How Programs Are Made

10

CS/COE 0449
Introduction to
Systems Software

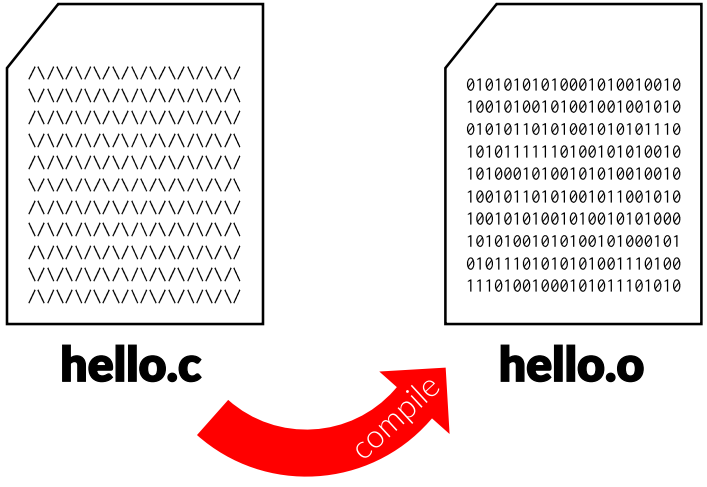
wilkie
(with content borrowed from Vinicius Petrucci
and Jarrett Billingsley)

Spring 2019/2020

Linkers

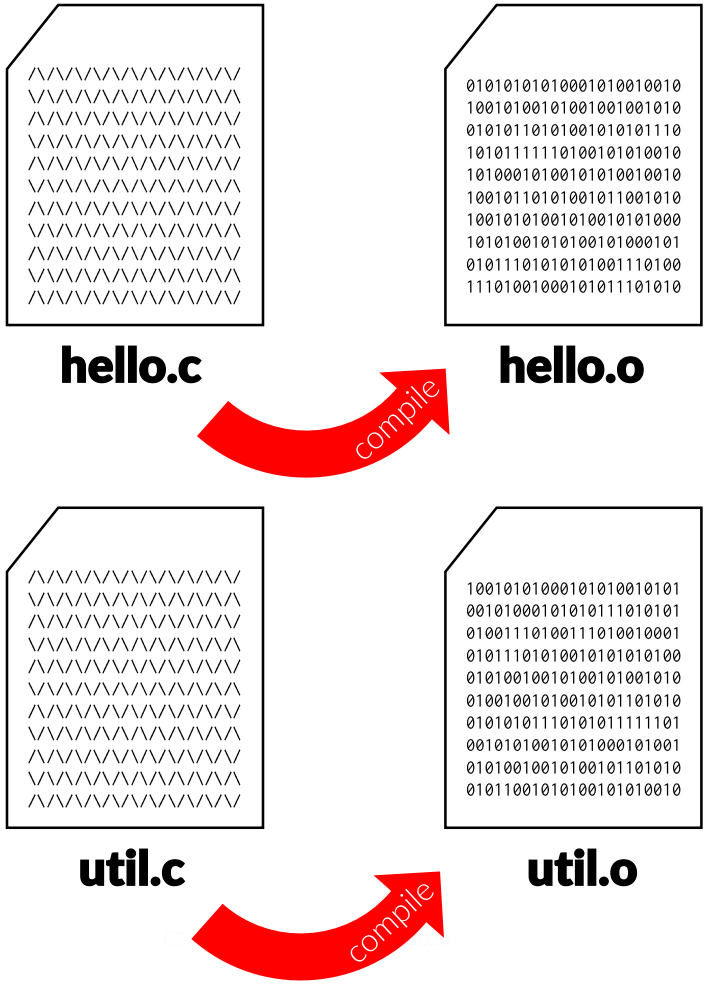
Filling in the blanks.

