git introduction

Alessandro Di Federico

ale@clearmind.me

Algorithms and Parallel Computing
Mathematical Engineering
Politecnico di Milano

October 20, 2015

Index

Ever e-mailed stuff like this?

my-program.c
my-program-2.c
my-program-3.c
my-program-just-a-test.c
my-program-final.c
my-program-true-final.c
my-program-true-final-i-promise.c
my-program-i-m-sick-of-this.c
my-program-true-final-last-2.c

You don’t have to do this anymore!

What is a VCS?

Version Control System
: A system which allows to track the evolution of your code in time and easily go back and forth. It is also useful to manage contribution to a project from different developers.

Centralized vs distributed

Centralized

The history of your changes is on a remote, central server.

Distributed

The history of your changes is on your local machine and can be synchronized with others.

git

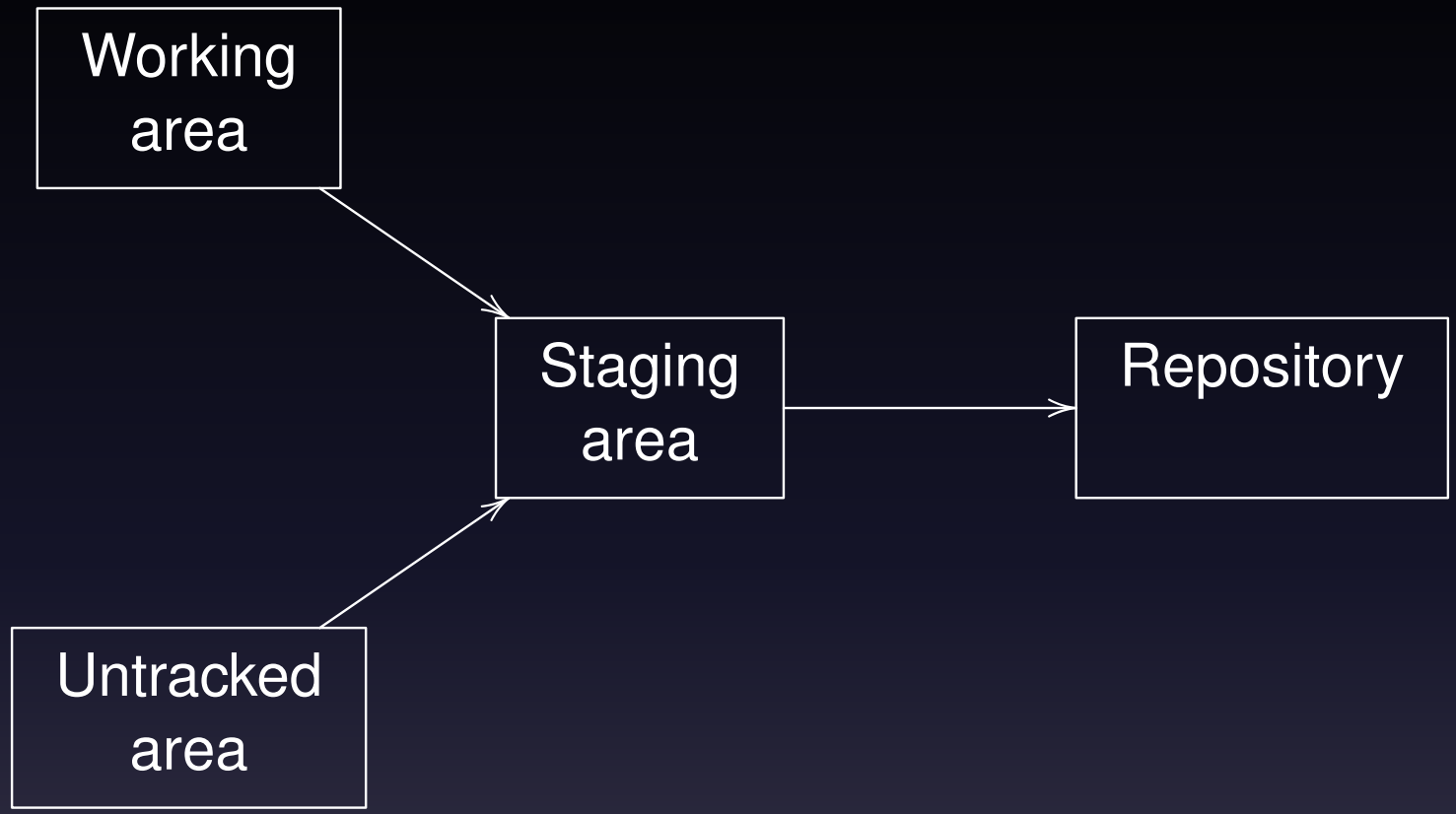
- Distributed VCS
- Developed by Linus Torvalds

Some definitions

Commit	A set of changes to one or more files along with a description, an author and a reference to the commit it is based on.
History	The chronology of commits from the start of the project up to now.
HEAD	Identifier for the most recent commit.
Repository	Location where all the commits are stored.

git tracks changes

- Suppose you have an existing file
- You want to change a line
- Your change can be in three different states



Working area	Changes you're still working on.
Untracked area	Contains files ignored by git are.
Staging area	Changes you're preparing for a commit.
Repository	Committed changes, ready to be shared.

