

**МОЛДАВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**  
**ФАКУЛЬТЕТ МАТЕМАТИКИ И ИНФОРМАТИКИ**  
**ДЕПАРТАМЕНТ ИНФОРМАТИКИ**

**ФАМИЛИЯ Имя**

**ИЗУЧЕНИЕ  $\text{\LaTeX}$  И GIT**

**0613.4 ИНФОРМАТИКА**

**Практика**

Директор департамента:	_____	КАПЧЕЛЯ Титу,
	(подпись)	доктор физико-математических наук,
		преподаватель университета
Научный руководитель:	_____	КУРМАНСКИЙ Антон,
	(подпись)	ассистент университета
Автор:	_____	ФАМИЛИЯ Имя,
	(подпись)	студент группы I2402

**КИШИНЕВ – 2025 Г.**

## Оглавление

<b>Аннотация</b>	<b>4</b>
<b>ВВЕДЕНИЕ</b>	<b>6</b>
<b>I. Введение в систему вёрстки L<sup>A</sup>T<sub>E</sub>X</b>	<b>8</b>
1.1. Зачем нужен L <sup>A</sup> T <sub>E</sub> X?	8
1.2. История создания L <sup>A</sup> T <sub>E</sub> X	8
1.3. Преимущества и ключевые особенности	8
1.4. Основные команды L <sup>A</sup> T <sub>E</sub> X	8
1.5. Дополнительные возможности	9
1.6. Редакторы и компиляция	9
1.7. Выводы	9
1.8. Выводы для главы 1	10
<b>II. Введение в систему контроля версий Git</b>	<b>11</b>
2.1. Зачем нужен Git?	11
2.2. История создания Git	11
2.3. Ключевые особенности Git	11
2.4. Основные команды Git	11
2.5. Источники и полезные ссылки	12
2.6. Выводы для главы 2	12
<b>III. Практика с Learn Git Branching</b>	<b>13</b>
3.1. Main — Базовые концепции Git	13
3.1.1. Introduction Sequence	13
3.1.2. Ramping Up	16
3.1.3. Moving Work Around	20
3.1.4. A Mixed Bag	22
3.1.5. Advanced Topics	26
3.2. Remote — Работа с удалёнными репозиториями	29
3.2.1. Push & Pull — Git Remotes	29
3.2.2. To Origin and Beyond — Advanced Git Remotes	36

3.3. Выводы для главы 3 . . . . .	41
<b>IV. Git Immersion — Пошаговое погружение в Git</b>	<b>42</b>
4.1. Lab 1: Setup . . . . .	42
4.2. Lab 2: More Setup . . . . .	42
4.3. Lab 3: Create a Project . . . . .	43
4.4. Lab 4: Checking Status . . . . .	43
4.5. Lab 5: Making Changes . . . . .	43
4.6. Lab 6: Staging Changes . . . . .	44
4.7. Lab 7: Staging and Committing . . . . .	44
4.8. Lab 8: Committing Changes . . . . .	44
4.9. Lab 9: Changes, not Files . . . . .	45
4.10. Lab 10: History . . . . .	46
4.11. Lab 11: Aliases . . . . .	47
4.12. Lab 12: Getting Old Versions . . . . .	47
4.13. Lab 13: Tagging Versions . . . . .	48
4.14. Lab 14: Undoing Local Changes (before staging) . . . . .	49
4.15. Lab 15: Undoing Staged Changes (before committing) . . . . .	49
4.16. Lab 16: Undoing Committed Changes . . . . .	50
4.17. Lab 17: Removing Commits from a Branch . . . . .	51
4.18. Lab 18: Remove the oops Tag . . . . .	51
4.19. Lab 19: Amending Commits . . . . .	52
4.20. Lab 20: Moving Files . . . . .	53
4.21. Lab 21: More Structure . . . . .	54
4.22. Lab 22: Git Internals — The .git Directory . . . . .	54
4.23. Lab 23: Git Internals — Working Directly with Git Objects . . . . .	55
4.24. Lab 24: Creating a Branch . . . . .	56
4.25. Lab 25: Navigating Branches . . . . .	57
4.26. Lab 25: Navigating Branches . . . . .	57
4.27. Lab 26: Changes in Main . . . . .	57
4.28. Lab 27: Viewing Diverging Branches . . . . .	58
4.29. Lab 28: Merging Branches . . . . .	58
4.30. Lab 29: Creating a Conflict . . . . .	58
4.31. Lab 30: Resolving Conflicts . . . . .	59

4.32. Lab 31: Rebasing vs Merging . . . . .	59
4.33. Lab 32: Resetting the Greet Branch . . . . .	60
4.34. Lab 33: Resetting the Main Branch . . . . .	61
4.35. Выводы для главы 4 . . . . .	61
<b>Заключение и рекомендации</b>	<b>62</b>
<b>Список литературы</b>	<b>63</b>

## Аннотация

**Практическая работа на тему “Изучение  $\LaTeX$  и Git”, студента Фамилия Имя Отчество, группа I2402.**

**Структура практической работы.** Практическая работа состоит из: Введения, трёх глав, Заключения с рекомендациями, библиографии и приложений. В работе представлены теоретические сведения, практические задания и иллюстрации, отражающие процесс изучения систем  $\LaTeX$  и Git. Основной текст включает подробные инструкции и скриншоты, демонстрирующие выполнение заданий на платформе Learn Git Branching [1].

**Актуальность.** Системы управления версиями и профессиональной вёрстки документов играют важную роль в современной IT-индустрии. Git широко используется в командной разработке программного обеспечения [2], [3], а  $\LaTeX$  — в научной и инженерной документации [4]—[6]. Их знание необходимо для любого начинающего специалиста в сфере ИТ.

**Цель и задачи исследования.** Целью данной работы является освоение основ использования  $\LaTeX$  и Git, а также практическое применение полученных знаний на платформе Learn Git Branching.

Для достижения цели были поставлены следующие задачи:

- Изучить синтаксис и особенности вёрстки с использованием  $\LaTeX$  [4], [5];
- Освоить ключевые команды Git и принципы работы с системой контроля версий [3];
- Пройти практические задания Learn Git Branching по ветвлению, слиянию и удалённым репозиториям [1];
- Подготовить отчётный документ в  $\LaTeX$ , используя современные пакеты, включая подсветку кода с помощью minted [7].

**Ожидаемые и полученные результаты.** Ожидалось овладение базовыми навыками вёрстки и контроля версий. В результате был оформлен структурированный документ в  $\LaTeX$ , содержащий полную практическую часть по Git и Learn Git Branching, включая графические иллюстрации и разъяснения к каждому этапу.

**Важные решённые проблемы.** В процессе работы были решены следующие задачи:

- организация многоуровневой структуры документа в  $\LaTeX$ ;
- изучение моделей ветвления и командной работы в Git;
- визуализация выполнения практических заданий;
- интеграция технического и визуального материала в единую отчётную работу.

**Практическая ценность.** Результаты работы могут быть использованы как руководство для студентов, начинающих осваивать  $\text{\LaTeX}$  и Git, а также как база для подготовки отчётной и научной документации в технических вузах.

Весь исходный код проекта доступен на GitHub по следующей ссылке: <https://github.com/USER/REPO>.

## ВВЕДЕНИЕ

### **Актуальность и важность темы.**

Освоение систем  $\text{\LaTeX}$  и Git [3], [4] является важным шагом в формировании технической грамотности современного IT-специалиста. Git применяется повсеместно в процессе разработки программного обеспечения, позволяя эффективно управлять изменениями и взаимодействовать в команде.  $\text{\LaTeX}$ , в свою очередь, обеспечивает профессиональную вёрстку документов, широко используемую в научной, инженерной и образовательной среде [5], [6].

Практическая направленность данной работы, подкреплённая выполнением заданий на интерактивной платформе Learn Git Branching [1], делает изучение не только теоретически обоснованным, но и прикладным.

### **Цель и задачи.**

Цель — получить практические навыки работы с  $\text{\LaTeX}$  и Git, а также закрепить знания через выполнение реальных задач, включая визуализацию ветвлений и взаимодействие с удалёнными репозиториями.

Задачи:

- Изучить принципы и команды Git, в том числе ветвление, слияние и работу с удалёнными репозиториями;
- Освоить синтаксис  $\text{\LaTeX}$ , включая структуру документа, оформление кода, таблиц, рисунков;
- Подготовить отчёт с использованием современных пакетов (minted, graphicx и др.) [7];
- Пройти все ключевые блоки платформы Learn Git Branching: Introduction, Ramping Up, Moving Work Around, Mixed Bag и Advanced Topics;
- Создать структурированный отчёт, включающий скриншоты выполнения заданий и объяснение решений.

### **Методологическая и технологическая база.**

В работе использованы:

- Платформа Learn Git Branching [1] для симуляции работы с Git;
- Компилятор XeLaTeX для создания PDF-документа;
- Пакеты minted, graphicx, biblatex для оформления кода, иллюстраций и библиографии;
- Git CLI и веб-интерфейс GitHub (в обучающей форме, без публикации проекта).

### **Научная новизна / оригинальность.**

Работа сочетает сразу два технологических направления — вёрстку и контроль версий — и демонстрирует практическое применение обеих технологий в едином отчёте. Также показан прогрессивный подход к обучению Git через визуальное взаимодействие, что повышает понимание абстрактных концепций.

### **Практическая ценность.**

Полученные материалы могут использоваться:

- как пособие для студентов технических направлений;
- как пример оформления отчётов в  $\text{\LaTeX}$ ;
- как руководство по решению задач Git через Learn Git Branching.

### **Краткое содержание работы.**

- Первая глава знакомит с основами системы  $\text{\LaTeX}$ , её назначением, преимуществами и базовыми конструкциями;
- Вторая глава описывает Git, его возможности и применимость в командной разработке;
- Третья глава посвящена практическому выполнению задач на платформе Learn Git Branching с подробным разбором каждой секции.



# I Введение в систему вёрстки L<sup>A</sup>T<sub>E</sub>X

## 1.1 Зачем нужен L<sup>A</sup>T<sub>E</sub>X?

L<sup>A</sup>T<sub>E</sub>X— это система вёрстки документов, созданная для подготовки высококачественных текстов. Особенно эффективно используется в научной, технической и инженерной среде. Система позволяет:

- структурировать текст с помощью разделов, глав и подглав;
- оформлять сложные математические формулы;
- вставлять таблицы, рисунки и графики;
- управлять библиографией и ссылками;
- создавать презентации, слайды и даже резюме.

## 1.2 История создания L<sup>A</sup>T<sub>E</sub>X

L<sup>A</sup>T<sub>E</sub>Xоснован на системе T<sub>E</sub>X, разработанной Дональдом Кнудом в 1978 году [5]. В 1980-х Лесли Лэмпорт разработал L<sup>A</sup>T<sub>E</sub>Xкак надстройку над T<sub>E</sub>X, сделав её более удобной для повседневного использования учёными и инженерами. С тех пор L<sup>A</sup>T<sub>E</sub>Xстал стандартом для научных публикаций [4].

## 1.3 Преимущества и ключевые особенности

- Полная автоматизация оформления (оглавления, списки рисунков, библиография).
- Идеальная типографика (переносы, выравнивание, интервалы).
- Гибкость в настройке внешнего вида.
- Возможность написания собственных макросов и пакетов.
- Совместимость с Git и другими системами контроля версий (текстовый формат).

## 1.4 Основные команды L<sup>A</sup>T<sub>E</sub>X

- `\documentclass{article}` — выбор типа документа (book, report, beamer и др.).
- `\usepackage{graphicx}` — подключение пакета для вставки изображений.
- `\section{Название}`, `\subsection{...}`, `\subsubsection{...}` — структура документа.
- `\textbf{жирный}`, `\emph{курсив}`, `\underline{подчёркнутый}` — оформление текста.
- `\begin{itemize}...\end{itemize}` — маркированный список.
- `\begin{enumerate}...\end{enumerate}` — нумерованный список.

- `\begin{table}...\end{table}` — таблицы с подписями.
- `\begin{figure}...\includegraphics{...}\end{figure}` — вставка изображений.
- `\begin{equation} a^2 + b^2 = c^2 \end{equation}` — математическая формула.
- `\label{...}, \ref{...}, \cite{...}` — система ссылок.
- `\bibliography{...}` и `\addbibresource{...}` — подключение библиографии.

## 1.5 Дополнительные возможности

- Работа с цветом: `\usepackage{xcolor}`, затем `\textcolor{red}{text}`.
- Создание графики внутри  $\text{\LaTeX}$ : `\usepackage{tikz}`.
- Создание слайдов: класс `beamer`.
- Код и подсветка синтаксиса: `\usepackage{minted}` или `listings` [7].
- Многоязычность: `\usepackage[main=russian,english]{babel}`.
- Индексы и оглавление: `\tableofcontents`, `\printindex`.
- Формулы в строке и на отдельной строке: `...\$` и `\[ ... \]` соответственно.

## 1.6 Редакторы и компиляция

Существует множество способов работы с  $\text{\LaTeX}$ . Наиболее популярные редакторы:

- Overleaf — онлайн-платформа с поддержкой совместной работы [6].
- TeXstudio — классический офлайн-редактор.
- VS Code + LaTeX Workshop — модульная и гибкая среда.

Компиляция может осуществляться движками `pdflatex`, `xelatex`, `lualatex` в зависимости от нужд (языки, шрифты и др.).

## 1.7 Выводы

$\text{\LaTeX}$  — мощный инструмент, требующий начального обучения, но обеспечивающий выдающееся качество выходного документа. Он идеален для научных, технических и структурно сложных работ.

## Источники

- [5] — официальный сайт проекта LaTeX.
- [4] — документация LaTeX2 $\epsilon$ .
- [6] — платформа Overleaf.
- [7] — информация о пакете `minted` для подсветки кода.

## 1.8 Выводы для главы 1

В первой главе была рассмотрена система вёрстки  $\text{\LaTeX}$ , её структура, синтаксис и основные преимущества. Изучены базовые команды, правила форматирования текста, оформление списков, таблиц, формул и изображений. Также были протестированы возможности расширения через подключение внешних пакетов, таких как `graphicx` для работы с графикой и `minted` для подсветки кода [7].

Работа с  $\text{\LaTeX}$  требует определённой дисциплины и внимания к структуре документа, но взамен предоставляет высокое качество выходного материала и гибкость при масштабировании проектов. В результате глава обеспечила фундамент для дальнейшего использования  $\text{\LaTeX}$  при оформлении технической документации.

## II Введение в систему контроля версий Git

### 2.1 Зачем нужен Git?

Git — это распределённая система контроля версий, созданная для эффективного управления изменениями в проектах с большим количеством исходного кода. Он позволяет:

- отслеживать историю изменений;
- работать в команде без конфликтов;
- откатываться к предыдущим версиям кода;
- создавать альтернативные ветви разработки (branching);
- автоматически сливать изменения (merging).

### 2.2 История создания Git

Git был создан Линусом Торвальдсом в 2005 году для управления исходным кодом ядра Linux после конфликта с предыдущей системой контроля версий — BitKeeper [3]. Требования:

- высокая скорость работы;
- надёжная защита от потерь данных;
- поддержка распределённой архитектуры;
- лёгкость в ветвлении и слиянии.

### 2.3 Ключевые особенности Git

- Каждый разработчик имеет полную копию репозитория (локально).
- Все изменения сохраняются в виде снапшотов.
- Ветвление и слияние — базовая часть рабочего процесса.
- Минимальная зависимость от центрального сервера.

Подробнее о философии Git — в книге Pro Git [3].

### 2.4 Основные команды Git

- `git init` — инициализация репозитория.
- `git clone` — копирование удалённого репозитория.
- `git status` — просмотр состояния файлов.
- `git add` — добавление файлов в индекс.
- `git commit` — фиксация изменений.
- `git branch` — работа с ветками.

- `git checkout` — переключение между ветками.
- `git merge` — слияние веток.
- `git pull`, `git push` — взаимодействие с удалёнными репозиториями.

Команды подробно описаны в официальной документации [2].

## 2.5 Источники и полезные ссылки

- [2] — официальная документация Git.
- [1] — платформа интерактивного обучения Learn Git Branching.
- [3] — Pro Git book.

## 2.6 Выводы для главы 2

Вторая глава была посвящена изучению системы контроля версий Git. Рассмотрены её архитектура, принципы работы и основные команды. Были объяснены понятия коммита, индексации, истории изменений, а также таких операций, как ветвление (branching), слияние (merging) и откат изменений (reset, revert) [2], [3].

Git продемонстрировал себя как надёжный инструмент для индивидуальной и командной разработки. Его использование позволяет систематизировать процесс разработки, минимизировать риски потери данных и упростить интеграцию изменений. Глава заложила основу для выполнения практических задач на платформе Learn Git Branching.

### III Практика с Learn Git Branching

В качестве обучающей платформы используется интерактивный тренажёр Learn Git Branching [1].

**Важно:** последующая работа подразумевает что читатель уже ознакомился с источниками [1]—[3].

#### 3.1 Main — Базовые концепции Git

##### 3.1.1 Introduction Sequence

##### Introduction to Git Commits

**Цель:** познакомиться с понятием коммита в Git.

Коммит в Git — это снимок состояния всех отслеживаемых файлов. Git старается хранить только изменения (дельты), чтобы минимизировать использование памяти и ускорить операции. Каждый коммит связан с предыдущим (родительским), что позволяет формировать полную историю проекта.

**Команда:**

```
1 git commit
```

фиг. 3.1 представлен скриншот данного задания.

