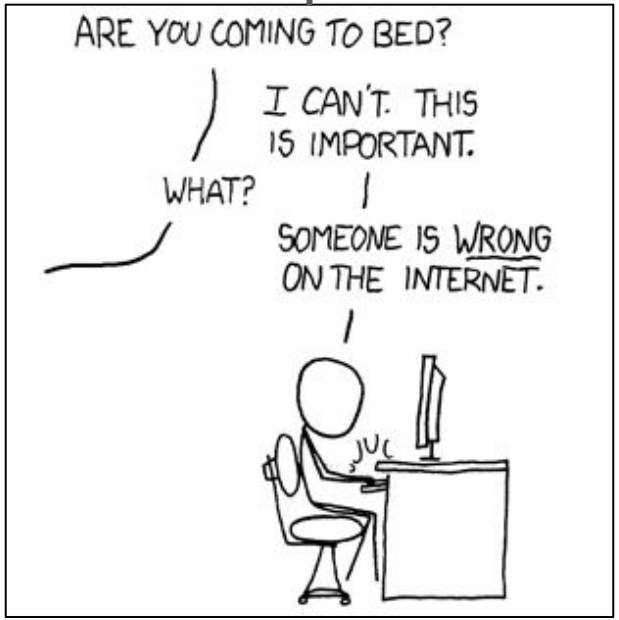
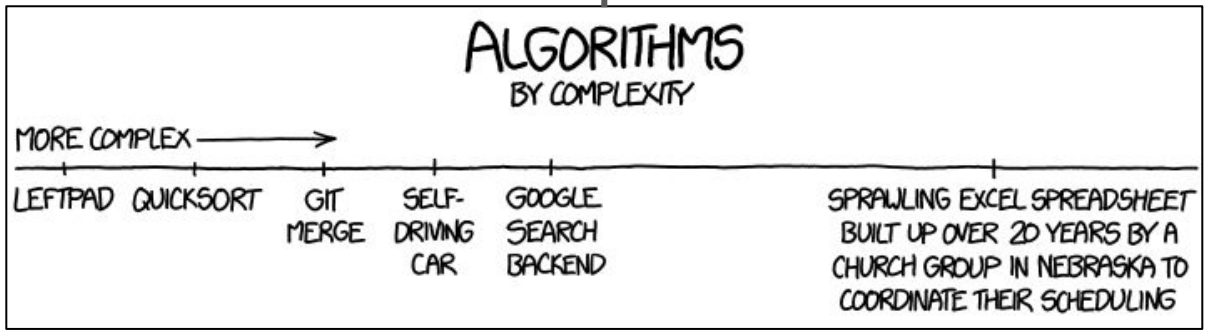
You came early and We appreciate it
Until we get started, we would like you to do the following

1. Open Ubuntu csuser
2. If you don’t have a Github account
   a. Create one
3. Checkout Hacktoberfest 2019 and ask us questions

A Walk through the Transformation Hierarchy



Problem
Algorithms
Program
ISA (Instruction Set Arch)
Microarchitecture
Circuits
Electrons



Problem

Algorithms

Program

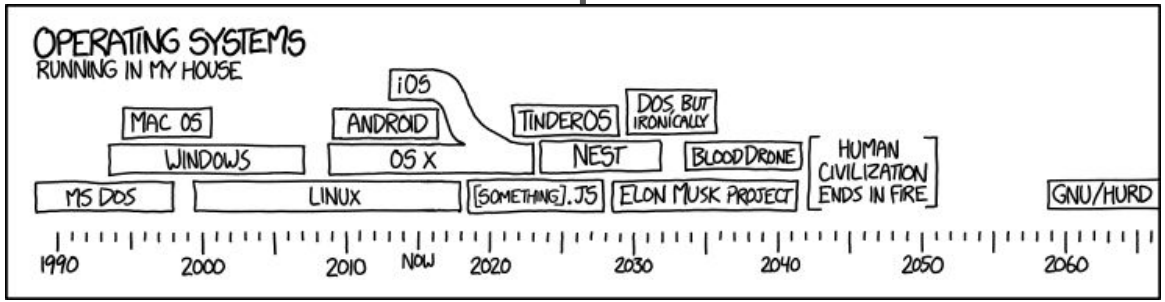
ISA
(Instruction Set Arch)

Microarchitecture

Circuits

Electrons

Problem
Algorithms
Program
ISA (Instruction Set Arch)
Microarchitecture
Circuits
Electrons



Problem

Algorithms

Program

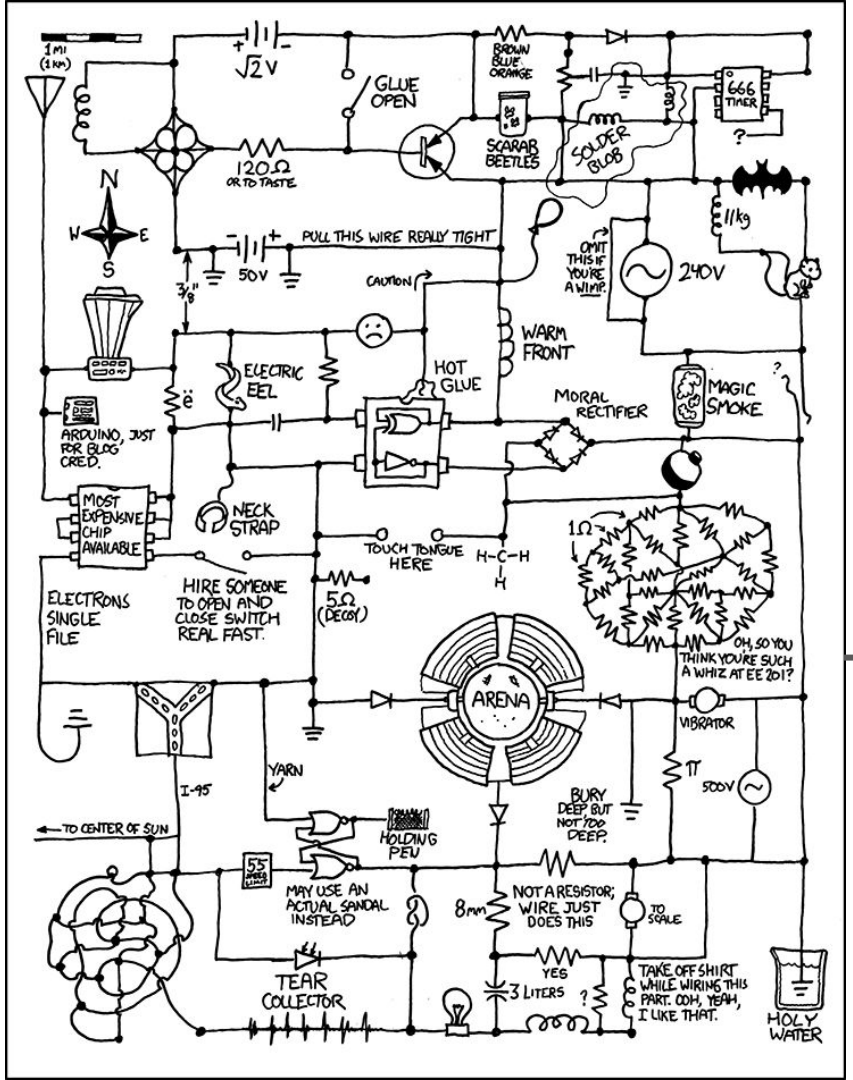
ISA
(Instruction Set Arch)