Compilation: Simple Overview – Step 1



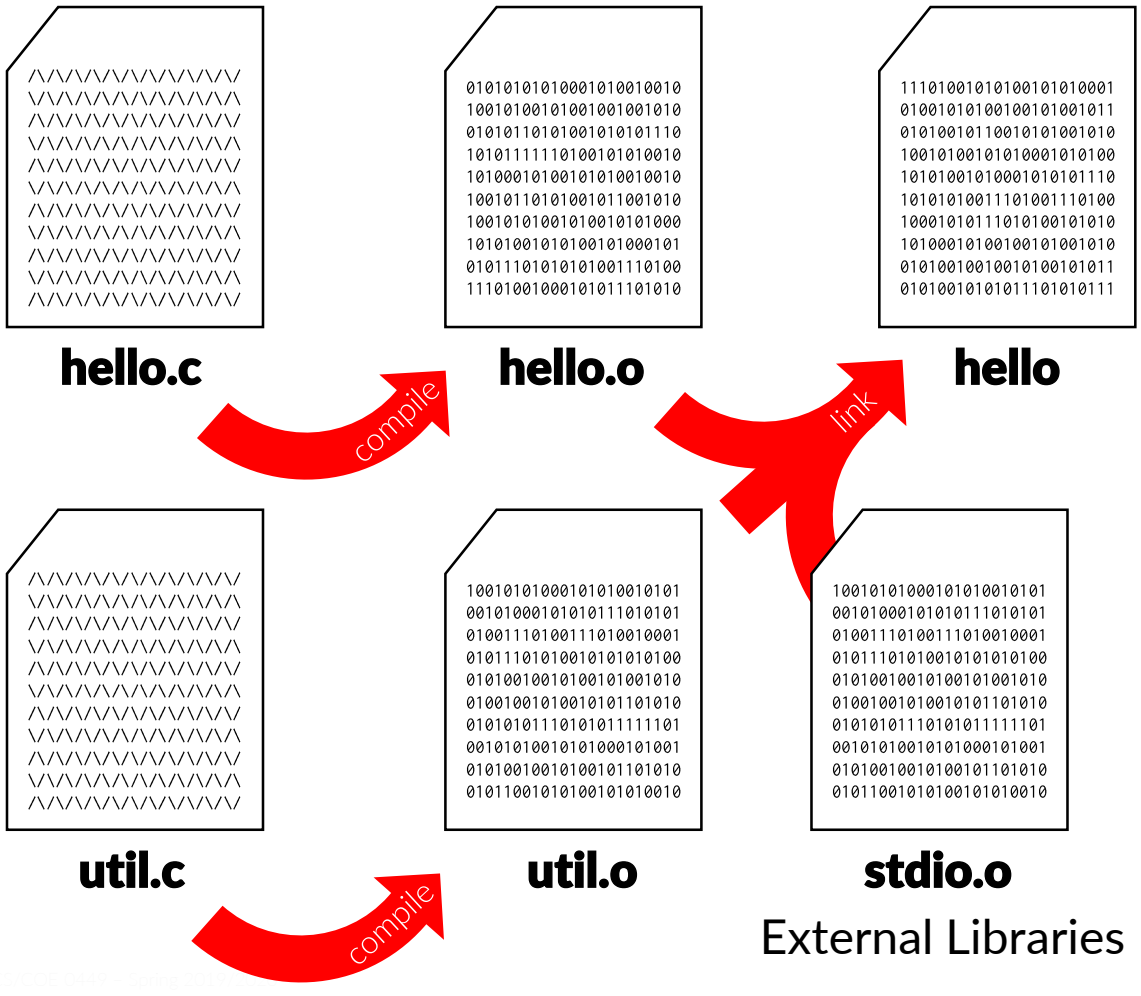
- The compiler takes source code (`*.c` files) and translates them into machine code.
- This file is called an "***object file***" and is just potentially one part of your overall project.
- The machine code is not quite an executable.
  - This object file is JUST representing the code for that particular source file.
  - You may require extra stuff provided by the system elsewhere.

Compilation: Simple Overview – Step 2



- You may have multiple files.
- They may reference each other.
  - For instance, one file may contain certain common functionality and then this is invoked by your program elsewhere.
- You break your project up into pieces similarly to your Java programs.
- The compiler treats them independently.

