Introductory Labs

- LAB: Periodic Scheduler
- LAB: Git
- Git Basics

LAB: Periodic Scheduler

The objective of this assignment is:

- Set up your development environment
- Learn how to run unit-tests
- Trial how to input your code to the Periodic Scheduler

For CmpE243, we will not be focusing on typical RTOS tasks like CmpE244. The reason is that we wish to use an approach that is typically seen in the Automotive industry, which is to design the logic of your autonomous RC car based on software instructions that occur periodically and consistently.

Part 0: Build Environment

Set up your development environment for this portion of the lab. Follow through and read all of the README files carefully that are linked here. Make sure you are able to run the unit tests, and also compile a hex file that you can load onto your board.

You can watch the following video to get started:

- Youtube: Compile project
- Youtube: Scons build system

Part 1: Blink LEDs

For this portion, edit the code such that it will start to blink four LEDs driven by the periodic scheduler. In particular, read the documentation of the `main.c` file, and enable the code for the periodic scheduler.

Study the overall structure of `main.c` , and then switch a `#if (1)` to `#if (0)` such that it will disable two blinky tasks, and instead run the periodic scheduler. The name "periodic scheduler" may sound fancier than what it actually is, but this is just a trivial piece of code that invokes function at `periodic_callbacks.c` file.

```c
// main.c
static void create_blinky_tasks(void) {
  /**
   * Use '#if (1)' if you wish to observe how two tasks can blink LEDs
   * Use '#if (0)' if you wish to use the 'periodic_scheduler.h' that will spawn 4 periodic tasks, one
   */
#if (0)
  // ...
#else
  periodic_scheduler__initialize();
  UNUSED(blink_task);
#endif
}
// periodic_scheduler.c
void periodic_scheduler__initialize(void) {
  /**
   ...
   */
  static StackType_t hz1_stack[4096 / sizeof(StackType_t)];
  static StackType_t hz10_stack[4096 / sizeof(StackType_t)];
  static StackType_t hz100_stack[4096 / sizeof(StackType_t)];
  /**
   ...
   */}
```

There are a few things to note for future reference:

- The stack size is chosen with a same value, and depending on the complexity of the functions you invoke at the `periodic_callbacks.c` file, you may have to increase this memory size. Also note that there are five tasks total that run the periodic callbacks, so if you input 2K, then you will end up using

10K for the memory footprint. Recommended size is 2-4K.

- The logic at `periodic_callbacks.c` the file should be function calls into your other code modules. This way, unit tests of this file will remain simple. You do not want to input branch statements here because this would make your code less modular, and difficult to unit-test.

Part 2: Switch and LED code module

Insert additional code to one of the periodic callbacks, and then observe its operation. In the example below, we are going to demonstrate the right way to build a module that reads a switch and lights up an LED.

DO NOT do the following because what you have done is that cluttered all the things that need to occur periodically. If we go down this path, you will end up creating a giant `periodic_callbacks.c` file that will be difficult to test, and your code will not be modular or broken down into these pieces. Unit-testing code will also be difficult because now you have to not only test the switch and LED logic but also test more unrelated subsequent code.

```c
// periodic_callbacks.c -- BAD example
static gpio_s my_led;
static gpio_s my_switch;
void periodic_callbacks__initialize(void) {
  my_led = gpio__construct_as_output(GPIO__PORT_2, 0);
  my_switch = gpio__construct_as_input(GPIO__PORT_2, 1);
}
void periodic_callbacks__1Hz(uint32_t callback_count) {
  gpio__toggle(board_io__get_led0());
  if (gpio__get(my_switch)) {
    gpio__set(my_led);
  } else {
    gpio__reset(my_led);
  }}
```

Instead, follow good code design, and create "modules" for your code. Using this approach, you have refactored your switch and LED logic to a new code module: `switch_led_logic.h`. You can test this code module separately and then testing the `periodic_callbacks.c` a code module is also

straightforward since you only have to set up a couple of "expect" function calls.

```c
// periodic_callbacks.c -- Good example
#include "switch_led_logic.h"
void periodic_callbacks__initialize(void) {
  switch_led_logic__initialize();
}
void periodic_callbacks__1Hz(uint32_t callback_count) {
  gpio__toggle(board_io__get_led0());
  switch_led_logic__run_once();}
```