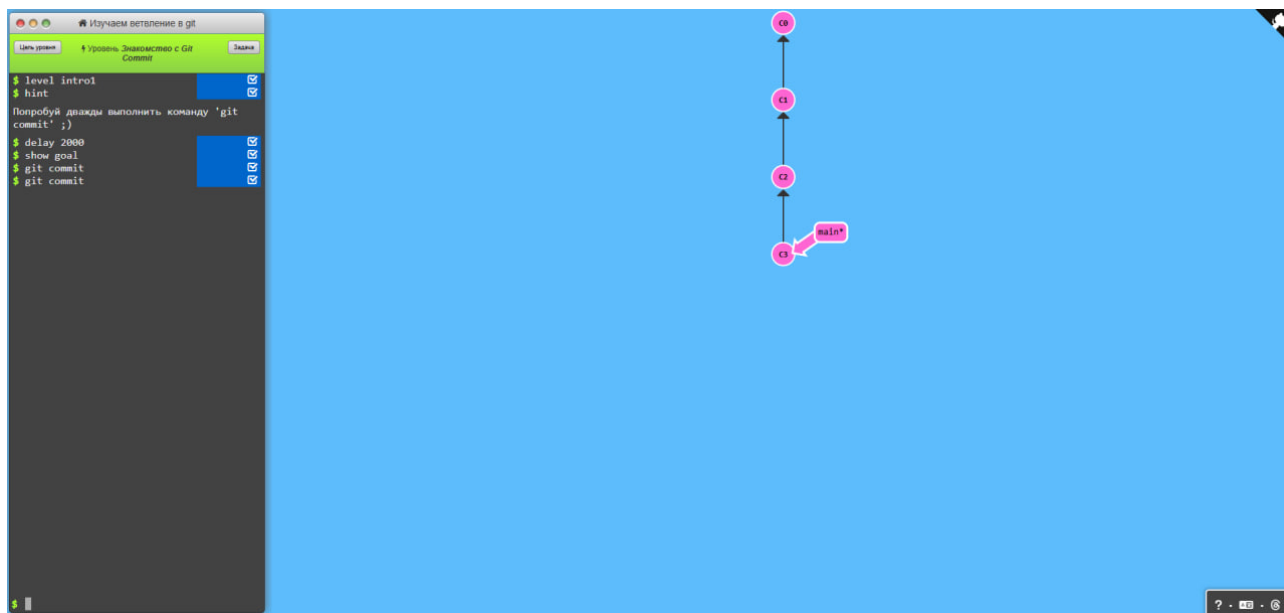


Рис. 3.1

После выполнения создаётся новый коммит, например C2, ссылающийся на C1 как на родителя.

## Branching in Git

**Цель:** научиться создавать и переключаться между ветками.

Ветки в Git – это указатели на определённые коммиты. Они практически не занимают места, поэтому создавать много веток – это нормально и даже рекомендуется.

### Команды:

```
1 git branch newImage # создание новой ветки
2 git checkout newImage # переключение на неё
3 git commit          # коммит будет в ветке newImage
```

### Альтернатива:

```
1 git checkout -b bugFix
```

фиг. 3.2 представлен скриншот данного задания.

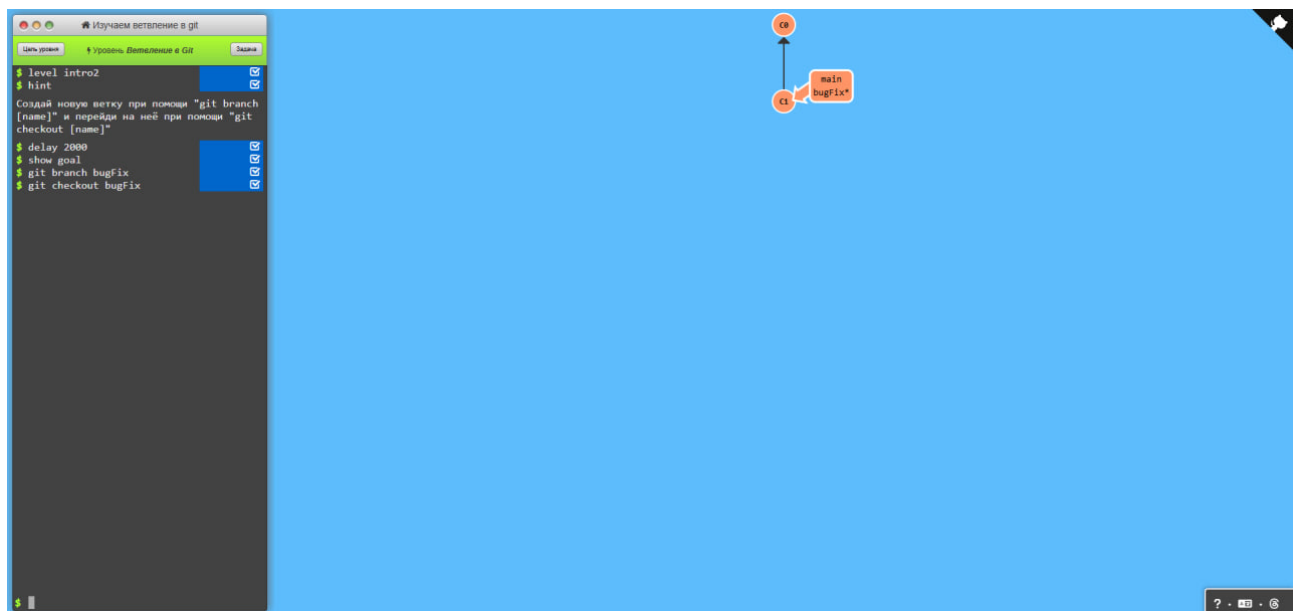


Рис. 3.2

**Важно:** коммиты записываются только в активную ветку (ту, где стоит символ \*).

## Merging in Git

**Цель:** объединить работу из разных веток в одну.

Merge создаёт специальный коммит с двумя родителями, включающий изменения из обеих веток. Это особенно полезно для совместной работы.

### Команды:

```
1 git merge bugFix      # слияние ветки bugFix в текущую (main)
2 git checkout bugFix   # переход на другую ветку
3 git merge main        # слияние main в bugFix
```

фиг. 3.3 представлен скриншот данного задания.

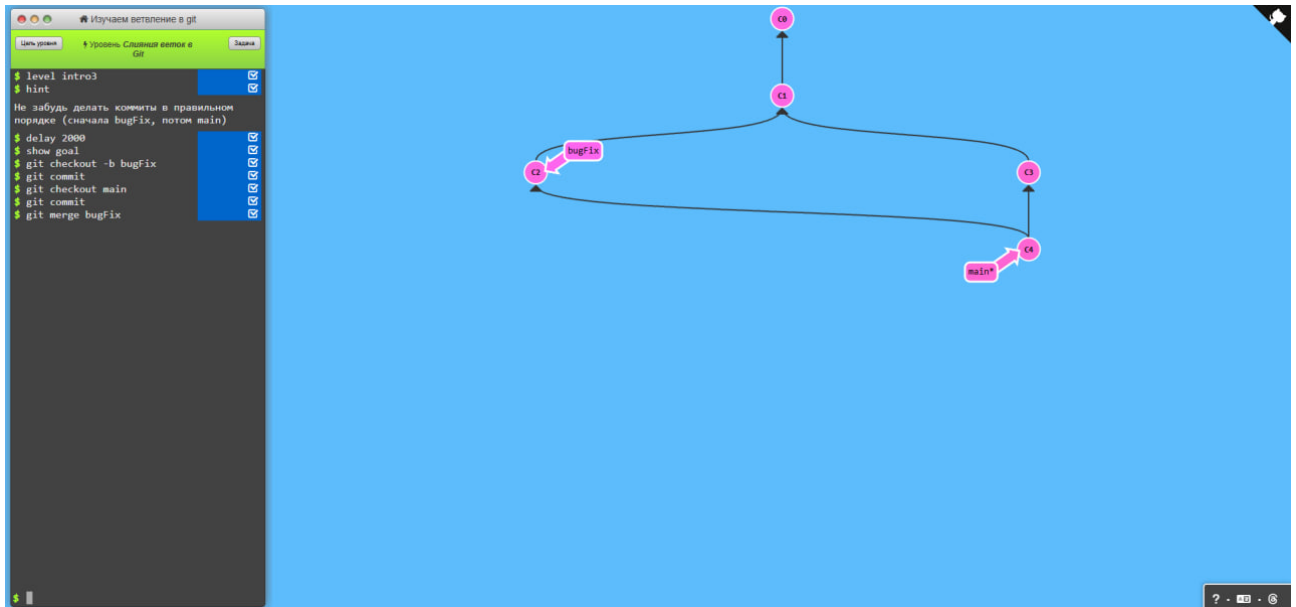


Рис. 3.3

После слияния все ветки включают весь набор коммитов проекта.

## Rebase Introduction

**Цель:** познакомиться с альтернативным способом объединения веток.

Rebase — это перемещение коммитов из одной ветки на основание другой. Это делает историю проекта линейной и упрощает её анализ. Однако следует использовать осторожно, особенно при работе с общими ветками.

### Команды:

- 1 `git rebase main` *# переместить коммиты bugFix поверх main*
- 2 `git checkout main`
- 3 `git rebase bugFix` *# переместить main после bugFix*

фиг. 3.4 представлен скриншот данного задания.



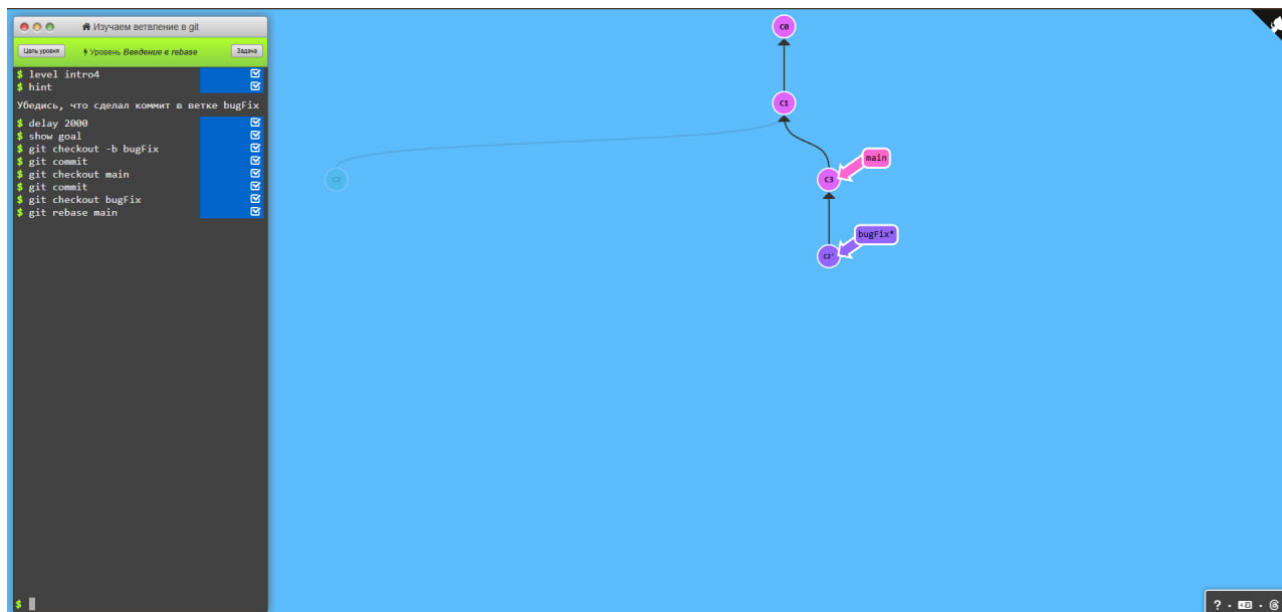


Рис. 3.4

**Важно:** Rebase изменяет историю коммитов и создаёт новые идентификаторы для ”перемещённых” коммитов.

Дополнительную информацию о командах и концепциях Git можно найти в документации и книге Pro Git.

### 3.1.2 Ramping Up

#### Detach yo’ HEAD

**Цель:** понять, как работает HEAD и что означает его ”отсоединение”.

В Git переменная HEAD указывает на текущий коммит или ветку, на которой вы находитесь. Обычно она указывает на имя ветки (например, main), но можно напрямую привязать HEAD к коммиту.

Это называется ”отсоединённым HEAD” (detached HEAD), и в этом режиме коммиты не записываются ни в какую ветку, пока не будет выполнено явно указание.

**Команды:**

```
1 git checkout C1          # переход на конкретный коммит
```

**Результат:** HEAD указывает напрямую на коммит C1, а не на ветку.

**Практика:** Проверьте команды:

```
1 git checkout main
2 git commit
```

Затем отсоедините HEAD:

```
1 git checkout C2
```

фиг. 3.5 представлен скриншот данного задания.

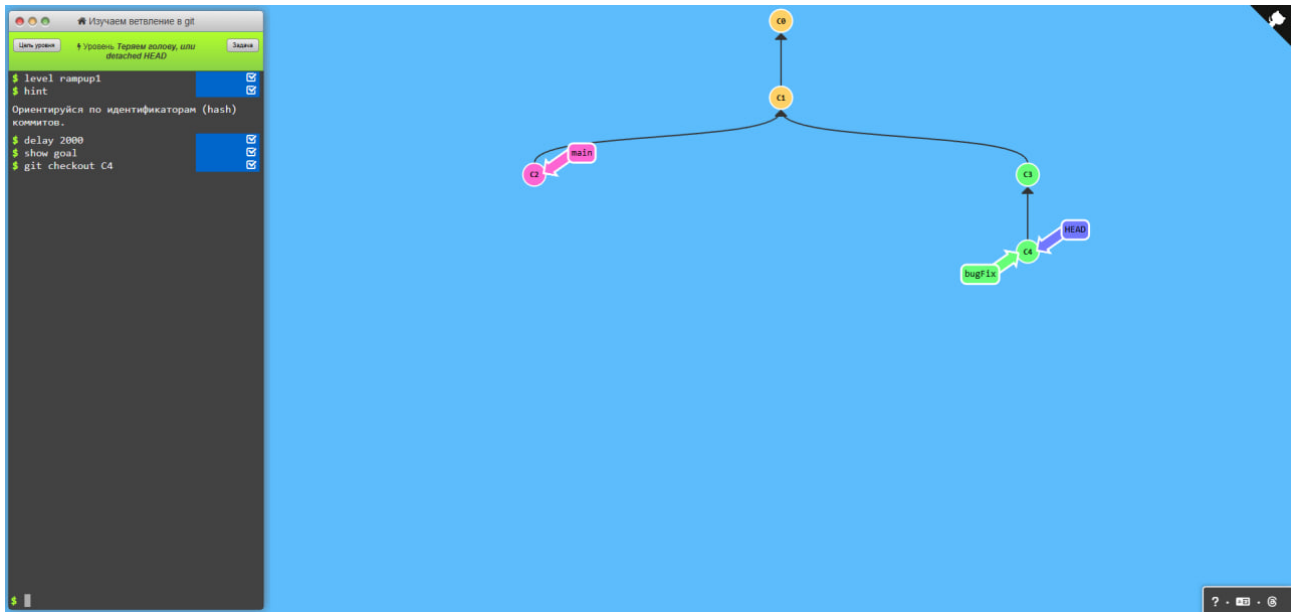


Рис. 3.5

## Relative Refs (^)

**Цель:** освоить навигацию по коммитам с помощью относительных ссылок.

Вместо указания полного хеша коммита, Git позволяет перемещаться по дереву коммитов с помощью символов ^ и ~:

- `main^` — родитель `main`; - `main^^` — дедушка; - `HEAD^` — один шаг назад от текущего коммита.

### Команды:

- 1 `git checkout main^`
- 2 `git checkout HEAD^`
- 3 `git checkout HEAD^`

фиг. 3.6 представлен скриншот данного задания.

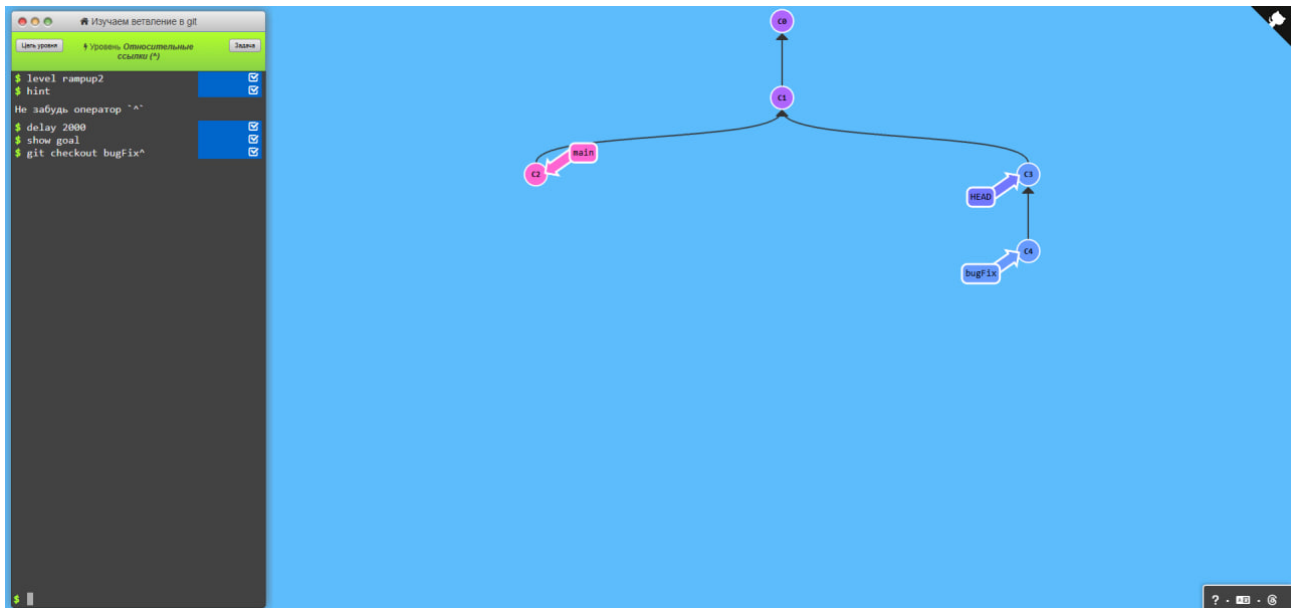


Рис. 3.6

**Результат:** Вы перемещаетесь назад по истории коммитов.

## Relative Refs (~)

**Цель:** быстро перемещаться назад по истории на несколько шагов.

HEAD~3 эквивалентно HEAD^^^, но значительно короче.