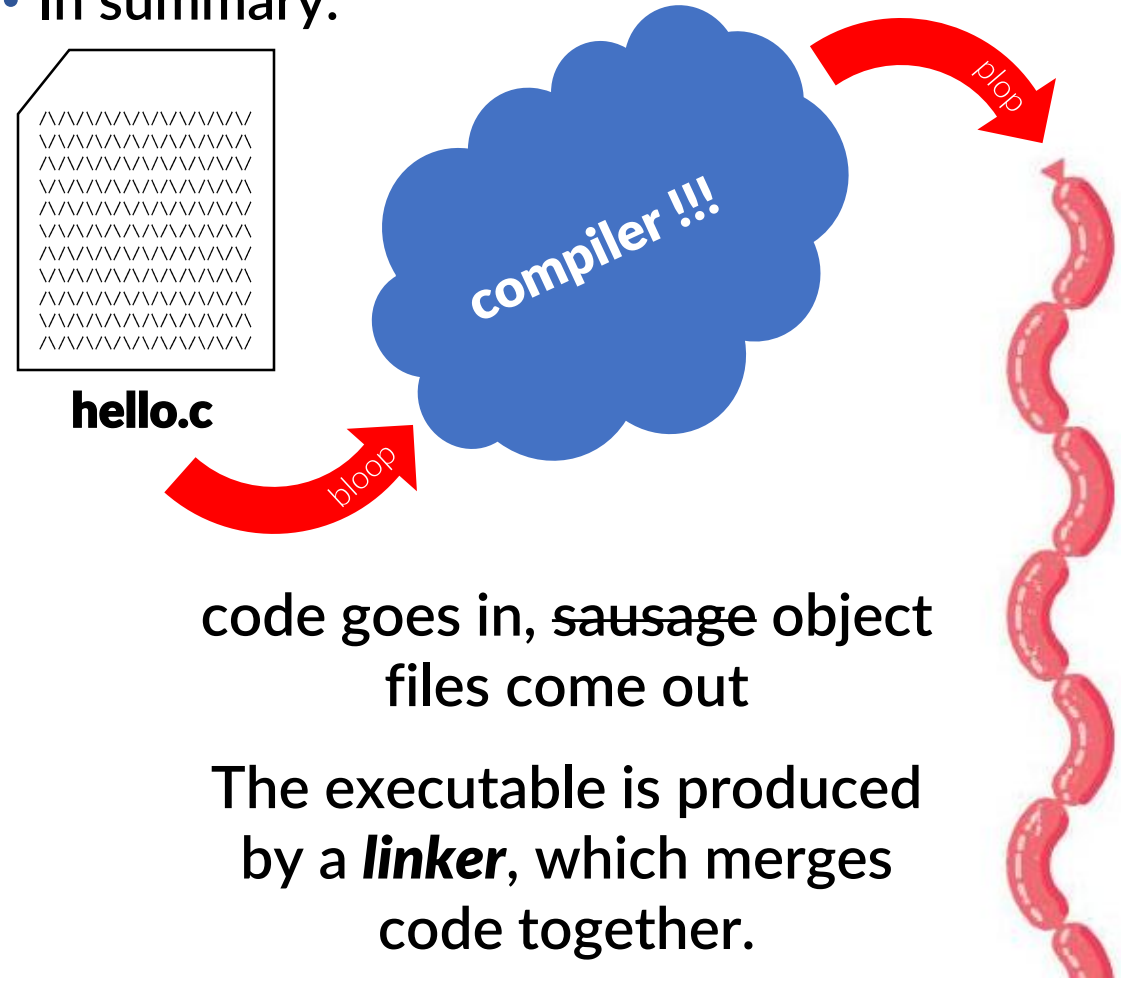
Compilation: Simple Overview – Step 3



- Then, each piece is merged together to form the executable.
- This process is done by a ***linker*** and is called ***linking***.
  - The name refers to how the references to functions, etc, between files are now filled in.
  - Before this step… it is unclear *where* functions will end up in the final executable.

It's just a grinder.

- In summary:



code goes in, ~~sausage~~ object files come out

The executable is produced by a ***linker***, which merges code together.

Some compilers output assembly and rely on an assembler to produce machine code

These days, it's common for the compiler itself to produce machine code, or some kind of platform-independent assembly code (typically: a bytecode)

The need for the linker

- A **compiler** converts source code into machine code.
- A **linker** merges pieces of machine code into an executable.

- Why have a separate tool for creating executables?
  - Mixing different languages together (C, C++, Python, Rust, Go…)
    - Lot's of complications we won't get to here.
    - Assembly is the glue… all high-level languages have to get there.
  - Let's us break large programs up into smaller pieces.
    - And we only have to recompile files that changed! (Faster)
  - Those small pieces can come from others. Code reuse!
    - We can share executable code among many running programs. (*Shared Libraries*)

What is inside that box?

- To understand what linkers do, we need to see what an executable is made out of.
  (Spoilers: it is not just code/data)

- A Linux executable is defined by the **Executable and Linkable Format** (ELF) standard.
  - Used for `*.o` files
  - And executables
  - And `*.so` (shared objects; soon!)

