



BEST PRACTICES FOR C++ and GIT

- Busra Sen

- Mechanical Engineering, Control Systems Technology Group

INTRODUCTION



<https://onix-systems.medium.com/how-to-determine-the-number-of-project-team-members-for-your-it-initiative-4282dcba3c74>

Achieve some parts of the task

Add new features

Share your work with other team members to fix a bug or get feedback

After a while



<https://dribbble.com/shots/11086928-Working-girl>

Recall

Understand the code

- *Understand requirements*
- *Plan and task divisions*

INTRODUCTION

What is a bad code?

- Not readable
- No consistency
- Difficult to modify
- Difficult to track

Okay, but the computer understands what I write. So why should I care?



Consequences of bad code

- Lack of motivation
- Missed deadlines
- Decreasing in productivity
- Financial losses

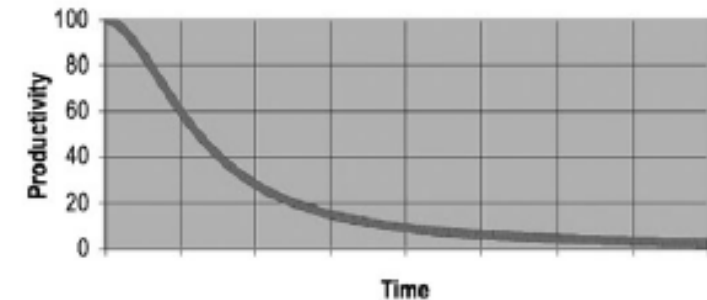


Figure 1-1
Productivity vs. time

Figure from Clean Code book

MOTIVATION

Okay, then let's talk about positive terms such as clean code 😊

- Clean code refers to well-structured, readable, and maintainable code that is easy to understand and change.
- Clean code is not about making the code work; it helps other developers to figure out and collaborate.

Let's summarize what you should expect from this lecture

- Best practices to improve software quality,
- Best practices to avoid pitfalls in C++,
- Best practices for the version control system in a teamwork

CONSISTENT CODE STYLE

- A good code should have a consistent style that covers indentation, braces, line length, spacing, naming convention, etc.

Brace Placement

a) Kernighan and Ritchie Style (K&R)

```
if (x < y) {  
    something();  
}
```

b) Allman Style

```
if (x < y)  
{  
    something();  
    myFunc();  
}
```

c) Whitesmiths Style

```
if (x < y)  
{  
    something();  
    myFunc();  
}
```

CONSISTENT CODE STYLE

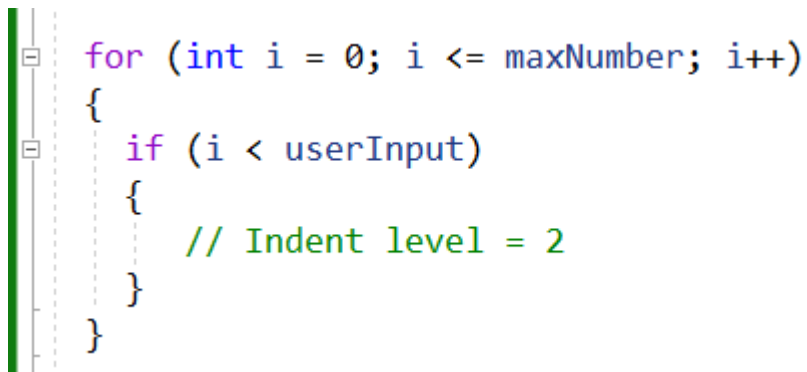
Keep lines at a reasonable length [3]

Lines length should be less than 80 characters per line.

```
if (x && y && hasPrime() && isCostPositive() && hasPositiveNumber && (x > y || y < z))
{
    // Hard to follow
}
```

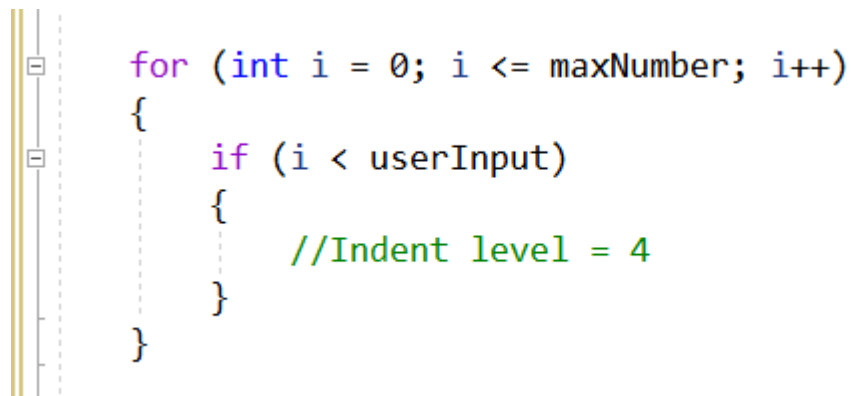
```
if (x && y && hasPrime() && isCostPositive()
    && hasPositiveNumber
    && (x > y || y < z))
{
    // Logical groups for reasonable line length
}
```

Indentation Level



A code snippet showing a for loop with an inner if statement. The inner if statement is indented by two levels relative to the for loop. A vertical green line on the left indicates the indentation level, with a dashed line showing the alignment of the inner if statement's opening brace.

```
for (int i = 0; i <= maxNumber; i++)
{
    if (i < userInput)
    {
        // Indent level = 2
    }
}
```



A code snippet showing a for loop with an inner if statement. The inner if statement is indented by four levels relative to the for loop. A vertical yellow line on the left indicates the indentation level, with a dashed line showing the alignment of the inner if statement's opening brace.

```
for (int i = 0; i <= maxNumber; i++)
{
    if (i < userInput)
    {
        //Indent level = 4
    }
}
```

CONSISTENT CODE STYLE

Horizontal Whitespace

- No space is left between the unary operator and its operand

```
++x;  
!isDoorOpen();  
*pName;
```

- Leave space between binary operator and operands

```
subtract = number1 - number2;  
if(x <= y)
```

- Leave space after each comma and semicolon. No space before a comma or semicolon

```
for (i = 0, j = 0; i < userInput; i++, j++)
```

Consistent Naming Style

value
myVariableName
myFuncName



https://en.wikipedia.org/wiki/Camel_case



https://commons.wikimedia.org/wiki/File:Emoji_u1f40d.svg

value
my_variable_name
my_func_name

Best practice

It is recommended for the team decide on a consistent code style that is accepted and followed by all members.

COMMENTS

- Transfer the intent of the code to humans such as new developers, and new team members.
- If you want to add a comment to explain your code, you should think to rewrite that code.

- **Redundant Comments**

- Do not comment on obvious things: You should not violate the DRY principle

```
// Calculate the distance
float distance = sqrt(pow(x2 - x1, 2) +
    pow(y2 - y1, 2)); // Distance formula

++counter; // increment counter
```

- Do not disable code with comments. Or do not use comments to substitute version control

Commented-out code

```
// This function is no longer used
/*
int computeManhattanDistance(const Point& p1, const Point& p2)
{
    int distance = abs(p1.x - p2.x) + abs(p1.y - p2.y);
    return distance;
}
*/
```

Tracking teammate activities

```
//
*****
// [01-04-2023] X fixed the bugs ...
// [05-05-2023] Y add feature ...
// [10-05-2023] Z remove function ..
//*****
```

Use a version control system! 😊

COMMENTS

Good Comments

- Make sure that your comments add value to the code
- Highlight design decisions and assumptions
- Explain always why, not how!
- Try to be as short and expressive as possible

```
// Compute heuristic function using Manhattan
distance - move in four directions

std::vector<GridPosition> findOptimalPath(const
GridPosition& start, const GridPosition& goal)
{
    // ...
}
```

Null Statement

```
for (int i = 0; i <= SIZE; i++)
    ; // null statement
```

Best practice

Critically evaluate the necessity of each comment

```
// Make sure to call initializeRobot()
before running this function.
void performTask() {
    // ...
}
```

```
// Decrease the joint angle within the
valid range
if (jointAngle < 0) {
    jointAngle = 0; // Ensure the joint
angle is non-negative
}
```

```
for (int i = 0; i <= SIZE; i++);
```

MEANINGFUL NAMES

- An identifier is a name used to show a variable, object, class, structure, function, etc.
- There are certain rules for naming [7]:
 1. Keywords cannot be used for identifier
 2. The identifier can include letters, numbers, and the underscore character,
 3. The identifier can begin with a letter or underscore character,
 4. The identifier is case-sensitive: the number is different from NUMBER

*Why do we need **good** names?*

- To increase your code readability
- To Reveal your intention to your teammates quickly,
- To recall what you intended after a few weeks/months,

MEANINGFUL NAMES

Reveal their intention

- A good name tells you why it exists [1].

```
int tm; // elapsed time in minutes
unsigned int num;
Sensor data;
```

```
int elapsedTimeMinutes;
unsigned int numOfDoors;
Sensor distance;
```

- You shouldn't need an explanation for a variable name.
- Data, num, flag, and list are not good names [2]. Which data do you mean?

