

# Embracing the Git Index

**“If you deny the Index, you really deny *git* itself.”**

By Jon Loeliger

Linus Torvalds once said, “If you deny the Index, you really deny *git* itself.” (February 4, 2006, Git List Archives). Rather than try to sweep the mysteries and complexities of the *git* Index under the rug, some explanation and examples can help clarify it, expose its power, and allow you to revel in it!

There are several articles that provide background information about *git*. The impetus for the software’s creation and some of the early history of *git*’s development are covered in Sam Williams’s article “Git With It!” in the August 2005 issue of *Linux Magazine* (available online at <http://www.linux-mag.com/2005-08/git/>). The first part of this series, “How to Git It” (available online at <http://www.linux-mag.com/2006-03/git/>), explains how to obtain and install *git*, and introduces the basics of *git*’s *Object Database* and *Index*.

In this second part of the series, let’s look more at the Index. The Index is fundamental to *git*, and provides the basis for almost every operation, simple or complex, that *git* provides. Let’s create a project, follow the code through several revisions implemented by two independent developers, and finally unify the results. Along the way, the subtle yet powerful operations within the *git* Index will be explained.

## Creating a Project With *git*

The simplest way to create a new project with *git* is to use the `git init-db` command. The command can be used to either create a brand new repository for new development, or can be used on a snapshot of an existing source tree to turn it into a *git* repository.

Here is a quick-and-dirty prime number program written in C. Never mind that it has issues — those will be fixed soon enough. Before making any modifications, place it under source control with `git init-db`.

```
$ pwd
/usr/src/primes

$ cat primes.c
int main(int argc, char **argv)
```

```
{
    int n;
    int d;
    int prime;

    for (n = 1; n <= 25; n++) {
        prime = 1; /* assume prime */

        for (d = 2; d <= sqrt(n); d++) {
            if ((float) n / d == n / d) {
                prime = 0;
                break;
            }
        }

        if (prime) {
            printf("%d is prime\n", n);
        }
    }
}
```

```
$ git init-db
defaulting to local storage area
```

```
$ ls -aF
./ ../ .git/ primes.c
```

While the *primes* directory has been turned into a *git* repository complete with a *.git* directory, the *git* Index still has no knowledge of what files need to be managed. You can add files one at a time using `git add file` or you can add an entire directory structure using `git add .`:

```
$ git add .
```

```
$ git status
#
# Initial commit
#
# Updated but not checked in:
#   (will commit)
#     new file: primes.c
#
```

`git add` tells the Index to add the new file *primes.c* to the repository. So far, though, *git* has only modified the Index and added the one blob object that represents *primes.c*. The state has not yet been committed to the Object Database.

Recall that the Index forms a staging ground where modifications to the repository collect until the time you want to commit and enter them into the Object Database. `git status` provides a very valuable guide to help determine what a `git commit` operation will do. In this case, there is one new file, *primes.c*, to check in and commit.

```
$ git commit -m "Initial primes project."
Committing initial tree
2b076ab786485f2af451b3f5139099155ae93f0c
```

```
$ git status
nothing to commit
```

After the commit, `git status` reports that there are no longer any changes to commit, nor are there any changes that are seen but won't be committed. The `git log` command can now be used to see historical commit information.

```
$ git log
commit
9ec29c6852926e5fb583d68275f6d28574b355d7
Author: Jon Loeliger <jdl@freescale.com>
Date: Tue Mar 14 19:36:21 2006 -0600
```

```
Initial primes project.
```

With a repository of at least one file, the interesting operations on the Index can be shown.

## Index as Staging Ground

The Index can be used as an incremental staging area for the development of the next revision of the repository to be committed. Any changes that need to be made must be introduced through the Index. For example, each new file must be explicitly added, as above, or *git* won't maintain it under revision control. However, you may wish to make a

distinction between three classifications of files: those that should be under revision control, those that are never to be managed by revision control (or even really thought about), and those that are not yet classified.

