MONTANA
STATE UNIVERSITY

Building Code

The Nitty Gritty

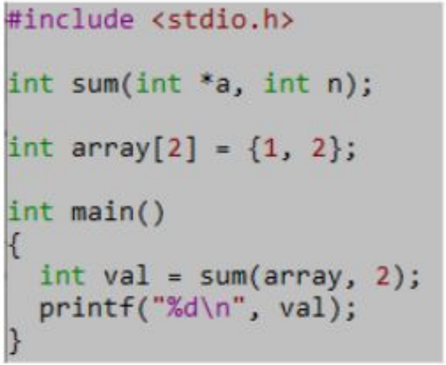
What Is “Building” Code?

- Using a “tool chain” (a set of tools) to convert textual data into a binary file that makes sense to a computer
- Common tool chain elements:
  - A compiler - turns code into an intermediate representation or unliked-binary
  - An assembler - turns assembly into unlinked-binary
  - A linker - turns unlinked-binaries into linked binaries
- Also potentially part of your tool chain
  - Source control
  - Test harness
  - Code verification

The Compiler

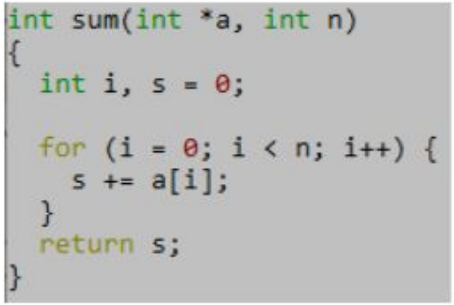
- Takes source code as an input (e.g. C, C++ or ASM)
- Produces a binary file (usually) as an output
  - Relocatable Object Files
- gcc is the standard compiler on linux
  - GNU C Compiler

main.c



```c
#include <stdio.h>

int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
  int val = sum(array, 2);
  printf("%d\n", val);
}
```

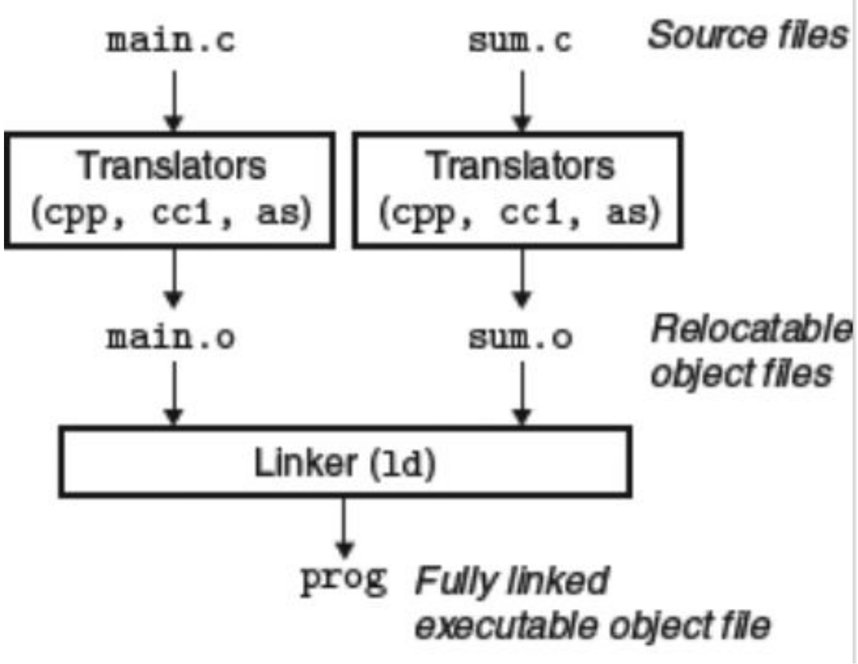
sum.c



```c
int sum(int *a, int n)
{
  int i, s = 0;

  for (i = 0; i < n; i++) {
    s += a[i];
  }
  return s;
}
```