### Команды:

```
1 git checkout HEAD~4
```

Можно использовать это в связке с -f (force) для перемещения веток:

```
1 git branch -f main HEAD~3
```

фиг. 3.7 представлен скриншот данного задания.

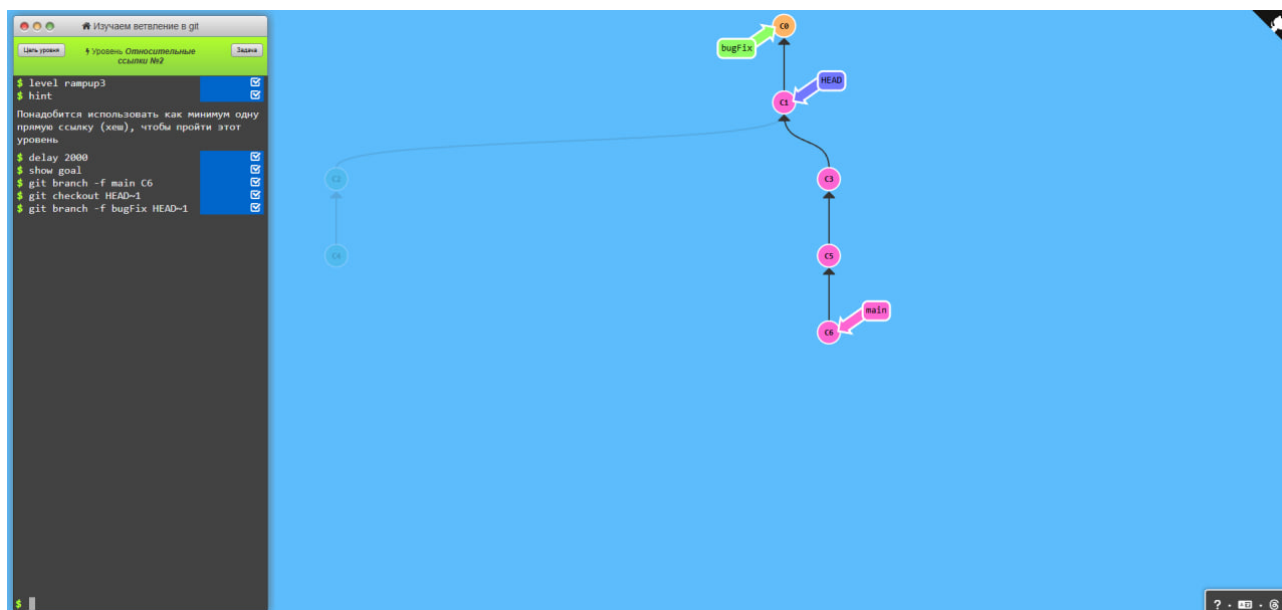


Рис. 3.7. Скиншот задания

**Примечание:** В реальных условиях запрещено перемещать текущую ветку таким способом — сначала нужно переключиться на другую.

## Reversing Changes in Git

**Цель:** научиться отменять изменения с помощью `reset` и `revert`.

**git reset** — откат ветки на предыдущий коммит (используется для локальной работы).

```
1 git reset HEAD~1
```

**git revert** — создаёт новый коммит, отменяющий предыдущий (удобно для публичных веток).

```
1 git revert HEAD
```

фиг. 3.8 представлен скриншот данного задания.

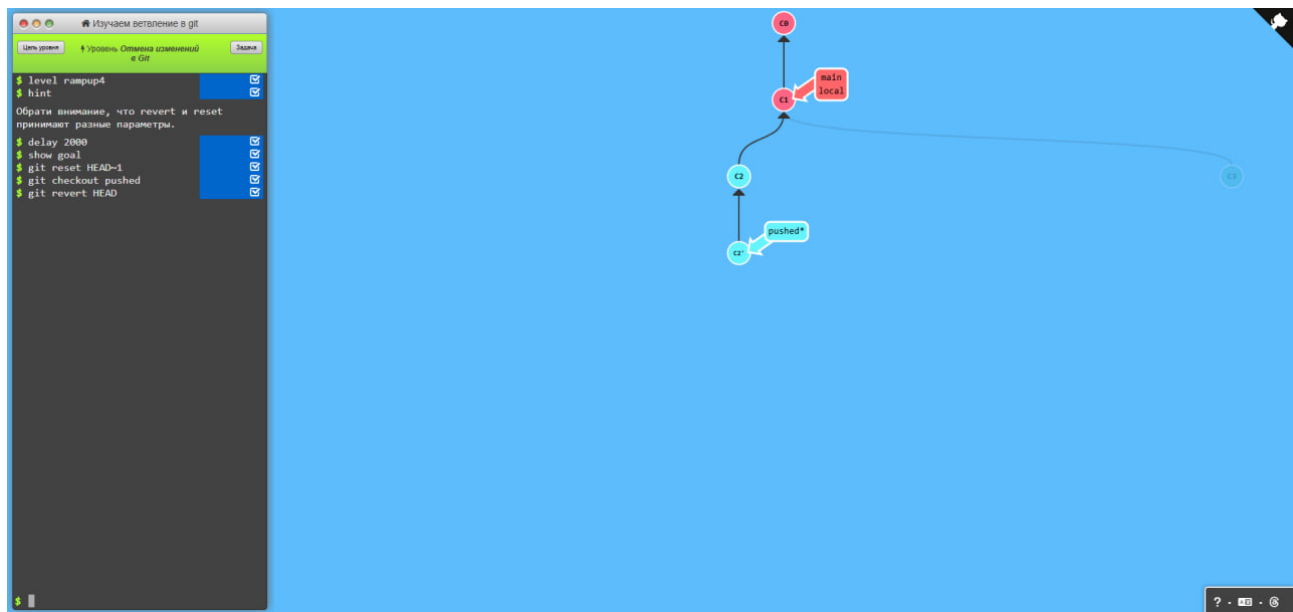


Рис. 3.8. Скиншот задания

**Вывод:** Reset переписывает историю. Revert создаёт откат как отдельный коммит и безопасен для совместной работы.

### 3.1.3 Moving Work Around

#### Cherry-pick Intro

**Цель:** научиться переносить отдельные коммиты между ветками.

Команда `git cherry-pick` позволяет выбрать один (или несколько) коммитов и ”применить” их в текущую ветку. Это удобно, когда нужно скопировать конкретные изменения, не сливая всю ветку целиком.

**Сценарий:** допустим, нужный коммит есть в ветке `feature`, но мы хотим перенести его в `main` без `merge` или `rebase`.

**Команда:**

```
1 git cherry-pick <hash-коммита>
```

**Пример:**

```
1 git checkout main
2 git cherry-pick C4
```

фиг. 3.9 представлен скриншот данного задания.

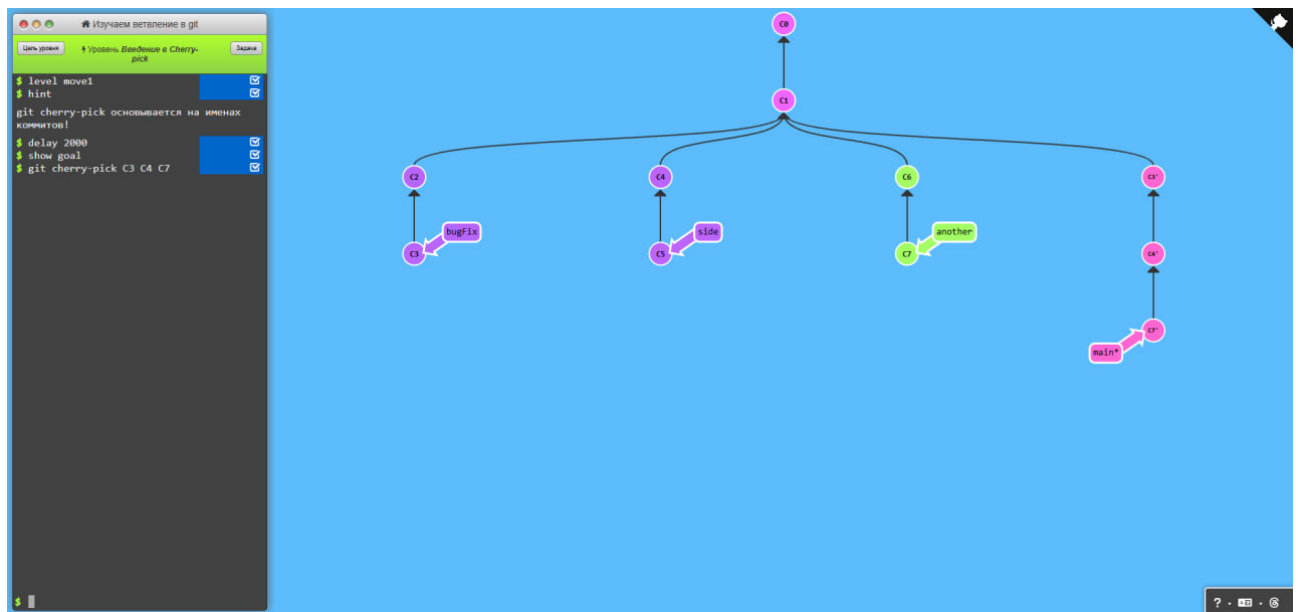


Рис. 3.9

**Результат:** Ветка main получит копию коммита C4.

**Важно:** коммит получает новый hash (так как история изменяется).

## Interactive Rebase Intro

**Цель:** познакомиться с интерактивной перезаписью истории коммитов.

`git rebase -i` позволяет:

- изменить порядок коммитов;
- объединить коммиты (squash);
- удалить ненужные коммиты;
- изменить сообщения к коммитам.

**Команда:**

```
1 git rebase -i HEAD~3
```

После запуска откроется список последних 3 коммитов. В интерактивном режиме можно выбрать действия: `pick`, `reword`, `edit`, `squash`, `drop`.

**Пример:** чтобы объединить два последних коммита:

```
1 pick 1a2b3c Первый коммит
2 squash 4d5e6f Второй коммит
```

фиг. 3.10 представлен скриншот данного задания.

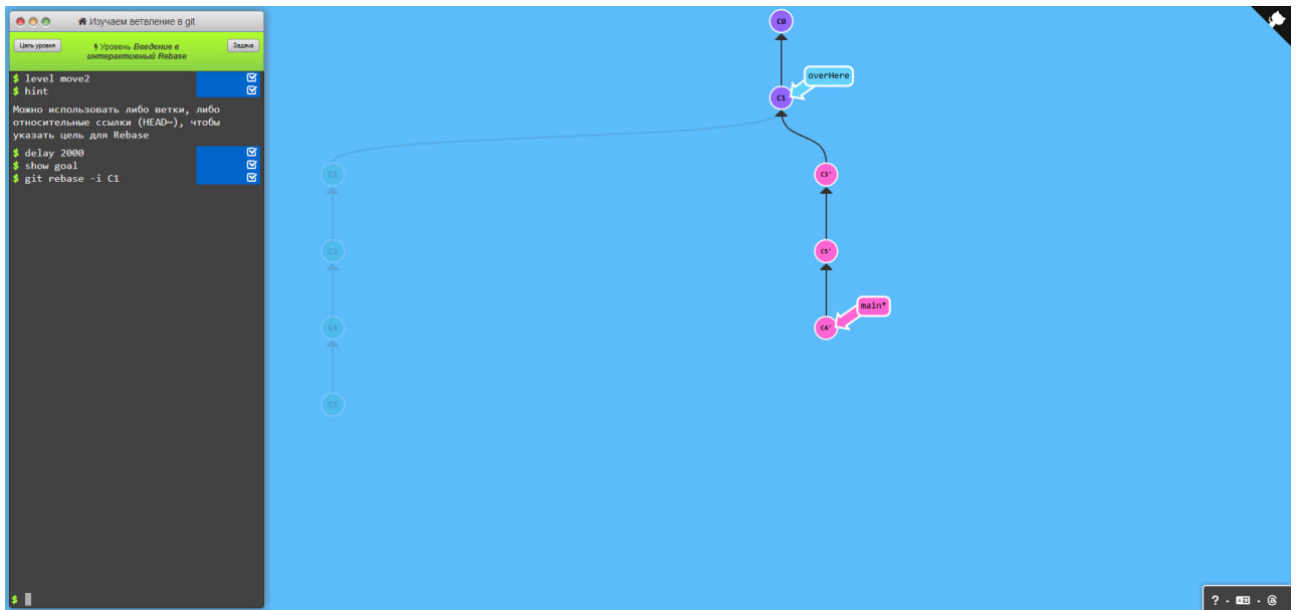


Рис. 3.10

**Результат:** оба коммита объединятся в один.

**Важно:** интерактивный rebase лучше использовать только в локальных ветках, до публикации в удалённый репозиторий.

### 3.1.4 A Mixed Bag

#### Grabbing Just 1 Commit

**Цель:** изолированно скопировать один конкретный коммит из другой ветки.

Иногда необходимо взять только один коммит из другой ветки, не объединяя всё её содержимое. Это решается с помощью `git cherry-pick`.

**Команды:**

- 1 `git checkout main`
- 2 `git cherry-pick <hash-коммита>`

фиг. 3.11 представлен скриншот данного задания.







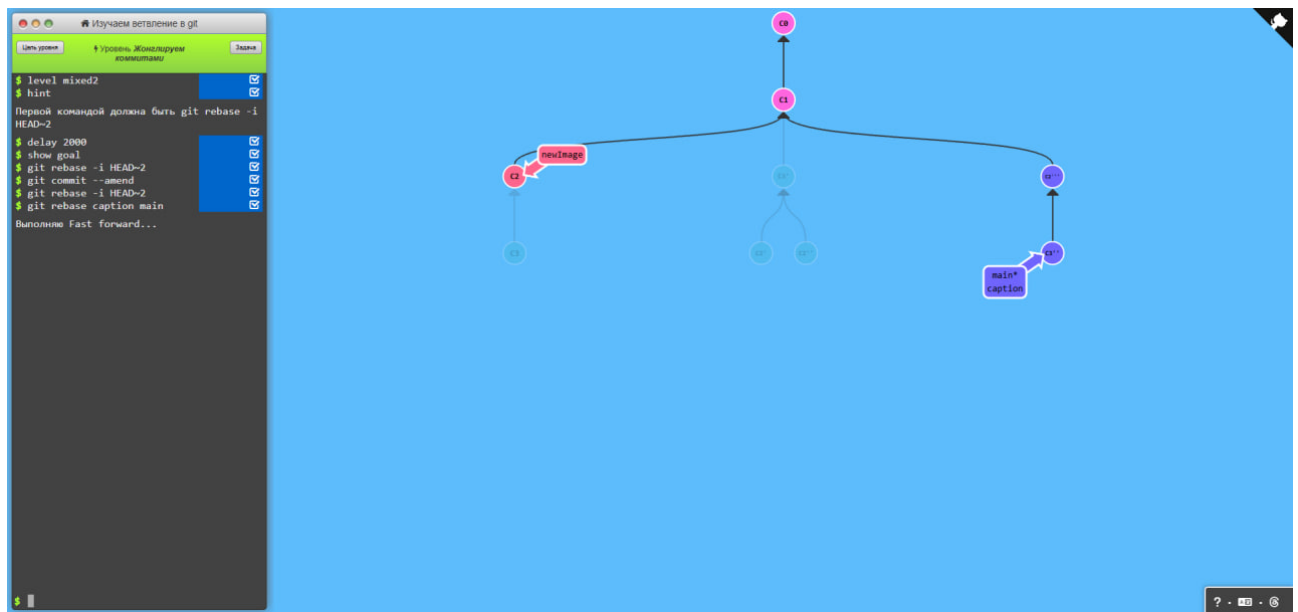


Рис. 3.13

**Результат:** создаётся нужная ветка на базе старого коммита с новым изменением.

## Git Tags

**Цель:** познакомиться с тегами в Git и их созданием.

Тег (tag) — это постоянная метка на конкретном коммите. Используется для пометки релизов, важных версий и контрольных точек.

**Команда:**

```
1 git tag v1 C1
```

фиг. 3.14 представлен скриншот данного задания.

