Revision Control and GDB

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Little topics:
  - Symbolic links
  - Exit status and exit()
- Revision Control
- Meet Git (again?)
- Meet the Gnu Debugger

Simplify Your Life

- I've started posting examples and exercises on AFS
  - You can just copy them if you're working on a common platform system
  - They are under the paths like:
    /afs/eos.ncsu.edu/courses/csc/csc230/common/www/sturgill/example03
    /afs/eos.ncsu.edu/courses/csc/csc230/common/www/sturgill/exercise03
- But, you don't want to type a path like this over and over
- So, let's make a symbolic link as a shortcut

Making a Symbolic Link

- Change directory to wherever you like to work for CSC230

```
$ cd 230
```

- Then, make a symbolic link pointing to where these files live

```
$ ln -s /afs/eos.ncsu.edu/courses/csc/csc230/common/www/sturgill sturg
```

- You only have to do this once
- Then, you can use short names to copy my examples and exercises

```
$ cp sturg/example03/* .
```

Exit Status Revisited

- You already know:
  - main() has a return type of int
  - indicating program success (return 0) or failure (return non-zero).
- There are named constants to make this more read readable.

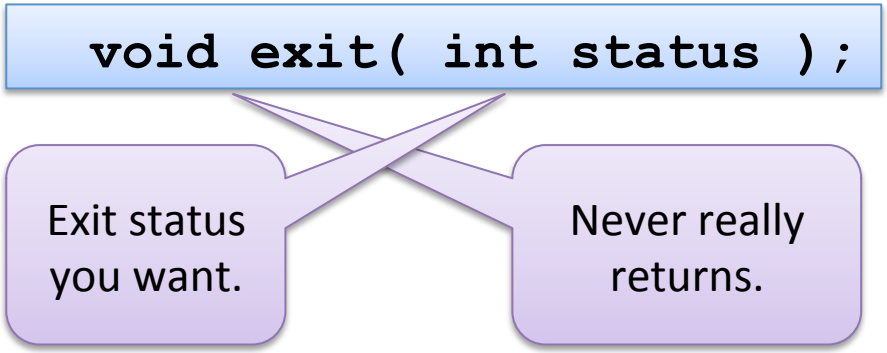
```
return EXIT_SUCCESS;
```

```
return EXIT_FAILURE;
```

- These are both defined in stdlib.h

Exit Status Revisited

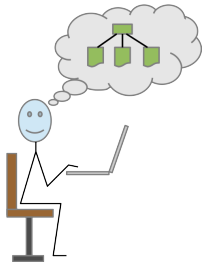
- Inside main, you can use return to exit.
- … but, this won't work from other functions
- For that, we can use the exit function.

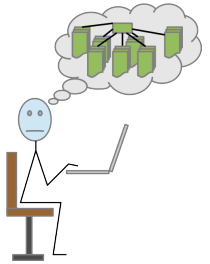


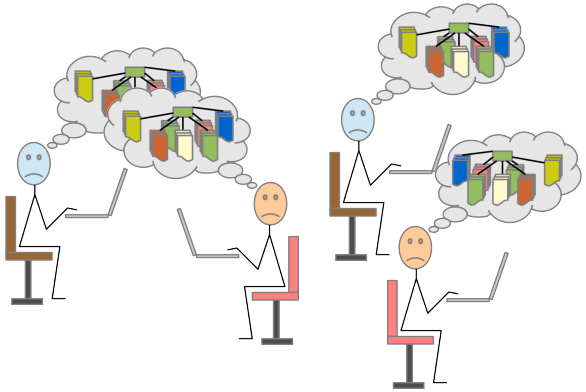
- Declared in the stdlib.h header.

Why We Need Revision Control

- How do you store your source code for a project?
  - Maybe just as a tree of files on your computer.
- That's fine, as long as you never need to …
  - Make changes to your source
  - Try out alternative versions of the same files
  - Backup your work
  - Coordinate your work with other developers

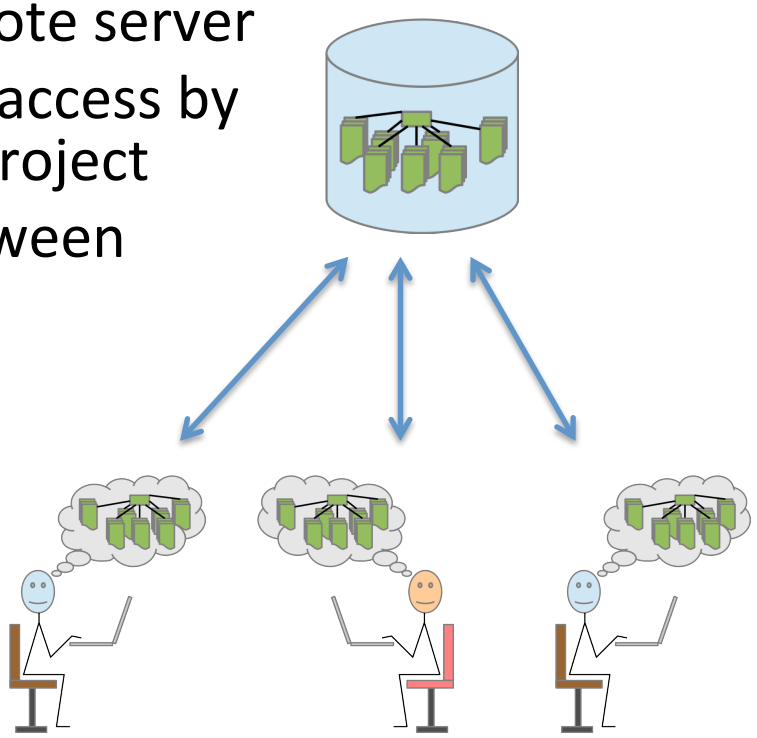






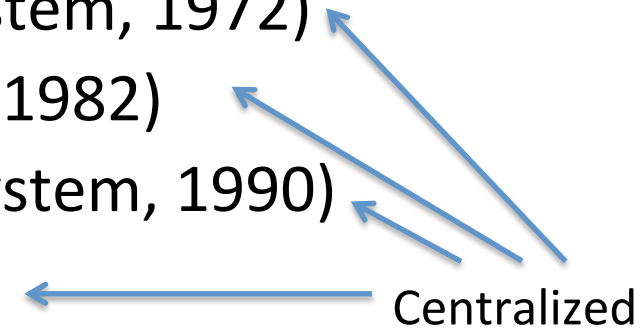
Revision Control Systems

- A Revision Control System helps with these problems
  - It maintains multiple versions of the project, including a version history
  - It can provide backup to a remote server
  - It provides convenient, secure access by many developers to a shared project
  - It provides conflict control between developers



