенесите в него тестовый модуль. Исключите, то есть удалите из дерева проекта без физического удаления, тестовый модуль из исходного проекта (ПКМ по runtests.cpp в исходном проекте -> Удалить -> Исключить), скопируйте в файловом менеджере тестовый модуль в каталог нового проекта и добавьте в новый проект. Так как в шаблонном решении проект был один, потребуется также в свойствах двух новых проектов задать Тип приложения Статическая библиотека и Приложение (.exe) для объекта тестирования и тестового модуля соответственно, а затем наладить компоновку тестового модуля с объектом тестирования.

Далее предстоит сделать следующее: клонировать репозиторий googletest, интегрировать googletest в имеющееся решение так, чтобы сборка googletest происходила перед сборкой остальных проектов, каталоги include включали в себя таковые для googletest, а тестовый модуль был скомпонован с googletest, собранным в исполняемый модуль (в данном случае статическую библиотеку).

После завершения декомпозиции шаблонного решения интегрируйте в него проект googletest. Настроив PATH, начните с клонирования удаленного репозитория googletest средствами Git.

В интерпретаторе команд, запущенном из каталога, не принадлежащего решению:

git clone <https://github.com/google/googletest>

Далее в клонированном проекте (который, к счастью, содержит готовый CMakeLists.txt, благодаря чему нет необходимости портировать проект на CMake, создавая для него CMakeLists.txt самостоятельно) выполните действия, описанные разделом «Генерация сценария нижележащей системы сборки с помощью CMake. Сборка с использованием нижележащей системы сборки» Приложения, исключая окончательную ручную сборку (в ней нет необходимости, она будет выполнена в ходе сборки решения).

Итак, решение, представляющее googletest, сгенерировано. В этом решении находится один или несколько проектов, представляющих библиотеку googletest. Включите минимальный их набор в основное решение, укажите в соответствии с Приложением зависимость проекта тестового модуля от проекта googletest, реализуйте в тестовом модуле набор тестов и их вызов, соберите решение (может потребоваться отдельная сборка проекта googletest: ПКМ по проекту в Обозревателе -> Собрать). Работа завершена.

# Ознакомительная работа 1.4 – Контроль версий с помощью git

Перед началом работы убедитесь, что в PATH установлено расположение git.

**О системе контроля версий Git**

Git реализует средства управления виртуальной файловой системой (и небольшой версионный front end над ними, позволяющий манипулировать этой файловой системой, переключаясь между ревизиями), почти полностью описываемой содержимым каталога .git, находящегося в корневом каталоге репозитория.

Git регистрирует изменения в репозитории, выполняя снимки (**snapshots**) этой файловой системы. Пользователь (программист) в процессе разработки время от времени вызывает Git для регистрации изменений файлов; Git же в свою очередь изменяет состояние виртуальной файловой системы.

Git – полностью распределенная система, в которой нет репозитория, который пользовался бы большим доверием, чем остальные; все репозитории равнозначны, нет разделения на клиенты и сервер, который предоставляет некое доверенное состояние репозитория, от которого наследуются все другие.

**О сущностях Git и контроля версий в целом**

**Revision** («ревизия») – состояние файла или файловой системы, в котором они находятся прямо сейчас или находились в какой-то момент времени.

**Commit** («коммит») – процесс создания revision, включающий в себя создание описания файлов пользователя или изменений в них и сопоставление этому описанию ключа (в случае git это SHA-1 хэш). Также применяется в значении revision. Также может обозначать процесс записи пары ключ-описание в репозиторий.

**Branch** («ветка») – именованная последовательность ревизий, как правило, имеющая семантическое значение (скажем, первоначально существующая в созданном репозитории ветка master рекомендуется для ревизий, соответствующих стабильной и корректной работе программы).

**Git Repository** (далее кратко **repository**, «репозиторий») – хранилище, содержащее виртуальную файловую систему, со всеми ее ревизиями, ветками, переходами между ними и т.д.

**Local Repository** – репозиторий, представленный виртуальной файловой системой, работа с которой идет в текущий момент. Файлы локального репозитория находятся в ./.git.

**Remote Repository** («удаленный репозиторий») – репозиторий, не являющийся local repository; также ссылка (представленная файлом) на remote repository, содержащаяся в local repository, позволяющая установить соединение и обновить состояния одного или обоих репозиториев.

**Working Tree** [[25]](#footnote-25)– это каталог физической файловой системы, в том числе его файлы и подкаталоги, которому соответствует репозиторий. Для Git это каталог репозитория, в котором находятся пользовательские файлы и каталог .git, представляющий виртуальную файловую систему.

**Staging Area** (также **Index**) – файлы и каталоги из Working Tree, отмеченные для добавления в Local Repository во время выполнения следующего Commit.

По умолчанию, Git в описании коммитов отождествляет коммиты с реквизитами учетной записи пользователя, в сессии которого была выполнена команда git. Вы можете задать новые данные о текущем пользователе, внеся изменения в файл конфигурации, сопоставленный с учетной записью пользователя в Windows.

git config --global user.name “John Doe”

git config --global user.email [johndoe@example.com](mailto:johndoe@example.com)

**Ручная работа с репозиторием средствами git**

Выберите выполненную работу или пример интеграции библиотеки, наиболее подходящий для формирования на его основе шаблона для дальнейшей работы. Перейдите в его каталог.

Создайте новый Local Repository, выбрав каталог проекта (текущий каталог) в качестве Working Tree.

git init

Обратите внимание, что в каталоге проекта появился каталог ,git, описывающий репозиторий.  
Выведите на экран текущее состояние Local Repository, Staging Area и Working Tree:

git status

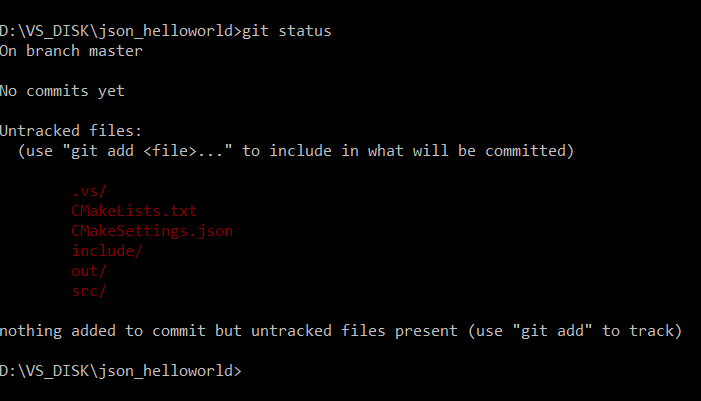


Рисунок 13 – Статус local repository: выбрана ветка master, которая не содержит ни одного коммита. В working tree находятся следующие файлы, которые не были добавлены в staging area.

Опишите изменения файловой системы, которые не должны фиксироваться локальным репозиторием (не должны добавляться в staging area и commit’иться в local repository). Это файлы настроек, специфичных для среды разработки, и каталоги результатов сборки.

echo 2> .gitignore [[26]](#footnote-26)// создание файла .gitignore в каталоге проекта

notepad .gitignore // открытие .gitignore

.gitignore – это файл, описывающий шаблоны игнорируемых git файлов. Откройте его для редактирования и добавьте каталоги .vs и .out – они не имеют отношения к распространяемому репозиторию.

В итоге .gitignore содержит как минимум эти два шаблона, описывающие конкретные каталоги и файлы:  
.vs/

out/[[27]](#footnote-27)

Сохраните .gitignore (обратите внимание, что теперь git status больше не выводит на экран игнорируемые файлы и каталоги) и зафиксируйте изменения в .gitignore в local repository.

git add .gitignore // добавить .gitignore из working tree в staging area для дальнейшего фиксирования

git commit -m “introduced .gitignore” // фиксировать изменения, объединив их в commit в ветке master. Staging area снова пуста, local repository теперь содержит commit с .gitignore.

Вызовите git status, чтобы отобразить оставшиеся в working tree файлы и каталоги, состояние которых расходится с таковым в local repository (по понятным причинам в их число больше не входит .gitignore, если он не менялся с момента последнего commit).

Вызовите git log, чтобы отобразить последовательность commit’ов вместе с ключом для их использования – хэшем SHA-1. Убедитесь, что присутствует коммит «introduced .gitignore».

Итак, .gitignore зафиксирован в системе контроля версий, и это последнее (а в нашем случае и первое) зафиксированное стабильное состояние репозитория. Для дальнейшей работы переключитесь на ветку разработки, для этого нужно синхронизировать working tree с local repository целиком. Так как фиксировать непроверенное состояние проекта в master не следует, воспользуйтесь git stash.

git add -A // добавить всё working tree в staging area

git stash // поместить staging area в stash

git status // диагностическая команда, ожидаемый вывод – «working tree clean»[[28]](#footnote-28)

git stash list // диагностическая команда, ожидаемый вывод – одна запись stash@{0}, содержащая файлы проекта

git branch development // создать ветку development

git checkout development // переключить Working Tree на ветку development

git stash pop // переместить последний (в нашем случае и единственный) снимок из stash в working tree; таким образом, продолжить работу с проектом в ветке development

git status // диагностическая команда, ожидаемый вывод – текущая ветка development, файлы проекта находятся в working tree

Приведите проект в некое шаблонное состояние, затем зафиксируйте его новым commit в ветку development.

git add -A // добавить все файлы проекта в staging area

git commit -m “project template, see more in README.md” // зафиксировать содержимое staging area в local repository

Проверьте работу (в том числе факт успешной сборки) проекта. В случае каких-то неудовлетворительных моментов, требующих доработки, внесите нужные изменения, добавьте измененные файлы в staging area (git add [имя файла]), затем зафиксируйте изменения новым commit’ом (git commit -m “[сообщение, описывающее суть проблемы]” или, в случае commit, вносящего доработки в предыдущий, git commit --fixup HEAD).

Приведя проект в состояние, в котором он может быть немедленно использован в качестве шаблона пользовательских проектов, переключитесь (убедившись, что working tree пусто) в ветку master и выполните слияние **(merge)** с веткой development,которое в нашем случае (master и development не имеют различающихся коммитов с одним родителем) будет заключаться в перемотке (**fast-forwarding**) ветки master, то есть последовательном применении к ней отсутствующих в master коммитов ветки development.

