



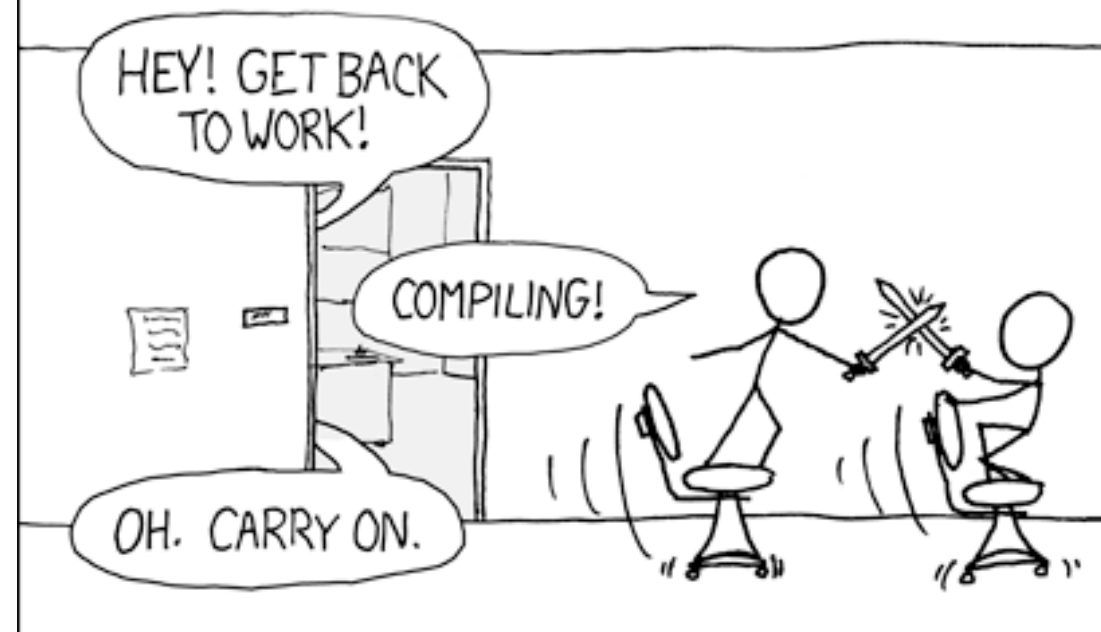
Makefiles & Git

Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- Compilation
- Makefiles
- Revision Control
 - Git

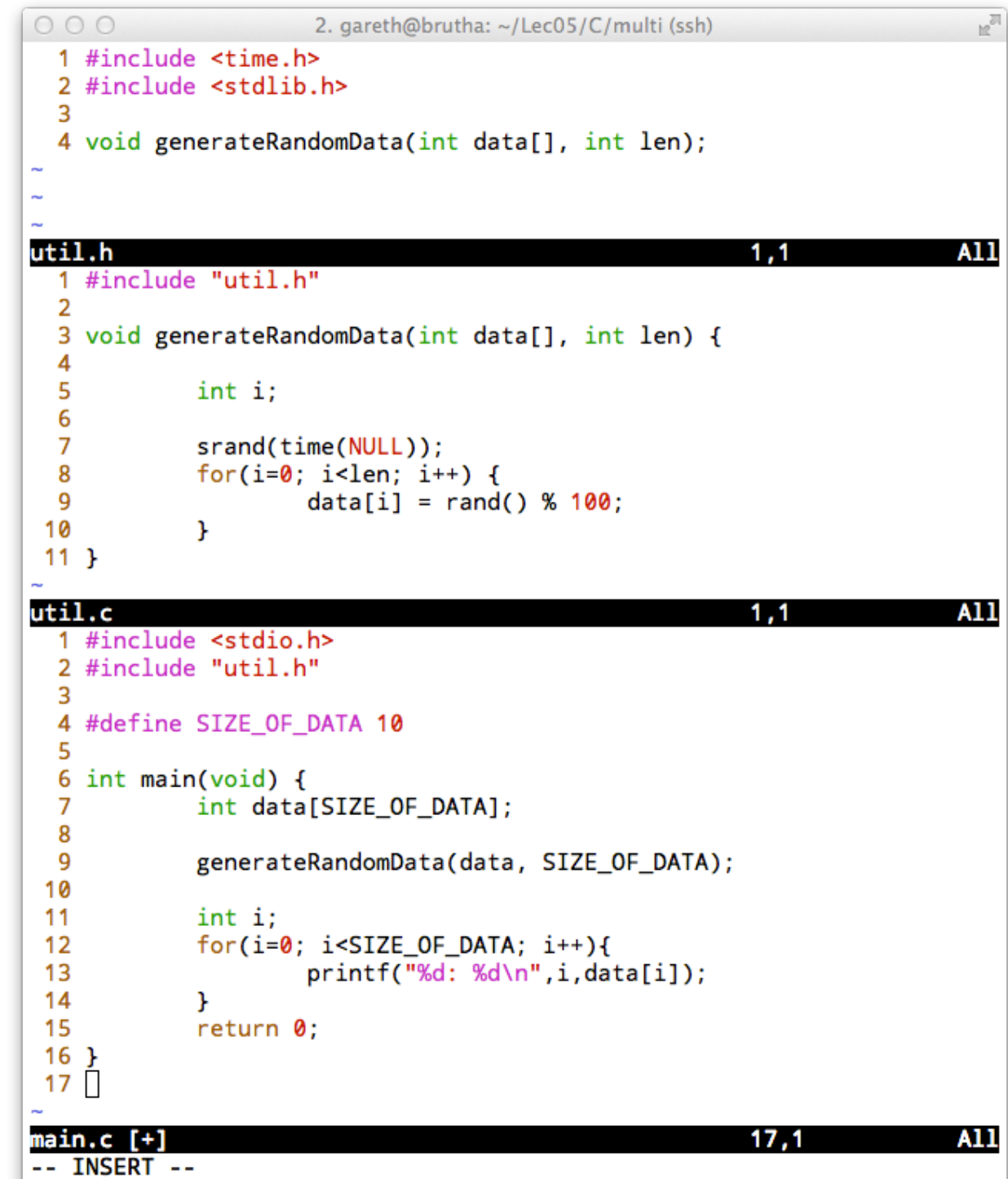
- Compilation
- Makefiles
- Revision Control
 - Git

THE #1 PROGRAMMER EXCUSE
FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."