Of course, you are not done yet, and you also have to modify `test_periodic_callbacks.c`

```c
#include "Mockboard_io.h"
#include "Mockgpio.h"

// Add mock of your new code module
#include "Mockswitch_led_logic.h"
#include "periodic_callbacks.h"
// Add expect during the periodic_callbacks__initialize() function
void test__periodic_callbacks__initialize(void) {
  switch_led_logic__initialize_Expect();
  periodic_callbacks__initialize();
}
void test__periodic_callbacks__1Hz(void) {
  gpio_s gpio = {};
  board_io__get_led0_ExpectAndReturn(gpio);
  gpio__toggle_Expect(gpio);
  switch_led_logic__run_once_Expect();
  periodic_callbacks__1Hz(0);}
```

Part 3: Experiment with Task Overrun

Deliberately overrun one of the periodic tasks and observe that your board will reboot. Since this will be sort of a "throw-away" code, you can opt to skip the unit-tests. Here is a sample code that will deliberately reboot the processor because of the missed deadline of the 1Hz function.

```c
// periodic_callbacks.c
// Include these files for RTOS task delay function
#include "FreeRTOS.h"
#include "task.h"
void periodic_callbacks__1Hz(uint32_t callback_count) {
  gpio__toggle(board_io__get_led0());
  // On the fifth function call to this function, sleep for 1000ms
  if (callback_count >= 5) {
    vTaskDelay(1000);
  }}
```

It is **strongly advised** NOT to skip the unit-tests in general. But if you are purely doing a code prototype to try things out, then use the `scons --no-unit-test` command.

What did you learn?

- Work with the periodic callbacks to add your code
- Design small code modules, and set up their expectation in unit-test code
- The first-hand account of what happens when you miss the deadline of a periodic callback

LAB: Git

This is definitely not an exhaustive tutorial about learning Git... Google would be better to reveal several great tutorials about Git. What we focus on instead is a simplistic workflow about publishing a "Pull Request" in Git.

What is Gitlab?

Gitlab provides services that allow hosting your project on a remote repository and provides additional features that help in continuous integration and deployment. Such as code sharing, code review, and bug tracking.

Part 0: Setup Gitlab account

For better or worse, we have decided to use Gitlab.com for the repository. You are also required to use this Gitlab repository because that keeps the entire class aligned to a single server type and reduces fragmentation while increasing the efficiency of the teacher and the ISA team.

For this part, establish your Gitlab.com account.

How to set up a Gitlab account?

1. Go to GitLab.com and create an account.
2. Sign in to GitLab.

In addition, also install Git to your machine such that you can successfully execute the `git` Commands from a terminal.

1. Download git from GIT_Install and install git.
2. Check git is installed on your system by using the "git --version" command in the terminal.

Part 1: Fork SJ2-C Project

When you fork a project, you essentially create a copy of the original SJ2-C repository. This will be your

version of the forked project, and you can use this throughout the semester for your private workspace to do the lab assignments.

Browse to the SJtwo-c repository, and click on the Fork button.

After you fork the repository, make sure you set the permissions to "public". Do this by going into your newly forked repository settings, and look for the "Visibility" setting.

How to change project visibility

1. Go to your newly forked project's **Settings**
2. Change **Visibility Level** to Public

Part 2: Basic Git Commands

INIT

The first thing you want to do before you init is to add a project on the Git website to see the "Setting up a new Git repository" section. If you have a folder with code that is not on Git, and you wish to put it on the Git server, then you need to initialize Git into your folder. This creates a .git folder, and the current directory is now a Git repository. The .git folder contains Git information such as branches. Initializing your folder is local to your computer and does not yet upload onto the server.

```
# To add your project to the git
# Initialize current directory
$ git init
# Initialize selected directory$ git init <directory>
```

CLONE

If you see a repository that you want to work on, you can "clone" it into a directory and start working on

it. Cloning it will download the entire repository as well as a .git folder. Note that the clone is different from "pull". This will be explained later. Just use this command once at the beginning of the project unless you want multiple folders.

```
# Downloads entire repository to current directory
$ git clone <repo>
# Downloads entire repository to selected directory$ git clone <repo> <directory>
```

> The difference between forking and cloning a GIT project means when you **fork** a repository, you create a copy of the original repository (upstream repository) but the repository remains on your personal Gitlab account. Whereas, when you **clone** a repository, the repository is copied onto your local machine with the help of Git.

