



UNIVERSIDAD TECNOLÓGICA DE QUERÉTARO

Diplomado en Software Embebido

Módulo 1: Bases de Ingeniería de Software

Tema: 1.4.4 “Herramientas para Control de Versiones de Software”

Juan Pablo Cruz Gutiérrez

Versión 3.0

I. Introducción

En el mundo del Software siempre se buscan maneras para mejorar y optimizar el desarrollo de código, existen cosas tan simples que muchas veces pasan desapercibidas, como un control de versiones, utilizado con el fin de gestionar ágilmente los cambios en un código fuente.

Hay muchas opciones para manejar versiones de software, cada persona o cada empresa puede generar sus propias reglas que pueden servir como guía.

Entre las formas más básicas que existen para el control de versiones están las siguientes, manejarlas por número, estabilidad o fecha, pero los principales problemas al usar estos métodos, es que son propensos a errores, pérdida de datos y son difíciles de manejar cuando hay más de una persona que modifica el proyecto.

Actualmente no existe un estándar o una normativa oficial para el control de versiones, pero existen diferentes herramientas para control de versiones que nos facilitan esta gestión.

¿Qué es una herramienta para control de Versiones?

Es un sistema que permite realizar seguimiento de una colección de archivos, además incluye la funcionalidad de revertir la colección de archivos actual a una colección anterior, cada versión podría considerarse como una fotografía en un momento determinado, algunas de estas herramientas son:

- PlasticSCM
- Git
- CSV
- SVN
- Mercurial
- Bazaar
- Monotone
- Synergy
- PVCS

Estos programas fueron ideados para gestionar los cambios de código fuente o poder revertirlos ágilmente, cuyo ámbito ha sido ampliado pasando del concepto control de versiones al de gestión de configuración de software, en el que se engloban todas las actividades que pueden realizarse por un equipo sobre un gran proyecto software u otra actividad que genere ficheros digitales (por ejemplo: documentos, ofertas, dibujos, esquemas, etcétera).

Se recomienda contar con un sistema de control de versiones, sin importar lo complejo del proyecto, ya que las ventajas que este añade son muy importantes, sobre todo cuando se tiene un equipo de trabajo muy grande y se tiene que trabajar en modo colaborativo, estos sistemas nos ayudan para tener trazabilidad en las modificaciones que se han agregado, o en caso de depuración es útil para excluir repositorios de manera ágil y así identificar más fácil que repositorio hace que nuestro sistema se comporte distinto a lo esperado.

II. Tipos de control de versiones

El control de versiones local, es usado por mucha gente y consiste en copiar los archivos a otro directorio (quizás indicando la fecha y hora en que se generó el archivo). Es un método muy simple pero como ya se había mencionado es altamente propenso a errores, pues es fácil olvidar en que directorio te encuentras y guardar accidentalmente en el archivo equivocado o sobrescribir en el archivo diferente al deseado.

Para enfrentar este problema hace tiempo se desarrollaron herramientas que contenían una simple base de datos en la que se llevaba registro de todos los cambios realizados sobre el archivo, ver esto en la Figura 1.4.4. 1 (Método de control de versiones local)

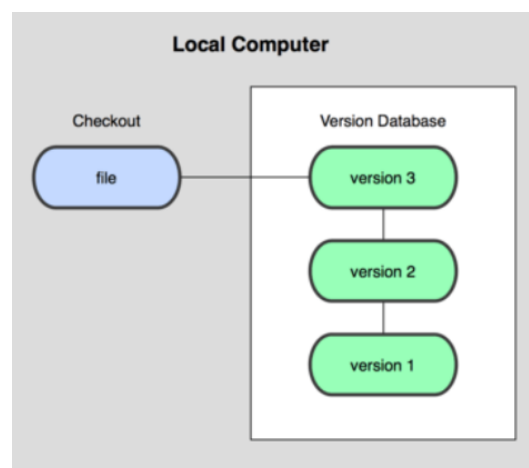


Figura 1.4.4. 1 (Método de control de versiones local)

Control de Versiones Centralizada, tienen la característica que funciona como un entorno clásico cliente-servidor, es decir se tiene un servidor en el que se aloja el repositorio del proyecto, con toda la información de los cambios.

En este entorno el usuario trabajara con una copia del servidor en una revisión determinada, normalmente se trabaja con la más actualizada, el usuario hace cambios sobre esa copia y cuando se considera que ha terminado con esa modificación la sube al servidor, el cual se encarga de unir los cambios al repositorio central e informar de los errores que se pudieran dar ver Figura 1.4.4. 2 (Método de control de versiones Centralizada)

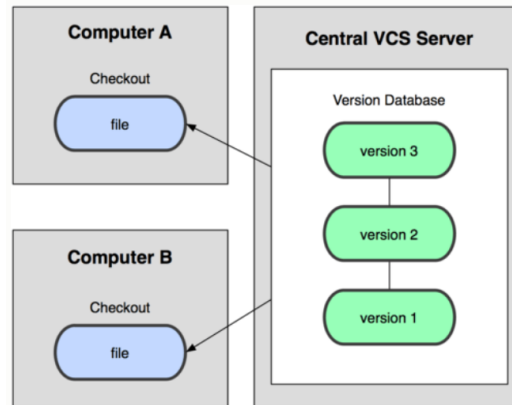


Figura 1.4.4. 2 (Método de control de versiones Centralizada)

El Control de versiones distribuida es un sistema donde hay un servidor central y cada usuario puede hacer una copia completa del directorio central, y por ser una copia posee las mismas funcionalidades que el directorio original, es decir, contiene la historia completa de la colección de archivos con la propiedad que el directorio puede intercambiar entre sus versiones, así los usuarios no sólo descargan la última versión de los archivos, también si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los usuarios puede copiarse en el servidor para restaurarlo. Funcionando, así como una copia de seguridad completa de todos los datos ver Figura 1.4.4. 3 (Método de control de versiones Distribuida)

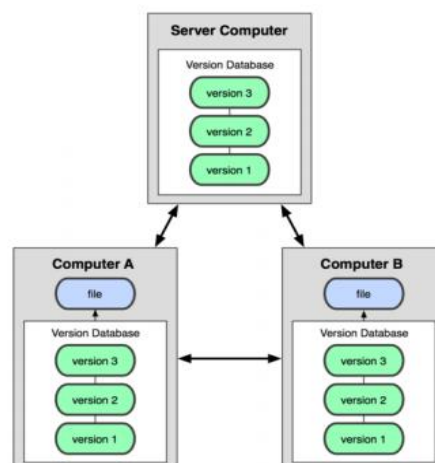


Figura 1.4.4. 3 (Método de control de versiones Distribuida)

En la imagen se muestra como es un sistema de control de versiones distribuida, en cada directorio A y B existe una copia exacta del directorio padre.

En la Tabla 1.4.4. 1 se muestra un comparativo, entre algunas de las herramientas que existen actualmente:

Tabla 1.4.4. 1

Software	Desarrollo	Soporte	Modelo de repositorio	Sistemas Operativos	Costo
Plastic SCM	Codice Software	Activo	Cliente servidor y Distribuido	Linux, Windows , OS X	Free for up to 15 users; else starting at \$595 per seat
Git	Junio Hamano	Activo	Distribuido	POSIX, Windows, OS X	Free
CVS	The CVS Team	Hay soporte, pero no se han agregado actualizaciones	Cliente-Servidor	Unix-like, Windows, OS X	Free
Subversion (SVN)	Apache Software Foundation[8]	Activo	Cliente-Servidor	Unix-like, Windows, OS X	Free
Mercurial	Matt Mackall	Activo	Distribuido	Unix-like, Windows, OS X	Free
GNU Bazaar	Canonical Ltd.	Activo	Cliente servidor y Distribuido	Unix-like, Windows, OS X	Free
Monotone	Nathaniel Smith, Graydon Hoare	Activo	Distribuido	Unix-like, Windows, OS X	Free
Synergy	IBM Rational	Activo	Cliente servidor y Distribuido	Linux, Windows , Unix-like	Non-free Contact IBM Rational[9]
PVCS	Serena Software	Activo	Cliente-Servidor	Windows, Unix	Non-free

III. Herramientas para control de cambios de Software



Plastic SCM (Codice Software, 2018): Es un sistema de control de versiones distribuido desarrollado por la empresa Española Codice Software tiene como objetivos fundamentales, dar un mayor soporte al desarrollo paralelo, creación de ramas, seguridad y desarrollo distribuido.

Para favorecer el desarrollo paralelo, dividiendo el desarrollo en distintas ramas, siguiendo una determinada política de uso, protección, desprotección, contenidos. La principal diferencia entre el modelo de desarrollo paralelo de Plastic y otros estriba en que en lugar de realizar una copia de todo (o solo los metadatos) a cada nueva rama que se genera, las ramas son creadas con objetos vacíos, solo cuando un ítem es modificado, se asigna una nueva revisión a la rama. De este modo la rama solo contiene ficheros o directorios que se han modificado con respecto a su rama padre.

Debido a la infraestructura de Plastic para el desarrollo paralelo, puede manejar miles de ramas en un solo repositorio sin perdida notable de rendimiento.



Git (Git-fast version control, 2018): Es un software de código abierto para control de versiones diseñado por Linus Torvalds (Creador de Linux), basado en el modelo de control distribuido ver Figura 1.4.4. 3 (Método de control de versiones Distribuida), diseñado para manejar desde pequeños hasta grandes

proyectos de manera rápida, eficiente y confiable.

Su propósito es llevar registro de los cambios de archivo de computadora y coordinar el trabajo que varias personas realicen sobre archivos compartidos. Al principio Git se pensó como un motor de bajo nivel sobre el cual otros pudieran escribir la interfaz de usuario. Sin embargo, Git sea convertido desde entonces en un sistema de control de versiones con funcionalidad plena.

Es muy rápido y eficiente aun con grandes proyectos y tiene un increíble sistema de ramificaciones, es muchos sistemas de control de versiones este proceso puede ser costoso, pues a menudo se requiere crear una nueva copia del código, lo cual puede tomar mucho tiempo cuando se trata de proyectos grandes, muchos usuarios resaltan esto como uno de los puntos más fuertes de Git, haciendo así las operaciones de ramificación casi instantáneo.



CVS (CVS - Concurrent Versions System, 2005-2006): Es uno de los sistemas para control de versiones de software más viejos, es una herramienta fácil de usar, mantiene el registro de todo el trabajo y los cambios en los ficheros (Codigo fuente principalmente, en un único archivo para cada fichero correspondiente), permite que diferentes desarrolladores colaboren a la vez.

Es un sistema centralizado ver Figura 1.4.4. 2 (Método de control de versiones Centralizada), en donde un servidor guarda las versiones actuales del proyecto y su historial, del servidor se crea una copia con la que se puede trabajar e ingresar los cambios con comandos GNU, varios usuarios pueden sacar copias del proyecto al mismo tiempo. Posteriormente, cuando incluyan sus modificaciones al proyecto, el servidor trata de acoplar las diferentes versiones. Si esto falla, debido a que dos clientes tratan de cambiar la misma línea en un archivo en particular, entonces el servidor deniega la segunda actualización e informa al cliente sobre el conflicto, que el usuario deberá resolver manualmente.



SVN (Apache Software Foundation, 2017): Apache subversión es una herramienta de código abierto para control de versiones, basada en un repositorio cuyo funcionamiento se asemeja al de un sistema de

ficheros. Es software libre bajo una licencia de tipo Apache/BSD.

Es un sistema centralizado y utiliza el concepto de revisión para guardar los cambios producidos en el repositorio. Entre dos revisiones sólo guarda el conjunto de modificaciones, optimizando así al máximo el uso de espacio en disco. SVN permite al usuario crear, copiar y borrar carpetas con la misma flexibilidad con la que lo haría si estuviese en su disco duro local. Dada su flexibilidad, es necesaria la aplicación de buenas prácticas para llevar a cabo una correcta gestión de las versiones del software generado.

Subversion puede acceder al repositorio a través de redes, lo que le permite ser usado varios usuarios en diferentes computadoras.



Mercurial (Mercurial, 2017): Es un sistema de control de versiones distribuido, esta implementado en Python, pero también incluye una parte de implementación en C. Mercurial fue desarrollado para ser usado sobre GNU/Linux pero ha sido adaptado para Window y Mac OS y la mayoría de otros sistemas Unix.

Las principales metas de desarrollo de Mercurial incluyen un gran rendimiento y escalabilidad, desarrollo completamente distribuido, sin necesidad de un servidor, es software libre, ayuda para tener una gestión robusta de archivos tanto de texto como binarios y capacidades avanzadas de ramificación e integración, todo ello manteniendo sencillez conceptual.

Para el acceso a repositorios mediante red, Mercurial usa un protocolo eficiente, basado en HTTP, que persigue reducir el tamaño de los datos a transferir, así como la proliferación de peticiones y conexiones nuevas.



GNU Bazaar (Bazaar, 2016): Es un sistema de control de versiones del tipo distribuido, es multiplataforma, que ayuda a dar seguimiento entre el historial del proyecto a través del tiempo y facilita el trabajo conjunto en proyectos

de software, es parte del proyecto GNU, es un software libre, al igual que mercurial es basado en Python.



Monotone (monotone, 2018): Es una herramienta para control de versiones de software, de código abierto, registra revisiones de ficheros, agrupa conjuntos de revisiones y mantiene históricos tras cambios de nombres, su principio de operación es distribuido

haciendo un uso intensivo primitivas criptográficas para trazar revisiones de ficheros y para autenticar acciones de usuarios, a pesar que es una herramienta muy estable estas acciones hacen que la herramienta sea lenta. Cada usuario mantiene su propio almacén de revisiones históricas en una base de datos SQLite local.



Synergy (Rational Synergy, 2012): es una herramienta de control de cambios de software

distribuida, que proporciona capacidades de gestión en el desarrollo de software, también proporciona el repositorio de la herramienta de administración de cambios conocida como Cambio Racional. Juntas, estas dos herramientas forman un entorno de gestión de configuración y gestión de configuración integrado que se utiliza en organizaciones de desarrollo de software que necesitan procesos controlados de SCM y una comprensión de lo que hay en una compilación de su software.

IV. Git

Para el desarrollo de este Modulo se ha decidido usar Git, debido a que es usado por muchos proyectos populares de código abierto como Android o Eclipse además cuenta con soporte y es gratuito.