Затем верните working tree в соответствие с веткой development для продолжения разработки. Если у вас в stash были изменения, которые вы не были готовы зафиксировать в local repository, примените их после переключения в ветку development.

[git add -A; git stash] // опционально, если working tree не пусто

git checkout master // переключить Working Tree обратно на ветку master

git merge development

git checkout development

[git stash pop] // опционально, если working tree не было пусто

Однако, README.md еще не написан! Создайте описание типа шаблона (типовой проект CMake, проект MSBuild (проект Visual Studio)), наиболее распространенных сценариев его применения и т.д., в README.md:

echo 2> README.md

notepad README.md

Закончив с readme, выберите стратегию, в соответствии с которой файл readme будет внесен в систему контроля версий:

* Будет сделан новый коммит, описывающий добавление readme, затем тривиальное слияние master с development, которое заключается в fast-forwarding для ветки master на один коммит;
* Будет выполнен fixup для предыдущего коммита (git commit --fixup HEAD), затем все тот же fast-forwarding;
* После создания нового коммита (fixup или нет), будет выполнено слияние или cherry-picking (применение конкретного коммита) из development, а затем перебазирование **[[29]](#footnote-29)(rebase)** ветки master.

Следует рассмотреть эти стратегии подробнее. Первая, очевидно, позволяет быстро продолжить работу, однако нарушает политику коммитов в master, согласно которой master корректно и стабильно работает в любом состоянии, в том числе когда пользователь имеет старую версию master. Вторая также нарушает эту политику, однако оставляет семантически ясное дополнение к некорректному коммиту, в котором заявлено о README.md, содержащем сведения об использовании проекта, который в действительности отсутствует. Третья соблюдает политику коммитов, однако приводит к тому, что все пользователи контроля версий, успевшие получить предыдущую версию коммитов master, подвергшихся перебазированию, и начавшие разработку на их основе, вынуждены выполнить вместо автоматического fast-forwarding сложное слияние[[30]](#footnote-30) зафиксированной в их копии репозитория старой версии коммита, своих результатов на его основе, и новой версии коммита, а также дальнейших коммитов в удаленном репозитории. Это слияние может завершиться неудачей и потребовать ручного разрешения противоречащих друг другу изменений из параллельных коммитов.[[31]](#footnote-31)

Итак, для реализации первой стратегии выполните коммит в development, добавляющий readme, затем переключитесь в master и выполните слияние (git add README.md; git commit –m “introduced README.md”; git checkout master; git merge development). Для реализации второй вместо коммита с явно указанным сообщением выполните git commit --fixup HEAD.

Что же касается третьей стратегии, то она несколько более громоздка и требует настройки remote repository для перебазирования.[[32]](#footnote-32)

Создайте новый репозиторий локально или на удаленном сервере, например, сервере Github. Для локального репозитория сохраните путь к каталогу, для удаленного – адрес до сервера и путь до каталога на сервере, например https://github.com/user/template\_project.git. Обратите внимание, что локальный репозиторий, который предполагается использовать в качестве remote repository, следует инициализировать с ключом bare (git init --bare), чтобы избежать некоторых трудностей с настройкой обновления его состояния.

Добавьте репозиторий (вне зависимости от его расположения, он будет для нашего репозитория remote repository) в число отслеживаемых remote repository (сокращенно remotes).

git remote add [путь] [имя remote, например origin]

Наконец, назначьте upstream[[33]](#footnote-33) для своего репозитория и выполните push, то есть обновление состояния удаленного репозитория.

[из соотв. локальной ветки, в нашем случае master]

git push --set-upstream origin master

Push в локальный репозиторий не требует дополнительных разрешений; настройка доступа к Github для обновления репозиториев на Github не рассматриваются, поскольку расширение Github для Visual Studio содержит в том числе OAuth-подсистему, которая при попытке выполнить Push должна запросить данные для аутентификации.

После успешной настройки remote repository приступите к реализации третьей стратегии.  
…  
git merge development // находясь, очевидно, в ветке master

ИЛИ  
git cherry-pick development~0[[34]](#footnote-34) // cherry-picking коммита development~0, описывающего README.md

Далее, выполните rebase, указав в качестве опорного коммит, находящийся перед объединяемыми, и притом не локальный, а соответствующий ему коммит в origin (remote repository, выбранный в качестве upstream).

git rebase --interactive [sha-1 предшествующего коммита или origin/master~[индекс коммита, начиная с 0, от головы master], например origin/master~2] // перебазирование в интерактивном режиме

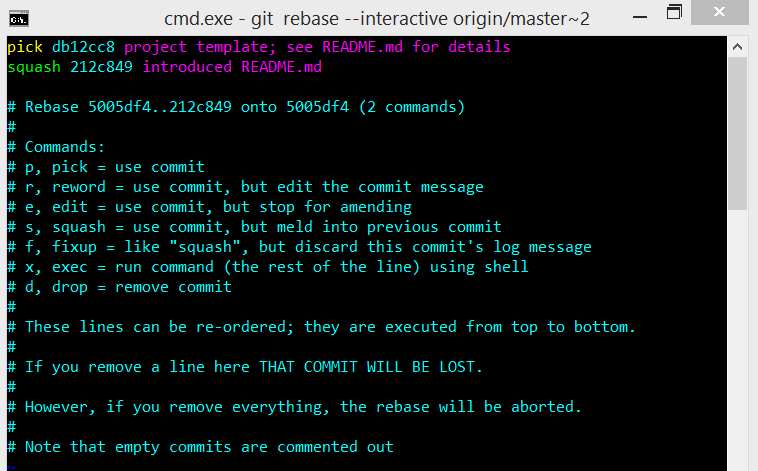


Рисунок 14 - Перебазирование; коммит 212c849 будет объединен с предыдущим; получившийся коммит будет иметь изменения из обоих

Ознакомьтесь с подсказкой git, закомментированной в файле, который открылся в текстовом редакторе[[35]](#footnote-35). В соответствии с подсказкой, выполните ‘squash’ над коммитом, описывающим README.md, объединив его с предыдущим. Сохраните изменения. Теперь отредактируйте в открывшемся окне редактора сообщение результирующего коммита. Снова сохраните изменения.

Поскольку удаленный репозиторий по-прежнему содержит предыдущее состояние, вы можете как обновить своё состояние в соответствии с удаленным таковым (git pull в каждой ветке; затем автоматический merge, и по завершении вы вернетесь к исходному результату, двум разным коммитам), или наоборот (git push, затем автоматический merge, и удаленный репозиторий принимает перебазированное состояние). Но в нашем случае расхождение истории коммитов, вызванное перебазированием, требует выполнить обновление собственного состояния в соответствии с удаленным, затем слияние, и только тогда обновить состояние удаленного репозитория.

Быстрый вариант, не подходящий для командной разработки (болезненное слияние потребуется всем другим пользователям при очередной попытке синхронизировать состояния с upstream):  
git push --force

Предпочитаемый вариант, содержащий самостоятельное слияние, но другие пользователи обходятся fast-forwarding’ом:  
git pull

[… автоматическое слияние, следуйте подсказкам git status …]  
git push

На этом третья стратегия завершена.

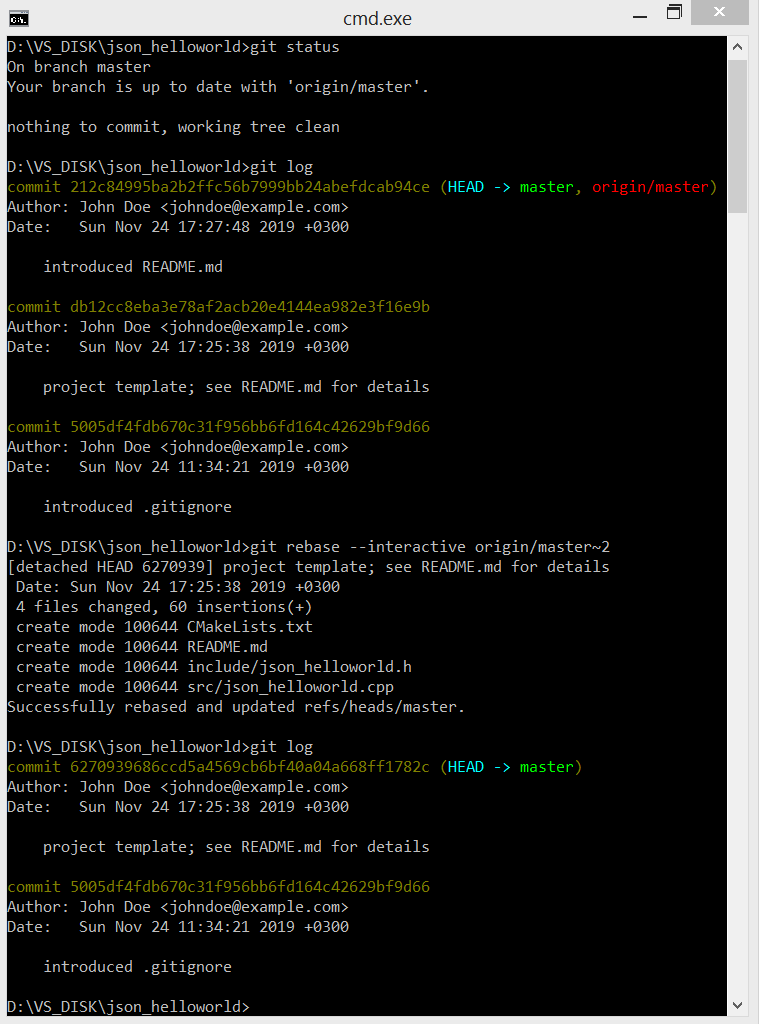


Рисунок 16 - Результат перебазирования. Коммиты объединены.