Part 3: Branch Workflow

The process of checking-in new code to your forked repository will involve "Branch Workflow". There are actually a number of ways to contribute code to your repository, and the branch workflow is just one of them that we will choose to use.

We are not going to discuss that in detail because it is already captured well in **this awesome article**. We will summarize the process that you will use to do this. The `$` indicates the commands you should try.

```
# See what is going on
$ git status
On branch master
# Create a new "branch" of code to work on
# You can use any name, and feature/foo is just a convention
$ git checkout -b feature/gpio_blinky_in_periodics
Switched to a new branch 'feature/gpio_blinky_in_periodics'
# Add or modify a file we want
$ touch file.txt
# Tell git to add it to be committed
```

```
$ git add file.txt
# Check what is going on
$ git status
On branch feature/gpio_blinky_in_periodics
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new file: file.txt
# Commit the change with a message
$ git commit -m "added file.txt"
[feature/gpio_blinky_in_periodics 5f76839] added file.txt
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 file.txt
# Check what is going on
$ git status
On branch feature/gpio_blinky_in_periodicsnothing to commit, working tree clean
```

Part 4: Merge Request (MR)

The typical name of a request to merge code is called a "Pull Request" or a "Merge Request". This is the chance to review the code and merge the code. In the end, `Part 3` you have a branch that only exists on your computer. In case you lose your computer or your storage device dies, then you **will lose** any work even though you have "committed" a change.

The distinction is that a commit only commits to your storage device, but does not send the data or the branch to the Git server. To actually push the code to the Git server, simply type `git push origin head`.

```
$ git push origin head
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 262 bytes | 262.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
```

```
remote:
remote: To create a merge request for feature/gpio_blinky_in_periodics, visit:
remote:    https://gitlab.com/sjtwo-c-dev/sjtwo-c/-/merge_requests/new?merge_request%5Bsource_branch%5D=
remote:
To gitlab.com:sjtwo-c-dev/sjtwo-c.git
 * [new branch]      head -> feature/gpio_blinky_in_periodics
```

This command will generate a URL for you, so copy and paste this URL to your web browser. For example, the URL above is:

```
https://gitlab.com/sjtwo-c-dev/sjtwo-c/-/merge_requests/new?merge_request%5Bsource_branch%5D=feature%2Fg
```

This will lead you to generate your "Merge Request". At the end of the web page that loads, click on "Submit Merge Request". At this point, you can view the changes, get feedback from others, and if the code looks good, you can then merge the code. But wait ... rarely will you be able to merge code without iterating and revising it, and that is what Part 5 is for.

Part 5: Revise an MR

Granted that you have an MR already out there, and you have got feedback from others, this section will teach you how to revise or amend your code.

```
# Modify any code
# In this case, we will dump 'hello' to our previously committed file: file.txt
$ echo "hello" >> file.txt
# Check what is going on
$ git status
On branch feature/gpio_blinky_in_periodics
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
 modified:   file.txt
# Add the file we want to re-commit (another commit on top of previous)
$ git add file.txt
$ git commit -m "Added hello to file.txt"
# Update the remote branch and the Merge Request$ git push origin head
```

After the `git push` command, your MR will be updated on the browser. This way, you can continue to revise your MR per the suggestions of other people. When you are satisfied with your MR, you can seek approval and officially hit the Merge button on the Gitlab.com webpage.

Part 6: Final Step

After you have merged your MR, it is time to go back to the master branch and grab the latest changes. Other users may have merged their code also, so pulling the latest master branch is going to get you the latest and greatest code.

```
# Go to the master branch
$ git checkout master
# Pull the latest master$ git pull origin master
```

Part 7: Going beyond . . .

There is of course A LOT more to Git, but once you master the basics, you can then Google your way through the rest of the world you will face such as:

- Handling merge conflicts
- Check out other people's branches

Rebase on the latest master branch.

```
$ git status
# Assume that you are on your feature branch
$ git checkout master
$ git pull origin master
# Go back to the previous branch you were working with (feature)
$ git checkout -
# Apply our commits to the latest master$ git rebase master
```

Part 8: Steps to create MR for Lab Submissions

The process of checking-in new code to your forked repository will involve "Branch Workflow" as explained in **PART 3**. The following steps will help you to add new code/files for each of your lab submissions.

