C Extras

Intro:

Perhaps you are wondering what is meant by “C Extras”. Perhaps you simply opened this doc out of curiosity, considering that the title is perhaps not the most aptly named. To clarify, the purpose of this document is to cover extra programs and features which are related to C programming, but which I felt were inappropriate for my C Programming Language document. Specifically, I’d like to cover Make and Makefiles, The GNU debugger (GDB), version control (with git, although I may touch on SVN as well), creating Markdown files, and autogenerating documentation with Doxygen. I felt that these lay outside of the scope of my C programming doc, but that they were still important topics to cover, hence the existence of this document.

Make/Makefile

Intro:

This section will be dedicated to GNU Makefiles. Most programming languages have some sort of tooling for build automation. The program most commonly used to automate builds in C is known as make. make is a command in Linux created by Richard Stallman and maintained by the FSF. It searches for a special file known as a Makefile which contains instructions on how to build our executables. Something that makes make unique is that it utilizes timestamps to detect which portions of the program need to be rebuilt, and which don’t. The GNU organization already has great documentation on make which can be found here [https://www.gnu.org/software/make/manual/make.html#toc-Overview-of-make](https://www.gnu.org/software/make/manual/make.html" \l "toc-Overview-of-make). While I encourage you to read through this documentation, it can be a bit daunting if you’re just writing your first Makefile. My aim is to give you a general overview of how we can use make; your experience in the field will fill in the missing gaps as you use make more and more.   
  
Rules:

The idea behind Makefiles is that we write out a series of “rules” which are used to ensure certain conditions are met as we build. Some rules are pre-requisites for other rules. This creates a sort of hierarchy which appears complex on the surface but is really just a sequence of simple rules tied together to form a finished product. Rules follow a special formula so that they can be recognized by make. The general layout of a rule is as follows:

target ... : prerequisites ...

recipe

...

...

Let’s break down these three peices that make up our rule:

**target:** The target is the desired output of processing the rule. In most cases, the target is a file which is created by running certain command line utilities to obtain the desired effect, though the target can also just be a nametag used to specify the rule that we want make to run.  
  
**prerequisites:** The prerequisites are conditions that must be met in order for our rule to be executed. If the prerequisites are not met, the rule cannot be satisfied. Prerequisites are also usually files which we expect to exist in order for our rule to be executed, but may also be other targets.

**recipe:** The recipe is the actual command that we want make to run. There is nothing special about what commands can be put in our recipe; make will execute whatever we add in our recipe in a shell as if we were actually entering those commands into our terminal.

Variables:

As you may expect, make is capable of keeping track of variables. Variables in make are very similar to variables in shell scripts. They are essentially expanded at runtime so that if we assign an entire command to a variable, that command will be executed on the spot. Variables are generally always upper-case as a convention, and they are assigned with the assignment operator (=). In order to expand a variable in make, we use the special operator ‘$’, followed by the variable name enclosed in brackets eg. $(VAR). Again, this is very similar to shell scripts.

Although we are able to define variables however we want in make, there are certain convensions which nearly all programs abide by. Specifically, there are certain variable names that ought to be used for readability. Here is a list of some conventional variable names:

**CC:** In case you needed a refresher, cc stands for c-compiler. In Unix, the compiler was simply called cc, but as you know, Linux was created for the GNU project, and with GNU came the GCC (GNU C-Compiler). Therefore, you would set CC = gcc on a Linux machine.

**CFLAGS:** The CFLAGS variable is used for flags that we wish to add to our compiler. These include the flags that we covered previously, such as -O, -g, -Wflags, etc.

**LDFLAGS:** In Unix, ld stood for “loader”, which at the time, was synonymous with “linker” ie. they were one in the same. Nowadays, the loader is a separate program from the linker, so this name is less relevant. None-the-less, LDFLAGS is used to describe flags that we want to pass to our linker which is usually just whatever libraries we want to link with the -l option.

**OBJS:** A list of object files that are a dependency for the final executable.

**BINS:** The BINS variable is typically used as a list of binaries that are to be made by the “all” target. We will look at a few standard target names momentarily.

**LIBS:** The LIBS variable usually contains the path(s) to a alternative directories that contain important libraries for creating our program that are not already in our system’s PATH variable. These directories can be searched with the -L option in our CFLAGS eg. CFLAGS = -L $(LIBS) where LIBS = /project/lib:/project/other/lib

The Default Goal:

If you run make without specifying a target, make searches for the first target who’s name does not start with ‘.’. This is called the default goal. Usually, we call the default goal ‘all’, which means run all other rules in the Makefile. The behavior of the default goal can be altered using an option when running make from the command line, or by overriding the .DEFAULT\_GOAL special variable. If the default goal depends on other targets, then those get ran first.  
  
Implicit Rules:

PHONY Targets:

Variable Assignment:

Make parses the Makefile in two separate runs (known as first phase and second phase). The first phase is when make reads all of the included Makefiles and internalizes all the variables, their values, implicit/explicit rules, and builds a dependency graph of all the targets and their prerequisites. The second phase uses all of this interalized data to determine which targets need to be updated and run and in what order.