Вне зависимости от того, какую вы выбрали стратегию для исправления проекта, настройте remote repository в соответствии с инструкциями из третьей и не забудьте выполнить слияние ветки development с веткой master перед продолжением разработки. Теперь каждый раз при старте нового проекта, возможно начать работу с клонирования (git clone) шаблона, сохраненного в удаленном репозитории (не забудьте перенастроить remote, выполняя git remote add .. git remote remove, чтобы удаленный репозиторий шаблона не подвергался обновлениям при разработке другого проекта).

В любой момент во время работы вы можете переключиться в нужное состояние с помощью git checkout [sha-1 коммита или название ветки для использования ее головного коммита] (обновить working tree в соответствии с коммитом) или git reset --hard [sha-1 и т.д.] (переместить курсор на указанный коммит, откатив на него ветку, и обновить working tree в соответствии с коммитом), оперировать изменениями коммита, перемещая их в working tree (git reset [sha-1] делает то же, что git reset --hard [sha-1], но working tree остаётся прежним) и сохраняя в stash (git stash; git stash list; git stash pop). Текущее состояние Local Repository представляет ссылка HEAD, и каждый раз, когда вы выполняете вышеперечисленные команды[[36]](#footnote-36), вы переводите HEAD на соответствующий коммит.

Предположим, что тип шаблонного проекта решено изменить на проект MSBuild (решение Visual Studio). При этом проект CMake в качестве шаблонного уже использует некоторое число людей, которые не только клонируют его репозиторий на старте и начинают работу, но также вносят некоторые изменения в шаблонный проект и обновляют состояние репозитория шаблонного проекта (выполняют push или оставляют заявку на pull request, если у них недостаточно прав на push).

Создайте ветку master\_legacy и уведомите пользователей о необходимости использовать её для дальнейшей работы с шаблонным проектом CMake. Не забудьте сохранить незафиксированное working tree в stash или зафиксировать его очередным commit’ом.

git checkout master // перейти в актуальное состояние master

git branch master\_cmake\_legacy // создать ветку master\_cmake\_legacy с тем же графом коммитов, что и у master

git push --set-upstream origin master\_cmake\_legacy // обновить удаленный репозиторий, создав на нем ветку master\_cmake\_legacy, если она отсутствует, и задать upstream для обновления ее в соответствии с локальной master\_cmake\_legacy при каждом push, и наоборот при каждом pull

git checkout master\_cmake\_legacy // опционально: переместить курсор на master\_cmake\_legacy для работы с наследуемым (legacy) шаблоном CMake

Теперь существует возможность работать со старой версией проекта, не вмешиваясь в текущие ревизии. Далее, наладьте процесс разработки проекта MSBuild:

git checkout master

git log

// найдите первый коммит проекта CMake, и запомните его хэш или его первые 6 символов, если они не пересекаются с другими, например, 6dab23

git checkout 6dab23~1 // обратите внимание на ~1, означающее «родитель». Вы переключаете working tree в состояние **перед** началом работы с проектом CMake, предположительно это коммит .gitignore[[37]](#footnote-37).

git branch development\_msbuild

Наконец, можно работать со старой и новой версией шаблона проекта раздельно, в ветках master\_cmake\_legacy и development\_msbuild. Пользователи шаблона проекта CMake и шаблона проекта MSBuild (даже если это один и тот же человек) теперь работают одновременно в пределах одного репозитория, и когда принято окончательное решение опираться на одну из систем сборки, слияние master с соответствующей веткой позволяет новым пользователям сразу начать работу.

Не забудьте, что перед началом работы над шаблоном проекта, нужно очистить working tree, воспользовавшись stash/commit для фиксирования изменений или reset –hard чтобы их отбросить, затем нужно переключиться на соответствующую ветку командой checkout. Завершив работу над шаблоном, выполните слияние master с одной из веток и обновите состояние удаленного репозитория, выполнив git push из master, затем выполните слияние или создание новой ветки development, предназначенной для разработки выбранного основным шаблона проекта. Второй шаблон по-прежнему находится в своей ветке и готов к работе, если вы решите разрабатывать в условиях строгого требования определенной системы сборки, соответствующей этому шаблону и т.д.

Теперь создание нового проекта (скажем, лабораторной работы) начинается с клонирования репозитория шаблона с Github или пути на диске. Не забудьте удалить remote по умолчанию, чтобы случайно не обновить однажды репозиторий шаблона проектом лабораторной работы.

git remote remove origin

И, возможно, настроить новый remote для совместной разработки.  
git remote add [путь] origin

**Использование расширения Github для Visual Studio**

Убедитесь, что установлено расширение Github (см. **Установка Visual Studio**).  
В окне Team Explorer / Обозревателя решений выберите вкладку Team Explorer, в списке поставщиков служб выберите Github, зарегистрируйтесь (если это еще не сделано) и выполните вход, выбрав «Подключиться…».

Вы можете продолжить работу, если при задании remote repository был указан репозиторий на Github; в противном случае, имеет смысл загрузить в ваше хранилище на Github актуальную копию репозитория (на сайте Github создать репозиторий, затем локально удалить старый и указать новый remote, а затем сделать git push) и начать работу с клонирования удаленного репозитория.

Чтобы клонировать удаленный репозиторий, выполните Файл -> Создать -> Репозиторий… -> в окне Team Explorer «Клонировать…», укажите URL или путь до клонируемого удаленного репозитория, если он расположен локально (не рекомендуется) и путь, по которому будет сохранен local repository, затем выберите клонированный репозиторий в Локальные репозитории Git, и, наконец, откройте его как проект (Файл -> Открыть -> … (в зависимости от системы сборки)).  
Помните, что расширение Github имеет собственный контекст открытого проекта; при каждой смене проекта сделайте это также в окне расширения.

После выбора репозитория, в Team Explorer будет доступна панель Проект с элементами «Изменения», «Ветви», «Синхронизация», «Теги», Параметры», обозначающими известные манипуляции с репозиторием (кроме тегов, они не рассматриваются; Синхронизация – то же, что обновление состояния, упоминающееся ранее). Перейдите в любой из элементов, найдите в пределах окна Team Explorer раскрывающийся список Действия и выберите Просмотреть историю… , чтобы вывести на экран удобное древовидное графическое представление графа переходов выбранной ветви. ПКМ по коммиту, чтобы создать от него ветку или выполнить сброс local repository (reset): --mixed соответствует команде git reset, --hard соответствует git reset –hard.

Продолжайте разработку, в этот раз управляя контролем версий средствами IDE. Фиксируйте состояние в local repository в расширении Github на панели Проект -> Изменения (навигация элементами «Назад» и «Вперед» в левом верхнем углу окна расширения), введя сообщение commit и нажав «Зафиксировать все», или фиксируйте его в stash («Спрятать -> Спрятать всё»), чтобы затем вернуть из stash в working tree («Прятания -> Извлечь -> Извлечь…»). Вы можете отменить любой из сделанных коммитов, перейдя в древовидное представление (см. выше) репозитория и выполнив ПКМ -> Отменить (аналог git revert [sha-1]; создает новый коммит, вносящий изменения, противоположные изменениям отменяемого коммита), выполнить слияние и создание веток на вкладке Ветви, и так далее.

Для обновления состояния локального или удаленного репозитория на вкладке Синхронизация окна расширения Github используйте Вытянуть для входящих фиксаций (аналог git pull, обновление состояния локального репозитория) и Отправить для исходящих (аналог git push).

Используйте Действия -> Открыть командную строку для ручного выполнения некоторых действий, не предусмотренных интерфейсом расширения Github, в том числе git push --force, git rebase --interactive [хэш], и так далее.

Работа завершена.

# Администрирование 1.1 - Настройка переменной окружения PATH

Для быстрого доступа к инструментарию CMake и утилитам в составе Visual C++ включите пути к ним в переменную PATH, используемую Windows для поиска по имени.

Перед включением очередной утилиты в PATH, убедитесь, что она первоначально там отсутствует, и не допускайте двусмысленных имен (например, две версии cmake в разных каталогах, каждый из которых упомянут в PATH).

В интерпретаторе команд запустите whereis [название\_утилиты], и продолжайте работу только после отрицательного результата. Например, если whereis git возвращает одно или несколько упоминаний git, имеет смысл удалить предыдущие экземпляры git из PATH и из системы. Для компиляторов, компоновщиков и систем сборки, выполняющих самостоятельный поиск утилиты перед ее использованием, это критически важно, поскольку расхождение в версиях на разных этапах сборки может привести к несовпадению **ABI** (**Application Binary Interface** - соглашения о вызовах (calling conventions), соглашения о расширениях имен и т.д.), и т.о. ошибке компоновки, неверной компоновке или генерации неверного вызывающего кода, что недопустимо.

Экземпляр интерпретатора команд должен быть запущен **после** изменения PATH для корректной работы.

Найдите каталог исполняемых файлов CMake, в частности, cmake.exe, генерирующего CMake cache при запуске. Как правило, пользователь взаимодействует с ним через «front end», предоставляемый настройками Visual Studio, однако в данном случае потребуется возможность вызвать CMake самостоятельно. cmake.exe может быть расположен в:

C:\Program Files (x86)\Microsoft Visual Studio\ 2019\Community\Common7\IDE\CommonExtensions\Microsoft\CMake\CMake\bin\cmake.exe , или по пути, указанному в журнале вывода Visual Studio в момент генерации нового кэша (воспользуйтесь работой «Внешняя система сборки CMake»).

Настройте переменную окружения PATH (для Windows 8.1: ПКМ по иконке Пуск -> Система -> Дополнительные параметры системы -> Дополнительно -> Переменные среды -> PATH в таблице пользовательских переменных), добавив к путям, перечисленным в ней через точку с запятой, путь до cmake.exe (не включающий в себя сам cmake.exe).

Найдите каталог Git (например, C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\IDE\CommonExtensions\Microsoft\TeamFoundation\Team Explorer\Git\cmd\) и повторите настройку PATH, добавив каталог Git.

Найдите каталог MSBuild (например, C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\MSBuild\Current\Bin), и проделайте вышеперечисленные манипуляции, добавив также каталог MSBuild.