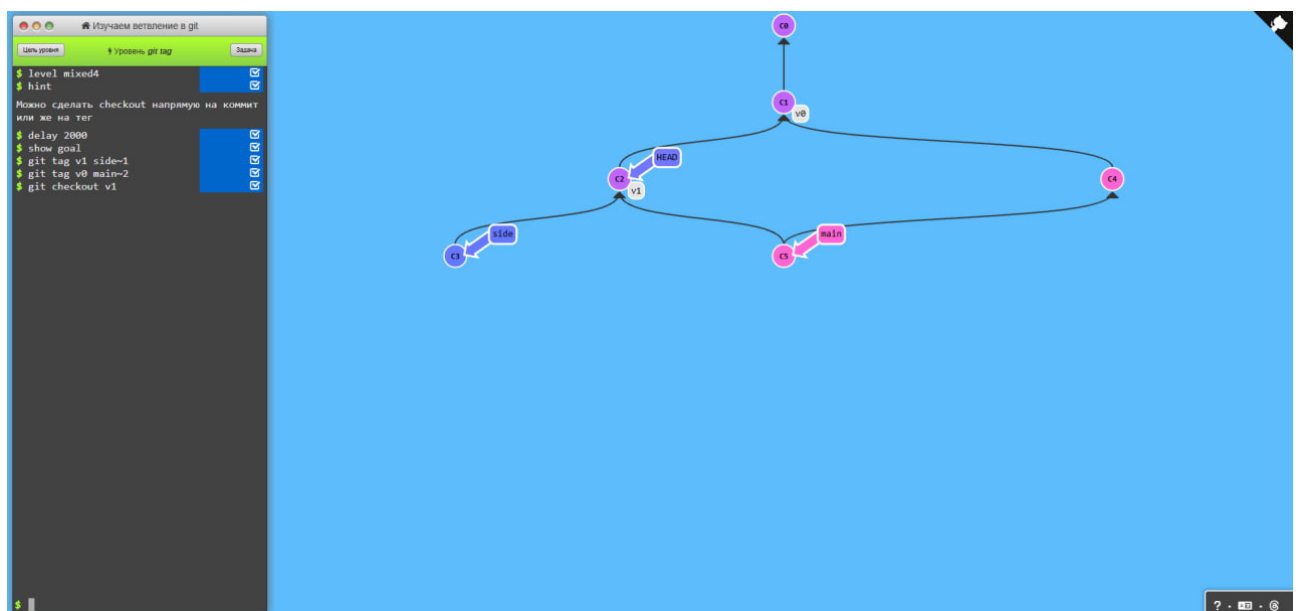


Рис. 3.14

**Примечание:** в отличие от ветки, тег не перемещается при новых коммитах.

## Git Describe

**Цель:** научиться использовать `git describe` для понимания положения HEAD относительно тегов.

`git describe` показывает ближайший тег и расстояние (число коммитов) от него до текущего состояния ветки.

**Команда:**

```
1 git describe
```

**Пример вывода:**

```
1 v1-2-gabcdef
```

фиг. 3.15 представлен скриншот данного задания.

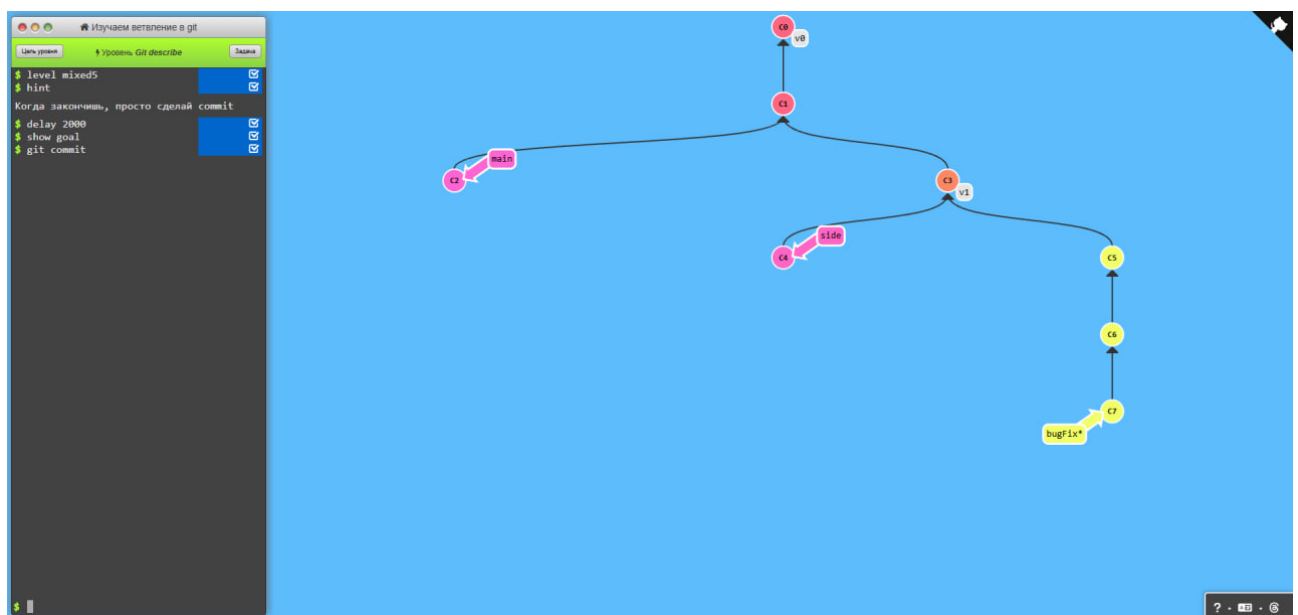


Рис. 3.15

v1 — ближайший тег, 2 — количество коммитов после тега, gabcdef — hash текущего коммита.

### 3.1.5 Advanced Topics

#### Rebasing over 9000 times

**Цель:** повторно закрепить использование `git rebase` и научиться выстраивать линейную историю при множественных ветках.

При сложной структуре ветвлений и параллельных разработках часто необходимо ”переносить” изменения с ветки на ветку, чтобы сохранить читаемость истории и избежать конфликтов при слиянии.

### Команды:

```
1 git rebase main
```

фиг. 3.16 представлен скриншот данного задания.

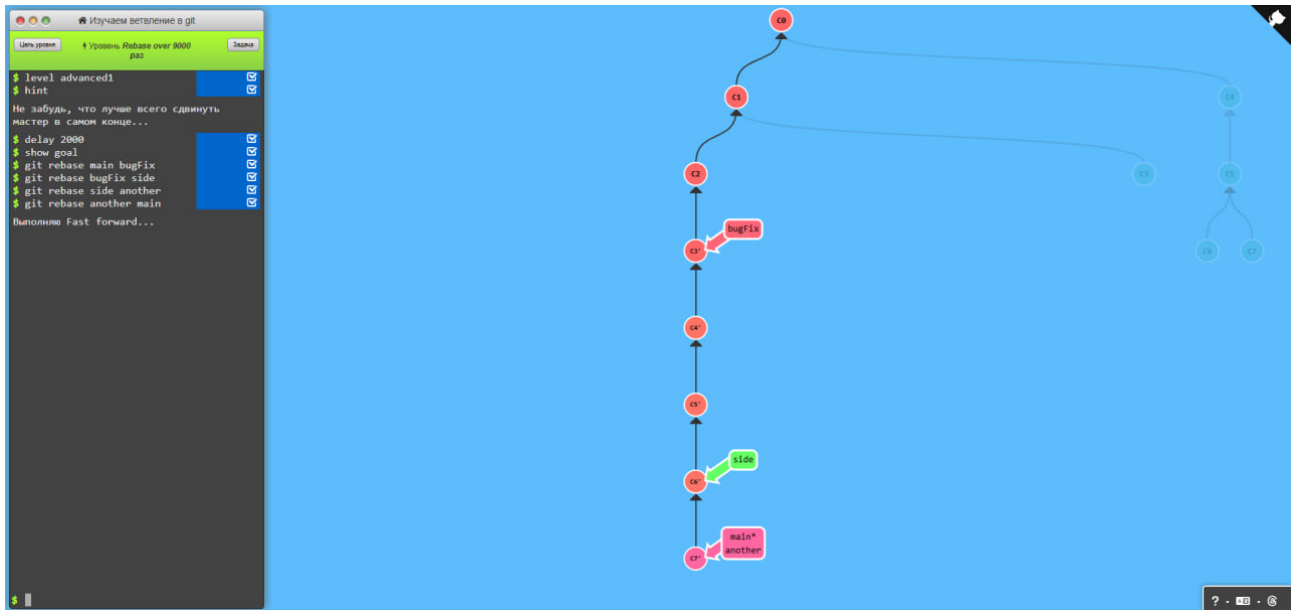


Рис. 3.16

**Примечание:** важно выполнять rebase с осознанием того, что вы меняете историю коммитов. Не используйте на общедоступных ветках.

## Multiple Parents

**Цель:** изучить коммиты с несколькими родителями (merge-коммиты).

Git позволяет объединять несколько веток, создавая коммит с двумя и более родителями. Это удобно, но влечёт за собой визуальную сложность истории.

### Команда:

```
1 git merge feature1
```

фиг. 3.17 представлен скриншот данного задания.

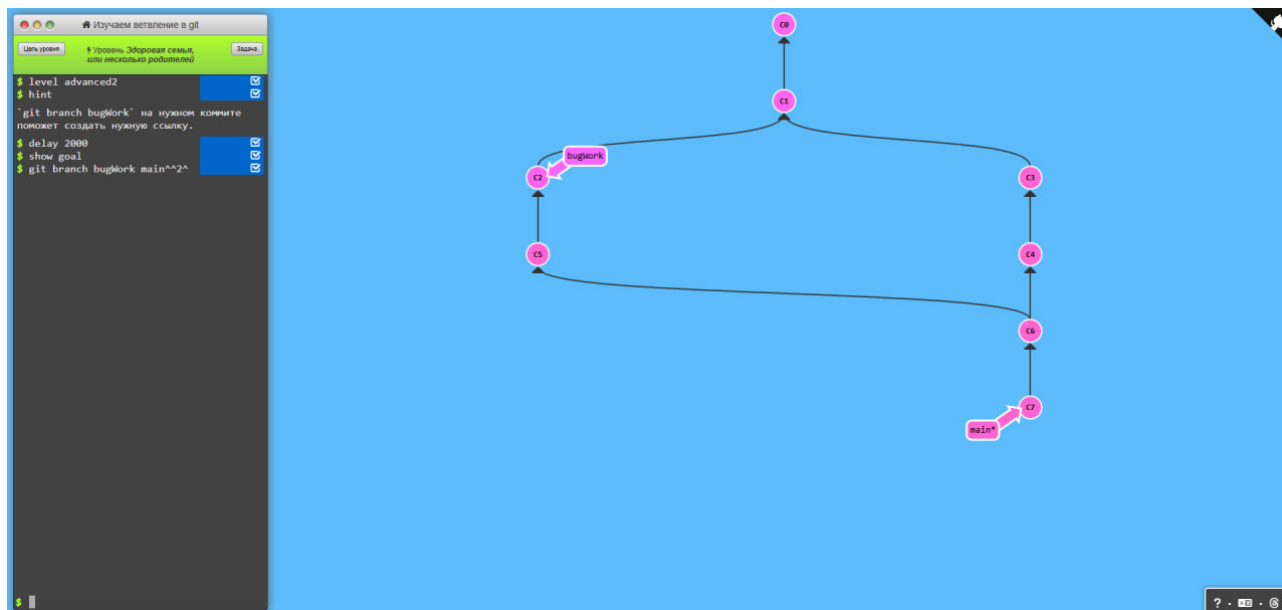


Рис. 3.17

**Пример:** если обе ветки имеют уникальные коммиты, результатом будет коммит с двумя родителями —  $C4 = \text{merge}(C2, C3)$ .

## Branch Spaghetti

**Цель:** визуализировать и понять последствия хаотичной работы с ветками.

Когда разработчики ветвятся без правил и контроля, история превращается в "спагетти" — множество пересечений, нестабильность и путаница.

Цель уровня — выявить, как избежать этой ситуации и как можно аккуратно провести слияния или rebase для восстановления структуры.

### Рекомендации:

- Используйте feature-ветки.
- Делайте squash перед merge.
- Поддерживайте линейную историю на основной ветке.

фиг. 3.18 представлен скриншот данного задания.

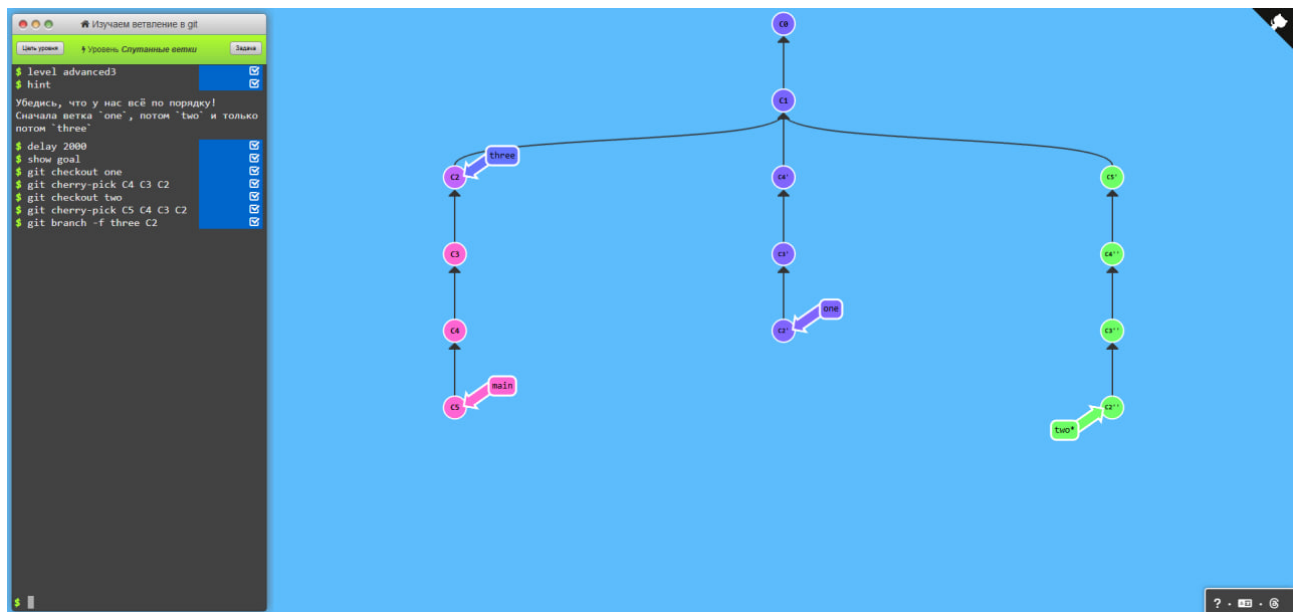


Рис. 3.18

## 3.2 Remote — Работа с удалёнными репозиториями

### 3.2.1 Push & Pull — Git Remotes

#### Clone Intro

**Цель:** понять суть удалённого репозитория и создать его копию локально.

Команда `git clone` создаёт локальную копию удалённого репозитория. В рамках LearnGitBranching она используется для создания виртуального удалённого репозитория и отображения его структуры.

**Команда:**

```
1 git clone
```

фиг. 3.19 представлен скриншот данного задания.

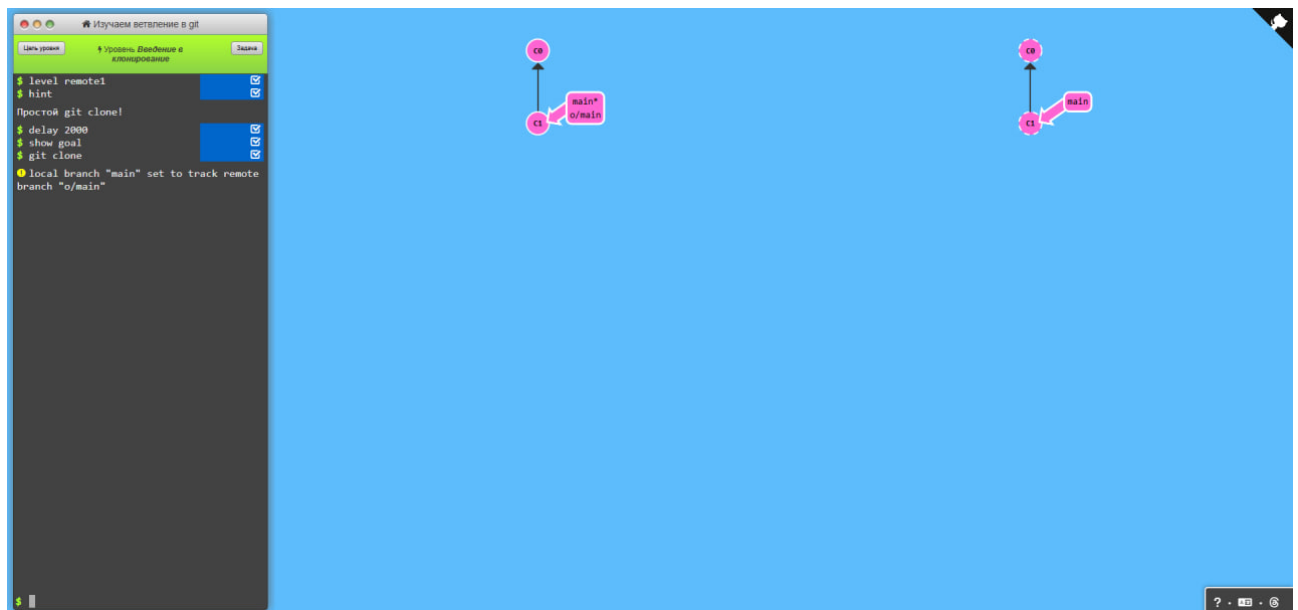


Рис. 3.19

**Результат:** появляется удалённый репозиторий (remote), связанный с текущим.

## Remote Branches

**Цель:** изучить поведение удалённых веток.

Удалённые ветки (например, o/main) отражают состояние удалённого репозитория. Они обновляются после fetch/pull и не могут быть изменены напрямую.