Introducing a new file, *Makefile*, to build the *primes* executable from *primes.c* exposes these classification issues.

```
$ cat Makefile
primes: primes.o
$(CC) -o $@ -lm $<
```

```
$ make
cc -c -o primes.o primes.c
primes.c: In function 'main':
primes.c:12: warning: incompatible implicit \
declaration of built-in function 'sqrt'
primes.c:19: warning: incompatible implicit \
declaration of built-in function 'printf'
cc -o primes -lm primes.o
```

Oh, no! Disastrous warnings! But it looks like it built nonetheless. Before the warnings are cleaned up, though, what does *git* "think" about all these new files? The command `git ls-files` asks for a list of files known to *git* and `git ls-files --others` asks for the set of files that are not known. (`git ls-files` is the basis for the `git status` command.)

```
$ git ls-files
primes.c
```

```
$ git ls-files --others
Makefile
primes
primes.o
```

*git* maintains a manifest of all the files that need to be managed under revision control. Use `git add` on *Makefile* to add it:

```
$ git add Makefile
$ git status
#
# Updated but not checked in:
#   (will commit)
#     new file: Makefile
#
# Untracked files:
#   (use "git add" to add to commit)
#     primes
#     primes.o
```

However, the other two files, *primes* and *primes.o* are generated files that likely shouldn't ever be under revision control. If no

other action is taken with respect to those two files, `git status` will continually issue a reminder that the files are “untracked files” each and every time they are present in the directory (that is, after a build). Instead, direct `git` to ignore these files, effectively classifying them as “never under revision control.”

To do that, add both files to a special file called `.gitignore` in the appropriate directory, and add *that file* to the Index:

```
$ cat .gitignore
primes
primes.o

$ git add .gitignore
```

```
$ git status
# Updated but not checked in:
# (will commit)
# new file: .gitignore
# new file: Makefile
```

While `Makefile` has been added, nothing has been committed to the Object Database yet. So far, all of the `git` commands have operated on the Index, building up a new state of the repository that can be captured with the next `git commit`. Since the status above shows just the addition of the two new files, and nothing else is left unresolved, the next commit has now been fully staged in the Index.

```
$ git commit -m "Add Makefile and .gitignore"
```

The Index can also be used to indicate that existing files are ready to be committed after some local modifications. For example, the compilation warnings can be eliminated by adding two `#include` lines to the top of the file `primes.c`:

```
#include <stdio.h>
#include <math.h>
```

To see the effect of this change, simply invoke `git diff`:

```
$ git diff
diff --git a/primes.c b/primes.c
index a8b37bb..1fa6c35 100644
-- a/primes.c
+++ b/primes.c
@@ -1,3 +1,6 @@
+#include <stdio.h>
+#include <math.h>
+
+ int
+ main(int argc, char **argv)
+ {
```

After making this modification, the local directory contents have been modified, but the Index has not. Running `git status` indicates that there is a modified file that is not yet updated, and issues a reminder that `git update-index` may be used to perform that step.

```
$ git status
# Changed but not updated:
# (use git-update-index to mark for commit)
# modified: primes.c
nothing to commit
```

```
$ git update-index primes.c
```

```
$ git status
#
# Updated but not checked in:
# (will commit)
# modified: primes.c
```

While this may seem like a trivial example, it's the crux of the mechanism `git` uses to determine that a file needs to be committed. In this state, the `primes.c` file has had a new object with a new SHA1 introduced into the object store as a result of the `update-index` request, and the Index now knows that it's been updated. The jargon used to describe this state is usually something like “a *clean but uncommitted* working directory.” Before the `update-index`, when the directory was modified but not reflected in the Index, the state was considered *dirty*.

To highlight an important feature of the Index, make one more modification before committing the revision: change the upper bound from 25 to 50 in `primes.c` and run `git diff` again:

```
$ git diff
diff --git a/primes.c b/primes.c
index 1fa6c35..04290d9 100644
-- a/primes.c
+++ b/primes.c
@@ -8,7 +8,7 @@ main(int argc, char **argv)
     int d;
     int prime;

-   for (n = 1; n <= 25; n++) {
+   for (n = 1; n <= 50; n++) {

         prime = 1; /* assume prime */
```

That may have come as somewhat of an unexpected surprise! What happened to the two `#include` lines? Use `git status` to help figure out what happened:

```
$ git status
```

```
# Updated but not checked in:
#   (will commit)
#     modified: primes.c
#
# Changed but not updated:
#   (use git-update-index to mark for commit)
#     modified: primes.c
```

*primes.c* is listed twice, both as a change that will be committed and as a change that will not be committed! In fact, the change that has been staged in the Index and that will be committed is the version of *primes.c* that existed when the `update-index` operation was performed on it, namely the addition of the `#include` lines. However, the Index also knows that the file was further modified after it was staged.