What the ELF ??

- Contains all of the segments and data sections defining a program.
- The ELF executable has roughly the following structure:

Offset	Name	Description
`0x00`	Magic Number	4 bytes: A `0x7F` byte followed by "`ELF`" in ASCII
`0x04`	Class	1 byte: `0x1` if 32-bit, 0x2 if 64-bit
`0x05`	Data	1 byte: `0x1` if little-endian, 0x2 if big-endian
`0x06`	Version	1 byte: `0x1` for the current version.
`0x07`	ABI	1 byte: `0x0` for System V (our C ABI)
`0x12`	Machine	2 bytes: `0x03` is x86, `0x08` is MIPS, `0xF3` is RISC-V, etc

What the ELF ??

- The remaining fields indicate where certain sections start.
- An ELF executable contains these sections:
  - Segment Headers (where .text, .data, .bss, etc, exist in the executable)
    - The initial data for each memory segment in the memory layout!
    - We will look at these again when we look at *loading*.
  - The Symbol Table
    - All of the “names” that may be referenced by other code.
- Symbols can consist of:
  - Functions
  - Global variables
  - Special sections (special compiler or OS areas)
- We will focus on function/variable symbols.

`readelf` – Viewing the symbol table

- You can investigate the symbols that are part of any object file using the `readelf -s` command on Linux/UNIX.

C (`gcc -c pirate.c`)

```c
#include <stdio.h>

void speak_like_pirate(const char* foo) {
  printf("Arr matey! %s\n", foo);
}

int main(void) {
  speak_like_pirate("Hello World!");
  return 0;
}
```

This is a symbol. It has a location.

`readelf -s pirate.o`

```
Symbol table '.symtab' contains 13 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS pirate.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    8
     8: 00000000     0 SECTION LOCAL  DEFAULT    6
     9: 00000000    39 FUNC    GLOBAL DEFAULT    1 speak_like_pirate
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    12: 00000027    23 FUNC    GLOBAL DEFAULT    1 main
```

Here it is! At 0x27 (39) bytes.

`static`; Controlling the symbols

- Remember the `static` keyword?

- This forces any symbol to be *local* to the current file. That is, it can not be referenced by an outside function.
  - This is because the symbol will not be included in the symbol table!
  - The linker will not be able to see it.

- This is useful for avoiding ***name collisions***, when two functions have the same name.
  - This normally would make using multiple files and other people's code troublesome.
  - Using `static` helps because it will not pollute the symbol table.

Controlled the symbols

- You can investigate the impact of using `static` by again using the `readelf -s` command on Linux/UNIX.

C (`gcc -c pirate-static.c`)

```c
#include <stdio.h>

static void speak_like_pirate(const char* foo) {
  printf("Arr matey! %s\n", foo);
}

int main(void) {
  speak_like_pirate("Hello World!");
  return 0;
}
```

This symbol has a location… but it can only be referenced in this file.

`readelf -s pirate-static.o`

```
Symbol table '.symtab' contains 13 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS pirate.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000    39 SECTION LOCAL  DEFAULT    1 speak_like_pirate
     7: 00000000     0 SECTION LOCAL  DEFAULT    7
     8: 00000000     0 SECTION LOCAL  DEFAULT    8
     9: 00000000     0 SECTION LOCAL  DEFAULT    6
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    12: 00000027    23 FUNC    GLOBAL DEFAULT    1 main
```

Our static function is now "LOCAL"

`extern`; when you used to be an intern

- The other side of the coin is the `extern` keyword.
- This tells the linker that it should expect the symbol to be found elsewhere.

C (pirate.c)

```c
#include <stdio.h>

extern const char* pirate_greet;
// pirate_greet is explicitly extern

void speak_like_pirate(const char* foo) {
  printf("%s %s\n", pirate_greet, foo);
}
```

This symbol is... somewhere.

C (main.c)

```c
const char* pirate_greet = "Arr matey!";

// Declare the function (implicitly extern)
void speak_like_pirate(const char*);

int main(void) {
  speak_like_pirate("Hello World!");
  return 0;
}
```

Here it is!!

Final thoughts of global variables

- You should always avoid global variables.

- However, if you are using them, make sure to liberally use `static`
  - This will stop the names of variables from polluting the symbol table.
  - The use of `extern` is likely indicating a poor design.

