**Методические рекомендации**

**по разработке с использованием IDE**

**Microsoft Visual Studio 2019 Community**

Оглавление

1. [Введение 3](#_Toc24386546)
2. [Установка Visual Studio 4](#_Toc24386547)
3. [Пользовательский интерфейс Visual Studio 5](#_Toc24386548)
4. [Теоретические основы организации разработки 10](#_Toc24386549)
5. [Стандарт программирования 20](#_Toc24386550)
6. [Важные замечания об ознакомительных работах 21](#_Toc24386551)
7. [Ознакомительная работа 1.1 – Быстрый старт разработки на C++ 23](#_Toc24386552)
8. [Ознакомительная работа 1.2 – Внешняя система сборки CMake 26](#_Toc24386553)
9. [Ознакомительная работа 1.3 – Автоматизированное тестирование с Google Test 31](#_Toc24386554)
10. [Ознакомительная работа 1.4 – Применение dynamic linking / static linking 35](#_Toc24386555)
11. [Ознакомительная работа 1.5 – Контроль версий с помощью git 35](#_Toc24386556)
12. [Ознакомительная работа 2.1 – Быстрый старт разработки на C# 35](#_Toc24386557)
13. [Приложение 1: Администрирование и build engineering 35](#_Toc24386558)

# Введение

Интегрированная среда разработки (**Integrated Development Environment, далее «среда разработки» или «IDE»**) – это программа, используемая 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++ и C# с использованием Microsoft Visual Studio 2019 Community (**далее – Visual Studio**) в ОС Windows.

Для C++ 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.

# Пользовательский интерфейс Visual Studio

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

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

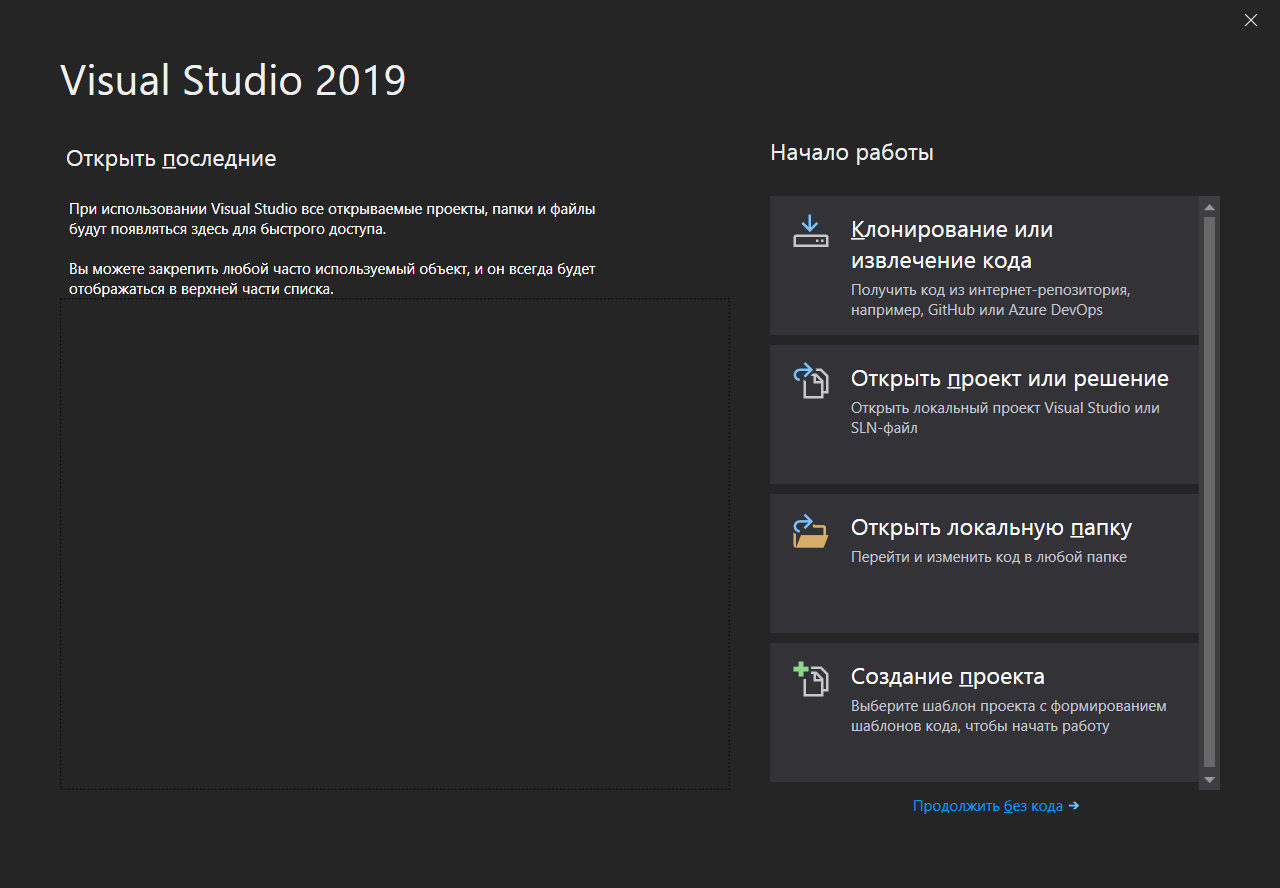


Рисунок 1 - Начальный экран

Проектом (**project**) в Visual Studio называется набор файлов, участвующих в сборке. Один проект представляет одно приложение, одну библиотеку или один набор ресурсов.