Microarchitecture

Circuits

Electrons

Problem
Algorithms
Program
ISA (Instruction Set Arch)
Microarchitecture
Circuits
Electrons



Problem
Algorithms
Program
ISA (Instruction Set Arch)
Microarchitecture
Circuits
Electrons

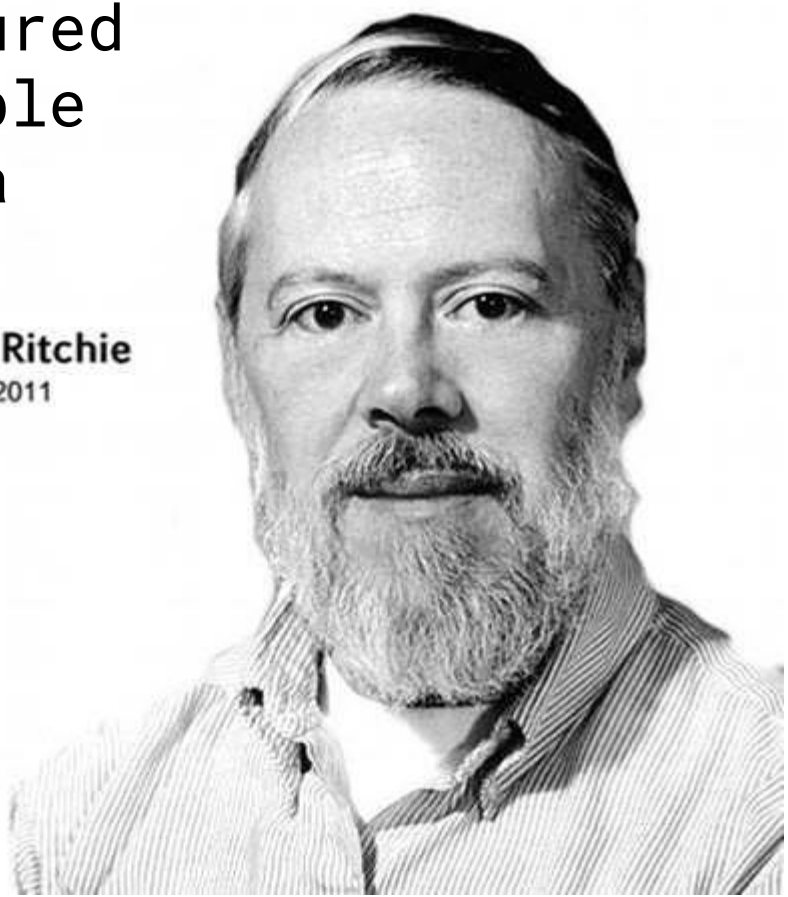
[Breaking down: C]

Why C if there is C++?

*“C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it. Quite frankly, even if the choice of C were to do *nothing* but keep the C++ programmers out, that in itself would be a huge reason to use C.”*

- Linus Torvalds,

C is a general purpose, procedural programming language supporting structured programming, lexical variable scope and recursion while a static type system



[A Bit of History]

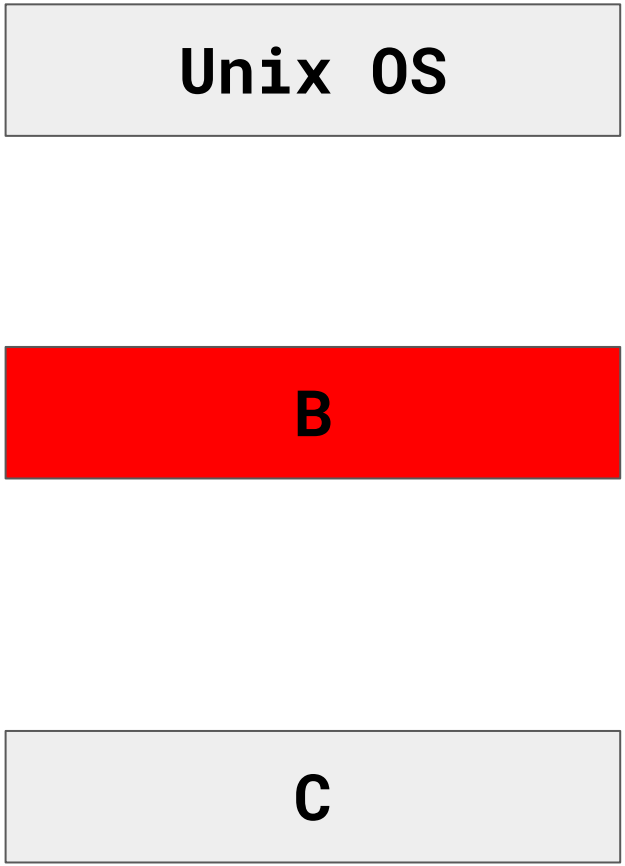
Unix was written in assembly by Dennis Ritchie and Ken Thompson for PDP-7. Thompson wanted to develop utilities for the new platform

Unix OS

B

C

Thompson modified what was then BCPL systems programming language and developed B. But the utilities written in B were slow couldn’t take advantage of PDP-11



Ritchie then improved B which resulted in C. Then the entire linux kernel was rewritten in C. Unix became the first system that was implemented in a high level language.