My first repository

```
mkdir helloworld
cd helloworld
git init
```

Create a file

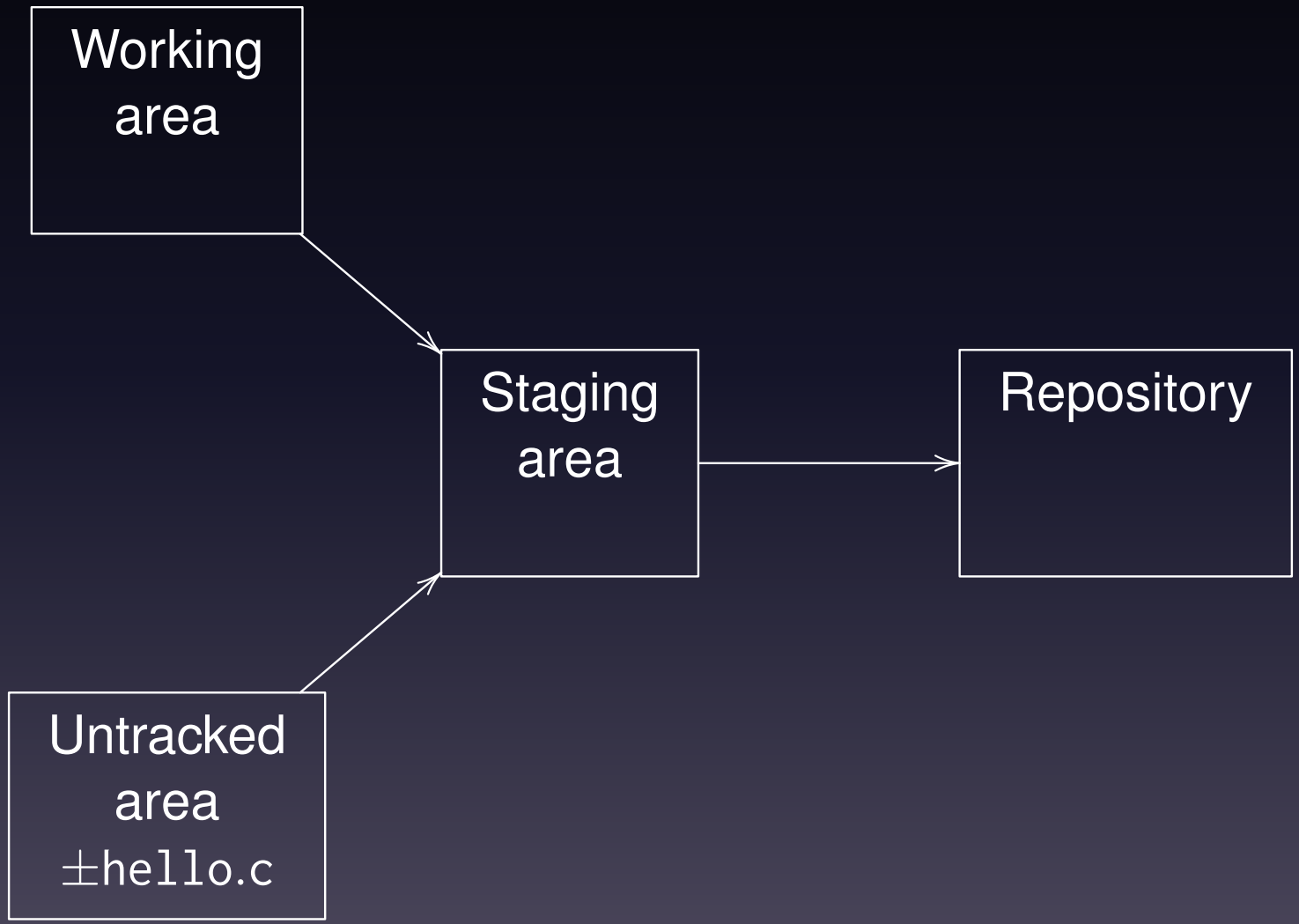
hello.c:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello world!\n");
  return 0;