Avoid exhaustive variable names

- No doubt, it is clear! But readability?
- Not easy to remember
- Difficult to type

```
double thisVariableShowsTheReadingFromTheInfraredDistanceSensorLocatedAtTheRearOfTheRobot;
double thisVariableShowsTheReadingFromTheInfraredDistanceSensorLocatedAtTheFrontOfTheRobot;
```

MEANINGFUL NAMES

Do not use I and O for your variable names

- Readability depends on fonts. But if you do not really need these variable names, do not use them!

```
#include <iostream>
using namespace std;

int main() {
    int l = 50;
    int result;

    if (l == 50) {
        result = l + 1;
    }
    else {
        result = l + 1;
    }

    cout << "Result: " << result << endl;
    return 0;
}
```

Avoid redundancy when choosing a name

- Do not repeat class names in their attributes

```
class Sensor {
private:
    std::string sensorName;
    // ...
};
```

- Do not repeat attribute type

```
class Sensor {
private:
    std::string stringName;
    // ...
};
```

Other Suggestions

- Do not use abbreviations, unless they are commonly known,
- Avoid using the same name for different purposes,
- If a variable name is widely used, make it more descriptive. Usually, it depends on the scope.

FUNCTIONS

```
returnType functionName(parameterType parameter)
{
  // Function Body
}
```

One Thing, No More!

Functions should do one thing. They should do it well. They should do it only.

—Robert C. Martin, *Clean Code* [Martin09]

Signs that your function does more than one thing [2]:

1. The function is too large (*Ideally 4-5 lines, maximum of 12-15 lines.*)
2. You cannot avoid using conjunctions such as “and ” or “or” to build the function name
3. The body of the function is vertically separated using empty lines
4. Function contains many conditional statements
5. The function has many arguments, especially flag arguments of type bool

- Clear, expressive, and self-explanatory
- Function name begins with a verb!
- The function name explains intention, not how it works.

AVOID MAGIC NUMBERS

- A magic number is any value that is used in code without a clear explanation of what it represents.
- This can be any literal such as strings, characters, integers, etc.

```
for (int i = 0; i < 10; i++)  
{  
    // generate motion plan for next waypoints  
    // ...  
    setMaxSpeed(1.5);  
    setMinDistance(0.1);  
}
```

- 10, 1.5, and 0.1 are magic numbers.
- The magic number makes the code difficult to read, understand and modify.
- What is 10?
- Should I change each 1.5 in my code?

What can we use instead of magic numbers?

```
const int waypointsNum = 10;  
const double maxSpeed = 1.5;  
const double minDistance = 0.1;  
  
for (int i = 0; i < waypointsNum; i++)  
{  
    // generate motion plan for next waypoints  
    // ...  
    setMaxSpeed(maxSpeed);  
    setMinDistance(minDistance);  
}
```

What about MACROS?

AVOID MACROS

- A macro is a feature that decides how the input text will be transformed into the output text.
- Two types: object-like macros, and function-like macros

```
#define PI 3.14159265358979 // object-like macro
#define SQUARE(x) (x) * (x) // function-like macro
#define MIN(a,b) ((a)<(b))? (a):(b) // function-like macro
```

```
#include <iostream>
#define SQUARE(a) ((a)*(a))

int main()
{
    int x = 10;
    int y = SQUARE(x++);

    std::cout << "x is: " << x << std::endl;
    std::cout << "y is: " << y << std::endl;

    return 0;
}
```

Undefined behavior :
(x++) * (x++);

x is: 11
y is: 100

Macros violate
argument-passing
rules!

```
#include <iostream>

template<typename T>
T square(T a) { return a * a; }

int main()
{
    int x = 10;
    int y = square(x++);

    std::cout << "x is: " << x << std::endl;
    std::cout << "y is: " << y << std::endl;

    return 0;
}
```

AVOID MACROS

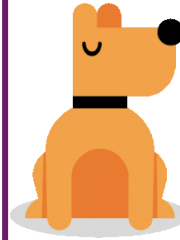
- Macros obey scope rules

```
#include <iostream>

void checkSomething()
{
    // ...
    int variable = 10;
    #define FLAG 1
    // ..
}

int main()
{
    std::cout << FLAG << std::endl;

    return 0;
}
```



Although macro is inside a function, its scope is not like function scope.

- If you still want to use macros :

- Avoid defining macros in a header file.
- Define macros immediately before their usage and remove the definitions (`#undef`) after.
- Select unique names for macros to prevent conflicts with other names.

ALWAYS INITIALIZE VARIABLES

- Initialization refers to providing an object with a known value at the moment of its definition.
- Assignment, on the other hand, involves assigning a known value to an object after its initial definition.
- If an object is uninitialized, it means that it has not been assigned a known value yet.

Which value is assigned to an uninitialized local variable?

```
#include <iostream>

struct Position
{
    double x;
    double y;
};

int main()
{
    Position robotPosition;
    std::cout << robotPosition.x << '\n';

    return 0;
}
```

UNDEFINED BEHAVIOR!

Assigning a meaningful value to a variable as close as possible to its declaration is considered a best practice.

```
int x = 1;
// ... no use of
// ... use of x
// ... here
++x;
```

Best practice

Always initialize variables.
Initialize variables, when you need them!

AVOID (non-const) GLOBAL VARIABLES

- Global variables have a file scope
 - Increase the complexity of debugging
 - They can be written and read from anywhere
 - Make the code harder to understand
- But maybe you need global variables, then some suggestions:
 - Choose global variables names beginning with “g” to clarify that this is a global variable
 - Define your global variables inside a namespace to avoid naming conflicts

```
namespace constants
{
    const double maxTranslationalSpeed = 0.5;
    const double maxRotationalSpeed = 1.2;
}
```

- Document the purpose of global variables using comments

Best practice

If it is possible, prefer local variables over global variables.

AVOID MIXING SIGNED AND UNSIGNED INTEGERS

- Unsigned integers hold non-negative numbers.
- n-bit unsigned integer has a range of $(2^n)-1$

```
#include <iostream>
```

```
int main()  
{
```

```
    unsigned int x = 4294967295;  
    unsigned int y = -1;  
    unsigned int z = 4294967296;
```

```
    std::cout << "x is " << x << '\n';  
    std::cout << "y is " << y << '\n';  
    std::cout << "z is " << z << '\n';
```

```
    return 0;
```

```
}
```

```
Microsoft Visual Studio Debug Console
```

```
x is 4294967295  
y is 4294967295  
z is 0
```

```
#include <iostream>
```

```
int main()  
{
```

```
    unsigned int x{ 3 };  
    unsigned int y{ 5 };
```

```
    std::cout << "x - y is " << x - y << '\n';
```

```
    return 0;
```

```
}
```

```
Microsoft Visual Studio Debug Console
```

```
x - y is 4294967294
```

```
#include <iostream>
```

```
int main()  
{
```

```
    int x{ -3 };  
    unsigned int y{ 5 };
```

```
    std::cout << std::boolalpha << (x < y) << '\n';
```

```
    return 0;
```

```
}
```

FALSE!

