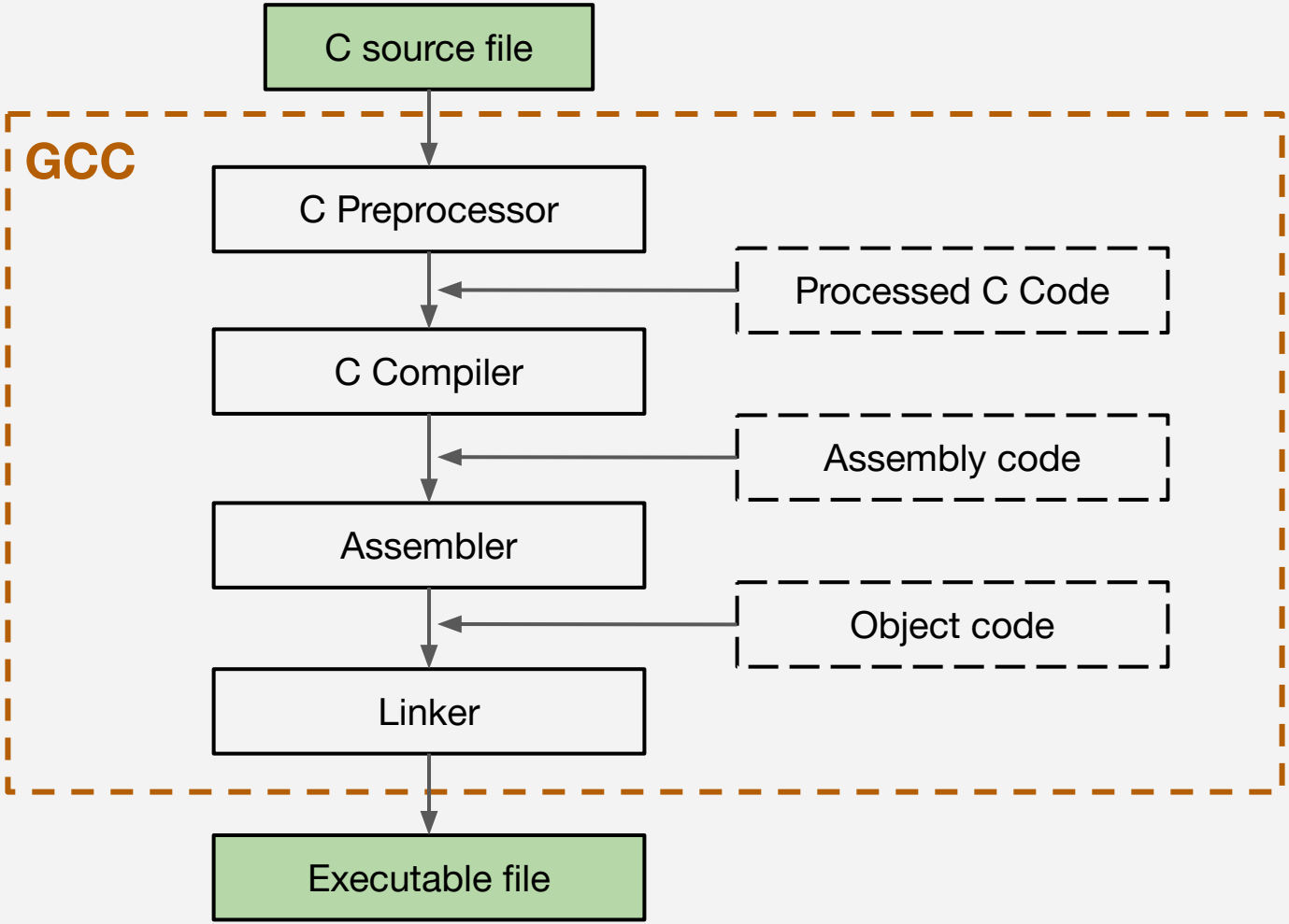