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

Одно решение может использоваться для представления одного программного продукта целиком.

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.

Репозиторием (**software** **repository,** также просто **repository**) называется централизованное хранилище файлов, как правило, представляющих решение.

Работа в Visual Studio возможна после создания проекта и включающего его решения, клонирования существующего решения из локального (на диске) или удаленного (в сети) репозитория, открытия существующего решения или открытия произвольной папки на диске. 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.

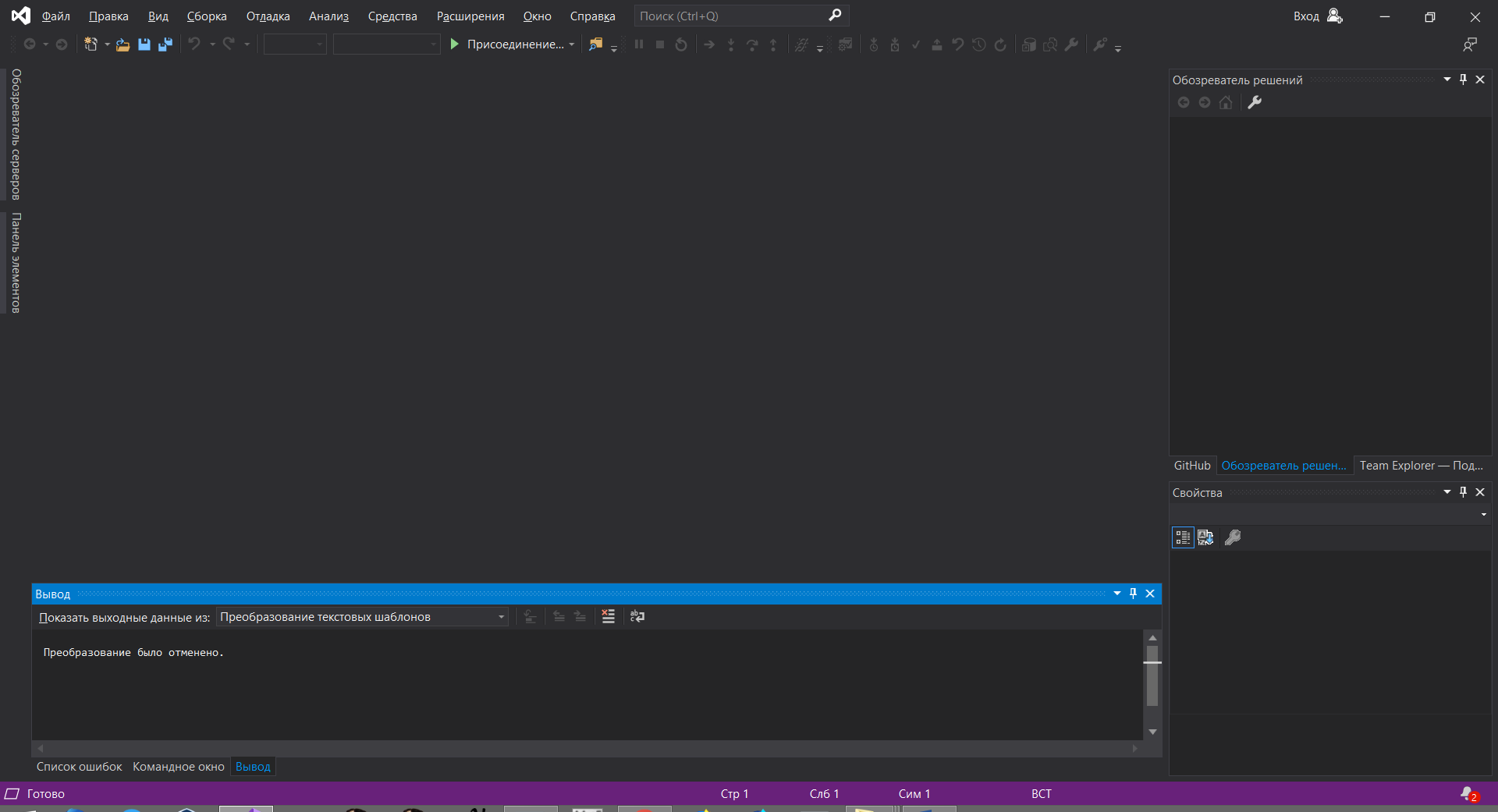


Рисунок 2 - Главный экран

**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.**

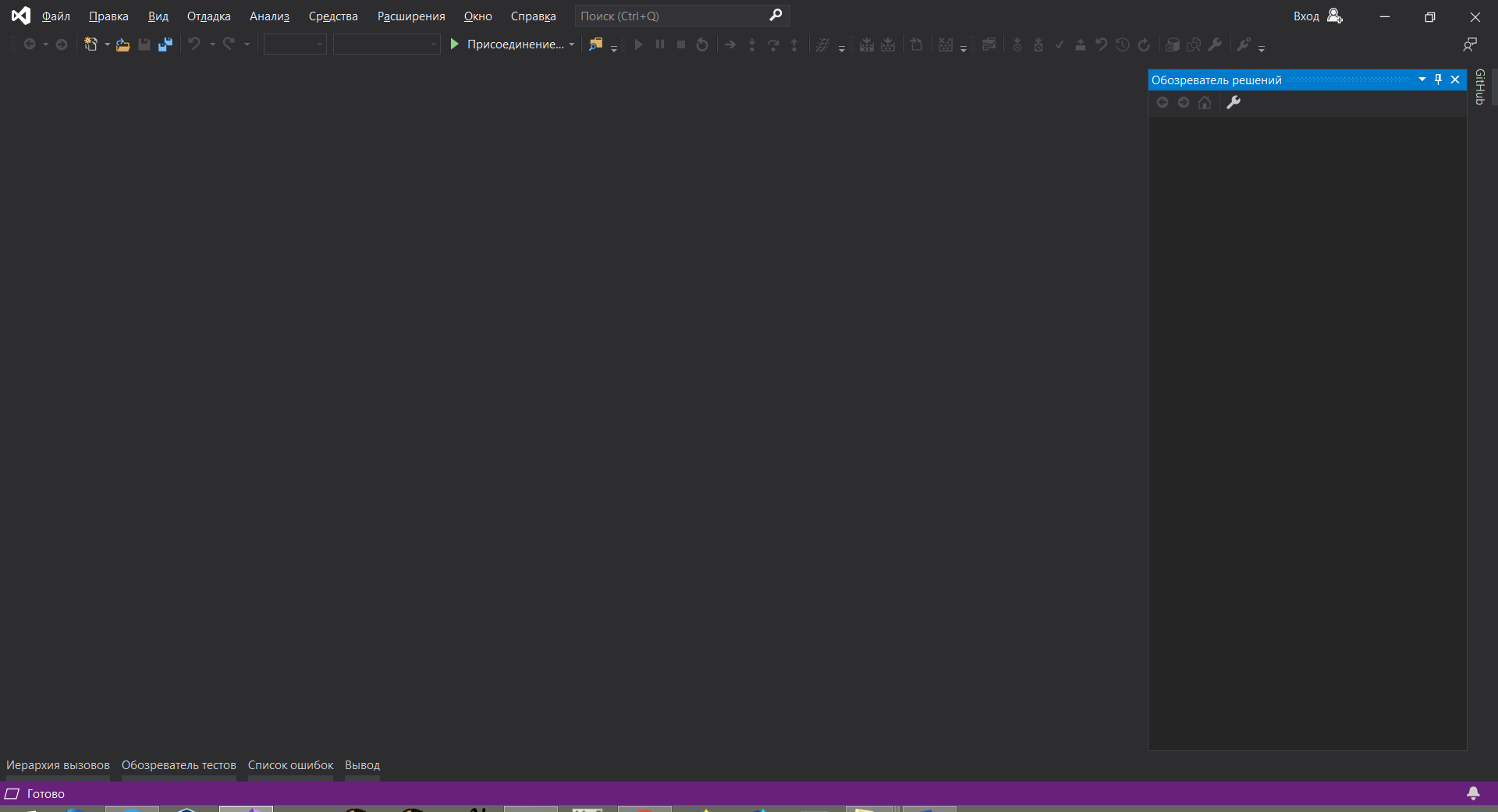


Рисунок 3 - Пример настройки окон и панелей инструментов на главном экране

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 организации разработки

**\*\*\***

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 трансляции программы;
* **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 дальнейшее 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».

**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, готовой к загрузке операционной системой для дальнейшего выполнения.

**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 (об этом далее в Послесловии). 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 только 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, не входящие в эти две категории, т. н. отладочные символы, или **debugging symbols** (н-р переменные из заведомо недоступной области видимости).[[9]](#footnote-9)

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.[[10]](#footnote-10)

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[[11]](#footnote-11), 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[[12]](#footnote-12), 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.[[13]](#footnote-13)

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,

**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. [[14]](#footnote-14)

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».