- This is also true for functions, too.
  - Generally declare them `static` unless you need them from within another file.
  - Helps make it clear what functions are important and which can be deleted or refactored.
  - (Much like private functions in classes)

- Always initialize your global variables!

Seeing through the linker's eyes

- Which symbols are part of each file?
- Which are local and which are global?
- Which symbols are satisfied by the other file?

The linker references "main" when it compiles the executable.

C (fibonacci.c)

A local symbol.

```
static int memoized[100] = {1, 1, 0};

A global symbol.
int fibonacci(int n) {
  if (!memoized[n]) {
    int tmp = fibonacci(n - 1);
    memoized[n] = tmp + fibonacci(n - 2);
  }
          Referenced here.

  return memoized[n];
}
```

C (main.c)

We need to tell the compiler that we are linking to a symbol.

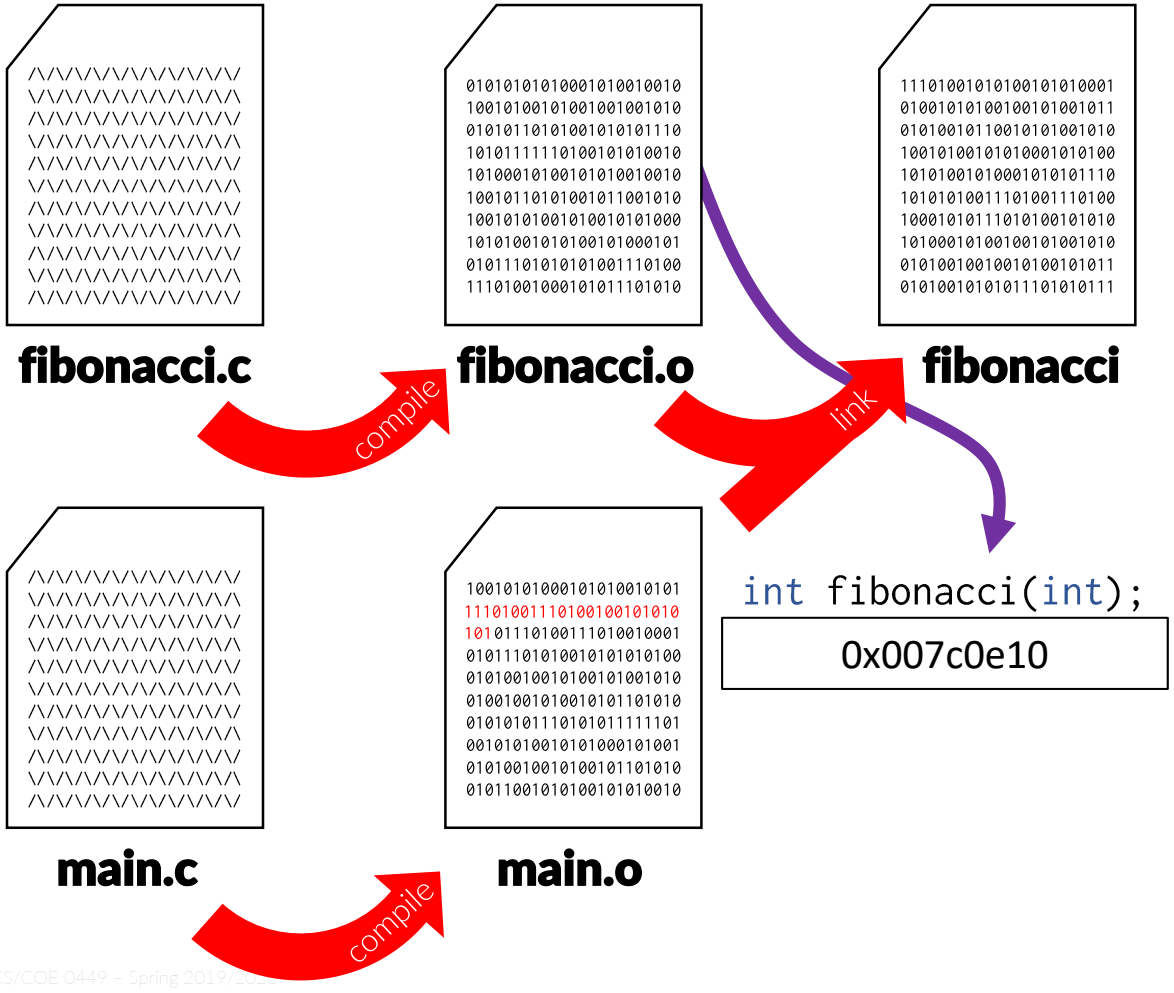
```
int fibonacci(int);

A global symbol.
int main(void) {
  int product = 1;
  for (int i = 0; i < 25; i++) {
    product *= fibonacci(i);
  }
                Referenced here.

  return product;
}
```

