

МОЛДАВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ МАТЕМАТИКИ И ИНФОРМАТИКИ
ДЕПАРТАМЕНТ ИНФОРМАТИКИ

МАЕВ Сергей

ИЗУЧЕНИЕ LATEX И GIT

ИНФОРМАТИКА

Практическая работа

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

КИШИНЕВ – 2025 Г.

Оглавление

ВВЕДЕНИЕ	3
I. Система вёрстки \LaTeX и её применение	6
1.1. Введение в \LaTeX	6
1.2. Установка WSL и \LaTeX на Windows	6
1.3. Базовый синтаксис \LaTeX	7
1.4. Выводы и актуальность использования \LaTeX	8
II. Изучение Git и работы с ветками на платформе LearnGitBranching	9
2.1. Введение	9
2.2. Урок 1. Коммиты в Git	9
2.3. Урок 2. Ветвление в Git	10
2.4. Урок 3. Ветки и слияния	11
2.5. Урок 4. Git Rebase	12
2.6. Урок 5. Прогулка по Git и указатель HEAD	14
2.7. Урок 6. Относительные ссылки в Git	15
2.8. Урок 7. Оператор ~ и перемещение ветки (branch forcing)	16
2.9. Урок 8. Отмена изменений в Git	17
2.10. Урок 9. Git Cherry-pick	19
2.11. Урок 10. Git Interactive Rebase	20
2.12. Урок 11. Выборочное извлечение нужного коммита	22
2.13. Урок 12. Жонглируем коммитами	22
2.14. Урок 13. Жонглируем коммитами №2 (через cherry-pick)	24
2.15. Урок 14. Теги в Git	25
2.16. Урок 15. Команда git describe	26
2.17. Урок 16. Rebase на нескольких ветках	27
2.18. Урок 17. Определение родителей коммита	29
2.19. Урок 18. Спутанные ветки	30
2.20. Урок 19. Удалённые репозитории в Git	32
2.21. Урок 20. Удалённые ветки в Git	33
2.22. Урок 21. Git Fetch	34

2.23. Урок 22. Git Pull	35
2.24. Урок 23. Симуляция совместной работы	36
2.25. Урок 24. Git Push	37
2.26. Урок 25. Когда наработки расходятся	38
2.27. Урок: Remote Rejected!	39
2.28. Выводы для главы 2	39
III. Практическое погружение в Git: Git Immersion	41
3.1. Lab 1: Настройка Git и Ruby	41

ВВЕДЕНИЕ

Актуальность и важность темы.

В наших реалиях становится особенно важным использование современных инструментов для эффективной организации работы с кодом и документацией. Одними из таких незаменимых средств являются системы контроля версий, такие как Git, и профессиональные системы верстки документов, такие как LaTeX. Git позволяет отслеживать изменения в проекте, управлять различными версиями, организовывать совместную работу команды и обеспечивать надёжное хранение кода. Этот инструмент стал стандартом де-факто в индустрии программирования, благодаря своей гибкости и широкому распространению. С другой стороны, LaTeX остаётся незаменимым инструментом для создания научных отчётов, дипломных и курсовых работ, статей и технической документации. Его возможности по точному форматированию текста, работе с математическими формулами, таблицами и библиографиями делают его актуальным в научной и инженерной среде. Совместное использование Git и LaTeX позволяет эффективно организовать процесс подготовки и редактирования документов, обеспечивая контроль версий и возможность командной работы над научными и техническими текстами. Это делает изучение и применение этих инструментов особенно актуальным для студентов, исследователей и разработчиков.

Цель и задачи.

Цель практики

Целью данной практики является овладение навыками профессиональной подготовки технической документации с использованием системы верстки LaTeX, а также изучение и применение системы контроля версий Git для управления проектами и совместной разработки.

Задачи практики

1. Ознакомиться с шаблоном дипломной работы, оформленным в LaTeX (ссылка на шаблон), установить необходимое программное обеспечение и адаптировать шаблон под свои нужды.
2. Изучить основные принципы работы с системой контроля версий Git на интерактивной платформе LearnGitBranching, включая создание веток, слияние, откат изменений и разрешение конфликтов.
3. Пройти минимум 30 уроков на платформе Git Immersion, закрепить знания, полученные

ранее, и научиться использовать Git в реальных рабочих сценариях.

4. Документировать выполненные упражнения и этапы изучения в \LaTeX , применяя изученный шаблон, обеспечивая при этом структурированное, читабельное и грамотно оформленное содержание.
5. Сформировать единый отчёт по практике, который отражает как освоенные технические навыки, так и уровень владения инструментами Git и \LaTeX .

Методологическая и технологическая база.

Методологическая база

Методологическую основу практики составляет современный подход к разработке программного обеспечения и технической документации. В рамках практики применяются:

- методы модульного и итеративного обучения;
- принципы инженерной документации;
- практика контроля версий и отслеживания изменений;
- стандарты оформления научно-технических текстов с использованием системы \LaTeX .

Технологическая база

Для реализации поставленных целей и задач использовались следующие инструменты и технологии:

- **LaTeX** — система компьютерной вёрстки, используемая для создания научной и технической документации;
- **Git** — распределённая система контроля версий;
- **GitHub** — платформа для хостинга репозитория и совместной работы;
- **LearnGitBranching** — интерактивная среда для визуального изучения Git;
- **Git Immersion** — онлайн-курс для практического освоения Git;
- **Операционная система:** Windows 10/11 или Linux (WSL/Ubuntu);
- **Дополнительное ПО:** браузер Google Chrome и текстовый редактор VS Code.

Практическая ценность.

Разработанное приложение и выполненная практика обладают практической ценностью в контексте реальных рабочих и учебных процессов. Освоенные технологии могут быть непосредственно применены в следующих сферах:

- **Подготовка технической и научной документации.** Использование \LaTeX позволяет создавать профессионально оформленные отчёты, статьи, диссертации и технические руководства, что актуально как в академической, так и в инженерной среде.

- **Организация командной разработки.** Навыки работы с Git позволяют эффективно участвовать в совместных проектах, управлять историей изменений, разрешать конфликты и вести параллельную разработку с другими участниками команды.
- **Автоматизация документооборота.** Связка Git и L^AT_EX может использоваться для автоматизации выпуска версий документации, ведения журналов изменений и управления различными вариантами документов в рамках одного проекта.
- **Публикация и открытые проекты.** Полученные знания позволяют разрабатывать и поддерживать открытые проекты на платформах вроде GitHub, что способствует участию в сообществе разработчиков и созданию собственных портфолио.
- **Интеграция с CI/CD.** Знания по Git могут быть расширены до автоматической сборки и тестирования документации и программных продуктов, что особенно полезно в профессиональной разработке.

Таким образом, полученные в ходе практики навыки и результаты могут быть использованы не только в учебных целях, но и в реальной инженерной, научной или производственной деятельности.

Краткое содержание диссертации.

Первая глава, Система вёрстки L^AT_EX и её применение, представляет общую / теоретическую информацию о

Вторая глава, ??, описывает реализацию, ...

Третья глава, ??, следует за реализацией ...

I Система вёрстки L^AT_EX и её применение

1.1 Введение в L^AT_EX

Система вёрстки L^AT_EX является мощным инструментом для подготовки научных, технических и академических текстов. Она основана на языках разметки TeX и позволяет пользователю сосредоточиться на содержании документа, не отвлекаясь на его оформление.

L^AT_EX широко используется во многих областях:

- подготовка отчётов, диссертаций, статей и презентаций;
- публикации в научных журналах;
- создание документов с большим количеством формул и таблиц;
- оформление лабораторных и курсовых работ.

Ключевыми преимуществами L^AT_EX являются:

- чёткая структура документа и автоматическое управление содержанием;
- высокое качество типографики;
- удобная работа с таблицами, рисунками, ссылками и библиографией;
- расширяемость с помощью пакетов.

1.2 Установка WSL и L^AT_EX на Windows

Рабочая среда для подготовки документов в системе L^AT_EX была развёрнута на Windows 10 с использованием технологии WSL (Windows Subsystem for Linux). Это позволило использовать команды и пакеты Linux без виртуальных машин или двойной загрузки.

Процесс установки включал следующие шаги:

1. Установка WSL:

В командной строке PowerShell была выполнена команда:

```
wsl --install
```

После перезагрузки системы был установлен дистрибутив Ubuntu.

2. Обновление системы:

После запуска Ubuntu были выполнены стандартные команды обновления:

```
cd ~ sudo apt update
```

3. Установка пакетов для L^AT_EX:

Были установлены следующие компоненты:

- базовый дистрибутив L^AT_EX: texlive, texlive-xetex;

- поддержка кириллицы: `texlive-lang-cyrillic`, `texlive-lang-european`;
- средства библиографии: `biber`, `texlive-bibtex-extra`;
- система автоматической сборки: `latexmk`;
- утилиты шрифтов: `fonts-liberation`, `xz-utils`;
- подсветка кода: `python3-pygments`.

Все пакеты устанавливались одной командой:

```
sudo apt install fonts-liberation xz-utils texlive-bibtex-extra biber texlive
texlive-lang-cyrillic texlive-lang-european python3-pygments latexmk texlive-
xetex
```

4. Установка шрифтов Times New Roman:

Для соответствия ГОСТ-оформлению был установлен шрифт Times New Roman из открытого источника. Использовались команды:

```
curl -L -O https://notabug.org/ArtikushHG/times-new-roman/raw/master/times.tar.xz
sudo tar -xf times.tar.xz -C /usr/share/fonts/
fc-cache -f -v
```

После этого я компилировал документы с помощью команды:

```
./render.sh -f -shell-escape
```

1.3 Базовый синтаксис L^AT_EX

Синтаксис L^AT_EX основан на командах, начинающихся с символа ‘\’, и на структурных блоках (окружениях), таких как ‘`itemize`’, ‘`figure`’, ‘`table`’.