The file, as it is in the working directory, really has both changes in it. But as the developer, you must now ask yourself the question, “Which version should be committed?” If you want the version staged in the Index, just perform `git commit`. After all, that’s the purpose of staging the changes in the Index! The hard constant change will remain in the working directory. On the other hand, if you want both changes to be committed, you must run `git update-index` to place the correct version in the Index before committing it.

## Finding Differences

The `git diff` command is really a swiss-army knife for determining the set of changes between various versions of files in the working directory, the Index, and the Object Database.

Without parameters, as used above, or with only file names, `git diff` generates the differences between the current working directory and the Index. Thus, the first invocation of `git diff` above compared the working directory to the Index and discovered that the working directory had added the `#include` lines.

Executing the `git update-index primes.c` command, though, brought the working directory version into the Index, thus synchronizing the Index and the working directory versions. Another `git diff` would show no differences.

To reveal the complete set of changes between the most recently committed version of the file, also known as the *HEAD* version, and the working directory version, the `git diff HEAD` command can be used.

```
$ git diff HEAD
diff --git a/primes.c b/primes.c
index a8b37bb..04290d9 100644
-- a/primes.c
+++ b/primes.c
@@ -1,3 +1,6 @@
```

```
+#include <stdio.h>
+#include <math.h>
+
+   int
+   main(int argc, char **argv)
+   {
@@ -5,7 +8,7 @@ main(int argc, char **argv)
+   int d;
+   int prime;
-   for (n = 1; n <= 25; n++) {
+   for (n = 1; n <= 50; n++) {
+
+       prime = 1; /* assume prime */
```

To reveal just the changes between the *HEAD* version and the version staged, or *cached*, in the Index, add the `--cached` flag:

```
$ git diff --cached HEAD
diff --git a/primes.c b/primes.c
index a8b37bb..1fa6c35 100644
-- a/primes.c
+++ b/primes.c
@@ -1,3 +1,6 @@
+#include <stdio.h>
+#include <math.h>
+
+   int
+   main(int argc, char **argv)
+   {
```

As that represents the differences between the *HEAD* and Index, those are the changes that will be committed with the first `git commit`. The `update-index` and second `commit` picks up the changes for the upper limit constant changes, commits them, and cleans up the working directory:

```
$ git commit -m "Add include headers"
$ git update-index primes.c
$ git commit -m "Change limit to 50"
```

## Branches

Development of software within a repository is often performed in discrete steps, and often on different development branches within the same repository.

A *branch* is a conceptual line of development that usually has split off of the main development line at some identifiable point in the past. The power of branches comes from the fact that multiple, independent, but hopefully coordinated development efforts can then happen on different branches. Later, if needed, a *merge* operation can bring multiple branches back together.

There are two ways to create new branches: `git branch` and `git checkout -b`. Both forms allow you to introduce a new, named line of development and to specify the basis location for the branch, defaulting to the current `HEAD` revision of the current branch.

```
$ git branch
* master
$ git branch brad
$ git branch
    brad
* master
```

Here, `git branch` without arguments is used to determine the current set of branches in the repository, namely `master`. This is the active branch, as denoted by the `*` next to it, and was created by the initial `git init -db` as the default branch to hold the main-line or master development. Up until now, all the development that's been done has occurred on this `master` branch!

A new branch for developer Brad has been introduced using `git branch brad`. As no additional argument was used, the `brad` branch splits from the main line development at the current `HEAD` revision of the `master` branch. Using `git branch` again shows that there are now two branches, `brad` and `master`, with `master` still the checked-out current branch.

Internally, `git` tracks these branch names in the `.git/refs/heads` directory. Branches are very inexpensive because they are nothing more than the recording of a SHA1 value in a file.

```
$ ls .git/refs/heads/
brad master
$ cat .git/refs/heads/brad
1964adf822b7317df39b95aab244521b04a51ee
$ cat .git/refs/heads/master
1964adf822b7317df39b95aab244521b04a51ee
```

Since a branch name in `git` refers to the most recent commit on that branch by its SHA1, it's clear that the `brad` and `master` branch are currently at exactly the same state.