La principal diferencia entre Git y cualquier otro sistema de control de versiones es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo. En cambio, Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del estado de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo —sólo un enlace al archivo anterior idéntico que ya tiene almacenado, ver Figura 1.4.4. 3 (Método de control de versiones Distribuida)

Luego de clonar o crear un repositorio el usuario tiene una copia completa del repositorio, y puede realizar operaciones de control de versiones contra este repositorio local como, por ejemplo:

- Modificar una serie de archivos en el directorio de trabajo (Working directory).
- Añadir modificaciones de los archivos al área de preparación (Staging area).
- Confirmar los cambios y tomarlos tal y como están en el área de preparación, y almacenarlos de manera permanente en el directorio de Git (git directory)

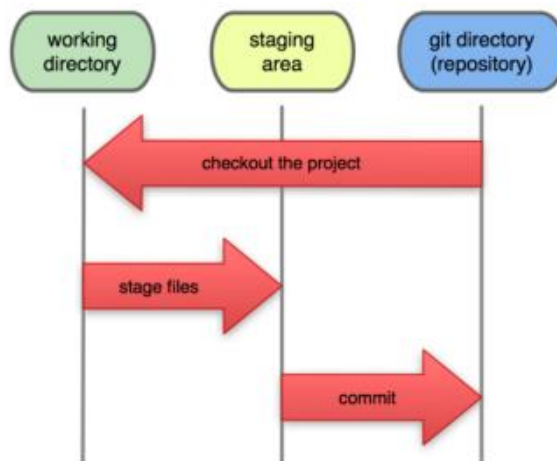


Figura 1.4.4. 4 (Operaciones locales)

Hay dos tipos de repositorios en Git:

- Repositorios “bare”, son generalmente usados en los servidores, aunque puede generarse en cualquier directorio local de quien desee crearlo. Se utiliza para compartir cambios provenientes de diferentes desarrolladores, no se utiliza para crear u compartir cambios desde sí mismo, ya que no cuenta con esas funcionalidades.
- Repositorios de trabajo, permiten crear archivos nuevos, generar cambios a los archivos existentes y a su vez crear nuevas versiones en el repositorio.

V. Comandos de Git

Git init - inicializar una carpeta como un repositorio Git.

Para crear un repositorio en una carpeta existente de archivos, puedes ejecutar el comando `git init` en esa carpeta

```
$ git init
```

Ahora se puede ver que hay una subcarpeta oculta llamada “.git” en el proyecto.

Este es tu repositorio donde se almacenan todos los cambios del proyecto.

Git clone - copiar un repositorio Git.

Si tienes que colaborar con alguien en un proyecto, o si deseas obtener una copia de un proyecto, debes ejecutar el comando `git clone [url]` con la URL del proyecto que deseas copiar, ejemplo:

```
$ git clone git://github.com/schacon/simplegit.git
```

Git add - agregar los contenidos de archivos al área de preparación.

En Git es necesario agregar previamente los cambios realizados al área de preparación, para luego poder hacer el commit correspondiente.

```
$ git add file.ext
```

Git status – Indica el estado de los archivos en el directorio de trabajo y en el área de preparación.

Se puede usar este comando para ver el estado de su área de preparación comparado con el código que se encuentra en su directorio de trabajo. Usando la opción “-s” nos mostrará la forma abreviada del informe.

```
$ git status -s
```

Git diff - muestra las diferencias de cambios que no están en el área de preparación.

Sin ningún argumento adicional, un simple git diff mostrará en formato unificado, qué código o contenido ha cambiado en el proyecto desde el último commit

```
$ git diff
```

Git diff --stat – muestra una lista de cambios en lugar de un diff completo por archivo

```
$ git diff --stat
```

La opción --stat, lo que nos dará un resumen de los cambios en su lugar.

Git commit – graba una instantánea del área de preparación.

Para guardar los cambios es necesario seguir el siguiente proceso, donde se finaliza con commit que es la confirmación de los cambios

```
$ git config - global user.name 'Tu nombre'
```

```
$ git config --global user.email tumail@dominio.com
```

```
$ git add file.ext
```

```
$ git status -s
```

```
$ git commit -m 'Agrega mis cambios al archivo'
```

Git reset – deshace cambios y commits

Se usa para deshacer el último commit y sacar del área de preparación los archivos modificados.

```
$ git reset HEAD -- file.ext
```

lo que hace en realidad es resetear las entradas del archivo en el “index” para que sea igual que en el commit anterior.

Git branch (nombre del branch) - crea una nueva rama

\$ *git branch rama*

\$ git Branch sin argumento lista las ramas existentes.

Git checkout -b (nombre del branch) - crear y cambiar inmediatamente a una rama

\$ *git checkout -b nueva_rama*

Con argumento -b crea una nueva rama y se cambia de inmediato a esta nueva rama

\$ *git checkout rama*

Se cambia a la rama especificada.

Git merge - fusionar una rama en tu contexto actual

Una vez que tengas el trabajo aislado en una rama, es muy probable que desees incorporarlo en la rama principal. Puedes combinar cualquiera de las ramas en su rama actual con el comando git merge

\$ *git merge rama*

Hace el merge de la rama principal master con la rama que se le indique

VI. Practica 1.4.4.1 Inicio con Git

A continuación, se describen los pasos a seguir para la primera práctica, con lo que se pretende, se ejecuten y comprendan los comandos básicos,

también que se entienda como funciona Git, *-git log*

-git config *-git diff*

-git add

-git status

-git commit

-git checkout

-git init *-git log*

-git config *-git diff*

-git add

-git status

-git commit

-git checkout

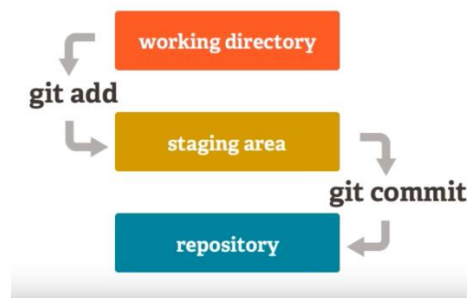


Figura 1.4.4. 5

1) Descargar la herramienta: puede ser desde la página de descargas de Git <https://git-scm.com/downloads>, o entrando a la plataforma de moodle, en el Módulo 1 sección 1.4.4. Herramientas para Control de Versiones de Software, entrar a la carpeta Herramientas, y descargar la carpeta Git.

- **Instalación:** Una vez que se tiene el ejecutable ver figuras (Figura 1.4.4. 6 (Paso 1) a Figura 1.4.4. 18 (Paso 13)), estas muestran los pasos a seguir para la instalación de Git.


Nombre	Fecha de modifica...	Tipo	Tamaño
 Git-2.18.0-64-bit	11/07/2018 09:42 ...	Aplicación	40,164 KB

Figura 1.4.4. 6 (Paso 1)

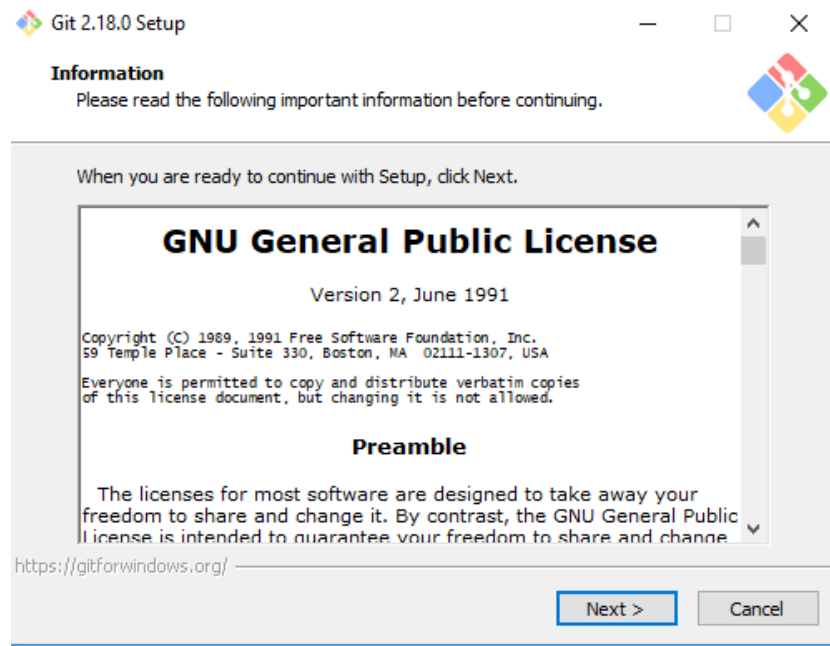


Figura 1.4.4. 7 (Paso 2)

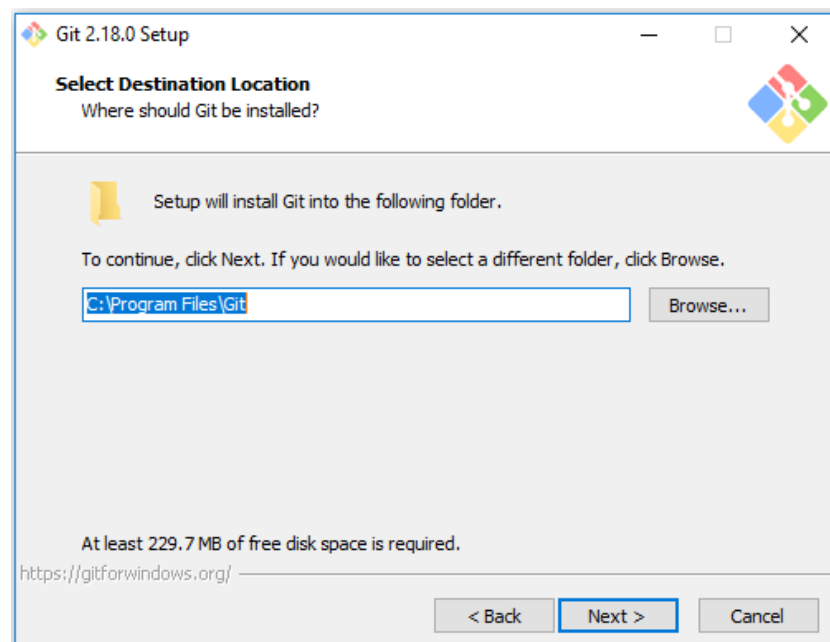


Figura 1.4.4. 8 (Paso 3)

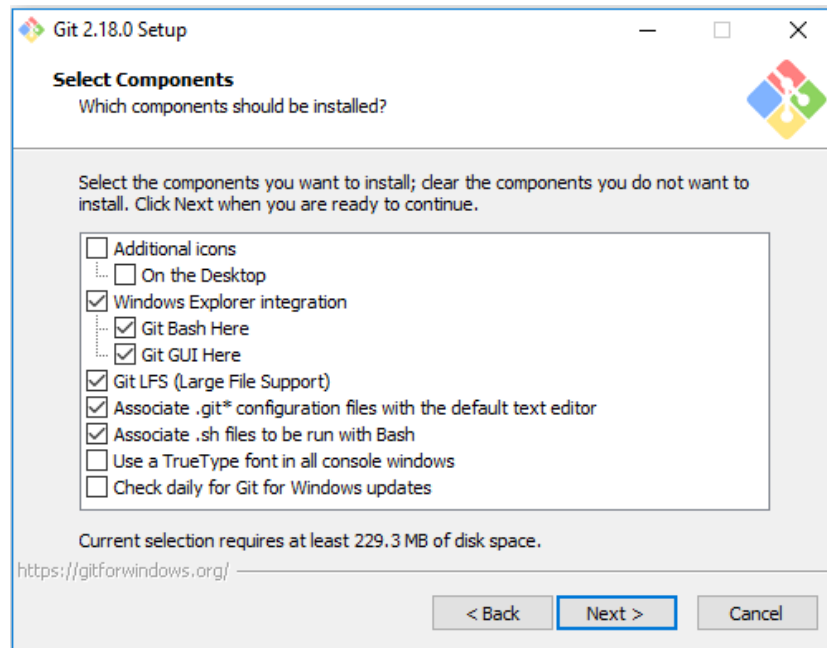


Figura 1.4.4. 9 (Paso 4)

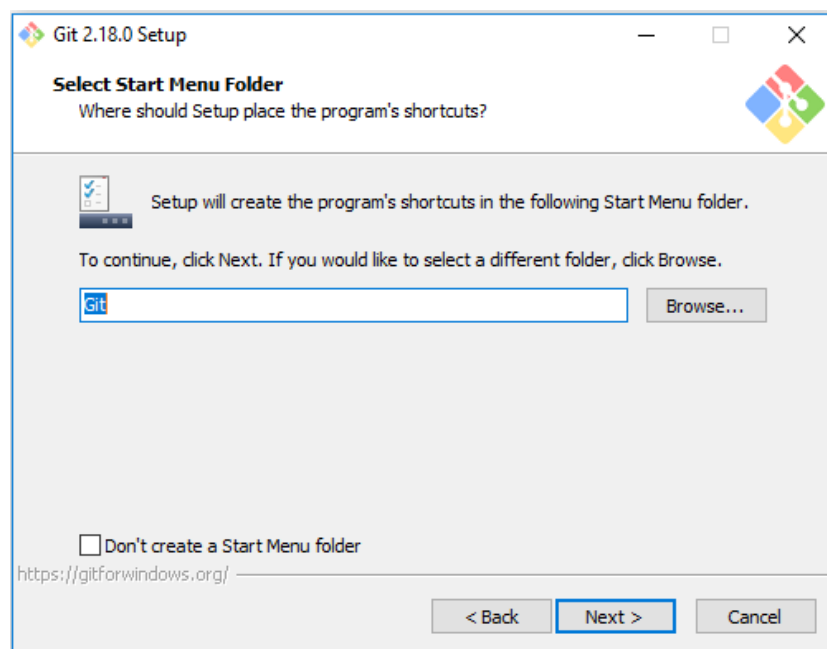


Figura 1.4.4. 10 (Paso 5)

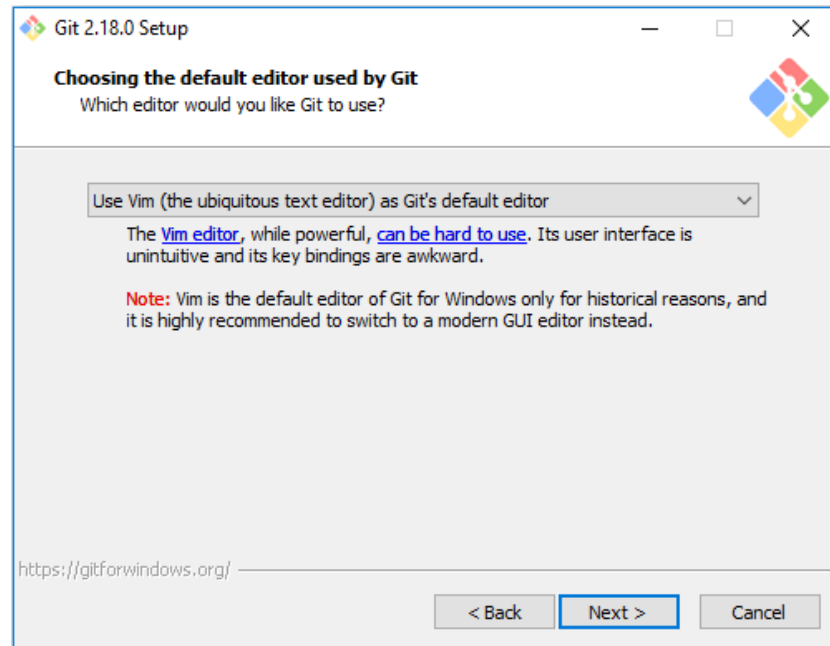


Figura 1.4.4. 11 (Paso 6)