**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 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)).

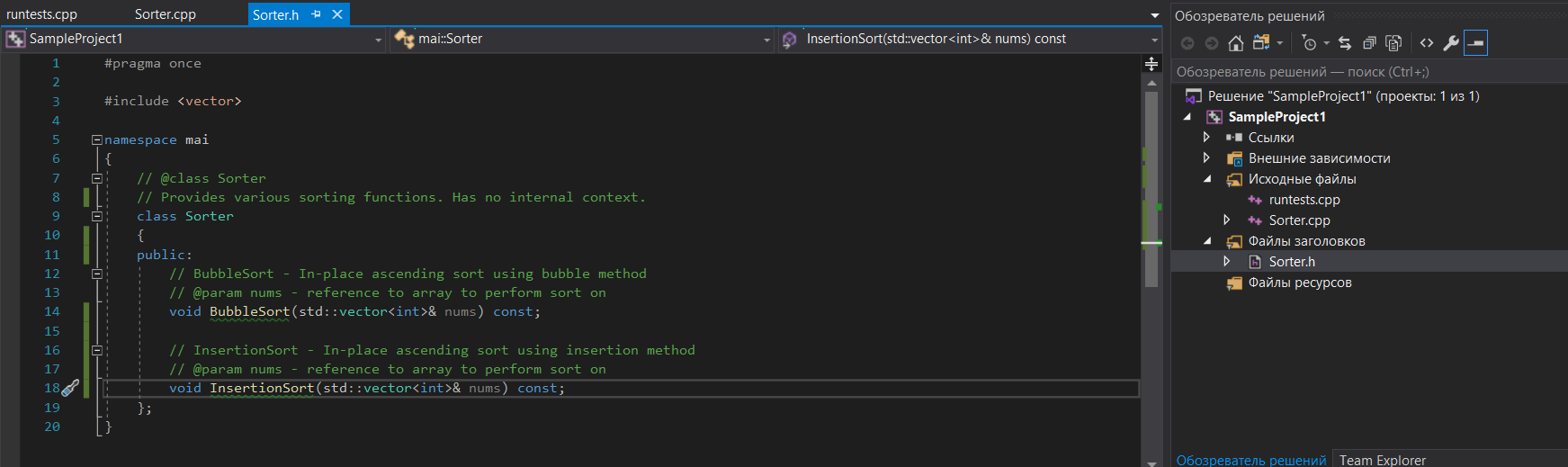


Рисунок 4 - Общий вид проекта и кода заголовочного файла класса Sorter

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).

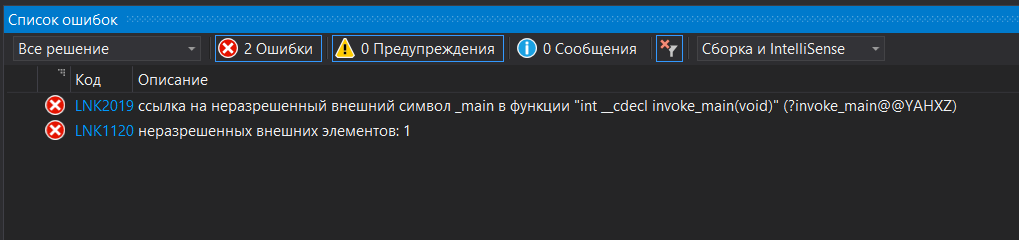


Рисунок 5 - Ошибка компоновки в ходе сборки приложения

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.[[15]](#footnote-15)

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[[16]](#footnote-16), 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.