Revision Control … Revisions

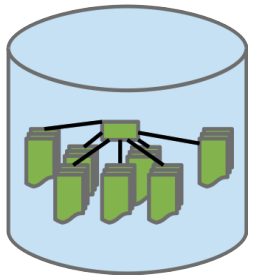
- We've developed many revision control systems
- I've used all these (and some others)
  - SCCS (Source Code Control System, 1972)
  - RCS (Revision Control System, 1982)
  - CVS (Concurrent Versioning System, 1990)
  - Subversion (Subversion, 2000)

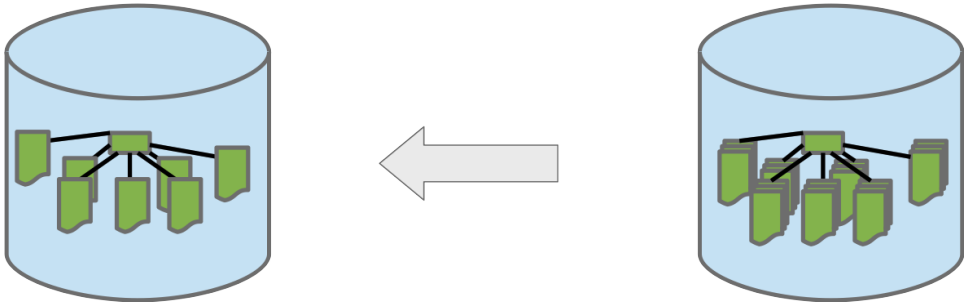


  - git and GitHub(2005)

Revision Control Basics

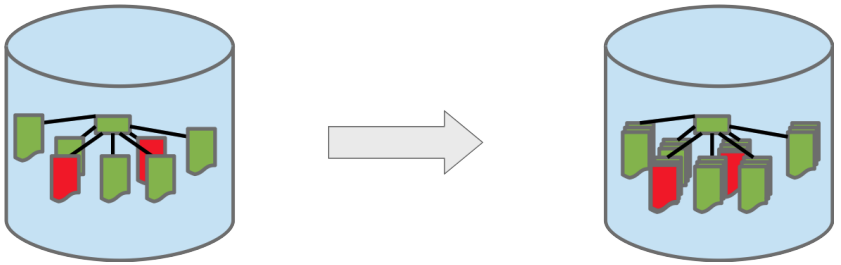
- All revision control systems have some things in common.
- They define a *repository*, where master copies of project files are maintained
  - Along with a revision history
- You can checkout a working copy from the repository
  - Typically the latest version





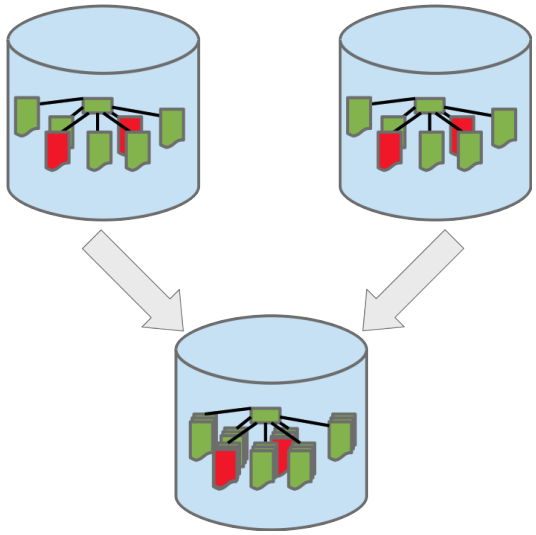
Revision Control Basics.

- There's a a mechanism to *commit* related changes all at once.
  - Collect related changes, probably across multiple files.
  - Put them into the repository all at once.
  - That's what defines a version in the repository.



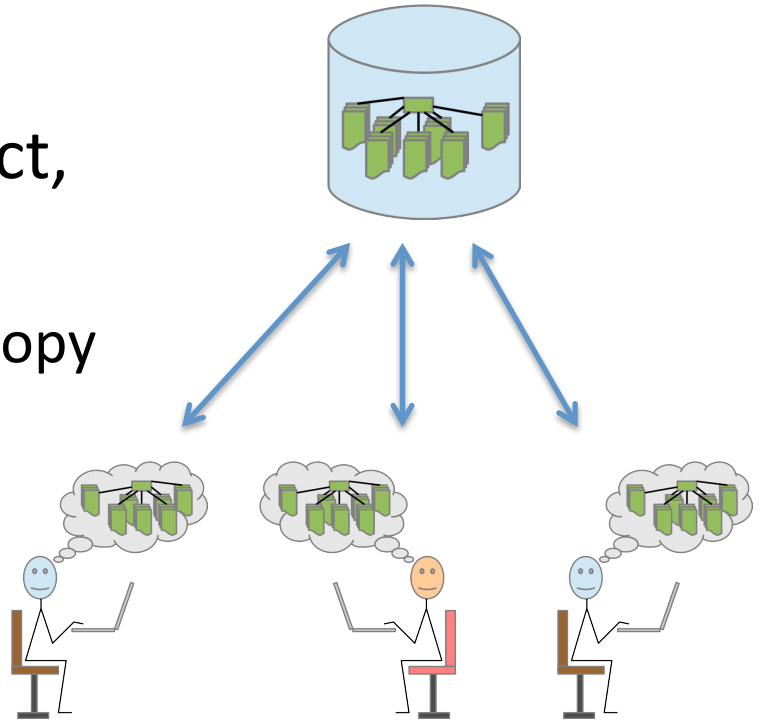
Revision Control Basics.

- There's some mechanism to coordinate concurrent edits
  - Some systems let you lock files in the repository, while you're making changes.
  - It's more common now to let developers edit files concurrently.
  - Understanding that they may have to merge their changes later.