Пример структуры документа:

```
1 \documentclass[a4paper,12pt]{report}
2 \usepackage[utf8]{inputenc}
3 \usepackage[russian]{babel}
4 \begin{document}
5 \chapter{Пример главы}
6 \section{Введение}
7 Пример параграфа.
8 \end{document}
```

Типичные команды форматирования:

- `\textbf{текст}` — жирный текст;
- `\textit{текст}` — курсив;
- `\texttt{текст}` — моноширинный текст (код);

Списки:

Маркированный список:

```
1 \begin{itemize}
2   \item Пункт 1
3   \item Пункт 2
4 \end{itemize}
```


Нумерованный список:

```
1 \begin{enumerate}
2   \item Первый шаг
3   \item Второй шаг
4 \end{enumerate}
```

Изображения:

```
1 \begin{figure}[!ht]
2   \centering
3   \includegraphics[width=0.5\textwidth]{example.png}
4   \caption{Пример изображения}
5   \label{fig:example}
6 \end{figure}
```

Подсветка кода через `minted`:

```
1 def hello():
2     print("Привет, мир!")
```

Для этого необходимо компилировать с опцией ‘`-shell-escape`’.

1.4 Выводы и актуальность использования \LaTeX

\LaTeX остаётся незаменимым инструментом в научной и инженерной среде благодаря следующим качествам:

- Поддержка сложной разметки (формулы, код, диаграммы);
- Универсальность — используется в физике, математике, информатике и др.;
- Совместимость с системами контроля версий (например, Git);
- Расширяемость — можно подключать десятки дополнительных пакетов.

На практике \LaTeX используется:

- для написания статей, тезисов и диссертаций;
- при оформлении документации к программному обеспечению;
- в журналах IEEE, ACM и других международных публикациях;
- в образовательных целях — при написании отчётов, курсовых и лабораторных.

Изучение \LaTeX дало мне не только практический инструмент для написания диссертации, но и позволило освоить принципы структурной разметки, автоматизации и профессионального оформления текста.

II Изучение Git и работы с ветками на платформе LearnGitBranching

2.1 Введение

Система контроля версий Git является одним из самых популярных инструментов в современной разработке программного обеспечения. Она позволяет отслеживать изменения в проекте, работать над несколькими задачами параллельно и эффективно организовывать командную работу. Одной из ключевых возможностей Git является работа с ветками (branches), позволяющая изолировать функциональность, экспериментировать с кодом и объединять изменения в основной поток разработки.

В рамках данной главы изучение Git и концепции ветвления производилось с помощью интерактивной обучающей платформы **Learn Git Branching**. Это веб-приложение предоставляет визуальные задания, эмулятор командной строки и пошаговое обучение основным и продвинутым возможностям Git. Все действия сопровождаются визуализацией дерева коммитов, что позволяет лучше понять структуру репозитория и поведение команд Git.

Платформа доступна на русском языке и охватывает следующие темы:

- базовая навигация и история коммитов;
- создание, переключение и удаление веток;
- слияние изменений с помощью merge и rebase;
- отмена изменений через reset и revert;
- работа с удалёнными репозиториями и управление конфликтами.

Каждый урок выполняется в браузере и предоставляет последовательные задачи, где необходимо ввести правильные команды Git, чтобы достичь требуемого состояния репозитория. Все команды и концепции, изученные в процессе, фиксировались в рамках данного отчёта.

В следующих разделах представлены краткие описания пройденных уровней, объяснение использованных команд и полученные выводы.

2.2 Урок 1. Коммиты в Git

Первое упражнение на платформе Learn Git Branching было посвящено основной единице хранения в Git — коммиту. Коммит представляет собой сохранение текущего состояния всех файлов в репозитории. При этом Git не сохраняет полную копию каждого состояния, а хранит лишь изменения (дельты) между ними, что делает работу быстрой и эффективной.

Каждый коммит связан с предыдущими коммитами — предками, и образует цепочку, ви-

зуализируемую в виде направленного графа. Это позволяет отслеживать историю разработки и в любой момент времени «откатиться» к предыдущему состоянию проекта.

Задание

В рамках упражнения нужно было выполнить две команды `git commit`, чтобы зафиксировать два изменения.

Результат

Ниже представлен скриншот успешно выполненного задания:

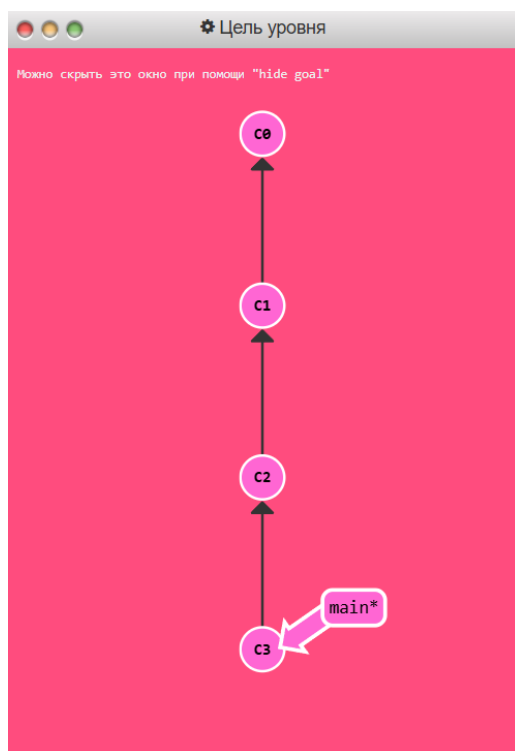


Рис. 2.1. Выполнение задания по созданию двух коммитов

2.3 Урок 2. Ветвление в Git

Вторая часть обучения на платформе Learn Git Branching была посвящена концепции ветвления.

Ветви (*branches*) в Git являются лёгкими указателями на коммиты. По сути, это просто имена, указывающие на конкретные точки в истории изменений. Благодаря этой простоте ветки создаются и переключаются практически мгновенно, не занимая дополнительных ресурсов. Такой подход помогает организовать параллельную работу над несколькими задачами без риска затронуть основную рабочую ветку.

Задание

В рамках упражнения нужно было выполнить две основные команды:

- `git branch` — создать новую ветку с именем `feature`;
- `git checkout` — переключиться на новую ветку.

После выполнения команды `checkout`, указатель `HEAD` начинает ссылаться на новую ветку, и последующие коммиты будут записываться в её историю.

Результат

Ниже представлен скриншот, подтверждающий успешное выполнение задания:

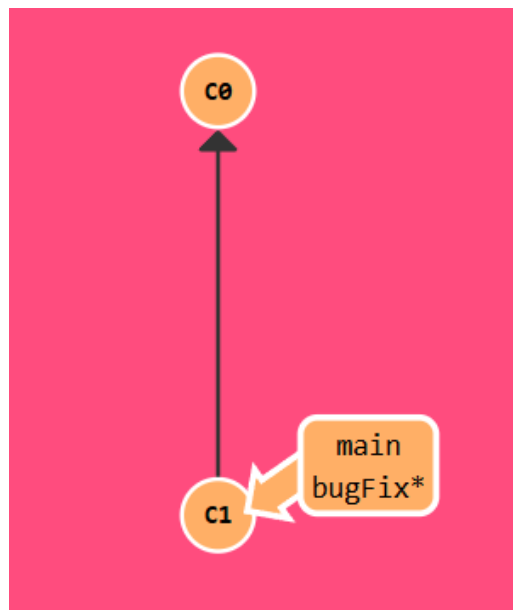


Рис. 2.2. Создание и переключение на новую ветку

2.4 Урок 3. Ветки и слияния

Третье упражнение на платформе `Learn Git Branching` посвящено механизму объединения изменений, сделанных в разных ветках. В реальных проектах часто бывает, что разработчики работают параллельно: каждый в своей ветке. После завершения работы возникает необходимость объединить эти изменения. В Git это делается с помощью команды `git merge`.

`git merge` создаёт новый *сливающий коммит* (`merge commit`), который объединяет изменения из двух веток. Такой коммит имеет двух родителей — один указывает на текущее состояние основной ветки, а другой — на ветку, с которой происходит слияние. Это позволяет сохранить обе истории изменений и продолжать разработку в объединённой ветке.

Задание

Для успешного прохождения уровня нужно было выполнить следующие шаги:

1. Создать новую ветку с именем bugFix: `git branch bugFix`
2. Переключиться на неё: `git checkout bugFix`
3. Сделать один коммит в ветке bugFix: `git commit`
4. Вернуться в основную ветку: `git checkout main`
5. Сделать ещё один коммит в main: `git commit`
6. Выполнить слияние ветки bugFix в main: `git merge bugFix`

После выполнения этих команд визуализация дерева коммитов изменилась: появился merge-коммит, у которого два родителя. Это означает, что изменения из обеих веток были успешно объединены.