Также возможна настройка PATH в пределах текущей сессии интерпретатора[[38]](#footnote-38). Выполните:

setx PATH "% PATH %;C:\new\path\directory"

чтобы добавить в PATH новое значение, работа с которым возможна только до перезапуска интерпретатора.

Настройка PATH для набора рассматриваемых в рекомендациях утилит завершена.

# Администрирование 1.2 – Генерация сценария нижележащей системы сборки с помощью CMake и ручная сборка по сценарию нижележащей системы сборки

Убедитесь, что PATH содержит путь к CMake. Запустите интерпретатор команд (cmd.exe, powershell.exe или аналог).

Запустите CMake в режиме вывода справки.

cmake --help

Изучите список CMake generators, представленных (для версии CMake 3.15) списком Generators в выводе cmake --help.

Перейдите в каталог проекта CMake (в нем содержится CMakeLists.txt).

cd D:\vsProjects\IntroductionToCMake\  
D:

Находясь в каталоге проекта CMake, создайте новый каталог для результата сборки (этот каталог можно в любой момент удалить и начать сборку сначала, если предыдущий результат по каким-то причинам не устроил). Перейдите в каталог сборки.

mkdir build

cd build

Запустите CMake в текущем каталоге (каталоге сборки), указав предпочитаемый генератор и указав каталог на уровень выше для входного сценария (CMakeLists.txt). Предположим, что цель генерации – решение Visual Studio 2019.

cmake -G “Visual Studio 16 2019” ../

Генерация завершена. Откройте каталог сборки в файловом менеджере, запустите сгенерированное решение, выберите нужный проект и продолжайте работу со сгенерированным решением Visual Studio.

Для продолжения работы вне среды разработки вызовите нижележащую систему сборки (для генератора Visual Studio 16 2019 это MSBuild, для Unix Makefiles это make, для Ninja это ninja, и т.д.) и передайте ей результат работы CMake. Убедитесь, что каталог нижележащей системы сборки установлен в PATH для быстрого доступа.

msbuild [название\_решения\_для\_сборки].sln // в случае системы MSBuild

msbuild [название\_конкретного\_проекта\_для\_сборки].\*proj

msbuild INSTALL.\*proj // для установки

make [название\_конкретной\_цели\_сборки\_например\_библиотеки] // система make

После успешной сборки в каталоге решения появится каталог сборки (для MSBuild это “out”, если не указано иного), в котором находятся результирующие исполняемые модули, отладочные символы и т.д.

Ручная сборка завершена.

# Администрирование 1.3 – Переименование вложенного проекта Visual Studio

Для переименования проекта Visual Studio (проекта в системе MSBuild), выполните следующие действия:

* Включенный в решение проект переименуйте в Обозревателе решений (ПКМ -> Переименовать);
* Удалите проект из решения (ПКМ -> Удалить);
* Вручную переименуйте каталог с проектом в файловом менеджере;
* В сценарии проекта (\*.\*proj в корне проекта) вручную измените упоминания старого названия проекта на новое; обратите внимание на теги RootNamespace и ProjectName в <PropertyGroup Label="Globals">;
* Добавьте переименованный проект в решение (ПКМ по решению в Обозревателе решений -> Добавить -> Существующий проект… -> укажите отредактированный сценарий проекта);
* Перенастройте зависимости в соответствии с «Работа с несколькими проектами Visual Studio».

Переименование завершено.

# Администрирование 1.4 – Работа с несколькими проектами Visual Studio

Предположим, что в решении Solution существует проект Base (результирующий исполняемый модуль – статическая библиотека(.lib) или динамическая библиотека (.dll)), не имеющий зависимостей, и проект Dependent (результирующий модуль – приложение (.exe) ) использующий заголовочные файлы Base и вызывающий методы Base, реализованные в Base.lib или Base.dll.

Для работы с Base в проекте Dependent необходимо:

* Назначить приоритет сборки: Base должен собираться до Dependent, что реализуется ссылкой Dependent на Base (ПКМ по элементу Ссылки в дереве Dependent в Обозревателе решений -> Добавить ссылку… -> отметить Base);
* Указать препроцессору при сборке Dependent, который включает (#include “some\_base\_header.h”) заголовочные файлы Base, каталоги для их поиска, реализуется указанием в свойствах Dependent каталога заголовков Base (ПКМ по имени Dependent в Обозревателе решений -> Свойства -> C / C++ -> Общие -> Дополнительные каталоги включаемых файлов (Additional Include Directories) -> …(следует добавить каталог, в котором находятся заголовочные файлы проекта Base));
* Указать компоновщику при компоновке Dependent, который использует сущности из Base, каталог поиска библиотеки Base.lib и название Base.lib:
  + ПКМ по имени Dependent в Обозревателе решений -> Свойства -> C / C++ -> Компоновщик -> Общие -> Дополнительные каталоги библиотек (Additional Library Directories) -> … (следует добавить каталог, в котором после сборки находится исполняемый модуль Base, например base/out/x64/debug);
  + ПКМ по имени Dependent в Обозревателе решений -> Свойства -> C / C++ -> Компоновщик -> Ввод -> Дополнительные зависимости -> … (следует дописать к имеющимся зависимостям имя библиотеки, полученной после успешной сборки Base, предположительно Base.lib[[39]](#footnote-39)).
* (только для случая, когда Base – динамическая библиотека; только для случая) Назначить post-build event - действие, буквально событие, выполняющееся после успешной сборки - копирование результирующего модуля проекта Base (Base.dll) в каталог сборки Dependent для успешной dynamic linking. Свойства проекта Dependent -> Свойства конфигурации -> События сборки -> Событие после сборки -> Командная строка -> (добавить xcopy /y /d "..\..\path\_to\_Base\_project\$(IntDir)Base.dll" "$(OutDir)", где $(IntDir) и $(OutDir) – макросы MSBuild, обозначающие промежуточный каталог сборки и каталог результата сборки (последний содержит исполняемые модули, в т.ч. приложения .exe).

Убедитесь, что как Base, так и Dependent используют одну разновидность библиотеки времени выполнения[[40]](#footnote-40), в каждом проекте установив в одно значение её разновидность (ПКМ по имени в Обозревателе решений -> Свойства -> C / C++ -> Создание кода -> Библиотека времени выполнения).

Несоблюдение этого правила приведет к ошибкам компоновки сущностей стандартной библиотеки, определения которых, очевидно, при попытке скомпоновать Dependent (имеющий свою библиотеку времени выполнения) с Base (имеющий свою, отличную от таковой Dependent) будут встречаться компоновщику дважды (по одному из каждой библиотеки), делая однозначный symbol resolution невозможным.

# Администрирование 1.5 – Использование собранных ранее исполняемых модулей в решениях Visual Studio

Повторите шаги, описанные в «Работа с несколькими проектами Visual Studio», но без ссылок: так как проект уже собран ранее (см. **«Администрирование 1.2 – Генерация сценария нижележащей системы сборки с помощью CMake и ручная сборка по сценарию нижележащей системы сборки»**), указание ссылки на проект избыточно, поскольку уже собранные модули не нуждаются в опережающей сборке.

# Администрирование 1.6 – Использование MSBuild вместо Ninja в качестве нижележащей системы сборки при работе с проектами CMake в Visual Studio

Сгенерируйте кэш для любого сценария CMakeLists.txt, дописав ко сценарию:  
message(${CMAKE\_MAKE\_PROGRAM})

и проанализируйте вывод в окне Вывод при генерации кэша. Убедившись, что текущий проект CMake использует Ninja (выведенный путь совпадает с путем до Ninja), переходите к дальнейшей работе.

Отредактируйте CMakeSettings.json вручную, перейдя в Проект -> Параметры CMake для [название проекта] -> ЛКМ по ссылке CMakeSettings.json, расположенной под заголовком Параметры CMake.

CMakeSettings.json по умолчанию декорирует вызов Ninja аргументом -v (от --verbose, «многословно»), который, наиболее вероятно, не совместим с аргументами MSBuild. Очистите поле «buildCommandArgs».

Теперь изучите:

<https://docs.microsoft.com/en-us/cpp/build/cmakesettings-reference?view=vs-2019>

и обратите внимание на следующую деталь: при том, что в **«Администрирование 1.2 – Генерация сценария нижележащей системы сборки с помощью CMake и ручная сборка по сценарию нижележащей системы сборки»** вызываемая вручную система CMake с генератором Visual Studio 16 2019 требует отдельно указать платформу с помощью внешнего ключа -A, в CMakeSettings.json возможно сделать это, как для генераторов для предыдущих версий Visual Studio, добавив к названию генератора платформу (Win64 для Windows x64, ARM для Windows ARM, ничего не добавлять для Windows x86).

Итак, укажите для «generator» нужный генератор, предположим, что это Visual Studio 16 2019 Win64.

Соберите простой проект, проанализировав вывод при генерации кэша CMake и сборке. Убедитесь, что компилятор C++ определен правильно и это компилятор в составе Visual C++.

В противном случае, изучите:  
<https://developercommunity.visualstudio.com/content/problem/114469/cmake-integration-with-ninja-generator-fails-to-fi.html>

и настройте наследуемое окружение (inherit environments) в соответствии с рекомендациями. Убедитесь, что используете соответствующее платформе наследуемое окружение, например msvc\_x64\_x64, и окружение-декоратор, вводящее приоритетные пути Visual C++.

Очистите решение (можно вручную, удалив каталог продуктов сборки по умолчанию out), удалите предыдущий кэш CMake, создайте новый кэш CMake и соберите проект. Убедитесь, что продукт генерации кэша – решение Visual Studio без промежуточного сценария Ninja в ./out, что CMAKE\_MAKE\_PROGRAM указывает на MSBuild.

Работа завершена.

# Интеграция 1.1 - SFML (библиотека мультимедиа) – использование в проекте Visual C++

Ознакомьтесь с информацией о загрузке на сайте sfml-dev.org. На данный момент (осень 2019) коллектив, сопровождающий SFML (SFML maintainers) не предоставляет готовую сборку для Visual Studio 2019. Попытка использовать сборку для другой версии Visual Studio (см. Использование собранных ранее проектов в проектах Visual Studio) может привести к несовместимости ABI и возможному неопределенному поведению во время работы программы.