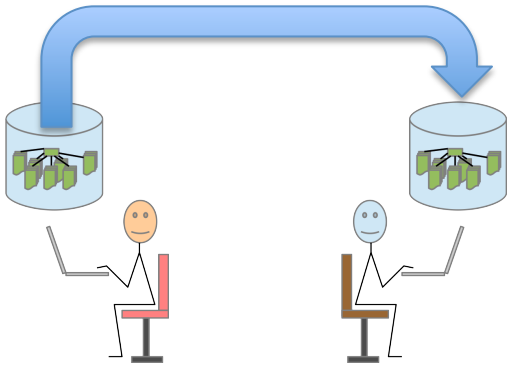
Centralized Revision Control

- Many revision control systems are centralized
  - There's a server where your project's repository lives
  - To work on a shared project, developers:
    - *Check out* a local, working copy of the project
    - *Commit* changes to the repository



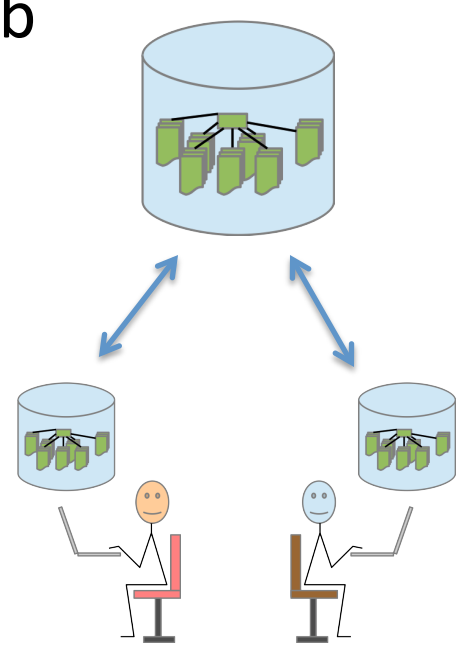
Git and GitHub

- Git is a decentralized revision control system
  - Developed by Linus Torvalds for the Linux kernel source
- Every codebase is a repository
  - Same idea, make a set of related changes, commit them
  - No central repository needed, you can commit locally
  - Every codebase maintains a revision history
- How do developers cooperate?
  - We each have our own independent repositories
  - We can *push* and *pull* changes between them …
  - then *merge* them with changes you've made locally.

