Team Git Merge Excercise

This document is written to guide your team through the workshop step by step. In this workshop you will be copying provided code snippets into the repository files, you will not be writing the code from scratch.

Overview

One teammate should fork the given repo and invite the others. Each student takes one feature branch (A, B, C, D) and will copy the provided code for their feature into the repo. An instructor feature branch will then introduce an intentional change that causes merge conflicts. Your job is to use git workflows (branching, merge, PRs) to resolve those conflicts, pass the tests, and merge everything into `main`.

These are the day-to-day workflows used in real software teams. Knowing how to merge, reason about conflicts, and keep tests green is essential for working on any team.

Objectives

- Use forks, remotes, branches, and PRs to collaborate.
- Resolve merge conflicts.

Team Setup

Steps for the Repo Owner:

1. Fork the repository on GitHub.
2. In your fork: Settings -> Collaborators -> add teammates with write access.
3. (Optional but recommended) Protect `main` so merges require PRs, approvals, and passing status checks.

All teammates then clone the team fork and add the instructor repo as `upstream`:

```
git clone https://github.com/<RepoOwner>/<team-fork>.git
git fetch --all --prune
```

Create and push a shared `development` branch (one-time):

```
git checkout -b development origin/main
git push -u origin development
```

Install the project dependencies:

MacOS with Homebrew

```
brew install cmake
```

Ubuntu/Debian

```
sudo apt-get install cmake
```

Windows

- Install CMake installer from https://cmake.org/download/
- During installation, check “Add CMake to the system PATH”

Verify CMake is installed:

```
cmake --version
ctest --version
```

Note that c++17 is recommended. If your compiler is older, you may need to update it.

The development branch serves as the team's central integration point. It's where feature pull requests are merged and continuous integration runs its checks before any changes are promoted to main

Feature work

Each student (A, B, C, D) will create a feature branch from `development` and copy the provided snippet for their role into the specified file(s). The students' job is to place the code, keep signatures consistent, run the tests, and follow the Git workflow.

Roles and where to paste the provided snippets:

- Student A — Implement the inverse derivative
  - src/math.cpp
    - At the beginning of the namespace (place where indicated):

```
double computeInverseDerivative(double x) {
    double deriv = fprime(x);
    if (std::abs(deriv) >= EPS) {
        return 1.0 / deriv;
    }
    return x;
}
```

    - In evaluate(...) (place where indicated):

```
if (modes & DERIVATIVE) {
    result = computeInverseDerivative(result);
}
```

- Student B — Implement the Newton-Raphson iteration
  - src/math.cpp
    - At the beginning of the namespace (place where indicated):

```
double applyNewtonStep(double x) {
    double deriv = fprime(x);
    if (std::abs(deriv) >= EPS) {
        return x - f(x) / deriv;
    }
    return x;  // Guard: if f'(x) ≈ 0, no refinement
}
```

    - In evaluate(...) (place where indicated):

```
if (modes & NEWTON_STEP) {
    result = applyNewtonStep(result);
}
```

- Student C — Implement the definite integral evaluation
  - src/math.cpp
    - At the beginning of the namespace (place where indicated):

```
double computeDefiniteIntegral(double x) {
    return Fantiderivative(x) - Fantiderivative(0.0);
}
```

    - In evaluate(...) (place where indicated):

```
if (modes & INTEGRAL) {
    result = computeDefiniteIntegral(result);
}
```

- Student D — Implement the function value with linearization
  - src/math.cpp
    - At the beginning of the namespace (place where indicated):

```
double computeFunctionValue(double x) {
    // Check if near critical points where f'(x) = 0 (x = ±1)
    if (std::abs(x - 1.0) < DELTA || std::abs(x + 1.0) < DELTA) {
        // Use linearization for stability
        double a = (std::abs(x - 1.0) < DELTA) ? 1.0 : -1.0;
        return f(a) + fprime(a) * (x - a);
    }
    return f(x);
}
```

    - In evaluate(...) (place where indicated):

```
if (modes & VALUE) {
    result = computeFunctionValue(result);
}
```

Create your feature branch and copy code

1. Create your feature branch from `development`:

```
git checkout development
git pull origin development
git checkout -b feature/<descriptive-feature-name>
```

1. Copy the code snippets into the target file(s). Use the placeholder markers above to find the right content and location.