The Compiler

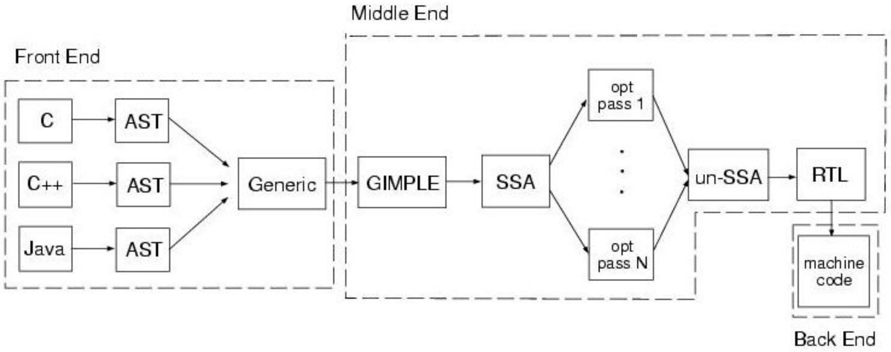
- Takes source code as an input (e.g. C, C++ or ASM)
- Produces a binary file (usually) as an output
  - Relocatable Object Files
- gcc is the standard compiler on linux
  - GNU C Compiler



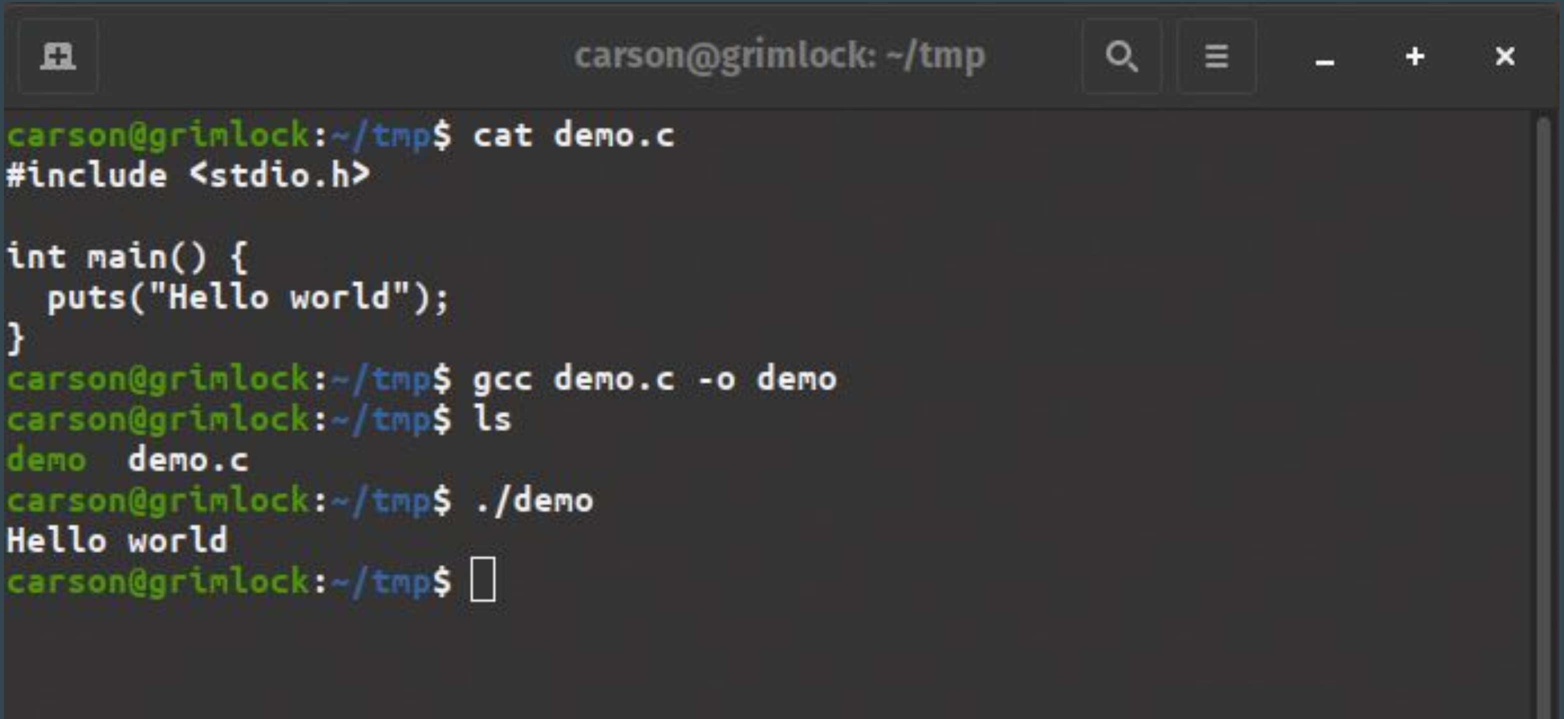
main.c
sum.c
Source files
Translators (cpp, cc1, as)
Translators (cpp, cc1, as)
main.o
sum.o
Relocatable object files
Linker (ld)
prog
Fully linked executable object file