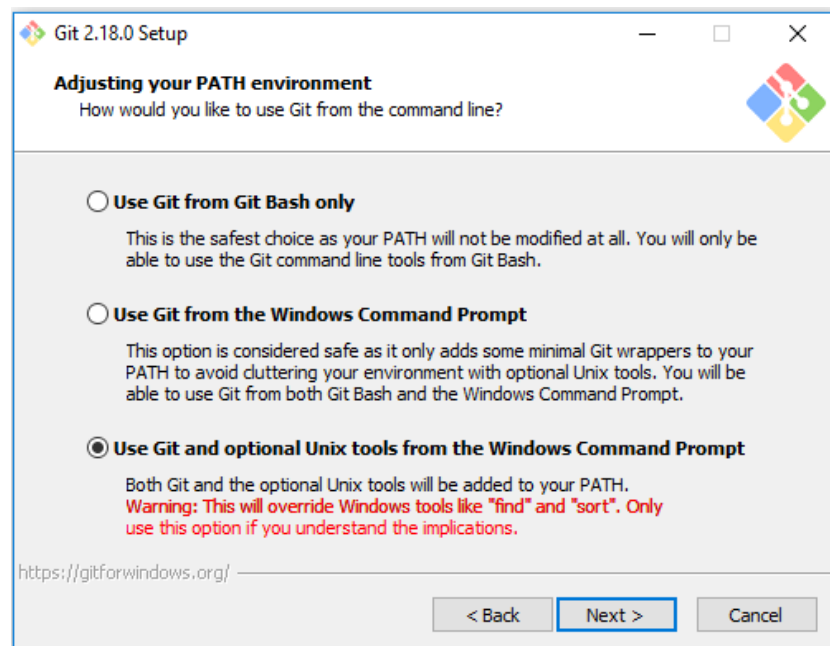


Figura 1.4.4. 12 (Paso 7)

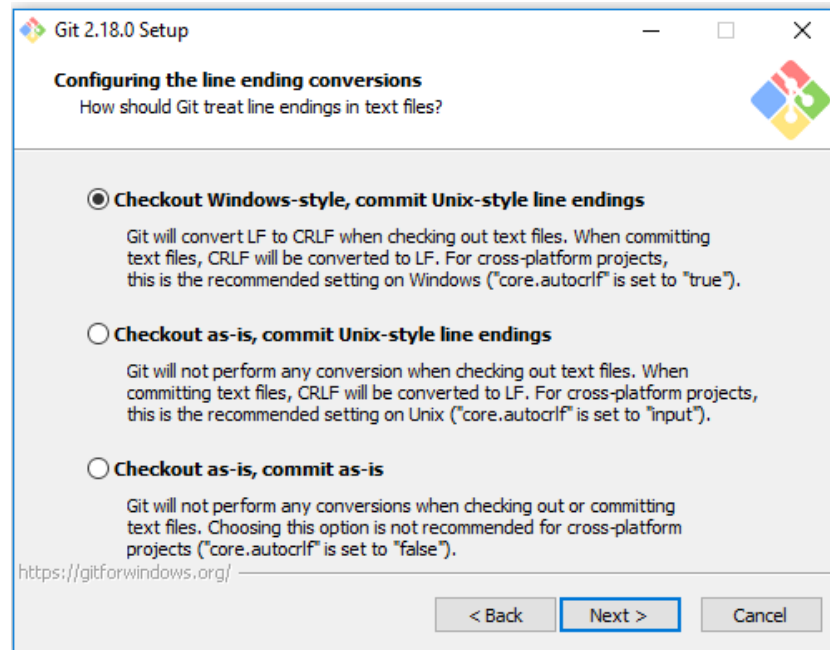


Figura 1.4.4. 13 (Paso 8)

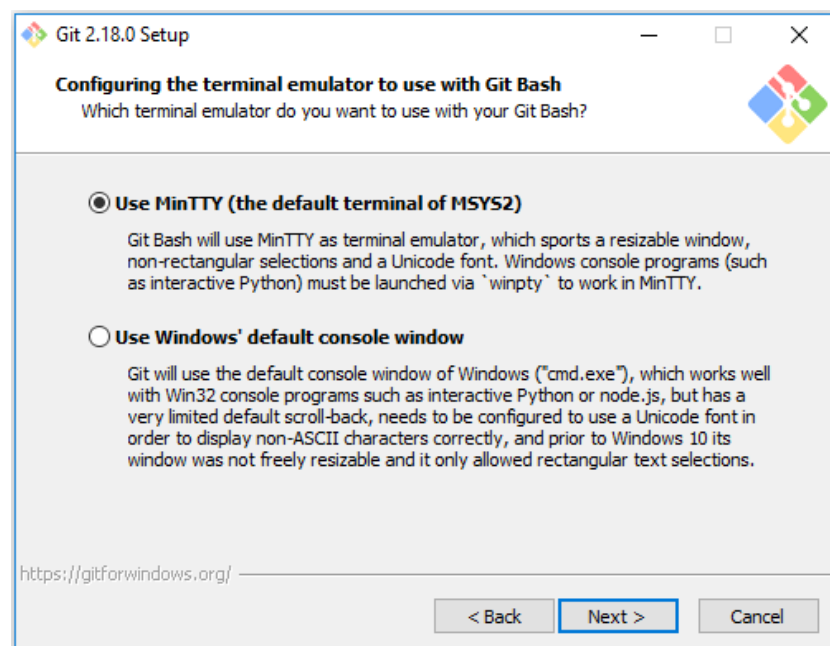


Figura 1.4.4. 14 (Paso 9)

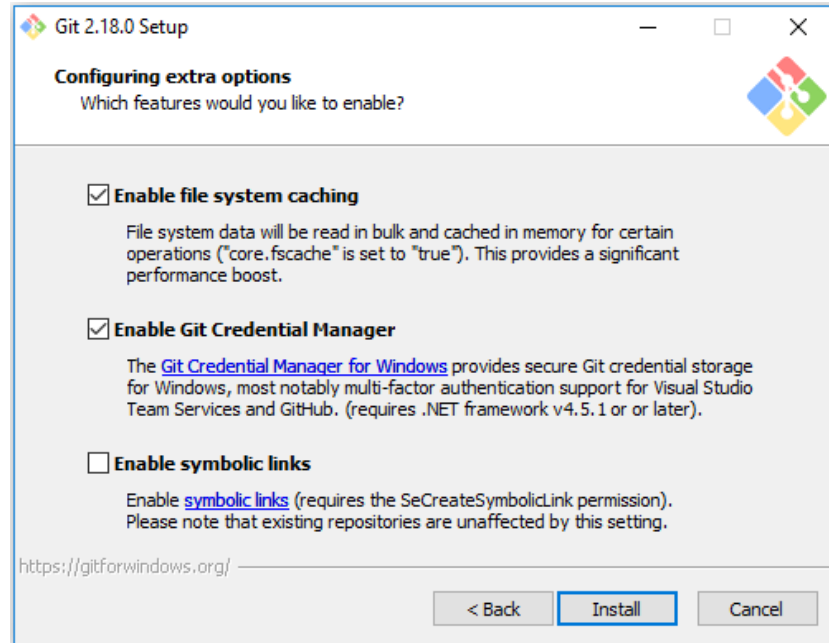


Figura 1.4.4. 15 (Paso 10)

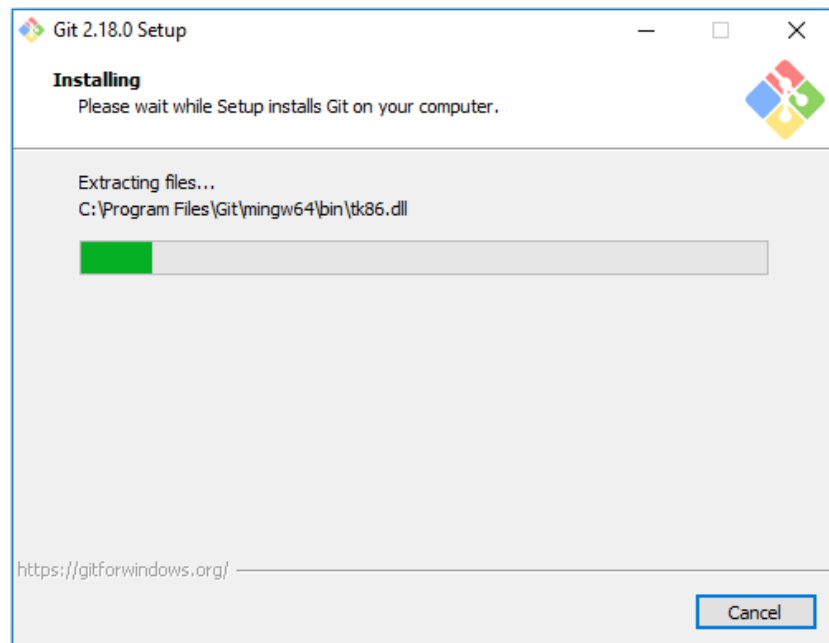


Figura 1.4.4. 16 (Paso 11)

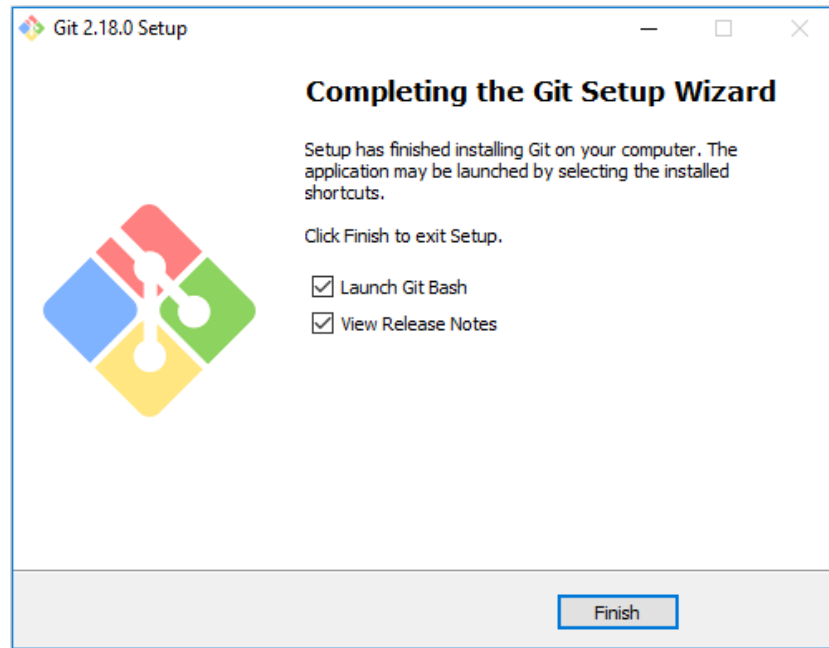


Figura 1.4.4. 17 (Paso 12)

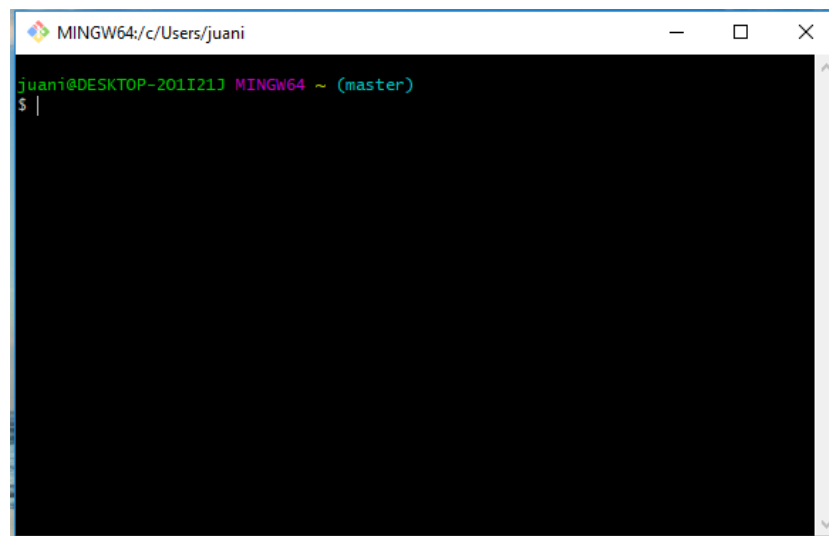


Figura 1.4.4. 18 (Paso 13)

- 2) Crear Proyecto:** Crear un repositorio con el nombre del proyecto y guardarlo en Documents, ahora usando el editor de texto de su preferencia generar un archivo y guardarlo con la extensión .c, Figura 1.4.4. 19, ya está todo listo para comenzar a usar Git.

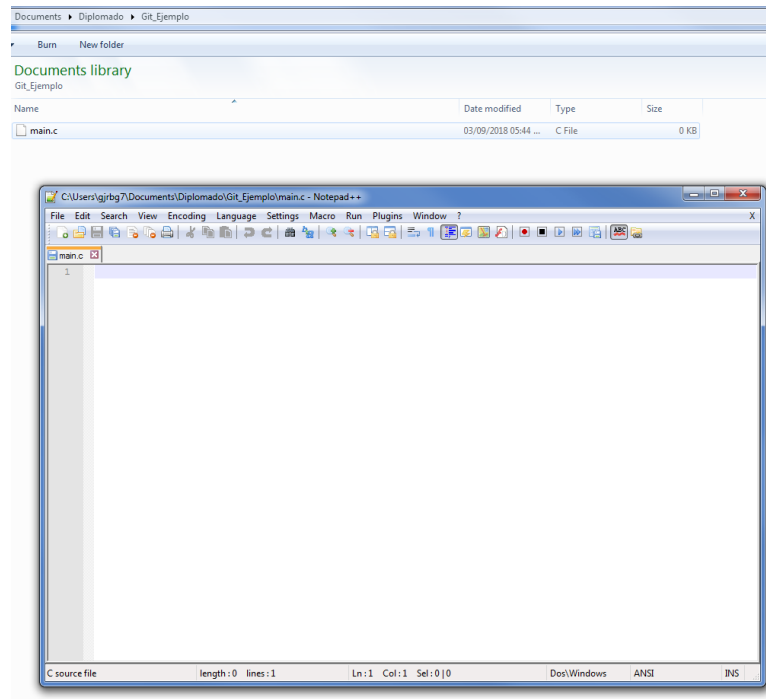


Figura 1.4.4. 19 (Crear área de trabajo)

- 3) Abrir Consola De Git:** Ver Figura 1.4.4. 20

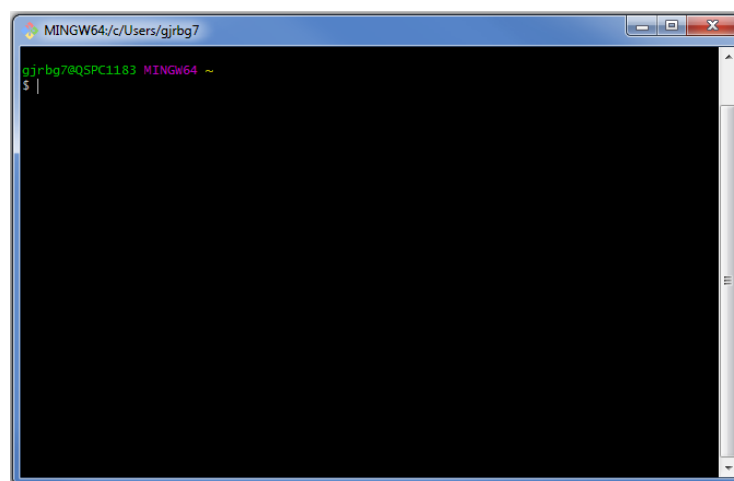
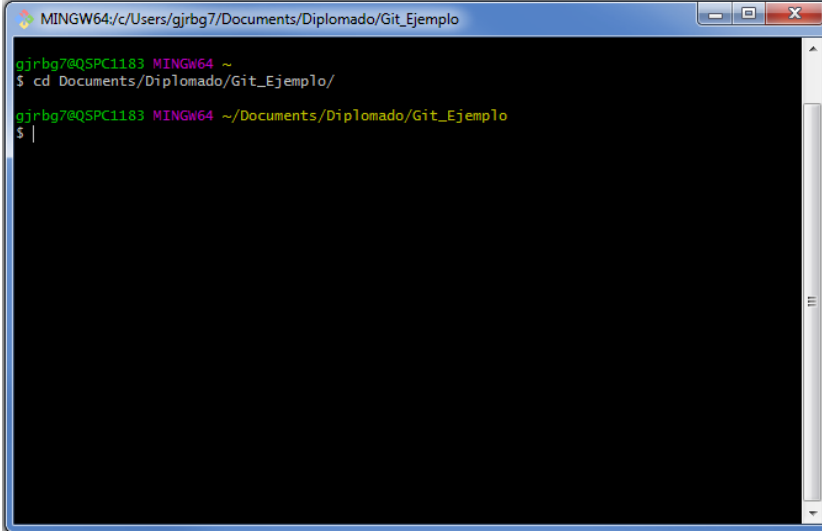