Best practice

Be careful when you use unsigned integers. Avoid mixing signed and unsigned integers!

USE EXPLICIT NAMESPACES

```
#include <iostream>

int add(int x, int y)
{
return x + y;
}

int main()
{
std::cout << add(2, 3);

return 0;
}
```

```
int add(int x, int y)
{
return x + y;
}
```

Linkage Error

```
#include <iostream>

int add(int x, int y)
{
return x + y;
}

int add(int x, int y)
{
return x + y;
}

int main()
{
std::cout << add(2, 3);

return 0;
}
```

Compile Error

- When the compiler or linker cannot distinguish between two identical identifiers, an error known as a "naming conflict" will occur [7].
- Use namespaces (scope region) to avoid naming conflicts.

USE EXPLICIT NAMESPACES

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    std::cout << add(2, 3);
    return 0;
}
```

- Std::cout -> cout function that lives inside std (standard) namespace.

Two representations

Explicit Namespaces

Using :: scope resolution operator

Using directives

using namespace NAME;

Using-directive may cause conflicts between any identifiers we define and the namespace's identifier. For example, maybe today you guarantee that there is no conflict between your identifier and standard library, but what about the next revisions to the language [7]?

Best practice

Prefer explicit namespaces over using directives.

```
#include <iostream>
using namespace std;

int cout()
{
    return 0;
}

int main()
{
    cout << "Hello world" << endl;
    return 0;
}
```

Compile Error

GIT BEST PRACTICES

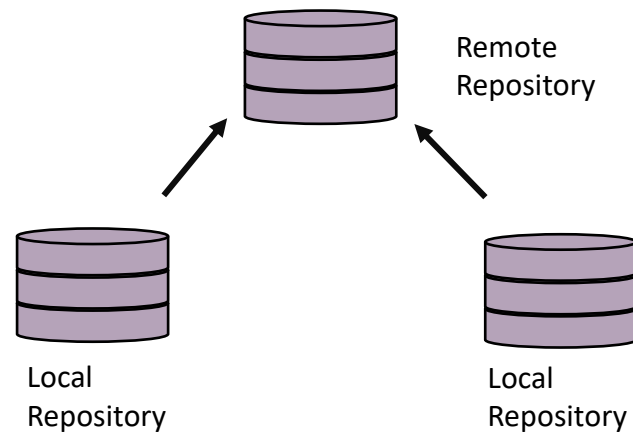
Best practice

Use a version control system for team collaboration!

- Git is a distributed version control system that is open-source and enables you to monitor modifications made to a file or a set of files.

What is the significance of Git?

- Maintain an organized project history.
- Easily revert to previous versions of the project.
- Track and attribute changes made by individuals.
- Facilitate independent work on the project.
- Safely experiment with new features without impacting the main codebase.
- Enhance code quality through code reviews.



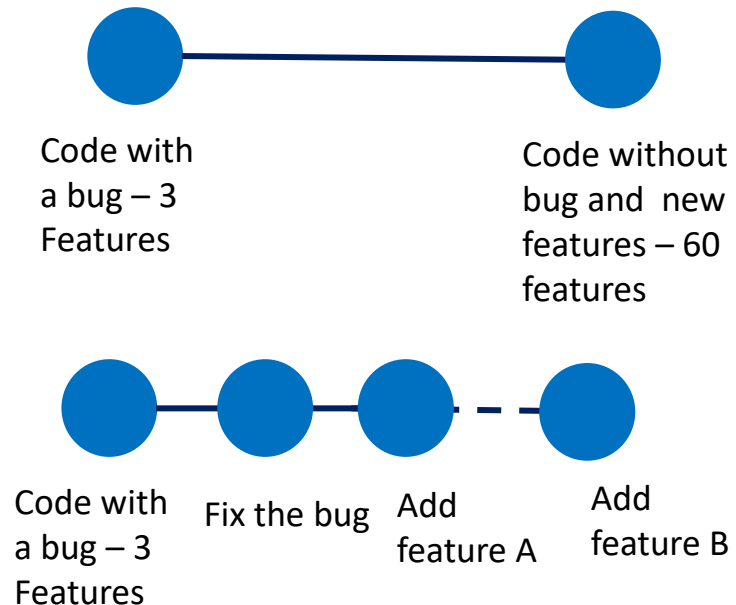
A GOOD COMMIT MESSAGE

A commit is a snapshot of the project.

- To remember what you did in the past,
- To reveal why these changes were made
- To track what others did on the project

Best practice

You should commit often!



Best practice

Good commit messages are structured like emails [5].

First line: Like a subject of an email

- Use imperative commands: such as fix, update, remove, add, etc.
- Explain what you changed
- Keep it small it should not exceed 50 characters

A blank line

- If you want to give more explanations, add a blank line between the subject of the message and the body of the message.

Message body

- Message should include why the change was made
- Each line must be less than 72 characters.
- You can use bullet points
- Use present tense
- No restrictions about the length of the message

A GOOD COMMIT MESSAGE

Bad commit messages:



Fix bugs

Busra Sen authored just now

Which bugs?



Update code

Busra Sen authored just now

Why?



Add Euclidian distance. We will need a distance from goal position to...

Busra Sen authored just now

Too long!

```
$ git log
commit f361fdc3261efb773a0448246e06820a7b08dfffc (HEAD -> featureX)
Author: Busra <b.sen@tue.nl>
Date: Mon May 1 23:57:31 2023 +0200

    Add Euclidian distance. We will need a distance from goal position to succes
    sor. Therefore, I computed
    Euclidian distance, this will be computed during search!
```

Good commit message



Add function to compute Euclidean distance heuristic

Busra Sen authored 2 minutes ago

```
* The function name is euclidianDistance
* It takes two nodes as input to estimate the distance
  from the current node to goal node
```

```
commit 7700dccf3de5568c0e9573a8284f3b6c09860cef (HEAD -> featureX)
Author: Busra <b.sen@tue.nl>
Date: Tue May 2 00:15:10 2023 +0200
```

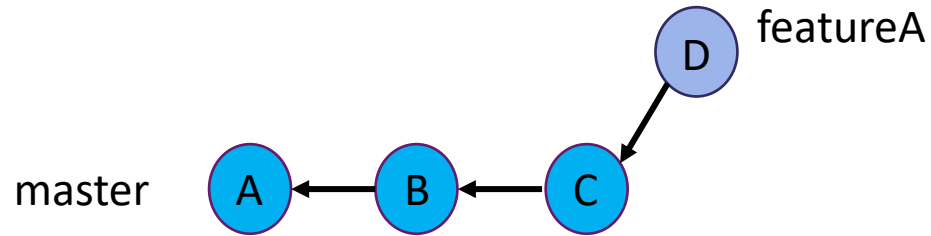
```
    Add function to compute Euclidean distance heuristic