Unix OS

B

C

[Hello World in C]

Let’s break it down line by line

```c
#include <stdio.h>

int main(){
  printf("Hello, World\n");
  return 0;
}
```

What is this?

```c
#include <stdio.h>

int main(){
  printf("Hello, World\n");
  return 0;
}
```

It is a preprocessor statement

```
#include <stdio.h>

int main(){
  printf("Hello, World\n");
  return 0;
}
```

Here, it includes the declaration of functions in **stdio**

```c
#include <stdio.h>

int main(){
  printf("Hello, World\n");
  return 0;
}
```

Now what is this?

```c
#include <stdio.h>

int main(){
  printf("Hello, World\n");
  return 0;
}
```

It is the entrypoint into the program

```c
#include <stdio.h>

int main(){
  printf("Hello, World\n");
  return 0;
}
```

Return type: int

```
#include <stdio.h>

int main(){
  printf("Hello, World\n");
  return 0;
}
```

Return value: 0

```c
#include <stdio.h>

int main(){
  printf("Hello, World\n");
  return 0;
}
```

Calls the stdlib function **printf** with **string**

```c
#include <stdio.h>

int main(){
  printf("Hello, World\n");
  return 0;
}
```

[Types]

What is this?

```
int i = 0;
```

This is a definition

```
int i = 0;
```

Types in C - https://en.cppreference.com/w/c/types

void

char

signed char, short, int, long, long long

unsigned versions

Floating point types - float, double, long double

Enumerated types

Types in C - https://en.cppreference.com/w/c/types

- Derived types
  - Array types
  - Structure types
  - Union types
  - Function types
  - Pointer types
  - Atomic types

Types in C - https://en.cppreference.com/w/c/types

There are some more too.

But along with this type there are **const, volatile and restrict**

https://godbolt.org/z/qg70Bd

Declaration:

- Introduces a symbol into the current scope and specifies meaning and properties.

```
int i = 0;
int func(int a)
```

Definition:

- A definition is a declaration that provides all information about the identifiers it declares.

```
int i = 0;
int func(int a) {
    return a;
}
```

Mathematically?

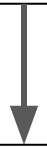
```
int i = 0;
```

$$i \in \{-65536, \ldots, 65535\}$$

```
int i = 0;
```

$$i \in \{-65536, \ldots, 65535\}$$

```
int i = 0;
```



```
int i = 1;
```

What is declaration of struct?

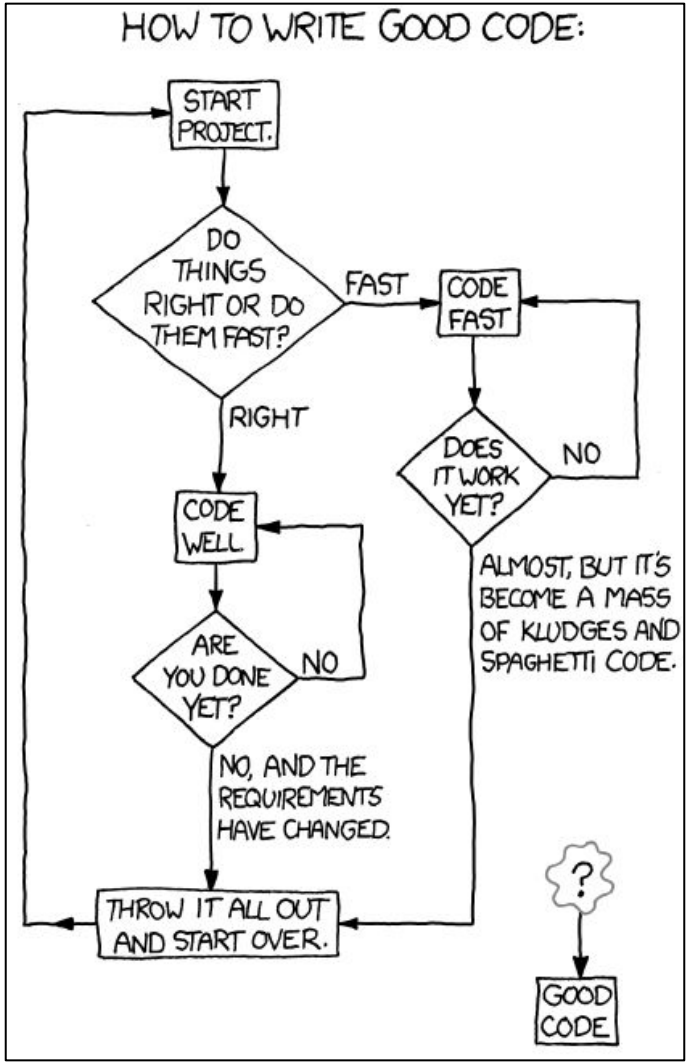
```
struct X; // declaration

struct X { int n; }; // definition
```

[Pointers - Oh My God]

What are Pointers?

Pointers are variables that point to a memory location. The memory location can be valid or invalid but accessing an invalid memory location is an undefined behavior


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

[Compilation Process - Demystified]


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."
HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

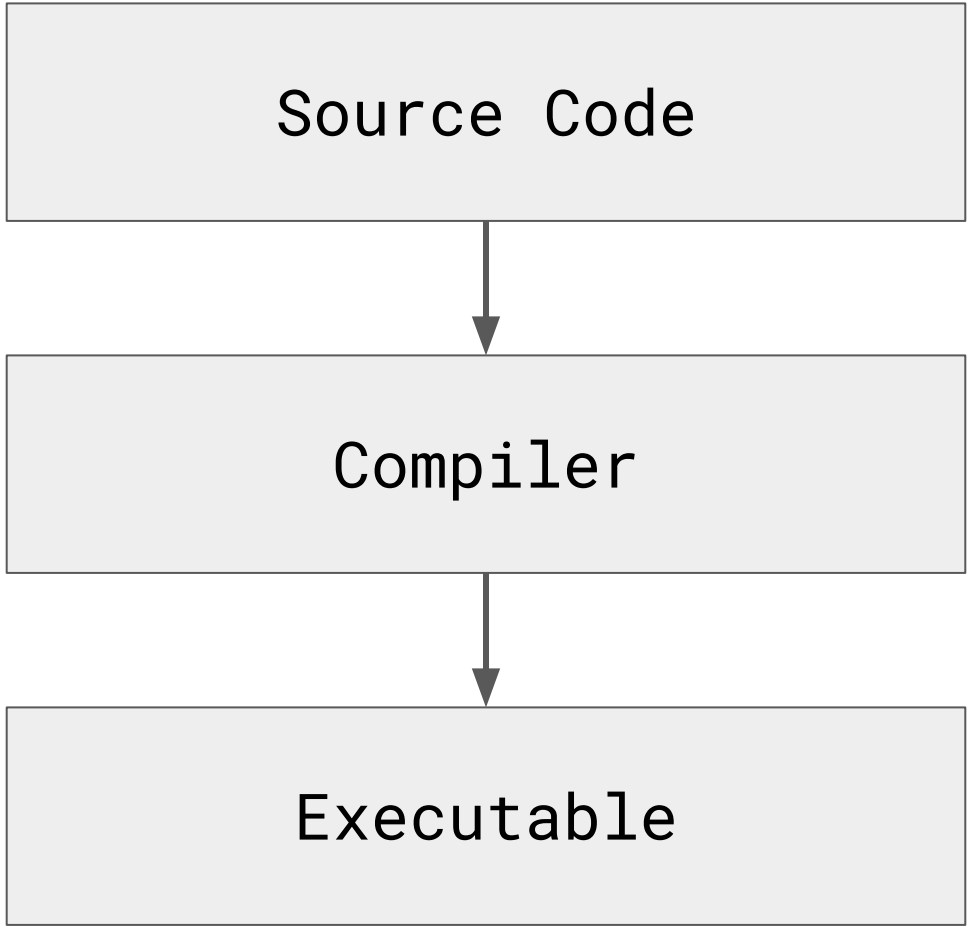
The compiler you typically think of is, not a compiler.