Результат

На рисунке ниже показан результат успешного слияния ветки bugFix с веткой main:

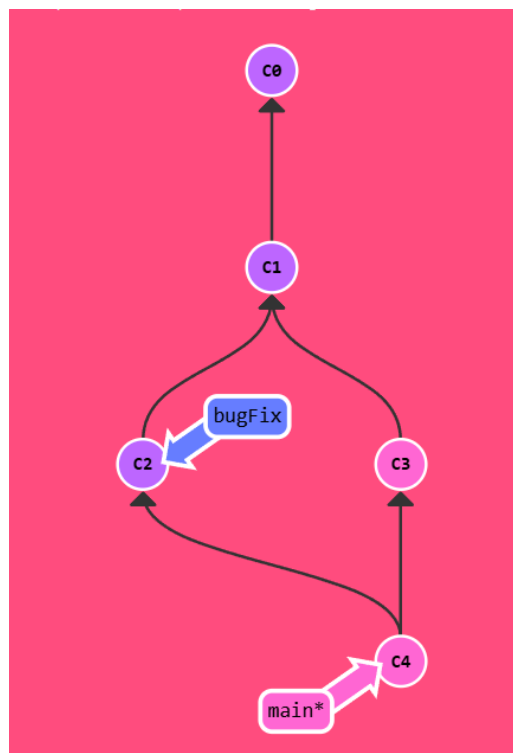


Рис. 2.3. Результат слияния ветки bugFix в main

2.5 Урок 4. Git Rebase

В этом упражнении рассматривался альтернативный способ объединения изменений — `git rebase`. В отличие от `merge`, который сохраняет все ветвления и создаёт дополнительный коммит слияния, `rebase` переписывает историю. Он переносит коммиты одной ветки в конец другой, создавая иллюзию линейной, непрерывной истории разработки.

Главное преимущество `rebase` — более чистая и читаемая история коммитов. Это осо-

бенно полезно при совместной разработке, когда важно понимать, какие изменения в каком порядке были внесены.

Важно: rebase изменяет историю, поэтому его следует применять только к локальным веткам, которые ещё не были опубликованы.

Задание

Для выполнения задания требовалось сделать следующее:

1. Создать ветку bugFix: `git branch bugFix`
2. Переключиться на ветку bugFix: `git checkout bugFix`
3. Сделать один коммит: `git commit`
4. Перейти на ветку main: `git checkout main`
5. Сделать ещё один коммит в main: `git commit`
6. Снова переключиться на bugFix и выполнить команду: `git rebase main`

В результате коммиты из bugFix были перенесены поверх последних коммитов из ветки main, образовав линейную историю без merge-коммита.

Результат

На изображении ниже показано дерево после выполнения команды rebase:

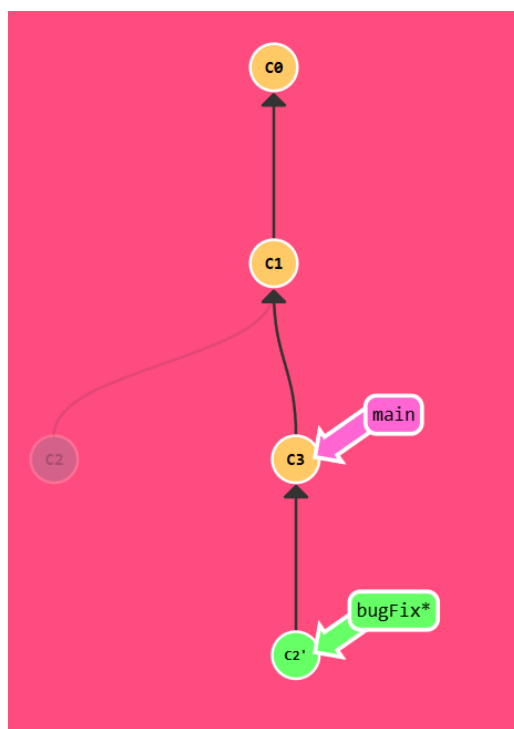


Рис. 2.4. Ветка bugFix после rebase на main

2.6 Урок 5. Прогулка по Git и указатель HEAD

Пятый урок интерактивного курса Learn Git Branching посвящён навигации по дереву коммитов. Понимание структуры коммитов и механизма перемещения по истории изменений — одна из важнейших основ эффективной работы с Git.

Понятие HEAD

HEAD — это символический указатель на текущий активный коммит. Обычно он указывает на последнюю фиксацию (*commit*) в текущей ветке. При переключении ветки или выполнении команды `checkout` HEAD изменяется и начинает указывать на другую ветку или конкретный коммит.

Когда HEAD указывает на имя ветки, он «привязан» к этой ветке. При создании коммита вместе с ней обновляется и HEAD. Однако Git также позволяет «отцепить» (*detach*) HEAD, напрямую указывая на конкретный коммит по его идентификатору (hash). Это полезно, например, для анализа кода или тестирования в прошлых состояниях проекта.

Задание

Чтобы пройти этот уровень, необходимо:

1. Определить идентификатор (hash) последнего коммита в ветке `bugFix` по визуализации.
2. Отцепить HEAD от ветки `bugFix`, указав конкретный коммит напрямую:

```
git checkout c4
```

После выполнения команды `git checkout <hash>` указатель HEAD больше не будет ссылаться на имя ветки — вместо этого он укажет на конкретный коммит. Это называется *detached HEAD* — «отсоединённая голова».

Результат

На изображении ниже показано, как HEAD был успешно перенаправлен на конкретный коммит по его хешу:

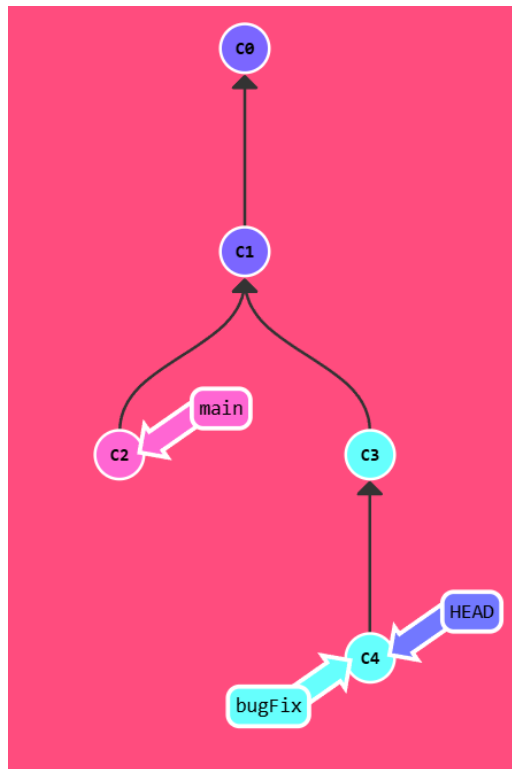


Рис. 2.5. HEAD указывает напрямую на коммит по идентификатору

2.7 Урок 6. Относительные ссылки в Git

В шестом уроке рассматриваются более удобные способы навигации по истории коммитов при помощи относительных ссылок. Работа напрямую с хешами коммитов, особенно длинными, может быть утомительной и неудобной. В реальных проектах идентификаторы коммитов (хеши) могут выглядеть, например, так: `fed2da64c0efc5293610bdd892f82a58e8cbc5d8`.

К счастью, Git позволяет:

- использовать только первые несколько символов хеша — например, `fed2`;
- использовать относительные ссылки, которые значительно упрощают навигацию.

Относительные ссылки в Git

Git поддерживает два базовых способа перемещения по истории:

- `^` — переход на первого родителя текущего коммита. Пример: `HEAD^`
- `<n` — переход на `n` коммитов назад. Пример: `HEAD~2`

Также можно использовать относительные ссылки от имени ветки: `bugFix^`, `main~1` и т.д.

Задание

Для выполнения задания нужно:

1. Перейти на первого родителя ветки `bugFix`, используя относительную ссылку: `git checkout`


```
bugFix^
```

После выполнения этой команды указатель HEAD будет «отсоединён» и направлен на родительский коммит ветки bugFix.

Результат

На изображении ниже показано, как HEAD был успешно перемещён на родительский коммит ветки bugFix, с использованием относительной ссылки:

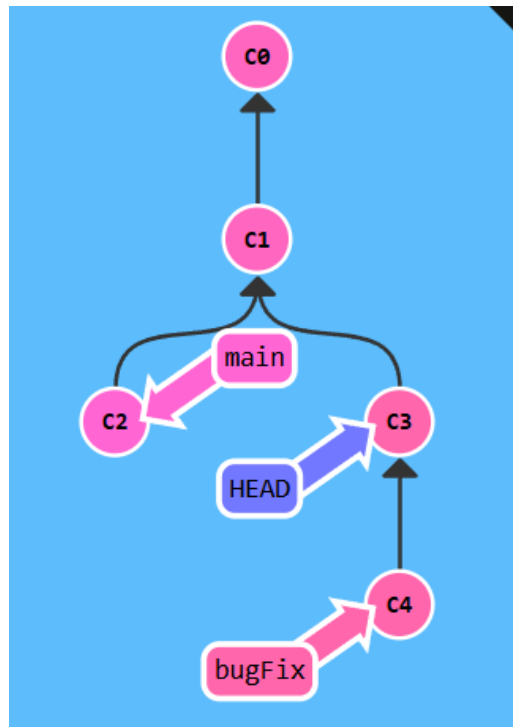


Рис. 2.6. Навигация по истории коммитов при помощи относительной ссылки bugFix^

2.8 Урок 7. Оператор ~ и перемещение ветки (branch forcing)

В этом уроке рассматривался мощный инструмент для работы с историей коммитов — оператор ~ (тильда), а также практическое применение относительных ссылок для перемещения веток.

Принудительное перемещение ветки (branch forcing)

Теперь, когда мы умеем ссылаться на коммиты относительно текущего положения, можно использовать эти знания для изменения положения веток. С помощью команды:

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

можно «перепривязать» ветку main к другому коммиту — в данном случае, к тому, что находится на три коммита позади текущего HEAD. Это называется принудительное перемещение

ветки, или *branch forcing*.

Такая операция не создаёт новых коммитов, а просто перемещает указатель ветки.