}
```

git status

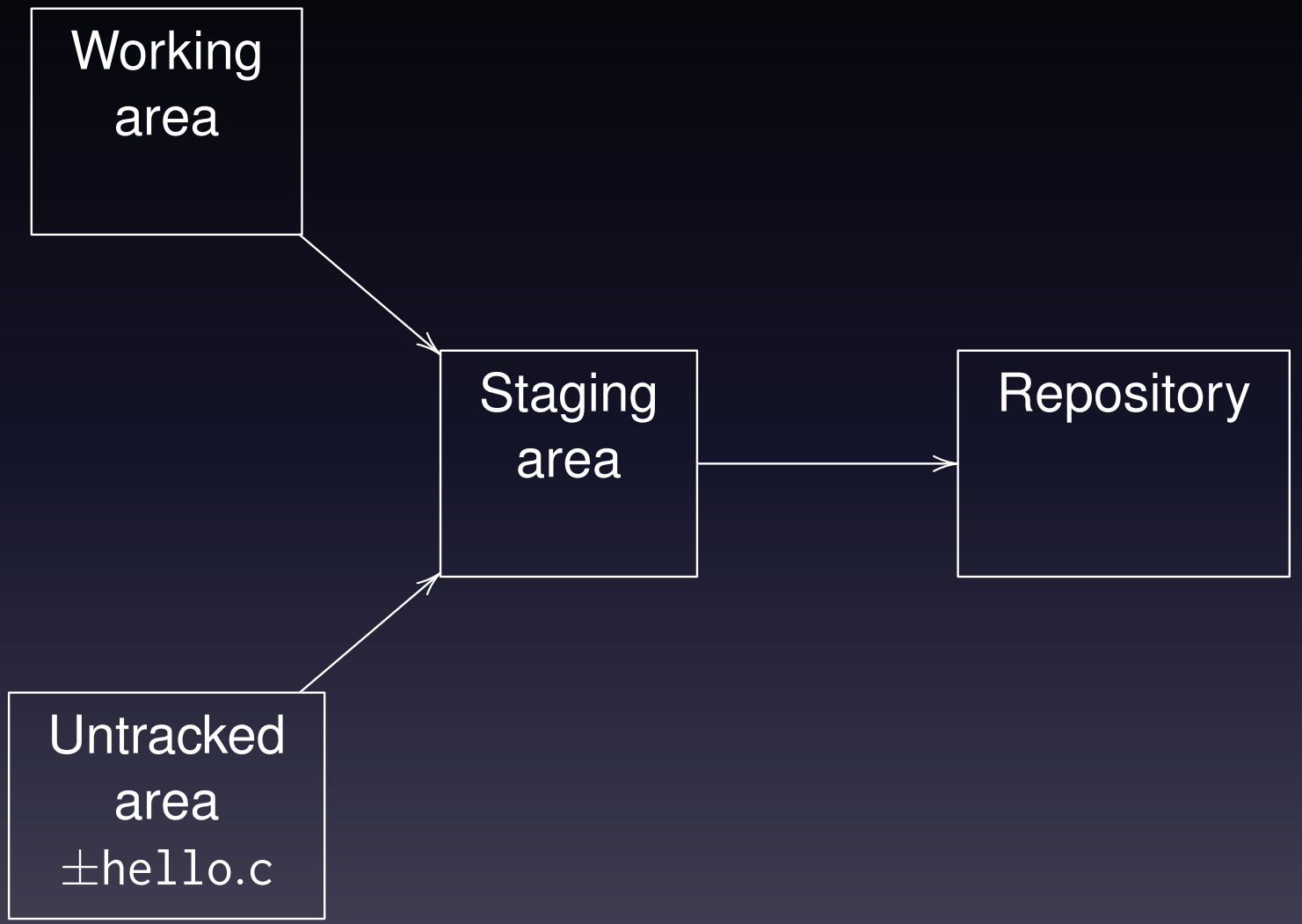
```
$ git status
Untracked files:
        hello.c
$
```

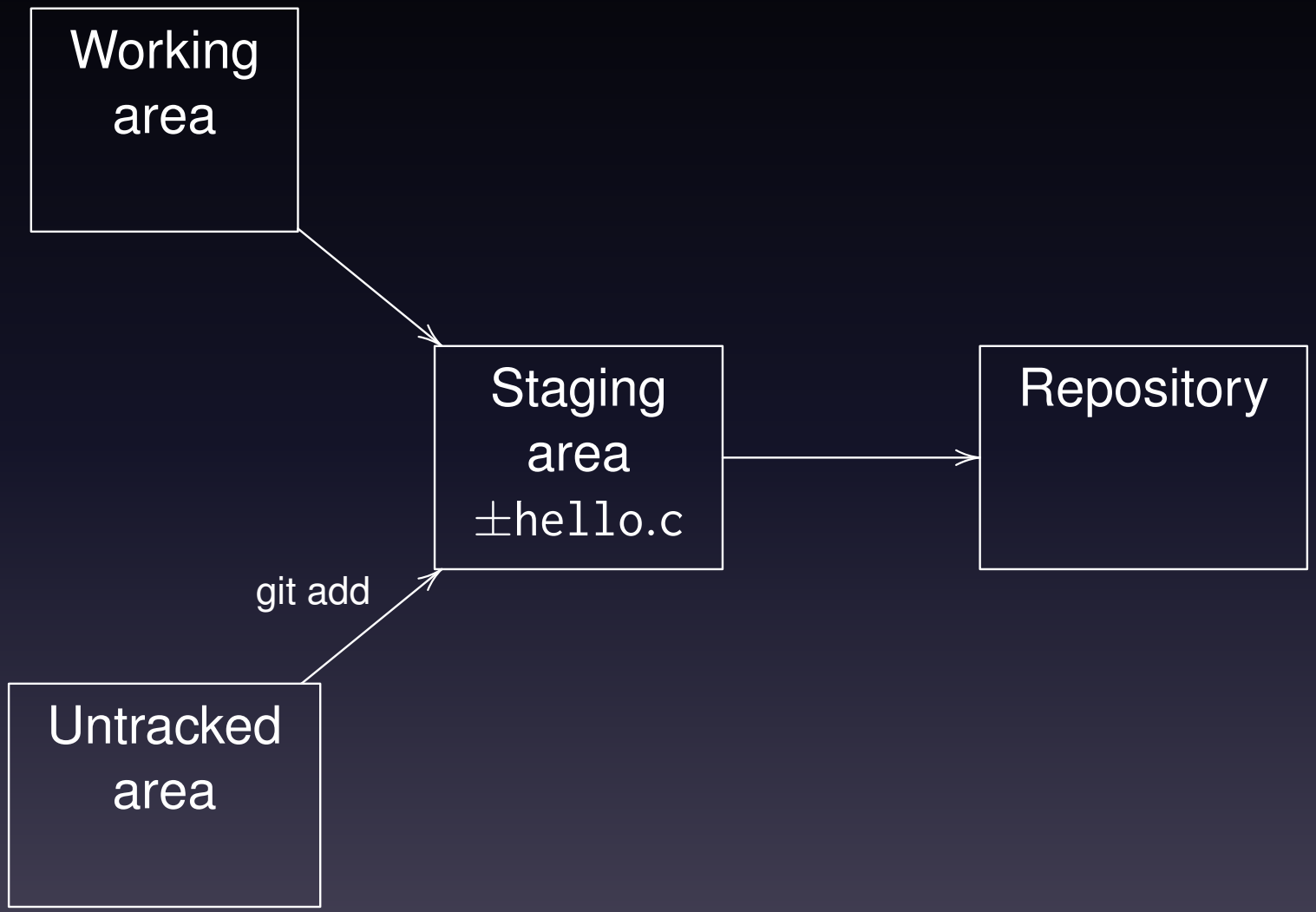
`hello.c` is initially untracked



Let’s stage it