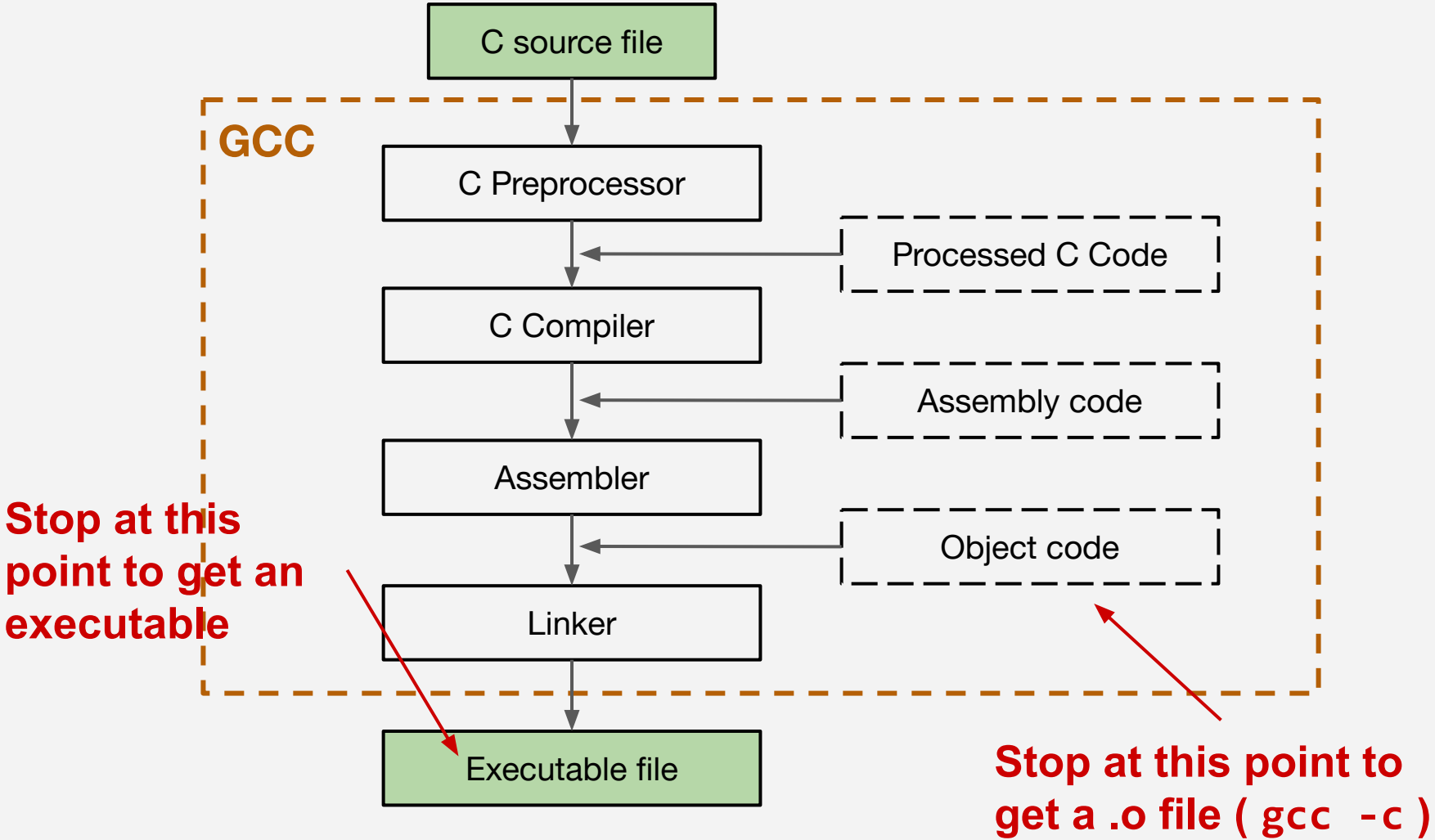
CSc 352
Compilation, ELF