The compiler you typically think of, is not a compiler. **Rather it is a compiler driver**

The compiler you typically think of, is not a compiler. **Rather it is a compiler driver**

A **Compiler driver** is basically a script that walks through the process of turning your source code into an executable or a library

The Brid’s Eye View



Source

Preprocessor

Compiler

Assembler

Linker

Executable

Takes care of preprocessor definitions and macros **-E**

https://godbolt.org/z/Wgju_I

Source

Preprocessor

Compiler

Assembler

Linker

Executable

Converts preprocessed source to assembly **-S**

https://godbolt.org/z/6p8fpN

Source

Preprocessor

Compiler

Assembler

Linker

Executable

Convert assembly into object code of a special format(ELF)

-c

Source

Preprocessor

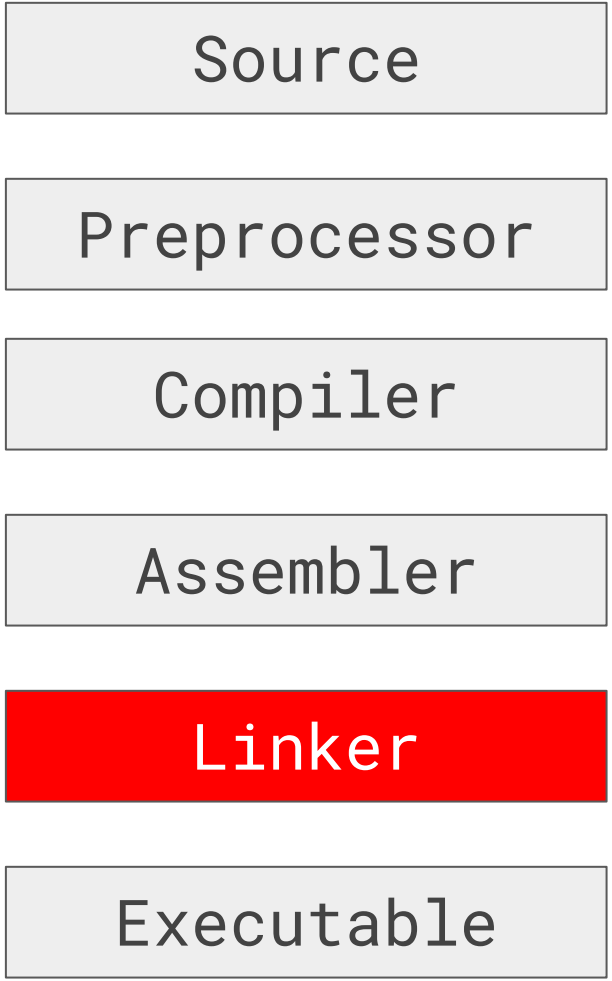
Compiler

Assembler

Linker

Executable

Then the linker combines all the object code and generates an executable



Is Compilation just translation?

Is Compilation just translation?

Well, It is not.

Is Compilation just translation?

Well, It is not.

https://gcc.godbolt.org/z/zsYmxb

[Executable Layout]

Linux-only

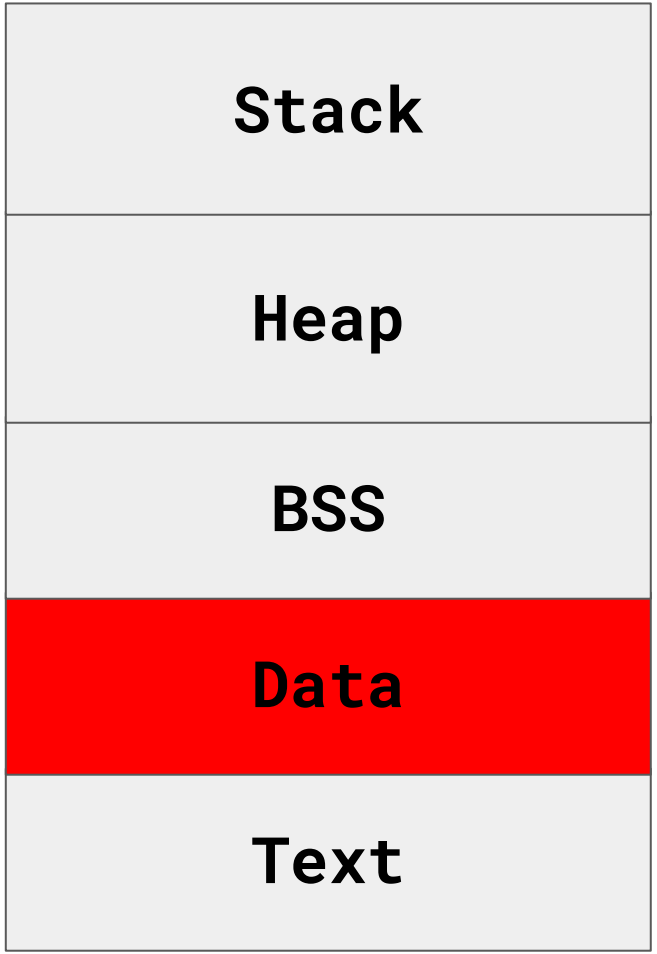
Compiler converts what you understand to what the operating system understand