A more useful way to see the branch state is to use the `git show-branch` command:

```
$ git show-branch
! [brad] Change limit to 50
  * [master] Change limit to 50
--
+* [brad] Change limit to 50
```

Above the `--` separator, each line represents a named branch within the repository, with the current branch denoted with

a `*`. A one-line commit line from the commit log message is used to help identify the `HEAD` commit represented by each branch. Associated with each branch in a vertical column below the `--` separator are characters that indicate if a particular commit is or is not present in a branch. As indicated by the last line with `+`, both the `brad` branch and the `master` branch contain the `Change limit to 50` commit. The column would be empty if the commit named on the line was absent.

If Brad is tasked with adding some documentation, he can do that independently in his branch after checking it out using:

```
$ git checkout brad
```

Assume Brad makes the following edit, commits it, makes a few more edits, commits them as well, and then finally re-examines the branch structure. Using `-a` with `git commit` causes it to automatically do an `update-index` before the actual commit.

```
$ git diff
diff --git a/primes.c b/primes.c
index 04290d9..910aa4e 100644
-- a/primes.c
+++ b/primes.c
@@ -1,3 +1,7 @@
+/*
+ * The Prime Number Project
+ */
+
+    #include <stdio.h>
+    #include <math.h>

$ git commit -a -m "Add header comment."

# ... a few more edits and commits...

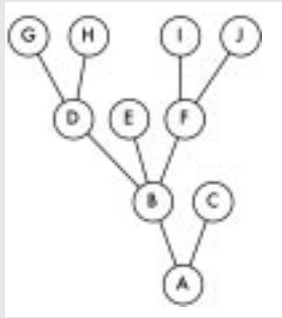
$ git show-branch
* [brad] Document divisor search.
  ! [master] Change limit to 50
--
* [brad] Document divisor search.
* [brad^] Document main loop.
* [brad~2] Add header comment.
+* [master] Change limit to 50
```

There are three more commits on the `brad` branch than there are on the `master` branch, namely the `brad`, `brad^` and `brad~2` commits; these commits have `*` in the `brad` column, while the corresponding `master` column entries are blank.

The funny names `brad^` and `brad~2` are short-hand names that refer to commits in the history of the `brad` branch.

## HOW TO READ COMMIT SHORT-HAND NAMES

The figure below shows a possible commit graph.



In the commit graph, both nodes B and C are commit parents of node A. Nodes D, E, and F are commit parents of node B. Node C is apparently the head of some development branch and has no parents.

Parent commits are ordered and numbered left-to-right, and denoted by the ^ (caret) symbol: ^1, ^2, ^3, and so on. A bare ^ with no following

number is the same as the first commit parent, ^1. For example, the third commit parent of node B is node F, and is denoted B^3.

It is more common to see a chain of commits one after the other that form a child, parent, grandparent relationship. The *n*th generation parent, strictly following the *first* parent, is represented with a tilde, ~, and the generation, *n*. For example, the first parent of A is A~1. The first parent of the first parent of A is D, also called A~2.

While the ^ and ~ identifiers can be combined to navigate back through the commit tree to any node in the graph, most often a near-linear sequence of commits using ^, ~2, ~3 is used.

```

A = A^0
B = A^  = A^1    = A~1
C = A^2
D = A^^ = A^1^1 = A~2
E = B^2 = A^^2
F = B^3 = A^^3
G = A^^^ = A^1^1^1 = A~3
H = D^2 = B^^2   = A^^^2 = A~2^2
I = F^  = B^3^   = A^^3^
J = F^2 = B^3^2  = A^^3^2
  
```

brad^ is the *parent*, or previous, commit before the current HEAD of the brad branch, and brad~2 is the parent of brad^ or grandparent of brad. (See the sidebar above to learn more about commit short-hand names.)