Benjamin Dicken

Compiling to Bytecode

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


C source file
GCC
C Preprocessor
Processed C Code
C Compiler
Assembly code
Assembler
Object code
Linker
Executable file


C source file
GCC
C Preprocessor
Processed C Code
C Compiler
Assembly code
Assembler
Object code
Linker
Executable file
Stop at this point to get an executable
Stop at this point to get a .o file (gcc -c)

Going through the steps

```
$ cpp test.c -E -o /tmp/test.i
$ gcc /tmp/test.i -S -o /tmp/test.s
$ as /tmp/test.s -o /tmp/test.o
$ ld /tmp/test.o
$ objdump -D -t /tmp/test.o > /tmp/test.d
$              # gcc -v test.c should help
```

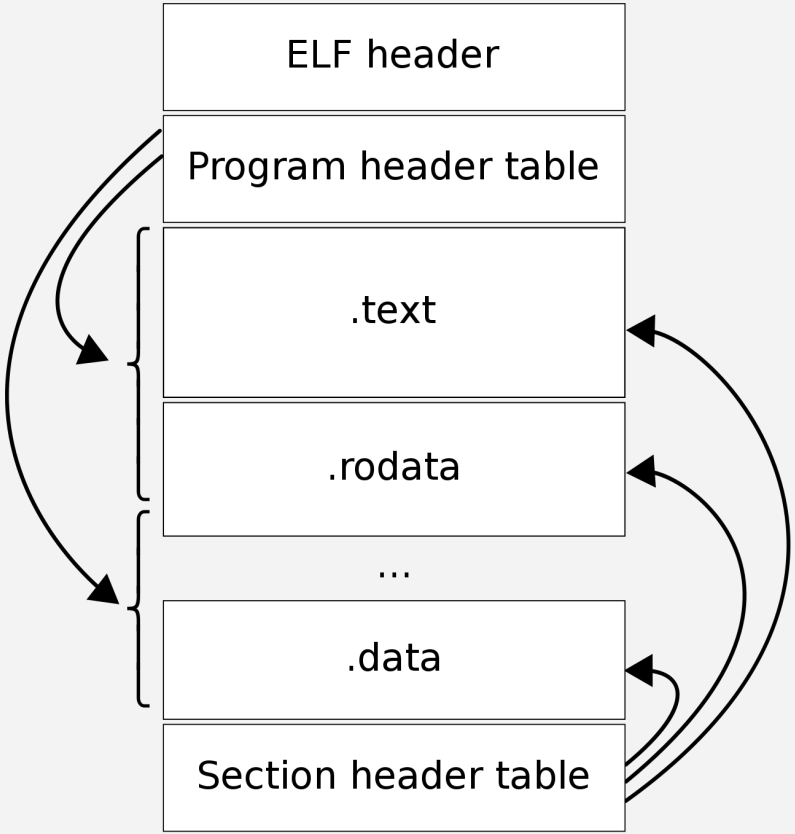
https://medium.com/@kunaljaydesai/understanding-linking-8709e2cc450e
https://medium.com/swlh/deep-dive-into-static-linking-c3b1f459c99d

Going through the steps

```
ld -plugin /usr/lib/gcc/x86_64-linux-gnu/9/liblto_plugin.so
-plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper -plugin-opt=-fresolution=/tmp/cc4CKbXW.res
-plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc
-plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --eh-frame-hdr -m elf_x86_64
--hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -z now -z relro
/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/Scrt1.o
/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/9/crtbeginS.o
-L/usr/lib/gcc/x86_64-linux-gnu/9 -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu
-L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib
-L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/9/../../.. /tmp/test.o
-lgcc --push-state --as-needed -lgcc_s --pop-state -lc -lgcc --push-state --as-needed -lgcc_s --pop-state
/usr/lib/gcc/x86_64-linux-gnu/9/crtendS.o /usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crtn.o
```

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

Reading ELF Files

- **`objdump -s`** Display full contents in hex
- **`objdump -d -S -h`** Display disassembled code, w/ source, headers
- **`readelf -h`** Display the header info
- **`readelf -a`** Display all the info