Figura 1.4.4. 20

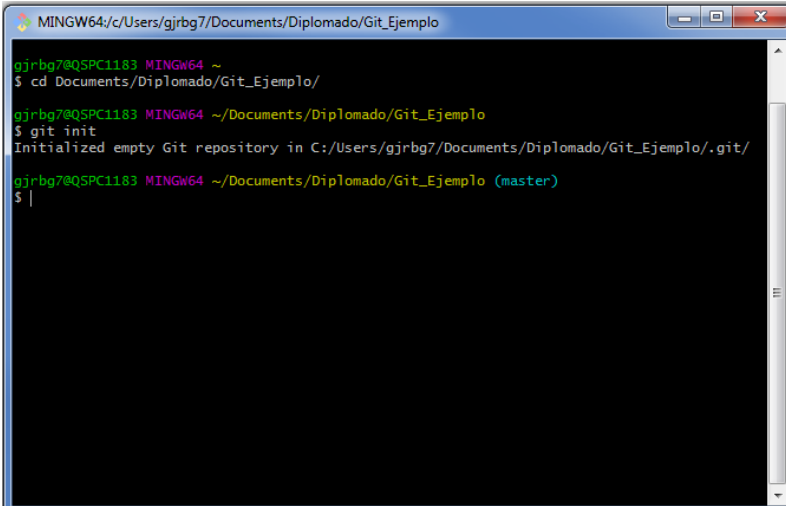
- 4) **Inicializar repositorio de trabajo:** Git es similar a una consola del sistema operativo, entonces para moverse a través de los repositorios se usa el comando `cd`, hasta ubicarse en la ruta de nuestro repositorio de trabajo y dar enter para validar la ruta Figura 1.4.4. 21



```
MINGW64/c/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo
gjrbg7@Q5PC1183 MINGW64 ~
$ cd Documents/Diplomado/Git_Ejemplo/
gjrbg7@Q5PC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo
$ |
```

Figura 1.4.4. 21

Para comenzar a gestionar el proyecto con Git ejecutar el comando: `git init` seguido de enter, Figura 1.4.4. 22, al ejecutar este comando se le informa a Git que se hará gestión de versiones de este repositorio.



```
MINGW64/c/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo
gjrbg7@Q5PC1183 MINGW64 ~
$ cd Documents/Diplomado/Git_Ejemplo/
gjrbg7@Q5PC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo
$ git init
Initialized empty Git repository in C:/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo/.git/
gjrbg7@Q5PC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ |
```

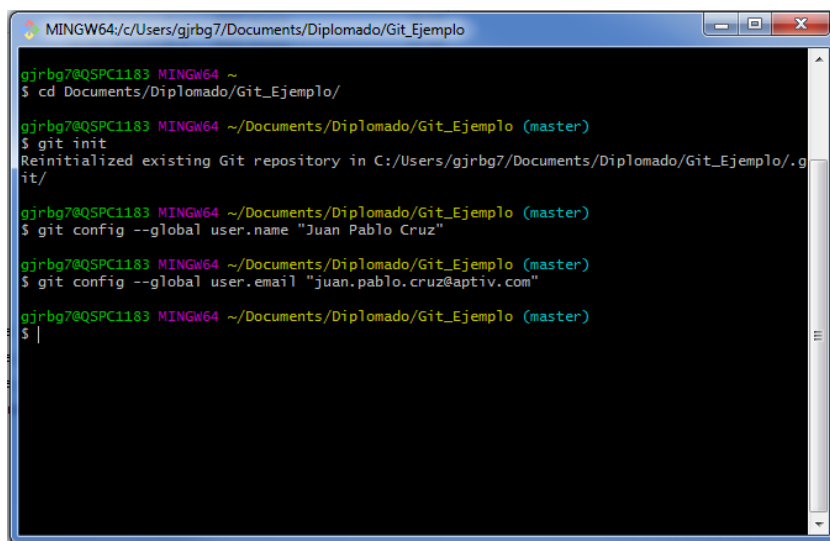
Figura 1.4.4. 22

5) Personalizar el Entorno Git: Es necesario hacerlo una sola vez, estas configuraciones se pueden cambiar en cualquier momento volviendo a ejecutar estos comandos, “en algunas versiones de Windows toma el nombre de usuario y correo del sistema”, para esto ejecutar:

```
git config --global user.email "tucorreo@dominio.com"
```

```
git config --global user.name "Nombre de usuario"
```

Es importante establecer esto debido a que las confirmaciones de cambios en Git usan esta información y es introducida de manera inmutable en los commit, esto para tener un registro de quien ha modificado el repositorio, Figura 1.4.4. 23.



```
MINGW64/c:/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo
gjrbg7@QSPC1183 MINGW64 ~
$ cd Documents/Diplomado/Git_Ejemplo/
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git init
Reinitialized existing Git repository in C:/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo/.git/
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git config --global user.name "Juan Pablo Cruz"
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git config --global user.email "juan.pablo.cruz@aptiv.com"
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ |
```

Figura 1.4.4. 23

6) Agregar archivos: Después de inicializar el repositorio, verificar su estatus usando el comando *git status* seguido de enter, Figura 1.4.4. 24, aquí se puede ver el archivo main.c en rojo, esto quiere decir que no se está siguiendo aún y sigue situado en el working directory, ahora para pasar al staging area, se usa el comando *git add* y en seguida el comando *git status*, ver Figura 1.4.4. 25, ahora el archivo cambio a verde, esto quiere decir que ya se está siguiendo.

```
MINGW64/c/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo
$ cd Documents/Diplomado/Git_Ejemplo/
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git init
Reinitialized existing Git repository in C:/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo/.git/
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git config --global user.name "Juan Pablo Cruz"
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git config --global user.email "juan.pablo.cruz@aptiv.com"
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    main.c

nothing added to commit but untracked files present (use "git add" to track)
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ |
```

Figura 1.4.4. 24

```
MINGW64/c/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    main.c

nothing added to commit but untracked files present (use "git add" to track)
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git add main.c
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   main.c

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ |
```

Figura 1.4.4. 25

- Ahora usar el comando *git commit* y aceptar dando enter, aquí será el punto de partida para las futuras versiones de este repositorio, lo anterior es como si se tomara una fotografía del proyecto y se guardara un registro.

Después de hacer commit, se abre un editor de texto en el que se puede agregar información presionando la letra i, seguida del texto (el

texto queda en amarillo), para salir hay que presionar la tecla esc seguido de :wq y enter ver Figura 1.4.4. 27

```
MINGW64~/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    main.c

nothing added to commit but untracked files present (use "git add" to track)

gjrbg7@Q5PC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git add main.c

gjrbg7@Q5PC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   main.c

gjrbg7@Q5PC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git commit
```

Figura 1.4.4. 26

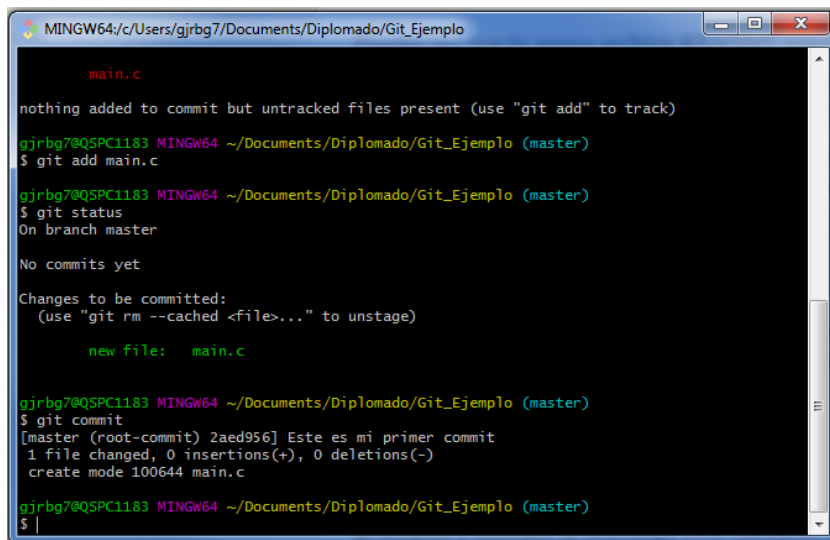
A screenshot of a Windows terminal window titled "MINGW64/c:/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo". The terminal displays the following text:

```
Este es mi primer commit  
# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.  
#  
# On branch master  
#  
# Initial commit  
#  
# Changes to be committed:  
#   new file:   main.c  
#  
~  
~  
~  
~  
~  
~
```


At the bottom of the terminal, there is a status bar indicating the current directory as "`</Documents/Diplomado/Git_Ejemplo/.git/COMMIT_EDITMSG[+] [unix]`", the date and time as "(15:06 04/09/2018)", and some additional information like "1,24 All" and ":wq".

Figura 1.4.4. 27

- En consola se muestra un resumen de la operación commit ejecutada anteriormente, Figura 1.4.4. 28, ahora usando *git log* se pueden ver todos los commit de este repositorio junto con un id y el texto que se agregó en el editor, Figura 1.4.4. 29, esto sirve para identificar el commit, hasta aquí solo existe uno.



```
MINGW64/c/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo

main.c

nothing added to commit but untracked files present (use "git add" to track)

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git add main.c

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git status
On branch master

No commits yet

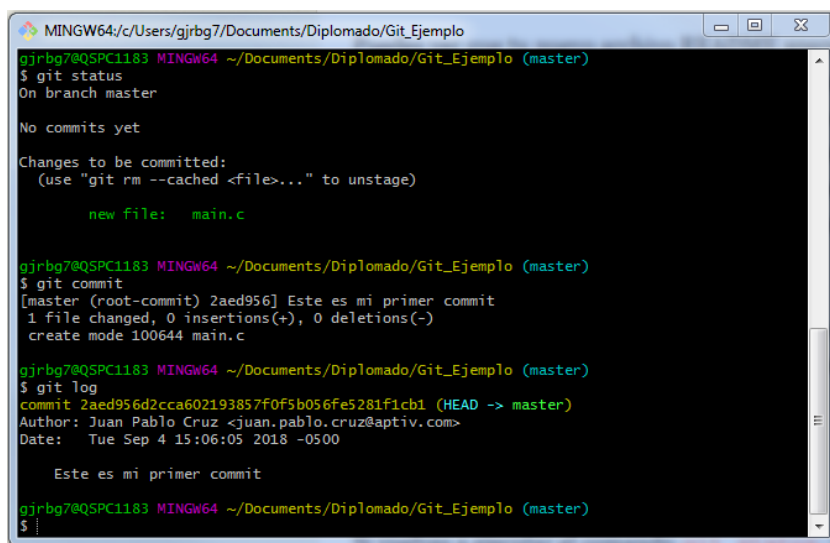
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   main.c

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git commit
[master (root-commit) 2aed956] Este es mi primer commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 main.c

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$
```

Figura 1.4.4. 28



```
MINGW64/c/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   main.c

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git commit
[master (root-commit) 2aed956] Este es mi primer commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 main.c

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git log
commit 2aed956d2cca602193857f0f5b056fe5281f1cb1 (HEAD -> master)
Author: Juan Pablo Cruz <juan.pablo.cruz@aptiv.com>
Date: Tue Sep 4 15:06:05 2018 -0500

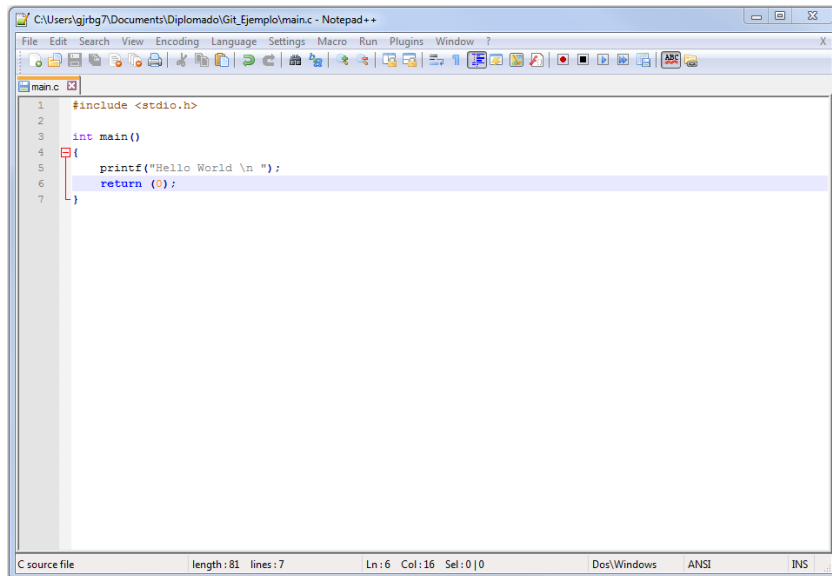
    Este es mi primer commit

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$
```

Figura 1.4.4. 29

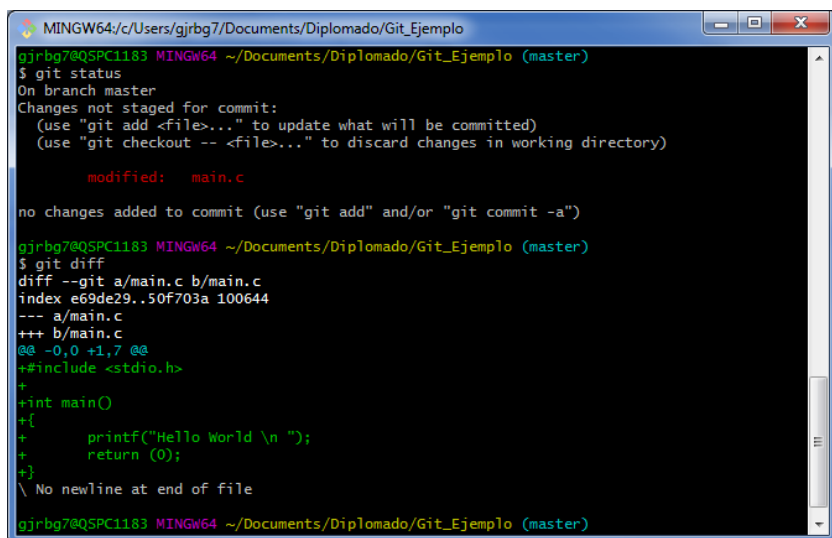
7) **Ver diferencias y agregar nuevos cambios:** Ahora usando un editor de texto abrir y modificar el archivo main.c, Figura 1.4.4. 30 ver que, al guardar, automáticamente git detectara las diferencias, ahora en la consola ejecutar el comando *git diff*, esto desplegara todas las diferencias que existen el archivo con respecto al commit anterior