    * The function name is euclidianDistance
    * It takes two nodes as input to estimate the distance
      from the current node to goal node
```

BRANCHES

The branch is a separate line of development that diverges from the other lines.

Each commit belongs to a branch.



Question:

Which commits belong to the featureA?

Which commits belong to the master?

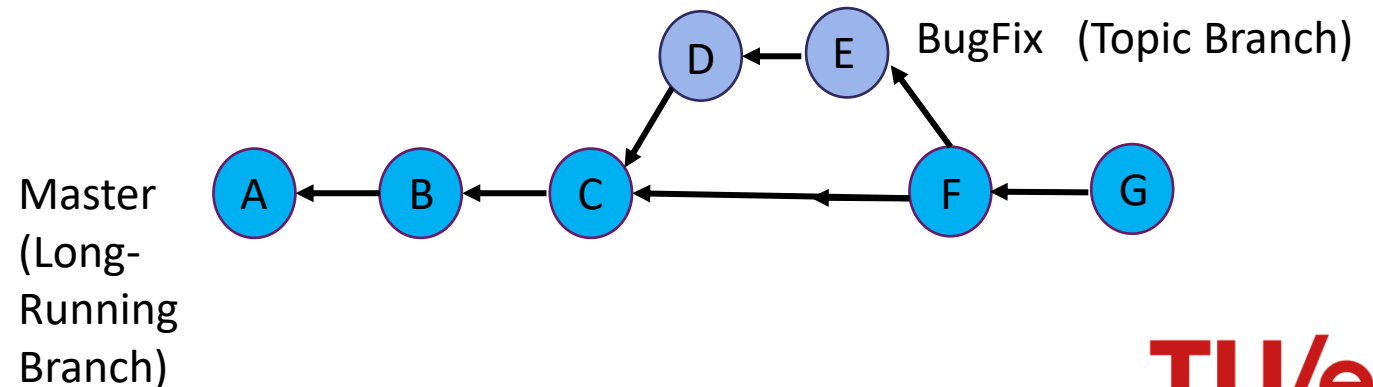
Best Practice

Clearly label and name your branches for easy identification.

Why should we use branches?

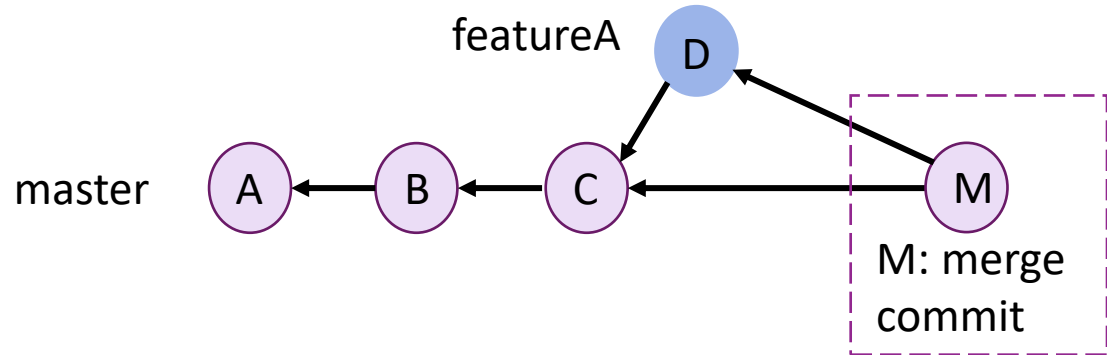
- Provides experimentation
- Multiple team members can work independently
- Keep multiple versions of the project

Topic and long-running branches



MERGING

- Merging is to combine independent branches.
- Usually, a topic branch is merged with a long-running branch such as a master branch!

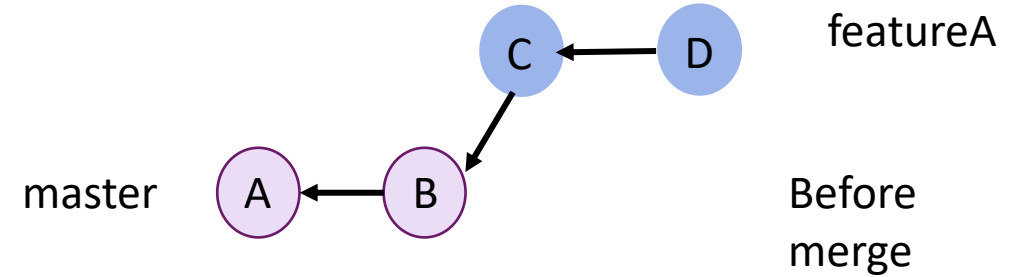


Commit M includes commits A, B, C, and D.

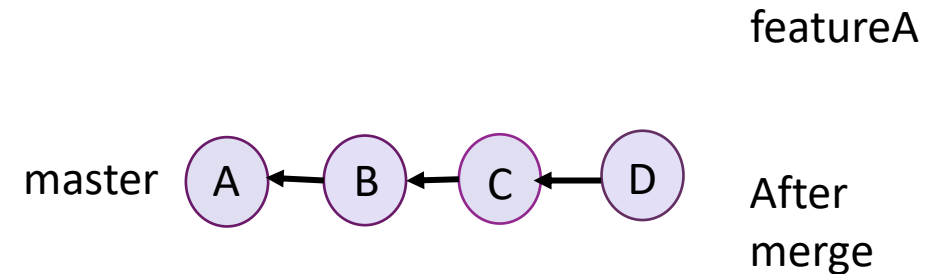
Main types of merges:

- Fast-forward merge
- Merge commit
- Squash merge
- Rebase

Fast-forward merge:



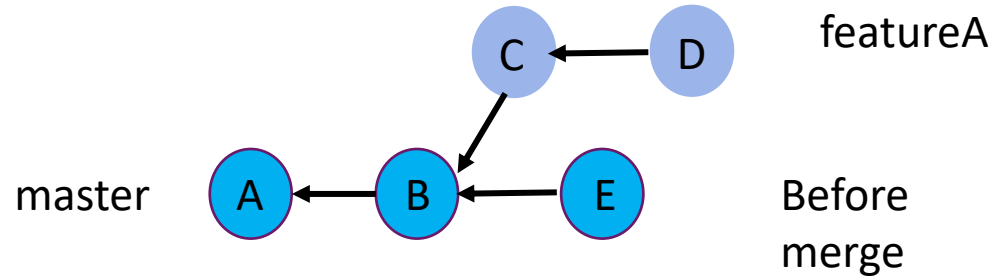
- ***git checkout master***
- ***git merge featureA***



- Default merge type
- Linear history

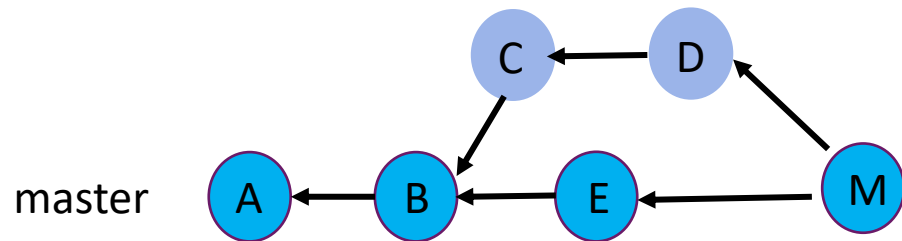
MERGING

Is fast-forward possible for the following situation?



Merge Commit:

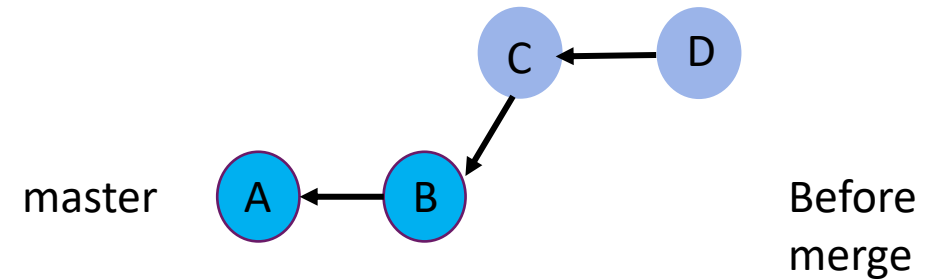
Merge commit combines the commit of the topic branch and base branch to a single merge commit.



Nonlinear graph – easy to see commit history

- *git checkout master*
- *git merge featureA*

Even though a branch is fast-forwardable, you can perform merge commit with `--no-ff` option

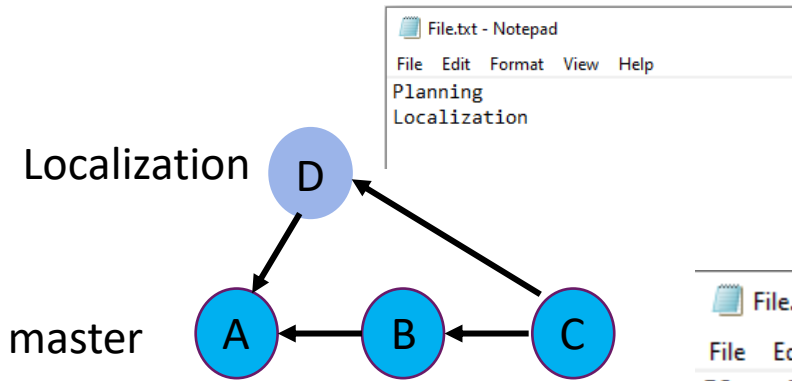


- *git checkout master*
- *git merge --no-ff featureA*

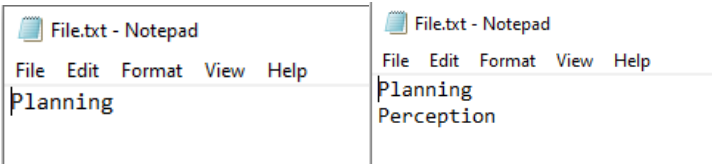
Best practice

Choose a merge policy as a team!
After merging, will you remove a branch to have a clear history? Or will you always keep topic branches to see the history?

RESOLVING MERGE CONFLICTS