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

const int abc = 100;
const char name[] = "Incredible";

int main(int argc, char* argv[]) {
  printf("hi\n");
  printf("hi");
  printf("abc: %d\n", abc);
  printf("name: %s\n", name);

  int length1 = strlen(name);
  printf("length1: %d\n", length1);

  char * x = malloc(150);
  int length2 = strlen(x);
  printf("length2: %d\n", length2);

  return 0;
}
```

Explore with this file

/tmp/test.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

const int abc = 100;
const char name[] = "Incredible";

int main(int argc, char* argv[]) {
  printf("hi\n");
  printf("hi");
  printf("abc: %d\n", abc);
  printf("name: %s\n", name);

  int length1 = strlen(name);
  printf("length1: %d\n", length1);

  char * x = malloc(150);
  int length2 = strlen(x);
  printf("length2: %d\n", length2);

  return 0;
}
```

Explore

(1) Compile (with -g)
(2) **`$ objdump -d -S -h a.out`**
(3) Investigate main
(4) How many calls to printf, puts, strlen do you see.
(5) Why?

`/tmp/test.c`

From output of objdump

```
. . . .
  int length1 = strlen(name);
    1209: c7 45 f0 0a 00 00 00  movl   $0xa,-0x10(%rbp)
  printf("length1: %d\n", length1);
    1210: 8b 45 f0              mov    -0x10(%rbp),%eax
    1213: 89 c6                 mov    %eax,%esi
    1215: 48 8d 3d 15 0e 00 00  lea    0xe15(%rip),%rdi        # 2031 <name+0x21>
    121c: b8 00 00 00 00        mov    $0x0,%eax
    1221: e8 7a fe ff ff        callq  10a0 <printf@plt>

  char * x = malloc(150);
    1226: bf 96 00 00 00        mov    $0x96,%edi
    122b: e8 80 fe ff ff        callq  10b0 <malloc@plt>
    1230: 48 89 45 f8           mov    %rax,-0x8(%rbp)
  int length2 = strlen(x);
    1234: 48 8b 45 f8           mov    -0x8(%rbp),%rax
    1238: 48 89 c7              mov    %rax,%rdi
    123b: e8 50 fe ff ff        callq  1090 <strlen@plt>
    1240: 89 45 f4              mov    %eax,-0xc(%rbp)
. . . .
```

Why no strlen call here?

But there is here?

From output of objdump

```
int main(int argc, char* argv[]) {
    11a9: f3 0f 1e fa           endbr64
    11ad: 55                    push   %rbp
    11ae: 48 89 e5              mov    %rsp,%rbp
    11b1: 48 83 ec 20           sub    $0x20,%rsp
    11b5: 89 7d ec              mov    %edi,-0x14(%rbp)
    11b8: 48 89 75 e0           mov    %rsi,-0x20(%rbp)
  printf("hi\n");
    11bc: 48 8d 3d 58 0e 00 00  lea    0xe58(%rip),%rdi        # 201b <name+0xb>
    11c3: e8 b8 fe ff ff        callq  1080 <puts@plt>
  printf("hi");
    11c8: 48 8d 3d 4c 0e 00 00  lea    0xe4c(%rip),%rdi        # 201b <name+0xb>
    11cf: b8 00 00 00 00        mov    $0x0,%eax
    11d4: e8 c7 fe ff ff        callq  10a0 <printf@plt>
. . . .
```

Why puts here?

And then printf here?

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**

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

Linking

- In the **linking** step, combining code from multiple ELF files together If needed)
- Can link from other .o file that are a part of your project
- Can also link to other shared object files, such as the standard library

Static and Dynamic Linking

- In **Static Linking** the code from the file being linked together is actually included in the executable (larger file size)
- With **Dynamic linking**, the other symbols (functions, data) are dynamically linked at **runtime**

https://stackoverflow.com/questions/311882

Compare and Contrast

Take the test program and compile in two ways:

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

What is the difference?

Investigate with **objdump** and **readelf**

```c
#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**

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