- Goto cmd OR terminal OR git bash. CD to the location of the cloned project(cd sjtwo-c/projects/lpc40xx_freertos/l5_application) and run the following commands.

```
 # You can use any name, it's better to use lab with the number as a branch name.
 # such as lab1,lab2
$ git checkout -b lab1
Switched to a new branch 'lab1'
# Add or modify files as per the given lab assignment
# for example lab 1 requires two files
$ touch lab_multitasks.c
$ touch lab_multitasks.h
# Check what is going on
$ git status
On branch lab1
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
lab_multitask.c
lab_multitask.h

# Tell git to add it to be committed
$ git add .
# Check what is going on
$ git status
On branch lab1
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
new file:   lab_multitask.c
new file:   lab_multitask.h
# Commit the change with a message
$ git commit -m "added lab1 files"
[lab1 e88f23d] added lab1 files
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 projects/lpc40xx_freertos/l5_application/lab_multitask.c
 create mode 100644 projects/lpc40xx_freertos/l5_application/lab_multitask.h
# Check what is going on
$ git status
On branch lab1
nothing to commit, working tree clean
# Update the remote branch and the Merge Request
$ git push origin head
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 262 bytes | 262.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
remote:
remote: To create a merge request for lab1, visit:
remote:    https://gitlab.com/sjtwo-c-dev/sjtwo-c/-/merge_requests/new?merge_request%5Bsource_branch%5D=
remote:
To gitlab.com:sjtwo-c-dev/sjtwo-c.git
 * [new branch]      head -> lab1
```

```
# Assume that you are on your lab branch
# To comeback to master branch$ git checkout master
```

- Create a merge request for each lab and use the merge request URL for your lab submissions.
    - Please follow **PART 4** to generate and submit "Merge Request" on Git.
    - After submitting a merge request you will receive a new URL on the browser. Use that URL for your canvas submission.
- Follow the same steps for creating the next lab branch(such as lab2), add new files to the lab2 branch(such as lab_gpio.c and lab_gpio.h), and create a merge request for the submission after completing your GPIO driver.

for appropriate to; intended for More (Definitions, Synonyms, Translation)

Note make mention of More (Definitions, Synonyms, Translation)

Git Basics

What is Gitlab?

Gitlab provides services that allow hosting your project on a remote repository and provides additional features that help in continuous integration and deployment. Such as code sharing, code review, and bug tracking.

GIT Workflow

In Git there is the notion of a "Master" code base which contains the work of all contributing members in a project.

There are two basic workflows that you may follow when using Git for version control.

1. Committing directly to the "Master" branch.
2. Creating branches from the "Master" branch and merging them back in when ready.

This section of the guide will walk you through these two workflow strategies.

1. Working off the "Master" Branch

Working directly off the "Master" branch can be advantageous to smaller groups who rarely (if ever) work on the same portions of the code at a time.
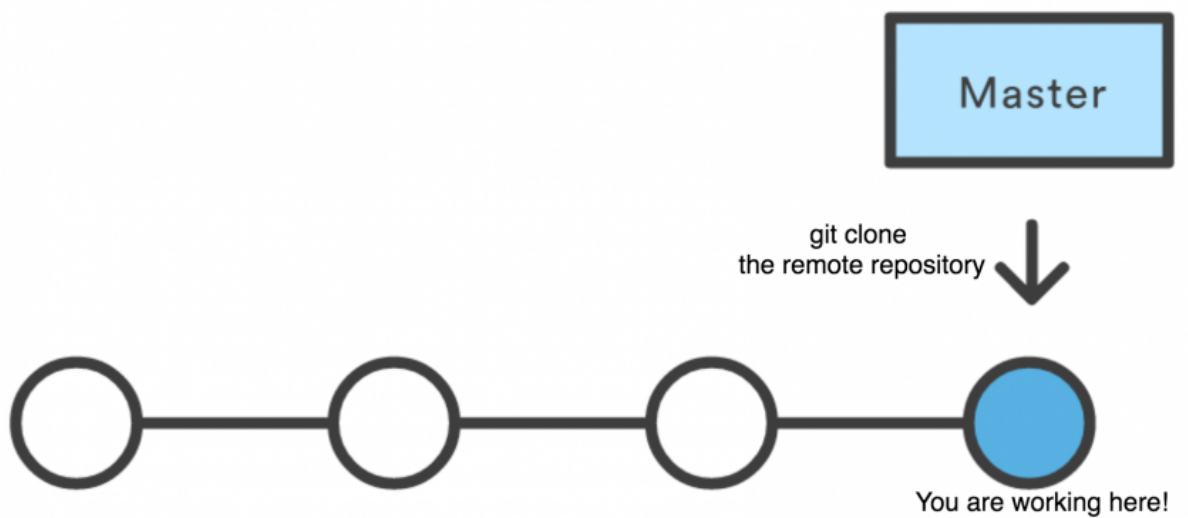
The basic workflow for this method is as follows:

1. "Pull" from the Master branch to ensure the local copy contains the latest version of the code.
2. Make necessary changes to the code in your local repository.
3. Commit your changes.
4. "Push" your changes to the remote repository.
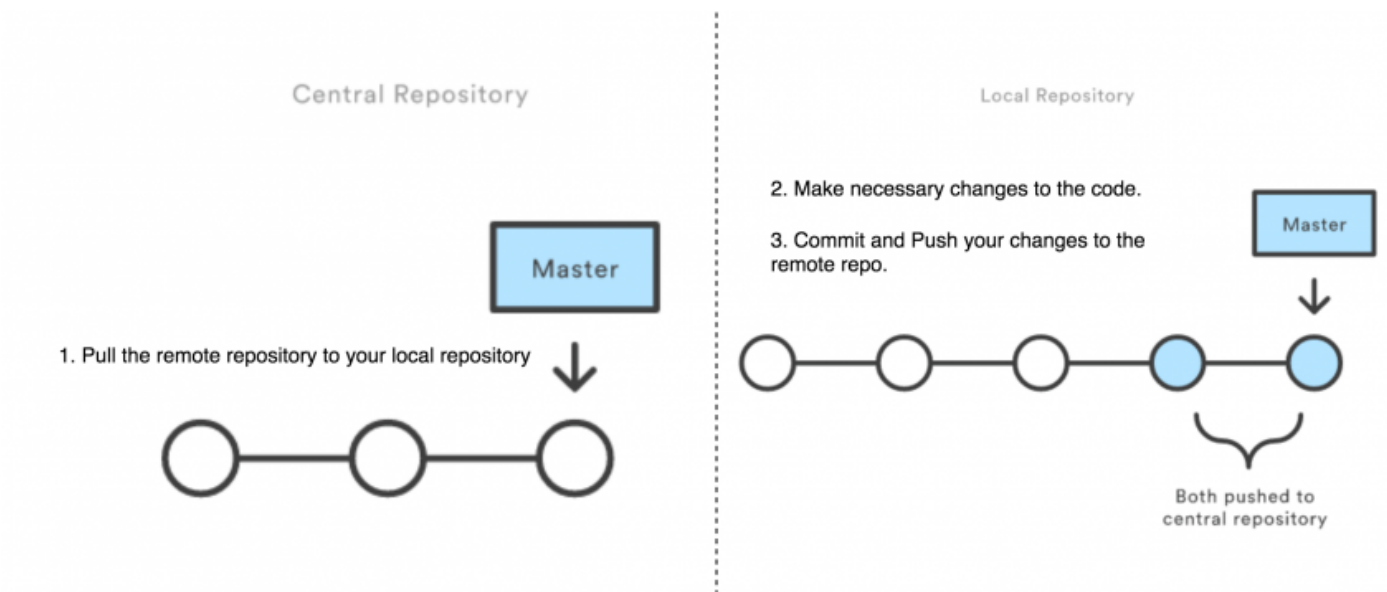
In git commands this would look like this:

```
# Make sure you are on master branch
```

```
git checkout master
# Make sure you have the latest code
git pull origin master
# Make your changes to your code
# Add files you may have changed to your commit
git add <file1> <file2> ...
# Add all untracked files to your commit
git add .
# Commit your changes
git commit -m "<commit message here>"
# Push your changesgit push origin HEAD
```

2. Working with feature branches

The second workflow takes advantage of the branching system in git. To protect your Master branch from code that may break your build or introduce bugs we can create what is called a "feature branch." These branches contain your development code and isolate it from the main code until you are ready to merge them together.