```
$ git add hello.c
$ git status
Changes to be committed:
        new file:   hello.c
$
```


Working area
Staging area
Repository
Untracked area
±hello.c


Working area
Staging area
±hello.c
Repository
git add
Untracked area

What's in the staging area?

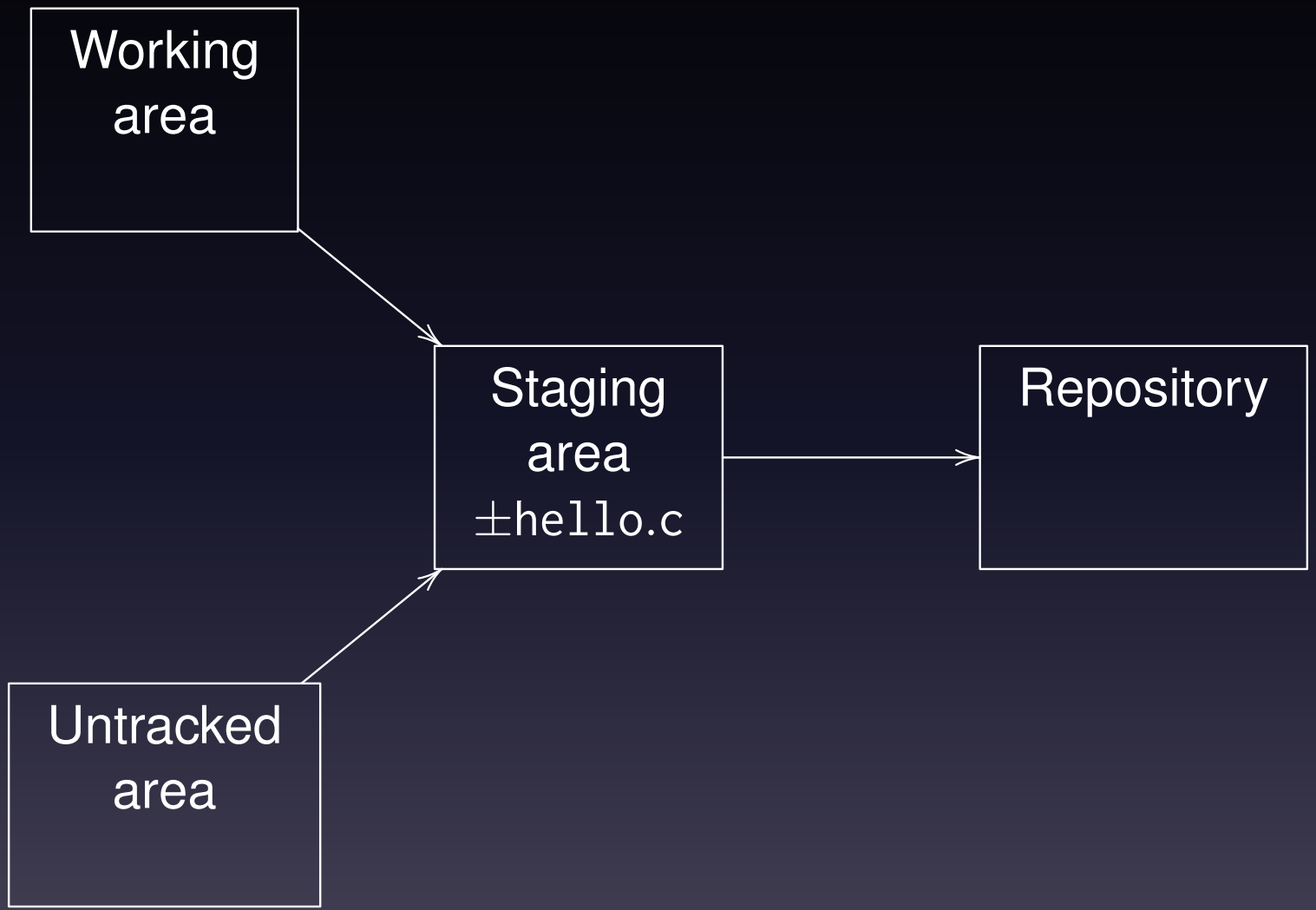
```
$ git diff --staged
diff --git a/hello.c b/hello.c
new file mode 100644
index 0000000..28bf75f
--- /dev/null
+++ b/hello.c
@@ -0,0 +1,6 @@
+#include <stdio.h>
+
+int main(int argc, char *argv[]) {
+  printf("Hello world!\n");
+  return 0;
+}
$
```

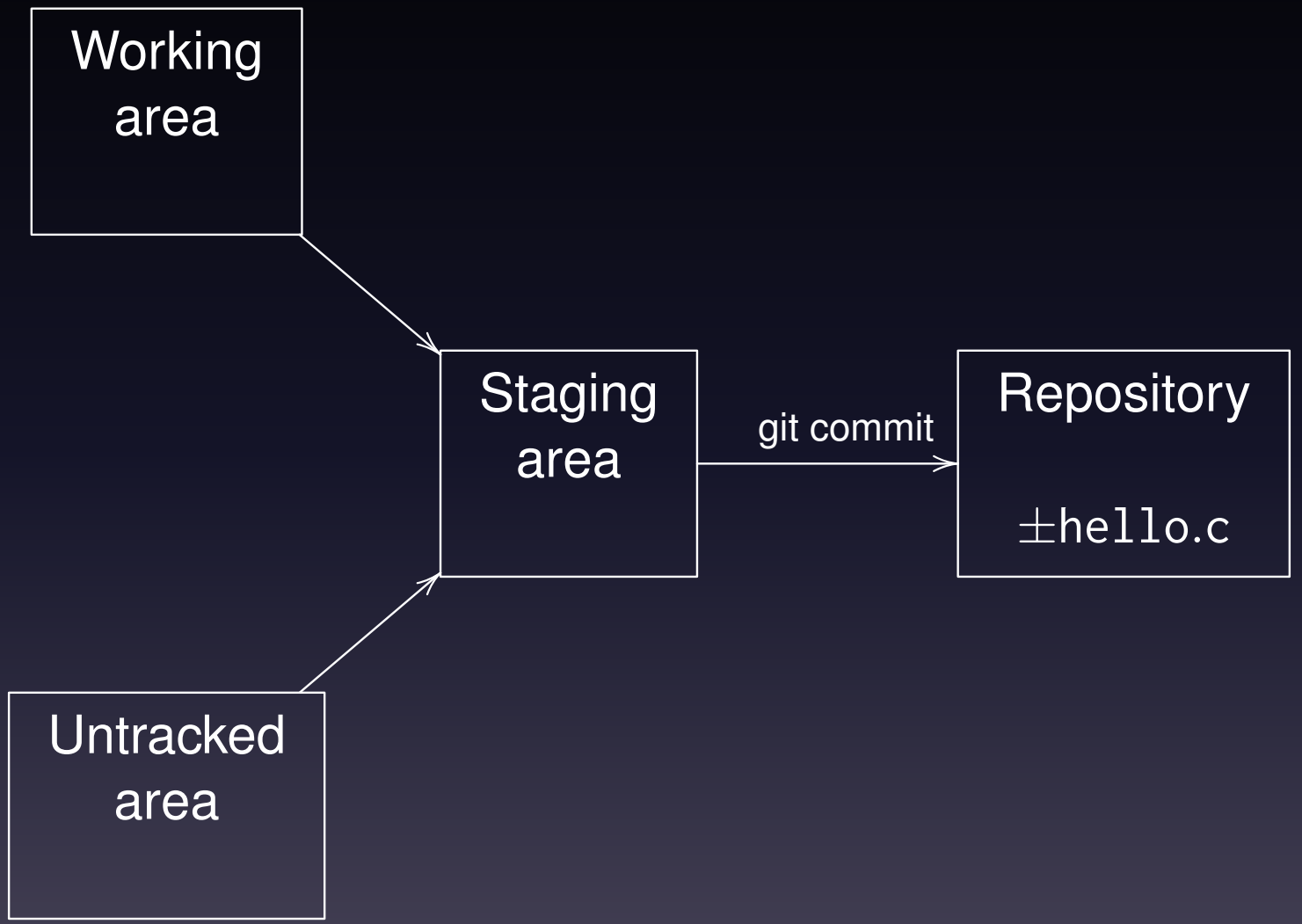
Looks good, let's commit

