

CSc 352

# Object Files, Executables, Linking

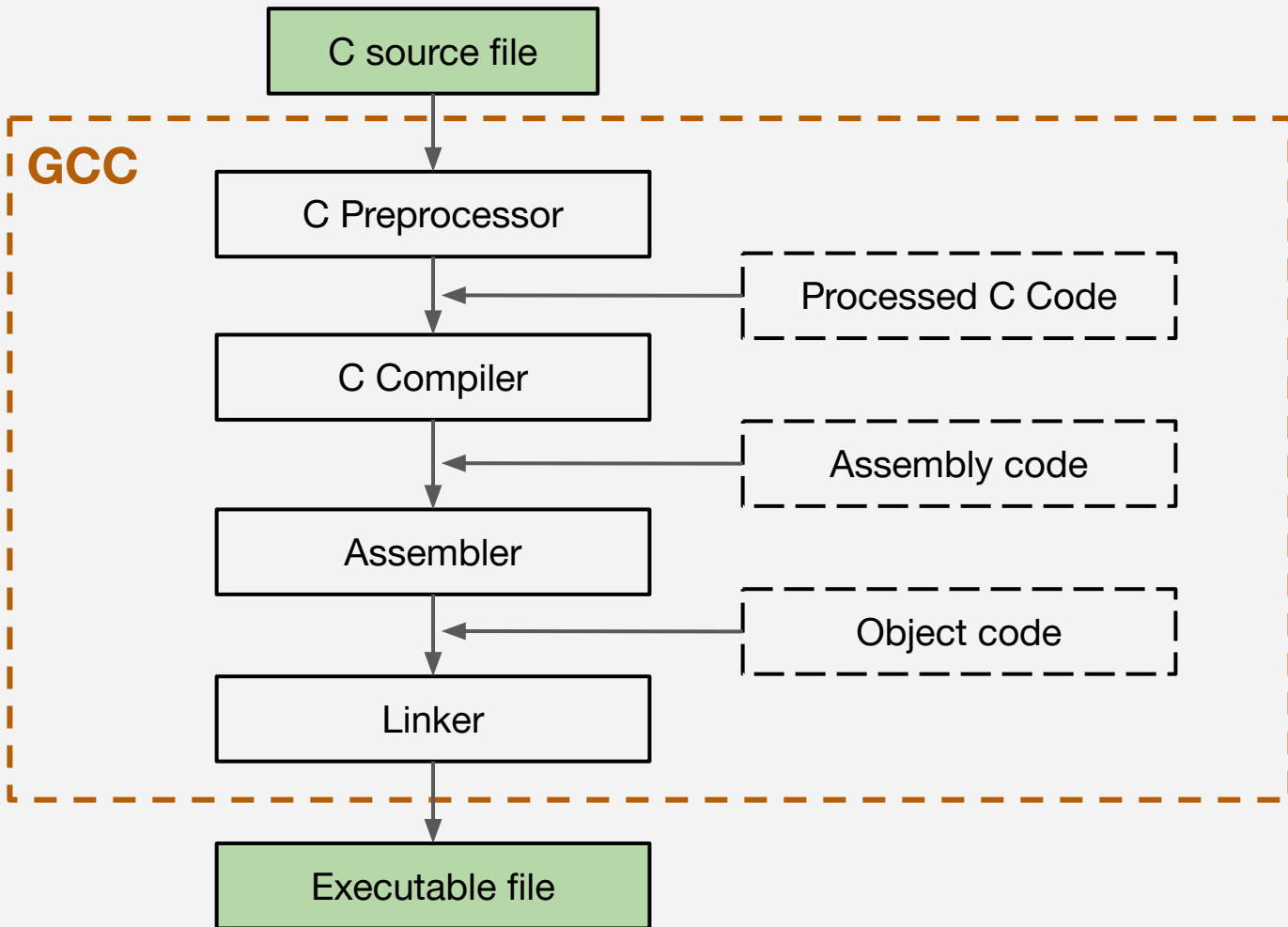
Benjamin Dicken

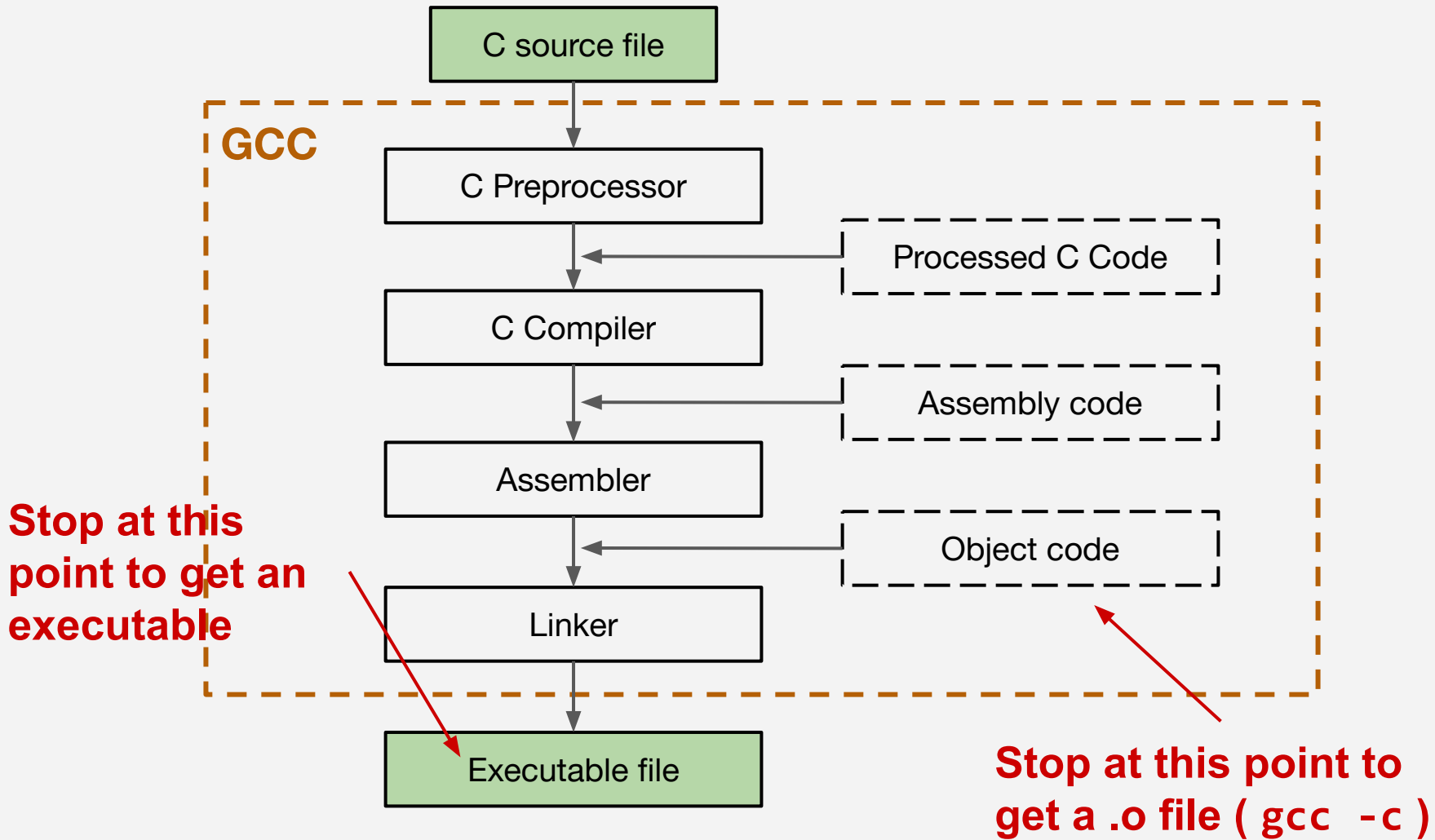
# Announcements

- Student Course Survey
  - 85.26% (Today)
- PA 10
- Final exam, May 6, 1-3pm, this room
  - Study guide for Final will be posted Friday

# Compiling to Bytecode

- When we compile a program with GCC, the eventual goal is to get binary code
  - Either executable, or object file (to be linked with other code)