```
20211855@TUE025832 MINGW64 /c/Users/20211855/repos/projectd (master)
$ git merge Localization
Auto-merging File.txt
CONFLICT (content): Merge conflict in File.txt
Automatic merge failed; fix conflicts and then commit the result.
```

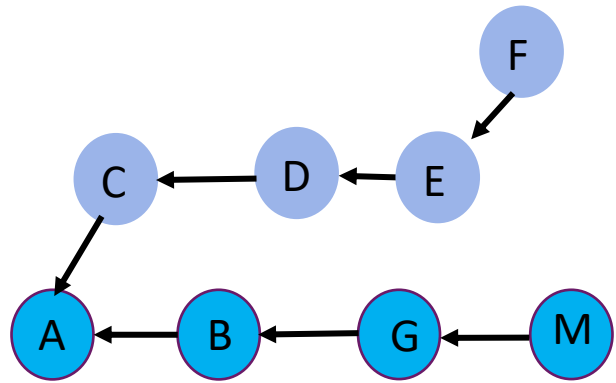


```
File.txt - Notepad
File Edit Format View Help
Planning
<<<<<<< HEAD
Perception
=====
Localization
>>>>>> Localization
```

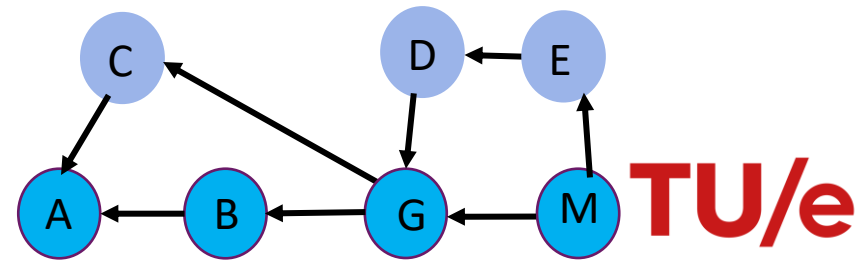
Human decides

- Merge conflicts occur when we change the same part of a file in different branches.

Best practice
Prefer small and frequent merges!

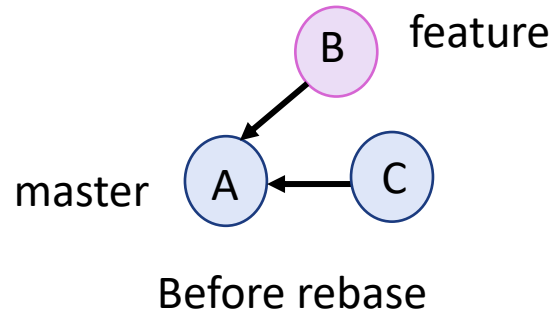


Yes!



REWRITING COMMIT HISTORY

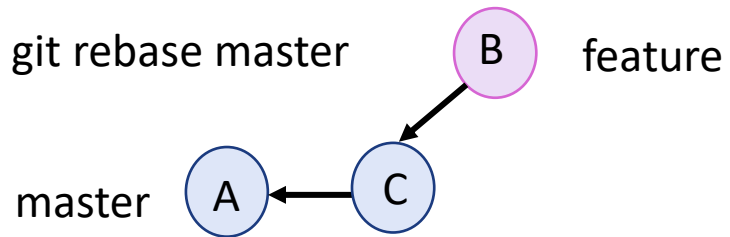
REBASE



```
20211855@TUE025832 MINGW64 ~/repos/projectrebase (master)
$ git log --oneline --graph --all
* 993b9b5 (HEAD -> master) add otherfile
| * a5f4fdf (feature) add rebase
|/
* 81e1219 add file.txt
```

git checkout feature

git rebase master



```
20211855@TUE025832 MINGW64 ~/repos/projectrebase (master)
$ git log --oneline --graph --all
* 7715126 (feature) add rebase
* 993b9b5 (HEAD -> master) add otherfile
* 81e1219 add file.txt
```

Best practice

If you share your branch with others, do not rewrite history!

- + After rebasing you have a clean history
- But you change the history
- It can cause merge conflicts

If there is a merge conflict:

git rebase master

Fix the conflict

git rebase --continue

```
20211855@TUE025832 MINGW64 ~/repos/projectrebase (master)
$ git rebase feature
Auto-merging newfile.txt
CONFLICT (content): Merge conflict in newfile.txt
error: could not apply c8c0f86... Add a new feature
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git reb
abort!".
Could not apply c8c0f86... Add a new feature
```

REWRITING COMMIT HISTORY

AMENDING A COMMIT

Change the commit message

Change the files using the same commit message

git commit - -amend -m "new commit message"

```
20211855@TUE025832 MINGW64 ~/repos/projectrebase (master)
$ git log --oneline -1
83fb5ee (HEAD -> master) Add a new feature
```

```
20211855@TUE025832 MINGW64 ~/repos/projectrebase (master)
$ git commit --amend -m "add a new feature"
[master 7803e89] add a new feature
```

```
20211855@TUE025832 MINGW64 ~/repos/projectrebase (master)
$ git log --oneline -1
7803e89 (HEAD -> master) add a new feature
```

```
20211855@TUE025832 MINGW64 ~/repos/projectrebase (master)
$ git log --oneline --graph --all
* 7803e89 (HEAD -> master) add a new feature
* 7715126 (feature) add rebase
* 993b9b5 add otherfile
* 81e1219 add file.txt
```

```
newfile.txt - Notepad
File Edit Format View Help
Rebase
Line1
Add new features do not change commit message|
```

git commit - -amend -no-edit

```
20211855@TUE025832 MINGW64 ~/repos/projectrebase (master)
$ git log --oneline --graph --all
* 6f4c4e8 (HEAD -> master) add a new feature
* 7715126 (feature) add rebase
* 993b9b5 add otherfile
* 81e1219 add file.txt
```


REWRITING COMMIT HISTORY

INTERACTIVE REBASE

- Edit any commit in any branch
- History will change

```
git rebase -i <after this commit>
```



```
20211855@TUE025832 MINGW64 ~/repos/projecte (master)
$ git log --oneline --graph
* fa56790 (HEAD -> master) add the second line
* 9e329f5 add first line
```

```
git rebase -i 9e32
```

Delete Commit

```
drop pick| fa56790 add the second line
# Rebase 9e329f5..fa56790 onto 9e329f5 (1 command)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#   commit's log message, unless -C is used, in which case
#   keep only this commit's message; -c is same as -C but
#   opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#   create a merge commit using the original merge commit's
#   message (or the oneline, if no original merge commit was
#   specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#   to this position in the new commits. The <ref> is
#   updated at the end of the rebase
.git/rebase-merge/git-rebase-todo[+] [unix] (16:02 30/05/2023) 1,5 Top
-- INSERT --
```