The Compilers Components

- Front End
  - Pre-processor
  - Source code to a generic C-like Abstract Syntax Tree (AST)
- Middle End
  - Transformations of AST
  - Optimizations, simplifications, etc.
- Back End
  - Production of machine code



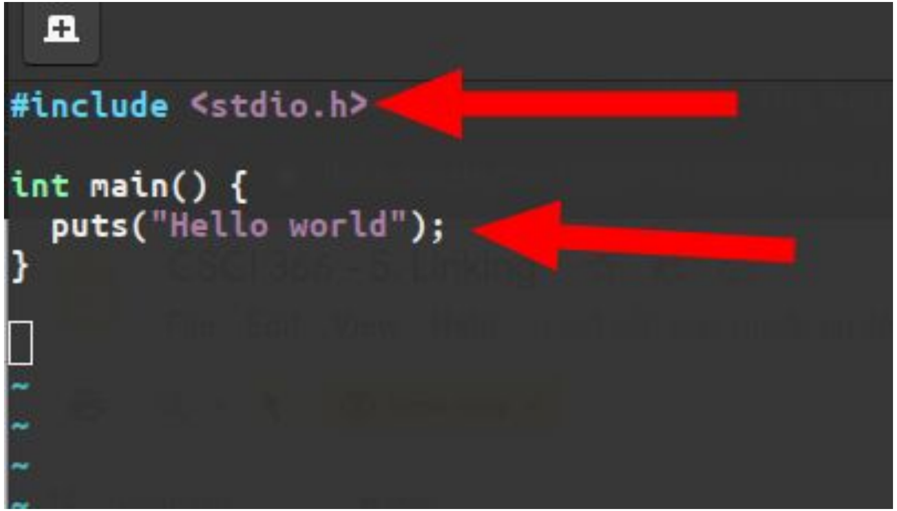
On The Command Line



```
carson@grimlock:~/tmp$ cat demo.c
#include <stdio.h>

int main() {
  puts("Hello world");
}
carson@grimlock:~/tmp$ gcc demo.c -o demo
carson@grimlock:~/tmp$ ls
demo  demo.c
carson@grimlock:~/tmp$ ./demo
Hello world
carson@grimlock:~/tmp$ 
```

Libraries

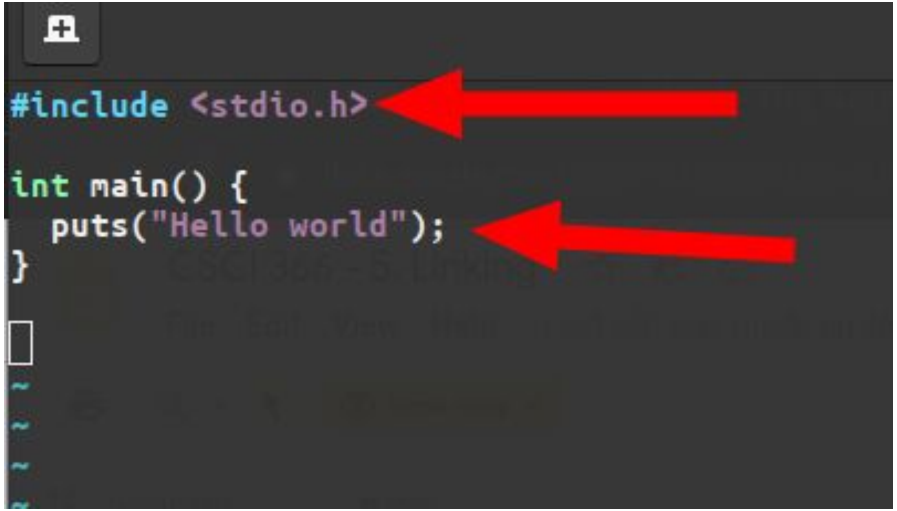
- You don’t want to write everything from scratch!
- In this example we are including the standard IO library (stdio)
- We reference a *header file*
  - Header files provide *declarations*
  - Implementations are elsewhere



```
#include <stdio.h>

int main() {
  puts("Hello world");
}

~
~
~
~
```