Ознакомившись с информацией о расположении репозитория, клонируйте его для локального доступа (убедитесь, что настроен PATH, см. соответствующий раздел), из каталога, в котором вы хотите разместить каталог с локальным репозиторием:

git clone [путь\_до\_SFML] // по состоянию на осень 2019 репозиторий SFML на Github - <https://github.com/SFML/SFML/>

Далее, т.к. CMakeLists.txt присутствует, соберите SFML вручную с использованием любой удобной нижележащей системы сборки (см. «Генерация сценария нижележащей системы сборки с помощью CMake.

Ручная сборка с использованием нижележащей системы сборки»), например MSBuild. Убедитесь, что при генерации CMake использует тот же toolset, что и среда для работы с зависимым проектом.

mkdir build

cd build

cmake -G “Visual Studio 16 2019” -A x64 ../ -DCMAKE\_INSTALL\_PREFIX=”D:\installed” -DCMAKE\_BUILD\_TYPE=”Release”

Следует рассмотреть последнюю команду подробнее. Указывая CMake generator, мы также указываем дополнительный параметр платформы (-A x64), при том что в generator для предыдущих версий Visual Studio проблема разрядности решена введением двух разных generator’ов (см. cmake –help, аналогичная команда для предыдущей версий могла бы выглядеть так: «Visual Studio 15 2017 Win64»). «../» указывает на каталог уровнем выше – каталог CMake project, где находится CMakeLists.txt.

Далее, все параметры CMake начинаются с префикса CMAKE\_\*, а их передача в CMake в качестве параметров команды возможна с ключом --D[команда]=[значение], например -DCMAKE\_BUILD\_TYPE=”Debug” означает, что параметр CMAKE\_BUILD\_TYPE равен Debug.

Почему используется значение Release? Дело в том, что отладка SFML представляется нецелесообразной – ошибка скорее в пользовательском коде, нежели в модулях SFML, и исполняемый модуль собирается в режиме наилучшего быстродействия и объёма итогового модуля, раз нет необходимости в обратном (см. Теоретические основы).

На параметре CMAKE\_INSTALL\_PREFIX следует остановиться и рассказать подробнее. Установка программного обеспечения может включать в себя копирование исполняемых модулей, заголовочных файлов, assemblies (см. Теоретические основы), сценариев сборки в различные каталоги, при том, что каталог по умолчанию обычно означает, что установка носит персистентный характер.

Например, установка исполняемого модуля в /usr/bin для \*nix-систем или установка в %systemroot%, %programfiles% и регистрация в системном реестре для Windows означает, что каждый модуль программы будет доступен всем компонентам и программам в системе, и будет доступен до тех пор, пока система работоспособна и программа не удалена, и если до ее удаления произошла установка зависящей от её модулей программы, то удаление повлечет её неработоспособность.

CMAKE\_INSTALL\_PREFIX позволяет задать изолированную среду для проектов Visual Studio и проектов CMake.

Итак, сценарий MSBuild сгенерирован. Выведите на экран содержимое build, выполнив в интерпретаторе из каталога build:

dir[[41]](#footnote-41)

и убедитесь в наличии INSTALL.vcxproj – проекта, целью сборки которого является установка SFML в CMAKE\_INSTALL\_PREFIX.

Если собрать проект INSTALL (и т.о. установить SFML в CMAKE\_INSTALL\_PREFIX) прямо сейчас, то тип сборки (Debug или Release) может не совпадать с требуемым, и его потребуется указать нижележащей системе сборки (MSBuild) явно.

Выполните:

msbuild INSTALL.vcxproj

пройдите в CMAKE\_INSTALL\_PREFIX, и убедитесь, что установлены модули, соответствующие типу сборки (для Debug они будут иметь постфикс –d, для Release нет). Расширение имён выполняет SFML в соответствии со своим сценарием сборки, не всегда возможно будет установить корректность типа сборки таким образом.[[42]](#footnote-42)

В противном случае, может потребоваться передача типа сборки в нижележащую систему сборки вручную, для MSBuild это:  
msbuild INSTALL.vcxproj /p:Configuration=Release

Или, с платформой:  
msbuild INSTALL.vcxproj /p:Configuration=Release /p:Platform=x64

Обратите внимание, что в результате сборки и установки даже теперь не все результирующие модули имеют тот же тип сборки, что указан явно при генерации CMake / запуске результата MSBuild. Некоторые из них (кодеки формата OGG, формата Vorbis и т.д.) являются внутренними зависимостями SFML и собираются строго в Release или распространяются, будучи предварительно собранными для разных платформ. Эти зависимости распространяются вместе с SFML (см. каталог extlibs репозитория).

Также обратите внимание, что при любом изменении параметров сборки (список входных заголовочных файлов и файлов исходного кода, тип сборки (debug / release), платформа) CMake cache должен быть перегенерирован (cmake --G …), и MSBuild должна быть запущена заново для нового сгенерированного сценария. Желательно при этом удалить каталог продуктов сборки (./build) полностью и начать генерацию и сборку сначала.

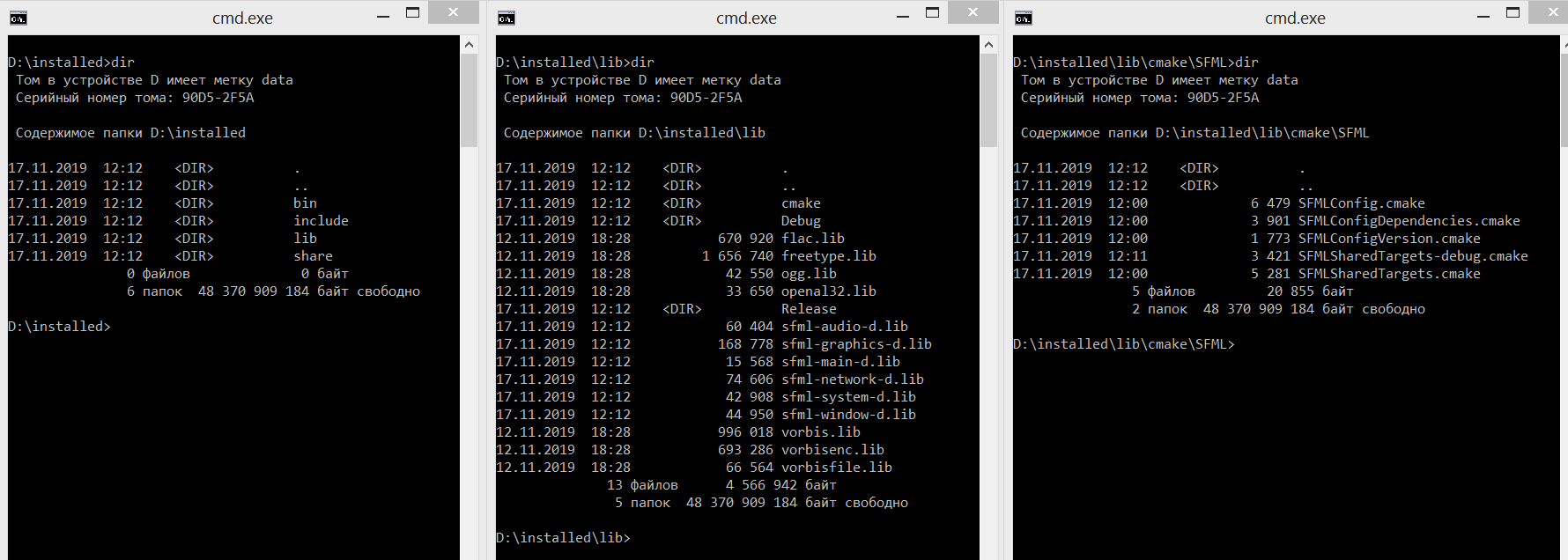


Рисунок 17 - Пример результата сборки и установки для типа сборки Debug.

Выполните **«Администрирование 1.5 – Использование собранных ранее исполняемых модулей в решениях Visual Studio»** для использования установленных заголовочных файлов SFML и компоновки с установленными модулями SFML.

Включите в проект файлы исходного кода приложения SFML, воспользовавшись примерами из репозитория или сайта SFML. Интеграция завершена.

