**Методические рекомендации**

**по разработке с использованием IDE**

**Microsoft Visual Studio 2019 Community**

Оглавление

1. [Введение 3](#_Toc21895002)
2. [Установка Visual Studio 5](#_Toc21895003)
3. [Пользовательский интерфейс Visual Studio 6](#_Toc21895004)
4. [Организация процесса сборки 11](#_Toc21895005)

# 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”, внешние программы являются “back end”).

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) 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» (**workloads**), 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 workloads, 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.

# Пользовательский интерфейс Visual Studio

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

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

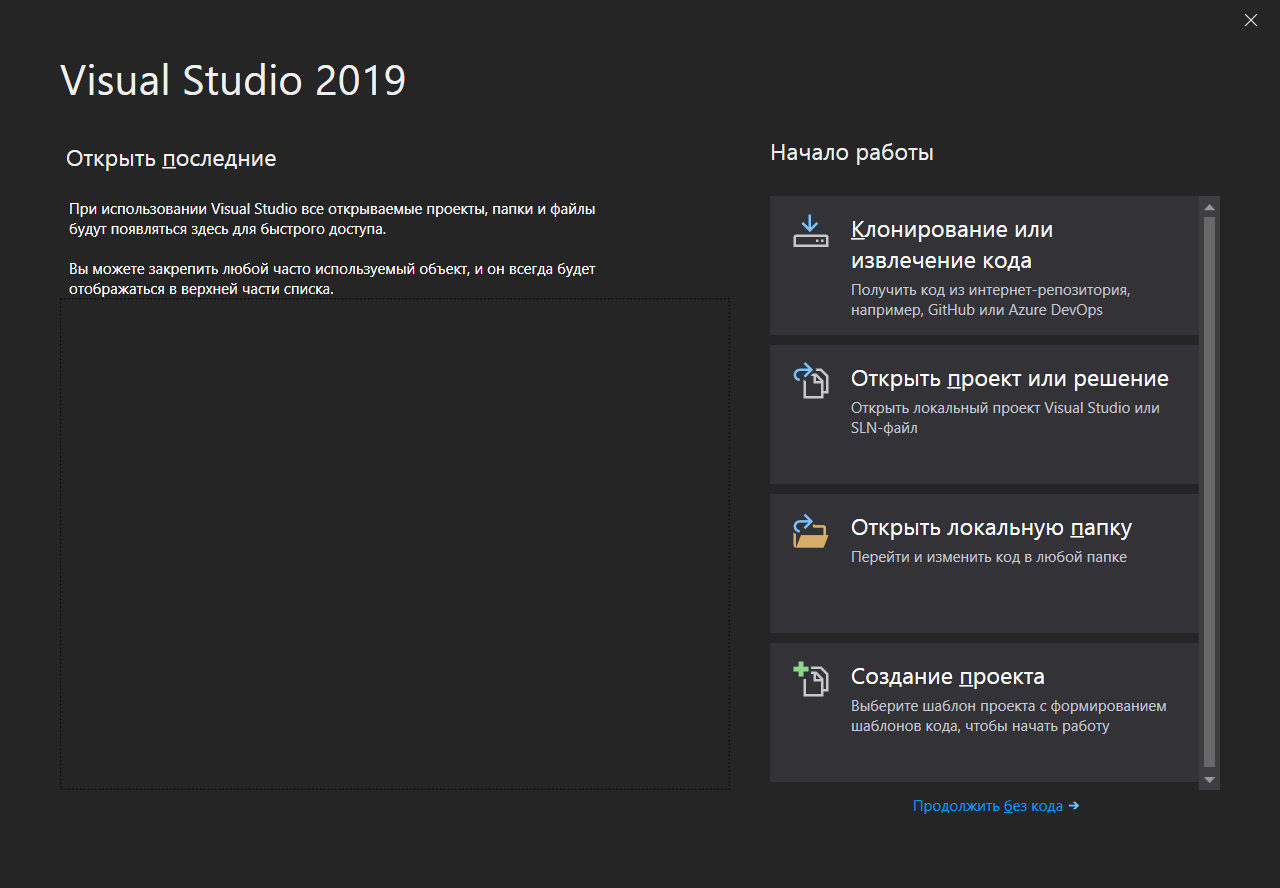


Рисунок - Начальный экран

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 проект.

Работа в Visual Studio возможна после создания проекта, клонирования существующего проекта из локального (на диске) или удаленного (в сети) репозитория, открытия существующего проекта 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.