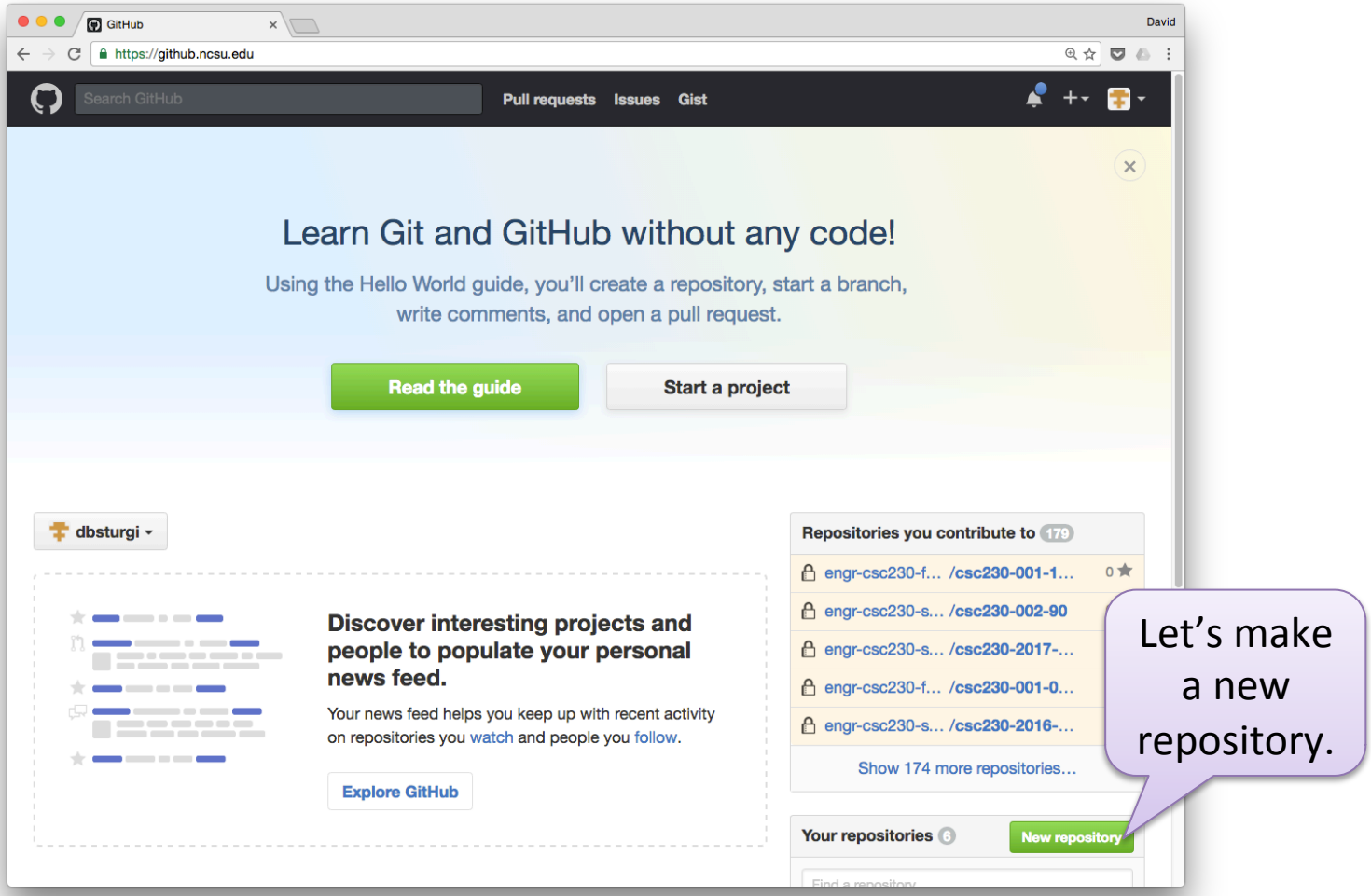


GitHub

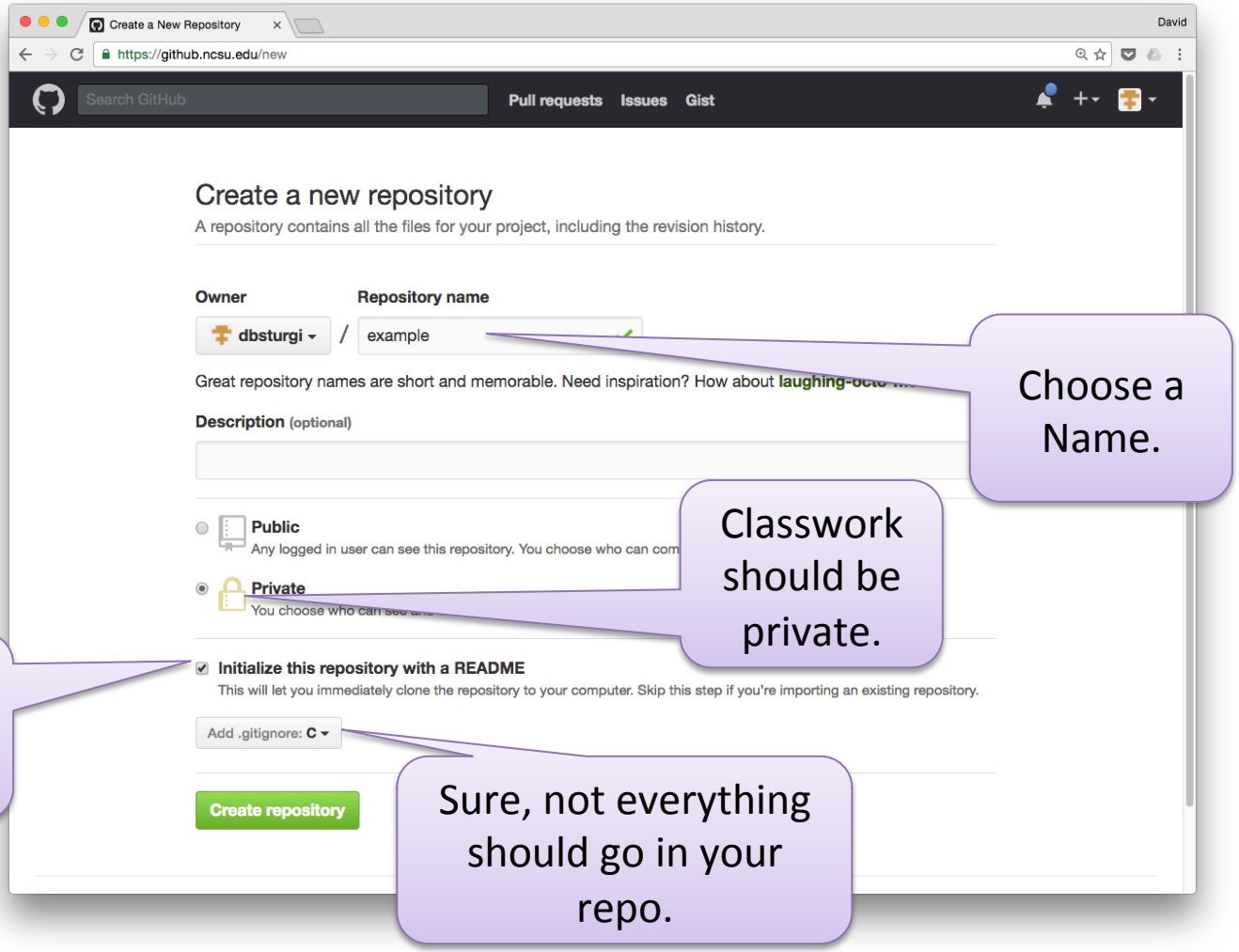
- GitHub, a service that hosts Git repositories
  - Along with a UI for managing your repositories
  - There's github.com
  - And, we have github.ncsu.edu
- We can push and pull from a repo on github
  - So, we can end up using it a lot a centralized revision control system.
- Starting from Project 2
  we will use github.ncsu.edu for assignment submission
- Let's give it a try