The workflow is as follows:

1. Do a "git fetch" to obtain the latest version of your source branch.
2. Check out a new branch.
3. Perform your work on your new branch (be sure to make regular commits to avoid losing any of your work.)
4. Merge the two branches.
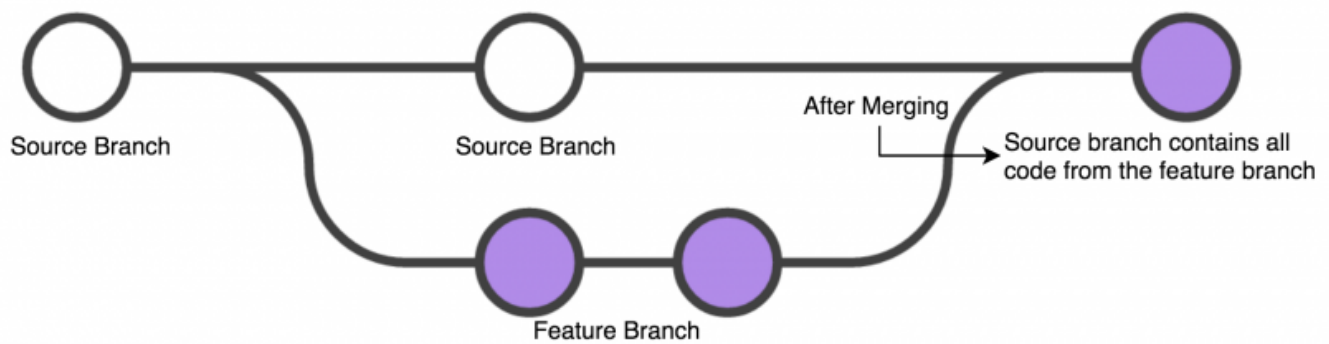
Here is the general workflow in git commands:

```
# Checkout your "source" branch (the branch you want to base your code off of)
git checkout master
# Obtain the latest code
git fetch origin
# Create a new branch from your source branch
git checkout -b <new branch name>
# Make your code changes and commit them regularly
git add <file1> <file2> ...
git commit -m "<commit message>"
# Push your changes to your FEATURE branch
# GIT server knows this branch after the push and other people can also check-out your branch
# But this branch is not yet merged to the master branchgit push origin HEAD
```

When you are ready to merge your branch back into the source branch there are two routes you may take:

1. Merge your feature branch directly into the source branch.
2. Open a pull request for peer code review prior to merging your branch.

To merge your feature branch into the source uses the following workflow:

1. Check out the source branch.
2. Ensure your source branch contains the most updated code from the remote repo.

3. Merge your feature branch into the source branch.
4. Push the newly merged source branch back to the remote repo.

The git commands for this workflow looks like this:

```
# Checkout the source branch that you want to merge your branch into (assuming your source was 'master
git checkout master
# Ensure your source branch is up-to-date
git pull origin master
# Merge your feature branch INTO the source branch
git merge <feature branch>
# At this point, you might need to resolve merge conflicts
# Push your changes to the remote repogit push origin master
```

3. Merge Conflicts

> When working in a team it will be inevitable that the same file will be touched by multiple developers. If multiple make changes in the same part of the file, then it will result in a merge conflict when attempting to merge the files together. These conflicts can be resolved in your IDE directly or in any text editor.

What is Git Merge Conflict?