Ход решения

В рамках упражнения был выполнен следующий набор команд:

```
git branch -f main C6
```

```
git checkout HEAD~1
```

```
git branch -f bugFix HEAD~1
```

- Сначала ветка `main` была принудительно перемещена к коммиту `C6`.
- Затем выполнен переход на предыдущий коммит через `HEAD~1`.
- Ветка `bugFix` также была перемещена на одну позицию назад от текущего `HEAD`.

Результат

На рисунке ниже представлен результат выполнения операции перемещения веток с использованием относительных ссылок:

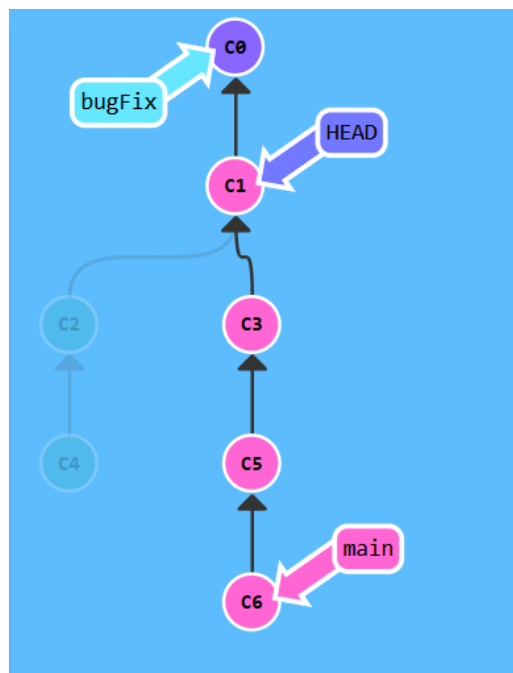


Рис. 2.7. Принудительное перемещение веток `main` и `bugFix` с помощью относительных ссылок

2.9 Урок 8. Отмена изменений в Git

Git предоставляет множество способов для отмены изменений, которые уже были зафиксированы в истории репозитория. Эти методы варьируются от работы с отдельными строками

до полной отмены коммитов. В этом упражнении фокус был сделан на ****высокоуровневых командах**** для отмены изменений.

Git reset и Git revert

Существует два основных способа отката изменений:

- `git reset` — откатывает текущую ветку к предыдущему коммиту, эффективно удаляя последний коммит. Эта операция изменяет историю и применяется в локальных ветках.
- `git revert` — создаёт новый коммит, отменяющий изменения, внесённые предыдущим коммитом. История при этом сохраняется. Применяется, как правило, для отката уже опубликованных изменений.

Цель упражнения

Чтобы пройти этот уровень, необходимо:

- Отменить последний коммит на ветке `local` с помощью `git reset`: `git reset local`
- Отменить последний коммит на ветке `pushed` с помощью `git revert`: `git checkout pushed; git revert pushed`

Результат

На рисунке ниже показан результат выполнения команд, иллюстрирующий различие между `reset` и `revert`:

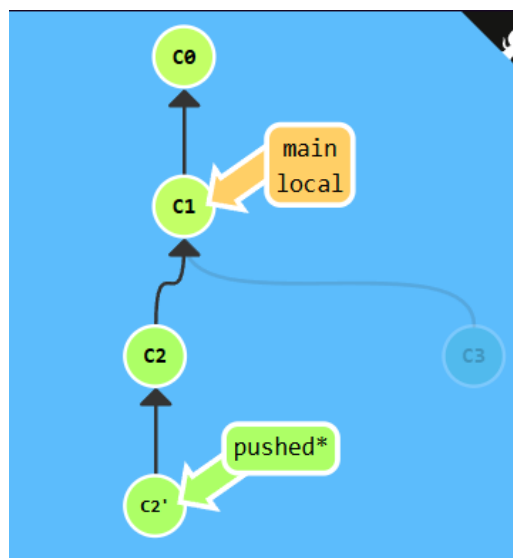


Рис. 2.8. Откат изменений: `reset` удаляет коммит, `revert` создаёт обратный

2.10 Урок 9. Git Cherry-pick

В этом уроке изучалась команда `git cherry-pick`, предназначенная для выборочного применения коммитов из других веток или частей истории к текущей позиции HEAD. Это один из самых прямолинейных способов «вытянуть» нужные изменения, не сливая полностью другие ветки.

Что делает `git cherry-pick`

Команда:

```
git cherry-pick <Commit1> <Commit2> <...>
```

создаёт новые коммиты на текущей ветке, копируя изменения, содержащиеся в указанных коммитах. Это особенно полезно, когда:

- Нужно взять только часть изменений из другой ветки
- Не хочется делать слияние всей ветки
- Нужно воспроизвести конкретные багфиксы или фичи на другой ветке

Цель задания

Визуализация уровня подсказывала, какие именно коммиты нужно было перенести. Задание состояло в том, чтобы:

- Перейти на ветку `main`
- Скопировать изменения из трёх указанных коммитов (с3, с4, с7) в `main`, используя `git cherry-pick`

Ход решения

Для прохождения уровня была выполнена следующая команда:

```
git cherry-pick c3 c4 c7
```

Результат

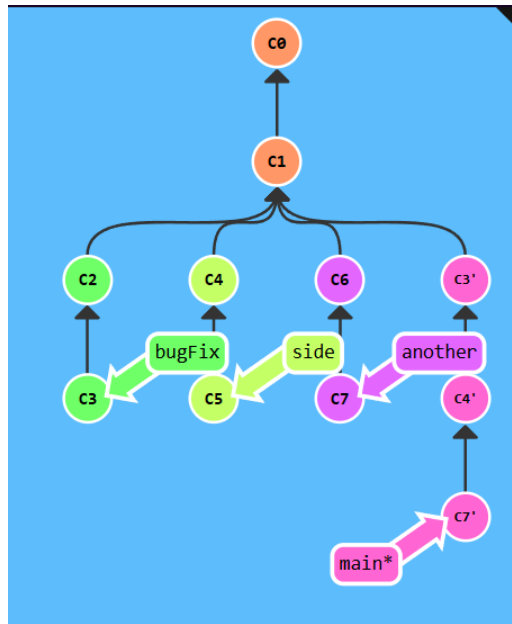


Рис. 2.9. Перенос изменений c3, c4 и c7 в main через `git cherry-pick`

2.11 Урок 10. Git Interactive Rebase

В этом уроке рассматривался продвинутый инструмент Git — **интерактивный rebase**, позволяющий гибко управлять порядком, включением или исключением коммитов. Это особенно полезно, когда необходимо:

- изменить порядок коммитов;
- удалить ненужные коммиты;
- объединить несколько коммитов в один (squash);
- переписать историю коммитов перед публикацией.

Что такое интерактивный rebase

При запуске команды:

```
git rebase -i HEAD~N
```

Git откроет список из последних N коммитов в редакторе. Пользователь может изменить:

- pick на drop — удалить коммит;
- порядок строк — переставить коммиты;
- объединить коммиты при помощи squash (не использовалось в этом упражнении).

Цель упражнения

Задание заключалось в следующем:

- Переставить и отфильтровать коммиты таким образом, чтобы в истории остались только следующие: c0, c1, c3, c4, c5
- Удалить из истории коммит c2
- Изменить порядок остальных коммитов по примеру

Ход выполнения

В редакторе интерактивного ребейза были выполнены следующие действия:

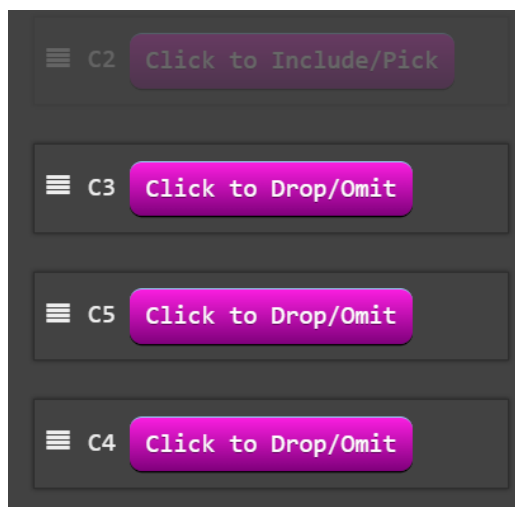
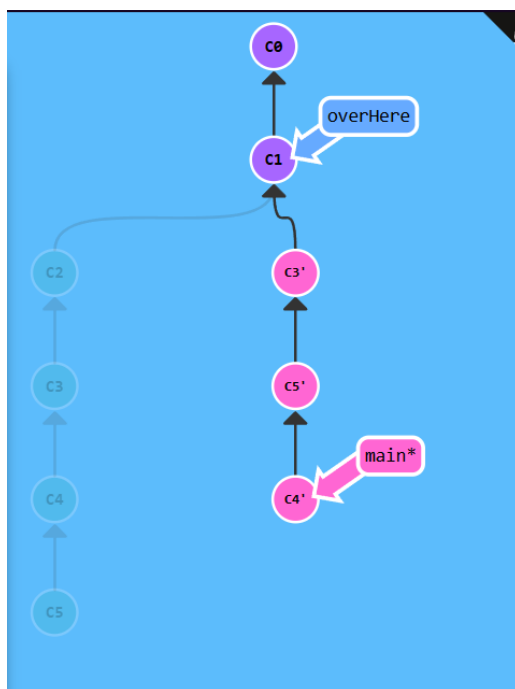


Рис. 2.10. Комит c2 удалён, остальные раставлены в нужном порядке

Результат



2.12 Урок 11. Выборочное извлечение нужного коммита

Ход выполнения

- Использовать `git rebase -i main`, чтобы из ветки `bugFix` оставить только коммит `c4`, содержащий исправление ошибки.
- После очистки ветки `bugFix`, использовать `git rebase bugFix main`, чтобы аккуратно перенести только нужный коммит в ветку `main`.