```
20211855@TUE025832 MINGW64 ~/repos/projecte (master)
$ git rebase -i 9e32
Successfully rebased and updated refs/heads/master.
```

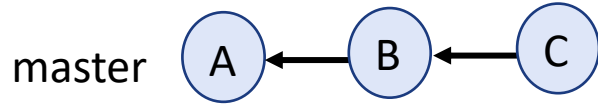
```
20211855@TUE025832 MINGW64 ~/repos/projecte (master)
$ git log --oneline --graph
* 9e329f5 (HEAD -> master) add first line
```



REWRITING COMMIT HISTORY

INTERACTIVE REBASE

Squash a commit

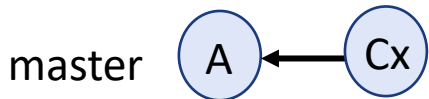


```
file.txt - Notepad
File Edit Format View Help
[INTERACTIVE REBASE
Squash a commit
- Combines commit messages and removes the newer commit
```

```
file.txt - Notepad
File Edit Format View Help
[INTERACTIVE REBASE
Squash a commit
|
```

```
20211855@TUE025832 MINGW64 ~/repos/newproject (master)
$ git log --oneline --graph
* 1b53402 (HEAD -> master) Explain squash a commit
* c56cb3c Add a subtitle
* 1cbb147 Add a title
```

Combine the commit messages and remove the newer commit



```
file.txt - Notepad
File Edit Format View Help
[INTERACTIVE REBASE
Squash a commit
- Combines commit messages and removes the newer commit
```

```
git rebase -i 1cbb
```

```
MINGW64:/c/Users/20211855/repos/newproject
pick c56cb3c Add a subtitle
pick 1b53402 Explain squash a commit
# Rebase 1cbb147..1b53402 onto 1cbb147 (2 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#   commit's log message, unless -C is used, in which case
#   keep only this commit's message; -c is same as -C but
#   opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
```

```
MINGW64:/c/Users/20211855/repos/newproject
pick c56cb3c Add a subtitle
squash| 1b53402 Explain squash a commit
# Rebase 1cbb147..1b53402 onto 1cbb147 (2 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#   commit's log message, unless -C is used, in which case
#   keep only this commit's message; -c is same as -C but
#   opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
```

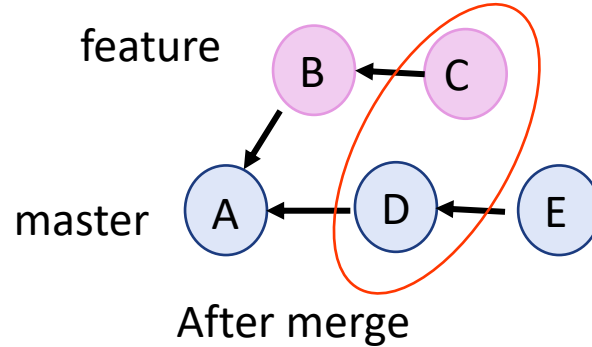
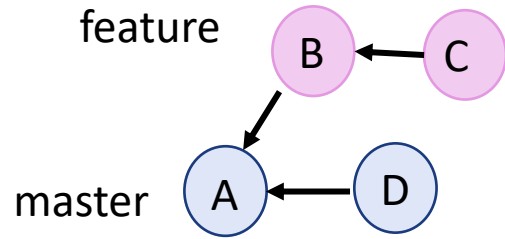
```
20211855@TUE025832 MINGW64 ~/repos/newproject (master)
$ git log --oneline --graph
* bf16650 (HEAD -> master) Squash a commit
* 1cbb147 Add a title
```

We can edit commit message

REWRITING COMMIT HISTORY

SQUASH MERGE

Before merge



```
file.txt - Notepad
File Edit Format View Help
CLEAN HISTORY

file.txt - Notepad
File Edit Format View Help
CLEAN HISTORY
Squash merge |

file.txt - Notepad
File Edit Format View Help
CLEAN HISTORY
Squash merge
- Clean history
- Rewrites history
```

A

B

C

D

```
file.txt - Notepad
File Edit Format View Help
CLEAN HISTORY
Commands:
git merge --squash feature
```

Clean history

Changes history

```
git merge --squash feature
```

```
file.txt - Notepad
File Edit Format View Help
CLEAN HISTORY
<<<<<<< HEAD
Commands:
git merge --squash feature
=====
Squash merge
- Clean history
- Rewrites history
>>>>>> feature
```

Fix
conflict

```
git commit
```

A linear
history!

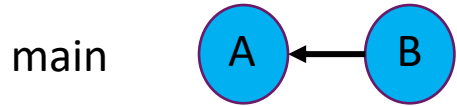
```
20211855@TUE025832 MINGW64 ~/repos/projectmerge (master)
$ git log --oneline --graph
* 66a9752 (HEAD -> master) Explain squash merge
* 72d17f3 Add commands
* dd6b040 Initial commit
```

FETCH AND PULL BEFORE PUSH

Network commands: Clone, Fetch, Pull and Push

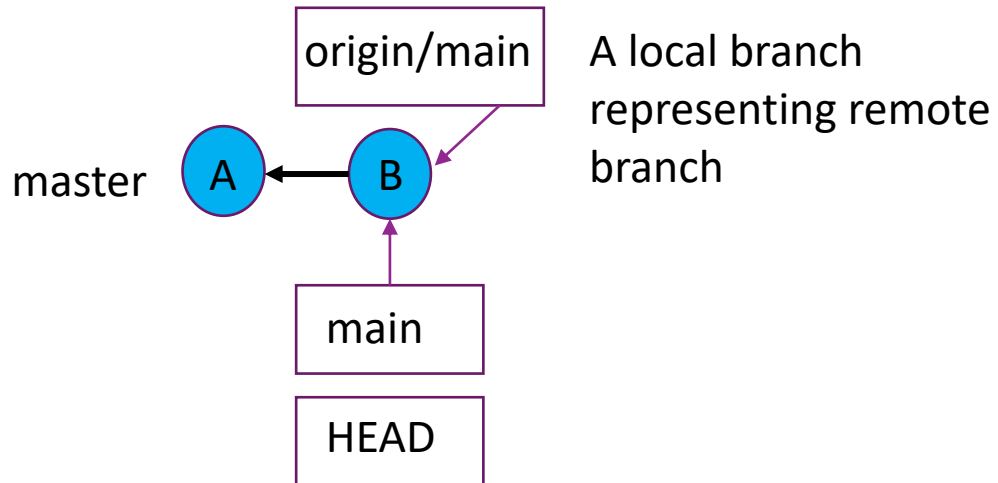
Fetch

View changes on the remote repository without merging
Updates tracking branches



git clone <repository>

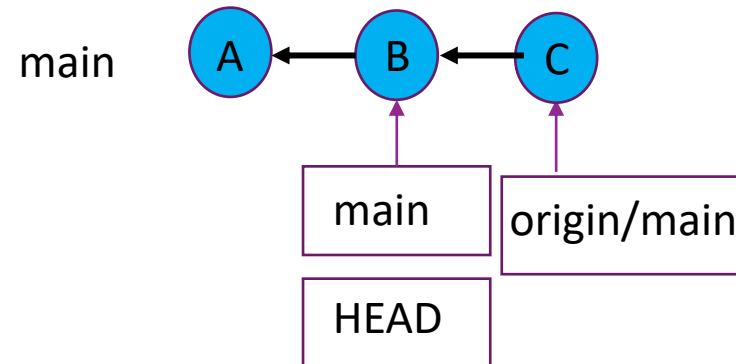
```
20211855@TUE025832 MINGW64 ~/repos/projectd (main)
$ git log --oneline --graph
* 4f9ef72 (HEAD -> main, origin/main, origin/HEAD) Add network command
* 8b17027 Remove lines
* 965e60d Initial commit
```



Let's assume that someone added a commit to the remote repository!

git fetch <repository>

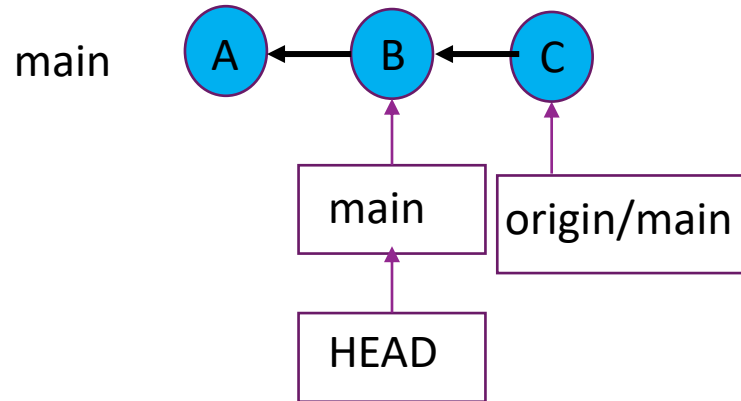
```
20211855@TUE025832 MINGW64 ~/repos/projectd (main)
$ git log --oneline --graph --all
* f52933d (origin/main, origin/HEAD) Try fetch command
* 4f9ef72 (HEAD -> main) Add network command
* 8b17027 Remove lines
* 965e60d Initial commit
```



The tracking branch will be updated after git fetch

FETCH AND PULL BEFORE PUSH

Network commands: Clone, Fetch, **Pull** and Push