**Пример поведения:**

- 1 git checkout o/main
- 2 git commit

фиг. 3.20 представлен скриншот данного задания.

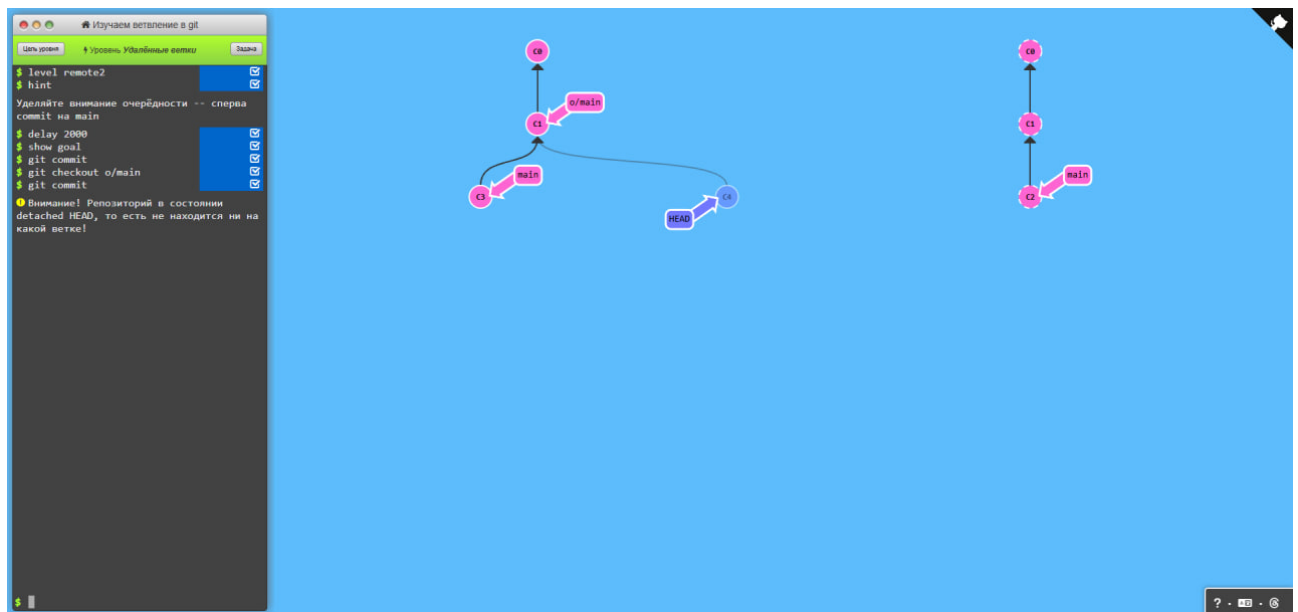


Рис. 3.20

**Результат:** создаётся коммит, но он не привязывается к ветке o/main, так как HEAD отсоединён.

## Git Fetchin'

**Цель:** получить изменения с удалённого репозитория без их применения.

git fetch загружает изменения и обновляет удалённые ветки, но не изменяет локальные.

**Команда:**

```
1 git fetch
```

**Поведение:**

- загружает недостающие коммиты;
- обновляет o/main (и др.);
- не влияет на ваш рабочий каталог.

фиг. 3.21 представлен скриншот данного задания.





**Результат:** локальная ветка будет дополнена новыми коммитами из o/main.

## Faking Teamwork

**Цель:** смоделировать параллельную работу нескольких разработчиков.

В этом упражнении мы тренируемся в синхронизации изменений между локальной и удалённой ветками. Один разработчик делает коммит локально, другой — на удалённом.

### Команды:

- 1 `git commit` *# локальный коммит*
- 2 `git fetch` *# подтянуть удалённый*

фиг. 3.23 представлен скриншот данного задания.

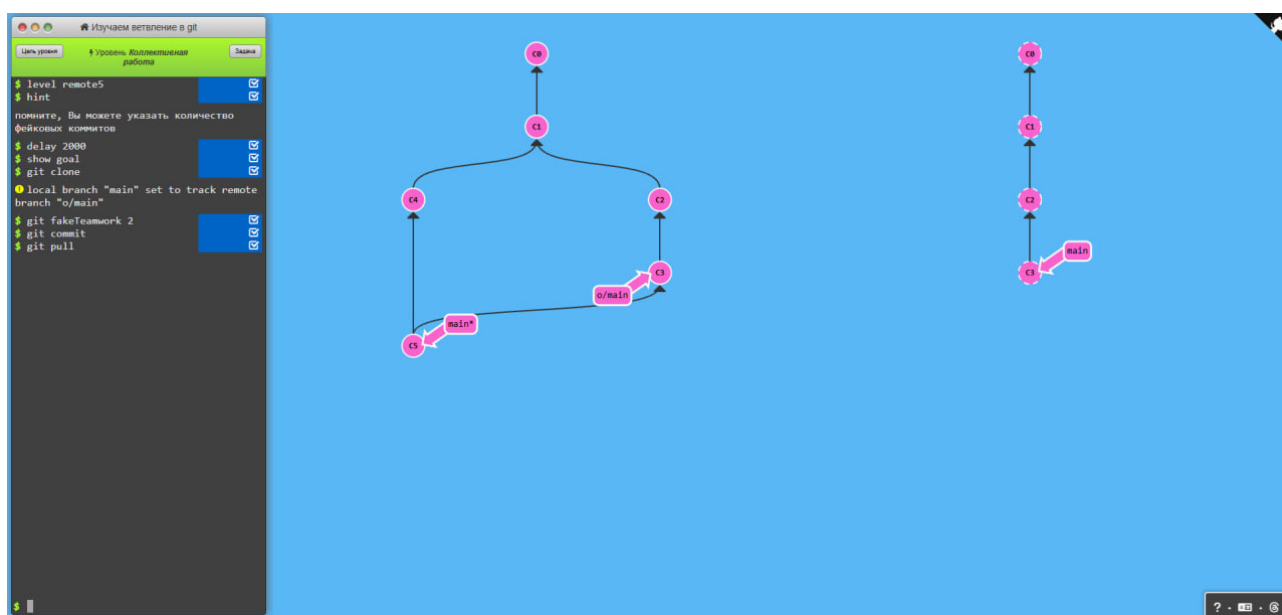


Рис. 3.23

**Задача:** объединить обе версии через merge или rebase.

## Git Pushin'

**Цель:** передать изменения в удалённый репозиторий.

`git push` отправляет ваши локальные коммиты в удалённую ветку. Только если история не расходитесь.

### Команда:

- 1 `git push`

фиг. 3.24 представлен скриншот данного задания.

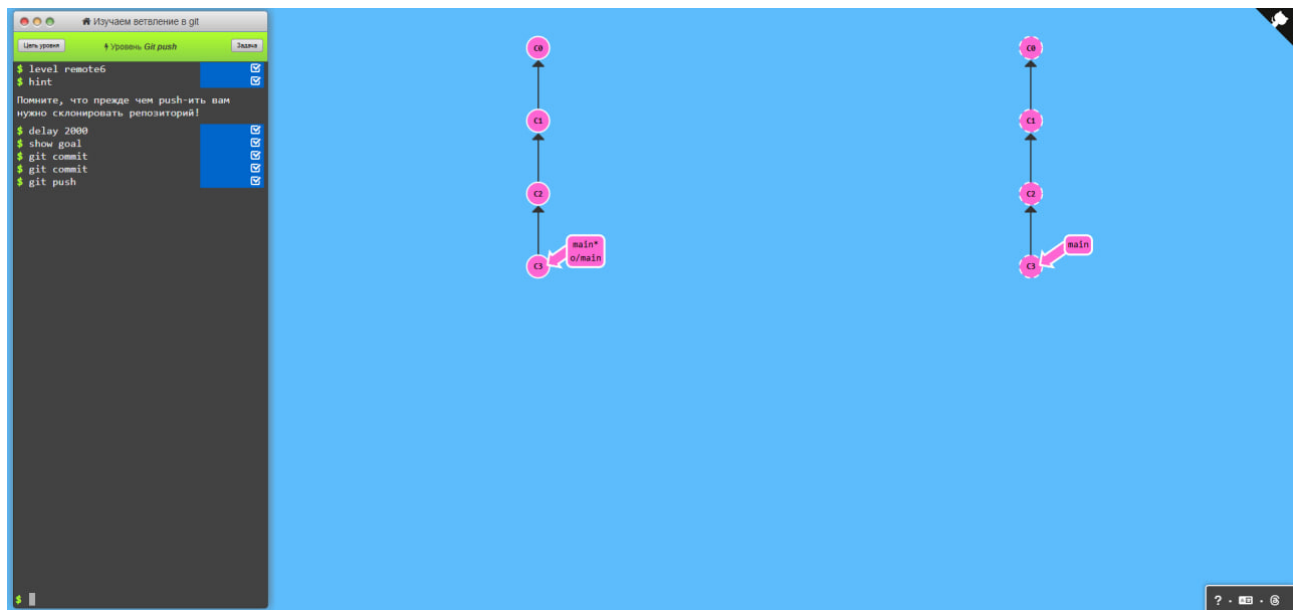


Рис. 3.24

**Важно:** если история различается, push будет отклонён.

## Diverged History

**Цель:** разрешить конфликты при расхождении истории.

Когда локальная и удалённая ветки имеют разные изменения, Git требует ручного вмешательства. Нужно выполнить:

### Команды:

```
1 git pull --rebase      # или git fetch + rebase
```

**После:** можно безопасно выполнить `git push`.

фиг. 3.25 представлен скриншот данного задания.

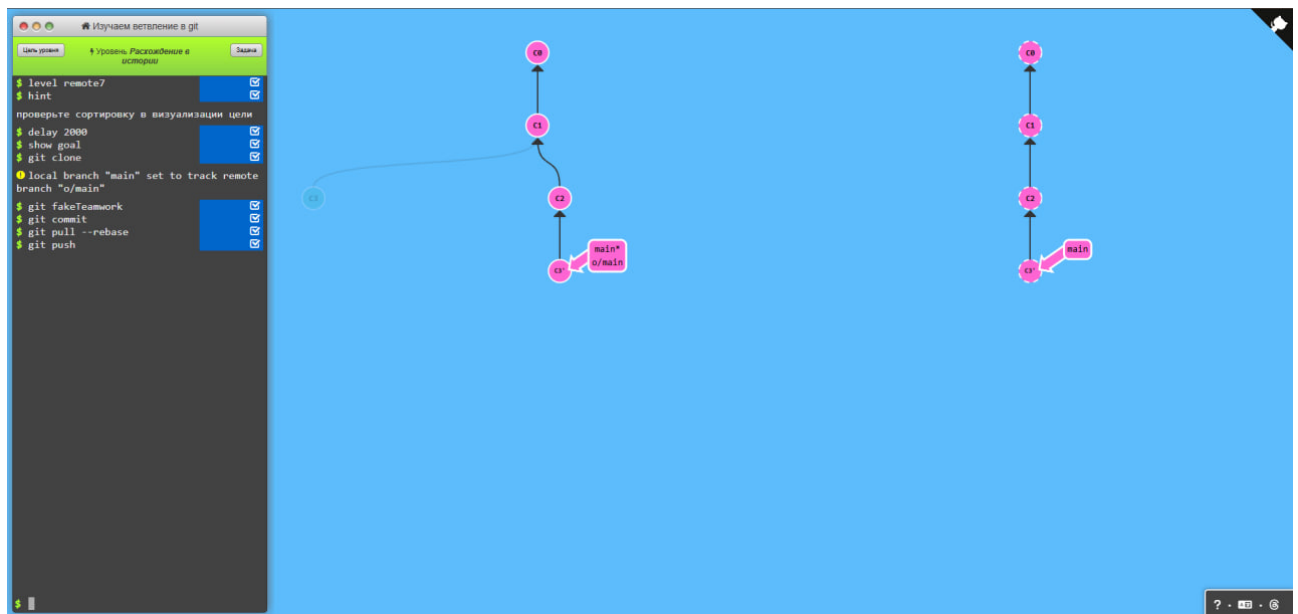


Рис. 3.25

## Locked Main

**Цель:** изучить ситуацию, когда push запрещён без обновления локальной истории.

Некоторые удалённые репозитории (например, GitHub) запрещают push, если локальная ветка не включает последние изменения.

### Решение:

- 1 `git pull --rebase`

фиг. 3.26 представлен скриншот данного задания.

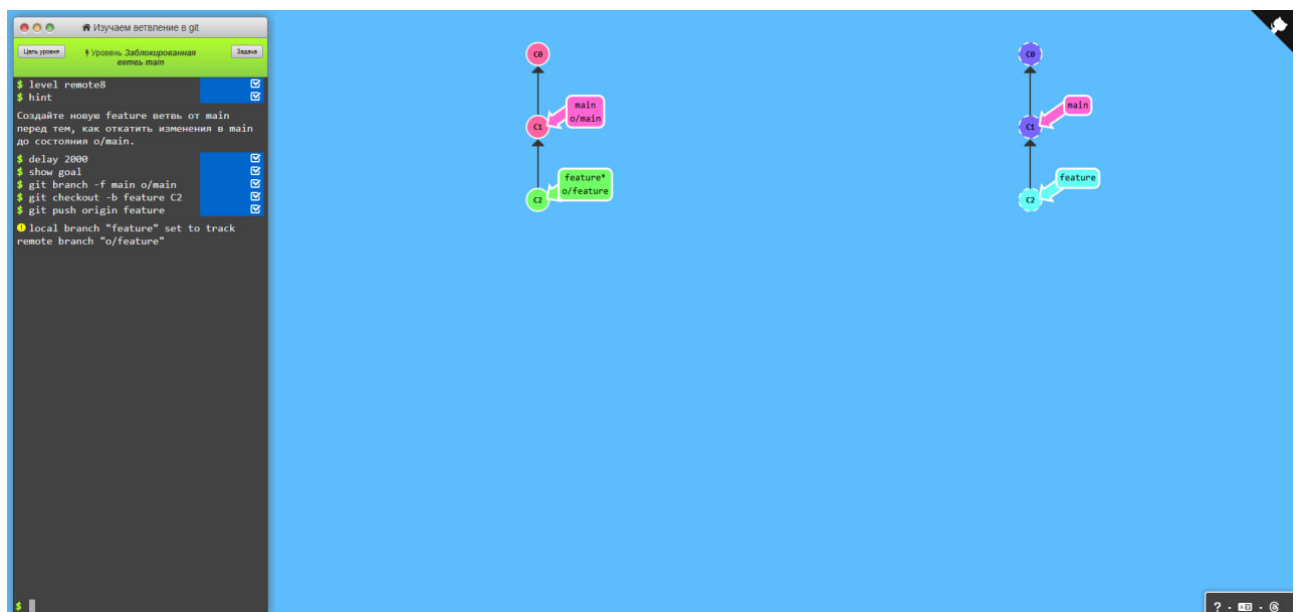


Рис. 3.26

**Результат:** локальная ветка обновляется и вы можете push без ошибок.

### 3.2.2 To Origin and Beyond — Advanced Git Remotes

#### Push Main!

**Цель:** закрепить основную команду для публикации ветки.

Задание фокусируется на использовании `git push` для основной ветки (`main`).

**Команда:**

```
1 git push origin main
```

фиг. 3.27 представлен скриншот данного задания.

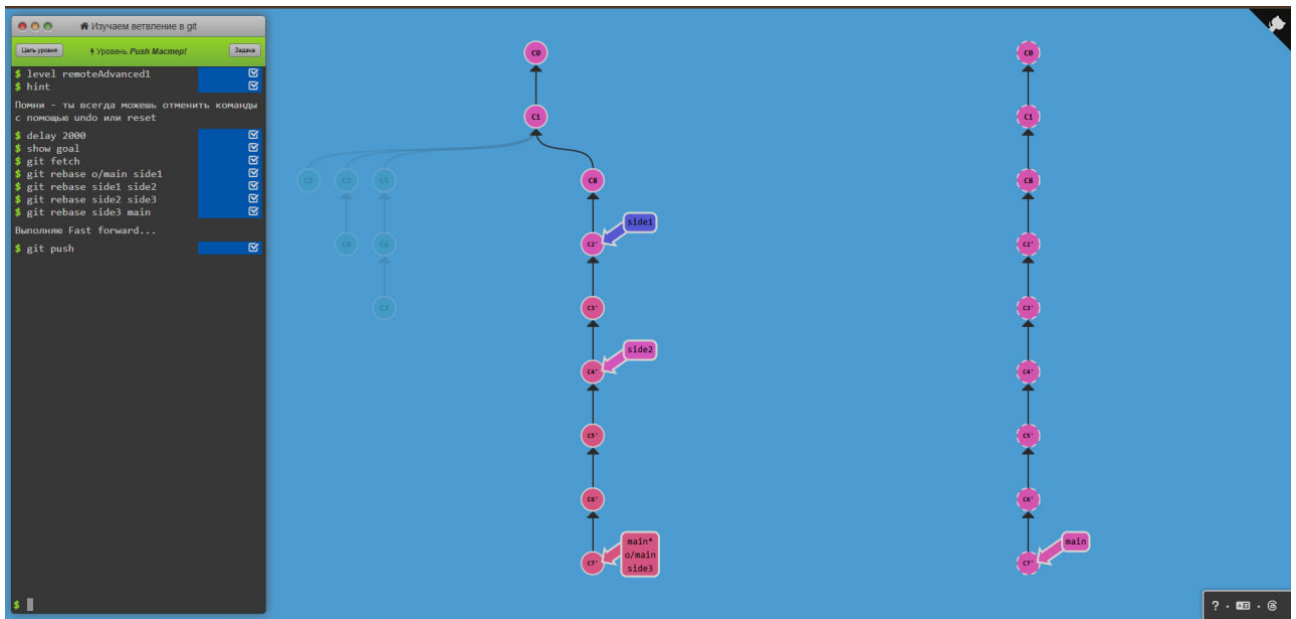


Рис. 3.27

**Примечание:** при явном указании ветки повышается контроль.

#### Merging with remotes

**Цель:** объединить удалённые изменения с локальными.

В случае, когда на удалённой стороне появились коммиты, которых нет у вас, нужно сначала их подтянуть:

```
1 git pull --rebase
```

Затем можно безопасно делать:

```
1 git push
```

фиг. 3.28 представлен скриншот данного задания.