Simple Example

- Last lecture we created a simple example project. It generated an array of random integers and wrote the contents of the array to screen.
- Three source files:
 - **main.c** - creates an array of integers, calls a function to generate random data and prints it to the screen.
 - **util.h** - includes **time.h** and **stdlib.h** and has a function prototype.
 - **util.c** - has a function that takes an array and fills it with a series of random numbers.



```
2. gareth@brutha: ~/Lec05/C/multi (ssh)
1 #include <time.h>
2 #include <stdlib.h>
3
4 void generateRandomData(int data[], int len);
~
util.h 1,1 All
1 #include "util.h"
2
3 void generateRandomData(int data[], int len) {
4
5     int i;
6
7     srand(time(NULL));
8     for(i=0; i<len; i++) {
9         data[i] = rand() % 100;
10    }
11 }
~
util.c 1,1 All
1 #include <stdio.h>
2 #include "util.h"
3
4 #define SIZE_OF_DATA 10
5
6 int main(void) {
7     int data[SIZE_OF_DATA];
8
9     generateRandomData(data, SIZE_OF_DATA);
10
11     int i;
12     for(i=0; i<SIZE_OF_DATA; i++){
13         printf("%d: %d\n",i,data[i]);
14     }
15     return 0;
16 }
17
main.c [+] 17,1 All
-- INSERT --
```

Compiling the Example

- We can compile all the code at once by doing:
 - **gcc -o myprog main.c util.c**
- Compiling the code this way is simple but it means we need to recompile **all** the code each time we build our program (on large projects this can be slow).
- Instead we can build in stages. To compile **util.c** into an object file:
 - **gcc -c util.c**
- Compile **main.c** into an object file:
 - **gcc -c main.c**
- Create the executable **myprog** by linking the object files together:
 - **gcc -o myprog main.o util.o**
- This has the benefit that when one file is changed only that file needs rebuilt into an object file, and the executable need remade.

```
1. gareth@brutha: ~/Lec06/multi (ssh)
gareth@brutha:~/Lec06/multi$ ls
main.c  util.c  util.h
gareth@brutha:~/Lec06/multi$ gcc -c util.c
gareth@brutha:~/Lec06/multi$ ls
main.c  util.c  util.h  util.o
gareth@brutha:~/Lec06/multi$ gcc -c main.c
gareth@brutha:~/Lec06/multi$ ls
main.c  main.o  util.c  util.h  util.o
gareth@brutha:~/Lec06/multi$ gcc -o myprog main.o util.o
gareth@brutha:~/Lec06/multi$ ls
main.c  main.o  myprog  util.c  util.h  util.o
gareth@brutha:~/Lec06/multi$ ./myprog
0: 92
1: 69
2: 62
3: 96
4: 47
5: 16
6: 93
7: 11
8: 32
9: 57
gareth@brutha:~/Lec06/multi$
```

Compiling the Example

- The downside of incremental builds is the programmer is required to keep track of all of the files that change. When code bases grow to 1000's of files this can be a problem
- Importantly we need to ensure no stale object files are linked into the main program.
- Manually writing compile lines can be a pain, and it's often easy to forget to link libraries, use the same compilation flags etc.
- Using a script doesn't help us here as we want to ensure dependancies and make sure only the code that needs to be built is built - it's possible in bash but the script will get complicated quickly.
- We need a solution that does all of these things for us.

```
1. gareth@brutha: ~/Lec06/multi (ssh)
gareth@brutha:~/Lec06/multi$ ls
main.c  util.c  util.h
gareth@brutha:~/Lec06/multi$ gcc -c util.c
gareth@brutha:~/Lec06/multi$ ls
main.c  util.c  util.h  util.o
gareth@brutha:~/Lec06/multi$ gcc -c main.c
gareth@brutha:~/Lec06/multi$ ls
main.c  main.o  util.c  util.h  util.o
gareth@brutha:~/Lec06/multi$ gcc -o myprog main.o util.o
gareth@brutha:~/Lec06/multi$ ls
main.c  main.o  myprog  util.c  util.h  util.o
gareth@brutha:~/Lec06/multi$ ./myprog
0: 92
1: 69
2: 62
3: 96
4: 47
5: 16
6: 93
7: 11
8: 32
9: 57
gareth@brutha:~/Lec06/multi$
```

- Compilation
- Makefiles
- Revision Control
 - Git

Makefiles

- Makefiles are a way of automating the build process without losing the flexibility of only building object files.
- A **Makefile** can be thought of as a set of rules that explain how a piece of code is to be built.
- **make**, the command, then runs through each rule required to build a target ensuring they are met.
- By default make will look for a file called **makefile** or **Makefile**. You can specify a different file by:
 - **make -f myMakefile**
- Additionally you can run a specific rule by specifying a target i.e. **make target**
- Using Makefiles takes care off:
 - Incremental builds
 - Dependency tracking
 - Cleaning build files
 - Ensuring clean and consistent builds
 - Release vs. Production etc.

[illegible]

Make Rules

file to create

what files are required to create it

```
main: main.c
```

gcc -o main main.c

required tab

command to create file

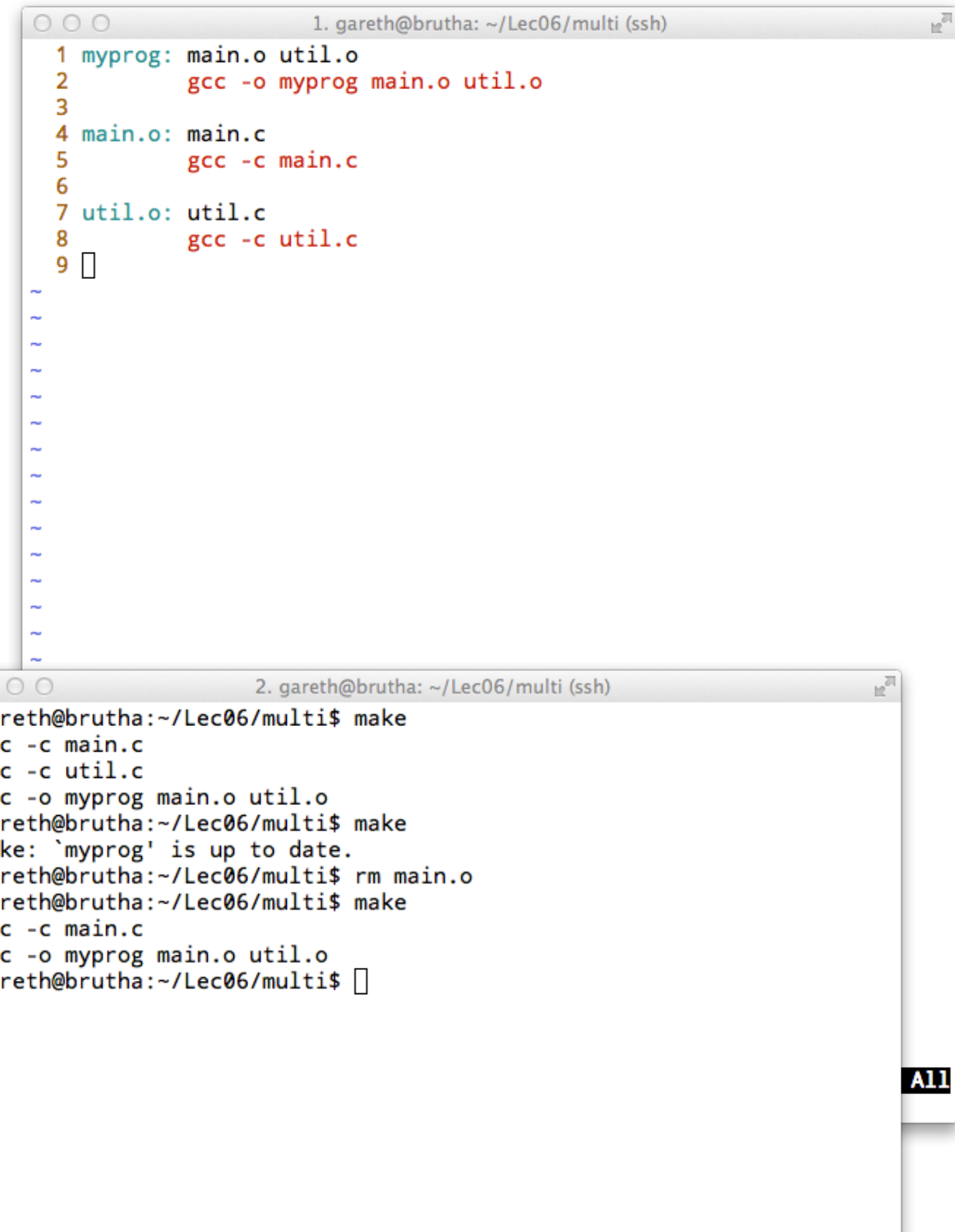
- Makefile rules require a **target**, it's **dependencies** and the **command** needed to produce the target from it's dependencies. In this instance `main` is produced from `main.c` by running `gcc`.
- Makefiles can be used for a number of things, not just compiling. For instance the following rule downloads the xkcd comic you saw before:

`compiling.png:`

```
wget http://imgs.xkcd.com/comics/compiling.png
```

Simple Example

- Our simple program consisted of **main.c**, **util.c** and **util.h**
- Our simple Makefile consists of three rules:
 - A rule to make **util.o** from **util.c**
 - A rule to make **main.o** from **main.c**
 - A rule to make **myprog** from **main.o** and **util.o**
- We can compile our code by running **make** with no parameters. Make will try and ensure the first target in it's list exists (in this case **myprog**).
- If we run it a second time make tells us there is nothing to be done, and **myprog** is up to date.
- If we delete **main.o** and rerun **make**, **main.o** is rebuilt along with **myprog** but **util.o** is left untouched.



The image shows two terminal windows. The top window, titled '1. gareth@brutha: ~/Lec06/multi (ssh)', displays the contents of a Makefile. The bottom window, titled '2. gareth@brutha: ~/Lec06/multi (ssh)', shows the output of running 'make' and 'rm main.o' followed by 'make' again.

```
1 myprog: main.o util.o
2     gcc -o myprog main.o util.o
3
4 main.o: main.c
5     gcc -c main.c
6
7 util.o: util.c
8     gcc -c util.c
9
```

```
gareth@brutha:~/Lec06/multi$ make
gcc -c main.c
gcc -c util.c
gcc -o myprog main.o util.o
gareth@brutha:~/Lec06/multi$ make
make: `myprog' is up to date.
gareth@brutha:~/Lec06/multi$ rm main.o
gareth@brutha:~/Lec06/multi$ make
gcc -c main.c
gcc -o myprog main.o util.o
gareth@brutha:~/Lec06/multi$
```

Variables

- Just like with a shell script you can assign variables which are expanded when executing commands
- Variable assignment works in the same way as Bash:
 - **VAR=value**
- Referencing the variable again is similar to Bash:
 - **\${VAR}**
 - **\$(VAR)**
- Variables are useful so that compile options can be consistent throughout the build. In our example we use **CC** to represent the compiler, **CFLAGS** to represent gcc compiler flags and **LDFLAGS** to represent linker flags (like loading math libraries **-lm**)

```
1. gareth@brutha: ~/Lec06/multi/vars (ssh)
1 # Comments in Make can look like this
2 CC=gcc
3 CFLAGS=-O2
4 LDFLAGS=-lm
5
6 # Debug flags, don't use for release
7 #CFLAGS=-g
8
9 myprog: main.o util.o
10     $(CC) $(LDFLAGS) -o myprog main.o util.o
11
12 main.o: main.c
13     $(CC) $(CFLAGS) -c main.c
14
15 util.o: util.c
16     $(CC) $(CFLAGS) -c util.c
17
~
~
~
~
```

```
2. gareth@brutha: ~/Lec06/multi/vars (ssh)
gareth@brutha:~/Lec06/multi/vars$ make
gcc -O2 -c main.c
gcc -O2 -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/vars$
```

Targets and Phony

- Targets in Makefiles are usually files, for instance **myprog** or **main.c**
- By default **make** attempts to create the first target in a Makefile. For each dependency required **make** checks that these files exist and if not attempts to carry out the rule needed to create it.
- We can build a specific a different target by:
 - **make <target>**
 - **make myprog**
- Sometimes we want targets that don't create files. For instance a “**clean**” target that removes all of the build files so we can rebuild the whole project.
- Problems occur if there was a file present named the same as one of these targets as the rule would never run. For instance “**make clean**” would never remove the build files.
- For these instances we can use a **.PHONY** target which means the target will be run whenever specified.

```
1. gareth@brutha: ~/Lec06/multi/vars (ssh)
1 # Comments in Make can look like this
2 CC=gcc
3 CFLAGS=-O2
4 LDFLAGS=-lm
5 EXE=myprog
6
7 # Debug flags, don't use for release
8 #CFLAGS=-g
9
10 $(EXE): main.o util.o
11     $(CC) $(LDFLAGS) -o $(EXE) main.o util.o
12
13 main.o: main.c
14     $(CC) $(CFLAGS) -c main.c
15
16 util.o: util.c
17     $(CC) $(CFLAGS) -c util.c
18
19
20 .PHONY: clean
21 clean:
22     rm *.o
23     rm $(EXE)
```

```
2. gareth@brutha: ~/Lec06/multi/vars (ssh)
gareth@brutha:~/Lec06/multi/vars$ ls
main.c main.o Makefile myprog util.c util.h util.o
gareth@brutha:~/Lec06/multi/vars$ make clean
rm *.o
rm myprog
gareth@brutha:~/Lec06/multi/vars$ ls
main.c Makefile util.c util.h
gareth@brutha:~/Lec06/multi/vars$ touch clean
gareth@brutha:~/Lec06/multi/vars$ make
gcc -O2 -c main.c
gcc -O2 -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/vars$ make clean
rm *.o
rm myprog
gareth@brutha:~/Lec06/multi/vars$ ls
clean main.c Makefile util.c util.h
gareth@brutha:~/Lec06/multi/vars$ rm clean
gareth@brutha:~/Lec06/multi/vars$
```