```
20211855@TUE025832 MINGW64 ~/repos/projectd (main)
$ git status
On branch main
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.
(use "git pull" to update your local branch)

nothing to commit, working tree clean
```

Pull: Fetch and merge together! After fetch, we use pull command to update our local branches and working area.

```
20211855@TUE025832 MINGW64 ~/repos/projectd (main)
$ git pull
Updating 4f9ef72..f52933d
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
```

```
README.md - Notepad
File Edit Format View Help
# Projectd
"Learning network commands"
"Understand fetch"
```

Push: Add commits to a remote repository

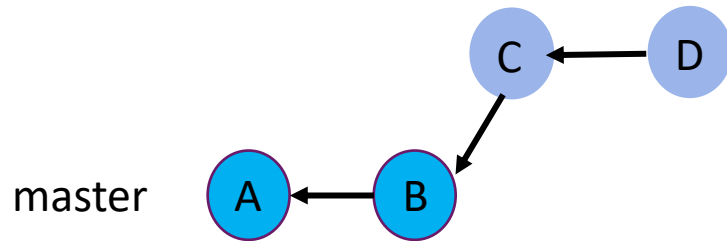
```
20211855@TUE025832 MINGW64 ~/repos/projectd (main)
$ git push -u origin main
Everything up-to-date
branch 'main' set up to track 'origin/main'.
```

Best practice

Fetch and Pull frequently!
Fetch and Pull before Push.

PEER CODE REVIEW

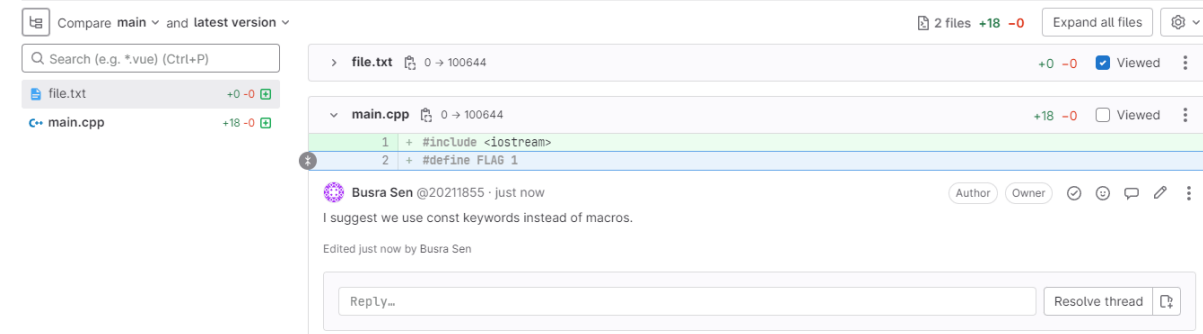
- Git hosting websites such as GitLab provides a pull request option.
- You can review the codes, or you can send a review invitation before merging the branch



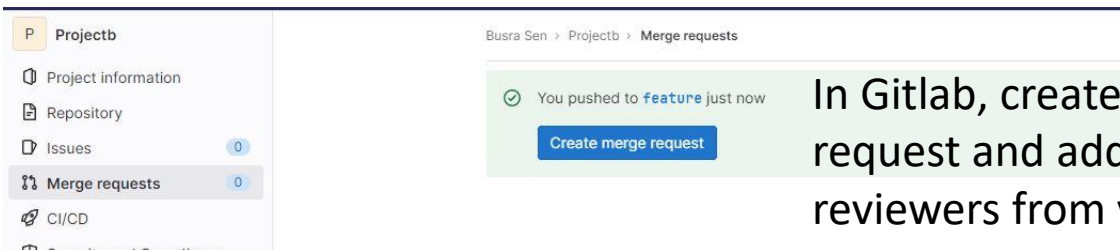
Your branch is ready to merge – ask for code review,

You need help because of bugs,

You want to discuss something about this branch



You should be respectful while you are giving feedback
Also, try to complete the review in a short time
Why the code needs to be changed, explain.

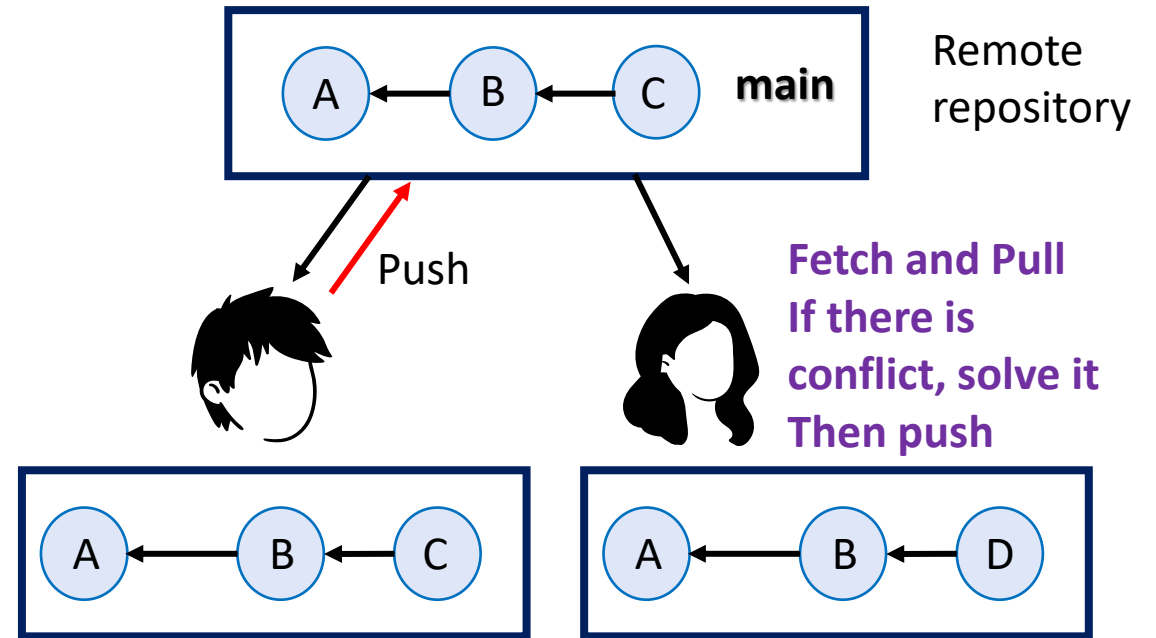
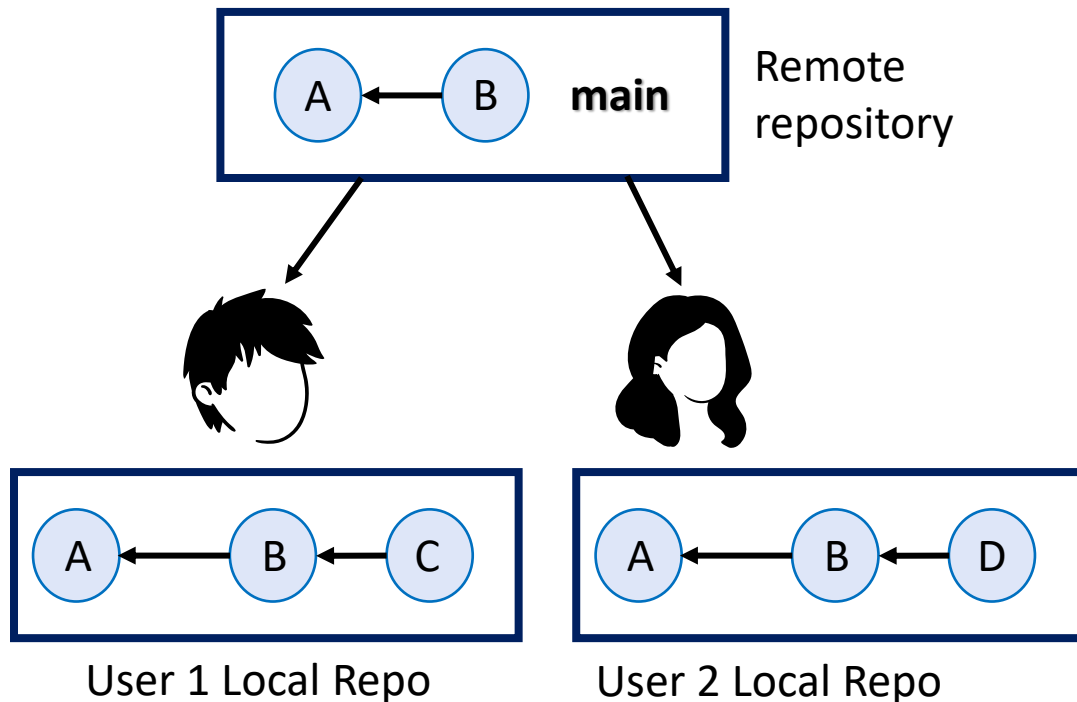


CHOOSE A WORKFLOW

- A workflow is the way that your team achieved the task.
- There are different types of workflows such as centralized, feature branch, and git flow workflows, etc.

1- CENTRALIZED WORKFLOW

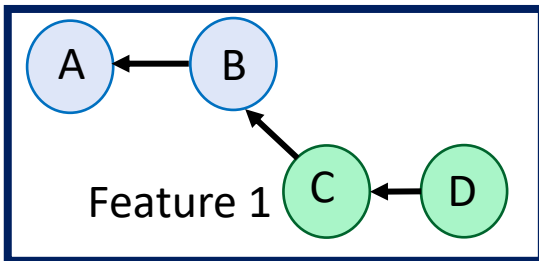
- Use only one branch (main)
- Work independently on your local repo



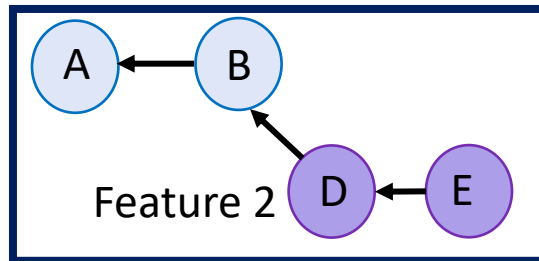
- Simple
- You can work independently but you do not take advantage of branching
- It is suitable for small size teams

CHOOSE A WORKFLOW