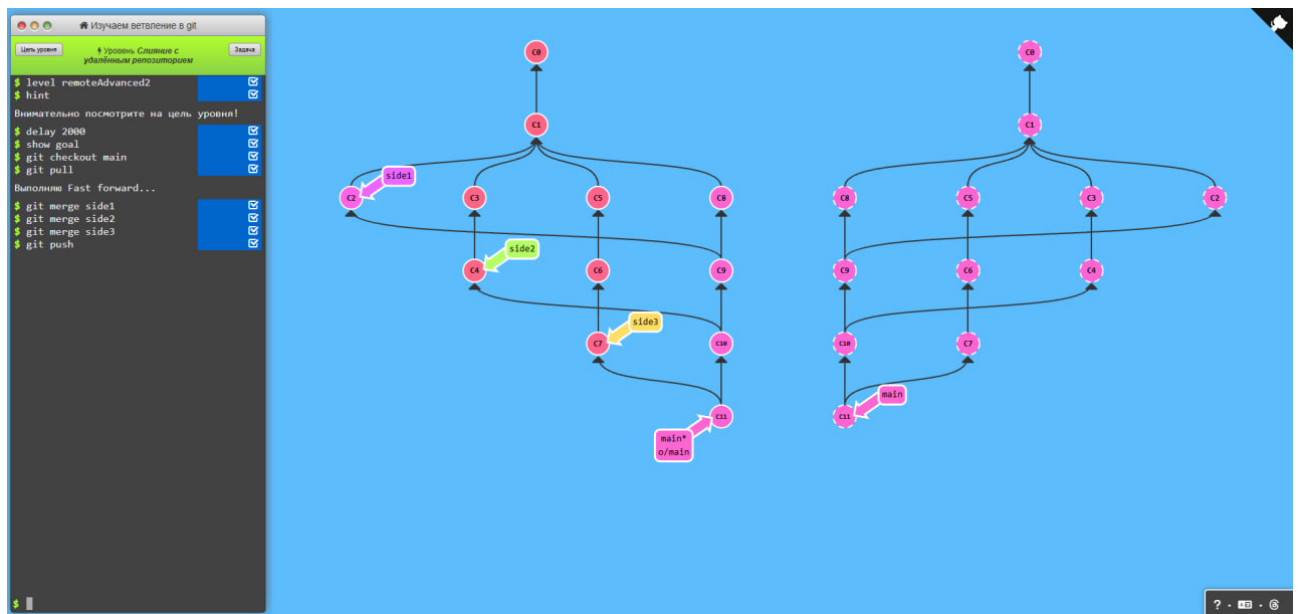


Рис. 3.28

## Remote Tracking

**Цель:** понять, как ветки отслеживают друг друга.

При клонировании создаются локальные ветки, "отслеживающие" удалённые:

```
1 git branch -vv
```

фиг. 3.29 представлен скриншот данного задания.

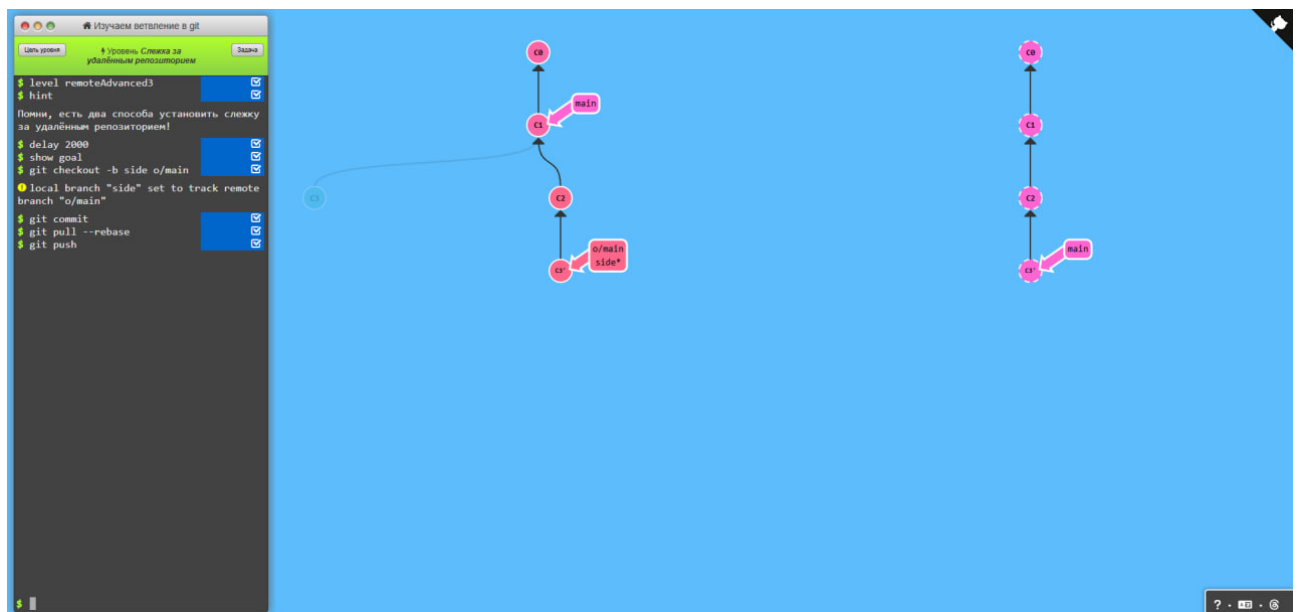


Рис. 3.29

**Вывод:** вы увидите, какие ветки связаны с удалёнными и на сколько коммитов они отстают или опережают.

## Git push arguments

**Цель:** научиться задавать явно, что и куда отправляется.

**Формат:**

```
1 git push <remote> <source>:<destination>
```

**Пример:**

```
1 git push origin bugFix:main
```

Это отправит ветку bugFix в удалённую main.

фиг. 3.30 представлен скриншот данного задания.

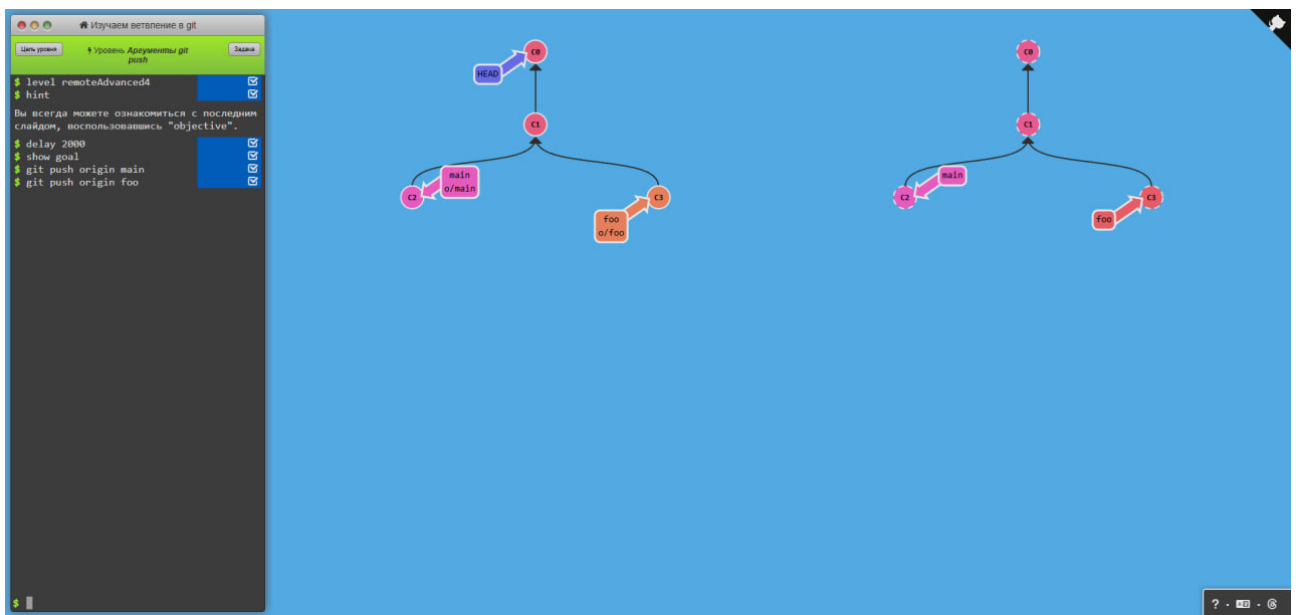


Рис. 3.30

## Git push arguments – Expanded!

**Цель:** управлять историей удалённого репозитория.

**Удаление ветки с сервера:**

```
1 git push origin :feature1
```

фиг. 3.31 представлен скриншот данного задания.

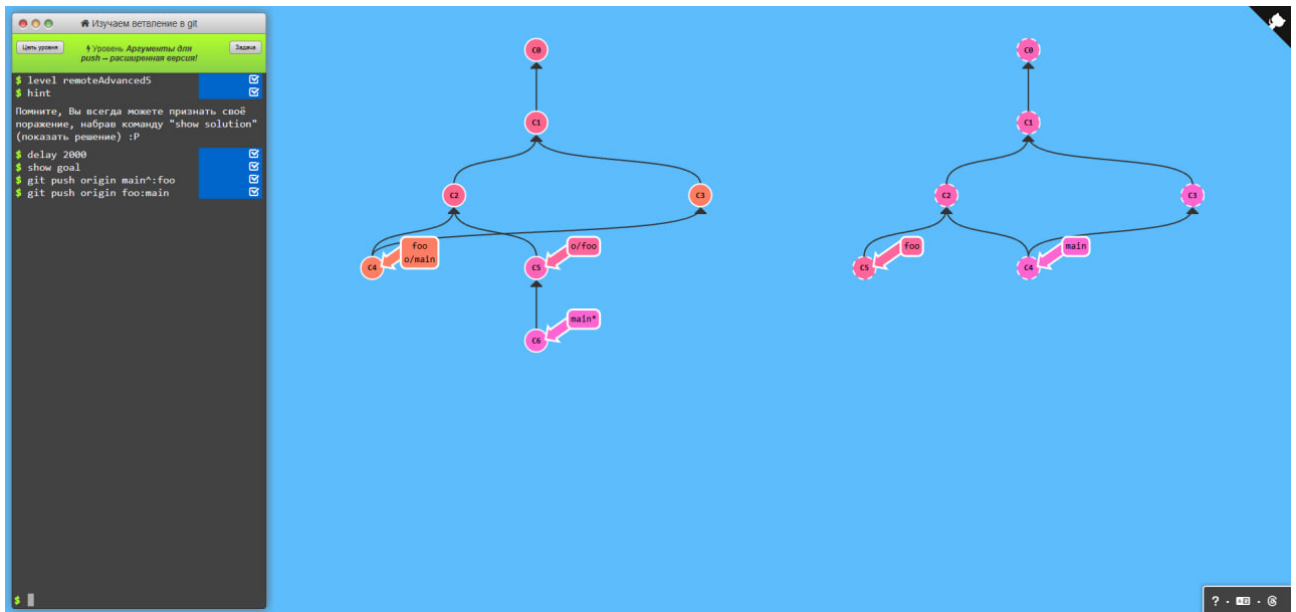


Рис. 3.31

**Результат:** удалённая ветка feature1 будет удалена.

## Fetch arguments

**Цель:** частично обновлять удалённые данные.

Можно получать не все изменения, а только конкретные ветки:

```
1 git fetch origin bugFix
```

фиг. 3.32 представлен скриншот данного задания.

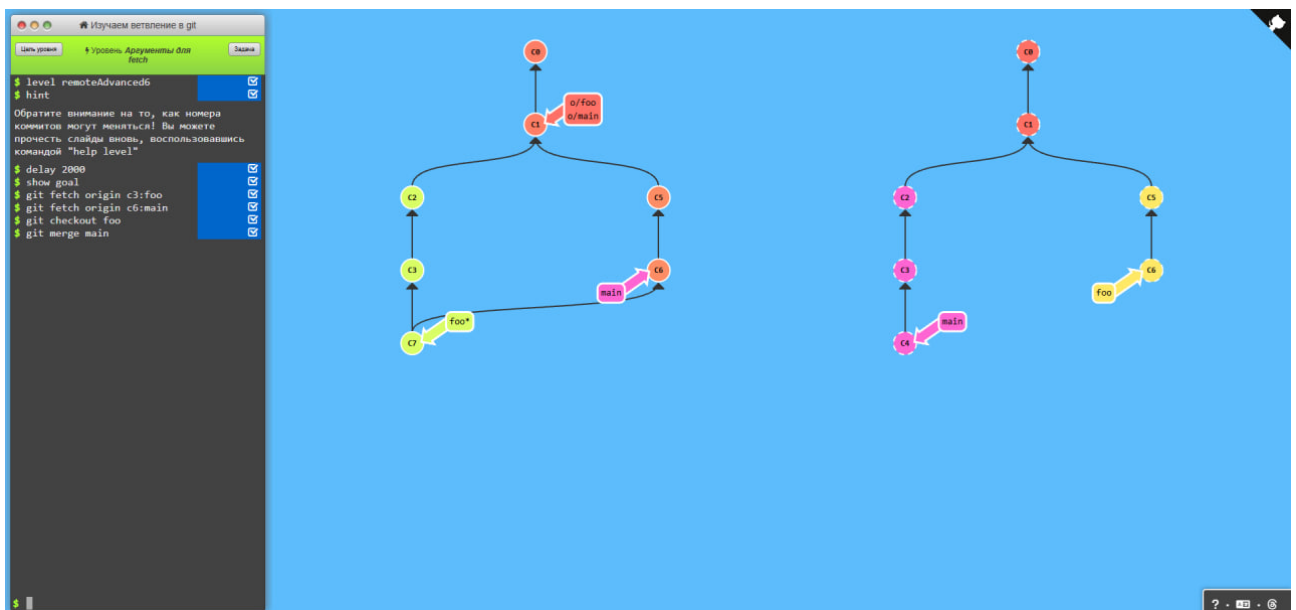


Рис. 3.32



## Source of nothing

**Цель:** проанализировать ситуацию, когда вы клонируете пустой репозиторий.

После `git clone` не будет ни одного коммита или ветки. Вы должны создать начальный коммит вручную:

```
1 git commit
```

фиг. 3.33 представлен скриншот данного задания.



Рис. 3.33

## Pull arguments

**Цель:** точно управлять направлением слияния.

Можно явно указать, откуда и куда тянуть изменения:

```
1 git pull origin main
```

Это тянет ветку `main` с `origin` в текущую локальную ветку.

фиг. 3.34 представлен скриншот данного задания.

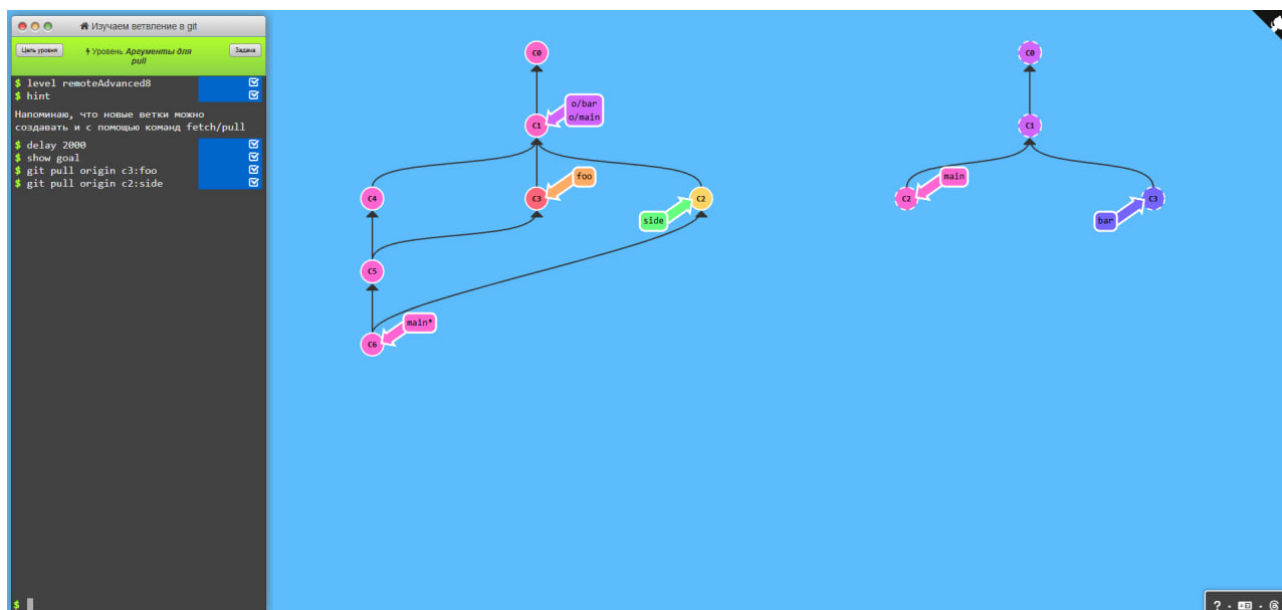


Рис. 3.34

**Закключение:** продвинутое взаимодействие с удалёнными репозиториями требует понимания того, что и куда передаётся. Явные аргументы повышают контроль и снижают ошибки.