Text section

Contains assembly code

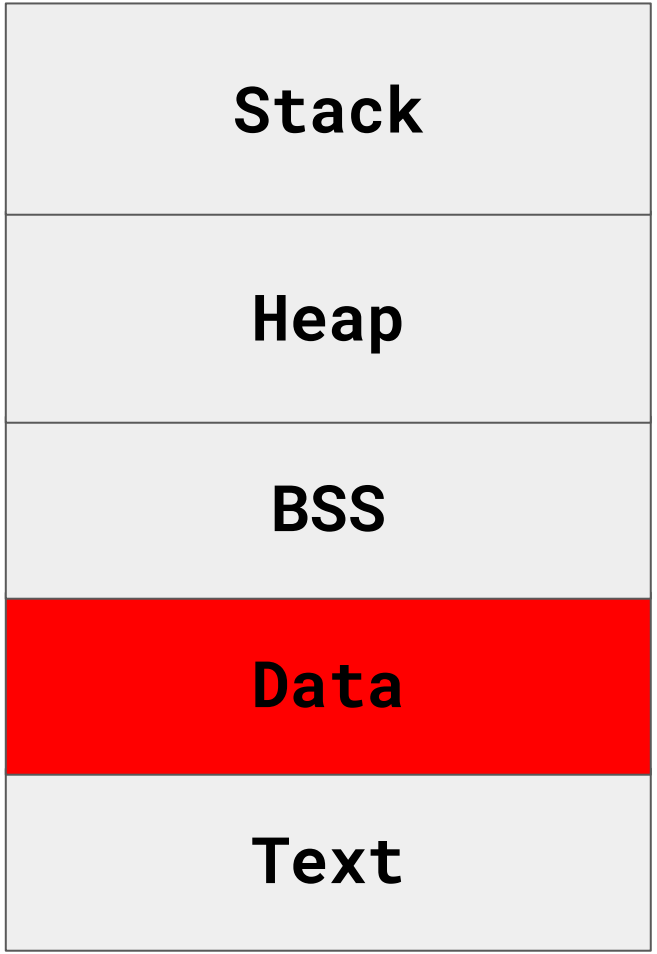
Data Section

Contains constant values



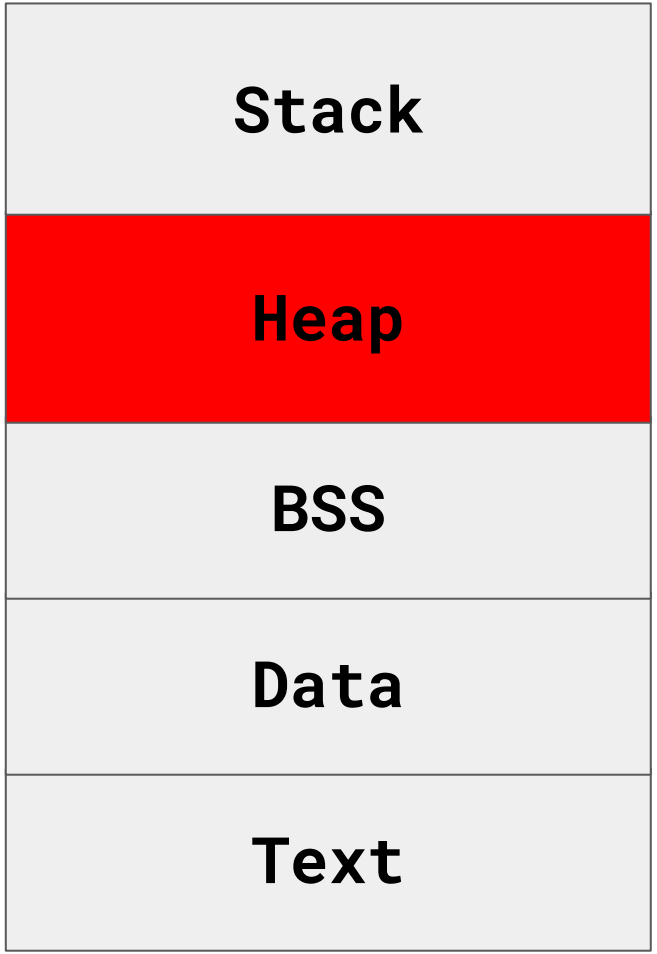
BSS Section

Contains constant values



Heap Section

Contains runtime allocated values



Stack Section

Contains contains local statically allocated variables

Some more things

- Header guards

[Makefiles]

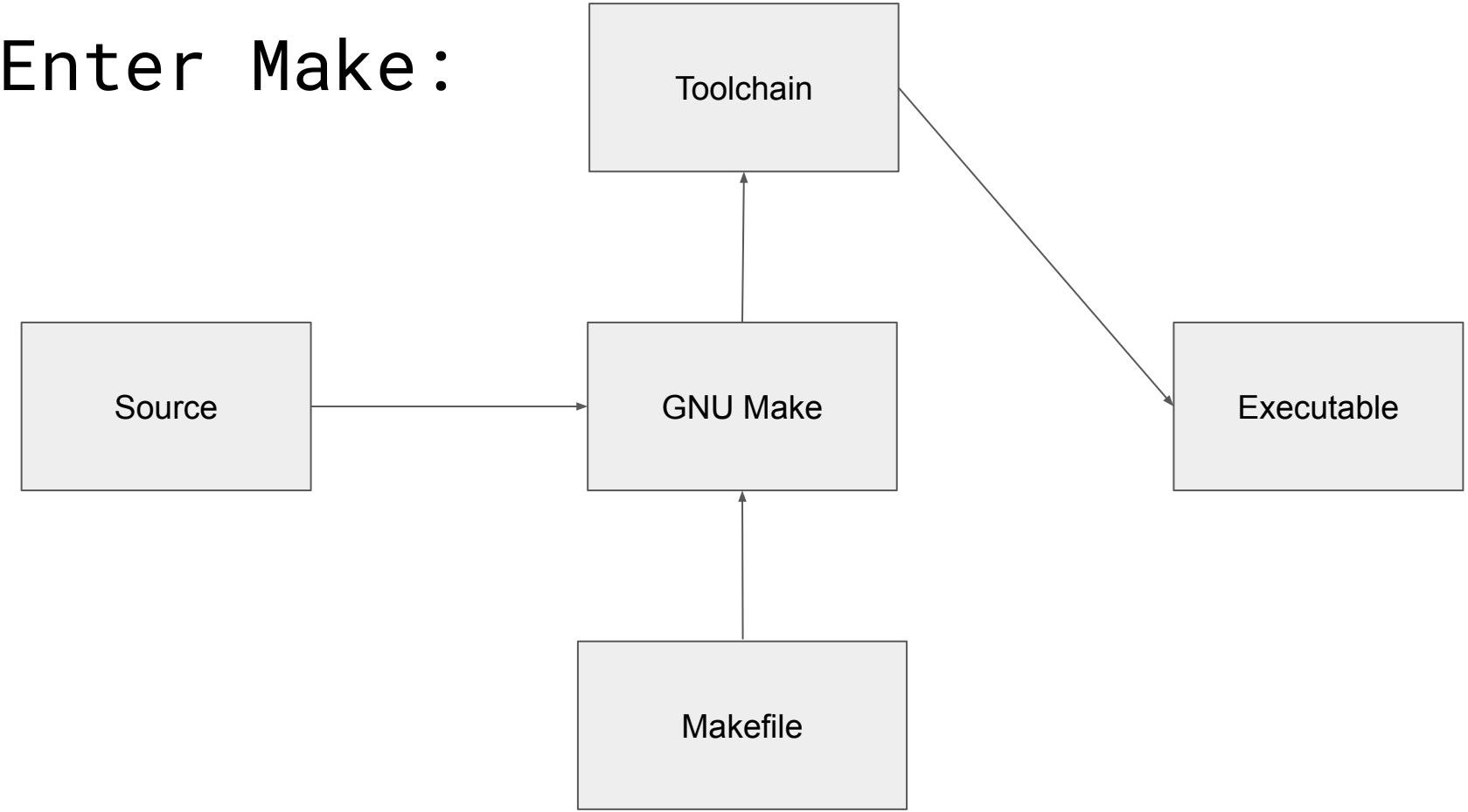
What’s the Largest codebase you’ve handled? How many files did it have?