2. Build and run the tests locally:

```
mkdir -p build
cd build
cmake ..
cmake --build .
ctest --output-on-failure
./app
cd ..
```

1. Commit and push your branch:

```
git add src/math.cpp src/math.hpp
git commit -m "Implemented <feature description>"
git push -u origin feature/<your-feature>
```

It's important to run tests locally as soon as possible since they help catch issues such as mismatched function prototypes early on. Identifying these problems right away not only saves time during conflict resolution but also ensures that pull requests ready for review.

Accept the Instructor PR

- At this point, you should accept the instructor pull request
- Make sure to do this before merging your feature branch into `development` or you will be marked off.
- This intentionally creates overlapping edits so that when you merge your feature branch into `development`, you will have to resolve conflicts.

We do this to practice resolving conflicts safely by taking two versions that are each correct in their own way and combining them into one final version that compiles.

Resolve conflicts by merging

When development receives instructor updates, each student must bring their work up to date and resolve any merge conflicts. Below we'll walk through VS Code Source Control (Any Git tool works like GitKraken, GitHub Desktop, JetBrains, etc.)

1. Update your local `development`:

```
git checkout development
git pull origin development   # update your local copy of development
```

1. Merge your feature branch into `development`:

```
git merge feature/<your-feature>   # merge your feature branch into development
```

When Git detects overlapping changes, it inserts `conflict markers` into the affected files (see Figure 1).

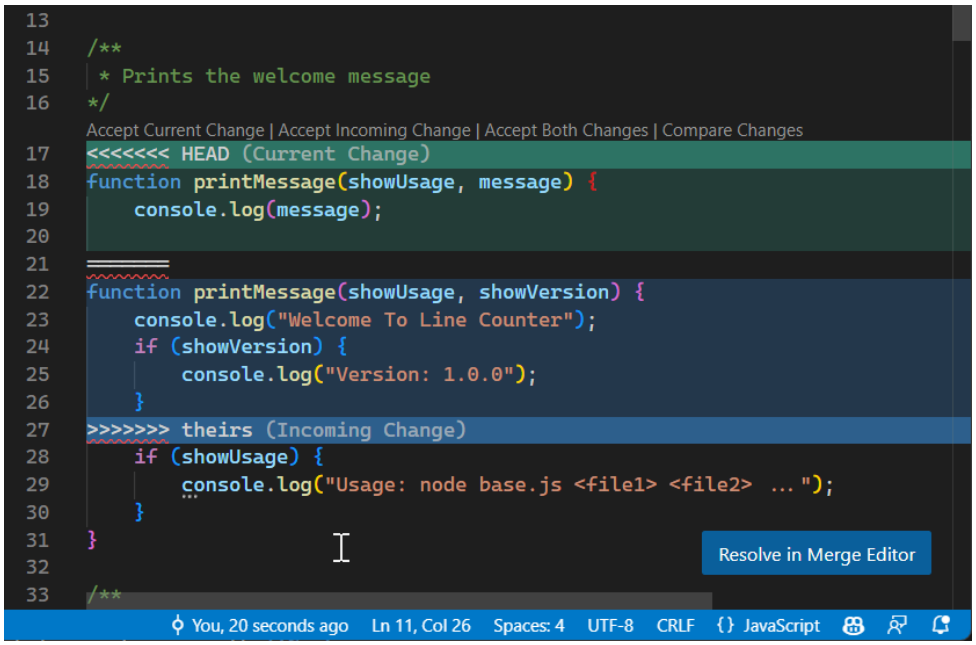

Figure 1: Example of Git conflict markers showing differences between branches. [1]

These markers indicate where the two branches made conflicting changes. The head section shows the `development` version since are checked to it. The bottom section shows your feature branch's changes. You must edit the file to consolidate these changes by deciding how to replace or combine code.

1. Resolve the conflicts

TODO

Reviewing and merging PRs into `dev`

Each student opens a PR with `base = dev` and `compare = feat/your-role`. Use brief descriptions and ask at least one teammate to review and approve. After approval and passing CI, merge into `dev` per your repo rules.

Code reviews are important because they ensure that another person looks over your work, providing an opportunity to catch errors or inconsistencies before any changes are merged into the main branch.

Final integration into `main` and submission

Helpful tips

Closing notes

Good luck and push often.

- NEED a few questions for reflection at the end?

Bibliography

[1] Microsoft, "Using Source Control in VS Code." Accessed: Oct. 15, 2025. [Online]. Available: https://code.visualstudio.com/docs/sourcecontrol/overview