```
git commit
```

- An editor pops up
- Insert a message describing the changes
- Save and close

```
[master (root-commit) ce68f48] Initial import
   of the hello world program
 1 file changed, 6 insertions(+)
 create mode 100644 hello.c
```


Working area
Staging area
±hello.c
Repository
Untracked area


Working area
Staging area
git commit
Repository
±hello.c
Untracked area

git log

- We now have the first commit
- Therefore, we have a history
- Let's check it out:

```
$ git log
commit ce68f48f1ec56abb5de5ab6010ddc6bbb0ac7346
Author: Alessandro Di Federico <...>
Date:   Mon Oct 19 16:03:09 2015 +0200

    Initial import of the hello world program
```

ce68f48f1e... WTF?

`ce68f48f1ec56abb5de5ab6010ddc6bbb0ac7346`

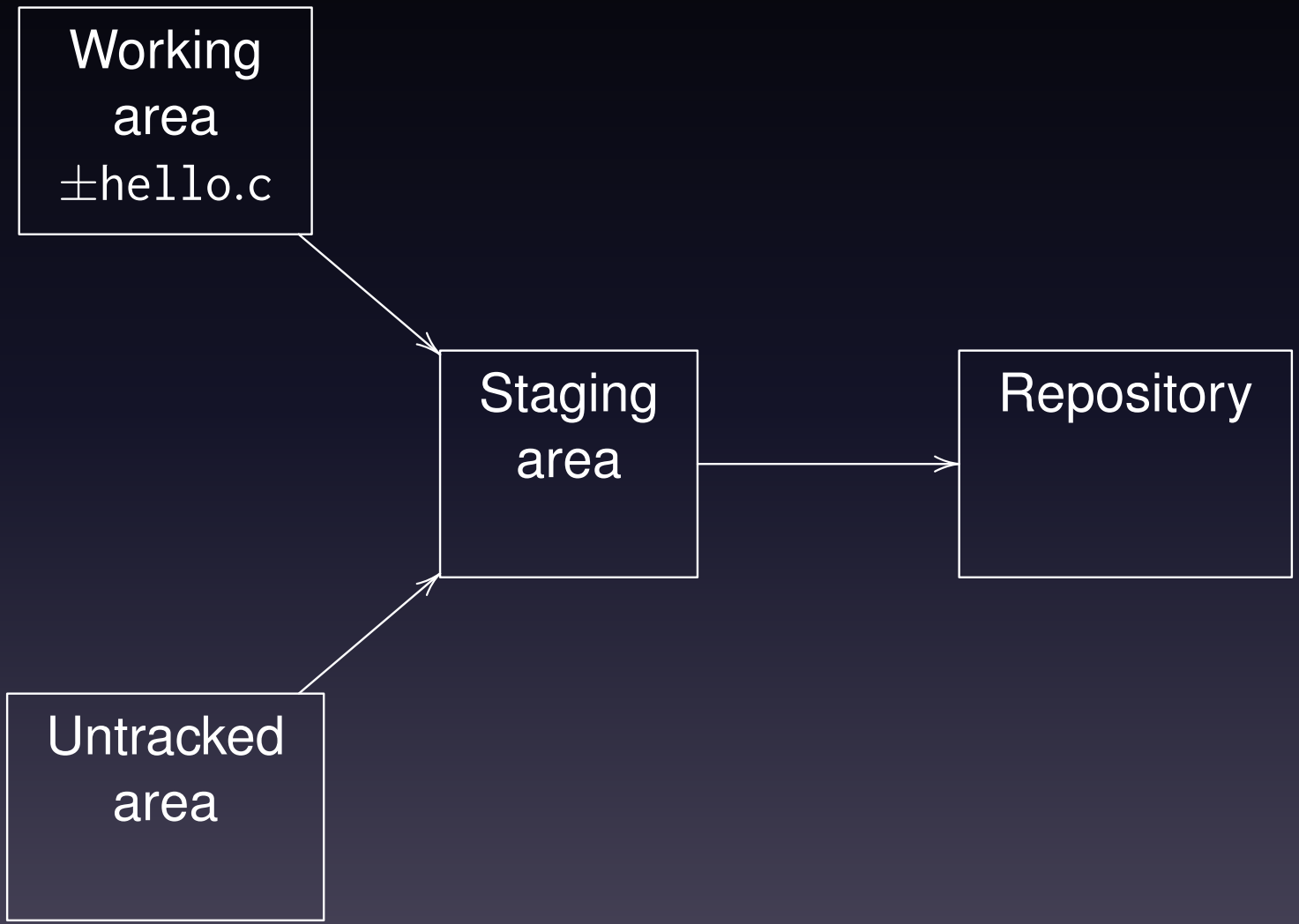
- It's the commit's unique identifier
- It's the hash of:
  - all the changes
  - commit message, author, date...
  - the reference of the parent commit
- Every time you want to refer to a commit use this
- Or a shorter version if it's unambiguous
- `git show` allows us to check the content of a commit

```
$ git show ce68f48f1e
commit ce68f48f1ec56abb5de5ab6010ddc6bbb0ac7346
Author: Alessandro Di Federico <...>
Date:   Mon Oct 19 16:03:09 2015 +0200

    Initial import of the hello world program

diff --git a/hello.c b/hello.c
new file mode 100644
index 0000000..28bf75f
--- /dev/null
+++ b/hello.c
@@ -0,0 +1,6 @@
+#include <stdio.h>
+
+int main(int argc, char *argv[]) {
+  printf("Hello world!\n");
+  return 0;
+}
```

Let’s make another change

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello world!\n");
  printf("I got %d args!\n", argc);
  return 0;
}
```