The compilation problem:

```
g++ -Wall -Wextra -v -g -O2 -o hello file_1.cpp … file_n.cpp
-lmath -I. -I./boost-1.71.0/gcc-head/include -std=gnu++2a
-pedantic
    …
    …
    --end-me
```

- What if I need to build only a part of the software?
- What if I want to support multiple architectures?
- What if I end up mashing the up arrow key one-too-many times?

Enter Make:



Object files?

One-liner/no artifacts:

```
g++ -o myprog file1.cpp file2.cpp
```

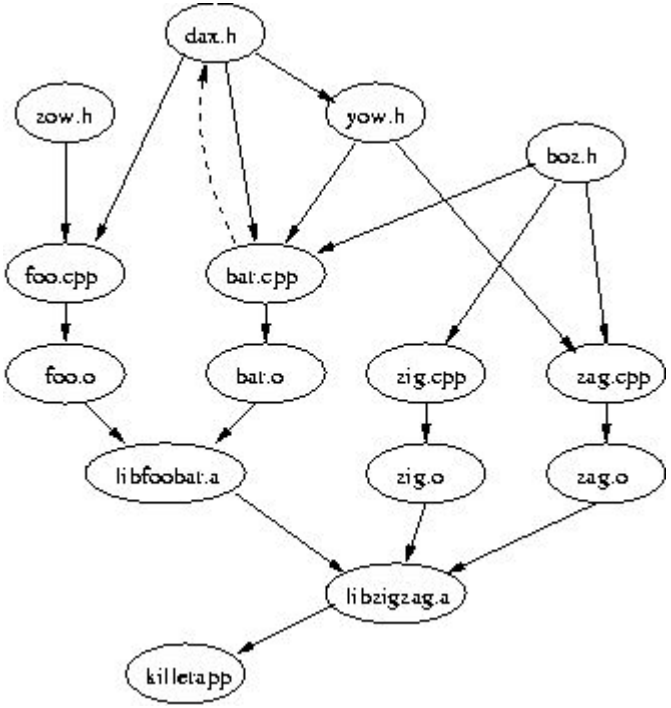
Verbose/persists object files:

```
g++ -c file1.cpp
g++ -c file2.cpp
g++ -o myprog file1.o file2.o
```

What benefits can leaving a clutter of .o files possibly have?

Dependencies:
For when your code needs code...

- Order of Compilation?
- Is parallel compilation possible?
- If no changes are made in the code, do I have to rebuild every file?



Makefile Deconstruction: Breaking down to build up

```
CC=gcc
CFLAGS=-I.
hellomake: hellomake.o hellofunc.o
     $(CC) -o hellomake hellomake.o hellofunc.o $(CFLAGS)
```

Makefile Deconstruction: Breaking down to build up

```
CC=gcc
CFLAGS=-I.
hellomake: hellomake.o hellofunc.o
     $(CC) -o hellomake hellomake.o hellofunc.o $(CFLAGS)
```

```
DEPS=hellomake.h
OBJ=hellomake.o hellofunc.o

%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

hellomakev2: $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS)
```

%	→	Pattern Matching
$@	→	Name of Target
$<	→	Name of first Dependency
$()	→	Variable substitution
$^	→	Dependencies

Makefile Deconstruction: Dem pesky object files

```
OBJDIR=obj
INCDIR =../include


.PHONY: clean

clean:
    rm -f $(OBJDIR)/*.o *~ core $(INCDIR)/*~
```

This doesn’t *make* things easier, does it?

Speaker notes:
Pause for resounding laughter.

CMake:
For when make needs make...

Never deal with a Makefile ever but follow a directory structure.

```
Project_name
|
|--CMakeLists.txt
|
\--include
|  |
|  \--Project_name
|      |--public_header.h
|
\--src
|  |--private_header.h
|  |--code.cpp
|
\--libs
|  \--libA
|  \--libB
|
\--tests
```

```
cmake_minimum_required(VERSION 3.0)
project(Project_name)

include_directories(include)

file(GLOB SOURCES "src/*.cpp")

add_executable(project ${SOURCES})

compile_options(-std=c++2a -Wall -std=gnu++2a -pedantic)
```

The Secret:

The Secret:

Nobody writes their own Makefiles.

The Secret:

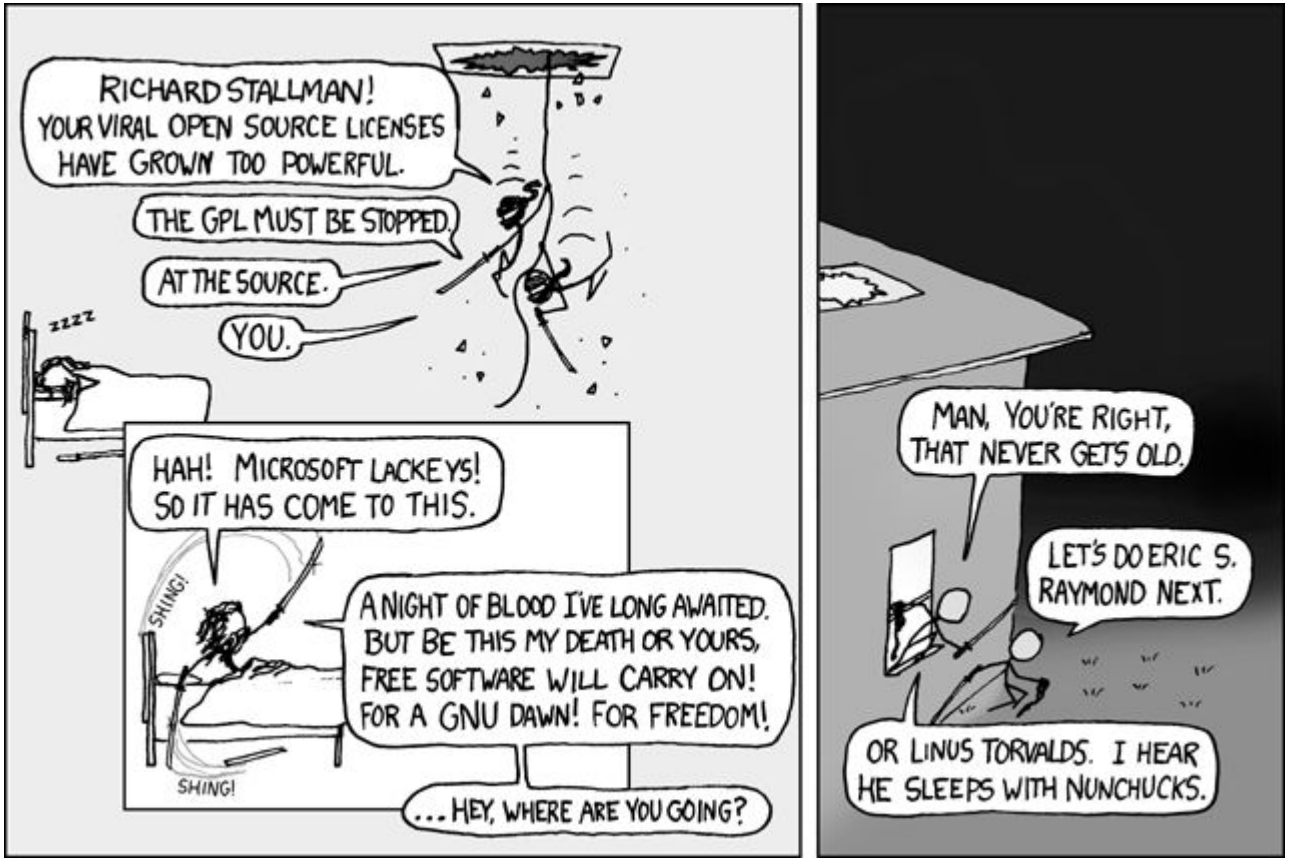
Nobody writes their own Makefiles.
Period.

The Secret:

Nobody writes their own Makefiles. Period.

The only Makefile written from scratch is probably the one written by the developer.

[FOSS]


RICHARD STALLMAN! YOUR VIRAL OPEN SOURCE LICENSES HAVE GROWN TOO POWERFUL.
THE GPL MUST BE STOPPED.
AT THE SOURCE.
YOU.
ZZZZ
HAH! MICROSOFT LACKEYS! SO IT HAS COME TO THIS.
SHING!
A NIGHT OF BLOOD I'VE LONG AWAITED. BUT BE THIS MY DEATH OR YOURS, FREE SOFTWARE WILL CARRY ON! FOR A GNU DAWN! FOR FREEDOM!
SHING!
...HEY, WHERE ARE YOU GOING?
MAN, YOU'RE RIGHT, THAT NEVER GETS OLD.
LET'S DO ERIC S. RAYMOND NEXT.
OR LINUS TORVALDS. I HEAR HE SLEEPS WITH NUNCHUCKS.

How it all started. :)

Richard Stallman, who was then a member of the MIT Artificial Intelligence labs announced the GNU Project in response to the change in culture in the computer industry, software development and its users.

This is a matter of freedom, not price, so think of “free speech,” not “free beer.”

[Git]

All the help you'll ever need.

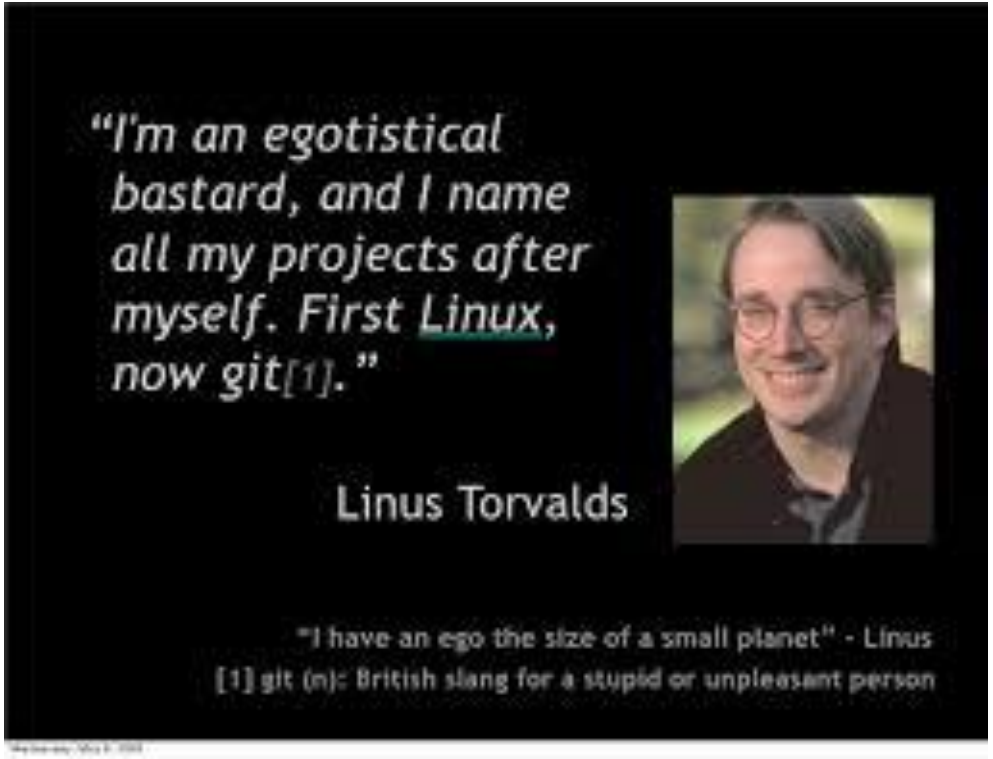
```
$ man git
$ man gittutorial
$ man giteveryday
```