All

Automatic Variables

- Make has a number of builtin automatic variables that help to make our Makefiles simpler.
 - `%` is analogous to `*` in a shell script. It matches anything but with a twist. It will not match NULL and it temporarily knows what it matched. Hence: `%.o: %.c` means “any file ending in `.o` depends upon a similarly-named `.c` file.”
 - `$@` is the name of the target
 - `$<` is the name of the first prerequisite
 - `^` is the list of prerequisites for the target you are compiling...more-or-less.
 - `?` is the list of prerequisites that are newer than the target and therefore must be compiled first.
- See: <http://www.gnu.org/software/automake/manual/make/> for a full manual...

```
1. gareth@brutha: ~/Lec06/multi/autovars (ssh)
1 # Comments in Make can look like this
2 CC=gcc
3 EXE=myprog
4
5 CFLAGS=-O2
6 LDFLAGS=-lm
7 OBJ=main.o util.o
8
9 # Build the EXE specified by $(EXE)
10 $(EXE): $(OBJ)
11     $(CC) $(LDFLAGS) -o $@ $^
12
13 #Build all source files into object files
14 %.o:%.c
15     $(CC) $(CFLAGS) -c $<
16
17
18 .PHONY: clean test
19 # Clean all build files
20 clean:
21     rm *.o
22     rm $(EXE)
23
24 #Run the executable
25 test:
26     ./$(EXE)
```

```
2. gareth@brutha: ~/Lec06/multi/autovars (ssh)
gareth@brutha:~/Lec06/multi/autovars$ make
gcc -O2 -c main.c
gcc -O2 -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/autovars$ make test
./myprog
0: 49
1: 85
2: 52
3: 51
4: 25
5: 80
6: 47
7: 19
8: 14
9: 78
gareth@brutha:~/Lec06/multi/autovars$
```


More Advanced Make

- Make allows targets to reference one another as dependencies.
- This can be used to build up more complicated interactions.
- For instance say you want to be able to use the same makefile to build an optimised build, or a debug build (or a release build).
- You could create targets that modified the flags passed to the build step so that different compiler options were set.
- The example Makefile add the contents of **OPTCFLAGS** to **CFLAGS** if the build is optimised and **DEBUGCFLAGS** to **CFLAGS** if the build is a debug.

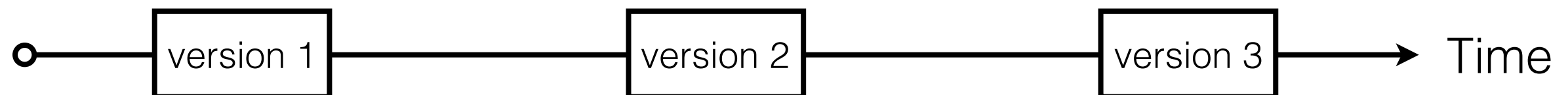
```
1. gareth@brutha: ~/Lec06/multi/advanced (ssh)
1 # Comments in Make can look like this
2 CC=gcc
3 EXE=myprog
4
5 CFLAGS=-Wall
6 LDFLAGS=-lm
7
8 OPTCFLAGS=-O2
9 DEBUGCFLAGS=-g
10
11 OBJ=main.o util.o
12
13 # Default Target, dependency on $(EXE) target
14 all: $(EXE)
15
16 # Optimised target, add OPTCFLAGS
17 opt: CFLAGS+=$(OPTCFLAGS)
18 opt: $(EXE)
19
20 # Debug target, add DEBUG Flags
21 debug: CFLAGS+=$(DEBUGCFLAGS)
22 debug: $(EXE)
23
24 $(EXE): $(OBJ)
25         $(CC) $(LDFLAGS) -o $@ $^
26
27 %.o:%.c
28         $(CC) $(CFLAGS) -c $<
29
30 .PHONY: clean test
31 clean:
32         rm *.o
33         rm $(EXE)
```

```
2. gareth@brutha: ~/Lec06/multi/advanced (ssh)
gareth@brutha:~/Lec06/multi/advanced$ make debug
gcc -Wall -g -c main.c
gcc -Wall -g -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/advanced$ make clean
rm *.o
rm myprog
gareth@brutha:~/Lec06/multi/advanced$ make opt
gcc -Wall -O2 -c main.c
gcc -Wall -O2 -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/advanced$ make clean
rm *.o
rm myprog
gareth@brutha:~/Lec06/multi/advanced$ make
gcc -Wall -c main.c
gcc -Wall -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/advanced$
```

- Compilation
- Makefiles
- Revision Control
 - Git

What is Revision Control?

- Files, such as C source files, change over time as modifications are made.
- Often, over a long period of time, the reason for the changes that have been made is lost. This is particularly difficult when a file is worked on by multiple people.
- Revision control (also known as Version control, Change control and Source Code management) is a method for managing the changes to a file, and maintaining a history of the changes that have taken place.



The screenshot shows a terminal window with three versions of a C source file, loop1.c, loop2.c, and loop3.c, displayed side-by-side. The terminal title is "1. gareth@brutha: ~/Lec06/revision (ssh)". The code is color-coded: blue for the first part of the main function, pink for the if statement, and light blue for the for loop. The status bar at the bottom shows the file names and their revision numbers: loop1.c 5,0-1 All, loop2.c 11,0-1 All, and loop3.c 14,0-1 All.

```
1. gareth@brutha: ~/Lec06/revision (ssh)

1 int main(int argc, char *argv[]) {
2     return 0;
3 }
4
5

loop1.c 5,0-1 All

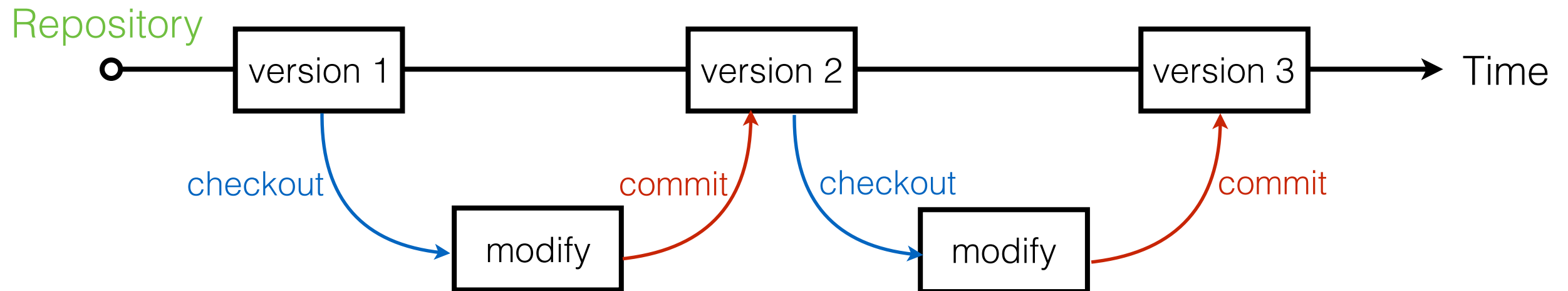
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     if (1 == argc) {
5         printf("Arghhhhh\n");
6         return 1;
7     }
8     return 0;
9 }
10
11

loop2.c 11,0-1 All

1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     if (1 == argc) {
5         printf("Arghhhhh\n");
6         return 1;
7     }
8     for (i=1; i < argc; i++) {
9         printf("Arg %d - %s\n",i,argv[i])
10    }
11    return 0;
12 }
13
14

loop3.c 14,0-1 All
```