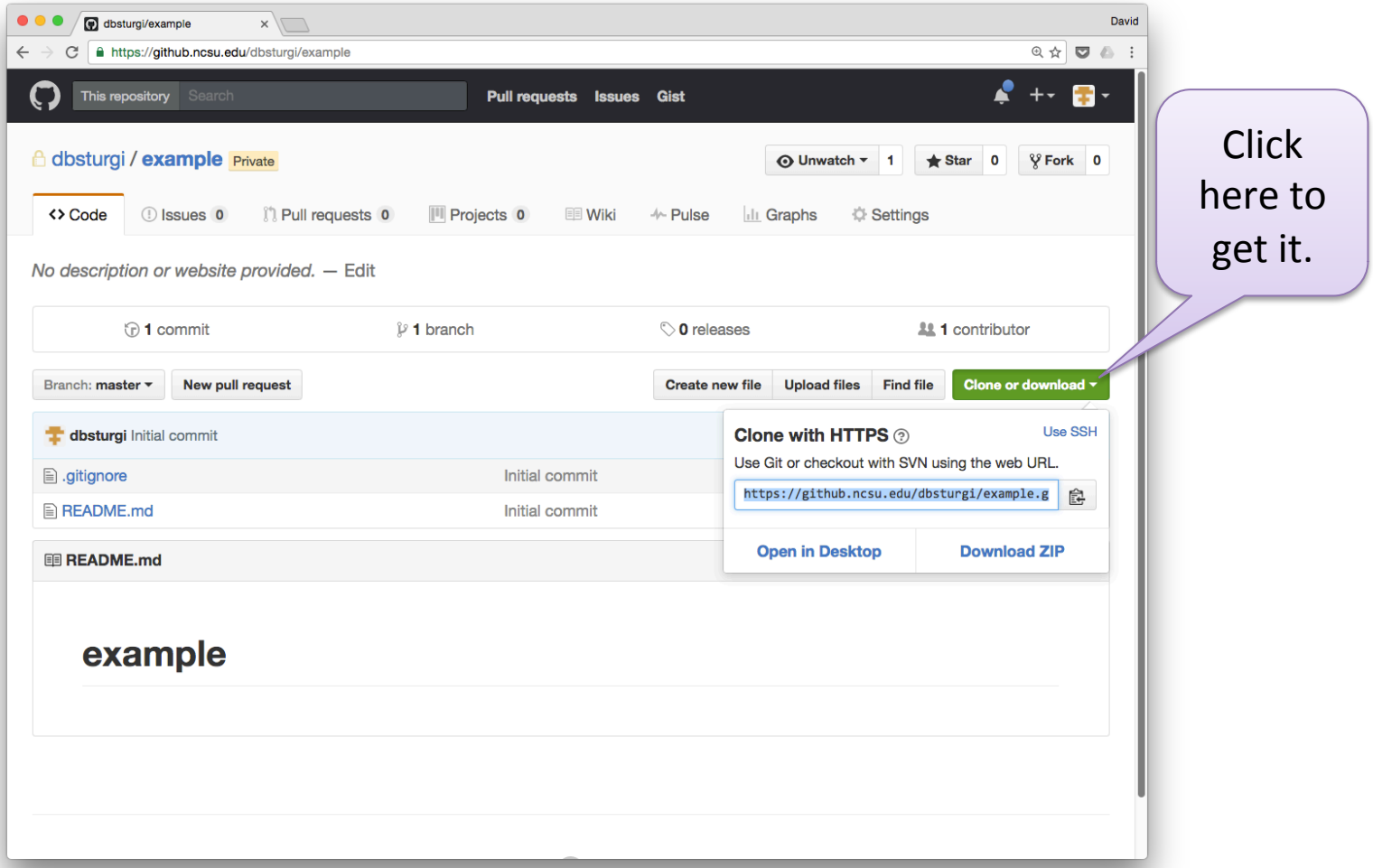
Visit github.ncsu.edu



Create a Repo

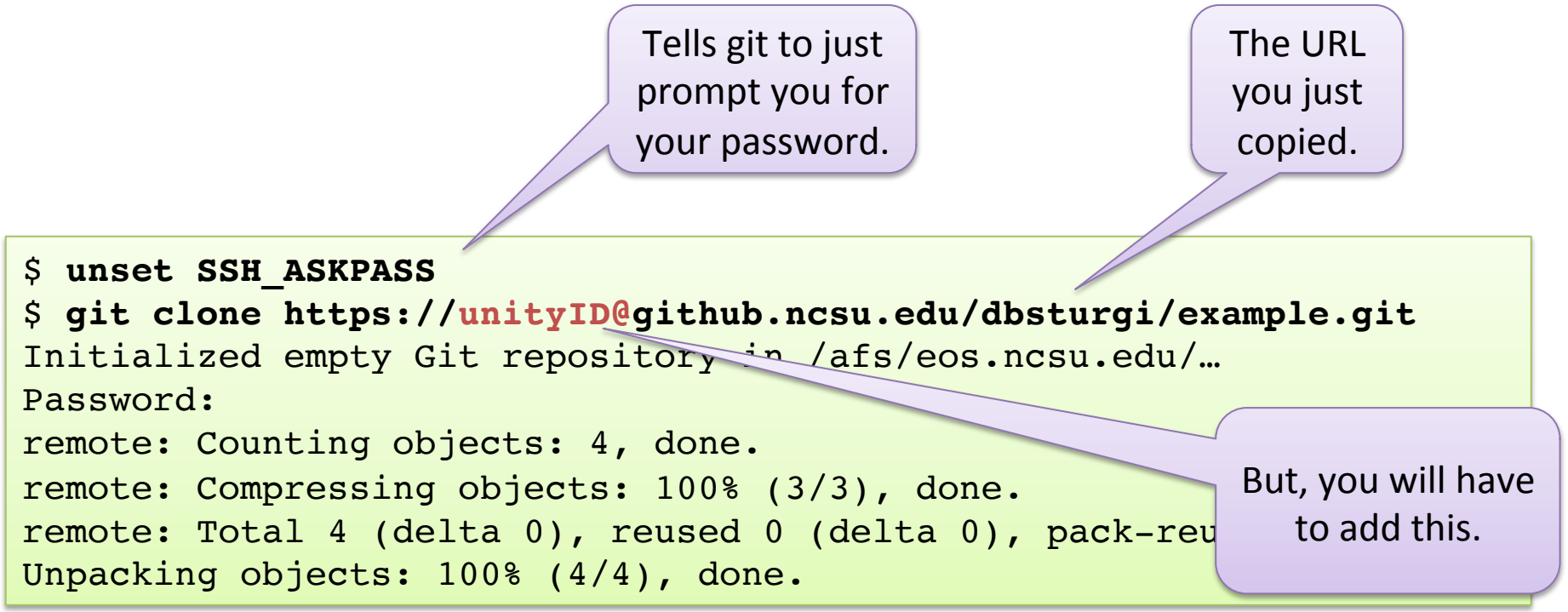


Copy its URL


dbsturgi/example
David
https://github.ncsu.edu/dbsturgi/example
This repository
Search
Pull requests
Issues
Gist
dbsturgi / example
Private
Unwatch 1
Star 0
Fork 0
Code
Issues 0
Pull requests 0
Projects 0
Wiki
Pulse
Graphs
Settings
No description or website provided. — Edit
1 commit
1 branch
0 releases
1 contributor
Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download
dbsturgi Initial commit
.gitignore
Initial commit
README.md
Initial commit
Clone with HTTPS
Use SSH
Use Git or checkout with SVN using the web URL.
https://github.ncsu.edu/dbsturgi/example.g
Open in Desktop
Download ZIP
README.md
example
Click here to get it.

Using Your New Repo

- Now, you can clone your repo from shell on a common platform system.



Creating and Editing

- Now, create and edit files, just like you normally would

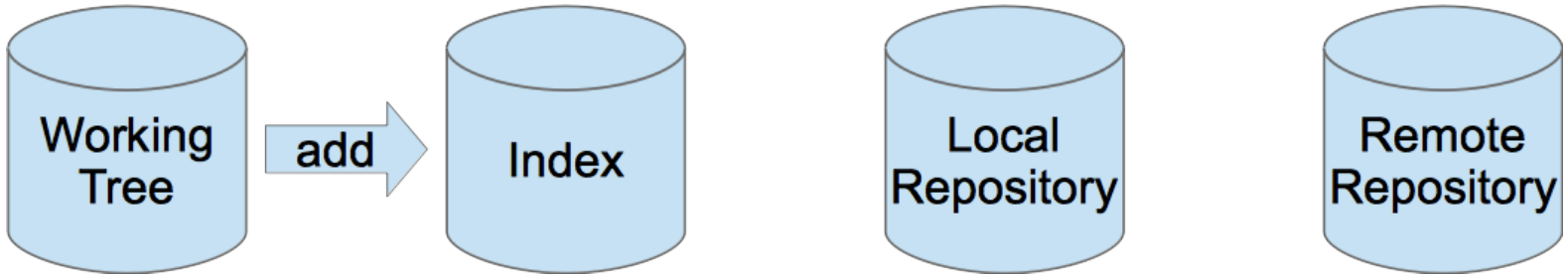
```
$ cd my-new-repo
$ nano hello.c
  ... edit ... edit ... Edit
```

Staging Files

- Tell git about any new files you create or change.

```
$ git add hello.c
```

- Really, this is staging the current version of a file to the *index*
  - The record of files for the next commit.