One of the most confusing aspects of make is how it expands variables. There are different assignment operators which determine how the variable will be expanded. There are essentially three options for expansion. Either the variable expansion is immediate (the variable is evalutated during the first phase), deferred (the expansion is either delayed until the variable is referenced by an immediate variable, or until the second phase if it does not get referenced), or it is deferred or immediate (deferred under certain circumstances or immediate under others). Here is a list of the operators:

immediate = deferred

immediate ?= deferred

immediate := immediate

immediate ::= immediate

immediate += deferred or immediate

immediate != immediate

These are each unique, but their use-case is not inherently obvious. The = operator is the most basic. It simply defers the expansion of the variable. The ?= operator is the same as =, but only assigns the value to the variable if the variable does not already contain a value (the = operator overwrites previously assigned values). The := and ::= (simple variable operators) are expanded before variable assignment even occurs. If a variable is assigned with := or ::= and then later reassigned with +=, then it will be immediately expanded. Otherwise, if we only assign a variable with the += operator without previously assigning it with a simple operator, it will be deferred. The append operator (+=) will append a value to a variable. Last but not least, the shell operator (!=) is immediately evaluated and then passed to the shell. The result of the shell is stored in the variable, and the variable becomes a simple variable. The man page for make has additional info on these operators if you need additional guidance.

Version Control With Git

What is Version Control?:

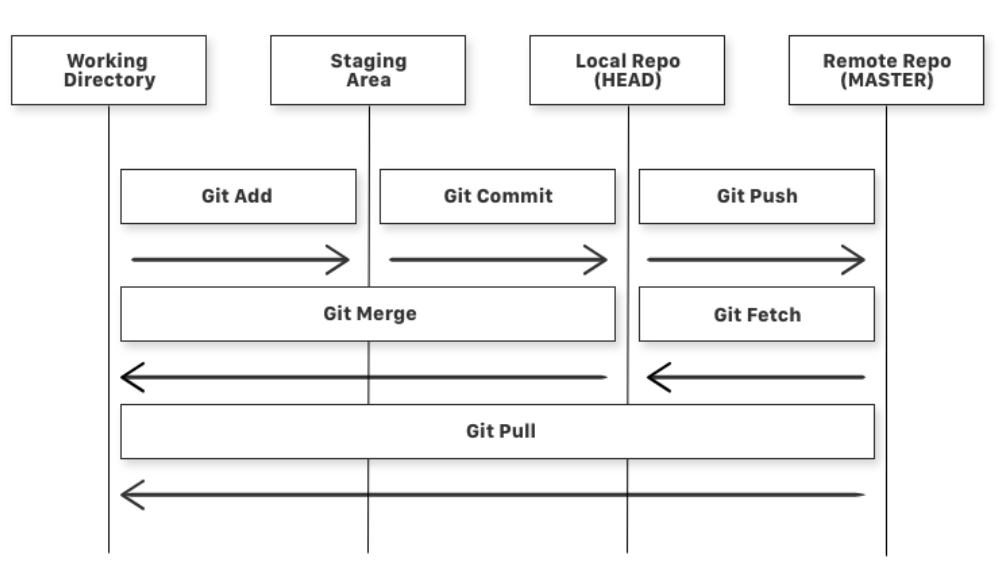
If you are unfamiliar with version control, then buckle up, because we have a lot to discuss! Version control, sometimes called source control, is the practice of tracking and managing changes to code. There are various version control tools which monitor and record changes to our code. This careful monitoring allows us to do very neat things, such as go back in time to a previous commit, create forks of our code into separate projects, or branch our code to work on a new feature without affecting our other branches. Arguably, the most well-known version control systems are Revision Control System (RCS) managed by the GNU project, Subversion (SVN), and Git (created by Linus Torvalds, the creator of the Linux kernel). The order in which I listed those three happen to be the order in which they were created timeline-wise. Git is the most recent, and well known, which is why we will be using it. As a developer, I’m sure you have at least heard of GitHub. Perhaps erroneously you assumed that GitHub and Git were the same thing. For this reason, we will be splitting up this section into 2 parts. The first part will focus on local source control using Git (the bulk of the material), and then later we will look into GitHub, which just happens to be one of many upstream servers for the front-end stuff. Git also encourages collaboration on projects since it allows us to merge changes made by our peers.

What Makes Git Special?:

Perhaps you’re wondering about some of the reasons that Git took off in popularity over its predecessors. Well one nice thing about it is that it is a Distributed Version Control System. This means that Git does not rely on a central server to store all the versions of a project’s files. Instead, Git will store the metadata locally in a hidden folder (called .git). When the code is cloned by someone else, they are downloading the source files, and also that .git folder containing all past changes that have been committed.