The Basics of Revision Control



Repository: Location of stored files, and file changes

Checkout: Get a complete version of the code from the repository

Commit: Send an updated version of the code to the repository
(Along with some message/log)

```
1. gareth@brutha: ~/Lec06/revision (ssh)

1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     if (1 == argc) {
5         printf("Arghhhhh\n");
6         return 1;
7     }
8     return 0;
9 }
10
11

loop1.c 5,0-1 All

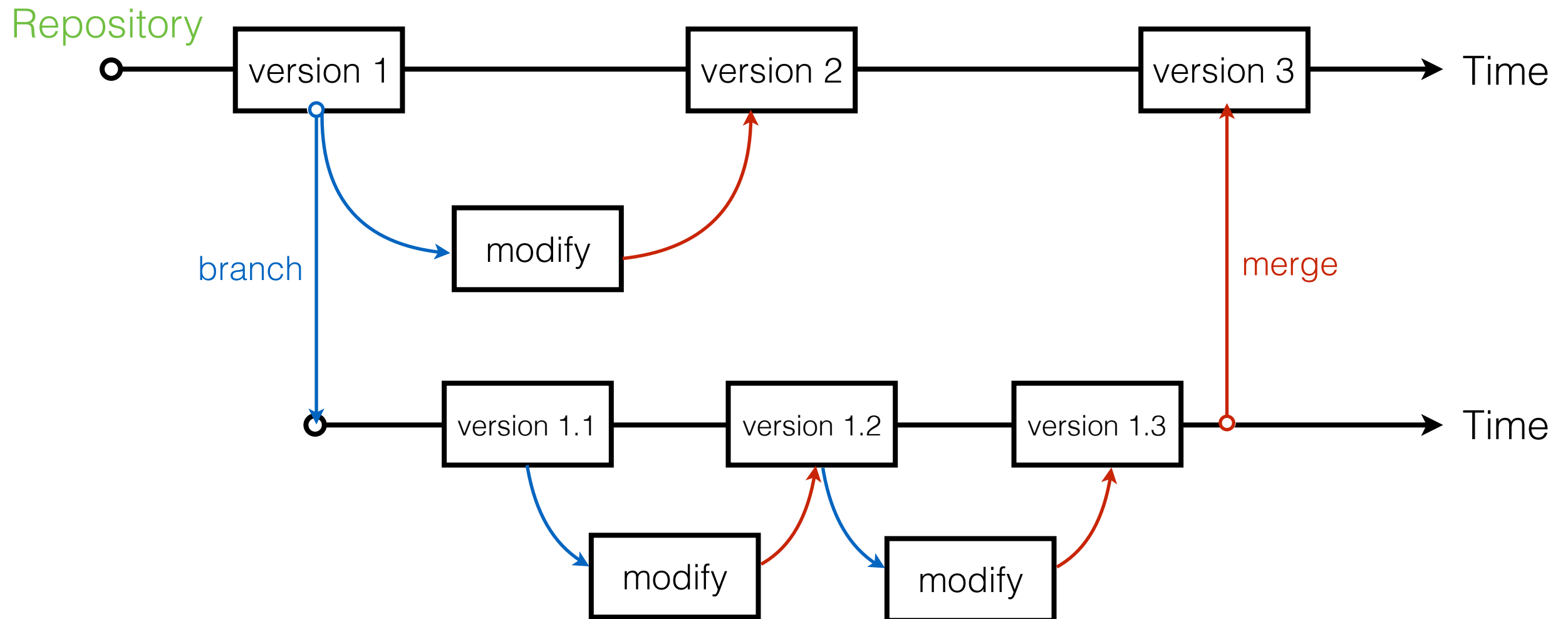
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     if (1 == argc) {
5         printf("Arghhhhh\n");
6         return 1;
7     }
8     for (i=1; i < argc; i++) {
9         printf("Arg %d - %s\n",i,argv[i])
10    }
11    return 0;
12 }
13
14

loop2.c 11,0-1 All

1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     if (1 == argc) {
5         printf("Arghhhhh\n");
6         return 1;
7     }
8     for (i=1; i < argc; i++) {
9         printf("Arg %d - %s\n",i,argv[i])
10    }
11    return 0;
12 }
13
14

loop3.c 14,0-1 All
```