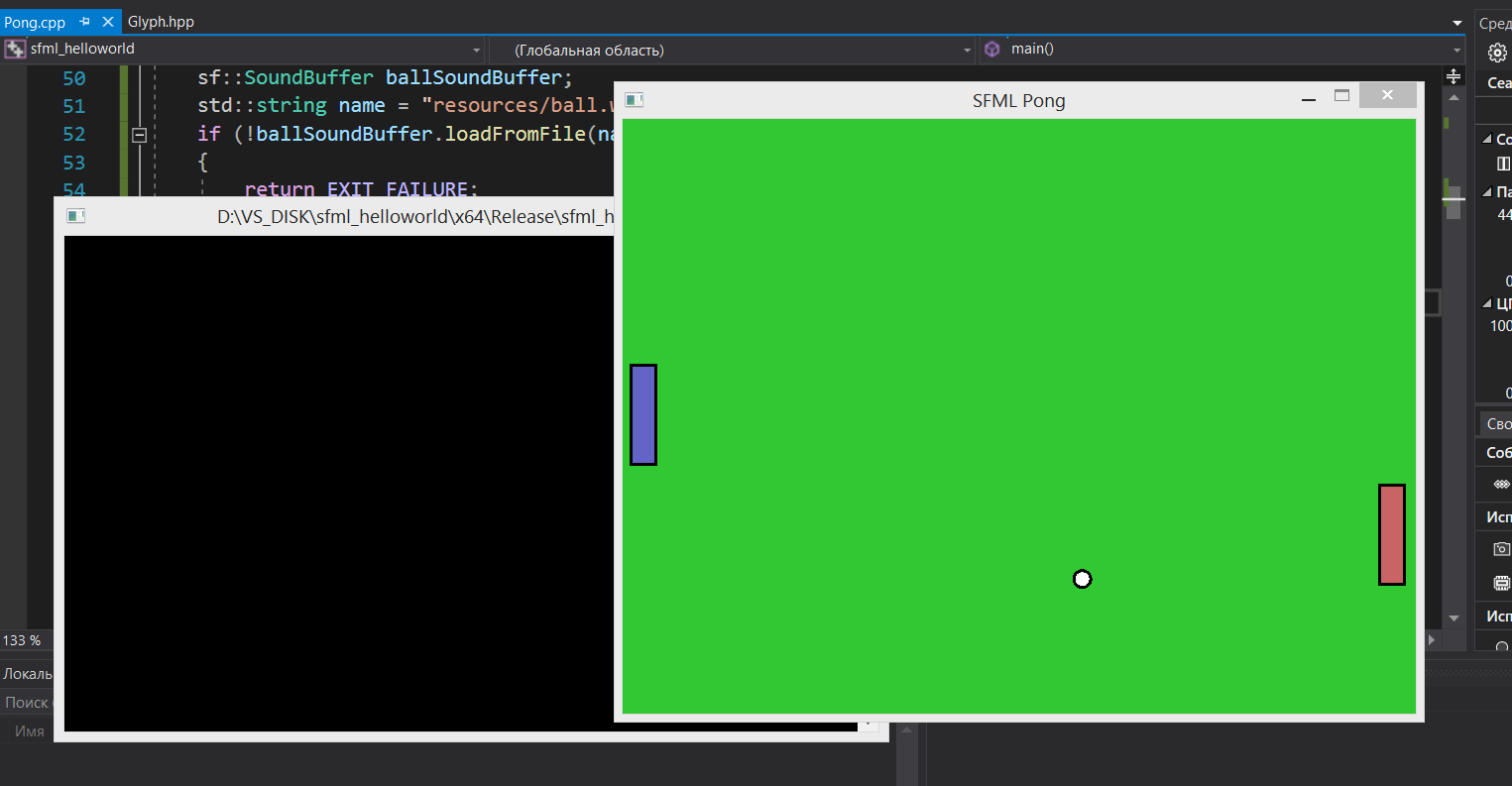


Рисунок 18 - Пример работы SFML Pong из SFML/examples/pong

# Интеграция 1.2 - nlohmann\_json – библиотека формата JSON для современного C++ - использование в проекте CMake

Начните с клонирования репозитория <https://github.com/nlohmann/json>

Выполните установку nlohmann\_json в соответствии с примером для SFML.

Убедитесь, что выбраны правильная платформа, CMake generator, debug/release, install prefix (в предыдущем примере D:\installed).

Создайте проект CMake, на этот раз разместив решение и проект в одном каталоге. Приведите структуру проекта в соответствие таковой из предыдущей работы, посвященной CMake (за исключением того, что сценарий CMake теперь один).

Теперь обратите внимание на содержимое [install prefix]/lib/cmake. Здесь находятся файлы **CMake packages,** обеспечивающие информацией о CMake targets, установленных ранее. Скажем, после успешной интеграции SFML в пользовательский проект (включающей в себя установку SFML в install prefix), здесь будут расположены SFML/SFMLConfig.cmake (каталог проекта / [название target]Config.cmake) и некоторые другие сценарии. Для устанавливаемого проекта каждый вызов install в сценарии CMakeLists.txt определяет новые сущности для установки, и, соответственно, для импорта.

Вызов find\_package(..) позволяет получить доступ к импортируемым CMake target’ам, в том числе к их include directories, объявленным с модификатором PUBLIC или INTERFACE, а последующий вызов target\_link\_libraries(..) задает компоновку с импортируемыми target’ами в ходе сборки.

Для генерации CMake cache на основе имеющихся данных (и далее сборки, включая компоновку с nlohmann\_json, и установки, если это библиотека для использования в других проектах) следует декорировать вызов CMake, происходящий в Visual Studio при генерации кэша, указанием параметра CMake – CMAKE\_INSTALL\_PREFIX – передаваемого в CMake самостоятельно при ручной сборке, а теперь требующего автоматической передачи средствами Visual Studio или ручного редактирования CMakeSettings.json – документа, описывающего декорацию вызова CMake. Также следует явно указать путь к файлам CMake packages, задав CMAKE\_MODULE\_PATH.

Итак, на панели Стандартная в списке конфигураций сборки / объектов сборки выберите Управление конфигурациями… , в открывшейся вкладке CMakeSettings.json прибавьте к значению поля Аргументы команд CMake строку «-DCMAKE\_MODULE\_PATH=[путь\_до\_installed/…/cmake] -DCMAKE\_INSTALL\_PREFIX=[путь\_до\_installed]», убедитесь, что в таблице Переменные и кэш CMake сняты галочки Сохранение в JSON, и нажмите Сохранить и создать кэш CMake для загрузки переменных[[43]](#footnote-43).

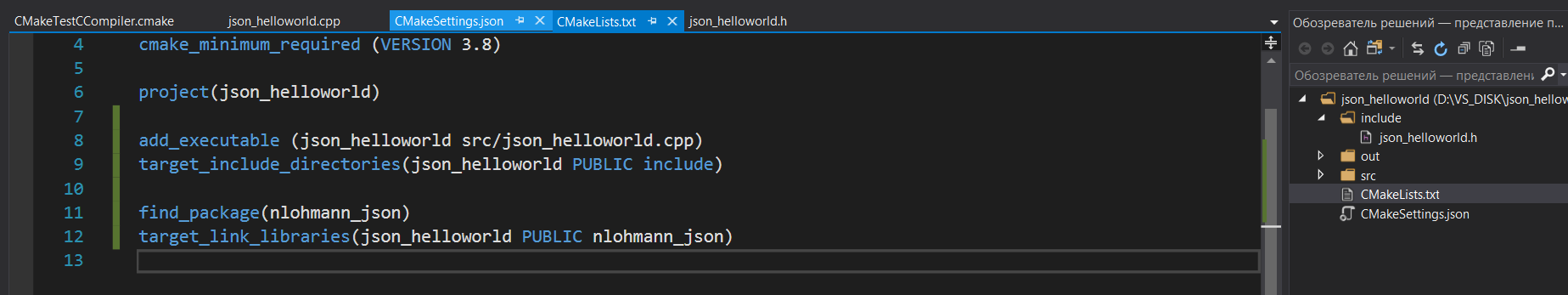


Рисунок 19 - Пример CMakeLists.txt для использования nlohmann\_json

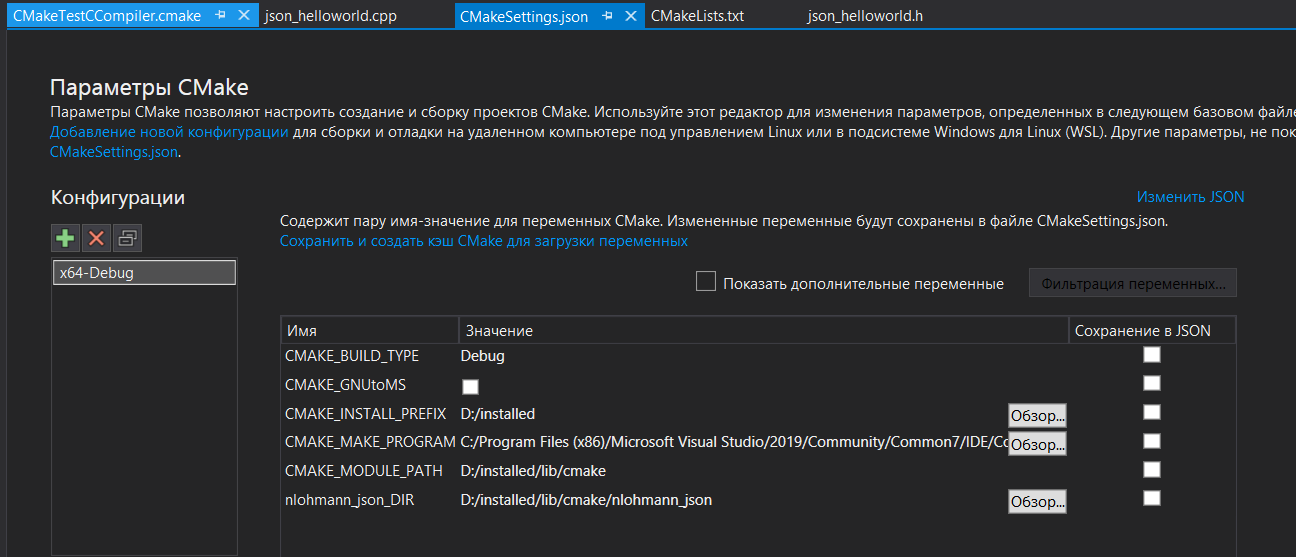


Рисунок 20 - Пример содержимого таблицы переменных CMake после создания кэша в соответствии с CMakeLists.txt