A merge conflict is an event that takes place when Git is unable to automatically resolve differences in code between two commits. Git can merge the changes automatically only if the commits are on different lines or branches.
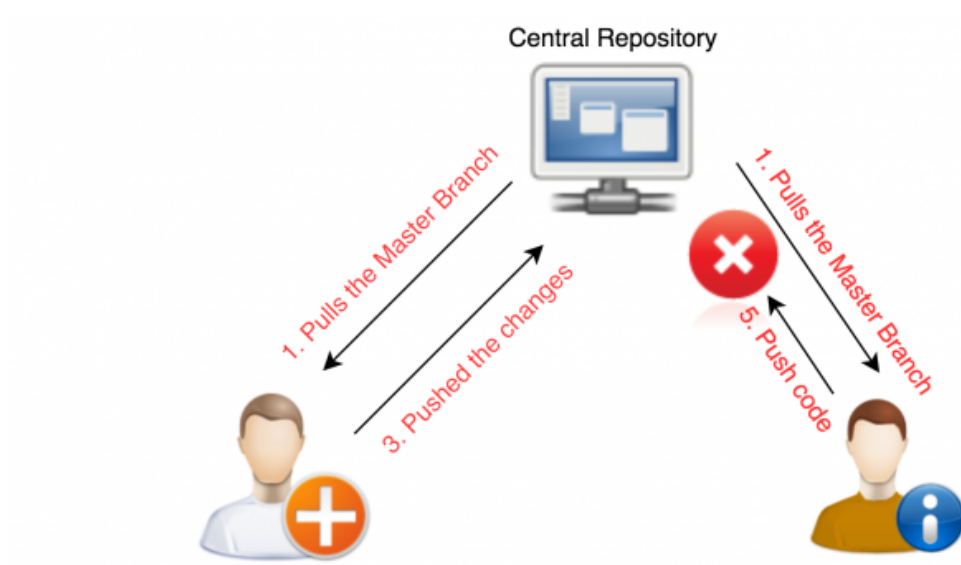
Let's assume there are two developers: Two developers pull the same code file from the remote repository and try to make various amendments to the same file. After making the changes, Developer 1 pushes the file back to the remote repository from his local repository. Now, when Developer 2 tries to push that file after making the changes from his end, he is unable to do so, as the file has already been changed in the remote repository.

To prevent such conflicts, developers work in separate isolated branches. The Git merge command combines separate branches and resolves any conflicting edits.

The git commands for this workflow looks like this:

```
# The status command will provide you with the current status of your branch. It provides information
# such as files changed or whether or not you are up-to-date with the remote branch.
$ git status
On branch merge_branch
Your branch is up to date with 'origin/merge_branch'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   merge_demo.c
no changes added to commit (use "git add" and/or "git commit -a")
# Make your code changes and commit them
$ git add .
$ git commit -m "chnages in the c file"
[merge_branch d2d4473] chnages in the c file
 1 file changed, 1 insertion(+), 1 deletion(-)
# Push your changes to the remote repo
$ git push
To https://gitlab.com/Jain_Vidushi/sjtwo-c.git
 ! [rejected]        merge_branch -> merge_branch (fetch first)
error: failed to push some refs to 'https://gitlab.com/Jain_Vidushi/sjtwo-c.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
# Ran into the MERGE CONFLICT
# Ensure your source branch is up-to-date
```

```
$ git pull
remote: Enumerating objects: 2, done.
remote: Counting objects: 100% (2/2), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (2/2), done.
From https://gitlab.com/Jain_Vidushi/sjtwo-c
   0d0ac2a..139b80d  merge_branch -> origin/merge_branch
Auto-merging merge_demo.c
CONFLICT (content): Merge conflict in merge_demo.c
Automatic merge failed; fix conflicts and then commit the result.
Users-MBP-2:sjtwo-c Macbook$ git status
On branch merge_branch
Your branch and 'origin/merge_branch' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)
Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   merge_demo.c
no changes added to commit (use "git add" and/or "git commit -a")
# At this point, you might need to resolve merge conflicts on your local machine
# Make your code changes as per the conflict and commit them again
$ git add .
$ git commit -m "conflict resolved"
[merge_branch 2f261aa] conflict resolved
# Push your changes to the remote repo
$ git push
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 537 bytes | 537.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote:
```

```
remote: View merge request for merge_branch:
remote:    https://gitlab.com/sjtwo-c-dev/sjtwo-c/-/merge_requests/157
remote:
To https://gitlab.com/Jain_Vidushi/sjtwo-c.git
   139b80d..2f261aa  merge_branch -> merge_branch
```

Synonyms for "fetch" fetch the action of fetching More (Definitions, Synonyms, Translation)