Branch & Merge

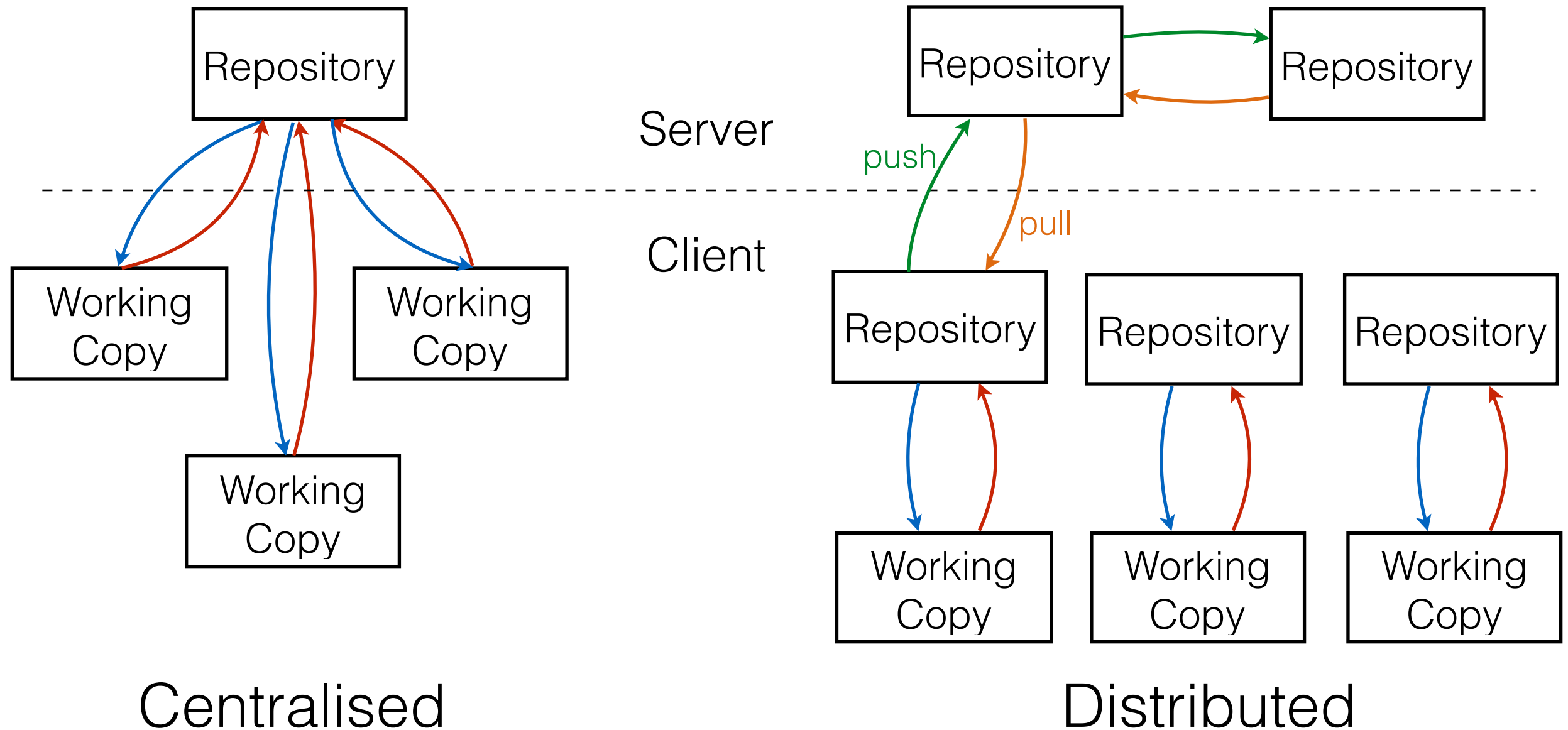


Another key concept is branching and merging.

Branch: Create a complete copy of the code with its own revision history.

Merge: Take all the changes that have occurred within a branch and merge them (including the history).

Distributed vs. Centralised



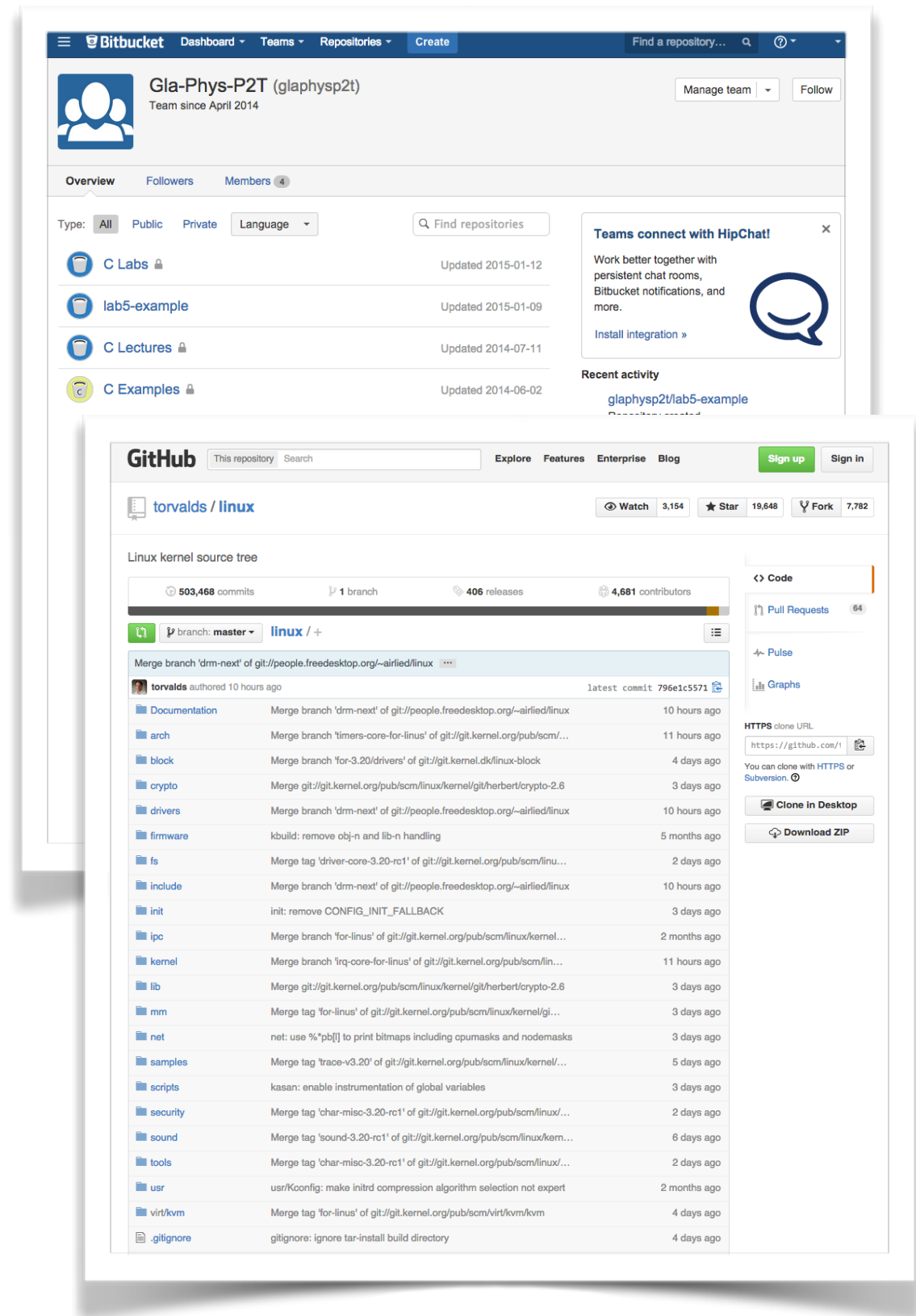
- All history, tags, branches are kept on the server.
- Only one canonical source for all code.
- Simple access controls, and user management.

- Each client has a complete copy of the history.
- Easy for offline work, simple syncing mechanisms.
- No single canonical source for all changes.