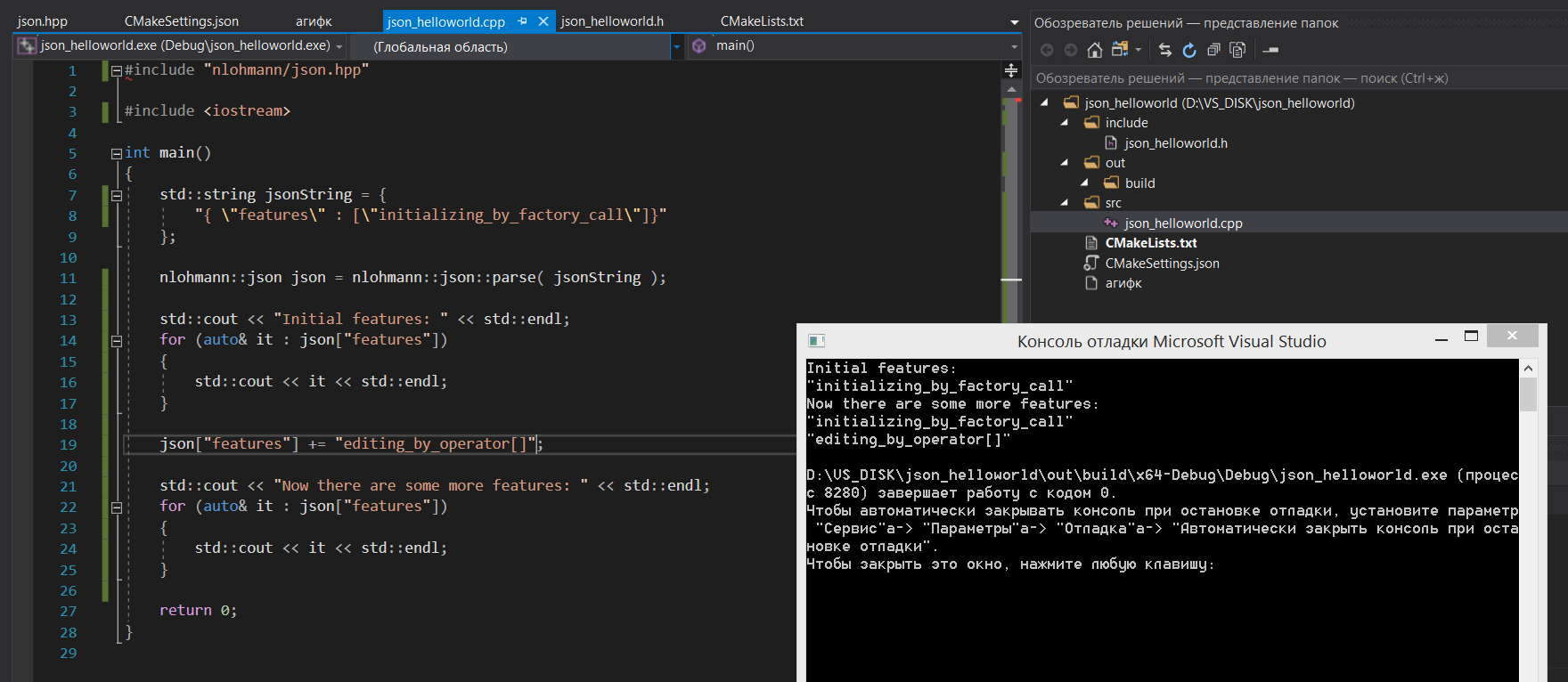


Рисунок 21 - Инициализация nlohmann::json вызовом фабричного метода (factory method) nlohmann::json::parse(..) и минимальный пример работы

После явного задания install prefix и module path, определения nlohmann\_json средствами процедуры find\_package(..) и компоновки средствами target\_link\_libraries(..) запустите любой пример, использующий возможности nlohmann\_json[[44]](#footnote-44).

Интеграция завершена.

# Заключение

В рекомендациях рассмотрена установка среды разработки Visual Studio, настройка её пользовательского интерфейса и работа с расширением Github.

В разделе о теоретических основах даны сведения о трансляции, компиляции и интерпретации, ходе сборки, различиях разных видов компоновки, реализации сборки, в том числе компиляции и компоновки кода на C++.

Задан coding standard для дальнейшего кода и определены шаблонные работы из числа ознакомительных работ. Кроме того, предоставлена типовая конфигурация для проектов различных систем сборки, в том числе примеры сценариев CMake и настроек MSBuild (ручное редактирование сценариев MSBuild не рассматривалось).

Ознакомительные работы включают в себя создание проектов CMake и MSBuild, их интеграцию с библиотекой тестирования и введение контроля версий. Работы по администрированию и интеграции включают в себя настройку PATH, работу с нижележащей для CMake системой сборки, ручную сборку и установку, работу с зависимостями MSBuild (в том числе задание последовательности сборки, задание модулей компоновки, каталогов этих модулей; задание каталогов включаемых файлов, задание библиотеки времени выполнения, подсистемы компоновщика, типа сборки (release/debug), компоновки с различными модулями – статическими и динамическими), и CMake, необходимую для компоновки нескольких исполняемых модулей на каждом этапе сборки и, в конечном итоге, выполнения, в программу. Рассмотрены примеры ручной сборки, установки и интеграции с библиотеками с открытым исходным кодом.

Таким образом, освещены вопросы, возникающие в процессе разработки, в том числе требующие навыков администрирования и build engineering.

1. Идентификатор — это имя функции, переменной, класса и т.д., по которому программист обращается к функции и т.д. в коде. [↑](#footnote-ref-1)
2. Возможен static linking не только продуктов текущей компиляции в ходе текущей сборки, но и продуктов компиляции, выполненной заранее, при условии имеющихся заголовочных файлов, описывающих доступ к таковым (см. упоминания «статическая библиотека» и «static linking» далее); [↑](#footnote-ref-2)
3. Для некоторых разновидностей external linkage возможен linking сущности с использующим её кодом, который не только находится в другой единице трансляции или к тому же находится в другом исполняемом модуле и принужден таким образом к runtime symbol resolution для её использования, но даже написан при этом на другом языке программирования (см. раздел 6.5 Стандарта «Program and linkage»). [↑](#footnote-ref-3)
4. С черновиком стандарта C++ (**C++ standard working draft**) от 27.11.2017 можно ознакомиться [здесь](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf).   
   В дальнейшем эту версию будем называть Стандартом. Директивы препроцессора описывает глава 19 Стандарта (Section 19: Preprocessing Directives). [↑](#footnote-ref-4)
5. Ход трансляции описывается разделом 2 главы 5 Стандарта (далее подобные ссылки кратко называются «раздел 5.2 Стандарта»). [↑](#footnote-ref-5)
6. Как правило, представляет собой машинный код вместе с некоторой дополнительной информацией. Не имеет отношения к промежуточному коду, упоминаемому далее. Важным является лишь список имён функций, переменных, классов и шаблонов классов и некоторых других сущностей языка C++ (всё вместе – т.н. символов, о них далее), присутствующих в этом представлении, и на данный момент не определённых. [↑](#footnote-ref-6)
7. Если код работает с аппаратным обеспечением – информация об этом обеспечении, если код работает с операционной системой – информация об этой системе, если код работает с неким программным модулем, примеры которых будут приведены далее – информация об этом модуле и способе компоновки с ним (о компоновке далее). [↑](#footnote-ref-7)
8. Таблицы символов являются популярной, но не обязательной реализацией хранения данных о символах, а Стандарт не регламентирует это хранение вовсе (оно относится ко внутреннему устройству компиляторов, компоновщиков, loader’ов и соглашениям между ними). Одно из таких соглашений для современных систем обязывает включать в образ программы данные, необходимые для runtime symbol resolution, для Portable Executable это ImportTable, ImportAddressTable и т.д. [↑](#footnote-ref-8)
9. В целях производительности результирующий исполняемый модуль с этими символами не рекомендуется использовать в иных целях, кроме отладки – как правило, в соответствующем режиме компиляции он лишен всех оптимизаций, меняющих сопоставление выполняемого машинного кода последовательному набору строк исходного кода, и имеет в своем платформозависимом представлении избыточное число символов (включая debugging symbols). [↑](#footnote-ref-9)
10. Например, пользователь библиотеки, подключая её заголовочный файл, в любом случае не будет иметь доступа к автоматическим переменным в теле метода, реализация которого содержится в файле исходного кода (.cpp), и информация об этих переменных нужна ему только при отладке (если он заниматься отладкой чужого продукта). [↑](#footnote-ref-10)
11. По историческим причинам. Когда-то было невозможным ввиду недостатка физической памяти транслировать весь исходный код целиком. Файлы исходного кода обрабатывались по одному, а полученный результат фиксировался в виде объектного файла, сопоставленного с конкретным файлом исходного кода. Существует несколько способов декомпозировать исходный код при компиляции, и этот способ рассматривается в данных рекомендациях и применяется в используемых toolchain. [↑](#footnote-ref-11)
12. Как правило, решается соглашением между компоновщиком и программным загрузчиком (**loader**), входящим в состав операционной системы, устанавливающим точку входа (**entry point**) – расположение в исполняемом модуле последовательности машинных команд, которой будет передано управление по окончании процесса загрузки. [↑](#footnote-ref-12)
13. Среди которых не было директивы #error (см. раздел 19.5 Стандарта). [↑](#footnote-ref-13)
14. Только для текущей версии Visual Studio (Community 2019). Для неё же возможно следующее расположение этих файлов: C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\[полный номер версии Visual Studio]\bin\Hostx64\x64 [↑](#footnote-ref-14)
15. Существует разновидность реализации этой системы, когда код, скомпилированный первым транслятором, не только непосредственно выполняется вторым, но также перекомпилируется с целью оптимизации прямо во время выполнения программы.

    Тогда первый транслятор называется **Ahead-Of-Time (AOT) Compiler (AOT-компилятор)**, а компонент, выполняющий повторную компиляцию некоторых фрагментов промежуточного кода перед их выполнением во втором трансляторе – **Just-In-Time (JIT) Compiler (JIT-компилятор).**

    В вырожденном случае, JIT-компиляция полностью заменяет собой интерпретацию (а JIT-компилятор заменяет собой второй транслятор), когда JIT-компиляция происходит безусловно (для всего выполняемого кода) и её продуктом является не оптимизированный промежуточный код, а код физической машины, выполняемый непосредственно процессором, без участия интерпретатора. [↑](#footnote-ref-15)
16. См. Стандарт 6.8.3.1 (Basic concepts -> Program execution -> Start and termination -> main function) [↑](#footnote-ref-16)
17. Существуют программы (назовем их драйверами, хотя это и не совсем корректно), предоставляющие пользователю интерфейс для указания модуля, не имеющего entry point, и конкретных сущностей из этого модуля для выполнения. В состав Windows входит являющаяся таким драйвером rundll32. [↑](#footnote-ref-17)
18. У CMakeLists.txt есть приблизительный аналог со стороны MSBuild, в случае когда последняя используется проектами Visual Studio (а так происходит почти всегда) – это файл проекта .\*proj, в нашем случае .vcxproj. Причем CMakeLists.txt берет на себя также функции файла решения (.sln), позволяя включать в себя дочерние CMakeLists.txt (используя процедуру add\_subdirectory(..)).