Working in Git

- See what's about to happen

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   hello.c
#
```

- Often, it's a good idea to check status before committing any changes.

.ignorance is Bliss

- You want to keep a clean repo
  - Without garbage or temporary files taking up space
  - There's a file size limit for the NCSU github!
- git helps with a file called .gitignore
  - It tells git files not to commit to the repo
  - Unix systems interpret this as a *hidden file*

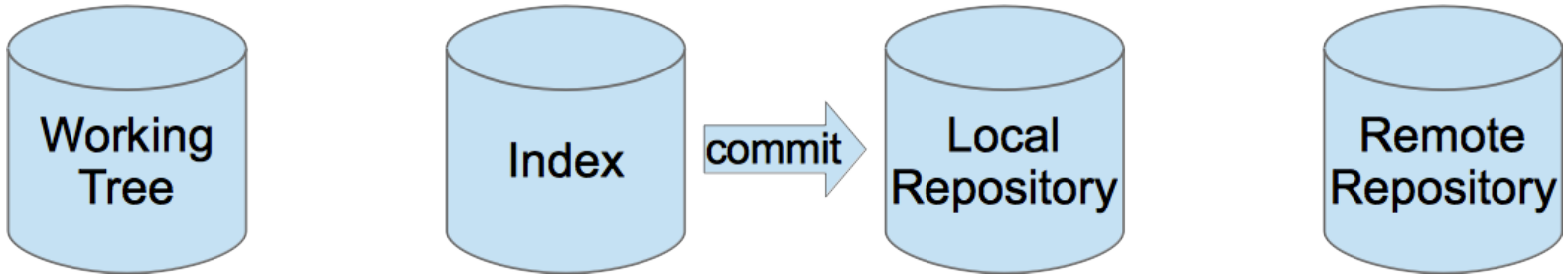
```
# ignore object files
*.o

# ignore our temporary output files
*/output.txt
```

Supports comments and a pattern matching syntax.

Working in Git

- When ready, commit to your local repo.

```
$ git commit
```



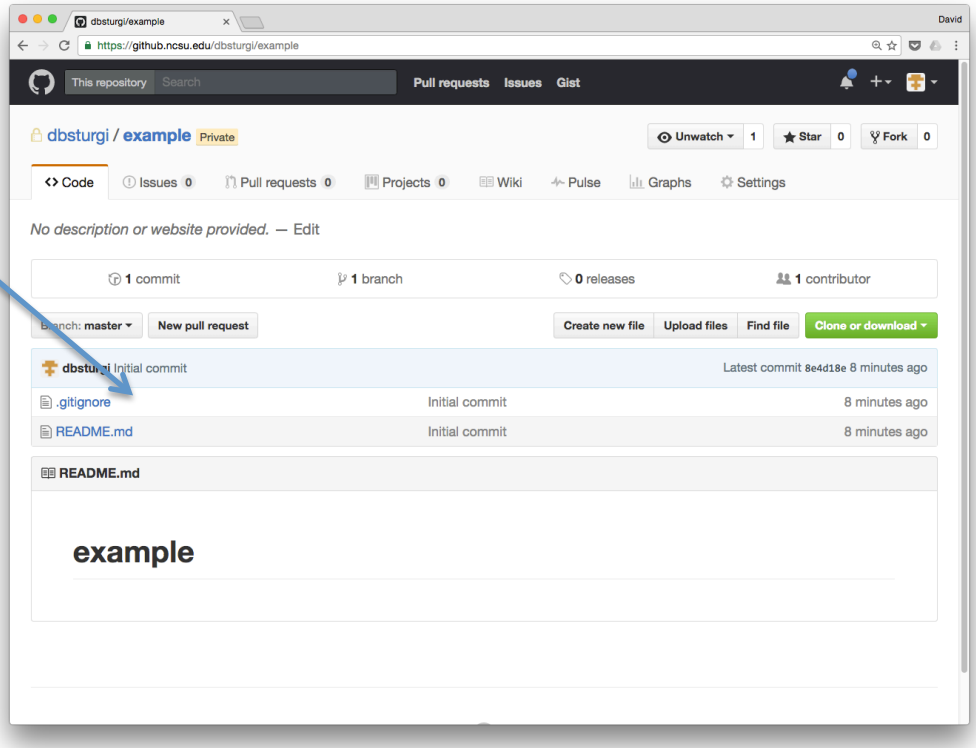
- Or, even better
  - -a : automatically add tracked files.
  - -m : here's a commit message

```
$ git commit -am "A new hello world program"
```

Working in Git

- But, your file still isn't here.
- status will tell you why.

```
$ git status
```

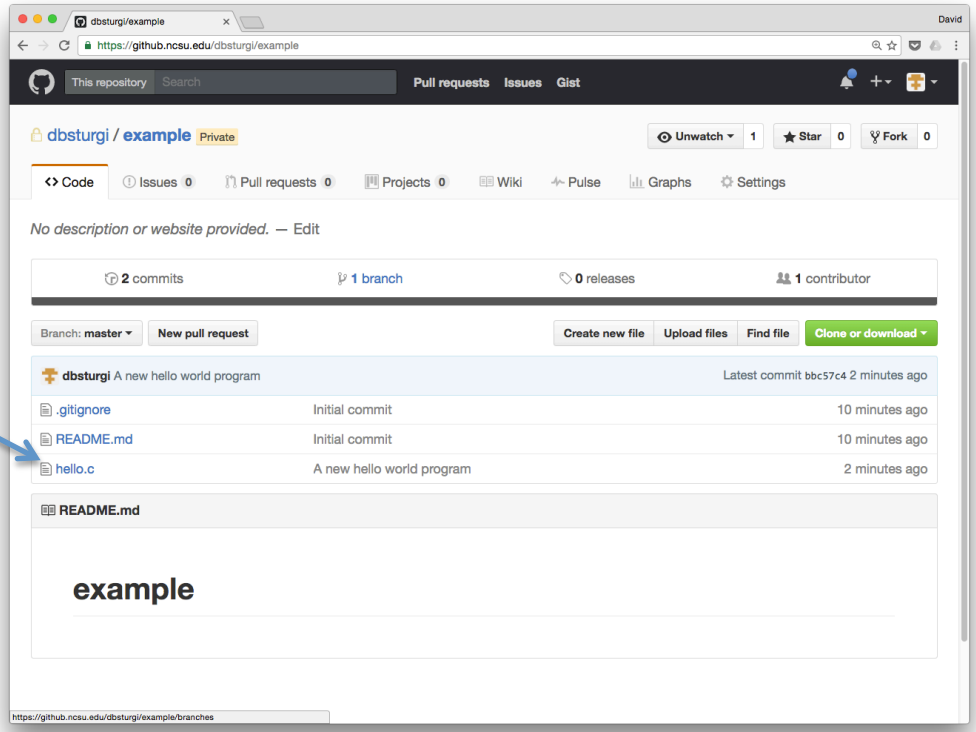


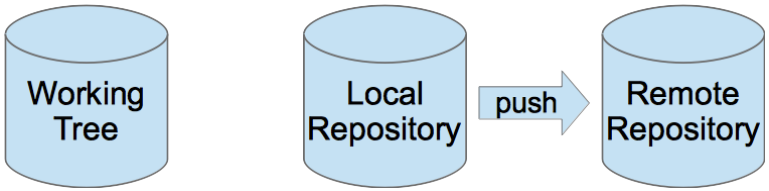
Push Your Changes

- Local commits must be pushed to a remote repo