Since branch names are merely references to a particular commit, they can be used in commands such as `git diff`. For example the differences between the `master` and `brad` branches can be obtained using `git diff master brad`. Differences from one commit on a branch to another can also be done using the shorthand references:

```

$ git diff brad^ brad
diff --git a/primes.c b/primes.c
index ccc134b..1a0fafd 100644
-- a/primes.c
+++ b/primes.c
  
```

```

@@ -17,6 +17,7 @@ main(int argc, char **argv)

    prime = 1; /* assume prime */

+   /* Search for a divisor */
    for (d = 2; d <= sqrt(n); d++) {
        if ((float) n / d == n / d) {
            prime = 0;
        }
    }
  
```

## Using the Index for Merging

One of the most powerful operations that *git* supports is a *merge* between two or more branches. A merge brings the changes introduced on different branches back into one branch. In *git*, any change introduced by a merge operation affects the current branch.

Normally, a successful merge will directly commit the resulting changes on the current branch. However, it is possible that independent changes on different branches cause conflicts when their branches are merged. In this case, the user is responsible for both resolving the conflicts and performing the `git commit`.

Suppose that another developer, Susan, wanted to introduce her own branch to do development, and that she wanted to base her branch on the version of the master branch after the `Makefile` was introduced and the `#include` file fixes were done. By inspecting the branch history, she can determine that the commit named `master^` should be used:

```

$ git show-branch --more=5
* [brad] Document divisor search.
  ! [master] Change limit to 50
-
* [brad] Document divisor search.
* [brad^] Document main loop.
* [brad~2] Add header comment.
** [master] Change limit to 50
** [master^] Add include headers
** [master~2] Add Makefile and .gitignore
** [master~3] Initial primes project.

$ git checkout -b susan master^
  
```

The `git checkout -b susan master^` command has created a new branch named `susan`, based at `master^`, and has also checked out a copy of that branch.

Suppose Susan decided to change the upper limit to the value provided as the first command-line argument and checked that code in. A `git diff` might look like this:

```

$ git diff
diff --git a/primes.c b/primes.c
  
```

```

index 1fa6c35..a6e699f 100644
-- a/primes.c
+++ b/primes.c
@@ -7,8 +7,13 @@ main(int argc, char **argv)
     int n;
     int d;
     int prime;
+ int max = 25;

- for (n = 1; n <= 25; n++) {
+ if (argc == 2) {
+     max = atoi(argv[1]);
+ }
+
+ for (n = 1; n <= max; n++) {

        prime = 1; /* assume prime */

$ git commit -a -m "Get limit from cmd line."

```

There are three branches now, with various commits in them:

```

$ git show-branch
! [brad] Document divisor search.
  ! [master] Change limit to 50
  * [susan] Get limit from cmd line.
--
  * [susan] Get limit from cmd line.
+ [brad] Document divisor search.
+ [brad^] Document main loop.
+ [brad~2] Add header comment.
++ [master] Change limit to 50
+++ [susan^] Add include headers

```

*git* can now be used to merge one or more branches into another. Before performing the merge, though, you must change branches to the desired destination branch for the merge. That is, if the `master` branch is to contain the final merge result, use `git checkout master` first.

While it is possible to merge more than two branches at a time (at the time of writing, Len Brown holds the record with a 12-branch merge into a Linux kernel commit), the following sequence shows the `brad` and `susan` branches being merged into the `master` branch one at a time using the command `git pull` to perform the branch merge:

```

$ git checkout master
$ git pull . brad
Updating from 1964ad to 6dc9493
Fast forward
  primes.c |    6 ++++++
  1 files changed, 6 insertions(+), 0

```

```

deletions(-)
$ git show-branch
! [brad] Document divisor search.
  * [master] Document divisor search.
  ! [susan] Get limit from cmd line.
--
  + [susan] Get limit from cmd line.
+* [brad] Document divisor search.
+* [brad^] Document main loop.
+* [brad~2] Add header comment.
+* [brad~3] Change limit to 50
+++ [susan^] Add include headers

```

After this merge, all of Brad's commits are also present in the `master` branch as indicated by `*` in the middle column everywhere a `+` appears in the first column. Susan, however, still has one commit that isn't in the `master` branch.

```

$ git pull . susan
Trying really trivial in-index merge...
fatal: Merge requires file-level merging
Nope.
Merging HEAD with
f7d3600df07430f5193e38eeab911c01d582ba10
Merging:
6dc94935e54c50c1da9285f6f06364d34425ab80
Document divisor search.
f7d3600df07430f5193e38eeab911c01d582ba10
Get limit from cmd line.
found 1 common ancestor(s):
b83ab56d1ca1f3570b0add0f4d63ec52c7c3c36e
Add include headers
Auto-merging primes.c
CONFLICT (content): Merge conflict in primes.c

Automatic merge failed; fix up by hand

```

Clearly, that didn't go too well. *git* first tried to use a merge technique that operates trivially out of the Index directly, but failed. *git* then tried an alternate merge technique, and again failed to do so, because some of Susan's changes to `primes.c` conflict with some of Brad's changes. Ultimately, *git* determined that user intervention is necessary to resolve the conflict here.