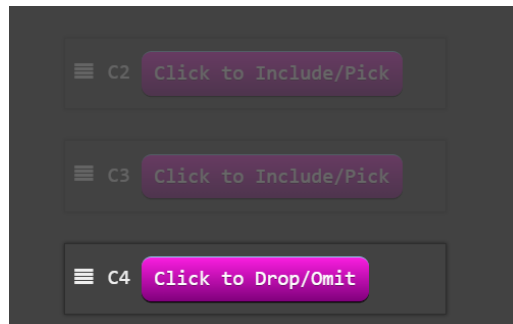


Рис. 2.11. Фото rebase

Результат

Ветка `main` получила только нужный коммит `c4`, в то время как все отладочные коммиты были отброшены и не попали в основную ветку разработки.

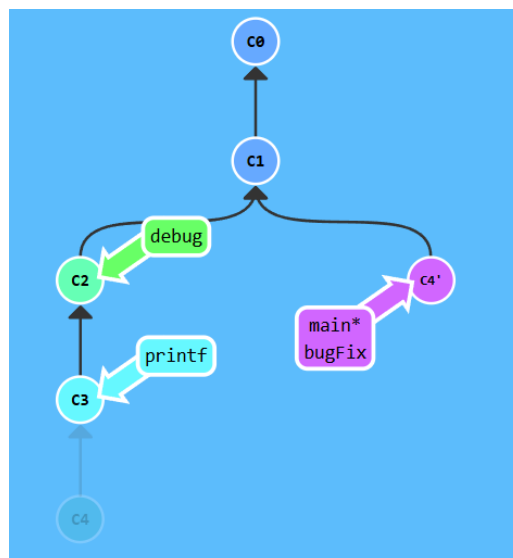


Рис. 2.12. Выборочное добавление коммита `c4` в ветку `main` через `rebase`

2.13 Урок 12. Жонглируем коммитами

Этот урок посвящён тонкому управлению историей Git: как изменить содержимое уже созданного (и не самого последнего!) коммита. Это важный приём, когда необходимо внести исправление в старый коммит, не нарушая целостность истории.

В данном уровне нужно отредактировать caption, хотя он находится *под* коммитом newImage. Для этого приходится «переиграть» историю.

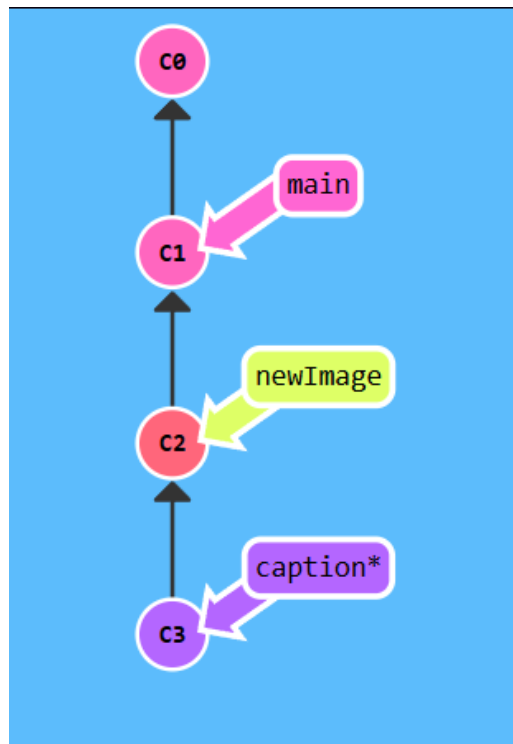


Рис. 2.13. исходные данные

Решение

1. Переставили коммит caption наверх:

```
git rebase -i HEAD~2 # Поменяли местами c3, c2
```

2. Отредактировали содержимое коммита с помощью:

```
git commit --amend
```

3. Вернули коммиты в исходный порядок:

```
git rebase -i HEAD~2
```

4. Переместили ветку main на актуальную часть истории:

```
git rebase caption main
```


Результат

Ветка `main` теперь включает отредактированный коммит `caption`, при этом структура истории остаётся линейной и понятной.

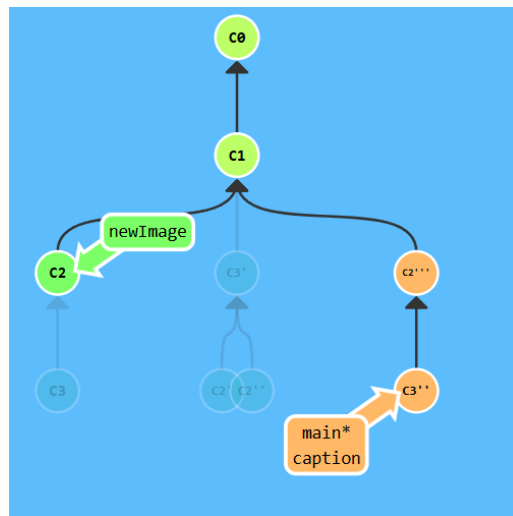


Рис. 2.14. Редактирование старого коммита с помощью `rebase` и `amend`

2.14 Урок 13. Жонглируем коммитами №2 (через `cherry-pick`)

В этом задании стояла цель достичь того же результата, что и в предыдущем уроке, но без использования интерактивного `rebase`. Вместо этого предлагалось использовать альтернативные команды Git — в частности, `git cherry-pick` и `git commit --amend`.

Ход выполнения

1. Перешёл на ветку `main`:

```
git checkout main
```

2. Взял коммит `C2` и применил его:

```
git cherry-pick C2
```

3. Внёс исправления в только что применённый коммит с помощью:

```
git commit --amend
```

4. Применил коммит `C3`, чтобы восстановить порядок:

```
git cherry-pick C3
```

Пояснение

Команда `git cherry-pick` позволяет избирательно переносить коммиты между ветками или на текущую позицию HEAD. Это удобно, когда необходимо сохранить порядок, но модифицировать определённые шаги.

В отличие от `rebase -i`, данный подход требует явного управления историей вручную, что может быть даже более прозрачно и гибко в некоторых случаях.

Результат

Структура дерева коммитов получилась идентичной предыдущему упражнению, с той лишь разницей, что вместо интерактивного ребейза использовалось ручное "жонглирование" с помощью `cherry-pick` и `amend`.

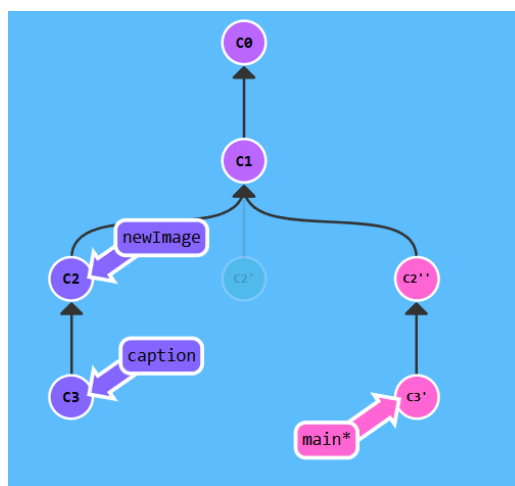


Рис. 2.15. Результат жонглирования коммитами через `git cherry-pick`

2.15 Урок 14. Теги в Git

До этого момента мы изучали ветки как основной способ ссылаться на определённые коммиты. Однако ветки динамичны: они могут легко передвигаться и изменяться. Это делает их не лучшим способом «зафиксировать» важные моменты в истории проекта, например, релизы.

Зачем нужны теги?

Теги (*tags*) в Git — это постоянные метки, указывающие на определённый коммит. В отличие от веток, теги не изменяются, их нельзя сдвинуть, и они служат в качестве «якоря» в истории.

Теги особенно полезны для:

- обозначения стабильных версий (v1.0, v2.3.1 и т.д.),
- возврата к точке релиза или важного изменения,

- публикации в открытом доступе.

Цель урока

Создать два тега:

- `v0` указывает на коммит `C1`,
- `v1` указывает на коммит `C2`.

Затем выполнить переход на тег `v1`, чтобы увидеть состояние проекта на момент этого коммита. Это приведёт к состоянию *detached HEAD*, поскольку нельзя делать коммиты прямо в тег.

Решение

```
git checkout C2
git tag v1 C2
git tag v0 C1
```

После выполнения этих команд:

- Коммит `C1` помечен тегом `v0`,
- Коммит `C2` — тегом `v1`,
- `HEAD` указывает на `v1`, но не на какую-либо ветку (*detached HEAD*).

Результат

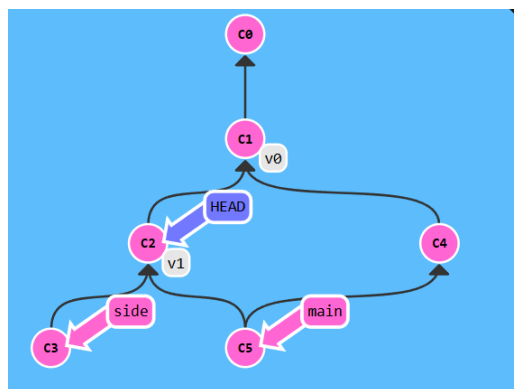


Рис. 2.16. Создание и переход к тегам в Git

2.16 Урок 15. Команда `git describe`

Команда `git describe` — полезный инструмент Git, который позволяет определить, где именно в истории коммитов вы находитесь по отношению к ближайшему тегу. Это особенно удобно после работы с такими командами, как `git bisect`, или при ручной навигации по истории.