Figura 1.4.4. 31, por defecto las diferencias se muestran en la consola de git, pero se puede configurar para usar otra herramienta, lo cual es recomendable pues en la consola es difícil apreciar diferencias cuando son muchas o el archivo es muy grande.



```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello World \n ");
6     return (0);
7 }
```

Figura 1.4.4. 30



```
MINGW64/c/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git status
On branch master
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:   main.c

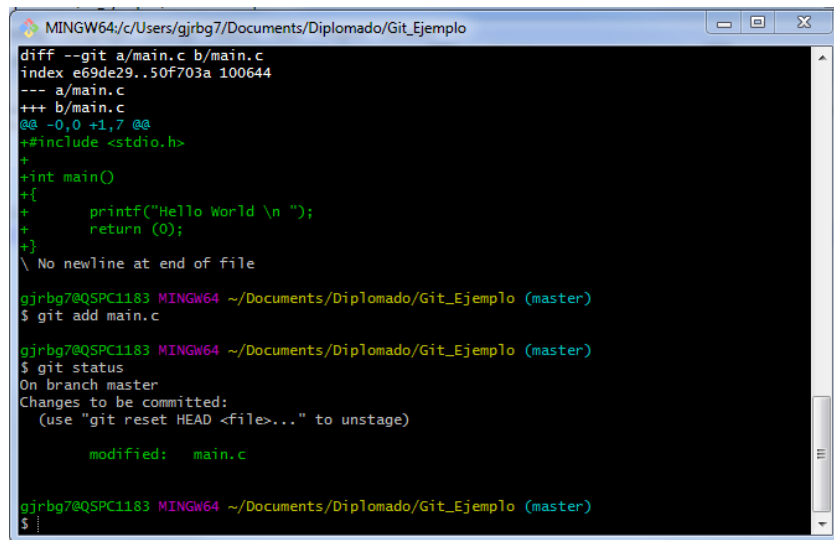
no changes added to commit (use "git add" and/or "git commit -a")

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git diff
diff --git a/main.c b/main.c
index e69de29..50f703a 100644
--- a/main.c
+++ b/main.c
@@ -0,0 +1,7 @@
+#include <stdio.h>
+
+int main()
+{
+    printf("Hello World \n ");
+    return (0);
+}
\ No newline at end of file

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
```

Figura 1.4.4. 31

- Agregar el archivo, ejecutando *git add* y confirmar estatus, ejecutando *git status* Figura 1.4.4. 32, después ejecutar el comando *git commit -m* “El archivo main fue modificado” y enter, al incluir el argumento -m la consola detecta que se insertara mensaje y ya no abre el editor de texto de Figura 1.4.4. 27, por ultimo con *git log* vemos el histórico de los cambios Figura 1.4.4. 33.



```
MINGW64/c/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo
diff --git a/main.c b/main.c
index e69de29..50f703a 100644
--- a/main.c
+++ b/main.c
@@ -0,0 +1,7 @@
+#include <stdio.h>
+
+int main()
+{
+    printf("Hello World \n");
+    return (0);
+}
\ No newline at end of file

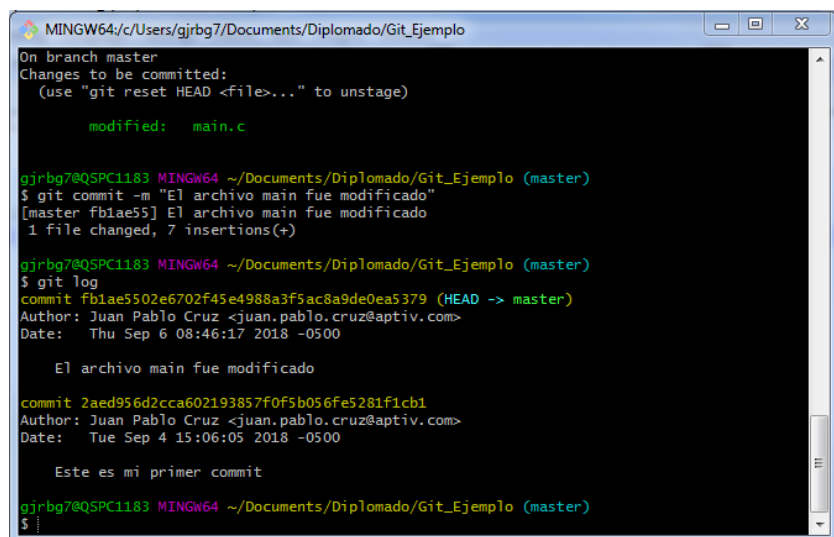
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git add main.c

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   main.c

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$
```

Figura 1.4.4. 32



```
MINGW64/c/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   main.c

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git commit -m "El archivo main fue modificado"
[master fb1ae55] El archivo main fue modificado
1 file changed, 7 insertions(+)

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ git log
commit fb1ae5502e6702f45e4988a3f5ac8a9de0ea5379 (HEAD -> master)
Author: Juan Pablo Cruz <juan.pablo.cruz@aptiv.com>
Date:   Thu Sep 6 08:46:17 2018 -0500

    El archivo main fue modificado

commit 2aed956d2cca602193857f0f5b056fe5281f1cb1
Author: Juan Pablo Cruz <juan.pablo.cruz@aptiv.com>
Date:   Tue Sep 4 15:06:05 2018 -0500

    Este es mi primer commit

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$
```

Figura 1.4.4. 33

8) Excluir archivos: Esta aplicación se usa para que git no tome en cuenta los repositorios o archivos que se le indican por ejemplo al usar algún entorno de desarrollo cuando se compila el proyecto se generan archivos de salida, de los que no interesa llevar un control de versiones pues son de uso exclusivo del entorno de desarrollo, en este ejemplo se generó una carpeta de nombre salida con un par de archivos dentro Figura 1.4.4. 34, al ejecutar *git status* indica que hay nuevos archivos sin seguir, los cuales se van a ignorar.

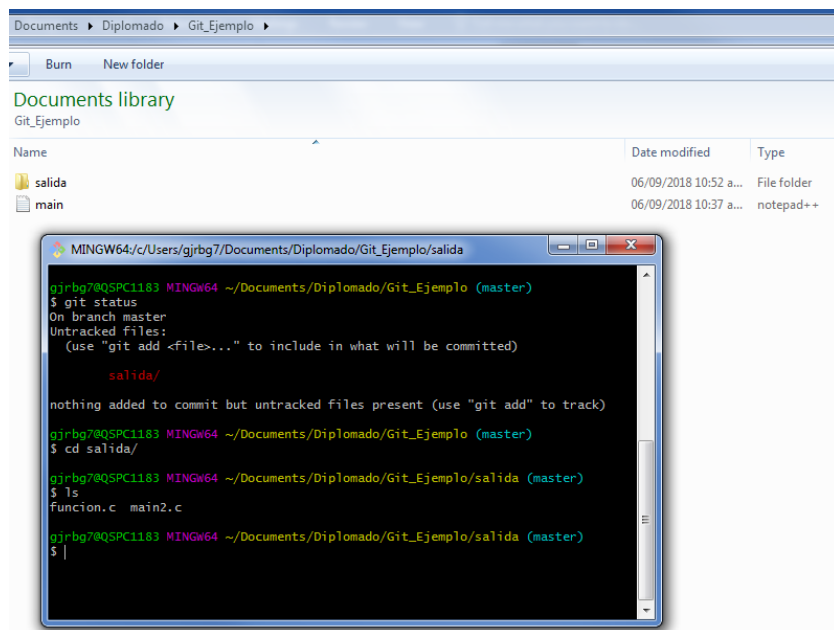


Figura 1.4.4. 34

- Para excluirlos hay que generar un archivo con la extensión *.gitignore* y listar los archivos que se desean ignorar “salida”, Figura 1.4.4. 35.

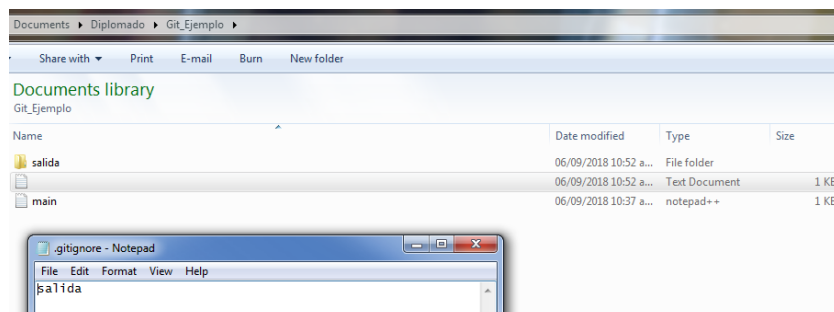
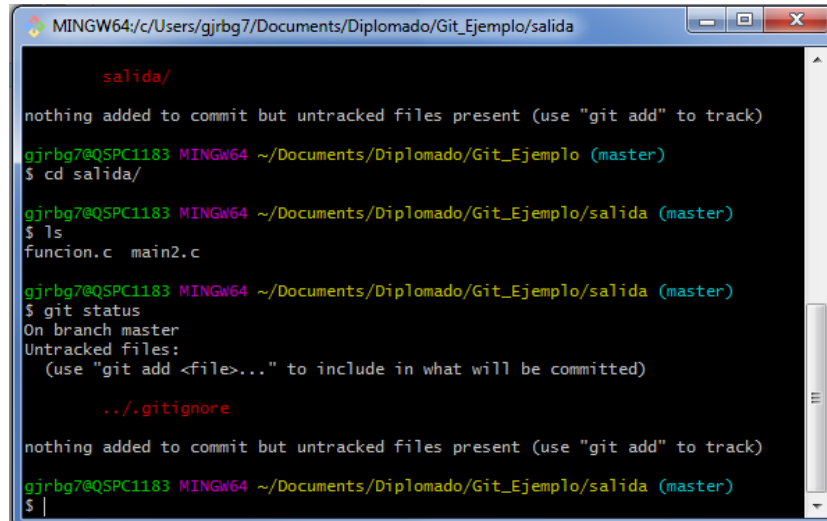


Figura 1.4.4. 35

- En la Figura 1.4.4. 36, se ve que la carpeta salida, está siendo ignorada, pero el archivo .gitignore ha aparecido, hay que agregarlo y validarlo en el proyecto usando *git add* y *git commit*.



```
MINGW64/c/Users/gjrbg7/Documents/Diplomado/Git_Ejemplo/salida

salida/
nothing added to commit but untracked files present (use "git add" to track)
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo (master)
$ cd salida/

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo/salida (master)
$ ls
funcion.c  main2.c

gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo/salida (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    ../.gitignore

nothing added to commit but untracked files present (use "git add" to track)
gjrbg7@QSPC1183 MINGW64 ~/Documents/Diplomado/Git_Ejemplo/salida (master)
$ |
```

Figura 1.4.4. 36

VII. Practica 1.4.4.2 Crear repositorio Github

- 1) **Registrarse:** Ir a la página de github <https://github.com/>, y registrarse para tener acceso al servidor aquí se puede subir y descargar proyectos.

A continuación, se muestra el proceso de registro en github ver (Figura 1.4.4. 37 a Figura 1.4.4. 39)

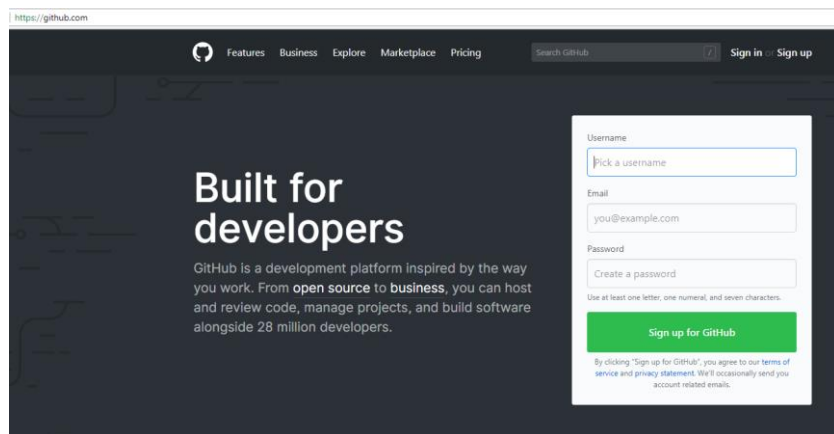


Figura 1.4.4. 37

Welcome to GitHub

You've taken your first step into a larger world, @jprbg27.

✓ Completed Set up a personal account	🔧 Step 2: Choose your plan	⚙️ Step 3: Tailor your experience
--	-------------------------------	--------------------------------------

Choose your personal plan

☒ Unlimited public repositories for free.

☐ Unlimited private repositories for \$7/month.

Don't worry, you can cancel or upgrade at any time.

☐ Help me set up an organization next
Organizations are separate from personal accounts and are best suited for businesses who need to manage permissions for many employees.
[Learn more about organizations](#)

☐ Send me updates on GitHub news, offers, and events
Unsubscribe anytime in your email preferences. [Learn more](#)

Continue

Both plans include:

- ✓ Collaborative code review
- ✓ Issue tracking
- ✓ Open source community
- ✓ Unlimited public repositories
- ✓ Join any organization

Figura 1.4.4. 38

Welcome to GitHub

You'll find endless opportunities to learn, code, and create, @jprbg27.

✓ Completed
Set up a personal account

🔧 Step 2:
Choose your plan

⚙️ Step 3:
Tailor your experience

How would you describe your level of programming experience?

☐ Totally new to programming ☐ Somewhat experienced ☐ Very experienced

What do you plan to use GitHub for? (check all that apply)

☐ Project Management ☐ Research ☐ School projects
☐ Design ☐ Development ☐ Other (please specify)

Which is closest to how you would describe yourself?

☐ I'm a professional ☐ I'm a hobbyist ☐ I'm a student
☐ Other (please specify)

What are you interested in?

e.g. tutorials, android, ruby, web-development, machine-learning, open-source

[Submit](#) [skip this step](#)

Figura 1.4.4. 39

2) Crear un repositorio en el servidor: Una vez en registrado, se muestra la siguiente página Figura 1.4.4. 40, este es el entorno de inicio, a partir de aquí se creará un nuevo proyecto, dando click en Start a Project, la página cambiará y se desplegará algo como en la Figura 1.4.4. 41

Learn Git and GitHub without any code!

Using the Hello World guide, you'll create a repository, start a branch, write comments, and open a pull request.

[Read the guide](#) [Start a project](#)

Our new Terms of Service and Privacy Statement are in effect.

Repositories [New repository](#)

You don't have any repositories yet!


Browse activity [Discover repositories](#)

Discover interesting projects and people to populate your personal news feed.

Your news feed helps you keep up with recent activity on repositories you [watch](#) and people you [follow](#).