Understanding the Git Workflow:

It is imperative that you grasp this next concept, because it will ultimately influence the way that we interact with Git. I am totally plagiarizing these notes from freeCodeCamp’s website, including the following image, so check out their site as well. Here we see the four fundamental elements of the Git workflow (Working Directory, Staging Area, Local Repository, and Remote Repository):



Let’s quickly give a briefing of each. The working directory is your project directory. We run the command git init to initialize our project directory. This creates the .git folder I mentioned earlier and prepares it to be tracked by git. The staging area is invisible to us (well it’s actually a file that we can open, but we don’t typically access it directly). It let’s Git know which files we actually want to track. Running git status gives us an overview of which files are and aren’t staged. One other thing to note about the staging area is that files which belong to it will have their changes recorded. If a file is being tracked (i.e. it belongs to the staging area) and we modify it, it will immediately become committable, even if it had already been committed. In the next “phase” (the HEAD) we have files which have been committed. When files are committed, we essentially create a save state or backup point that we can revert to if needed. When a file has been committed, it’s state is actively preserved, unlike files which are staged, which have modifications tracked, but not preserved. Finally, we have MASTER, which is the remote repository (repo for short). This could be GitHub, GitLab, Savannah, etc. From the remote repo, others can clone (download) our code to their own local repo.

What’s very nice about the graph above is that it shows us which git subcommands correspond to which action. Here is a summary of each:

**git add:** This command adds files to the staging area. Typically you need only do git add \* to add all files.

**git commit:** Pretty self-explanatory, the git commit subcommand takes all of our staged/tracked files and commits them. This command is almost always used in conjunction with the -m option which specifies the commit message (all commits require a commit message i.e. a summary of why the commit is being created).

**git push:** Git push pushes our committed files to the remote repo. Usually you would do something like git push -u origin master, where the -u means upstream i.e. MASTER (in our case the master’s name is origin), followed by the name of our local branch (in this case, master). Kind of confusing that our local branch is named master and not the remote repo, but that’s just how it is…

**git fetch:** This one often confuses newcomers. Git fetch just pulls files that exist in the remote repo to the local repo. If files exist on the remote repo that don’t exist on the local repo, you will not be able to push your changes until the local repo is up to date with the remote. You can either do git fetch, or git --force push to overwrite the remote repo with the contents of the local one.

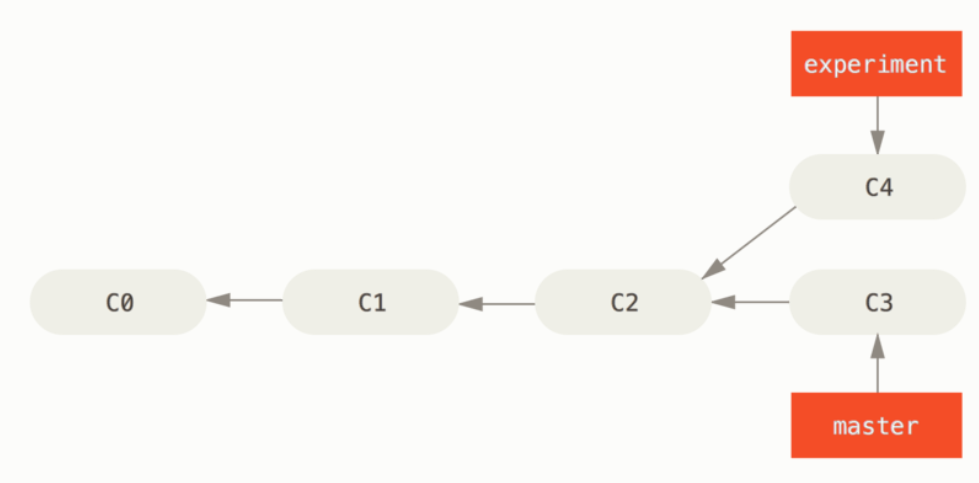
**git merge:** This one is similar to git fetch, except that git merge pulls files into our working directory. Git merge will fail if a file from the local repo (HEAD) has changed that do not exist in the working directory.

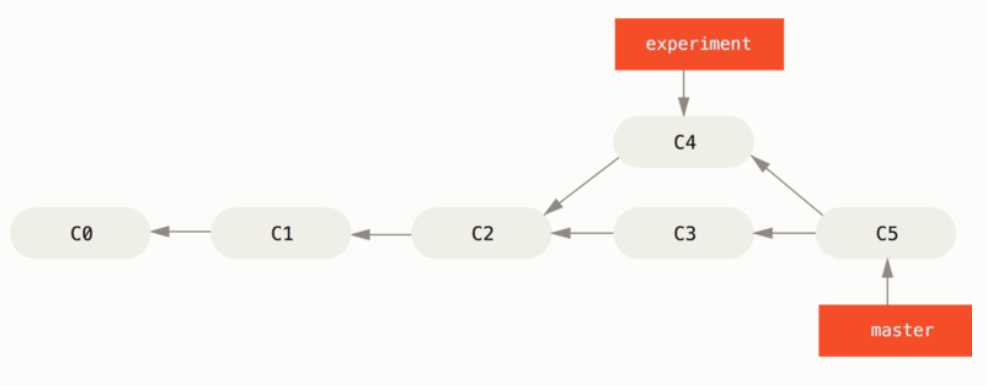
**git pull:** Git pull is equivalent to doing git fetch, followed by git pull. This will essentially pull files that exist on the remote repo into both HEAD and the working directory.