Libraries

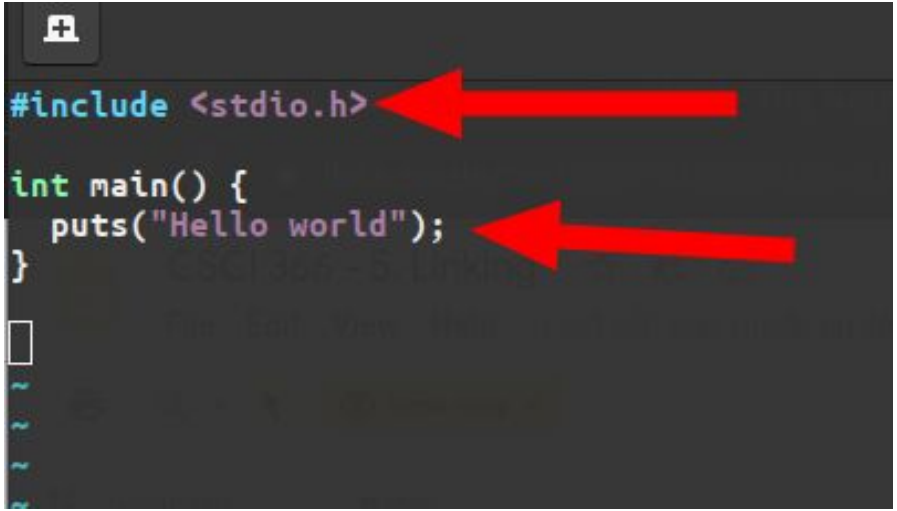
- You must include a header file to use functions from a library
  - Unless you us an extern hack :)
- The inclusions are handled by the C Preprocessor
- Angle brackets indicate a system library
- Quotes indicate a local library



```
#include <stdio.h>

int main() {
  puts("Hello world");
}
```