[Explore GitHub](#)

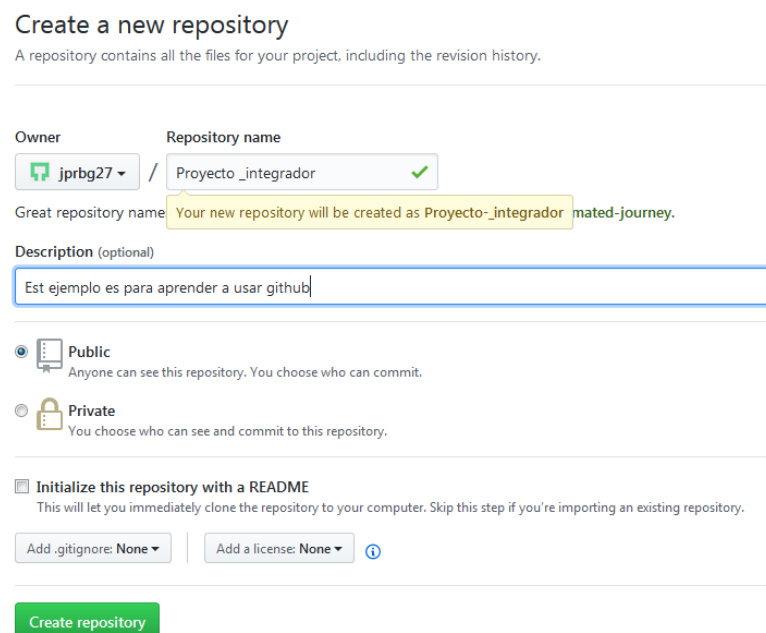
© 2018 GitHub, Inc. [Terms](#) [Privacy](#) [Security](#) [Status](#) [Help](#)



[Contact GitHub](#) [Pricing](#) [API](#) [Training](#) [Blog](#) [About](#)

Figura 1.4.4. 40

- En este punto se deben llenar los campos de Repository name y se recomienda llenar el campo Description, con información relacionada al proyecto, se selecciona el modo público, esto quiere decir que cualquier persona tendrá acceso al repositorio, en modo privado se restringe el acceso, pero esto tiene un costo mensual, por ultimo dar click en Create repository



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: jprbg27 / Repository name: Proyecto_integrador ✓

Great repository name: Your new repository will be created as Proyecto-integrador mated-journey.

Description (optional): Est ejemplo es para aprender a usar github

☒ Public
Anyone can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

☐ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None ⓘ

Create repository

Figura 1.4.4. 41

3) Subir cambios al repositorio: una vez creado el repositorio en el servidor la página mostrara algo como la **¡Error! No se encuentra el origen de la referencia.**, aquí ya se pueden subir los cambios al repositorio, para esto se necesita copiar la URL que se genera donde dice HTTPS y ejecutar los siguientes comandos en este orden:

git remote add origin https://github.com/jprbg27/Proyecto_Integrador.git

git push -u origin master

- Después de ejecutar el comando *git push*, hay que esperar un momento hasta que se abra una ventana de inicio de github Figura 1.4.4. 43, en la que es necesario registrarse para poder subir los archivos al servidor.

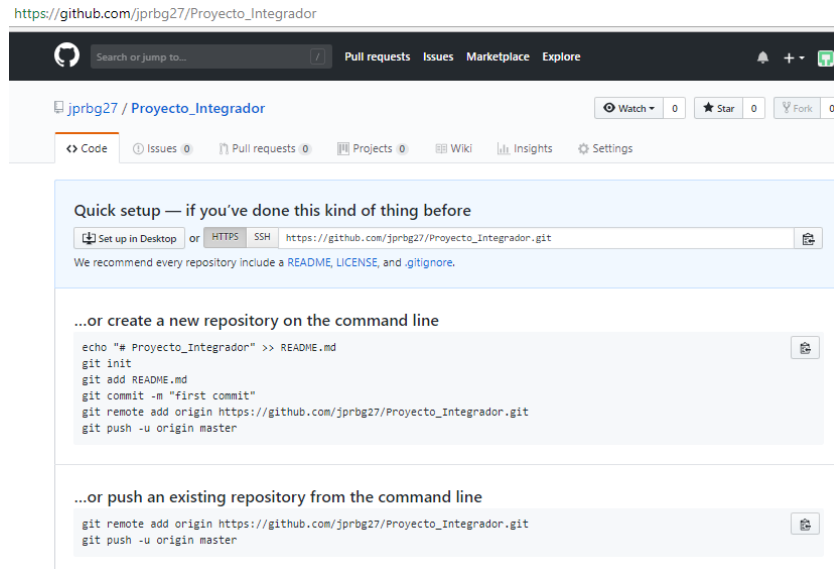


Figura 1.4.4. 42

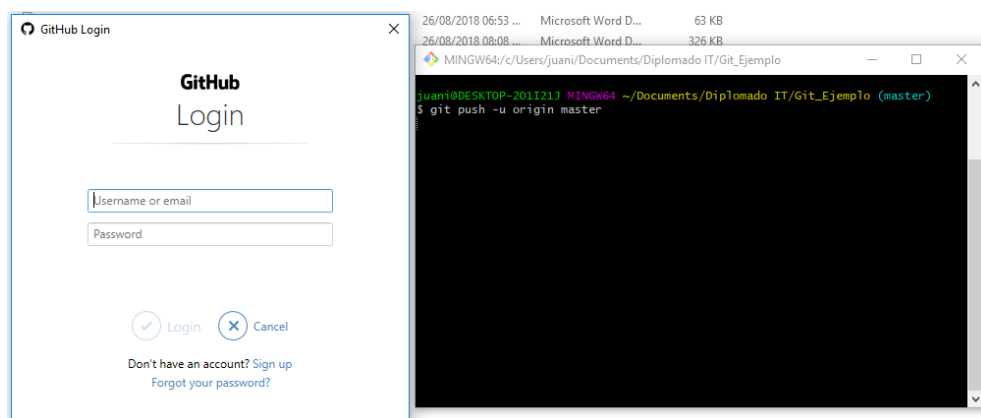
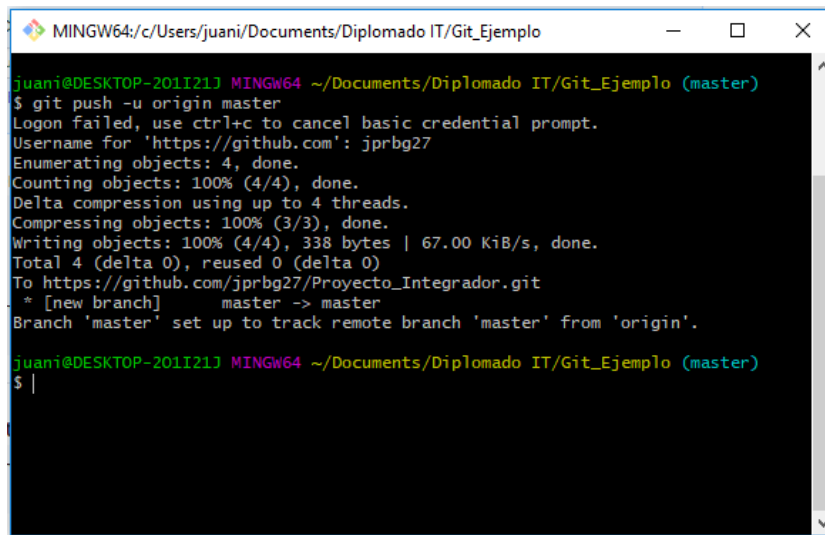


Figura 1.4.4. 43

- Después de hacer registro, en la consola aparecen los mensajes que nos indican que los archivos se han subido exitosamente a nuestro repositorio ver Figura 1.4.4. 44, para confirmar que los archivos se subieron correctamente al repositorio cambiar a la página de [GitHub](#) y refrescar la página aquí ya se encuentran los archivos que se agregaron al proyecto Figura 1.4.4. 45, es recomendable agregar el read.mk que brinda información del repositorio, dar click en add readme

y después en commit new file y este archivo read se habrá agregado al repositorio.



```
MINGW64:/c/Users/juani/Documents/Diplomado IT/Git_Ejemplo
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git push -u origin master
Logon failed, use ctrl+c to cancel basic credential prompt.
Username for 'https://github.com': jprbg27
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 338 bytes | 67.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To https://github.com/jprbg27/Proyecto_Integrador.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ |
```

Figura 1.4.4. 44

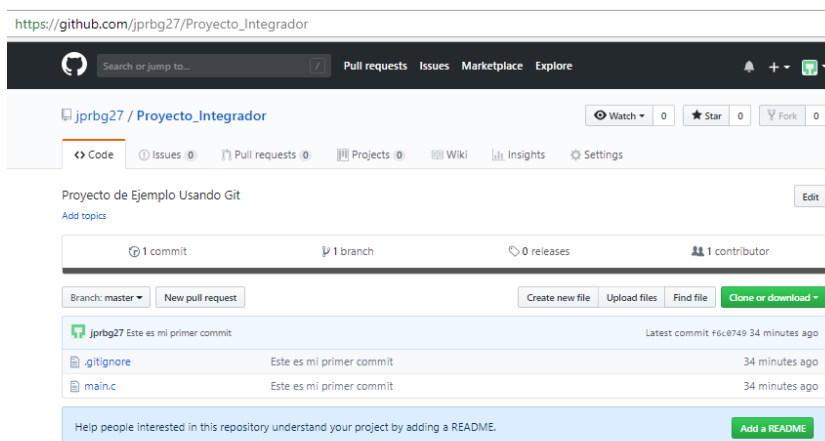


Figura 1.4.4. 45

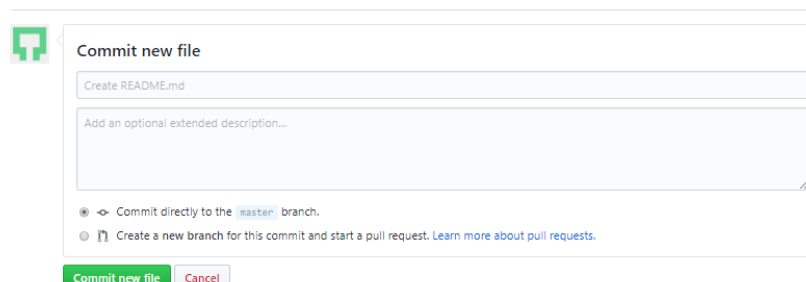
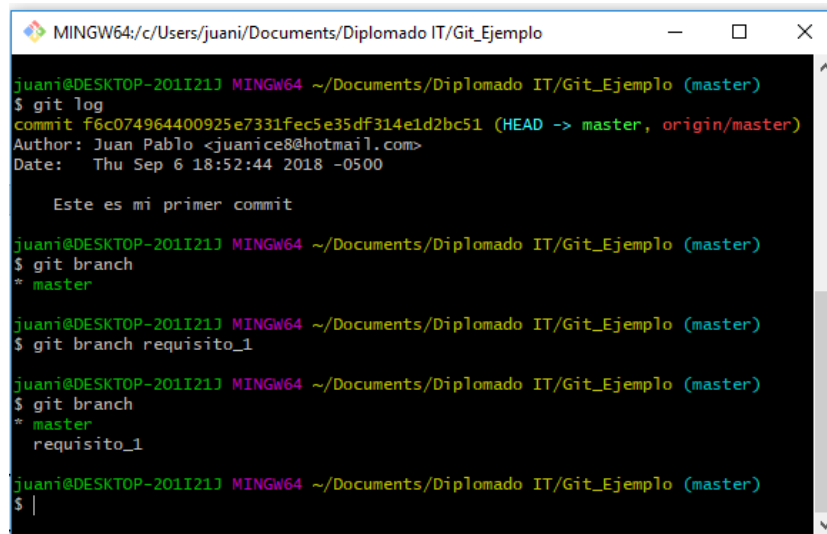


Figura 1.4.4. 46

VIII. Practica 1.4.4.3 Crear rama y trabajar en modo colaborativo.

- 1) **Introducción:** Los proyectos en la actualidad son cada vez más grandes y es prácticamente imposible que una sola persona haga todo. A continuación, se muestra cómo pueden trabajar al mismo tiempo en el mismo proyecto varios usuarios. Para esto hay que entender el concepto de rama en git, en resumen, esto es una copia o versión alternativa que se genera a partir de la versión principal “master”, para saber en qué rama se encuentra se ejecuta el comando *git branch*,
- 2) **Crear ramas de trabajo:** para crear una rama se ejecuta el comando *git branch rama*, es el mismo comando, pero seguido del nombre de la nueva rama ver Figura 1.4.4. 47, se ve que se creó la rama requisito_1, pero sigue posicionado en master.



```
MINGW64:/c:/Users/juani/Documents/Diplomado IT/Git_Ejemplo
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git log
commit f6c074964400925e7331fec5e35df314e1d2bc51 (HEAD -> master, origin/master)
Author: Juan Pablo <juanice8@hotmail.com>
Date: Thu Sep 6 18:52:44 2018 -0500

    Este es mi primer commit

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git branch
* master

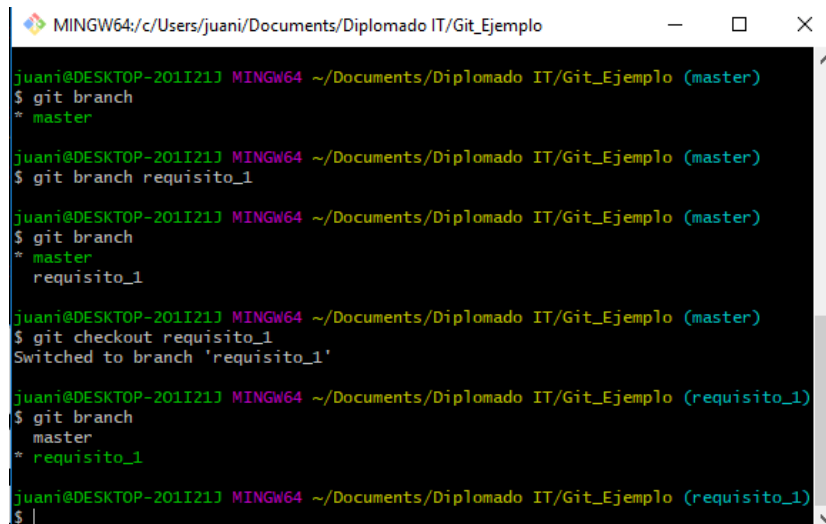
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git branch requisito_1

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git branch
* master
  requisito_1

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ |
```

Figura 1.4.4. 47

- Para cambiar a la nueva rama se ejecuta el comando *git checkout rama*, posteriormente de nuevo el comando *git branch*, para confirmar que está en la rama deseada ver Figura 1.4.4. 48, para simular que dos usuarios están modificando el repositorio, a partir de la rama master generar otra rama repetir pasos de y listarla ahora se mostraran 3 ramas, ver Figura 1.4.4. 49.



```
MINGW64:/c/Users/juani/Documents/Diplomado IT/Git_Ejemplo
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git branch
* master

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git branch requisito_1

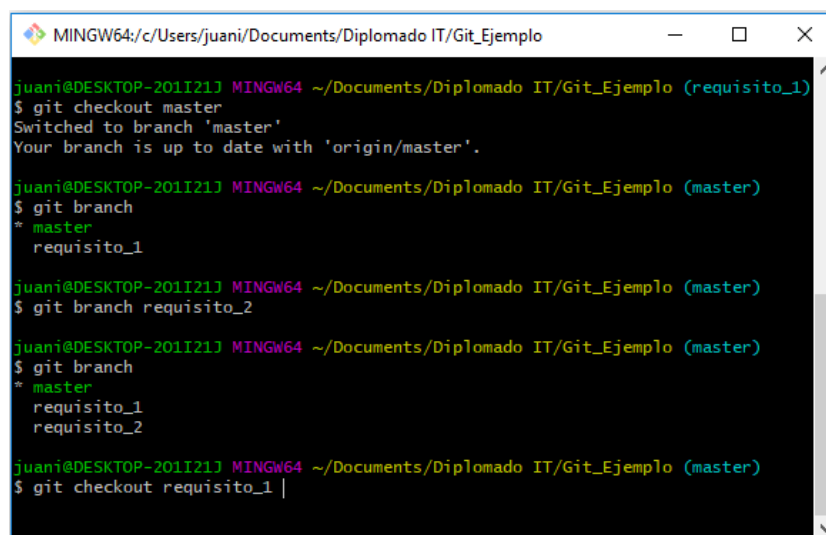
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git branch
* master
  requisito_1

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git checkout requisito_1
Switched to branch 'requisito_1'

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$ git branch
* master
  requisito_1

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$
```