Формат вывода

При вызове команды `git describe <ref>` Git возвращает строку формата:

- `<tag>` — ближайший тег, найденный в истории коммитов;
- `<numCommits>` — количество коммитов от тега до текущего состояния;
- `<hash>` — сокращённый хеш текущего коммита.

Если не указать `ref`, Git будет использовать текущий HEAD.

Примеры

В этом упражнении были опробованы следующие команды:

```
git describe main
```

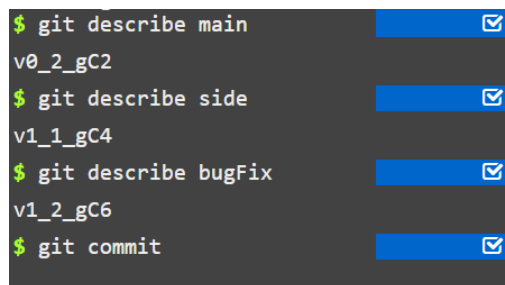
```
git describe side
```

```
git describe bugFix
```

Каждая команда показала расстояние текущей позиции ветки до ближайшего тега. Это помогает понять, в какой части истории находится ветка, особенно если вы работаете с длинной цепочкой коммитов.

Заключение

После того как были выполнены команды `git describe` на нескольких ветках, был сделан дополнительный коммит, чтобы пройти уровень.



```
$ git describe main
v0_2_gC2
$ git describe side
v1_1_gC4
$ git describe bugFix
v1_2_gC6
$ git commit
```

Рис. 2.17. Пример использования `git describe` для определения расстояния до ближайшего тега

2.17 Урок 16. Rebase на нескольких ветках

В этом упражнении было необходимо выполнить цепочку операций `rebase` для нескольких веток так, чтобы итоговая история коммитов стала линейной и строго упорядоченной. Это требование особенно важно в больших проектах, где история должна быть чистой, понятной и предсказуемой.

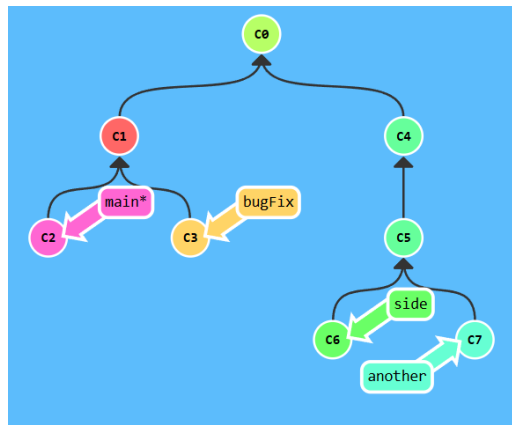


Рис. 2.18. исходные данные

Цель — объединить все изменения из этих веток в ветку `main`, но не с помощью `merge`, а строго через `rebase`, сохранив при этом строгий порядок коммитов: `C6` ', затем `C7` ', и так далее.

Решение

Для достижения цели использовалась последовательность следующих команд:

```
git rebase main bugFix
git rebase bugFix side
git rebase side another
git rebase another main
```

Эти действия:

1. Перенесли ветку `bugFix` поверх `main`,
2. Затем ветку `side` поверх `bugFix`,
3. Затем ветку `another` поверх `side`,
4. И, наконец, весь стек веток — обратно в `main`.

Такой подход гарантирует, что каждый коммит будет находиться строго после предыдущего, сохраняя нужный порядок.

Результат

На изображении 2.19 видно, как история проекта стала линейной благодаря последовательному применению `rebase` к каждой ветке.

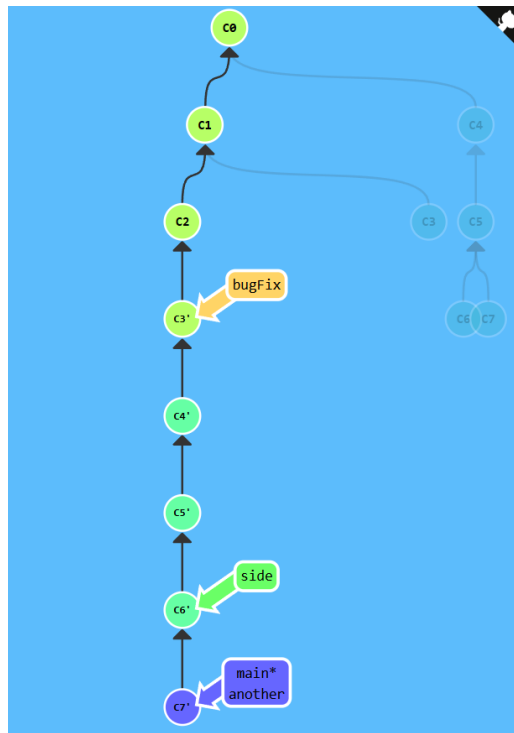


Рис. 2.19. Последовательный rebase нескольких веток на main

2.18 Урок 17. Определение родителей коммита

Git предоставляет гибкие способы навигации по дереву коммитов. В дополнение к тильде (~), которая используется для перемещения назад по цепочке коммитов, также существует оператор каретки (^), позволяющий указать конкретного родителя коммита.

Навигация по родителям

В случае merge-коммитов (слияний), у которых обычно два родителя, Git по умолчанию переходит к первому родителю при использовании ^. Однако, добавляя цифру после каретки, можно выбрать, к какому родителю перейти:

- HEAD^ — первый родитель (по умолчанию),
- HEAD^2 — второй родитель.

Это особенно полезно при анализе истории, откате изменений или создании новых веток от нужного состояния дерева коммитов.

Решение

В качестве решения использована следующая команда:

```
git branch bugWork HEAD~^2~
```

Она означает следующее:

1. HEAD² — перейти ко второму родителю текущего коммита;
2. ~ — перейти на один коммит назад от выбранного родителя;
3. создать ветку bugWork в этом положении.

Результат

На рисунке 2.20 показано, где создаётся ветка bugWork после выполнения команды.

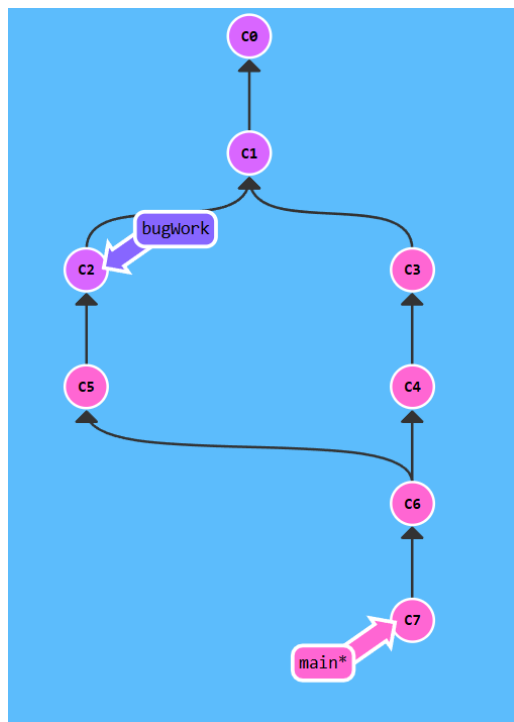


Рис. 2.20. Ветка bugWork, созданная с использованием ² и ~

2.19 Урок 18. Спутанные ветки

Этот уровень представляет собой усложнённую задачу по манипуляции ветками в Git. У нас есть три ветки: one, two и three, каждая из которых нуждается в переработке истории коммитов. Вместо простого ребейза здесь применяются инструменты ручного управления — cherry-pick и branch -f.

Описание проблемы

- Ветка one требует изменения порядка коммитов и удаления коммита C5.
- Ветка two должна включать все коммиты в новом порядке.
- Ветка three должна быть перемещена к одному конкретному коммиту — C2.

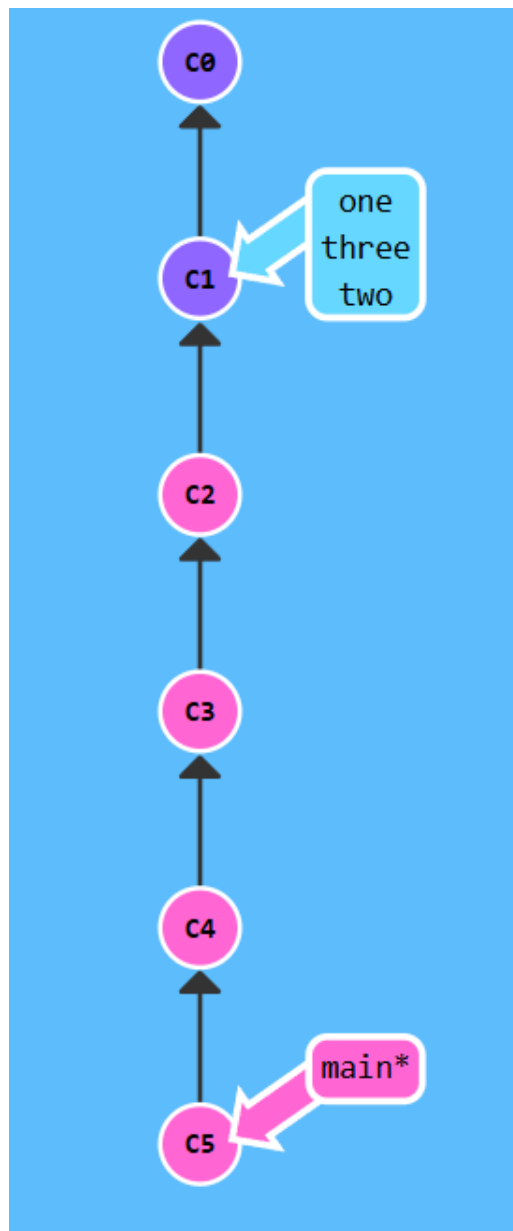


Рис. 2.21. Исходные данные

Решение задачи

Шаг 1. Изменение ветки one:

```
git checkout one
git cherry-pick C4 C3 C2
```

Шаг 2. Изменение ветки two:

```
git checkout two
git cherry-pick C5 C4 C3 C2
```

Шаг 3. Жёсткое перемещение ветки three:

```
git branch -f three C2
```


Результат

Ветка one теперь содержит только C4, C3, C2, без C5. Ветка two включает C5, C4, C3, C2 в указанном порядке. Ветка three указывает строго на C2.