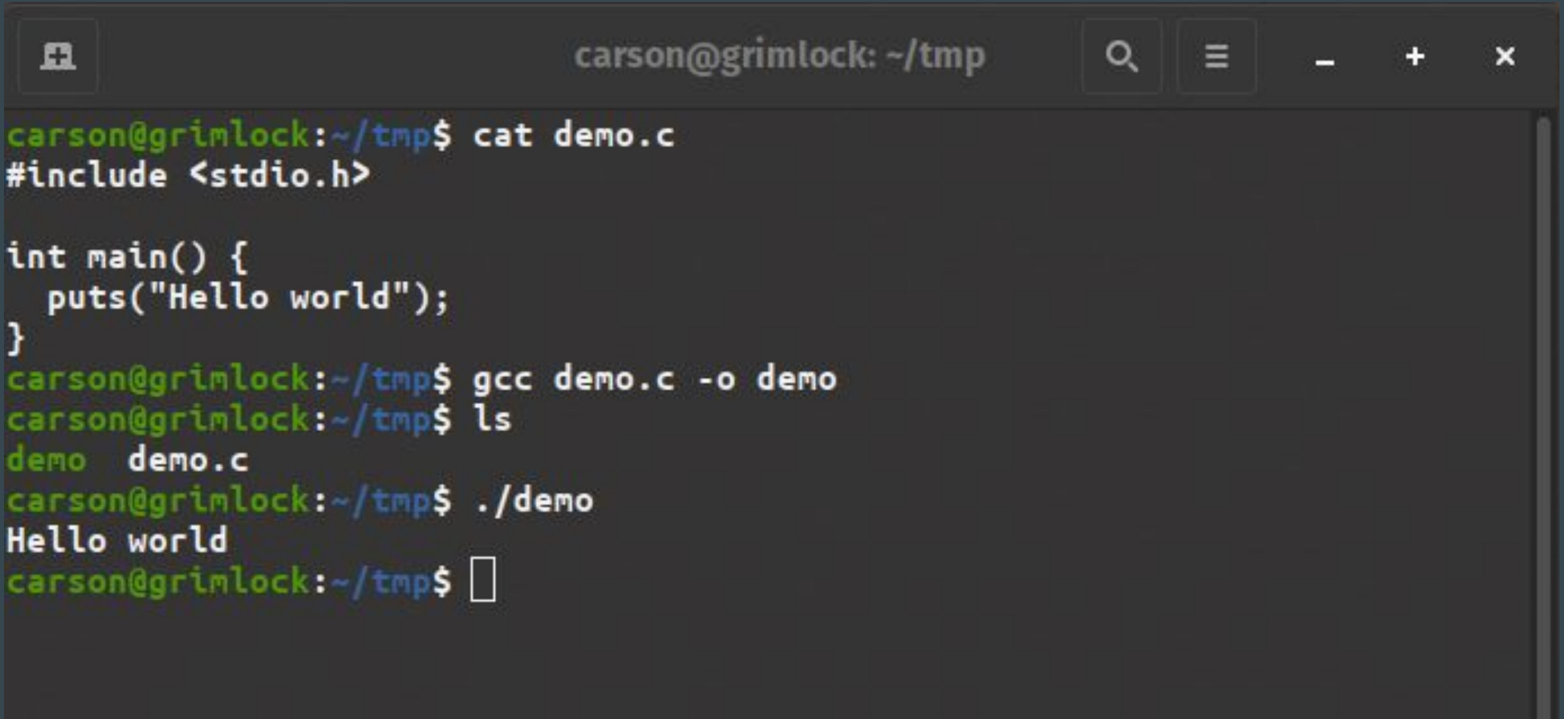
Libraries

- Header file allows you to access the functions of a library, but doesn't provide an implementation
- The implementation is hooked in later, via the linker
- We will talk more about header files when we get into C programming



```
#include <stdio.h>

int main() {
  puts("Hello world");
}
```

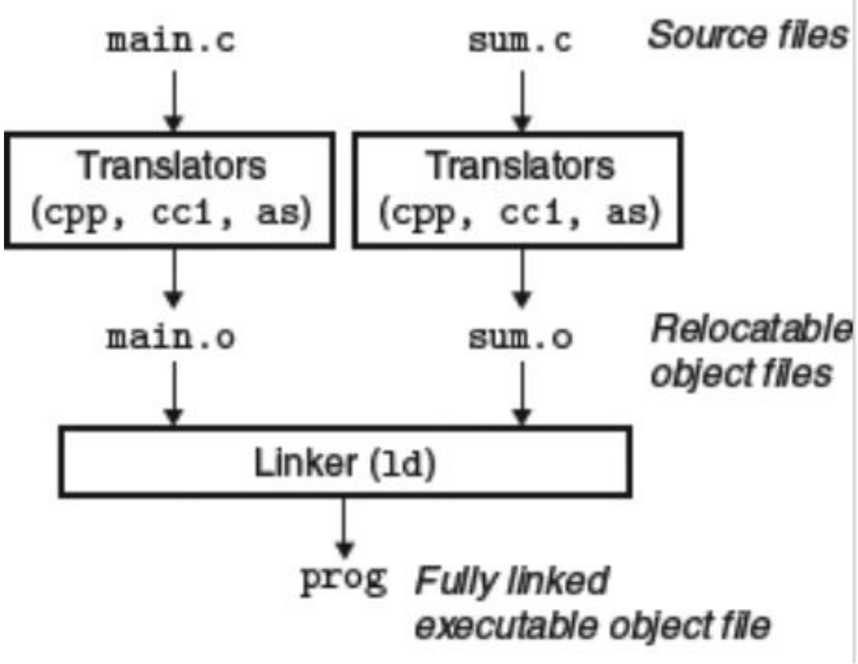
Wait, Where's the Linker?!?