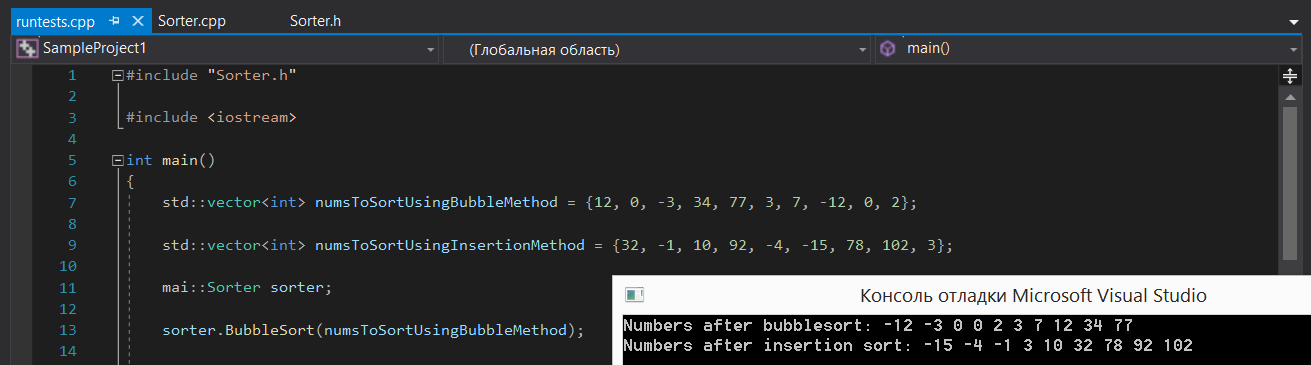


Рисунок 6 - Возможная реализация runtests.cpp и корректная работа класса Sorter

# 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).

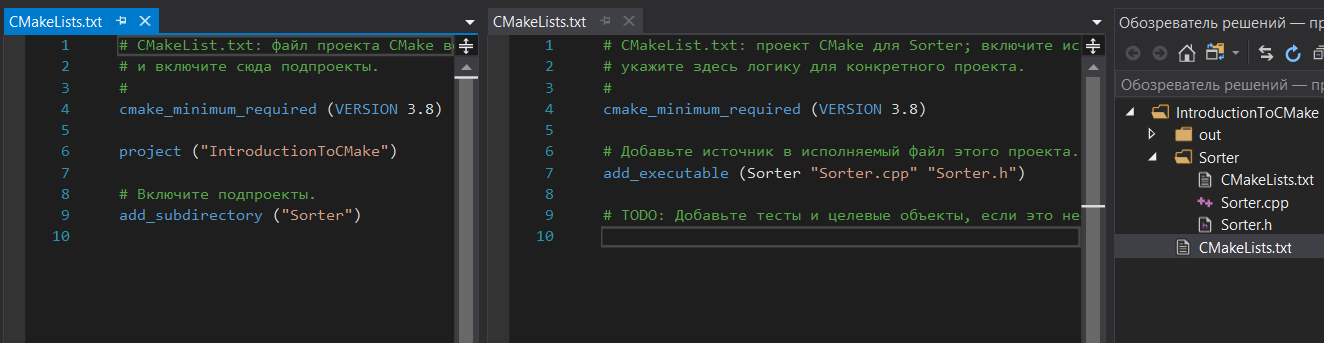


Рисунок 7 - Автогенерируемые сценарии CMake для решения и вложенного проекта

**О сущностях CMake**

**CMake target** – это приложение (**CMake executable**), библиотека (**CMake library**) или какой-то другой, неспециализированный результат сборки (**CMake custom target**).

**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).

**CMakeLists.txt[[17]](#footnote-17)** – 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.

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[[18]](#footnote-18), 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.

**CMake cache** – это результат работы CMake, включающий в себя сценарий нижележащей системы сборки.

Результатом запуска непосредственно 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) и не случилось еще некоторых изменений в проекте CMake.

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. [[19]](#footnote-19)

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).

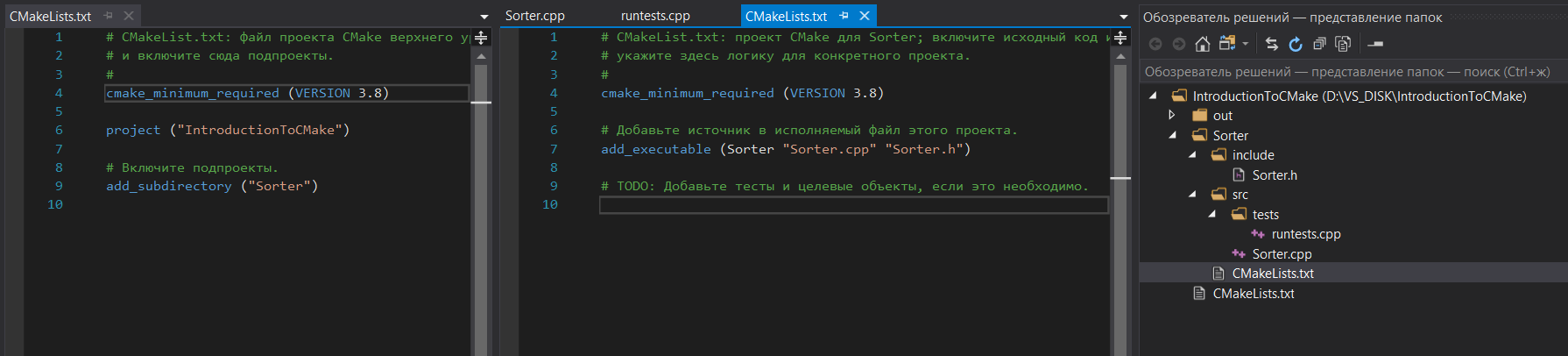


Рисунок 8 - Итоговая структура проекта Sorter в составе решения IntroductionToCMake

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 тип исполняемого модуля, получаемый после успешного завершения сборки, указав спецификатор STATIC):

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.