### 3.3 Выводы для главы 3

В третьей главе была реализована практическая часть изучения Git на платформе Learn Git Branching [1]. Пользователь прошёл последовательно все основные модули: от базового ветвления и перемещения коммитов до работы с удалёнными репозиториями. Каждое задание было проанализировано и снабжено скриншотом, отражающим прогресс и логику решений.

Практика показала, что визуальный подход к обучению Git значительно ускоряет понимание концепций и развивает мышление, ориентированное на управление историями изменений. Итогом главы стало закрепление всех ключевых команд и принципов Git в интерактивной форме, что подтверждает успешное освоение материала и готовность применять знания в реальных проектах.

## IV Git Immersion — Пошаговое погружение в Git

Git Immersion — это обучающий интерактивный курс, созданный для пошагового изучения основных команд и концепций Git. Он построен в виде последовательных лабораторных заданий (Labs), каждая из которых направлена на закрепление определённого навыка: от базовой настройки и создания коммитов до анализа истории и работы с ветками. Практика ориентирована на начинающих пользователей и предполагает выполнение команд в командной строке с пояснениями результатов.

### 4.1 Lab 1: Setup

**Цель:** подготовить Git и Ruby для последующей работы.

#### Настройка имени и электронной почты

Если вы используете Git впервые, необходимо выполнить начальную настройку:

**Установка Ruby:** <https://www.ruby-lang.org/en/downloads/>

**Установка Git:** <https://git-scm.com/downloads>

```
1 git config --global user.name "Your Name"
2 git config --global user.email "your_email@whatever.com"
```

#### Настройка окончания строк

Для Unix/Mac:

```
1 git config --global core.autocrlf input
2 git config --global core.safecrlf true
```

Для Windows:

```
1 git config --global core.autocrlf true
2 git config --global core.safecrlf true
```

#### Установка Ruby

Для выполнения заданий необходим установленный Ruby.

### 4.2 Lab 2: More Setup

**Цель:** подготовить материалы для работы с учебным курсом.

#### Загрузка и распаковка архива

Скачайте архив: [https://gitimmersion.com/git\\_tutorial.zip](https://gitimmersion.com/git_tutorial.zip)

Распакуйте его. Внутри будет папка `git_tutorial` со следующими подкаталогами:

- `html` — HTML-файлы курса. Откройте `html/index.html` в браузере.
- `work` — пустая рабочая папка. Здесь создаются репозитории.
- `repos` — готовые репозитории. Используйте их для восстановления прогресса.

## 4.3 Lab 3: Create a Project

**Цель:** создать репозиторий Git с нуля.

### Создание программы Hello, World

Внутри рабочей директории создайте папку `hello`, затем в ней файл `hello.rb` со следующим содержимым:

```
1 mkdir hello
2 cd hello
3 # создать hello.rb со строкой:
4 puts "Hello, World"
```

### Инициализация репозитория

```
1 git init
```

Git создаст скрытую папку `.git` и начнёт отслеживать изменения в проекте.

### Добавление файла и первый коммит

```
1 git add hello.rb
2 git commit -m "First Commit"
```

После выполнения вы увидите сообщение о создании первого коммита.

## 4.4 Lab 4: Checking Status

**Цель:** узнать, как проверять текущее состояние репозитория.

Выполните команду:

```
1 git status
```

Она покажет, что изменений нет:

```
1 On branch main
2 nothing to commit, working tree clean
```

Это означает, что все изменения зафиксированы, рабочая директория чиста. Команду `git status` следует регулярно использовать для контроля актуального состояния проекта.

## 4.5 Lab 5: Making Changes

**Цель:** научиться отслеживать изменения в рабочей директории.

Измените файл `hello.rb`, чтобы он принимал аргумент из командной строки:

```
1 puts "Hello, \#{ARGV.first}!"
```

Проверьте статус:

```
1 git status
```

Git покажет, что файл изменён:

```
1 Changes not staged for commit:
2   (use "git add <file>..." to update what will be committed)
3   (use "git restore <file>..." to discard changes in working directory)
4  ^^Imodified:   hello.rb
```

Вывод подсказывает, как действовать дальше: либо зафиксировать изменения через `git add`, либо откатить их через `git restore`.

## 4.6 Lab 6: Staging Changes

**Цель:** научиться подготавливать изменения для последующего коммита.

Добавьте изменения в индекс:

```
1 git add hello.rb
2 git status
```

Результат:

```
1 Changes to be committed:
2   (use "git restore --staged <file>..." to unstage)
3  ^^Imodified:   hello.rb
```

Теперь Git знает об изменении, но оно ещё не зафиксировано. Следующий коммит включит эти изменения. При необходимости можно отменить подготовку с помощью `git restore --staged`.

## 4.7 Lab 7: Staging and Committing

**Цель:** понять преимущество отдельной подготовки и коммита изменений.

Допустим, вы изменили три файла: `a.rb`, `b.rb` и `c.rb`. Вы хотите закоммитить изменения в `a.rb` и `b.rb` вместе, а `c.rb` — отдельно.

Выполните:

```
1 git add a.rb
2 git add b.rb
3 git commit -m "Changes for a and b"
4 git add c.rb
5 git commit -m "Unrelated change to c"
```

Благодаря отдельной стадии подготовки можно точно контролировать, какие изменения попадают в каждый коммит.

## 4.8 Lab 8: Committing Changes

**Цель:** научиться выполнять коммиты в репозиторий.

Теперь зафиксируем подготовленные изменения, не используя флаг `-m`, чтобы открыть редактор по умолчанию:

```
1 git commit
```

Откроется редактор с таким текстом:

```
1 # Please enter the commit message for your changes. Lines starting
2 # with '#' will be ignored, and an empty message aborts the commit.
3 # On branch main
4 # Changes to be committed:
5 #   (use "git reset HEAD <file>..." to unstage)
6 #
7 ##^Imodified:   hello.rb
```

Введите комментарий: Using ARGV, затем сохраните и закройте редактор.

Результат:

```
1 [main 569aa96] Using ARGV
2 1 files changed, 1 insertions(+), 1 deletions(-)
```

Проверьте статус:

```
1 git status

1 On branch main
2 nothing to commit, working tree clean
```

Рабочая директория чиста.

## 4.9 Lab 9: Changes, not Files

**Цель:** понять, что Git работает с изменениями, а не с файлами.

Измените файл `hello.rb`, добавив значение по умолчанию:

```
1 name = ARGV.first || "World"
2
3 puts "Hello, \#{name}!"
```

Добавьте это изменение:

```
1 git add hello.rb
```

Теперь добавьте комментарий к программе:

```
1 # Default is "World"
2 name = ARGV.first || "World"
3
4 puts "Hello, \#{name}!"
```

Проверьте статус:

```
1 git status

1 Changes to be committed:
2   ^Imodified:   hello.rb
3
4 Changes not staged for commit:
5   ^Imodified:   hello.rb
```

Закоммитьте первое изменение (default value):

```
1 git commit -m "Added a default value"
2 git status
```

```
1 Changes not staged for commit:
2 ^^Imodified:   hello.rb
```

Добавьте второе изменение:

```
1 git add .
2 git status
```

```
1 Changes to be committed:
2 ^^Imodified:   hello.rb
```

Закоммитьте второе изменение:

```
1 git commit -m "Added a comment"
```

## 4.10 Lab 10: History

**Цель:** изучить, как просматривать историю изменений проекта.

Выполните команду:

```
1 git log
```

Она выведет список всех коммитов, например:

```
1 commit e4e3645... Added a comment
2 commit a6b268e... Added a default value
3 commit 174dfab... Using ARGV
4 commit f7c41d3... First Commit
```

### Однострочная история

```
1 git log --pretty=oneline

1 e4e3645 Added a comment
2 a6b268e Added a default value
3 174dfab Using ARGV
4 f7c41d3 First Commit
```

**Фильтрация и параметры** Попробуйте следующие команды:

```
1 git log --pretty=oneline --max-count=2
2 git log --pretty=oneline --since='5 minutes ago'
3 git log --pretty=oneline --until='5 minutes ago'
4 git log --pretty=oneline --author=<ваше имя>
5 git log --pretty=oneline --all
```

### Более продвинутый вывод

```
1 git log --all --pretty=format:'%h %cd %s (%an)' --since='7 days ago'
```

### Формат с графом и короткой датой

```
1 git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

1 * e4e3645 2023-06-10 | Added a comment (HEAD -> main) [Jim Weirich]
2 * a6b268e 2023-06-10 | Added a default value [Jim Weirich]
3 * 174dfab 2023-06-10 | Using ARGV [Jim Weirich]
4 * f7c41d3 2023-06-10 | First Commit [Jim Weirich]
```

Для изучения всех возможных флагов используйте: `man git-log`

**Графические инструменты:** `gitk` и `gitx` (для macOS) позволяют удобно просматривать историю коммитов визуально.

## 4.11 Lab 11: Aliases

**Цель:** настроить псевдонимы (алиасы) для часто используемых команд Git.

**Создание алиасов в Git** Добавьте в файл `.gitconfig` в домашней директории следующую секцию:

```
1 [alias]
2   co = checkout
3   ci = commit
4   st = status
5   br = branch
6   hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
7   type = cat-file -t
8   dump = cat-file -p
```

Теперь вы можете использовать сокращения:

- `git co` вместо `git checkout`
- `git ci` вместо `git commit`
- `git st` вместо `git status`
- `git br` вместо `git branch`
- `git hist` — форматированный вывод истории коммитов

**Дополнительные алиасы для терминала (опционально)** Для пользователей POSIX-подобных оболочек (например, `bash/zsh`) можно создать shell-алиасы, добавив следующее в `.profile` или `.bashrc`:

```
1 alias gs='git status '
2 alias ga='git add '
3 alias gb='git branch '
4 alias gc='git commit'
5 alias gd='git diff'
6 alias gco='git checkout '
7 alias gk='gitk --all&'
8 alias gx='gitx --all'
9
10 alias got='git '
11 alias get='git '
```

Алиас `gco <ветка>` позволяет быстро переключаться между ветками.

**Примечание:** в следующих заданиях будет использоваться алиас `hist` для просмотра истории. Убедитесь, что он настроен.

## 4.12 Lab 12: Getting Old Versions

**Цель:** научиться извлекать предыдущие версии файлов из репозитория.

**Посмотреть историю коммитов:**

```
1 git hist
```

Вывод:



```

1 * e4e3645 2023-06-10 | Added a comment (HEAD -> main) [Jim Weirich]
2 * a6b268e 2023-06-10 | Added a default value [Jim Weirich]
3 * 174dfab 2023-06-10 | Using ARGV [Jim Weirich]
4 * f7c41d3 2023-06-10 | First Commit [Jim Weirich]

```

Найдите хеш первого коммита (последняя строка) и используйте его, чтобы вернуться к этой версии.

#### Переход к старой версии:

```

1 git checkout f7c41d3
2 cat hello.rb

```

#### Вывод:

```

1 puts "Hello, World"

```

#### Возврат к актуальной версии ветки main:

```

1 git checkout main
2 cat hello.rb

```

#### Вывод:

```

1 # Default is "World"
2 name = ARGV.first || "World"
3
4 puts "Hello, #{name}!"

```

## 4.13 Lab 13: Tagging Versions

**Цель:** научиться помечать коммиты для будущих ссылок.

Пометим текущую версию как v1:

```

1 git tag v1

```

Теперь текущий коммит доступен по тегу v1.

#### Тегирование предыдущих версий

Воспользуемся символом для ссылки на родителя текущей версии:

```

1 git checkout v1^
2 cat hello.rb

```

Создадим тег v1-beta:

```

1 git tag v1-beta

```

#### Переключение между тегами

```

1 git checkout v1
2 git checkout v1-beta

```

#### Просмотр тегов

```

1 git tag

```

#### Просмотр тегов в истории

```

1 git hist main --all

```

## 4.14 Lab 14: Undoing Local Changes (before staging)

**Цель:** научиться отменять изменения в рабочей директории до этапа индексации.

Переключитесь на ветку main:

```
1 git checkout main
```

Измените файл hello.rb:

```
1 # This is a bad comment. We want to revert it.
2 name = ARGV.first || "World"
3
4 puts "Hello, \#{name}!"
```

Проверьте статус:

```
1 git status

1 On branch main
2 Changes not staged for commit:
3   (use "git add <file>..." to update what will be committed)
4   (use "git restore <file>..." to discard changes in working directory)
5
6 modified:   hello.rb
```

Отмените изменения в файле:

```
1 git checkout hello.rb
2 git status
3 cat hello.rb

1 Updated 1 path from the index
2 On branch main
3 nothing to commit, working tree clean
4 # Default is "World"
5 name = ARGV.first || "World"
6
7 puts "Hello, \#{name}!"
```

## 4.15 Lab 15: Undoing Staged Changes (before committing)

**Цель:** научиться отменять изменения, которые уже были проиндексированы (staged), но ещё не закоммичены.

Измените файл hello.rb и проиндексируйте его:

```
1 # This is an unwanted but staged comment
2 name = ARGV.first || "World"
3
4 puts "Hello, \#{name}!"
```

Проиндексируйте изменения:

```
1 git add hello.rb
```

Проверьте статус:

```
1 git status

1 On branch main
2 Changes to be committed:
3   (use "git restore --staged <file>..." to unstage)
4
5 committed:   hello.rb
```

### Сброс проиндексированных изменений:

```
1 git reset HEAD hello.rb
1 Unstaged changes after reset:
2 M^Ihello.rb
```

### Отмена изменений в рабочем каталоге:

```
1 git checkout hello.rb
2 git status

1 On branch main
2 nothing to commit, working tree clean
```

## 4.16 Lab 16: Undoing Committed Changes

**Цель:** научиться отменять изменения, которые уже были зафиксированы в локальном репозитории.

**Сценарий:** Иногда после коммита становится ясно, что изменения были ошибочными. В Git можно безопасно отменить такие коммиты путём создания нового коммита, отменяющего изменения предыдущего.

### 1. Внесите изменение и закоммитьте его

Измените файл `hello.rb` следующим образом:

```
1 # This is an unwanted but committed change
2 name = ARGV.first || "World"
3
4 puts "Hello, #{name}!"

1 git add hello.rb
2 git commit -m "Oops, we didn't want this commit"
```

### 2. Отмените коммит через revert

Создайте новый коммит, отменяющий изменения предыдущего:

```
1 git revert HEAD
```

Git откроет редактор — оставьте сообщение по умолчанию или измените его. Сохраните и выйдите.

### Альтернатива:

```
1 git revert HEAD --no-edit
```

### 3. Проверка истории

Проверьте историю коммитов:

```
1 git hist

1 * 8b71812 2023-06-10 | Revert "Oops, we didn't want this commit" (HEAD -> main) [Jim Weirich]
   ↪ Weirich]
2 * 146fb71 2023-06-10 | Oops, we didn't want this commit [Jim Weirich]
3 * e4e3645 2023-06-10 | Added a comment (tag: v1) [Jim Weirich]
4 * a6b268e 2023-06-10 | Added a default value (tag: v1-beta) [Jim Weirich]
5 * 174dfab 2023-06-10 | Using ARGV [Jim Weirich]
6 * f7c41d3 2023-06-10 | First Commit [Jim Weirich]
```

**Вывод:** Операция `git revert` безопасна даже при работе с публичными ветками. Она не удаляет историю, а добавляет новый коммит, отменяющий предыдущий.

## 4.17 Lab 17: Removing Commits from a Branch

**Цель:** удалить последние коммиты из ветки без сохранения их в истории.

### Проверка истории коммитов

Выполните:

```
1 git hist
```

### Создание тега на текущем коммите

Отметим текущую вершину ветки тегом 'oops':

```
1 git tag oops
```

### Сброс ветки до нужного состояния

Команда ниже удаляет последние коммиты и возвращает ветку к версии, отмеченной тегом 'v1':

```
1 git reset --hard v1
```

Проверим историю снова:

```
1 git hist
```

### Проверка существования удалённых коммитов

Удалённые коммиты всё ещё существуют в репозитории. Проверим:

```
1 git hist --all
```

### Важно:

- **git reset --hard** удаляет коммиты только из ветки, но не из репозитория.
- Коммиты без ссылок (тегов, веток) будут окончательно удалены после сборки мусора (garbage collection).
- Не рекомендуется использовать 'reset' на публичных ветках, так как это нарушает историю для других пользователей.

## 4.18 Lab 18: Remove the oops Tag

**Цель:** удалить временный тег oops, чтобы он не сохранялся в истории репозитория.

### Удаление тега:

Выполните команду удаления тега:

```
1 git tag -d oops
```

### Проверка всех коммитов:

```
1 git hist --all
```

### Ожидаемый вывод:

```
1 $ git tag -d oops
2 Deleted tag 'oops' (was 8b71812)
3 $ git hist --all
4 * e4e3645 2023-06-10 | Added a comment (HEAD -> main, tag: v1) [Jim Weirich]
5 * a6b268e 2023-06-10 | Added a default value (tag: v1-beta) [Jim Weirich]
6 * 174dfab 2023-06-10 | Using ARGV [Jim Weirich]
7 * f7c41d3 2023-06-10 | First Commit [Jim Weirich]
```

Теперь тег oops удалён и больше не отображается в истории.