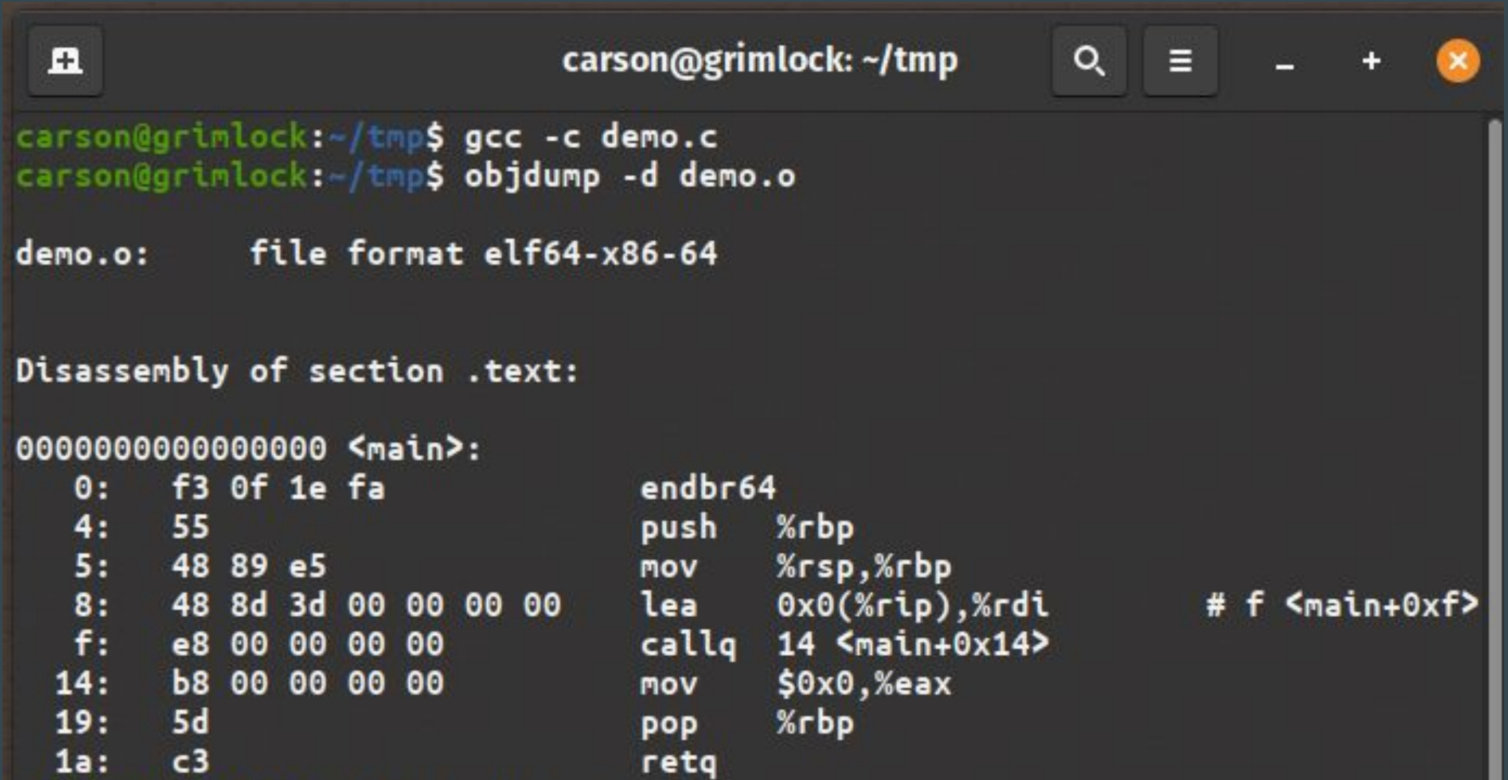
```
carson@grimlock:~/tmp$ cat demo.c
#include <stdio.h>

int main() {
  puts("Hello world");
}
carson@grimlock:~/tmp$ gcc demo.c -o demo
carson@grimlock:~/tmp$ ls
demo  demo.c
carson@grimlock:~/tmp$ ./demo
Hello world
carson@grimlock:~/tmp$ 
```

The Linker

- Takes relocatable object files as an input (e.g. a.out)
- Produces a binary file that has been linked up properly
- ld is the standard linker on linux
    - The GNU linker
    - Why not gl?
    - ¯_(ツ)_/¯


main.c
sum.c
Source files
Translators (cpp, cc1, as)
Translators (cpp, cc1, as)
main.o
sum.o
Relocatable object files
Linker (ld)
prog
Fully linked executable object file

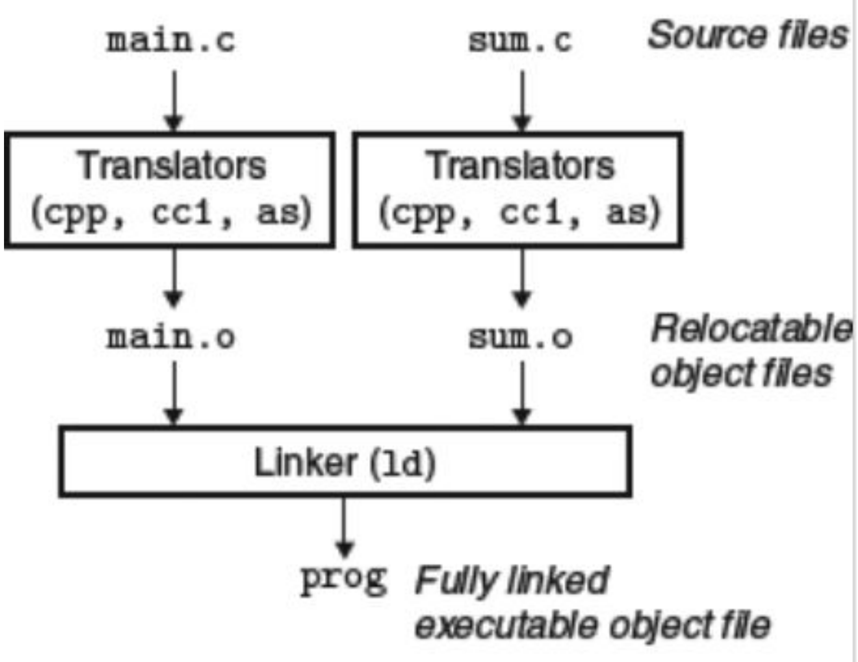
gcc -c “Don’t Link”



```
carson@grimlock:~/tmp$ gcc -c demo.c
carson@grimlock:~/tmp$ objdump -d demo.o

demo.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:	f3 0f 1e fa          	endbr64 
   4:	55                   	push   %rbp
   5:	48 89 e5             	mov    %rsp,%rbp
   8:	48 8d 3d 00 00 00 00 	lea    0x0(%rip),%rdi        # f <main+0xf>
   f:	e8 00 00 00 00       	callq  14 <main+0x14>
  14:	b8 00 00 00 00       	mov    $0x0,%eax
  19:	5d                   	pop    %rbp
  1a:	c3                   	retq
```