Git Branches:

An important feature of Git is its ability to create branches. Recall that we can create commits which are just snapshots of a period in time where our file contents are preserved. A Git branch is essentially a pointer to one of these commits which can be moved to other commits. The default branch is called master. You can reassure yourself of this fact by entering the subcommand git branch, which will display a list of the created branches with an asterisk next to the current branch. This should look like \*master by default. As we create new commits, the HEAD keeps up to date with the latest one. Once again, this can be confirmed by running git log --oneline. Running this command will show you which branch HEAD is pointing to, as well as which commit it is currently on. For example, it might look something like the following: 4960843 (HEAD -> master) Initial commit. Here we see the commit hash, 4960843. This is a unique identifier given to each commit. We see that HEAD indeed points to the master branch, as that is the one we are currently on. Finally, we see the commit message created using git commit.

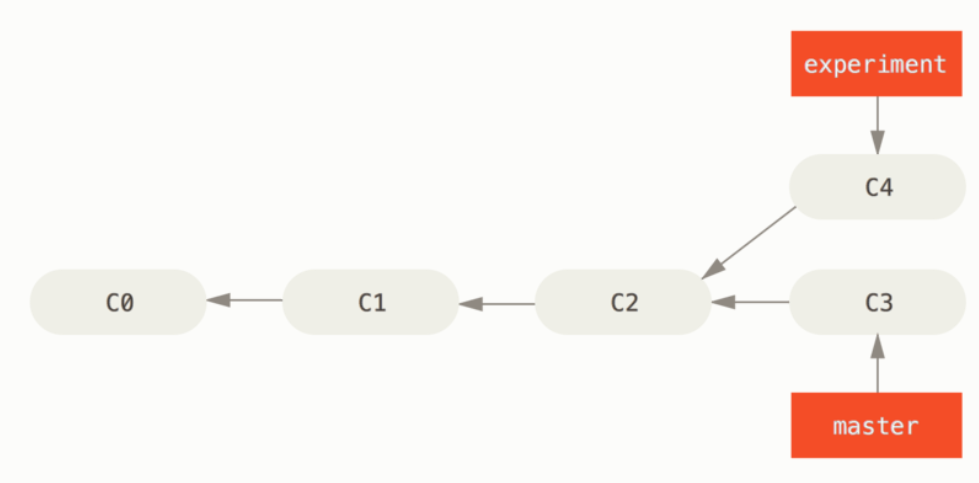
So when should we create a new branch? Well, typically you create a new branch when you want to add a new feature to your program that you don’t want to affect your main branch. Typically, the main branch is referred to as the “mainline” branch, and then you may have a “dev” branch for development, and possibly branches for specific features or versions that are known to be working well. In order to create a new branch, we use the git branch subcommand once again, except this time, supply it with the name of the branch you want to create. For example, git branch new. Once this is created, running git branch by itself will display that we now have a branch called master, and a branch called new. Note that the asterisk is still on master, meaning that HEAD is still looking at master. In order to switch over to our new branch, we use the checkout subcommand e.g. git checkout new. Now, when we run git log --oneline, we see a bit of a change. The log now appears as 4960843 (HEAD -> new, master) Initial commit. This indicates that HEAD points to new, and then the comma followed by master indicates that new was branched off from master. From our new branch, we can continue development and rest assured that master will remain at the commit that we branched from.   
  
 Now let’s say that we finish developing a feature in our feature branch, and we now want to get the mainline branch up to date with the feature branch so that it will implement the changes. There are two ways that we can do this: git merge, and git rebase. Git merge is the easier of the two. It will create an intermediary commit which both the mainline and feature branch can agree to join up at. Here are some illustrations:





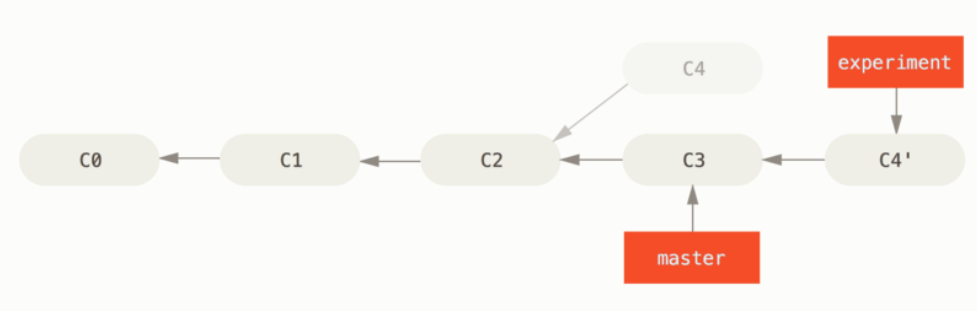
In the first image, we created an experiment branch from commit 2 (C2). We then did some work on both the experiment branch, and the master branch, and created another commit on both (C3 and C4). In order to get these up to date, we create a new intermediary commit C5 using git merge.