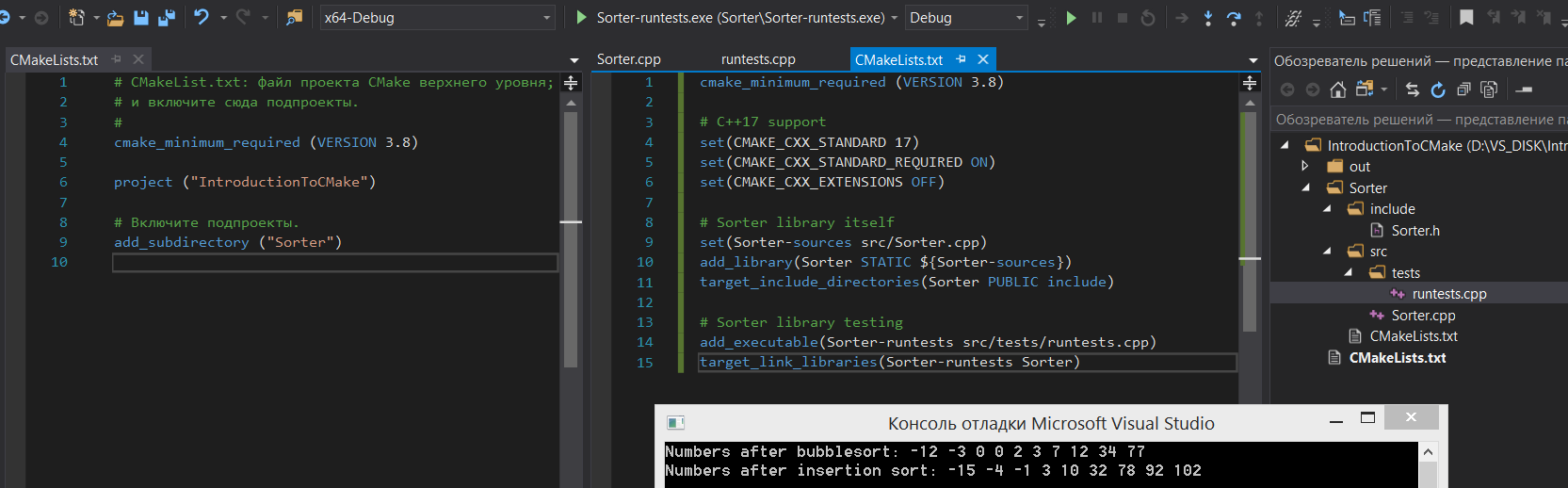


Рисунок 9 - Итоговый вид и результат работы модуля тестов в составе CMake проекта Sorter

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 («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[[20]](#footnote-20): модуль gtest предоставляет возможности для тестирования, а gtest\_main тривиальную реализацию main по умолчанию.[[21]](#footnote-21) Для декорирования хода работы тестовой оснастки main можно реализовать самостоятельно, остальное Visual Studio сделает автоматически.

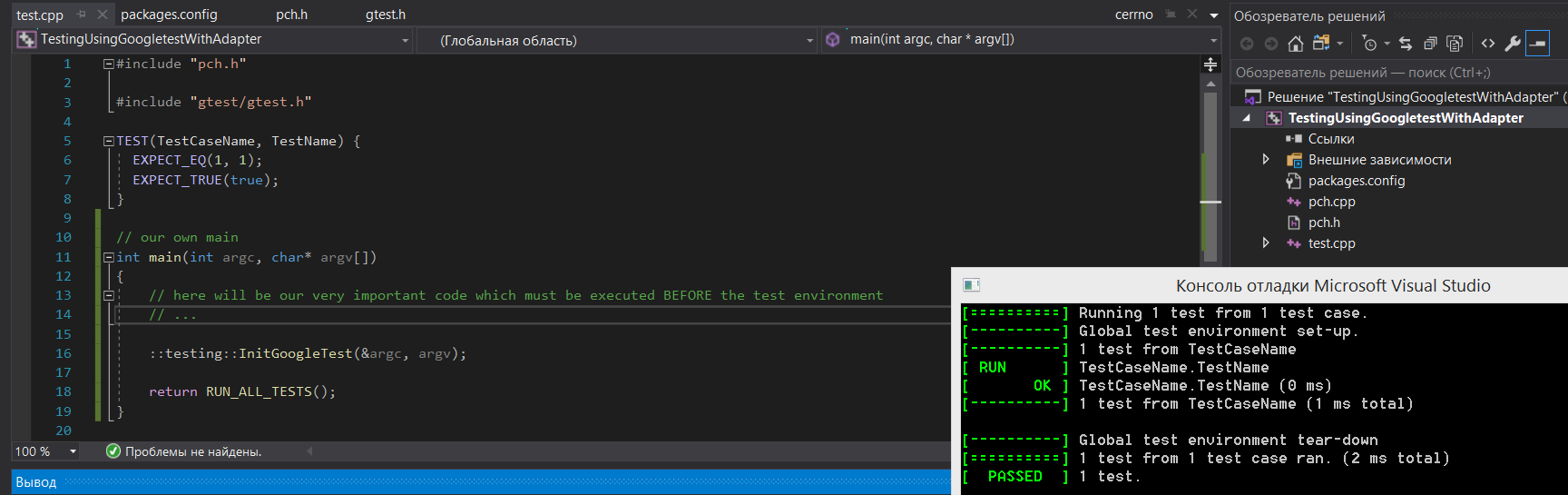


Рисунок 10 - Собственная реализация main, инициализирующей Google Test и запускающей тесты

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

**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[[22]](#footnote-22);
* 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.