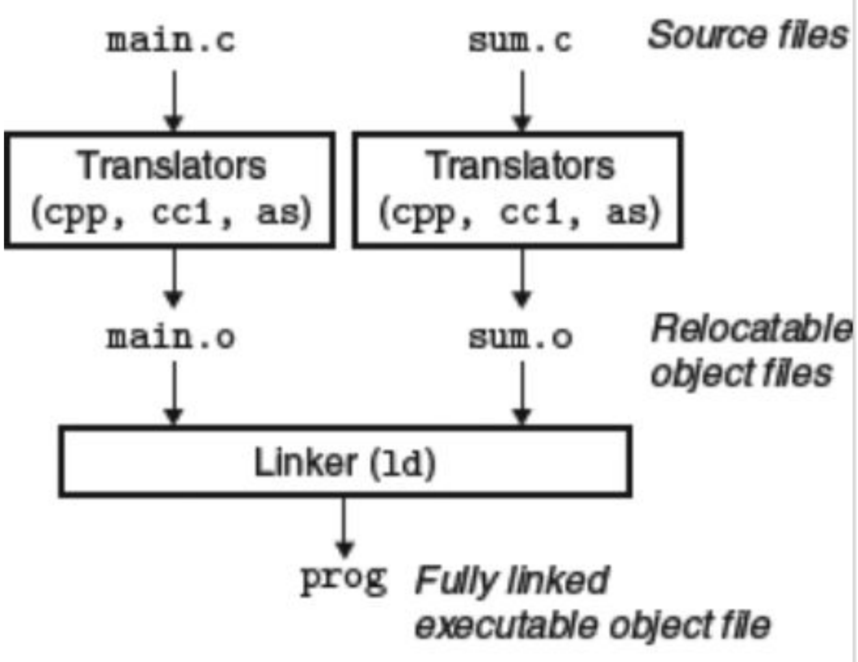
The Linker

- What does “linked up” mean?
- Depends on what you are talking about!
- Statically Linked:
  - Code or variables are moved to specific positions and references are updated
- Dynamically Linked:
  - same process, but done at at **runtime**



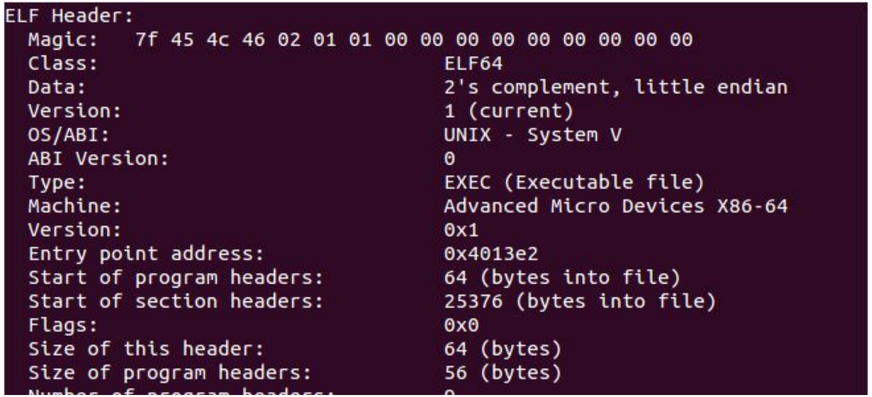
The Linker

- Why statically linked?
  - No runtime linking errors
  - Known library versions
- Why dynamically linked?
  - Smaller generated code
  - Libraries can be updated to fix bugs