The other method of getting these up to date is using git rebase. Rebase functions a bit differently. Rather than creating a new commit which joins the two branches, we take all of the changes that were made on one of the branches and replay them on another branch as if they had occurred on that branch in the first place. Let’s look at how this works using the same starting illustration.



$ git checkout experiment

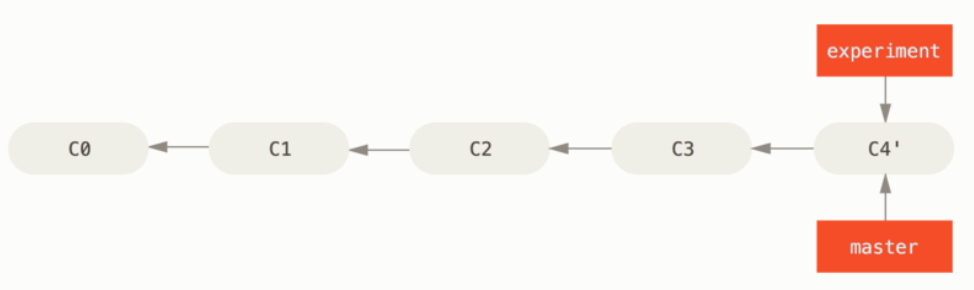
$ git rebase master



Starting from the first image, we have the same scenario. We then checkout to the experiment branch, so HEAD will be pointing to C4. Then we rebase master. What this does is select master as a starting point for replaying the changes made to experiment after it diverged from master. Note though, that master is still behind. We can fix this by using what’s called a fast-forward merge:

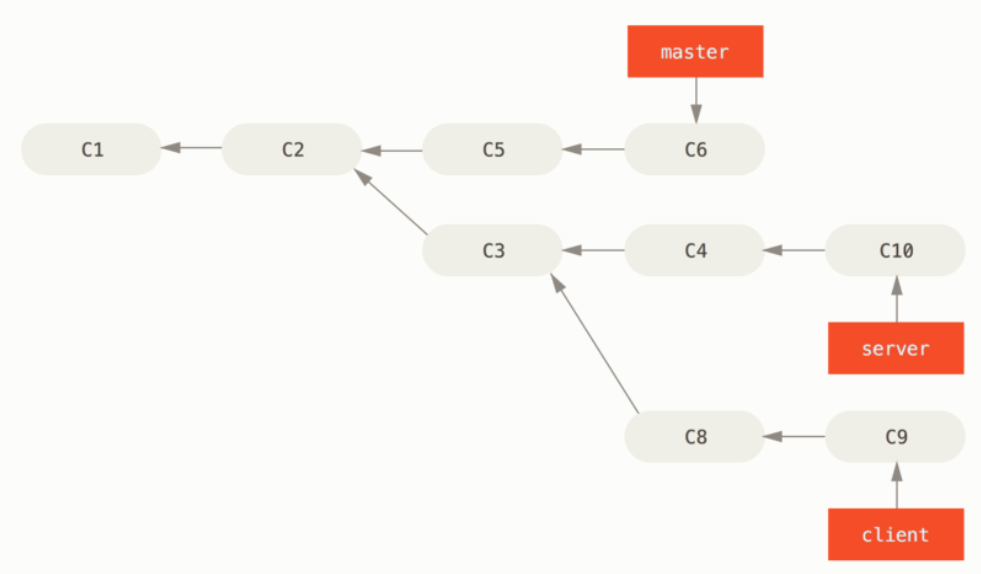
$ git checkout master

$ git merge experiment



This puts us up to date with experiment. I’ve heard the argument that rebase is evil because it deletes history and modifies it to look as if the changes actually occurred in that order even though they didn’t. On the other hand, I’ve heard the argument that rebase makes for a cleaner history and is therefore easier to read. Personally, I haven’t used git enough to have an opinion on this, but I just wanted to bring those two sides to light.