https://github.com/dictcp/awesome-git

https://github.github.com/training-kit/downloads/github-git-cheat-sheet/

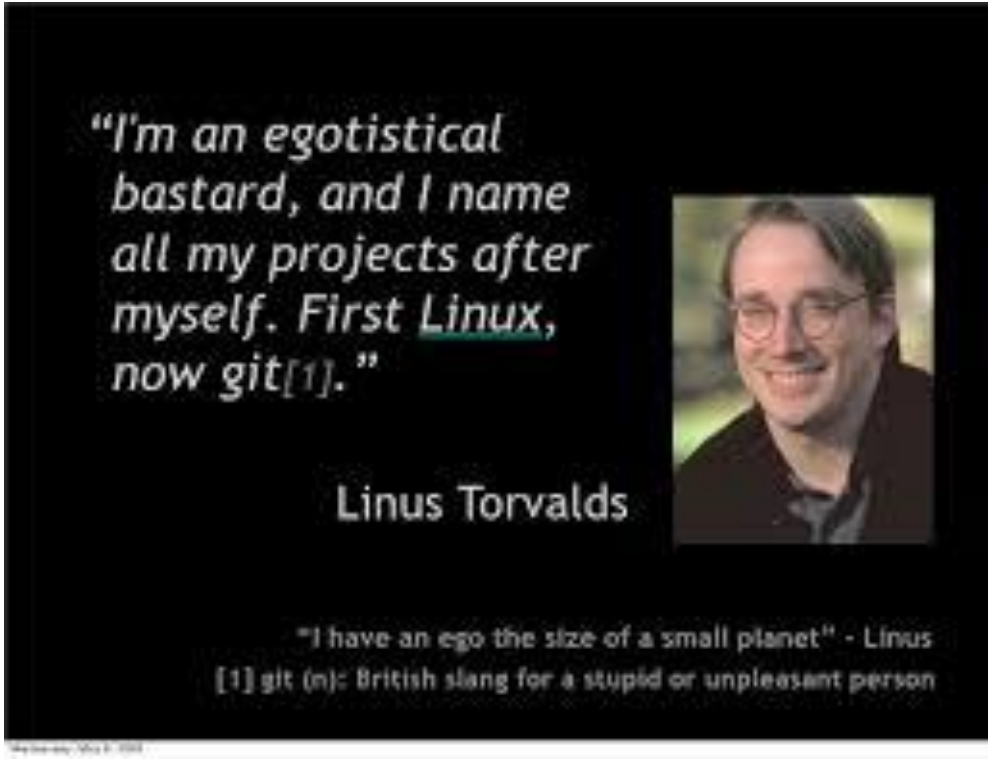
What is git?

- Git is a distributed version-control system which is used to track changes during software development.



What is git?

- Git is a distributed version-control system which is used to track changes during software development.
- **Linus Torvalds** started the project in April 2005 due to the increasing needs of the Linux kernel that existing version-control systems could not handle. **Junio Hamano** is now the mainter of git.



Some jargon that can help.

- **edit** - The part of the process when you make changes.

Some jargon that can help.

- **stage** - When changes are satisfactory enough to turn into a commit or how I like to call it a **save point**.

```
$ git add . //Track files.
```

Some jargon that can help.

- **commit** - When staged changes are now deemed ready to be made into substantial change **recorded** to be **pushed** to **remote. A commit** represents a particular state of the repository.

  $ git commit -m “This is a commit message”

Some jargon that can help.

- **remote** - Usually a copy the repository in a remote location/repository, which in our case is **GitHub.**

- **origin** - origin signifies the default **remote**.

- **upstream** - Usually, upstream signifies the **remote** from which we **fork** a **repository.**

- **local -** represents the hosts’ copy of the repository.

Some jargon that can help.

- **tracking** - When a **branch** is set to be tracked by git in relation to a particular remote/branch.

```
$ git checkout --track origin/develop
```

Some jargon that can help.

- **push** - When commits at **local** needs to be updated to a **remote.**

```
$ git push
```

Some jargon that can help.

- **push** - When commits at **local** needs to be updated to a **remote.**
- **pull** - When commits at a **remote** needs to fetched to **local**

$ git pull

Some jargon that can help.

- **push** - When commits at **local** needs to be updated to a **remote.**
- **pull** - When commits at a **remote** needs to fetched to **local**
- **branch** - A branch represents an independent line of development. Branches serve as an abstraction for the **edit/stage/commit** process.

```
$ git checkout -b new_branch old_branch
```

Some jargon that can help.

- **push** - When commits at **local** needs to be updated to a **remote.**
- **pull** - When commits at a **remote** needs to fetched to **local**
- **branch** - A branch represents an independent line of development. Branches serve as an abstraction for the **edit/stage/commit** process.
- **master branch -** A branch in a repository conventionally set as the mainline of the codebase which at any point in time is deployable.

Some jargon that can help.

- **push** - When commits at **local** needs to be updated to a **remote.**
- **pull** - When commits at a **remote** needs to fetched to **local**
- **branch** - A branch represents an independent line of development. Branches serve as an abstraction for the **edit/stage/commit** process.
- **master branch -** A branch in a repository conventionally set as the mainline of the codebase which at any point in time is deployable.
- **merge -** Join two or more development histories/branches together.

Basics - Something you will use 90% of the time.

```
$ git init //Initialize a repo.

$ git add //Track files.

$ git status //Status of git repo.

$ git commit //Commit staged files

$ git push //Push changes to upstream.
```

Git is actually a combination of multiple modules that have been written in C/Perl/Shell/Tcl!

Try these commands,you can append --help

Please play around with them and ask doubts.

Now Let’s try to add ourselves to the MEMBERS.md markdown file at this repository https://github.com/DAMCS/pragma

Please check the Contribution Guidelines.

And make a pull request!

[Valgrind & Sanitizers]

Valgrind

Valgrind is a programming tool for memory debugging, memory leak detection, and profiling.

Valgrind is actually a collection of tools, one of which is memcheck(default).

Valgrind is essentially a virtual machine that injects instrumentation code. Instrumentation code does not modify the state of the application because they are purely additive. It is used to help valgrind debug the program and provide information.

```
$ valgrind --tool=memcheck myprog arg1 arg2
```

Sanitizers

Sanitizers are compiler-based instrumentation components contained in external/compiler-rt that can be used during development and testing to push out bugs and make Android better.

Sanitizers are a part of most standard compilers. It's a modern method of debugging and is relatively new unlike valgrind.

Sanitizers work at the compiler level and require source code to work, enabled by compiler flags.

```
$ gcc myprog -fsanitize=address
```

C References

- https://en.cppreference.com/w/c
- https://github.com/isocpp/CppCoreGuidelines

[Winter is coming]

[Winter is coming]

We challenge you to implement a dynamic array or vector in C. We will upload an implementation.

I think we are done for the day(and for this semester) but If you wish to talk to us you can stay here and share your thoughts. We also want you to post questions in the forum and respond(It is package time). This is not a classroom, It is a community. We want you to interact and solve interesting problem, learn cool stuff and make sure you have fun through programming


Hacktober
FEST
2019
presented by
DigitalOcean
and
DEV