ELF Files

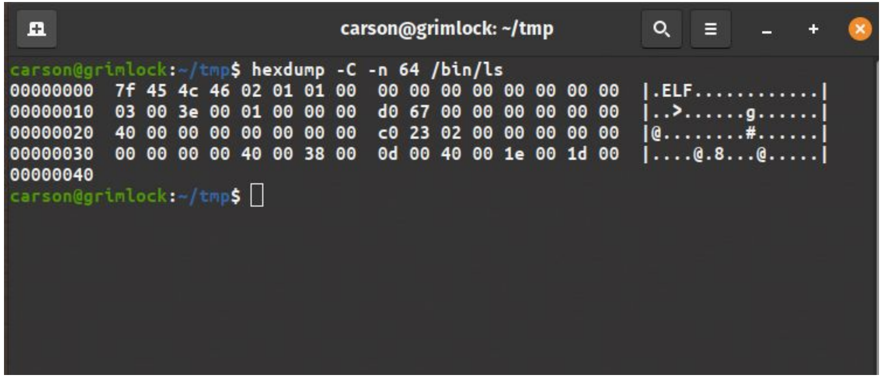
- "Executable & Linkable" Format
- Standard binary file format for most *nix systems
- Invented way back in 1989
- File consists of
  - Header
  - Sections
  - Linking information



```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4013e2
  Start of program headers:          64 (bytes into file)
  Start of section headers:          25376 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         0
```

ELF Files

- You can examine files with the command objdump, readelf, hexdump
- Sections
  - .text - executable code
  - .data - read/write data
  - .rodata - read only data
  - .bss - read/write, uninitialized
- Lots of stuff, but remember: It's just a file format!



```
carson@grimlock:~/tmp$ hexdump -C -n 64 /bin/ls
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  d0 67 00 00 00 00 00 00  |..>......g......|
00000020  40 00 00 00 00 00 00 00  c0 23 02 00 00 00 00 00  |@........#......|
00000030  00 00 00 00 40 00 38 00  0d 00 40 00 1e 00 1d 00  |....@.8...@.....|
00000040
carson@grimlock:~/tmp$ 
```

Managing Builds

•••

Make & CMake

What’s a Build Tool?

- Software that manages the build process for a program
- As projects grow in size, building them with simple scripts becomes untenable
    - Some people still prefer plain scripts

Make

- One of the first build tools, created in April 1976
  - Older than me!
- Keeps track of modified files and recompiles them
- Included in early Unix
- Specify build in a Makefile
  - Syntax is obscure and touchy
  - White space sensitive

CMake

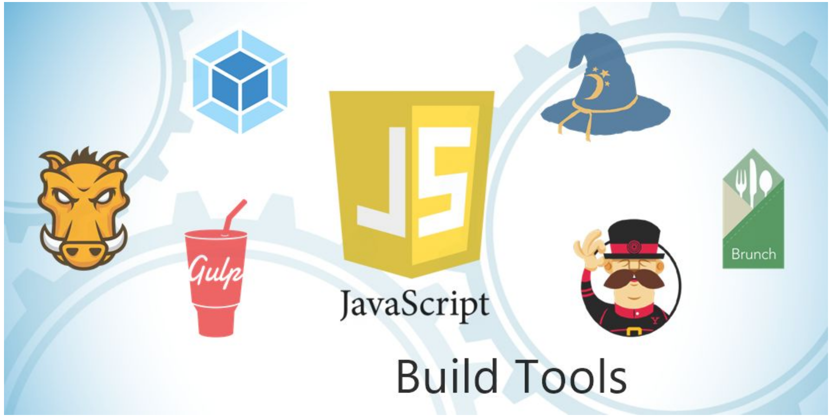
- Introduced in 1999
- Big rewrite in 2014: Version 3
- Considered the “modern” C/C++ build tool
- Build File is CMakeLists.txt
    - Less obscure than Makefile
    - Still uses Make files internally
    - Look in cmake-build-debug folder!

CMake

- Syntax is more project oriented rather than file oriented
- Integrates with CPack, a system for including libraries with your project
- CLion understands CMakeLists.txt!

Other Languages

- There are a million build tools out there for various languages
- Java has Maven, Ant and a few others
- Javascript has tens of them, and new ones popping up every few months



CS 366

- We will be using CMake
- You will not be responsible for dealing with it
    - You’re welcome!
- In corporate environments there is often an entire build team

Building Software Sucks

- Build systems are usually no fun to work with
- Best case scenario you don’t notice them much
- Worst case scenario is hours or even days of lost time due to an obscure issue
- Let us handle the build issues!

MONTANA

STATE UNIVERSITY