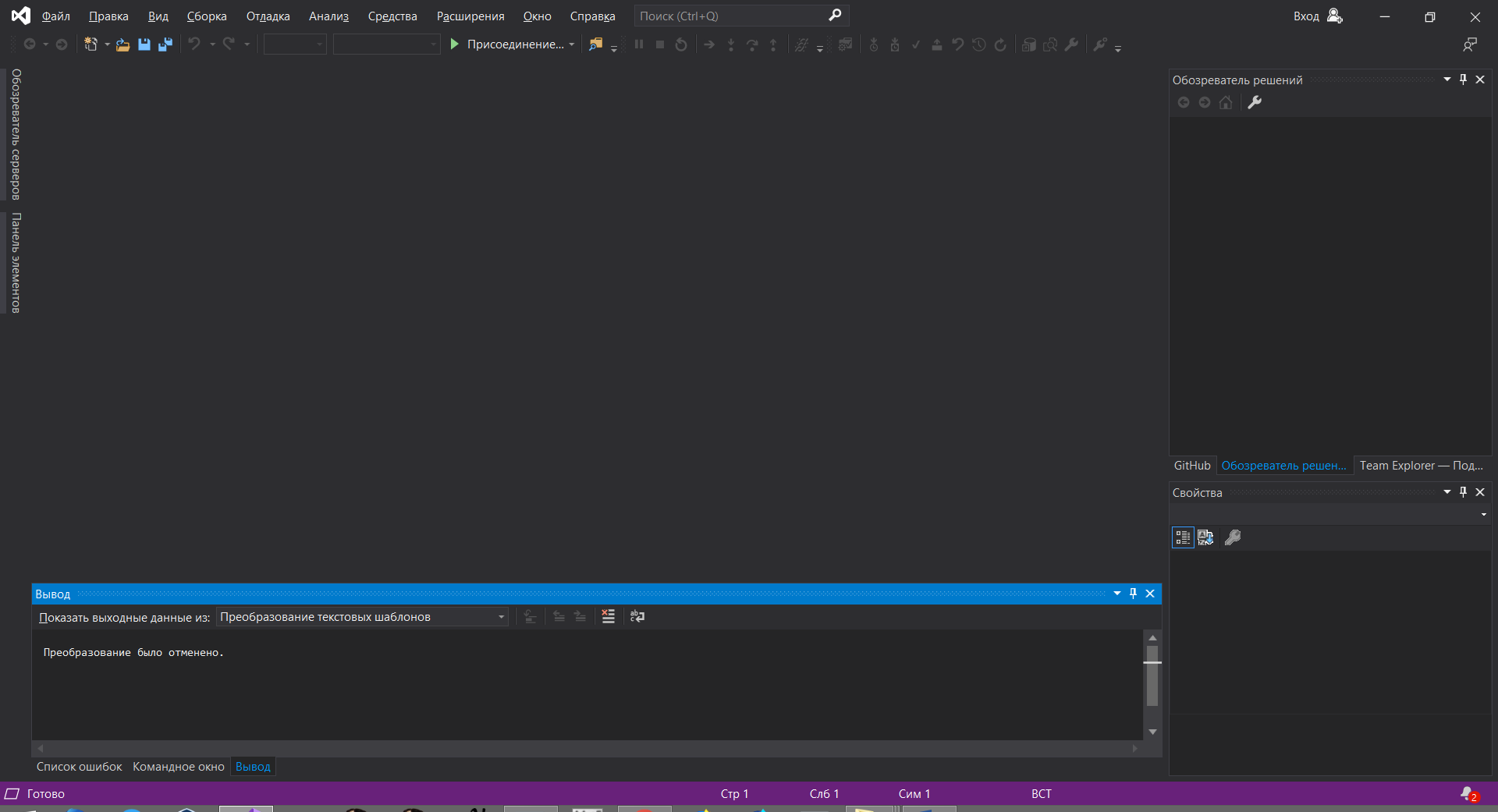


Рисунок - Главный экран

**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.**

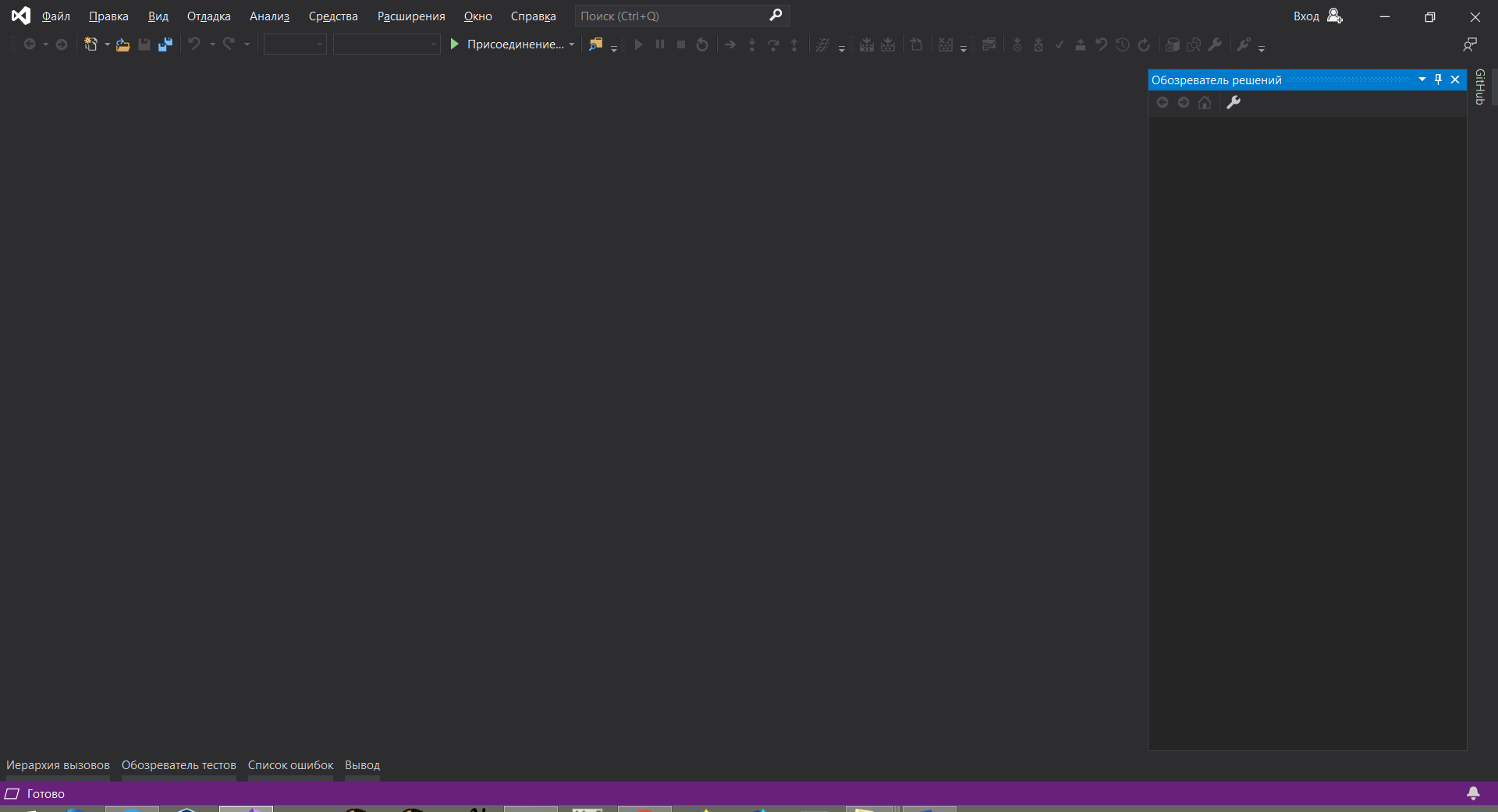


Рисунок - Пример настройки окон и панелей инструментов на главном экране

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.

# Организация процессов сборки и выполнения

**\*\*\***

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**).

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

Стандарт языка C++ [[1]](#footnote-1) 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[[2]](#footnote-2) 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[[3]](#footnote-3), 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[[4]](#footnote-4) с существующим определением функции (**function definition**), определением шаблона класса (**template class definition**) и так далее.

Такое сопоставление, носящее название разрешения символов (**symbol resolving**, symbol – то же, что идентификатор) или разрешения ссылок (**reference resolving**), 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[[5]](#footnote-5), завершает ход трансляции исходного кода в исполняемый модуль.

**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 – вместе с упомянутой дополнительной информацией о целевой платформе передается обычно 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.

Целевая платформа (**target platform**), на которой будет выполняться скомпилированный модуль, и хост-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 уже собранного для другой платформы компилятора, передачей ему собственного исходного кода и указанием целевой платформы. В данных рекомендациях разработка с использованием кросс-компиляции не рассматривается.

Как правило, за один вызов конвейера из двух программ – препроцессора и компилятора – транслируется только часть исходного кода (называемая единицей трансляции, **translation unit**), предназначенного для сборки.[[6]](#footnote-6)

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[[7]](#footnote-7), 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:

* Система сборки запускает для каждого файла исходного кода последовательно препроцессор и компилятор 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[[8]](#footnote-8), компилятор без ошибок скомпилировал исходный код, а компоновщик выполнил компоновку, в результате которой не осталось неразрешенных ссылок, и 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.

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 MSBuild, MSBuild по-прежнему будет задействована, хотя пользователь будет работать только с CMake.

**Трансляция в промежуточный код**

Возможно сначала транслировать исходный код программы в некий промежуточный код (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. [[9]](#footnote-9)

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**, являющийся частью workload разработки на языке C#**.**

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. 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 открытые реализации среды CLI, например, любительский Mono или .NET Core, реализованный и поддерживаемый Microsoft,

1. 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-1)
2. Ход трансляции описывается разделом 2 главы 5 Стандарта (далее подобные ссылки кратко называются «раздел 5.2 Стандарта»). [↑](#footnote-ref-2)
3. 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-3)
4. Идентификатор — это имя функции, переменной, класса и т.д., по которому программист обращается к функции и т.д. в коде. [↑](#footnote-ref-4)
5. 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-5)
6. 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-6)
7. 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-7)
8. Среди которых не было директивы #error (см. раздел 19.5 Стандарта). [↑](#footnote-ref-8)
9. 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. В таком случае 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, поскольку JIT-компиляция происходит безусловно и её продуктом является не оптимизированный промежуточный код, а код физической машины, выполняемый без участия интерпретатора. [↑](#footnote-ref-9)