```
$ git push
```

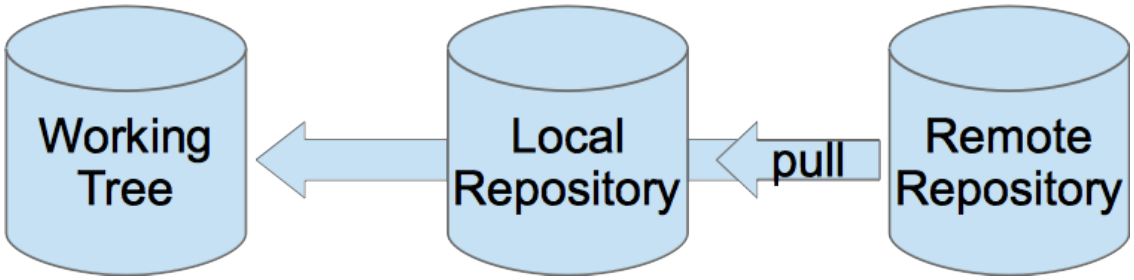
  - There it is.





Pull Updates

- Use pull to get a fresh copy of the repository



- So, you can share a remote repository
  - To share work between developers
  - Or, just copies of your code in more than one place.

Pull Updates

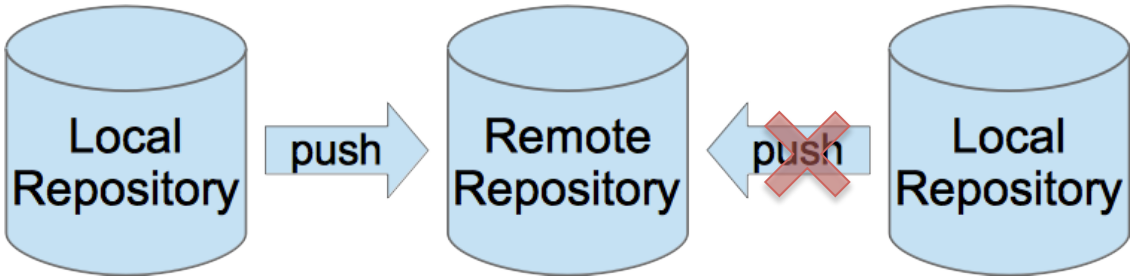
- There's a command for that:

```
$ git pull
Password:
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.ncsu.edu/dbsturgi/example
   bbc57c4..dedbf3f  master     -> origin/master
Updating bbc57c4..dedbf3f
Fast-forward
 hello.c |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Conflicting Edits

- Conflicting edits may be committed to different copies of the repo.
  - Git doesn't want to clobber changes during a push
  - So. It will refuse to push if your local repo isn't completely up to date.

```
$ git push
To https://dbsturgi@github.ncsu.edu/dbsturgi/example.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'https://dbsturgi@github.ncsu....
```



Merging Edits

- Issuing a pull will try to merge the edits.

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.ncsu.edu/dbsturgi/example
   907f08a..4221f31  master     -> origin/master
Auto-merging hello.c
CONFLICT (content): Merge conflict in hello.c
Automatic merge failed; fix conflicts and then commit the result.
```

git will try to merge text files if it can.

… but it may need help from you.

Merging Edits

- You may have to edit your source files … to decide what parts of the code to keep.

```
…
<<<<<<< HEAD
  printf( "hello world!\n" );
=======
  printf( "Hello World.\n" );
>>>>>>> 1b27c3….
…
```

Local version of your code.

Version you just pulled from the remote repo.

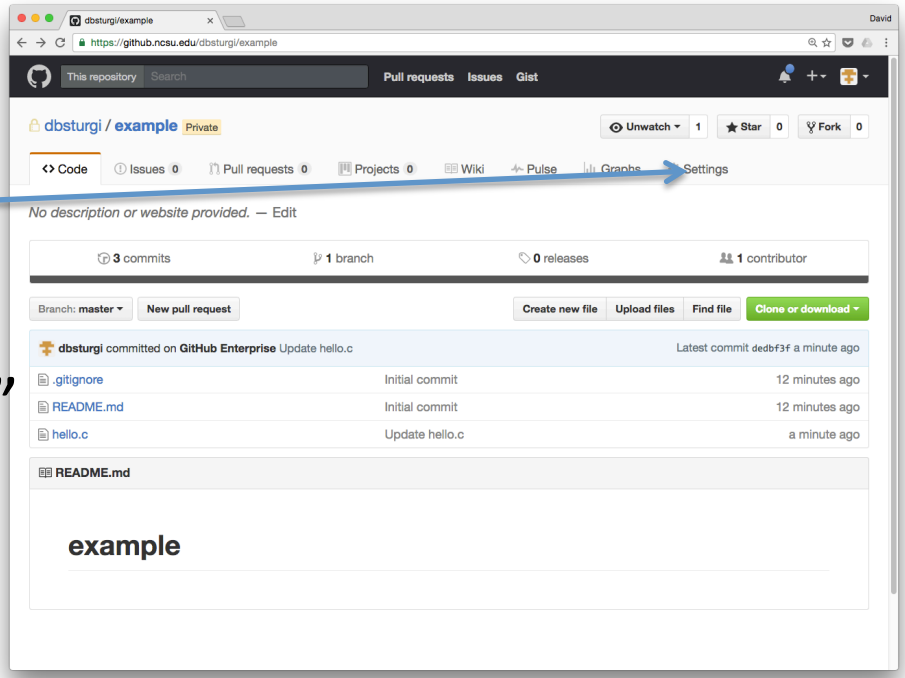
- Edit the file, to make it look the way you want.

Merging Edits

- Now, you have a new version of the project.
- You'll need to:
  - add
  - commit
  - push

Cleaning Up

- If you worked along with this exercise, you can delete your repository when you're done
  - Of course, you'll normally keep a repository around
  - Choose settings.
  - Then, choose "Delete this Repository" on the next page.



Using Git on Projects

- For future homework assignments
  - You will use the NCSU github
  - We'll make a new repo for you.
  - A push to your repo is like a submission
- When you complete a project, you can check the web interface to make sure your changes made it.

Git Workflow

- Clone your repository once (or for each system you plan to work on)

```
$ git clone url-for-my-repo
```

- Repeat
  - Pull a fresh copy of the repository

```
$ git pull
```

If anyone else might be making changes.

  - Edit, edit, edit
  - Add changes to the index.