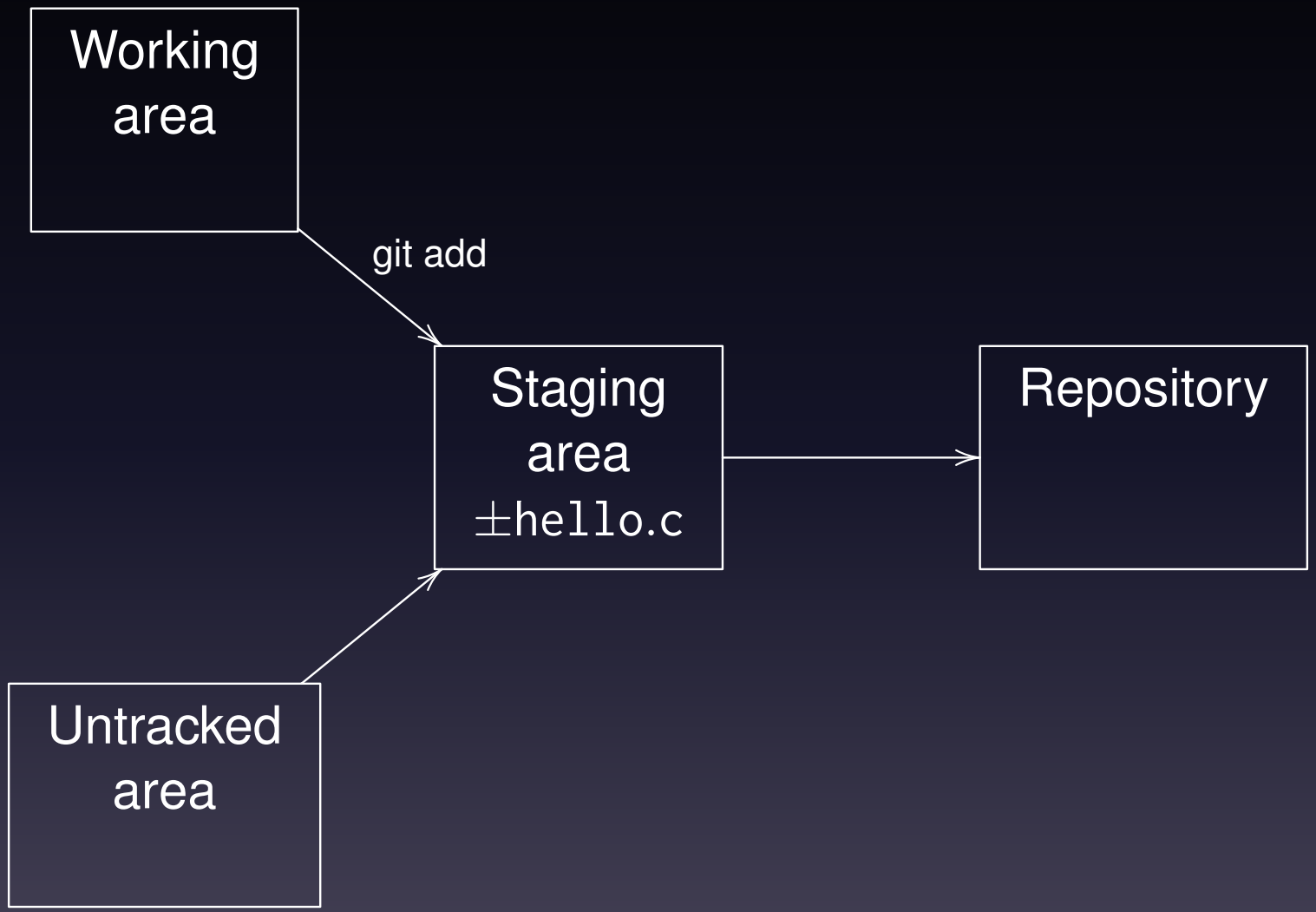
`hello.c` is now tracked, so the changes are in the working area:

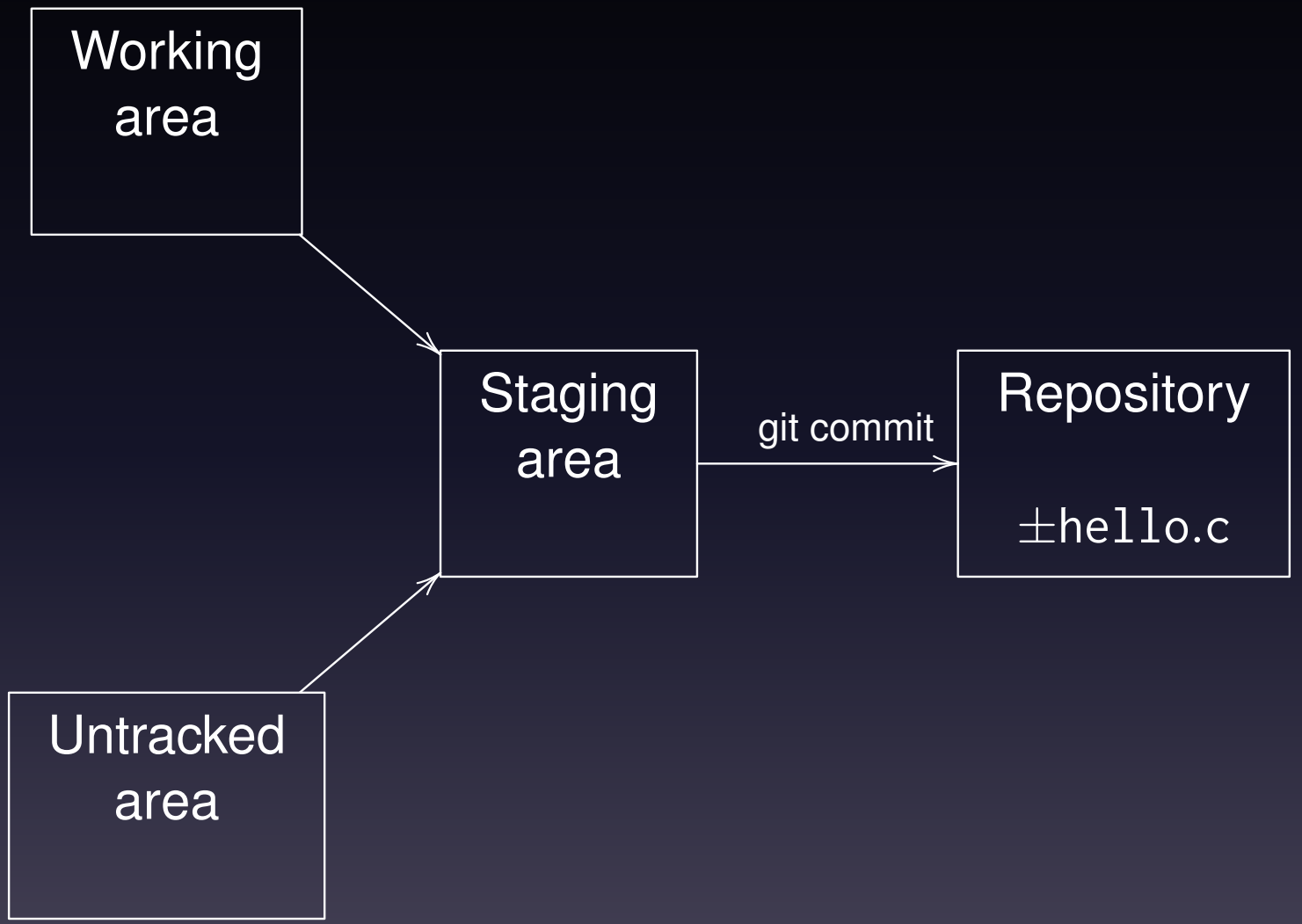


git diff

`git diff` shows what's in the working area:

```
$ git diff
diff --git a/hello.c b/hello.c
index 28bf75f..1d68247 100644
--- a/hello.c
+++ b/hello.c
@@ -2,5 +2,6 @@

 int main(int argc, char *argv[]) {
   printf("Hello world!\n");
+  printf("I got %d args!\n", argc);
   return 0;
 }
$
```


Working area
git add
Staging area
±hello.c
Repository
Untracked area


Working area
Staging area
git commit
Repository
±hello.c
Untracked area

Useful commands/1

`git add --patch` If you have multiple changes in a file in the working area, selectively move them to the staging area.

`git commit --amend` Take the last commit, add what's in the current staging area and let the user change the message.

`git reset filename` Move changes in the staging area back to the working area. If no *filename* is given, unstage everything.

`git checkout filename` Drops all the changes to *filename* and restores the version of the current HEAD commit.

Useful commands/2

`git rm filename` Deletes *filename* from the working area, and marks it as to be deleted in the staging area.

`git rm --cached filename` Leaves *filename* in the working area, but marks it as to be deleted in the staging area (will become untracked).

`git mv oldname newname` Renames *oldname* to *newname* both in the working and staging areas.

Important rule

Thou shall never commit generated files

- Compiled programs
- Documentation
- Exported PDFs
- ...

.gitignore

- The repository should not contain generated files
- If you really need them, use `.gitignore`
- `.gitignore` tells git to ignore certain files

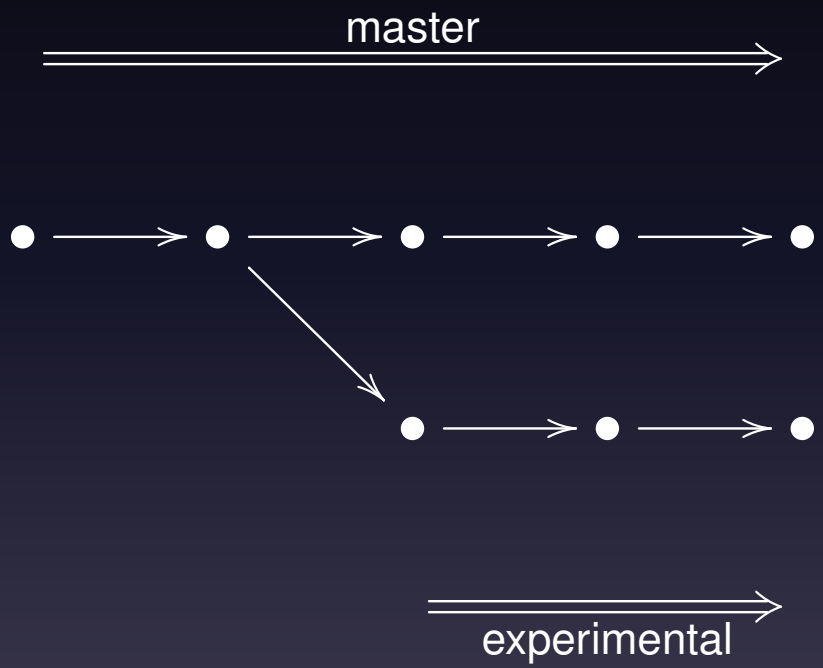
```
$ cat .gitignore
*.o
hello
```

Index

Branches

- git has the concept of *branches*
- Suppose you have two development lines:
  - regular development
  - development of an invasive, experimental feature
- You can create two branches

Schema of branches



Branches

- Each branch is associated to a commit
- `master` is the default branch
- You can list branches with their latest commit with `git branch -v`

Creating and changing branch

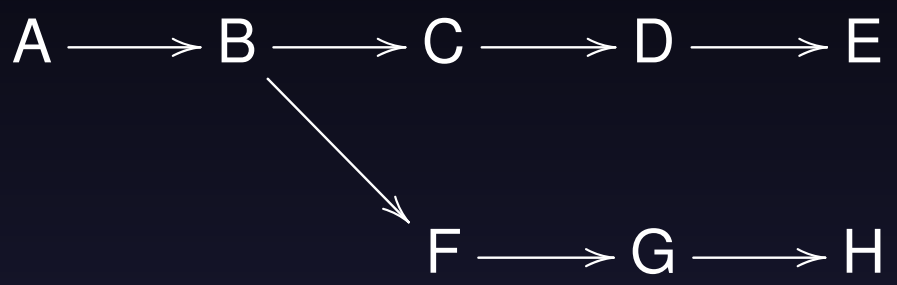
```
$ git branch experimental
$ git branch -v
  experimental ce68f48 Initial import of...
* master       ce68f48 Initial import of...
$ git checkout experimental
$ git branch -v
* experimental ce68f48 Initial import of...
  master       ce68f48 Initial import of...
```