- Compilation
- Makefiles
- Revision Control
 - Git

Git

- **git** is a distributed revision control system developed by Linus Torvalds in 2005 to manage the development of the Linux Kernel.
- Due to its distributed nature it became popular amongst the open source community.
- Lots of online resources are available as well as free online hosting of repositories.
- Two of the biggest sites are:
 - github.com - unlimited public repositories, need to pay for private, “social coding”.
 - bitbucket.org - unlimited private repositories, tools not quite so good as github.
- Both of these sites have lots of links to tutorials and training material. if your interested have a look!



Git - Getting Started

- A git repository can be created in two ways, we can:
 - 1) create an new, empty repository
 - 2) copy/clone an existing repository (i.e. pull down a complete copy)
- To create a new, empty repository we can do:
 - **git init <directory>**
 - **git init .**
- All of the files that git uses to manage your repository are located in a special hidden directory call **.git** in your project folder.
- To copy an exiting project, we need a link to it's location (for instance in the lab you'll be asked to):
 - **git clone https://bitbucket.org/glaphysp2t/lab5-example.git**
- This will copy down a complete version of the code including it's entire history.

```
1. gareth@brutha: ~/Lec06/git (ssh)
gareth@brutha:~/Lec06/git$ git init proj
Initialized empty Git repository in /home/gareth/Lec06/git/proj/.git/
gareth@brutha:~/Lec06/git$ tree
.
├── proj

1 directory, 0 files
gareth@brutha:~/Lec06/git$ tree -a
.
├── proj
│   └── .git
│       ├── branches
│       ├── config
│       ├── description
│       ├── HEAD
│       ├── hooks
│       │   ├── applypatch-msg.sample
│       │   ├── commit-msg.sample
│       │   ├── post-update.sample
│       │   ├── pre-applypatch.sample
│       │   ├── pre-commit.sample
│       │   ├── prepare-commit-msg.sample
│       │   ├── pre-rebase.sample
│       │   └── update.sample
│       ├── info
│       │   └── exclude
│       ├── objects
│       │   ├── info
│       │   └── pack
│       └── refs
│           ├── heads
│           └── tags
```

```
3. gareth@brutha: ~/Lec06/git/lab5-example (ssh)
gareth@brutha:~/Lec06/git$ git clone https://bitbucket.org/glaphysp2t/lab5-example.git
Cloning into 'lab5-example'...
remote: Counting objects: 33, done.
remote: Compressing objects: 100% (31/31), done.
remote: Total 33 (delta 11), reused 0 (delta 0)
Unpacking objects: 100% (33/33), done.
gareth@brutha:~/Lec06/git$ ls
lab5-example  notes  proj
gareth@brutha:~/Lec06/git$ cd lab5-example/
gareth@brutha:~/Lec06/git/lab5-example$ ls
draw.c draw.h histogram.c makefile README.md util.c util.h
gareth@brutha:~/Lec06/git/lab5-example$
```


Git - Adding & Committing

- Now that we have a repository we'd like to add some files to our repository.
- To do this we use the “**git add**” command.
 - **git add <filename>**
- It's important to note that this only stages the file to be added, but hasn't actually added it yet.
- **To make any change to the repository we must save the action, or commit it.**
- To do this we use the “**git commit**” command, and supply it a message (usually a reason for what has added or has been changed)
 - **git commit -m “Adding some files”**
- If the “**-m**” parameter is not specified, the commit log will be opened in whatever editor is specified in \$EDITOR or \$VISUAL.
- You can remove a file from the repository using the “**git rm**” command.
 - **git rm <filename>**
- As with **git add** changes don't take effect until you commit them.

```
1. gareth@brutha: ~/Lec06/git/proj (ssh)
gareth@brutha:~/Lec06/git/proj$ touch test.c
gareth@brutha:~/Lec06/git/proj$ git add -v test.c
add 'test.c'
gareth@brutha:~/Lec06/git/proj$ git commit -m "Added a file"
[master (root-commit) 7ea05b6] Added a file
 0 files changed
 create mode 100644 test.c
gareth@brutha:~/Lec06/git/proj$ git rm test.c
rm 'test.c'
gareth@brutha:~/Lec06/git/proj$ git commit -m "Removed a file"
[master 48c8921] Removed a file
 0 files changed
 delete mode 100644 test.c
gareth@brutha:~/Lec06/git/proj$ ls
gareth@brutha:~/Lec06/git/proj$
```


Git - Status

- It's often difficult to see the status of all the files we're working on.
- Often we've created new files, modified some and deleted some.
- We can check the status of all the files in the repository using:

- **git status**

- “**git status**” will tell us about files that are present but not part of the repository, files that have been added and not committed, files that have been modified or files that have been delete.
- It will omit any files that are unchanged, which are included in the repository.
- If the output is overly verbose we can get a short description by using:

- **git status -s**

```
1. gareth@brutha: ~/Lec06/git/proj (ssh)
gareth@brutha:~/Lec06/git/proj$ touch test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       test.c
nothing added to commit but untracked files present (use "git add" to
track)
gareth@brutha:~/Lec06/git/proj$ git add test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   test.c
#
gareth@brutha:~/Lec06/git/proj$ git commit -m "Added a file"
[master f029e5a] Added a file
0 files changed
create mode 100644 test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
nothing to commit (working directory clean)
gareth@brutha:~/Lec06/git/proj$ rm test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working dir
ectory)
#
#       deleted:    test.c
#
no changes added to commit (use "git add" and/or "git commit -a")
gareth@brutha:~/Lec06/git/proj$ git status -s
D test.c
gareth@brutha:~/Lec06/git/proj$
```

Git - Adding & Committing (2)

- If we modify a file and want to commit the changes we need to use the “**git add**” command again:

- **git add <filename>**

- The file is now tagged as modified in the output of “**git status**”

- Once again the changes aren’t saved in the repository until we commit them:

- **git commit -m “Modified a file”**

- With modified files we can use a short cut and add, commit them at the same time using:

- **git commit -a -m “Commit and Add”**

- **Note:** This only works for modified files, not files that are not in the repository already.

```
1. gareth@brutha: ~/Lec06/git/proj (ssh)
gareth@brutha:~/Lec06/git/proj$ git status -s
gareth@brutha:~/Lec06/git/proj$ echo Hello > test.c
gareth@brutha:~/Lec06/git/proj$ git status -s
 M test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   test.c
#
no changes added to commit (use "git add" and/or "git commit -a")
gareth@brutha:~/Lec06/git/proj$ git add test.c
gareth@brutha:~/Lec06/git/proj$ git status -s
 M test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   test.c
#
gareth@brutha:~/Lec06/git/proj$ git commit -m "Modified test.c"
[master 6cb577d] Modified test.c
 1 file changed, 1 insertion(+)
gareth@brutha:~/Lec06/git/proj$ echo Hello2 >> test.c
gareth@brutha:~/Lec06/git/proj$ git status -s
 M test.c
gareth@brutha:~/Lec06/git/proj$ git commit -a -m "Committed and added"
[master c10555f] Committed and added
 1 file changed, 1 insertion(+)
gareth@brutha:~/Lec06/git/proj$ git status -s
gareth@brutha:~/Lec06/git/proj$
```

Git - Log

- To see a log of all the commits, and the messages associated with them we can use the “**git log**” command.