```
$ git add some-new-files-I-just-added
```

  - Commit the changes

```
$ git commit –am "a message about what changed"
```

  - Push the changes

```
$ git push
```

Meet GDB

- GDB : the GNU debugger
  - Debugger for C, C++ and other programs compiled with the GNU compiler suite
- GDB will let you
  - Run and resume your program's execution
  - Set breakpoints at particular source lines
  - Examine the program state
  - Change things in the program.

GDB and the IDE

- gdb is a *symbolic debugger*.
  - It lets us work with the names in our source code … instead of just memory addresses.
- Like the compiler, it's the kind of tool that's normally hidden inside a user-friendly IDE
  - e.g., IDEs like Eclipse can provide a GUI for gdb
- It's also a tool we can use directly if we're working from the command line

Preparing for GDB

- To get the most out of gdb, you need to use the -g flag when you compile.
  - This tells the compiler to include symbol names and extra information not strictly necessary for running your program.
  - But, gdb can use this information to show you more of what's going on inside your program.
- So, we'll want to compile like this:

```
$ gcc -Wall -std=c99 -g myProgram.c -o myProgram
```

GDB Demonstration

- In class, we'll try out gdb using a small collection of bad programs I wrote.
  - primes1.c : is supposed to report prime numbers … but it crashes.
  - primes2.c : is supposed to report prime numbers … but it gets stuck in an infinite loop.
  - primes3.c : is supposed to report prime numbers … but it only gets a couple of them.
- The next few slides show what we're supposed to learn from working with gdb

Running with GDB

- To debug our program, we'll need to start gdb and tell it to work with our program:

```
$ gdb ./myProgram
```

  - You can also attach gdb to a program that's already running (handy for debugging a program you didn't expect to debug)

```
$ gdb ./myProgram processID
```

- Then, you can set breakpoints by function name.

```
(gdb) break someFunctionName
```

- Or by line number:

```
(gdb) break 25
```

Running with GDB

- When you're ready, you can let your program run:

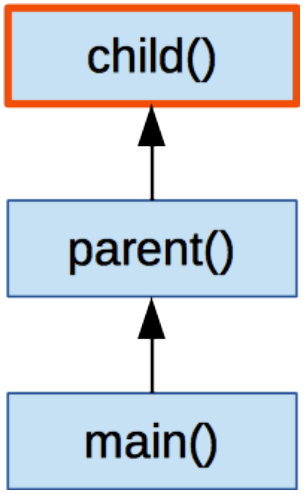
```
(gdb) run
```

- If you need to do I/O redirection, this is where you do it.

```
(gdb) run < input-file.txt > output-file.txt
```

Looking Around with GDB

- Then, gdb will run until our program
  - terminates
  - or until it crashes
  - or until it hits a breakpoint.
- Then, you'll get the gdb prompt back.
- You can look at a *stack trace*, a list of the function calls that got the program to this point:




```
(gdb) backtrace
#0 child (x=3,y=99)
#1 parent (param=7) at …
#2 main () at …
```

Looking Around with GDB

- You can look at the values of local variables:

```
(gdb) print i
```

- Or, really, any expression:

```
(gdb) print i * n + j
```

- You can even change the value of a variable:

```
(gdb) set variable n = 25
```

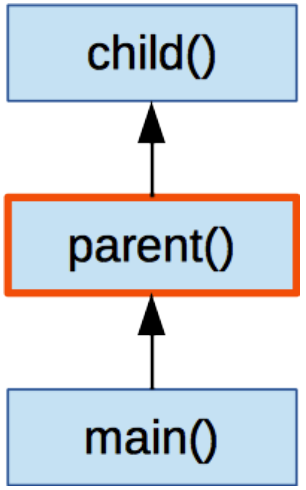
Looking Around with GDB

- You can move up a stack frame
  - To look at variable values in the caller

```
(gdb) up
```

  - This doesn't end the current function … you're just looking around
- You can move both directions in the stack trace

```
(gdb) down
```



Looking Around with GDB

- GDB only shows a little bit of source code at a time.
- You can look at surrounding source lines:

```
(gdb) list
```

- Or, code for any function you want.

```
(gdb) list someFunctionName
```

- Or, if you run gdb with the -tui option, it will try to show your source code in part of the terminal window.

```
$ gdb -tui ./myProgram
```

Resuming Execution

- You can resume running your program
  - Until it terminates, hits a breakpoint, etc.

```
(gdb) continue
```

- You can run to just the next instruction (stepping over any function calls)

```
(gdb) next
```

Resuming Execution

- You can step ahead, going into any function calls.:

```
(gdb) step
```

- You can run ahead to any line of code you want:

```
(gdb) until 46
```

- Or, run until the current function returns:

```
(gdb) finish
```

Wrapping Up

- While you're looking at your program's state, it's frozen in place.
- You can kill it when you're done:

```
(gdb) kill
```

- And, maybe re-run it later if you want.
- If you have a makefile (later), you can even re-build from the gdb prompt.

```
(gdb) make
```

Wrapping Up

- When you're done, you can quit gdb to get back to the shell

```
(gdb) quit
```

GDB Command Summary

- While your program is stopped, you can issue commands to gdb.

Command	abbrev	
break *line*	b	Set a breakpoint at the given line
break *sourcefile:line*	b	Set a breakpoint in a particular file
break *functionName*	b	Set a breakpoint at a given function
break *line* if *condition*	b	Only break if condition evaluates to true
run	r	Start running the program
run arg1 args2 < file.txt	r	Run with arguments and redirection
continue	c	Continue running the program

GDB Command Summary

- While your program is stopped, you can issue commands to gdb.

Command	abbrev	
next	n	Step over function calls to next statement
step	s	Single-step to next statement, entering function calls.
until *line-number*		Run until the given line number
finish		Finish running the current function

GDB Commands

- While your program is stopped, you can examine and change its state.

Command	abbrev	
print *expr*	p	Evaluate and print the given expression
set variable *var* = *expr*		Set the given *var* to the the value of *expr*
list	l	Show surrounding source file lines
list functionName	l	Show code around the given function
list lineNumber	l	Show code around the given line number

More GDB Commands

Command	abbrev	
backtrace	bt	Show a stack trace
up		Move up one stack frame
down		Move down one stack frame
delete *n*	d	Delete breakpoint *n*
ctrl-C		Break the currently running program
condition *breakpoint condition*		Make a breakpoint conditional
kill	k	Kill the program being debugged
quit	q	Exit gdb