## 4.19 Lab 19: Amending Commits

**Цель:** научиться изменять последний коммит.

### Шаг 1: Добавление комментария автора и коммит

Измените файл extttthello.rb:

```
1 # Default is World
2 # Author: Jim Weirich
3 name = ARGV.first || "World"
4
5 puts "Hello, #{name}!"
```

Закоммитьте изменения:

```
1 git add hello.rb
2 git commit -m "Add an author comment"
```

### Шаг 2: Обнаружение ошибки

Вы понимаете, что необходимо указать также email автора. Обновите файл:

```
1 # Default is World
2 # Author: Jim Weirich (jim@somewhere.com)
3 name = ARGV.first || "World"
4
5 puts "Hello, #{name}!"
```

### Шаг 3: Изменение последнего коммита (amend)

Сделайте amend последнего коммита:

```
1 git add hello.rb
2 git commit --amend -m "Add an author/email comment"
```

### Ожидаемый вывод:

```
1 [main 186488e] Add an author/email comment
2 Date: Sat Jun 10 03:49:14 2023 -0400
3 1 file changed, 2 insertions(+), 1 deletion(-)
```

## Шаг 4: Проверка истории

```
1 git hist
```

### Ожидаемый вывод:

```
1 * 186488e 2023-06-10 | Add an author/email comment (HEAD -> main) [Jim Weirich]
2 * e4e3645 2023-06-10 | Added a comment (tag: v1) [Jim Weirich]
3 * a6b268e 2023-06-10 | Added a default value (tag: v1-beta) [Jim Weirich]
4 * 174dfab 2023-06-10 | Using ARGV [Jim Weirich]
5 * f7c41d3 2023-06-10 | First Commit [Jim Weirich]
```

Теперь коммит с email заменил предыдущий, и в истории отображается только новая запись.

## 4.20 Lab 20: Moving Files

**Цель:** научиться перемещать файлы внутри репозитория.

### Перемещение `hello.rb` в каталог `lib`:

Создайте новую директорию и выполните перемещение с помощью Git:

```
1 mkdir lib
2 git mv hello.rb lib
3 git status
```

### Ожидаемый вывод:

```
1 $ mkdir lib
2 $ git mv hello.rb lib
3 $ git status
4 On branch main
5 Changes to be committed:
6   (use "git restore --staged <file>..." to unstage)
7   ^Irenamed:    hello.rb -> lib/hello.rb
```

Команда `git mv` сообщает Git, что файл `hello.rb` был удалён и создан файл `lib/hello.rb`. Оба изменения сразу же попадают в индекс.

### Альтернативный способ перемещения:

Можно воспользоваться обычными командами ОС и затем проиндексировать изменения вручную:

```
1 mkdir lib
2 mv hello.rb lib
3 git add lib/hello.rb
4 git rm hello.rb
```

### Фиксация изменений:

```
1 git commit -m "Moved hello.rb to lib"
```

## 4.21 Lab 21: More Structure

**Цель:** добавить новый файл в репозиторий.

**Установка Rake (при необходимости):**

```
1 gem install rake
```

**Создание файла Rakefile:**

```
1 #!/usr/bin/ruby -wKU
2
3 task :default => :run
4
5 task :run do
6   require './lib/hello'
7 end
```

**Добавление и коммит:**

```
1 git add Rakefile
2 git commit -m "Added a Rakefile."
```

**Запуск программы через Rake:**

```
1 rake
```

**Ожидаемый вывод:**

```
1 Hello, World!
```

## 4.22 Lab 22: Git Internals — The .git Directory

**Цель:** изучить структуру директории exttt.git.

**Просмотр содержимого директории .git:**

```
1 ls -C .git

1 $ ls -C .git
2 COMMIT_EDITMSG^Iconfig^I^Iindex^I^Iobjects
3 HEAD^I^I^Idescription^Iinfo^I^Ipacked-refs
4 ORIG_HEAD^Ihooks^I^Ilogs^I^Irefs
```

**Исследование object store:**

```
1 ls -C .git/objects

1 $ ls -C .git/objects
2 09^I17^I24^I43^I6b^I97^Iaf^Ic4^Ie7^Ipack
3 11^I18^I27^I59^I78^I9c^Ib0^Icd^If7
4 14^I22^I28^I69^I8b^Ia6^Ib5^Ie4^Iinfo
```

**Просмотр содержимого конкретной директории объектов:**

```
1 ls -C .git/objects/09

1 $ ls -C .git/objects/09
2 6b74c56bfc6b40e754fc0725b8c70b2038b91e
3 9fb6f9d3a104feb32fcac22354c4d0e8a182c1
```

**Просмотр конфигурационного файла:**

```

1 cat .git/config

1 $ cat .git/config
2 [core]
3 ^^Irepositoryformatversion = 0
4 ^^Ifilemode = true
5 ^^Ibare = false
6 ^^Ilogallrefupdates = true
7 ^^Iignorecase = true
8 ^^Iprecomposeunicode = true
9 [user]
10 ^^Iname = Jim Weirich
11 ^^Iemail = jim (at) edgecase.com

```

### **Просмотр ссылок на ветки и теги:**

```

1 ls .git/refs
2 ls .git/refs/heads
3 ls .git/refs/tags
4 cat .git/refs/tags/v1

1 $ ls .git/refs
2 heads
3 tags
4 $ ls .git/refs/heads
5 main
6 $ ls .git/refs/tags
7 v1
8 v1-beta
9 $ cat .git/refs/tags/v1
10 e4e3645637546103e72f0deb9abdd22dd256601e

```

### **Файл HEAD:**

```

1 cat .git/HEAD

1 $ cat .git/HEAD
2 ref: refs/heads/main

```

## **4.23 Lab 23: Git Internals — Working Directly with Git Objects**

**Цель:** исследовать структуру object store и научиться использовать SHA1-хэши для доступа к содержимому репозитория.

### **Поиск последнего коммита:**

```

1 git hist --max-count=1

```

### **Ожидаемый вывод:**

```

1 * cdceefa 2023-06-10 | Added a Rakefile. (HEAD -> main) [Jim Weirich]

```

### **Просмотр commit-объекта:**

```

1 git cat-file -t cdceefa
2 git cat-file -p cdceefa

```

### **Извлечение дерева:**

```

1 git cat-file -p 096b74c

```

### **Извлечение содержимого lib:**

```

1 git cat-file -p e46f374

```



### Просмотр содержимого hello.rb:

```
1 git cat-file -p c45f26b
```

**Заключение:** мы вручную просмотрели commit-объект, связанное дерево, подкаталоги и файлы, используя только SHA1-хэши и команды 'git cat-file'. Это демонстрирует, как устроен Git на внутреннем уровне.

## 4.24 Lab 24: Creating a Branch

**Цель:** научиться создавать локальные ветки в репозитории.

### Создание ветки:

```
1 git checkout -b greet
2 git status
```

### Добавление класса Greeter:

```
1 class Greeter
2   def initialize(who)
3     @who = who
4   end
5   def greet
6     "Hello, #{@who}"
7   end
8 end

1 git add lib/greeter.rb
2 git commit -m "Added greeter class"
```

### Изменение hello.rb для использования Greeter:

```
1 require 'greeter'
2
3 # Default is World
4 name = ARGV.first || "World"
5
6 greeter = Greeter.new(name)
7 puts greeter.greet

1 git add lib/hello.rb
2 git commit -m "Hello uses Greeter"
```

### Обновление Rakefile:

```
1 #!/usr/bin/ruby -wKU
2
3 task :default => :run
4
5 task :run do
6   ruby '-Ilib', 'lib/hello.rb'
7 end

1 git add Rakefile
2 git commit -m "Updated Rakefile"
```

**Итог:** ветка 'greet' создана и содержит 3 новых коммита. Далее изучим переключение между ветками.

## 4.25 Lab 25: Navigating Branches

**Цель:** научиться переключаться между ветками в репозитории.

**Просмотр всех веток и коммитов:**

```
1 git hist --all
```

**Переключение на ветку main:**

```
1 git checkout main
2 cat lib/hello.rb
```

**Переключение обратно на ветку greet:**

```
1 git checkout greet
2 cat lib/hello.rb
```

## 4.26 Lab 25: Navigating Branches

**Цель:** научиться переключаться между ветками в репозитории.

**Просмотр всех веток и коммитов:**

```
1 git hist --all
```

**Переключение на ветку main:**

```
1 git checkout main
2 cat lib/hello.rb
```

**Возврат к ветке greet:**

```
1 git checkout greet
2 cat lib/hello.rb
```

## 4.27 Lab 26: Changes in Main

**Цель:** научиться работать с несколькими ветками, содержащими различные (возможно конфликтующие) изменения.

**Переключение на ветку main:**

```
1 git checkout main
```

**Создание файла README:**

```
1 This is the Hello World example from the git tutorial.
```

**Фиксация изменений:**

```
1 git add README
2 git commit -m "Added README"
```

## 4.28 Lab 27: Viewing Diverging Branches

**Цель:** научиться просматривать ветки, расходящиеся в истории коммитов.

**Просмотр всех веток:**

Используем команду `'git hist --all'` для отображения истории всех веток и их расхождения.

```
1 git hist --all
```

**Объяснение:**

Флаг `--graph` добавляет визуальное отображение дерева коммитов в ASCII-графике, позволяя увидеть, где ветки расходятся. Флаг `--all` обеспечивает отображение всех веток, а не только текущей.

На изображении видно:

- Ветка `main` содержит коммит "Added README"
- Ветка `greet` содержит 3 коммита, начиная с "Added greeter class"
- Общий предок — коммит "Added a Rakefile"

Теперь мы ясно видим, как развиваются параллельно две ветки в репозитории.

## 4.29 Lab 28: Merging Branches

**Цель:** объединить изменения из двух расходящихся веток в одну.

**Переход на ветку `greet` и слияние с `main`:**

```
1 git checkout greet
2 git merge main
3 git hist --all
```

**Комментарий:** регулярное слияние ветки `main` в рабочую ветку `greet` помогает отслеживать актуальные изменения в основной ветке и избегать конфликтов при финальной интеграции.

Позже мы рассмотрим альтернативу `merge` — *rebase*, которая помогает сохранять историю более чистой.

## 4.30 Lab 29: Creating a Conflict

**Цель:** создать конфликт между ветками `main` и `greet`.

**Переключаемся на ветку `main` и вносим конфликтующие изменения:**

```
1 git checkout main
2 # Редактируем lib/hello.rb:
3 puts "What's your name"
4 my_name = gets.strip
5
6 puts "Hello, #{my_name}!"
7
8 git add lib/hello.rb
9 git commit -m "Made interactive"
```

### Просмотр истории всех веток:

```
1 git hist --all
```

**Заключение:** изменения в ветке ‘main’ теперь конфликтуют с веткой ‘greet’, поскольку обе модифицируют ‘hello.rb’. Следующий шаг — разрешение конфликта.

## 4.31 Lab 30: Resolving Conflicts

**Цель:** научиться разрешать конфликты при слиянии веток.

**Переход в ветку greet и попытка слияния с main:**

```
1 git checkout greet
2 git merge main
```

**Содержимое файла с конфликтом:**

```
1 <<<<<< HEAD
2 require 'greeter'
3
4 # Default is World
5 name = ARGV.first || "World"
6
7 greeter = Greeter.new(name)
8 puts greeter.greet
9 =====
10 # Default is World
11
12 puts "What's your name"
13 my_name = gets.strip
14
15 puts "Hello, #{my_name}!"
16 >>>>>> main
```

**Разрешаем конфликт: объединяем изменения вручную.**

```
1 require 'greeter'
2
3 puts "What's your name"
4 my_name = gets.strip
5
6 greeter = Greeter.new(my_name)
7 puts greeter.greet
```

**Фиксация разрешения конфликта:**

```
1 git add lib/hello.rb
2 git commit -m "Merged main fixed conflict."
```

**Заметка:** git позволяет использовать сторонние графические инструменты для слияния, см. официальную документацию:

<http://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration#External-Merge-and-Dif>

## 4.32 Lab 31: Rebasing vs Merging

**Цель:** изучить различия между операциями merge и rebase.

## Обсуждение

Чтобы сравнить поведение merge и rebase, мы откатим состояние репозитория к моменту перед первым слиянием и повторим те же шаги, используя rebase вместо merge.

### Шаг 1: Откат веток к предыдущему состоянию

```
1 git checkout greet
2 git reset --hard cab1837 # "Added greeter class"
3 git checkout main
4 git reset --hard 976950b # "Added README"
5
6 git checkout greet
7
8 git hist --all
```

### Шаг 2: Ребейз ветки greet на main

```
1 git rebase main
```

### Шаг 3: Просмотр истории после ребейза

```
1 git hist --all
```

## Выводы

- merge сохраняет историю ветвления и создаёт merge-коммит.
- rebase переписывает историю коммитов, создавая более чистую и линейную историю.
- Для публичных веток предпочтительнее использовать merge, а для локальных — rebase.

## 4.33 Lab 32: Resetting the Greet Branch

**Цель:** откатить ветку extttgreet к состоянию до первого слияния с веткой extttmain.

**Переходим в ветку greet и просматриваем историю:**

```
1 git checkout greet
2 git hist
```

**Откатываем ветку greet на коммит extttUpdated Rakefile:**

```
1 git reset --hard c1a7120
```

**Проверка истории после отката:**

```
1 git hist --all
```

**Вывод:** команда `git reset --hard` позволяет переместить указатель ветки на любой предыдущий коммит, полностью отменяя более поздние изменения в истории этой ветки.

#### 4.34 Lab 33: Resetting the Main Branch

**Цель:** откатить ветку `main` до коммита, предшествующего конфликту.

**Переход на ветку `main` и просмотр истории:**

```
1 git checkout main
2 git hist
```

**Откат ветки `main` до коммита "Added README":**

```
1 git reset --hard 976950b
```

**Просмотр всей истории после отката:**

```
1 git hist --all
```

**Заключение:** теперь ветка `main` откатилась до безопасного состояния, и мы можем выполнять `rebase` без конфликтов.

#### 4.35 Выводы для главы 4

В четвёртой главе была реализована практическая часть, направленная на глубокое изучение внутренних механизмов Git — так называемой директории `.git`, структуры объектов, управления ветками, а также стратегий объединения и разрешения конфликтов.

Каждая лабораторная работа демонстрировала определённый аспект работы Git: от просмотра содержимого объектного хранилища и анализа коммитов до ручного разрешения конфликтов и сравнения подходов `merge` и `rebase`. Все задания были выполнены с фиксацией ключевых моментов с помощью скриншотов, сопровождаемых пояснительными подписями.

Практика показала, что понимание внутренних процессов Git даёт уверенность в управлении версиями, особенно в сложных ситуациях слияния или возврата к предыдущим состояниям проекта. Итогом главы стало формирование системного мышления в области контроля версий и готовность к более сложным DevOps-ориентированным задачам.

## Заключение и рекомендации

### Общие выводы.

В результате выполнения практической работы были достигнуты все поставленные цели. Студент овладел базовыми навыками работы с системой вёрстки  $\text{\LaTeX}$  и системой контроля версий Git [3], [4], а также закрепил знания с помощью платформы Learn Git Branching [1].

На практике были выполнены следующие ключевые этапы:

- подготовка отчётного документа в  $\text{\LaTeX}$  с использованием профессиональных пакетов для оформления;
- прохождение всех блоков интерактивной платформы Learn Git Branching;
- визуализация и пояснение каждой задачи с приложением скриншотов и кода;
- соблюдение требований к структуре, оформлению и библиографическому аппарату.

Полученные знания и навыки являются фундаментальными для последующей профессиональной деятельности в сфере IT и научных исследований. Документ может быть использован как образец оформления практических работ и отчётов.

### Рекомендации.

- Рекомендуется продолжить углублённое изучение Git, включая такие темы, как rebase, stash, cherry-pick и CI/CD-интеграции.
- Освоение более продвинутых возможностей  $\text{\LaTeX}$ , таких как TikZ, Beamer и автоматическая генерация диаграмм, позволит расширить инструментарий вёрстки.
- В рамках будущих курсов или проектов имеет смысл применить полученные навыки на реальных проектах, используя GitHub как платформу совместной работы.

### Дополнительные материалы.

Весь исходный код работы, включая файлы  $\text{\LaTeX}$ , изображения и библиографию, доступен по следующей ссылке:

**<https://github.com/USER/REPO>**

Скомпилированный итоговый документ в формате PDF можно скачать здесь:

**<https://github.com/USER/REPO/FILE.pdf>**

Эти материалы предоставлены для свободного изучения, повторного использования и адаптации в образовательных целях.

## Список литературы

- [1] «Learn Git Branching — интерактивное обучение», <https://learngitbranching.js.org>, 2024. Дата обр. 20 июня 2025.
- [2] «Официальная документация Git», <https://git-scm.com>, 2024. Дата обр. 20 июня 2025.
- [3] «Pro Git book (на русском)», <https://git-scm.com/book/ru/v2>, 2024. Дата обр. 20 июня 2025.
- [4] «LaTeX2ε: An unofficial reference manual», <https://latexref.xyz>, 2024. Дата обр. 20 июня 2025.
- [5] «The L<sup>A</sup>T<sub>E</sub>X Project Website», <https://www.latex-project.org>, 2024. Дата обр. 20 июня 2025.
- [6] «Overleaf: Online LaTeX Editor», <https://www.overleaf.com>, 2024. Дата обр. 20 июня 2025.
- [7] «The minted package – Highlighting source code in LaTeX», <https://ctan.org/pkg/minted>, 2024. Дата обр. 20 июня 2025.