# Going through the steps

```
$ cpp test.c -o /tmp/test.i
```

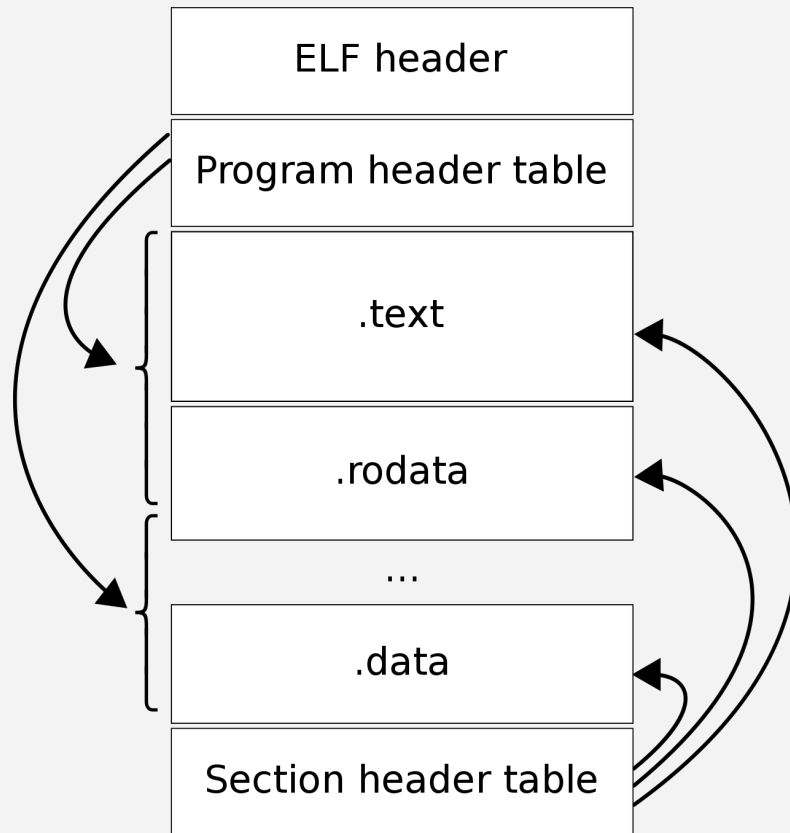
```
$ cc -S -fverbose-asm -static /tmp/test.i -o /tmp/test.s
```

```
$ as /tmp/test.s -o /tmp/test.o
```

```
$ ld -e main -lc /tmp/test.o -o a.out
```

# ELF

- Executable and Linkable Format (ELF) is the standard format used for executable files and object files on UNIX systems
- The ELF format specifies where various parts of the program go (the code, the constants, the symbol table, etc)
- Remember: Files are just a bunch of 1s and 0s - We can choose how to interpret these files!



[https://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format)



# Reading ELF Files

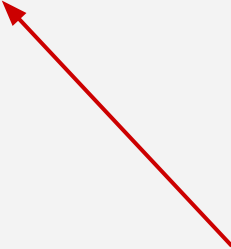
- `objdump -s` Display full contents in hex
- `objdump -d -S` Display disassembled code, with source intermixed
- `readelf -h` Display the header info
- `readelf -a` Display all the info

# Explore with a simple file

```
#include <stdio.h>
int main() {
    printf("sup\n");
    return 0;
}
```

# From output of objdump

```
0000000000001149 <main>:
#include <stdio.h>
int main() {
    1149: f3 0f 1e fa          endbr64
    114d: 55                   push   %rbp
    114e: 48 89 e5            mov    %rsp,%rbp
    printf("sup\n");
    1151: 48 8d 3d ac 0e 00 00 lea    0xeac(%rip),%rdi    # 2004 <_IO_stdin_used+0x4>
    1158: e8 f3 fe ff ff      callq 1050 <puts@plt>
    return 0;
    115d: b8 00 00 00 00      mov    $0x0,%eax
}
    1162: 5d                   pop    %rbp
    1163: c3                   retq
    1164: 66 2e 0f 1f 84 00 00 nopw   %cs:0x0(%rax,%rax,1)
    116b: 00 00 00
    116e: 66 90               xchg  %ax,%ax
```



**What happens  
when we call a  
standard library  
function?**

## From output of readelf -a

Symbol table '.dynsym' contains 7 entries:

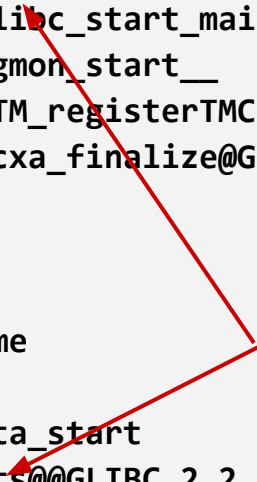
Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_ITM_deregisterTMCloneTab
2:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	puts@GLIBC_2.2.5 (2)
3:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.2.5 (2)
4:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
5:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_ITM_registerTMCloneTable
6:	0000000000000000	0	FUNC	WEAK	DEFAULT	UND	__cxa_finalize@GLIBC_2.2.5 (2)

. . .

Symbol table '.symtab' contains 70 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
53:	0000000000004000	0	NOTYPE	WEAK	DEFAULT	25	data_start
54:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	puts@@GLIBC_2.2.5
55:	0000000000004010	0	NOTYPE	GLOBAL	DEFAULT	25	_edata

. . .



# Compare and Contrast

Take the test program and compile in two ways:

```
$ gcc -Wall -Werror -std=c11 -g test.c -o dynamic
```

```
$ gcc -static -Wall -Werror -std=c11 -g test.c -o static
```

What is the difference?

Investigate with `objdump` and `readelf`

```
#include <stdio.h>
int main() {
    printf("sup\n");
    return 0;
}
```

# Compare and Contrast

Take the test program and compile in two ways:

```
$ gcc -Wall -Werror -std=c11 -g test.c -o dynamic
```

```
$ gcc -Wall -Werror -std=c11 test.c -o dynamic
```

What is the difference?