Figura 1.4.4. 48



```
MINGW64:/c/Users/juani/Documents/Diplomado IT/Git_Ejemplo
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git branch
* master
  requisito_1

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git branch requisito_2

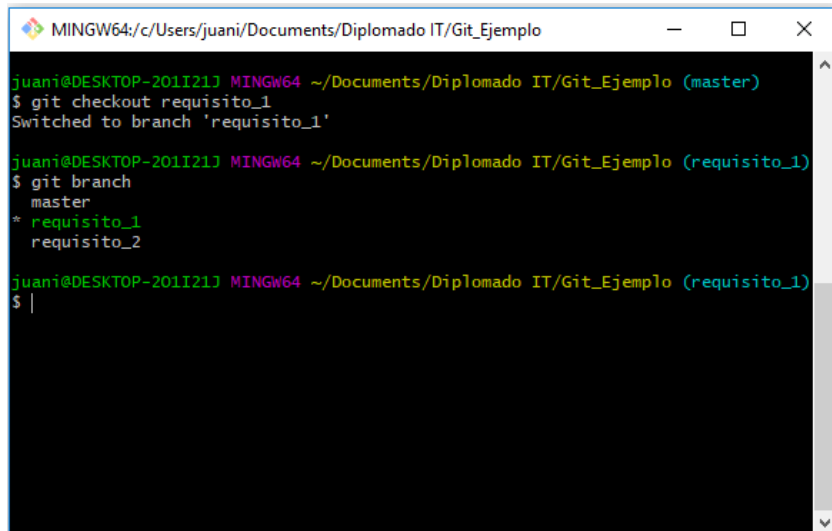
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git branch
* master
  requisito_1
  requisito_2

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git checkout requisito_1 |
```

Figura 1.4.4. 49

3) **Agregar cambios a las ramas:** primero hay que cambiar de rama y agregar los cambios, primero se modificara la rama requisito_1, aquí se agregó la carpeta Doc, que contiene documentos del proyecto y se modificó el archivo main.c, se agregan y se validan los cambios Figura 1.4.4. 51, para la rama requisito_2, solo se modificó el archivo main.c, de igual manera como en requisito_1, se agregan y se validan los cambios con commit.

- En la Figura 1.4.4. 52, se muestra el archivo original “master” y en la Figura 1.4.4. 53, se muestran los cambios que se agregaron a cada rama.

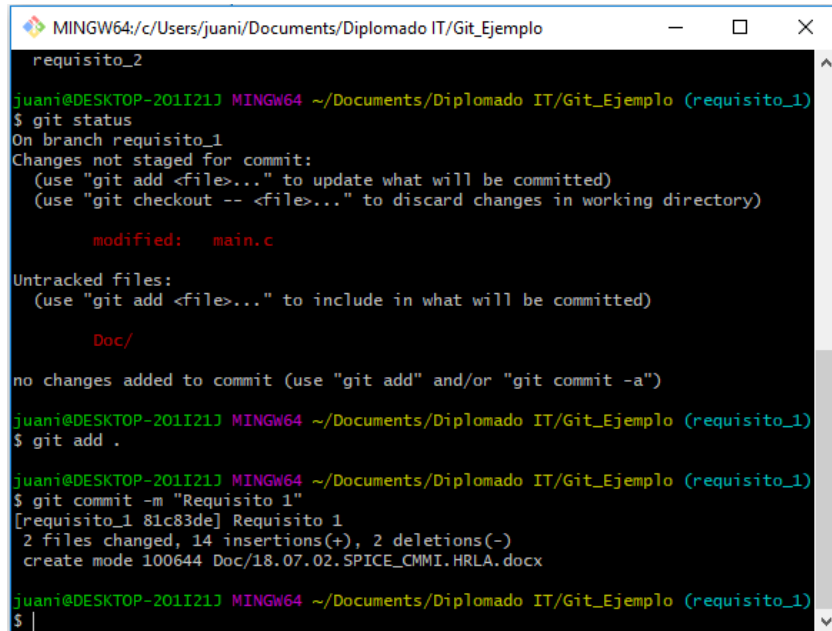


```
MINGW64:/c/Users/juani/Documents/Diplomado IT/Git_Ejemplo
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git checkout requisito_1
Switched to branch 'requisito_1'

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$ git branch
  master
* requisito_1
  requisito_2

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$ |
```

Figura 1.4.4. 50



```
MINGW64:/c/Users/juani/Documents/Diplomado IT/Git_Ejemplo
requisito_2

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$ git status
On branch requisito_1
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:   main.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Doc/

no changes added to commit (use "git add" and/or "git commit -a")

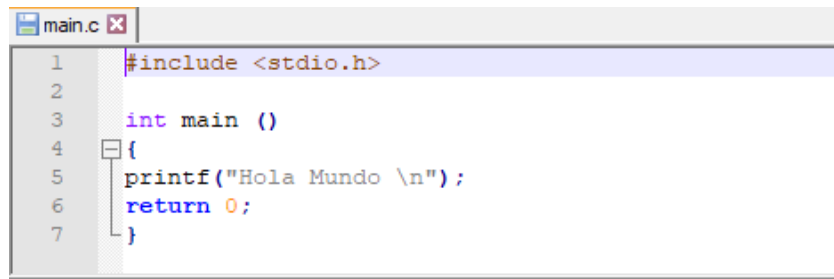
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$ git add .

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$ git commit -m "Requisito 1"
[requisito_1 81c83de] Requisito 1
 2 files changed, 14 insertions(+), 2 deletions(-)
 create mode 100644 Doc/18.07.02.SPICE_CMMI.HRLA.docx

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$ |
```

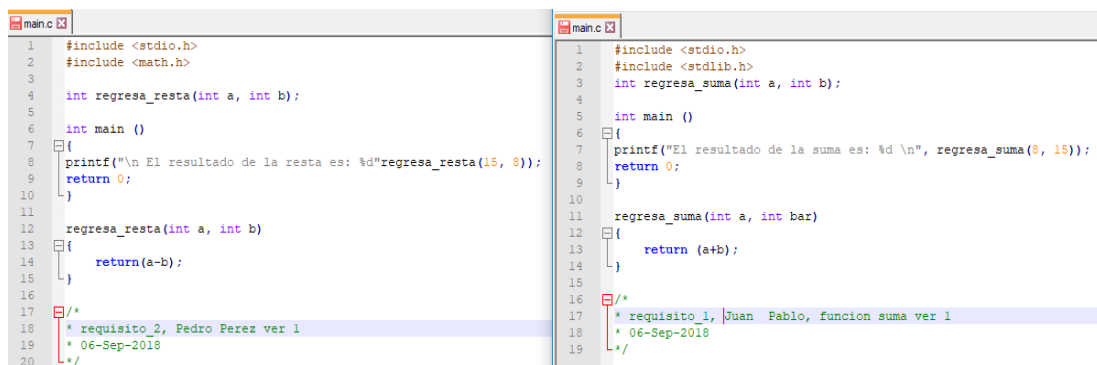
Figura 1.4.4. 51

- En la figura Figura 1.4.4. 53, del lado izquierdo están los cambios en el archivo main.c, de la rama requisito_2, y del lado derecho los cambios de la rama requisito 1, como se puede apreciar al intentar fusionar los archivos en uno solo y meterlo al master, se generarían conflictos pues se modifican las mismas líneas.



```
1 #include <stdio.h>
2
3 int main ()
4 {
5     printf("Hola Mundo \n");
6     return 0;
7 }
```

Figura 1.4.4. 52

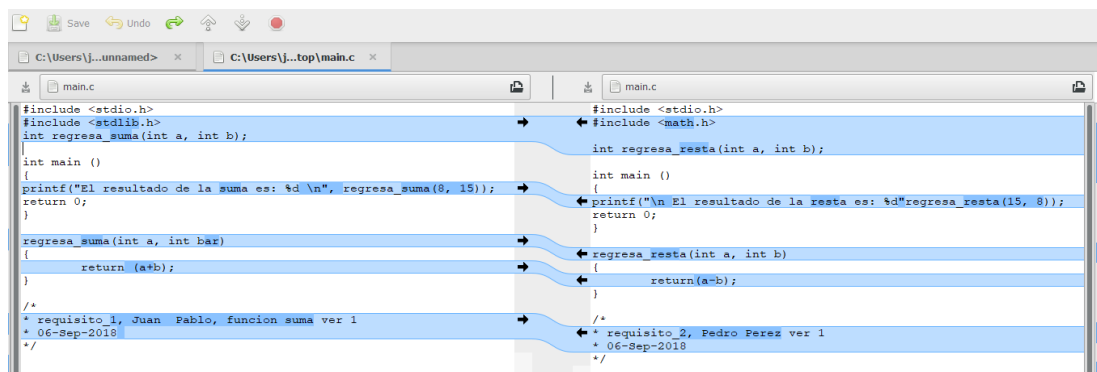


```
Left Pane (requisito_2):
1 #include <stdio.h>
2 #include <math.h>
3
4 int regres_a_resta(int a, int b);
5
6 int main ()
7 {
8     printf("\n El resultado de la resta es: %d",regresa_resta(15, 8));
9     return 0;
10 }
11
12 regres_a_resta(int a, int b)
13 {
14     return(a-b);
15 }
16
17 /*
18  * requisito_2, Pedro Perez ver 1
19  * 06-Sep-2018
20  */

Right Pane (requisito 1):
1 #include <stdio.h>
2 #include <stdlib.h>
3 int regres_a_suma(int a, int b);
4
5 int main ()
6 {
7     printf("El resultado de la suma es: %d \n", regres_a_suma(8, 15));
8     return 0;
9 }
10
11 regres_a_suma(int a, int bar)
12 {
13     return (a+b);
14 }
15
16 /*
17  * requisito_1, Juan Pablo, funcion suma ver 1
18  * 06-Sep-2018
19  */
```

Figura 1.4.4. 53

- Para poder visualizar de manera más clara estas diferencias existen herramientas como Meld, ver Figura 1.4.4. 54, que nos ayudan a ver de una manera mas clara las diferencias y facilita el resolver conflictos.

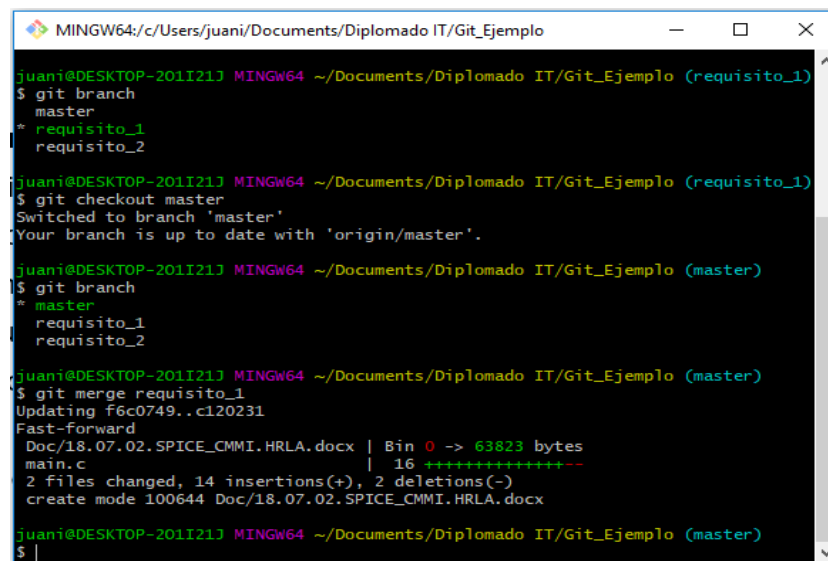


```
Left Pane (requisito_2):
#include <stdio.h>
#include <math.h>
int regres_a_resta(int a, int b);
int main ()
{
    printf("\n El resultado de la resta es: %d",regresa_resta(15, 8));
    return 0;
}
regresa_resta(int a, int bar)
{
    return (a+b);
}
/*
 * requisito_2, Pedro Perez ver 1
 * 06-Sep-2018
 */

Right Pane (requisito 1):
#include <stdio.h>
#include <stdlib.h>
int regres_a_suma(int a, int b);
int main ()
{
    printf("El resultado de la suma es: %d \n", regres_a_suma(8, 15));
    return 0;
}
regresa_resta(int a, int b)
{
    return (a+b);
}
/*
 * requisito_1, Juan Pablo, funcion suma ver 1
 * 06-Sep-2018
 */
```

Figura 1.4.4. 54

- 4) **Hacer merge y Solucionar diferencias:** En este caso el usuario que importe primero los cambios al master, no tendrá conflictos dejando la resolución de estos al segundo usuario, para pasar los cambios al master, primero hay que estar en la rama master, para esto ejecutar *git checkout master*, seguido de el comando, *git merge branch*, este es el nombre del branch de donde provienen los cambios, Figura 1.4.4. 55, con este branch no existen problemas.



```
MINGW64:/c:/Users/juani/Documents/Diplomado IT/Git_Ejemplo
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$ git branch
* master
  requisito_1
  requisito_2

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (requisito_1)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

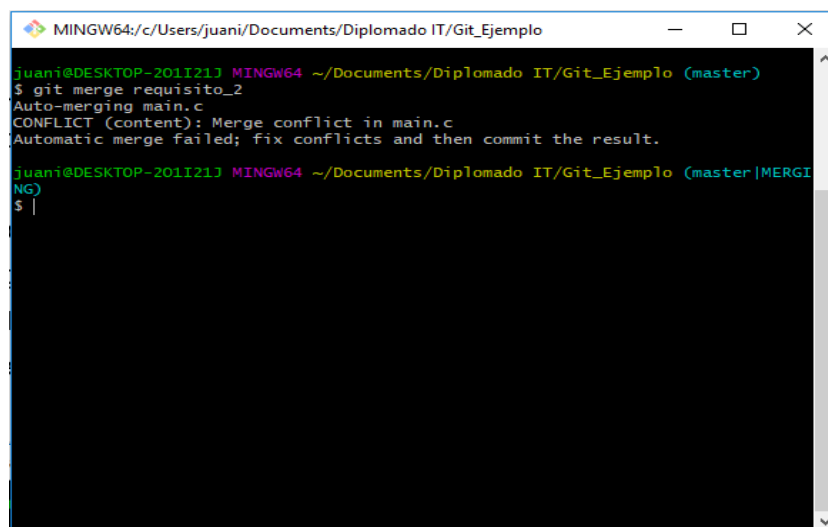
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git branch
* master
  requisito_1
  requisito_2

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git merge requisito_1
Updating f6c0749..c120231
Fast-forward
 Doc/18.07.02.SPICE_CMMI.HRLA.docx | Bin 0 -> 63823 bytes
 main.c                            | 16 ++++++-----
 2 files changed, 14 insertions(+), 2 deletions(-)
 create mode 100644 Doc/18.07.02.SPICE_CMMI.HRLA.docx

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ |
```