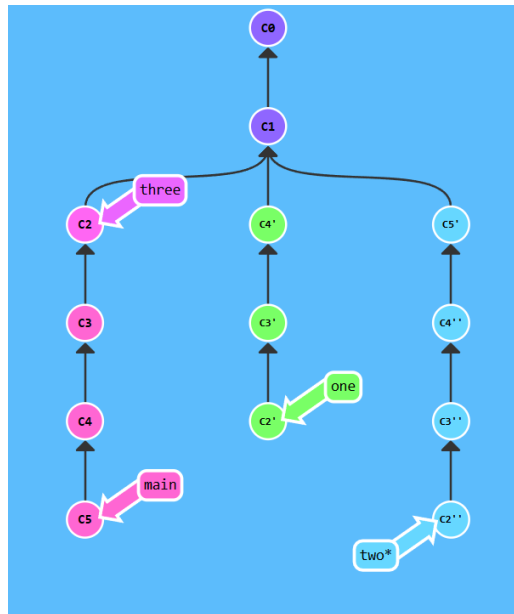


Рис. 2.22. Обновлённые ветки one, two и three после cherry-pick и переопределения

2.20 Урок 19. Удалённые репозитории в Git

Удалённые репозитории в Git — это копии локальных репозиториев, расположенные на других устройствах или серверах, чаще всего с возможностью подключения через Интернет. Они необходимы для совместной работы над проектами и резервного хранения данных.

Что такое удалённый репозиторий?

На практике удалённый репозиторий — это не что иное, как точная копия текущего проекта, доступная по сети. Самые популярные примеры: GitHub, GitLab, Bitbucket.

Зачем нужны удалённые репозитории?

- **Резервное копирование.** Все коммиты и история проекта сохраняются в надёжном удалённом хранилище.
- **Совместная работа.** Разработчики могут вносить изменения независимо и синхронизироваться с помощью pull и push.
- **Интеграция с инструментами.** GitHub, GitLab и другие платформы предоставляют графический интерфейс, CI/CD, issue tracker и многое другое.

Команда для создания удалённого репозитория

В реальных условиях команда `git clone` используется для создания локальной копии уже существующего удалённого репозитория. Однако в рамках симуляции на сайте `learngitbranching.js` команда `git clone` делает наоборот — создаёт удалённый репозиторий из текущего локального, чтобы потренироваться в работе с такими хранилищами.

Команда

```
git clone
```

Итог

После выполнения этой команды в симуляторе создаётся зеркальный удалённый репозиторий. Это позволит в следующих уроках практиковаться с отправкой (`push`) и получением (`pull`) изменений.

Рис. 2.23. Удалённый репозиторий после выполнения команды `git clone`

2.21 Урок 20. Удалённые ветки в Git

После знакомства с командой `git clone` важно понять, что происходит с ветками в локальном репозитории.

Что такое удалённые ветки?

При выполнении `git clone` в локальном репозитории появляется новая ветка с именем `o/main`. Такой тип веток называется *удалёнными ветками* и отражает состояние веток на удалённом репозитории.

Удалённые ветки:

- Отражают состояние удалённого репозитория на момент последнего обращения к нему.
- Позволяют сравнивать локальные изменения с удалёнными.
- Не позволяют напрямую работать в них — для внесения изменений необходимо создать локальную ветку.

Обозначение удалённых веток

Имя удалённой ветки всегда состоит из двух частей:

<имя удалённого репозитория>/<имя ветки>

Например, в ветке `o/main` `o` — это имя удалённого репозитория, а `main` — имя ветки. В реальных проектах удалённый репозиторий часто называется `origin`.

Отделение HEAD при работе с удалёнными ветками

При переключении на удалённую ветку происходит *detached HEAD* — Git не позволяет делать коммиты прямо в удалённой ветке. Для работы нужно создать локальную ветку на основе удалённой.

Задача

Для выполнения уровня необходимо:

1. Сделать один коммит на локальной ветке `main`.
2. Переключиться на удалённую ветку `o/main` (которая будет в состоянии *detached HEAD*).
3. Сделать один коммит на ветке `o/main`.

```
$ git commit
$ git checkout o/main
$ git commit
```

Рис. 2.24. Команды

Таким образом наглядно видно, как локальные и удалённые ветки взаимодействуют и обновляются.

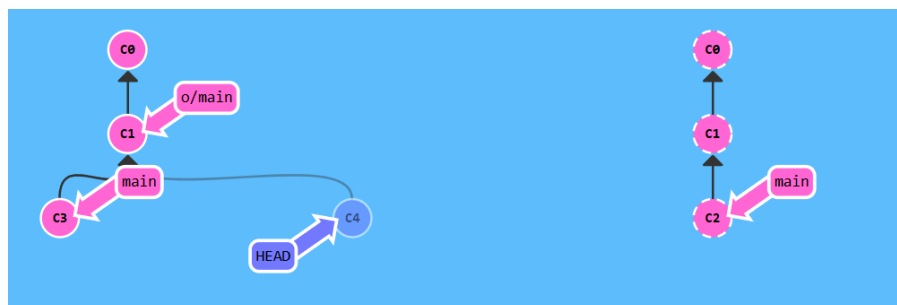


Рис. 2.25. Работа с удалёнными ветками `o/main`

2.22 Урок 21. Git Fetch

Введение

Работа с удалёнными репозиториями Git сводится к обмену данными между локальным и удалённым репозиториями. Это позволяет делиться изменениями, файлами, идеями и другими важными обновлениями.

В этом уроке мы познакомимся с командой `git fetch`, которая отвечает за извлечение данных из удалённого репозитория.

Что делает команда `git fetch`?

Команда `git fetch` выполняет две основные задачи:

- Связывается с указанным удалённым репозиторием и загружает все новые данные, которых ещё нет в локальном репозитории.
- Обновляет ссылки на все ветки из удалённого репозитория (например, `o/main`), синхронизируя локальное представление удалённых репозитория с актуальным состоянием.

Таким образом, `git fetch` обновляет локальную копию удалённых веток, но не меняет локальные ветки автоматически — для этого потребуется дополнительная команда, например, `git merge` или `git pull`.

Цель урока

Чтобы успешно пройти уровень, необходимо выполнить команду:

```
git fetch
```

Эта команда скачает все новые коммиты с удалённого репозитория и обновит ссылки на удалённые ветки.

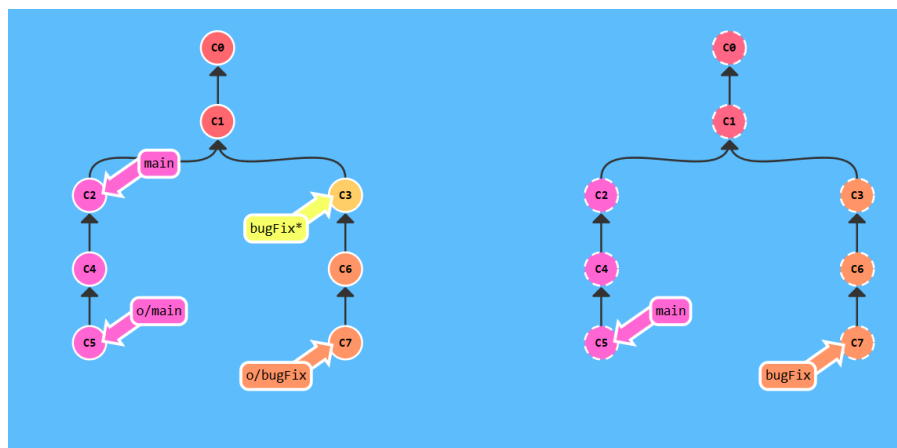


Рис. 2.26. Пример обновления удалённых веток после `git fetch`

2.23 Урок 22. Git Pull

Введение

После того, как мы научились извлекать данные из удалённого репозитория с помощью команды `git fetch`, следующий логичный шаг — обновить локальную работу, чтобы отобразить эти изменения.

Объединение изменений

Получив новые коммиты с удалённого репозитория, можно объединить их с локальной веткой несколькими способами:

- `git cherry-pick o/main` — применить изменения из конкретного коммита;
- `git rebase o/main` — переместить локальные коммиты поверх удалённой ветки;
- `git merge o/main` — объединить удалённую ветку с локальной.

Эти операции позволяют интегрировать обновления из удалённого репозитория в вашу текущую работу.

Команда `git pull`

Процедура скачивания изменений (*fetch*) и последующего объединения (*merge*) встречается очень часто. Для удобства Git объединяет эти действия в одну команду — `git pull`.

Таким образом, команда `git pull` выполняет:

1. `git fetch` — скачивает изменения с удалённого репозитория;
2. `git merge` — объединяет эти изменения с текущей локальной веткой.

Цель урока

Для прохождения уровня достаточно выполнить команду:

`git pull`

Эта команда обновляет локальную ветку последними изменениями из удалённого репозитория.