Investigate with `objdump` and `readelf`

```
#include <stdio.h>
int main() {
    printf("sup\n");
    return 0;
}
```

# Compare and Contrast

Take the test program and compile in two ways:

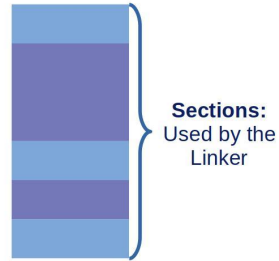
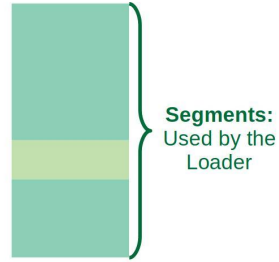
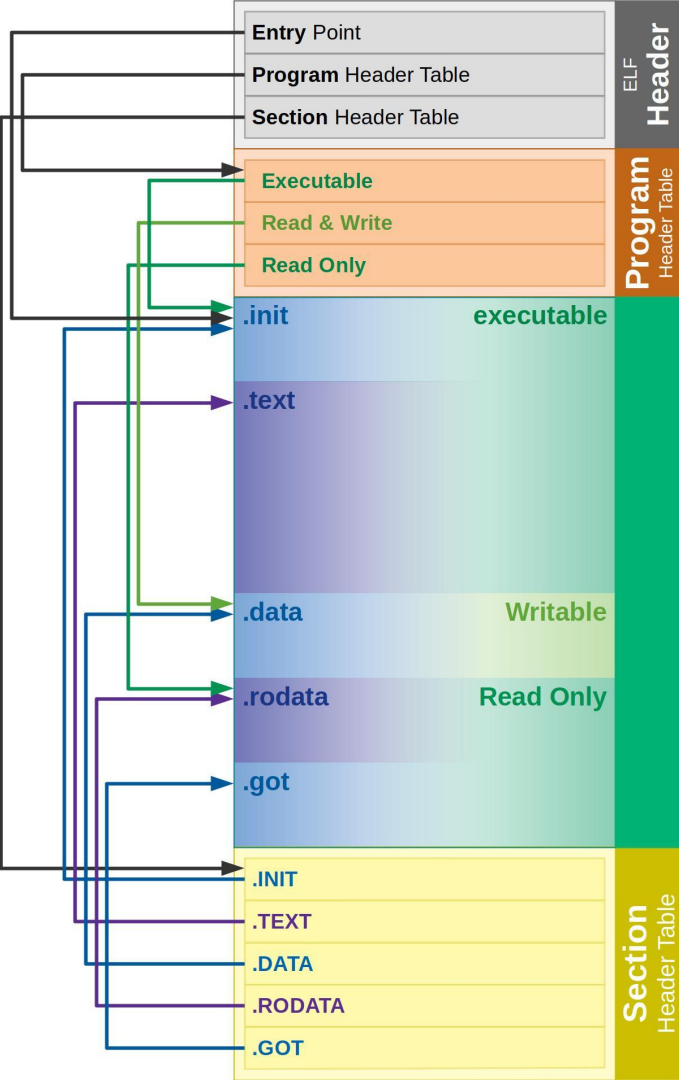
```
$ gcc -Wall -Werror -std=c11 -c test.c -o dynamic.o
```

```
$ gcc -Wall -Werror -std=c11 test.c -o dynamic
```

What is the difference?

Investigate with `objdump` and `readelf`

```
#include <stdio.h>
int main() {
    printf("sup\n");
    return 0;
}
```





# Linker and Loader

- ELF Files are used by two steps: The **Linker** and the **Loader**
- In the **linking** step, combining code from multiple ELF files together (if needed)
- In the **Loading** step, copying an ELF file into memory to prepare for execution
- See: <https://www.ics.uci.edu/~aburtsev/238P/hw/hw3-elf/hw3-elf.html>

# Loader

In Unix, the loader is the handler for the system call `execve()`. The Unix loader's tasks include:

- validation (permissions, memory requirements etc.);
- copying the program image from the disk into main memory;
- copying the command-line arguments on the stack;
- initializing registers (e.g., the stack pointer);
- jumping to the program entry point (`_start`).

[https://en.wikipedia.org/wiki/Loader\\_\(computing\)](https://en.wikipedia.org/wiki/Loader_(computing))

# Load a program

```
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("me!\n");
    char* arguments[] = {NULL};
    char* environment[] = {NULL};
    int value = execve("./a.out", arguments, environment);
    if (value == -1) {
        printf("Failed to execute\n");
    }
    return 0;
}
```