Figura 1.4.4. 55

- Al hacer el merge con el segundo archivo, se observa que esto genero un conflicto, como se informa en la consola, ver Figura 1.4.4. 56

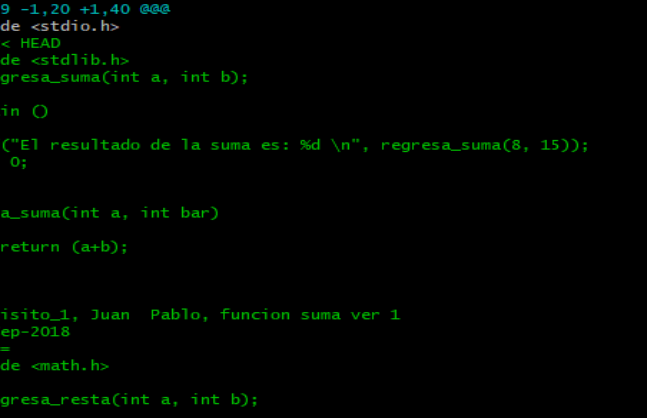


```
MINGW64:/c:/Users/juani/Documents/Diplomado IT/Git_Ejemplo
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git merge requisito_2
Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the result.

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master|MERGI
NG)
$ |
```

Figura 1.4.4. 56

5) Solucionar Diferencias: Para ver las diferencias generadas por este merge con conflictos, usamos *git diff*, en la consola se puede ver todos los cambios generados por el merge Figura 1.4.4. 57



```
MINGW64/c/Users/juani/Documents/Diplomado IT/Git_Ejemplo
+++ b/main.c
@@@ -1,19 -1,20 +1,40 @@@
#include <stdio.h>
+<----- HEAD
+#include <stdlib.h>
+int regresa_suma(int a, int b);
+
+int main ()
+{
+printf("El resultado de la suma es: %d \n", regresa_suma(8, 15));
+return 0;
+}
+
+regresa_suma(int a, int bar)
+{
+    return (a+b);
+}
+
+/*
+* requisito_1, Juan Pablo, funcion suma ver 1
+* 06-Sep-2018
+
+=====
+* #include <math.h>
+*
+* int regresa_resta(int a, int b);
+*
+* int main ()
+* {
+*
+* }
```

Figura 1.4.4. 57

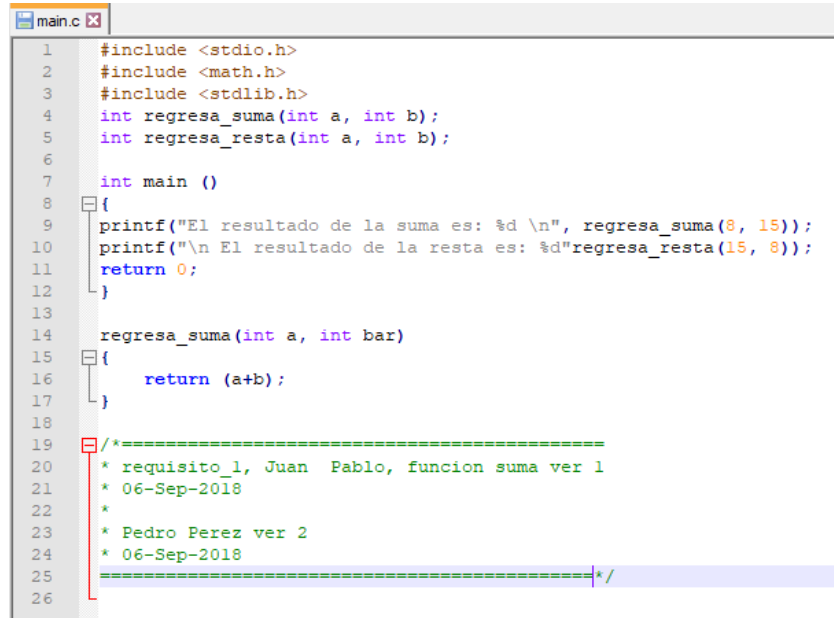
```

1 #include <stdio.h>
2 <<<<<< HEAD
3 #include <stdlib.h>
4 int regresa_suma(int a, int b);
5
6 int main ()
7 {
8     printf("El resultado de la suma es: %d \n", regresa_suma(8, 15));
9     return 0;
10 }
11
12 regresa_suma(int a, int bar)
13 {
14     return (a+b);
15 }
16
17 /*
18  * requisito_1, Juan Pablo, funcion suma ver 1
19  * 06-Sep-2018
20  * =====
21  * #include <math.h>
22
23  * int regresa_resta(int a, int b);
24
25  * int main ()
26  * {
27  *     printf("\n El resultado de la resta es: %d",regresa_resta(15, 8));
28  *     return 0;
29  * }
30
31  * regresa_resta(int a, int b)
32  * {
33  *     return(a-b);
34  * }
35
36  * /*
37  *  * Pedro Perez ver 1
38  *  * 06-Sep-2018

```

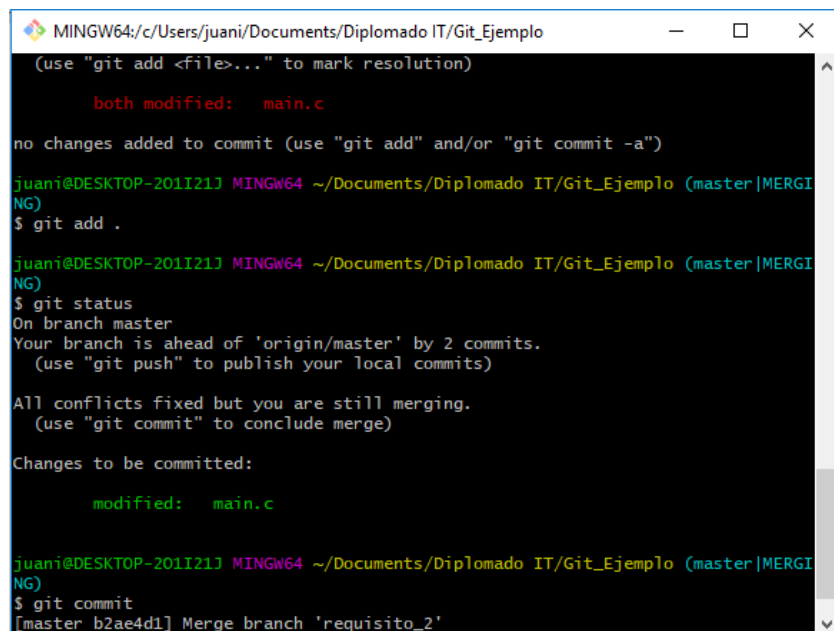
Figura 1.4.4. 58

- Despues estos cambios se van agregando o descartando según se necesite, para al final incluir los cambios de ambas ramas, como se muestra en la figura Figura 1.4.4. 59., siguiendo con el proceso se agregan los archivos y se validan haciendo commit, ver Figura 1.4.4. 60



```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4  int regresasuma(int a, int b);
5  int regresaresta(int a, int b);
6
7  int main ()
8  {
9      printf("El resultado de la suma es: %d \n", regresasuma(8, 15));
10     printf("\n El resultado de la resta es: %d", regresaresta(15, 8));
11     return 0;
12 }
13
14 regresasuma(int a, int bar)
15 {
16     return (a+b);
17 }
18
19 /*=====
20  * requisito_1, Juan Pablo, funcion suma ver 1
21  * 06-Sep-2018
22  *
23  * Pedro Perez ver 2
24  * 06-Sep-2018
25  *=====
26  */
```

Figura 1.4.4. 59



```
MINGW64:/c:/Users/juani/Documents/Diplomado IT/Git_Ejemplo
(use "git add <file>..." to mark resolution)

both modified:   main.c

no changes added to commit (use "git add" and/or "git commit -a")

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master|MERGI
NG)
$ git add .

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master|MERGI
NG)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
(use "git push" to publish your local commits)

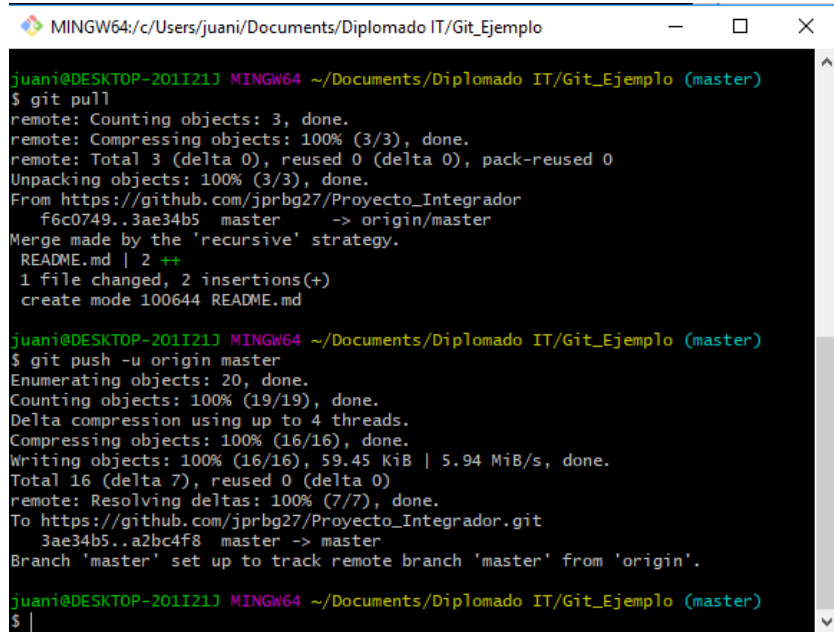
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:
  modified:   main.c

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master|MERGI
NG)
$ git commit
[master b2ae4d1] Merge branch 'requisito_2'
```

Figura 1.4.4. 60

- Por ultimo se necesita hacer *git pull*, para actualizar el repositorio principal con la información de los branches creados, finalmente se ejecuta *git push -u origen master*, para subir al servidor los cambios anteriores ver, Figura 1.4.4. 61, luego refrescar la página del servidor y verificar los cambios, Figura 1.4.4. 62, fin.

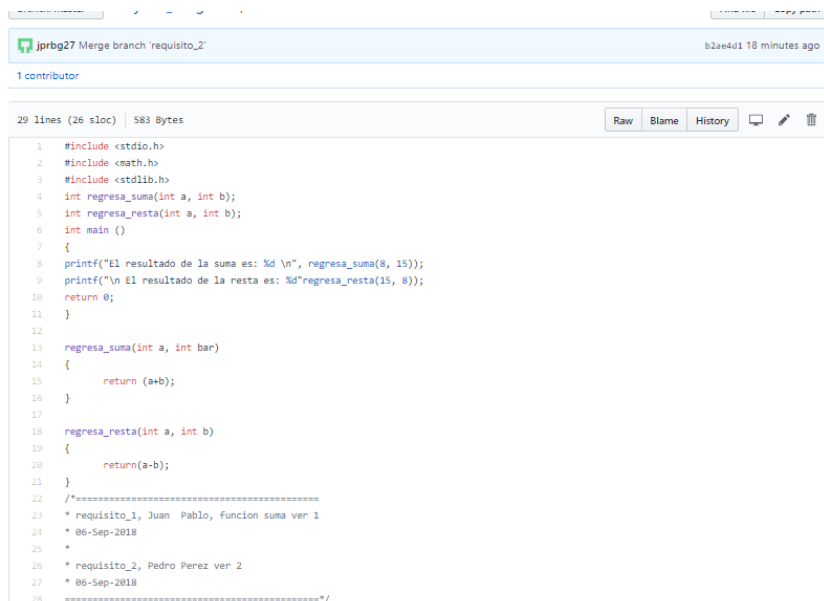


```
MINGW64~/c/Users/juani/Documents/Diplomado IT/Git_Ejemplo
juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/jprbg27/Proyecto_Integrador
f6c0749..3ae34b5 master -> origin/master
Merge made by the 'recursive' strategy.
 README.md | 2 ++
1 file changed, 2 insertions(+)
create mode 100644 README.md

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ git push -u origin master
Enumerating objects: 20, done.
Counting objects: 100% (19/19), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (16/16), 59.45 KiB | 5.94 MiB/s, done.
Total 16 (delta 7), reused 0 (delta 0)
remote: Resolving deltas: 100% (7/7), done.
To https://github.com/jprbg27/Proyecto_Integrador.git
3ae34b5..a2bc4f8 master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

juani@DESKTOP-201I21J MINGW64 ~/Documents/Diplomado IT/Git_Ejemplo (master)
$ |
```

Figura 1.4.4. 61



```

1 #include <stdio.h>
2 #include <math.h>
3 #include <stdlib.h>
4 int regresa_suma(int a, int b);
5 int regresa_resta(int a, int b);
6 int main ()
7 {
8     printf("El resultado de la suma es: %d \n", regresa_suma(8, 15));
9     printf("\n El resultado de la resta es: %d", regresa_resta(15, 8));
10    return 0;
11 }
12
13 regresa_suma(int a, int bar)
14 {
15     return (a+b);
16 }
17
18 regresa_resta(int a, int b)
19 {
20     return(a-b);
21 }
22
23 /*=====
24  * requisito_1, Juan Pablo, Funcion suma ver 1
25  * 06-Sep-2018
26  *
27  * requisito_2, Pedro Perez ver 2
28  * 06-Sep-2018
29  * =====*/
```

Figura 1.4.4. 62

IX. Practica 1.4.4.4 Trabajo en pareja

- ☐ Un integrante debe, generar un repositorio local vacío
- ☐ El mismo debe subir el repositorio a github
- ☐ El otro integrante debe clonar este repositorio
- ☐ Cada uno debe agregar un archivo, con una versión inicial al master
- ☐ Cada uno debe generar una rama de trabajo
- ☐ Modificaran el mismo archivo generando conflictos
- ☐ Hacer merge y resolver los conflictos
- ☐ Subir a github

Bibliography

Apache Software Foundation. (2017). Obtenido de <https://subversion.apache.org/>

Bazaar. (2016). Obtenido de <http://bazaar.canonical.com/en/>

Codice Software. (2018). Obtenido de <https://www.plasticscm.com/>

CVS - Concurrent Versions System. (2005-2006). Obtenido de <https://www.nongnu.org/cvs/>

Git-fast version control. (2018). Obtenido de <https://git-scm.com/>

Github, S. (s.f.). Obtenido de <https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf>

Mercurial. (2017). Obtenido de <https://www.mercurial-scm.org/>

monotone. (2018). Obtenido de <https://www.monotone.ca/>

PVCS. (s.f.). Obtenido de <https://en.wikipedia.org/wiki/PVCS>

Rational, S. (s.f.). Obtenido de <https://www.ibm.com/us-en/marketplace/rational-synergy>

Wikipedia, C. (s.f.). Obtenido de https://en.wikipedia.org/wiki/Comparison_of_version_control_software