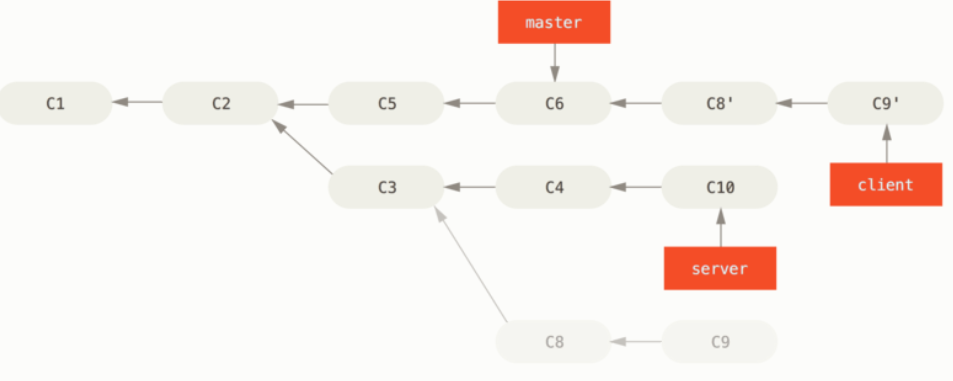
Here is another more complicated example (I know complicated is probably the last thing you want right now, but it is good to look at more realistic examples).



Here we have a more complicated commit history with three active branches. I’m taking this example from Git’s official documentation, so I’ll explain the scenario that they’ve invented. You, the user, have created a new branch for a server application. You also created a branch from that for the client code since they are similar. Now you want to merge the client code into mainline for release, but you want to hold off on the server code until it’s tested further. We can take the changes made for the client code (C8 and C9) and replay them on master using the –onto option of git rebase e.g.

$ git rebase --onto master server client

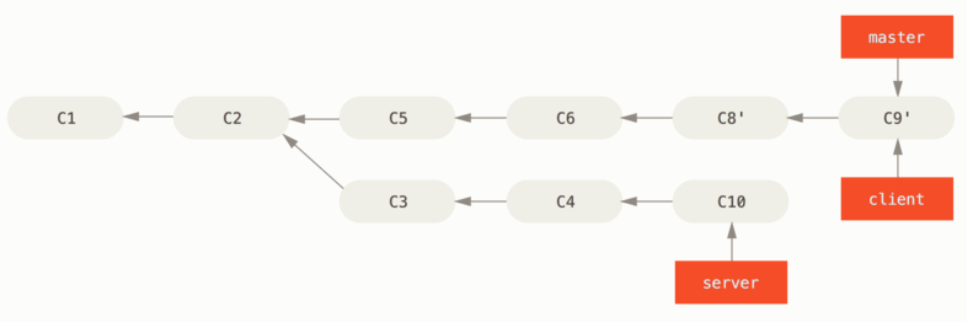
This takes the client branch and pastes it onto the end of master like so:



The branch that comes after the –onto option (master in our case) is the starting point at which we append the commits. We specify both server and client to tell Git to figure out which changes occurred on client since it diverged from server (i.e. after we created the client branch). Once again we can fast-forward merge master to get it up to date with client like so:

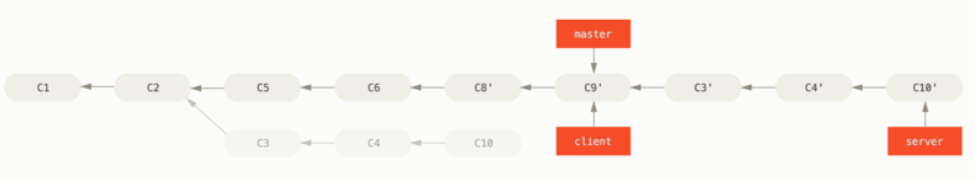
$ git checkout master

$ git merge client



Now in the example, you decide that you’re ready to merge to server. We can rebase server onto the master branch like so:

$ git rebase master server



Once again, this selects master as the start node and then replays changes made to server since it was diverged beginning at master. And once again we can run the following to fast-forward merge master:

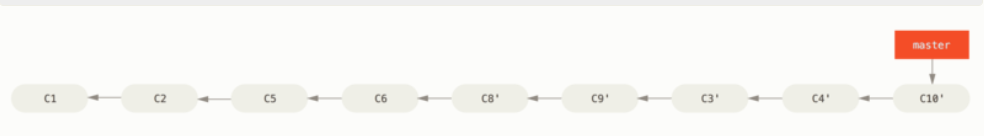
$ git checkout master

$ git merge server

Not only that though, but since we no longer need the client or server branches, we can do the following to delete them:

$ git branch -d client

$ git branch -d server



As mentioned in the Git docs, there is one rule which you must follow when rebasing code: “Do not rebase commits that exist outside your repository and that people may have based work on.” For example, if a co-worker clones the repo and is working on a feature branch, and you rebase the feature branch, your co-worker will have to re-merge their local work into the updated branch structure, which will get messy.

Reverting to Previous Commits:

Sometimes you wish to undo changes to the repos commit history. Subcommands like git checkout and git reset move the HEAD and branch ref. pointers (pointers to existing branches) to a specific commit. git revert is special though, because it does not move ref. pointers to the commit. Instead, it performs the inverse changes made to get to the current commit resulting in a new “revert commit”. So, when we use git revert, it is sort of like a merge where we create an intermediary revert commit that joins HEAD with the commit we want to revert to, this way, we do not delete anything. This is in opposition to git reset which actually destroys subsequent commits in order to go back to a previous one.