- **git log**

- By default “**git log**” will output:
 - the full hash for the commit (it's ID).
 - the full commit message.
 - who performed the committed
 - when the commit took place.
- This can be quite verbose so we can simplify this by doing:

- **git log --oneline**

```
1. gareth@brutha: ~/Lec06/git/proj (ssh)
gareth@brutha:~/Lec06/git/proj$ git log
commit c805352a3edca2214321b5ce00aa8f870ed70a06
Author: Gareth Roy <gareth.roy@glasgow.ac.uk>
Date: Tue Feb 17 11:19:59 2015 +0000

Deleted another file

commit f029e5aacf05b1e070d5a924ce61dc3ab084bfa7
Author: Gareth Roy <gareth.roy@glasgow.ac.uk>
Date: Tue Feb 17 11:10:47 2015 +0000

Added a file

commit 48c89211b8bc858a0727ff98a18587f234348003
Author: Gareth Roy <gareth.roy@glasgow.ac.uk>
Date: Tue Feb 17 11:06:56 2015 +0000

Removed a file

commit 7ea05b6cbb2934604fc0a9ef00b6366c7fe0b36e
Author: Gareth Roy <gareth.roy@glasgow.ac.uk>
Date: Tue Feb 17 11:05:59 2015 +0000

Added a file
gareth@brutha:~/Lec06/git/proj$ git log --oneline
c805352 Deleted another file
f029e5a Added a file
48c8921 Removed a file
7ea05b6 Added a file
gareth@brutha:~/Lec06/git/proj$
```

```
2. gareth@brutha: ~/Lec06/git/lab5-example (ssh)
gareth@brutha:~/Lec06/git/lab5-example$ git log --oneline
8fbd35f Refactored to move utility code out of the main file
e532f4f Added randomly generated data
6dcdfe5 Refactored code so that the drawing routines are in their own file, added to the mak
6b4875c Added a simple makefile to build histogram, included a make test which runs the prog
8b2c746 Added a function that calculates the maximum of a list of data and returns it's valu
66ee505 Added in a simple axis to the histogram
c496941 Refactored to create a printBar function
2a65f62 A simple program that prints out an ASCII histogram based on a test set of integer d
d71c39b Added an initial README to the project
gareth@brutha:~/Lec06/git/lab5-example$
```


Git - Log

```
4. git
Folkvangr:CLabs-TeX gareth$ git log --oneline --graph --decorate
* dfb76d6 (HEAD, origin/master, master) Removed more questions from labs
* 3e037f8 Removed some questions
* 14b61b3 Removed some questions
* d9768f0 Draft changes to labs SCS
* 1170b7a First fixed revision, lab01 SCS
* e96d985 Merge branch 'master' of bitbucket.org:glaphysp2t/c-labs
| \
| * 03fbab5 Merge branch 'master' of bitbucket.org:glaphysp2t/c-labs
| |
| | * aeaac3a Merge branch 'master' of bitbucket.org:glaphysp2t/c-labs
| | |
| | | * b404d78 Tidied up some of the text - GDR
| | | * b0210c1 lab6: challenging question added
| | |
| | | * b27b03f lab5: minor changes
| | |
| | |
| | | * a69e7a0 lab6: minor corrections
| | | * 840609e lab6: minor corrections
| | | * b58571d lab6: background theory added
| | |
| | |
| | | * 9b5392e minor changes
| | | * d8e4004 lab5: background theory/information added
| | | * 476a117 Added loop examples
| | | * 1e3da6c Merge branch 'master' of bitbucket.org:glaphysp2t/c-labs
| | |
| | | \
| | | * 6ca1227 lab3: minor corrections
| | | * 3e88faa Updated and re-worked lab02 - GDR
| | |
| | | /
| | | * 0c46ce9 lab4: description of functions
| | | * 268c124 lab1: expanded on precedence
| | | * 307388f lab2: minor correction
| | | * 32d36ce lab3: background theory/information
| | | * 9246f34 lab1: char type decription modified
| | | * fbb77a5 lab1: modified puts() and printf()
| | | * 07d3346 lab1: question 2 modified
| | | * 1378206 lab1: question 1 merged with question 4
| | | * f217d69 Merge branch 'master' of bitbucket.org:glaphysp2t/c-labs
| | |
| | | \
```

Git - Branch and Merge

- To create a branch we do the following:
 - **git branch <branch>**
- We can get a list of all the branches in our repository by doing:
 - **git branch --list**
- We can delete an unneeded branch by doing:
 - **git branch -d <branch>**
- To change into a branch we have to “checkout” that branch into our working directory:
 - **git checkout <branch>**
- We can create a new branch and change into it all at once doing:
 - **git checkout -b <branch>**
- Finally we can merge a branch back into the current branch by doing:
 - **git merge <branch>**

```
1. gareth@brutha: ~/Lec06/git/proj (ssh)
gareth@brutha:~/Lec06/git/proj$ git branch myfeature
gareth@brutha:~/Lec06/git/proj$ git branch --list
* master
  myfeature
gareth@brutha:~/Lec06/git/proj$ git checkout myfeature
Switched to branch 'myfeature'
gareth@brutha:~/Lec06/git/proj$ git branch --list
master
* myfeature
gareth@brutha:~/Lec06/git/proj$ cat test.c
Hello
Hello2
gareth@brutha:~/Lec06/git/proj$ echo Hello3 >> test.c
gareth@brutha:~/Lec06/git/proj$ git commit -a -m "Test"
[myfeature c985651] Test
 1 file changed, 1 insertion(+)
gareth@brutha:~/Lec06/git/proj$ cat test.c
Hello
Hello2
Hello3
gareth@brutha:~/Lec06/git/proj$ git checkout master
Switched to branch 'master'
gareth@brutha:~/Lec06/git/proj$ cat test.c
Hello
Hello2
gareth@brutha:~/Lec06/git/proj$ git merge myfeature
Updating 2ae35bd..c985651
Fast-forward
 test.c | 1 +
 1 file changed, 1 insertion(+)
gareth@brutha:~/Lec06/git/proj$ cat test.c
Hello
Hello2
Hello3
gareth@brutha:~/Lec06/git/proj$ git branch --list
* master
  myfeature
gareth@brutha:~/Lec06/git/proj$ git branch -d myfeature
Deleted branch myfeature (was c985651).
gareth@brutha:~/Lec06/git/proj$
```

- Compilation
- Makefiles
- Revision Control
 - Git