    Там, где в решении Visual Studio существовал бы один .sln, ссылающийся на несколько .\*proj, по одному на каждый проект, в проекте CMake существовал бы корневой CMakeLists.txt, ссылающийся на проекты с помощью add\_subdirectory(..), и CMakeLists.txt для вложенных проектов, объявляющие каждый свой проект и свой набор target’ов, которых, в отличие от проектов Visual Studio, может быть больше одного на проект. [↑](#footnote-ref-18)
19. Сценарий утилиты make – низкоуровневой системы сборки, получившей распространение в 1980-х и являющейся fallback-вариантом (для отката на нее в случае отказа сложных сборочных средств вроде CMake), поддерживаемым в том числе Windows и множеством систем Linux, а для последних как правило являющейся основной системой сборки. CMake поддерживает генерацию makefile для работы с этими платформами, об этом далее. [↑](#footnote-ref-19)
20. Аналогичное действие могло бы иметь смысл в проектах Visual Studio (напомним, что имеются ввиду проекты MSBuild) при ручном редактировании «сценариев» MSBuild. Однако Visual Studio предоставляет достаточные возможности для управления сборкой MSBuild (реализованные практически полностью в свойствах проекта), и ручная работа со сценариями таковой как правило не требуется.

    Для поддержания целостности пары «сценарий – проект» (и исключения аномалий обновления, когда сценарий сборки уже изменился, но при выполнении сборки процесс идет по-старому) редактирование сценария MSBuild одновременно с работой с проектом средствами Visual Studio и вовсе запрещено.

    Редактировать сценарий MSBuild по-прежнему возможно, выгрузив связанный с ним проект (ПКМ по имени проекта (не решения!) в Обозревателе решений -> Выгрузить проект), и, после внесения необходимых изменений (ПКМ по имени проекта -> Изменить \*.\*proj) загрузив снова (ПКМ по имени проекта -> Перезагрузить проект) для обновления списка целей сборки и т.д, хотя и не рекомендуется без четкого понимания системы MSBuild и не имеет смысла для повседневной разработки. [↑](#footnote-ref-20)
21. Или SHARED, если требуется исполняемый модуль разделяемого типа – в нашем случае это динамическая библиотека (.dll). [↑](#footnote-ref-21)
22. Google Mock распространяется вместе с Google Test, используется для тестирования с применением fixture classes и де-факто является третьим используемым target (“gmock”). [↑](#footnote-ref-22)
23. Подробнее: <https://github.com/google/googletest/blob/master/googletest/docs/primer.md#writing-the-main-function> [↑](#footnote-ref-23)
24. Это изменение имеет следующее значение: сценарий описывает не обобщенные процессы, связанные с проектами CMake, а настройку googletest, и притом в ходе сборки этот сценарий является входным файлом; таким образом, название googletest.cmake.in является более осмысленным. [↑](#footnote-ref-24)
25. Строго говоря, как содержимое staging area, так и содержимое local repository являются также частью working tree, поскольку описывающие их файлы находятся в каталоге проекта. Но мы будем говорить «находится в working tree» только тогда, когда файл или каталог не отмечен в staging area или не зафиксирован в local repository. [↑](#footnote-ref-25)
26. Ближайший аналог такому применению echo – утилита touch из окружения GNU. Несмотря на то, что touch может присутствовать на машине, например, в составе окружения Cygwin для Windows, в общем случае надеяться на это не приходится. [↑](#footnote-ref-26)
27. Для проекта CMake, с которым пользователь работает в Visual Studio. Для проекта MSBuild потребуется учесть также пользовательские настройки в .suo – они тоже не должны распространяться с репозиторием. [↑](#footnote-ref-27)
28. См. замечание о Working Tree в «О сущностях Git …». Имеется ввиду, что разница в файлах между Working Tree и Local Repository отсутствует. [↑](#footnote-ref-28)
29. Перебазирование (rebase) – это откат ветки на некое состояние, заведомо за проблемными коммитами, а затем их последовательное применение заново с некоторыми заданными пользователем изменениями. [↑](#footnote-ref-29)
30. Поскольку локальная ветка master в их случае больше не является подграфом удаленной ветки master; существует коммит, который присутствует в их master (скажем, они клонировали удаленный репозиторий до перебазирования), но отсутствует в удаленном master (вместо него результат перебазирования). [↑](#footnote-ref-30)
31. Если с вами это случится (git merge завершается неудачей), следуйте инструкциям в git status, чтобы заново добавить файлы в staging area (после ручного разрешения конфликтов), и завершить коммит. [↑](#footnote-ref-31)
32. При том, что в настоящее время master и development идентичны (если это не так, значит, в какой-то момент они не были merged («слиты»), то есть не было выполнено слияние), возможно перебазирование относительно development, но в общем случае это не всегда так. [↑](#footnote-ref-32)
33. Репозиторий, из которого во время будничной разработки каждый пользователь получает актуальные данные. С точки зрения архитектуры git, все репозитории равноправны и в качестве upstream может быть назначен любой из них. При командной разработке репозиторий на GIthub или корпоративном сервере Git предпочитаем в качестве upstream. [↑](#footnote-ref-33)
34. При том что development также указывает на последний коммит ветки, то есть development~0, явно указать коммит является значимым в образовательных целях. Зафиксировав после README.md, скажем, описание лицензии в LICENSE, вы по-прежнему могли бы выполнить cherry-picking, указав нужный коммит по индексу относительно ветки (development~1) или по его SHA-1 хэшу (см. git log ветки, в которой содержится коммит). [↑](#footnote-ref-34)
35. По умолчанию – vim, ознакомьтесь с опциями git config для интеграции с удобным редактором или руководством по минимальному обращению с vim (рекомендуется). [↑](#footnote-ref-35)
36. Кроме git stash, разумеется, поскольку stash не влияет на состояние local repository, а только на состояние staging area во время вставки в stash, и на состояние working tree при извлечении из stash. [↑](#footnote-ref-36)
37. Важное замечание: когда вы выполняете git checkout [sha-1], вы переводите ссылку HEAD на позицию, не соответствующую ни одной ссылке на ветку, и если вы сделаете commit, находясь в этом состоянии (оно называется detached HEAD, от detached – отделенный, оторванный), таким образом переместив HEAD на этот новый коммит, вместе с HEAD вы не переместите автоматически ссылку ни одной из веток репозитория, и HEAD будет единственной ссылкой, указывающей на ваш новый коммит. Не делайте коммитов из состояния detached HEAD (можно проверить в git status, см. сообщение «HEAD detached at [sha-1]»), не ассоциировав текущее положение HEAD с новой веткой (находясь в detached HEAD, сделайте git branch [имя новой ветки]).

    Если вы уже переключили HEAD на ветку, а неассоциированные коммиты остались позади, сделайте git reflog, найдите по сообщению свой коммит, переключитесь обратно и создайте ветку. После сборки мусора (как правило, она происходит во время обновления состояния удаленного репозитория, то есть git push) все неассоциированные коммиты будут скорее всего потеряны (если на них не было меток, tags, которые в данных рекомендациях не рассматриваются). [↑](#footnote-ref-37)
38. Строго говоря, это более предпочтительный вариант, к которому следует прибегнуть, если на компьютере есть несколько toolset’ов, например Visual Studio + Visual C++ + CMake 3.14 + MSBuild и Codeblocks + MinGW + CMake 3.11 + Make, во избежание конфликта имен. Имеет смысл создать start\_session.bat, в котором настроить PATH для текущей сессии и здесь же продолжить работу. [↑](#footnote-ref-38)
39. Даже если Base – динамическая библиотека, компоновщик в составе Visual C++ всё равно генерирует модуль-заглушку для статической компоновки (Base.lib) во время компоновки Base, а во время компоновки Dependent выполняет symbol resolution с участием Base.lib, при этом Base.lib содержит некоторую информацию об ABI, и может быть исключена из процесса компоновки, если явно определить некоторые детали, которые не рассматриваются в данных рекомендациях. [↑](#footnote-ref-39)
40. Кроме того, что версии библиотеки времени выполнения, входящие в состав Microsoft Visual C++ Redistributable, имеют отладочные версии (например, vcruntime140.dll, следует убедиться, что вы не используете эти версии, если собираетесь запускать свои программы на машине пользователя, поскольку у пользователя их может не оказаться), они также имеют по историческим причинам версии для работы в многопоточной среде, отличающиеся некоторым разделяемым для разных потоков контекстом. Использование однопоточной версии библиотеки времени выполнения не рекомендуется.

    Убедитесь, что во всех компонуемых проектах используете одну и ту же библиотеку времени выполнения, или (не рекомендуется) различные варианты библиотеки скомпонованы каждая со своим модулем статически. [↑](#footnote-ref-40)
41. Ближайший аналог – POSIX ls. [↑](#footnote-ref-41)
42. SFML устанавливает постфикс -s для статической компоновки, и -d для отладочных символов в составе библиотеки; таким образом модуль оконной подсистемы, предназначенный для статической компоновки и последующей отладки, будет иметь вид sfml-window-s-d.lib). [↑](#footnote-ref-42)
43. Используемая по умолчанию система сборки Ninja имеет ошибку (bug), из-за которой приоритет Visual C++ при поиске компилятора заведомо ниже, чем всех прочих компиляторов в системе, что можно выяснить, изучив содержимое окна Вывод при сборке проекта CMake (ошибки проекта будут выведены с указанием пути до программы, например неверно выбранного компилятора).  
    Подробнее см. <https://developercommunity.visualstudio.com/content/problem/114469/cmake-integration-with-ninja-generator-fails-to-fi.html> и **«Администрирование 1.6 - Использование MSBuild вместо Ninja в качестве нижележащей системы сборки при работе с проектами CMake в Visual Studio».** [↑](#footnote-ref-43)
44. Строго говоря, так как проект nlohmann\_json не производит никаких исполняемых модулей, а состоит целиком из шаблонного кода и сущностей, не требующих компоновки, компоновка является избыточной, и интеграция конкретно nlohmann\_json завершена сразу после указания install prefix, в котором находятся headers, соответствующие проекту. [↑](#footnote-ref-44)