Forking Repositories:

Doxygen

What is Doxygen?:

If you are familiar with Javadoc, then you will have a good idea of what Doxygen is. If not, Doxygen is a static site generator that creates HTML/CSS files to neatly display your code’s internal documentation. It does this by reading tags within our code comments. Doxygen is a very useful tool for large projects (e.g. a library) so that developers can navigate the code base and understand what each function does, what parameters it accepts, etc. The tool is relatively easy to use once you’ve learned the basics, which is my aim to teach right now! Be sure to read the official documentation here (created using Doxygen!): <https://doxygen.nl/index.html>

Getting Started With Doxygen:

As always, I am extremely oppressive towards Windows users, so if that includes you then tough luck following along. On Linux, all you must do is install the doxygen package using your native package manager. Once installed, I like to create a directory in my project (usually something like $(project\_root)/res/doc/). Inside of this directory, we want to run the doxygen command with the -g option to generate a default config file. The default name of the file that gets generated is Doxyfile. If, for whatever reason, you don’t like the default name, you can append and alternate name after the -g flag.

Editing the Doxyfile:

Once we’ve generated the default config, we will want to change a few options. And there are many, many, many options. Luckily, we won’t usually care about most of them. Here are a few of the main options and what they do:

**DOXYFILE\_ENCODING:** This sets the encoding format of the Doxyfile. We can pretty much always leave this as UTF-8.  
**PROJECT\_NAME:** The name of your project (surrounded by double quotes).

**PROJECT\_NUMBER:** This represents the revision number of your project. Typically useful for older version control systems such as SVN.

**PROJECT\_BRIEF:** A one line summary of the project.

**PROJECT\_LOGO:** You may optionally include a icon file for your project with max height of 55px and max width of 200px.

**OUTPUT\_DIRECTORY:** This is used if you want to change the location of the output directory for the files which will be generated by doxygen. It can be a relative path or absolute path. If left blank, doxygen assumes the current directory.

**OUTPUT\_LANGUAGE:** This just tells doxygen which encoding scheme to use in the HTML header. For example, it may need a different encoding format for French since it uses a different character set than UTF-8.

**JAVADOC\_AUTOBRIEF:** This one is important if we plan on using Javadoc formatting (my personal preference). The autobrief causes doxygen to interpret the first line of a Javadoc-style comment as the briefing. If this is set to NO (the default), then the briefing must manually be set using the @brief tag. I prefer the latter, but it’s a matter of preference.

**JAVADOC\_BANNER:** Once again, this is only used if you’re using Javadoc-style comments. Setting this to YES will cause doxygen to interpret lines such as /\*\*\*\*\*\*\*\*\* as a Javadoc-style comment “banner”. Otherwise, if set to NO (the default), doxygen just interprets this as the start of a comment.

**QT\_AUTOBRIEF:** This is similar to JAVADOC\_AUTOBRIEF but for QT-style comments. If this is set to YES, then doxygen interprets comments starting with //! or /// as a brief description. I do not personally like this since C++ style comments are only allowed as of C11.

**OPTIMIZE\_\*:** The options beginning with OPTIMIZE\_ are used to generate comments more efficiently for a target language. The options here are for C, Java, FORTRAN, and VHDL. I usually set this to YES since my code is typically written in C, but I doubt it makes a very big difference.

**MARKDOWN\_SUPPORT:** This just lets doxygen use Markdown for generating certain static page elements. I suggest you leave this as YES.

**INPUT:** One of the most important options and it’s way down on line 867! This specifies the files that should be checked for doxygen comments. It can either be a list of files delimited by spaces, or it can be one absolute/relative path to the project directory. Since I like to put my Doxyfile in $(project\_root)/res/doc/ I change INPUT = ../../$(project\_root).

There are obviously many other options which can be set. Doxygen does a pretty good job of giving explanations of what each one does so you can skim through it more thoroughly if there is something specific that you want to change. I just wanted to cover the basic options to give an idea of what I might change.

Doxygen Tags:

Doxygen provides many tags, which are simply called special commands in the official documentation. These are sort of the typical things that you would expect to find in a Javadoc comment. They can begin with an escape character ‘\’ or with an ‘@’ symbol. I prefer the latter because I like to stick with Javadoc-style, but just pick one and be consistent. Many of the tags are (in my opinion) rather useless. I will only cover the ones that I think are useful but be sure to check out the official docs if you’re looking for something specific.

**@class <name> [header file] [header name]:** Used for C++ or Java code when there is a class. You must provide the class name after @class, but you can optionally also specify the header file that it belongs to, or just the header name as it would be #included in your C++ file. Here is an example of using both the header file and header name:

/\*\*

\* @class Main main.h “includes/main.h”

\*/

**@enum <name>:** Pretty straightforward – gives a description of the enum type/class.

**@example[‘lineno’] <file-name>:** This one is definitely niche, but I could see it being used. It essentially points to a file which provides example code. This is nice if you are writing a library and want to provide the end user with some example files.