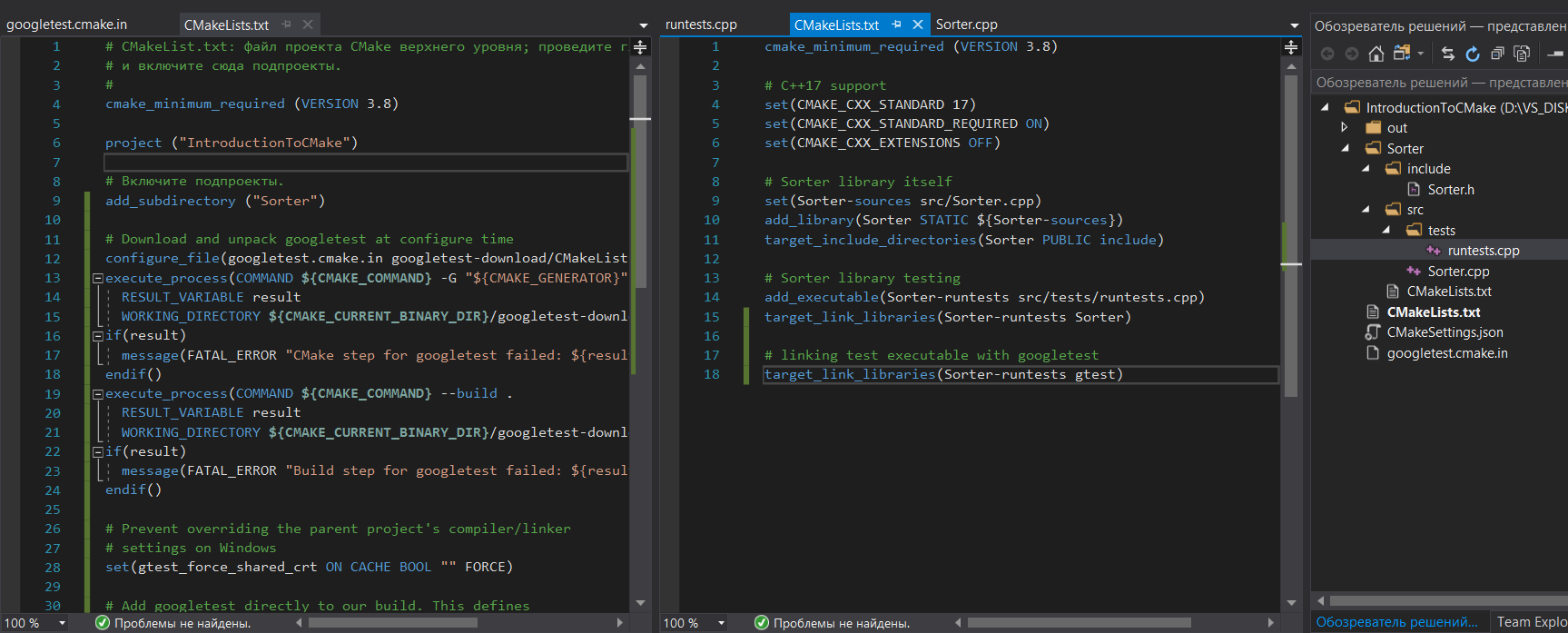


Рисунок 11 - Окончательная структура проекта,включая средства тестирования

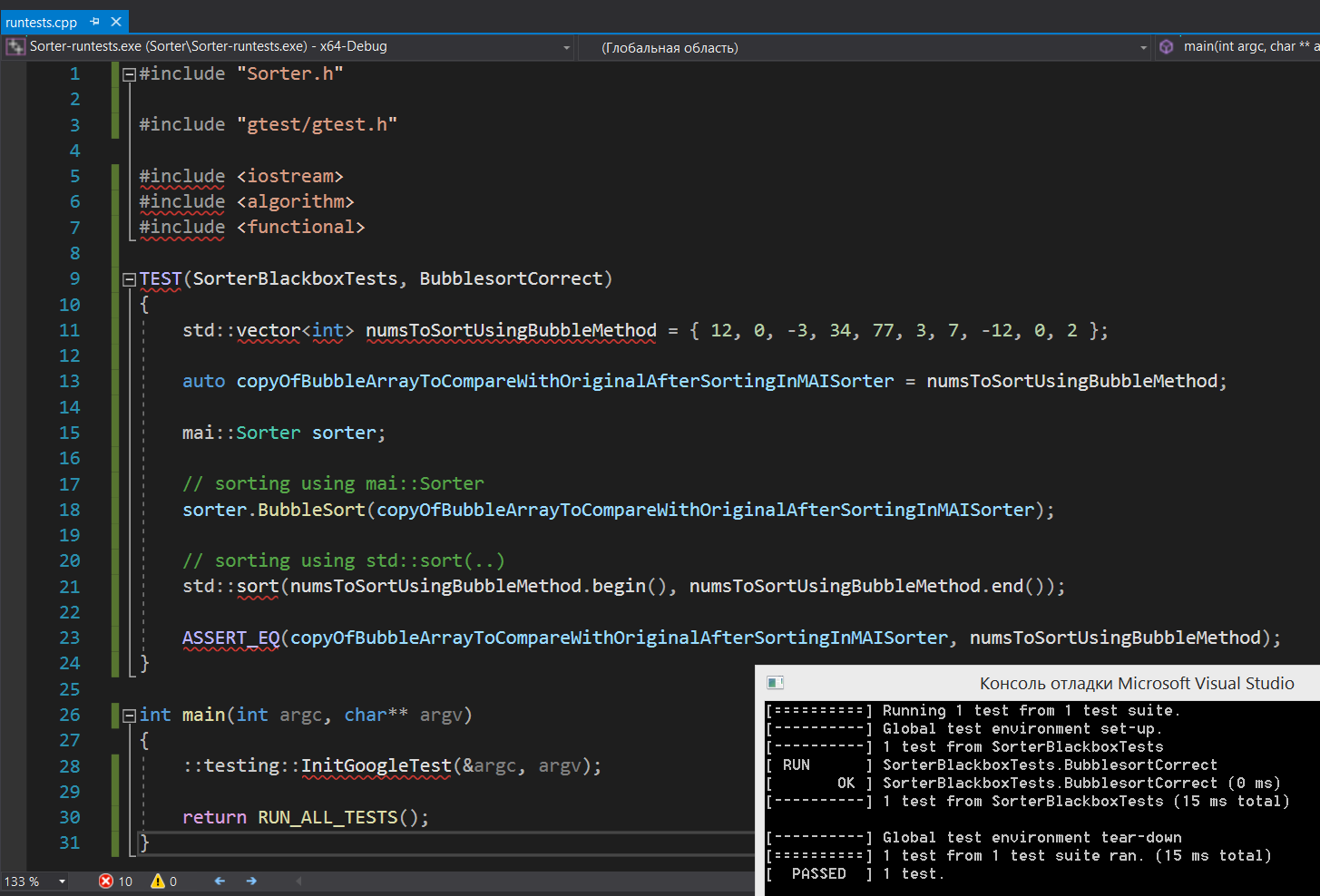
- 

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 Приложению 1.

Воспользуйтесь результатом работы «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. Ознакомившись с Приложением и 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 – Применение dynamic linking / static linking

# Ознакомительная работа 1.5 – Контроль версий с помощью git

# Ознакомительная работа 2.1 – Быстрый старт разработки на C#

# Приложение 1: Администрирование и build engineering

**Установка каталогов внешних утилит в PATH для быстрого доступа**

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 с компьютера и настроить только один. Для компиляторов, компоновщиков и систем сборки это критически важно, поскольку расхождение в версиях на разных этапах сборки может привести к несовпадению ABI (расхождение соглашений о вызовах, соглашений о расширениях имен и т.д. может привести к ошибке компоновки, неверной компоновке или какой-либо 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.

Настройка PATH для набора рассматриваемых в рекомендациях утилит завершена.

**Генерация сценария нижележащей системы сборки с помощью CMake.**

**Сборка с использованием 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” ../

После завершения генерации 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

make [название\_target] // система make

После успешной сборки в каталоге решения появится каталог сборки (если не указано иного), в котором находятся результирующие исполняемые модули, отладочные символы и т.д. Работа завершена.

**Переименование вложенного проекта Visual Studio**

Для переименования 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);