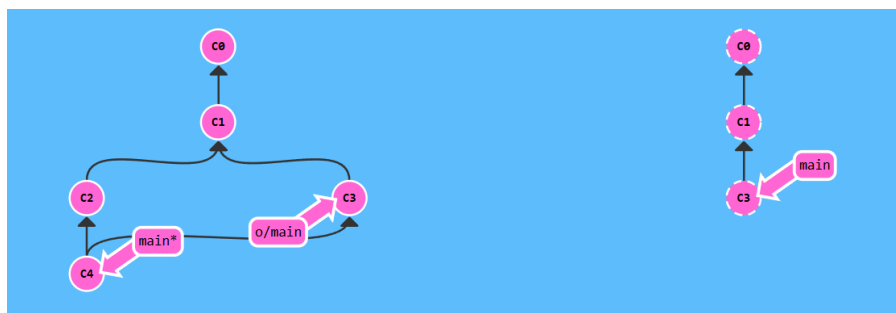


Рис. 2.27. Пример работы команды `git pull`

2.24 Урок 23. Симуляция совместной работы

В этом уроке имитируется ситуация, когда удалённый репозиторий изменён другими участниками команды. Для отработки навыков синхронизации локальных изменений с удалёнными

нам предлагается использовать команду `git fakeTeamwork`, которая создаёт такие изменения.

Для выполнения задания необходимо:

1. Клонировать репозиторий командой `git clone`.
2. Смоделировать изменения коллег с помощью `git fakeTeamwork main 2`.
3. Сделать локальный коммит.
4. Выполнить `git pull` для слияния удалённых изменений с локальными.

Таким образом, этот урок помогает понять, как работать с обновлениями, сделанными другими в удалённом репозитории, и синхронизировать локальную работу.

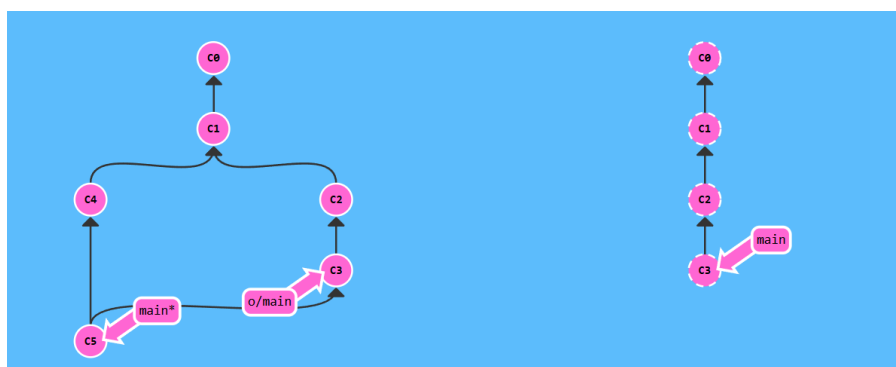


Рис. 2.28. Пример симуляции командной работы

2.25 Урок 24. Git Push

После того как мы скачали изменения из удалённого репозитория и объединили их с локальными, следующим шагом является публикация своих изменений обратно в удалённый репозиторий. Для этого используется команда `git push`. Команда `git push` загружает локальные коммиты в удалённый репозиторий, делая их доступными для других участников проекта. Это обратная операция по отношению к `git pull`. В ходе урока необходимо было сделать два локальных коммита и выполнить `git push`, чтобы опубликовать изменения в удалённом репозитории. Важно учитывать, что поведение `git push` без аргументов зависит от настройки `push.default`, которая обычно устанавливается в значение `upstream`. Таким образом, `git push` позволяет делиться своей работой и поддерживать проект в актуальном состоянии для всей команды.



Рис. 2.29. Пример Git Push

2.26 Урок 25. Когда наработки расходятся

В данном уроке мы рассмотрели ситуацию, когда локальные и удалённые наработки расходятся, то есть история коммитов в репозиториях не совпадает. Это частая проблема при совместной работе, когда несколько разработчиков вносят изменения параллельно. Чтобы избежать конфликтов при объединении таких изменений, в уроке была использована команда `git pull --rebase`, которая позволяет "переписать" локальные коммиты поверх обновлённой истории удалённого репозитория. Это помогает сохранить более чистую и линейную историю изменений.

Последовательность действий была следующей:

- клонирование репозитория (`git clone`)
- имитация изменений в удалённом репозитории (`git fakeTeamwork`)
- локальный коммит
- обновление локального репозитория с применением ребейза (`git pull --rebase`)
- публикация изменений в удалённый репозиторий (`git push`)

Таким образом, использование `git pull --rebase` помогает правильно интегрировать локальные изменения с удалёнными, минимизируя возможные конфликты.

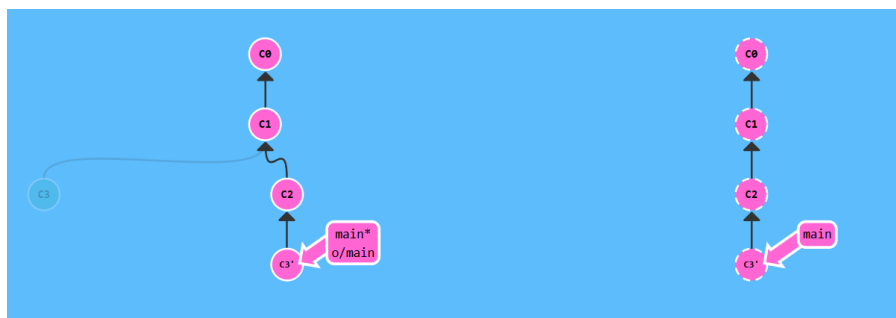


Рис. 2.30

2.27 Урок: Remote Rejected!

В этом уроке рассматривается ситуация, когда при попытке загрузить (push) изменения в ветку `main` удалённого репозитория возникает ошибка отклонения. Это происходит потому, что в удалённом репозитории для ветки `main` настроена политика, требующая использовать Pull Request для внесения изменений.

Причины отклонения:

- Ветка `main` заблокирована для прямых изменений.
- Все изменения должны вноситься через отдельные ветки и Pull Request.

Решение задачи:

1. Сбросить локальную ветку `main` к состоянию удалённой ветки, чтобы синхронизироваться:

```
git reset --hard o/main
```

2. Создать новую ветку `feature` на основе коммита `C2`:

```
git checkout -b feature C2
```

3. Отправить ветку `feature` на удалённый репозиторий:

```
git push origin feature
```

Таким образом, мы соблюдаем политику работы с веткой `main`, используя отдельную ветку для разработки и Pull Request для объединения изменений.



Рис. 2.31. Remote Rejected

2.28 Выводы для главы 2

В ходе выполнения заданий я подробно изучил основные команды и принципы работы с системой контроля версий Git. Особое внимание я уделил работе с ветками, их созданию,

слиянию и перемещению, а также освоил методы управления историей коммитов через `rebase` и `cherry-pick`. Мне удалось понять, как правильно организовать совместную работу в команде, включая работу с удалёнными репозиториями как скачивать обновления (`fetch`, `pull`), так и отправлять свои изменения (`push`). Особенно полезным оказался опыт работы с конфликтами и ситуациями, когда локальные и удалённые версии расходятся. Я узнал, как правильно восстанавливать состояние веток, создавать новые ветки для разработки новых функций и соблюдать правила работы с защищёнными ветками, используя `pull request` вместо прямого пуша. Выполнение практических заданий на сайте learngitbranching.js.org помогло закрепить теоретические знания и увидеть визуальное представление всех операций, что значительно упростило понимание. В целом, эта работа позволила мне уверенно ориентироваться в Git, что является важным навыком для эффективной командной разработки и поддержки проектов.

III Практическое погружение в Git: Git Immersion

3.1 Lab 1: Настройка Git и Ruby

Цель: настроить Git и Ruby для начала работы.

Перед началом работы с Git необходимо указать имя и email пользователя, а также настроить поведение с переносами строк. Также потребуется интерпретатор Ruby для последующих лабораторных.

Настройка имени и электронной почты

Команды, необходимые для глобальной настройки имени и почты:

```
1 git config --global user.name "k1tetssu"
2 git config --global user.email "sereja.maev05@gmail.com"
```

Эти данные будут отображаться в каждом сделанном вами коммите.

Настройка переноса строк

Для корректной работы с переносами строк:

Для пользователей Windows:

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

Эти параметры обеспечивают правильное преобразование переносов строк при сохранении файлов, избегая лишних изменений при коммитах.

Установка Ruby

Для выполнения некоторых лабораторных заданий необходим установленный интерпретатор Ruby. **Проверка установки Ruby:**

```
1 ruby --version
```

Если Ruby установлен корректно, вы увидите версию установленного интерпретатора.

Вывод

В данной практической работе я изучил и применил на практике два важных инструмента для работы с документами и кодом — \LaTeX и Git. В первой части я освоил основы работы с \LaTeX : познакомился с его синтаксисом, установил необходимое программное обеспечение в среде WSL на Windows и научился создавать хорошо структурированные и профессионально

оформленные документы. Это позволило понять, насколько \LaTeX удобен для создания научных и технических текстов с правильным форматированием. Во второй части я прошёл обучающие уроки по работе с Git, что дало мне понимание, как эффективно использовать систему контроля версий. Я научился создавать коммиты, работать с ветками, выполнять слияния и ребе́йзы, а также управлять удалёнными репозиториями и совместной работой над проектами. Практические задания помогли закрепить знания и подготовили меня к реальной работе с Git в командной среде. В целом, данная работа значительно расширила мои навыки в области подготовки документов и контроля версий, что важно для успешной работы над проектами в профессиональной среде.