2- FEATURE BRANCH WORKFLOW



User 1 Local Repo



User 2 Local Repo

- After one of the users completes the feature, the other user performs a code review then the user can merge the feature to the main branch!

3- GITFLOW WORKFLOW

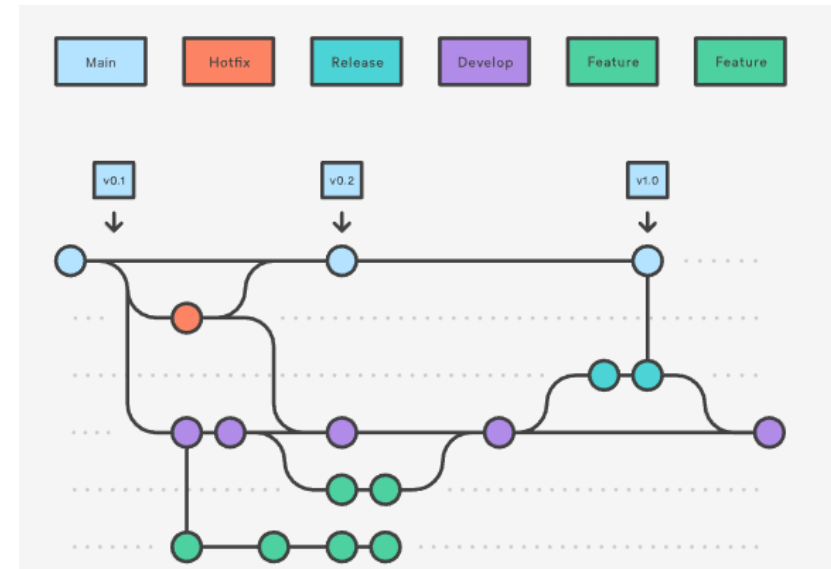


Figure from <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

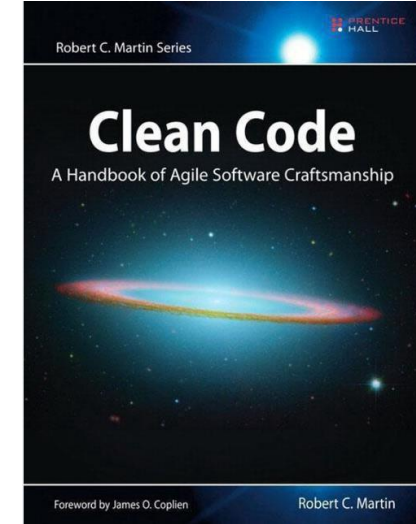
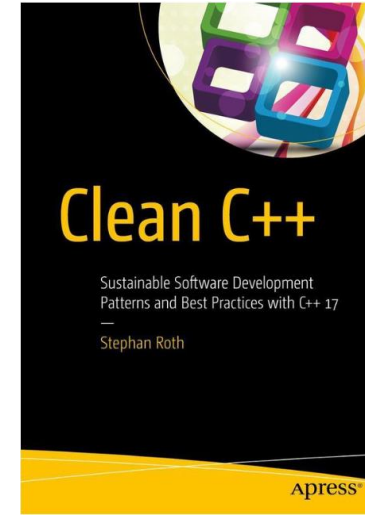
Best practice

You should decide which workflow you will use as a team! You do not have to use existing flows. You can mix them or you can create your own workflow. The important thing is to make sure all team members agree with the workflow!

CONCLUSION

- As a team you should think about some points to have a clean code
 - Consistent code style
 - Meaningful names
 - Functions that do not have more than one thing
 - Good comments explain the reason behind decisions
 - Avoid magic numbers
 - Avoid unsigned signed numbers
 - Initialize your local variables
 - Define a variable when you need them
 - Avoid non-const global variables
 - Use explicit namespace representation

- Choose a workflow when you start a project
- Merge frequently
- Commit frequently and write a good commit message
- Do not change history when you are sharing a branch with others
- Use code review to learn and improve your coding skills



THANK YOU FOR YOUR ATTENTION!

REFERENCES AND FURTHER READING

- [1] Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.
- [2] Roth, S. (2017). *Clean C++: Sustainable Software Development Patterns and Best Practices with C++ 17*. Apress.
- [3] <https://lefticus.gitbooks.io/cpp-best-practices/content/03-Style.html>
- [4] [//github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#es20-always-initialize-an-object](https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#es20-always-initialize-an-object)
- [5] McQuaid, M. (2014). *Git in practice*. Simon and Schuster
- [6] <https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud>
- [7] <https://www.learncpp.com/>