* Исправьте старые зависимости, во всех зависимых от переименованного проектах изменив Проект -> Свойства -> Свойства конфигурации -> C / C++ -> Общие -> Дополнительные каталоги включаемых файлов в соответствии с новым именем проекта, если существовали ссылки на его каталог; Свойства конфигурации -> C / C++ -> Все параметры -> Дополнительные каталоги включаемых файлов; Свойства конфигурации -> Компоновщик -> Общие -> Дополнительные каталоги библиотек; Компоновщик -> Ввод -> Дополнительные зависимости аналогичным образом.

**Компоновка проекта Visual Studio с собранным ранее исполняемым модулем**

Каждый проект в составе решения Visual Studio имеет собственные зависимости, конфигурируемые в свойствах проекта и передаваемые затем в систему MSBuild.

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 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. 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-10)
11. 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-11)
12. Среди которых не было директивы #error (см. раздел 19.5 Стандарта). [↑](#footnote-ref-12)
13. 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-13)
14. 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-14)
15. См. Стандарт 6.8.3.1 (Basic concepts -> Program execution -> Start and termination -> main function) [↑](#footnote-ref-15)
16. 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-16)
17. 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-17)
18. 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-18)
19. 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 не рассматривается в данных рекомендациях.

    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 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-19)
20. Google Mock распространяется вместе с Google Test, используется для тестирования с применением fixture classes и де-факто является третьим используемым target (“gmock”). [↑](#footnote-ref-20)
21. Подробнее: <https://github.com/google/googletest/blob/master/googletest/docs/primer.md#writing-the-main-function> [↑](#footnote-ref-21)
22. 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-22)