**@extends <name>:** Another C++/Java-ism – Defines the base class that the derived class derives from.

**@file [name]:** Defines the current file. You should include this in all of your Doxygen comments. Although not necessary, it is required if you want to document external variables, as they rely on this tag to reference the location in which they are declared.

**@implements <name>:** Same as extends but for interfaces.

**@internal:** This is used for an internal comment that should not be included in the static page generated by Doxygen. The comment will terminate naturally at the end of the comment block, but if you require that it terminates before the end of the comment block, you can use **@endinternal**.

**@mainpage [title]:** This tells Doxygen what the front page should be in the static site.

**@namespace <name>:** Indicates that the comment relates to the namespace indicated by name.

**@overload [function decl]:** Creates documentation for overloaded functions.

**@package <name>:** Indicates that the comment addresses the package indicated by name.

**@private, @protected, @public:** Indicates that a member or function is private, protected, or public respectively. Note that you do not need to include these in OOP languages such as C++/Java, as Doxygen will detect this for you. These are only used for languages that don’t support protection natively (which… why would you use these for those?)

**@static:** Similar to private, protected, and public, static should be detected by languages which natively support it. Judging by the fact that the official documentation uses C as an example of a language which doesn’t natively support static (even though it has the static keyword), I’m assuming Doxygen means a static variable in the OOP sense of the word.

**@struct, @union <name> [header file] [header name]:** Indicates that the comment is in reference of the struct/union indicated by name.

**@author, @authors:** Gives the author(s) name(s). These tags function identically to one another.

**@brief {description}:** Gives a one line briefing. If the brief is split into multiple lines, only the first one is considered as the brief.

**@bug {description}:** Reports a known bug in the code. Multiple @bug tags next to one another are just merged into one bug. The tag ends either when a blank line or other tag is encountered.

**@copyright {description}:** Primarily used for larger corporations if you have some sort of legally binding copyright for the written code.

**@deprecated {description}:** Describes a description as to why the particular entity is deprecated.

**@param ‘[‘dir’]’ <param name> [param description]:** This tag is a bit unique. It declares a function parameter. Square brackets may be added after the @param tag with a value of “in”, “out”, or “in,out” depending on whether or not the parameters are incoming, outgoing, or both. This might look something like the following (taken from the Doxygen docs):

/\*\*

\* @param[out] dest The memory area to copy to.

\* @param[in] src The memory area to copy from.

\*/

**@return, @returns {description}:** Gives a description of what the function returns. The @return and @returns tags are identical.

**@see {references}:** Similar to @example, describes one or more classes, functions, methods, variables, files, or URLs as references to view for more information. Two names joined with :: or # are understood as referring to a class (left-hand side) and one of its members (right-hand side).

**@since {date}:** Declares the date or time that the entity was added to the codebase.

**@exception, @throw, @throws <exception object> {description}:** All equivalent in functionality. They give a description of a throwable exception object within the function or class.

**@todo {description}:** Describes a TODO item. Doxygen is nice enough to create a TODO list with all @todo items. Personally, I would rather keep my TODOs as @internal TODO:, but that is up to you.

**@version {version num}:** Gives the version number of the program/application.

**@note, @warning {description}:** Creates a note/warning message. The paragraph generated will be indented by Doxygen and have a little yellow or red indicator bar next to the comment respectively. Similar to @bug, multiple @note/@warning tags next to one another will be joined into one note/warning.

As you can see, there are many commands/tags available to us in Doxygen, many of which resemble Javadoc, and all of which are intuitive and easy to remember. As I mentioned, there are many more of these tags. I included ones which I think I would reasonably see someone using, though I’m sure there are others that I omitted which might be beneficial under certain circumstances. Once more, I implore you to read the official docs for a better overview.

One Other Detail:

One final detail which I did not include is single line Doxygen comments. I wasn’t sure where to mention that, so I’ll mention it here. If you would like to document a variable on the same line as its declaration, you will need to use a single line Doxygen comment. This is accomplished using the same Javadoc or Qt syntax. The catch is that you must use a ‘<’ symbol to indicate that we’re referencing a variable. Here are a few examples:

int var; /\*!< Multiline description \*/

int var; /\*\*< Multiline description \*/

int var; ///< Single line description

int var; //!< Single-line description

There exist other comment syntaxes for multiline comments, however, I think you’ll agree with me upon seeing them that they are rather quite hideous. These 4 seem to be the most sane ones, the second being my personal favorite because of the consistency between Javadoc, and support on C99 or lower. On C++ or Java, the choice will have even less of an impact.

Compiling Doxygen:

It is probably good that you actually know how to compile your comments into the proper HTML files. To do this, simply run the doxygen command followed by the name of the configuration file (Doxyfile is the default). Note that up until now, I’ve only mentioned support for HTML. In reality, Doxygen supports other output formats such as latex, rtf (MS Word), and a few others. I figure that if you are familiar with these formats, you’re probably smart enough to figure out how to compile to them.

Markdown