But what's the conflict? And how do you resolve it? And once it's resolved, what do you do?

There are several ways to determine what went wrong. Since *git* leaves the conflicting and unmerged files in the working directory, it's easy to use `git diff` to determine what files need to be fixed. The tried and true `git status` also lists `primes.c` as being "unmerged."

Another painfully detailed way is to ask the Index what files are left unmerged:

```
$ git ls-files --unmerged
100644 1fa6c35270e35b781fe 1      primes.c
100644 1a0fafd894b68a1ac8b 2      primes.c
100644 a6e699f039c82160f3b 3      primes.c
```

This shows that *primes.c* is still unmerged and needs to be resolved. In fact, the Index knows about three versions of that file and supplies the SHA1 for each one. Roughly speaking, these three versions of the file correspond to the original version, the version from the *master* branch after *brad*'s changes have been introduced, and the version from the *susan* branch. You could use `git cat-file blob` on each of the SHA1s above, try to piece it together and figure out some resolution from there. The Index kept track of it all for you, but that's the hard way.

Instead, *git* has done most of the work already and has left the file as merged as possible in the working directory. The places with actual conflicts are delineated with the traditional markers still in the file.

```
<<<<<< HEAD/primes.c
/* Look for primes between 1 and 50 */
for (n = 1; n <= 50; n++) {
=====
    if (argc == 2) {
        max = atoi(argv[1]);
    }

    for (n = 1; n <= max; n++) {
>>>>>> f7d3600df07430f519/primes.c
```

It's left up to you to pick which variant, combination or modifications makes sense, and to remove the markers.

```
$ git diff
diff --cc primes.c
index 1a0fafd,a6e699f..0000000
@@@ -11,9 -7,13 +11,14 @@@
     int n;
     int d;
     int prime;
+   int max = 25;

-   /* Look for primes between 1 and 50 */
-   for (n = 1; n <= 50; n++) {
+   if (argc == 2) {
+       max = atoi(argv[1]);
+   }
+ 
```

```
++ /* Look for primes between 1 and max */
+   for (n = 1; n <= max; n++) {

        prime = 1;          /* assume prime */
```

This output effectively represents combined, multiple, unified diff's, and has multiple columns of + and - characters to represent changes versus multiple versions of the file in the Index simultaneously.

Once the file has been resolved, it's time to tell *git* via `git update-index`, and commit it:

```
$ git update-index primes.c
$ git commit
$ git show-branch
! [brad] Document divisor search.
 * [master] Merge in Susan's branch
 ! [susan] Get limit from cmd line.
--
- [master] Merge branch 'susan'
++ [susan] Get limit from cmd line.
+* [brad] Document divisor search.
+* [brad^] Document main loop.
+* [brad~2] Add header comment.
+* [brad~3] Change limit to 50
+++ [brad~4] Add include headers
```

Finally, both Brad's and Susan's changes are all in the *master* branch. If a development branch is no longer needed, it may be removed with `git branch -d`.

## More Exploration

There are a number of areas to explore further.

The branches introduced here are entirely self-contained and created solely to house local development even within the same repository. Branches can be used to track development from other repositories both on the same file-system and across the network. The details on how to set this up are described in the documentation for `git pull`.

If you need to base your work on an existing *git* repository that someone else has already published, look at `git clone`. Make your own development branches for your modifications, as described above.

More can be learned about merging and the internal workings of the Index itself by reading the `git-read-tree` and `git-diff-tree` documentation.

Many more advanced development mechanisms use the same principles as those shown here.

---

*Jon Loeliger currently works at Freescale Semiconductor developing Linux for the PowerPC.*