Merging back into master

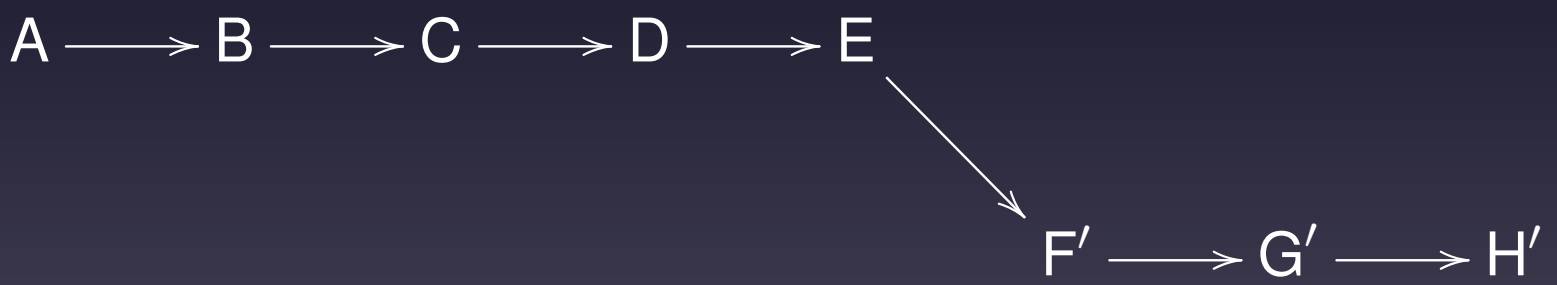
- After a while our feature is finalized
- We want to *merge* it back into the main development line
- We have two options:
  - use `git rebase`
  - use `git merge`

git rebase

From:



To:



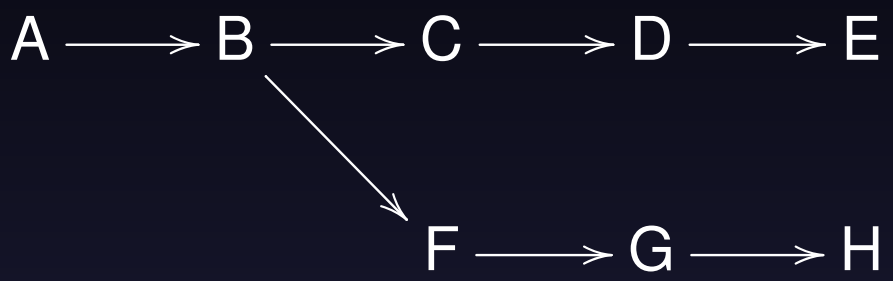
What does `git rebase` do?

```
$ git checkout experimental
$ git rebase master
```

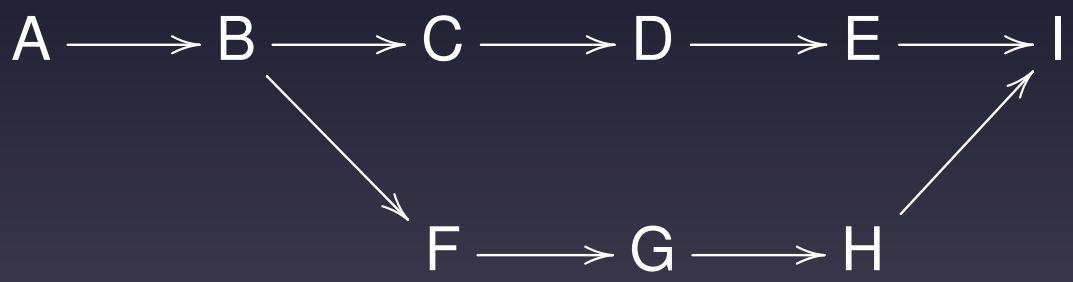
- Appends the current branch at the end of `master`
- It flattens the history
- It doesn’t introduce any new commit
- Rewrites the history of the merged branch

git merge

From:



To:



What does `git merge` do?

```
$ git checkout master
$ git merge experimental
```

- Creates a new commit with two parents
- The histories get merged
- No commit is rewritten

Conflicts

- While merging or rebasing you might have *conflicts*
- Conflicts are changes that git can’t merge automatically
- Typically when two commits act on the exact same code
- You have to fix them manually, `git mergetool` might help

rebase or merge?

- General rule: prefer rebase
- Having a lot of merge commits is ugly
- `rebase`: for small changes easily portable over master
- `merge`: for large changes generating lots of conflicts

git tag

```
git tag v1.0
```

- You can also label a commit with a tag
- It's just a label, not a development line
- `git tag` associates the label with the current HEAD

Index

Cloning an existing repository

- `git init` creates a new local repository
- What if you want to import an existing one?

git clone

- Creates a repository
- Imports all the commits
- Checks out the `master` branch
- Creates a new remote called *origin*

What is a remote?

- A remote is a URL with a name
- git uses this URL to sync (i.e. send and receive) commits
- The given command creates a remote “origin”:

https://github.com/torvalds/linux.git

- `git remote -v` lists remotes along with their URLs

Fetch and push

`git fetch remote` Downloads all the commits from the specified *remote*.

`git push remote branch` Pushes changes to a remote for a certain branch.

- Each branch tracks a remote branch
- You can view it with `git branch -vv`
- So, usually just `git fetch` and `git push` is fine

Remote branches

- `git branch -av` shows also remote branches
- Just a branch so you can merge and rebase them
- `git merge` merges from the tracked remote branch

`git pull`

- Equivalent to:
  - `git fetch`: sync with the default remote
  - `git merge`: merge from tracked branch

Index

GUIs

There are several:

- `gitk`: useful to visualize the history
- `gitg`: idem
- `git gui`: useful to selectively choose what to stage

Best practices: commit messages

- Write meaningful commit messages!
  - No “today’s work”
  - No “12feb2015”
  - No “bugfix”
- Write a short messages on the first line (subject)
- Detail better afterwards

The seven rules

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

Frequent, small commits

- Commits are quick and easy
- Make small but meaningful commits
- If possible split unrelated changes in multiple commits

No generated files!

Further references

- `man git-command` (e.g. `man git-log`)
- Pro Git: available online
- Github.com

License