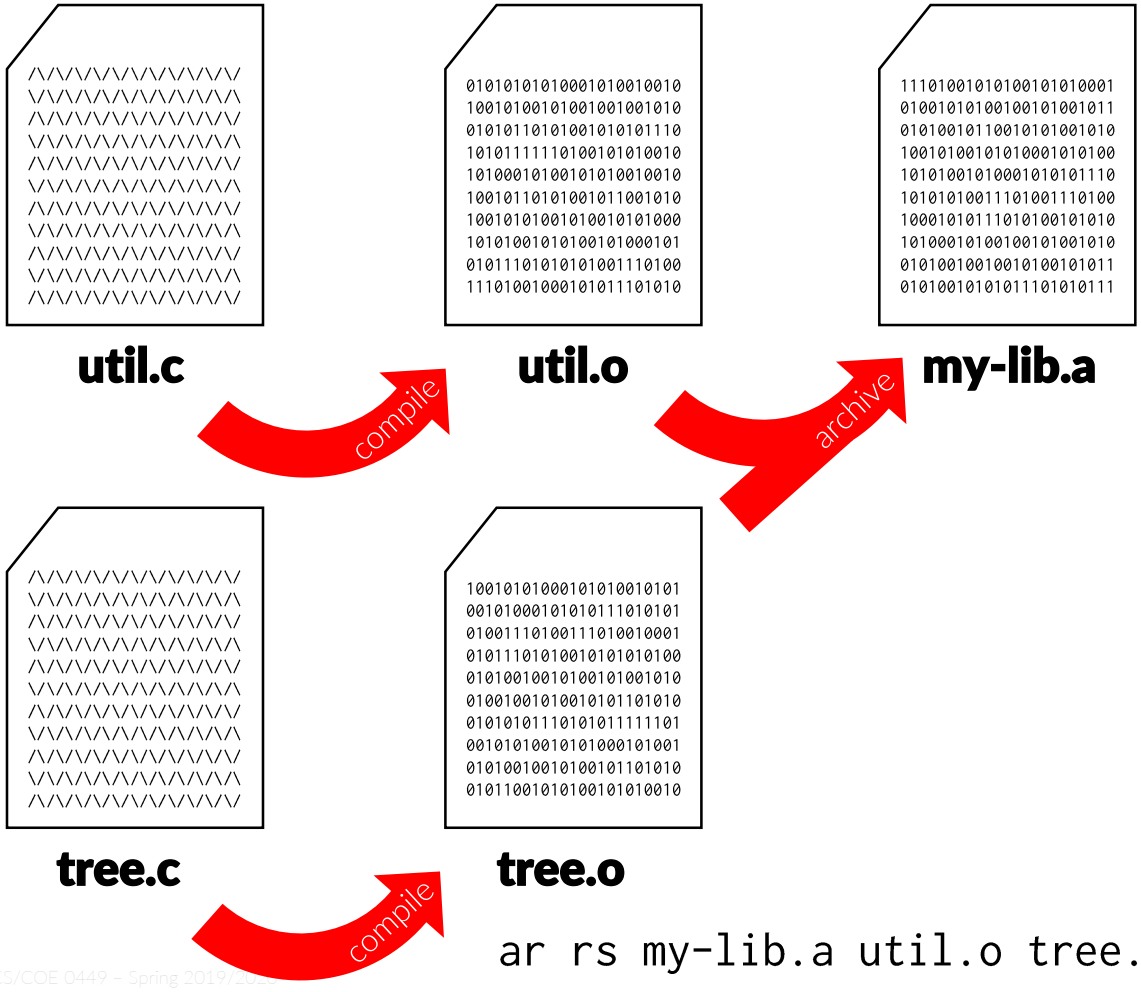
Linker doesn't see these temporary variables.

Summing it up: Playing mad-libs



- The compiler hands off object files with blanks where referenced symbols reside.
- The linker's job is to fill in those blanks with the location of the symbol in the final executable.

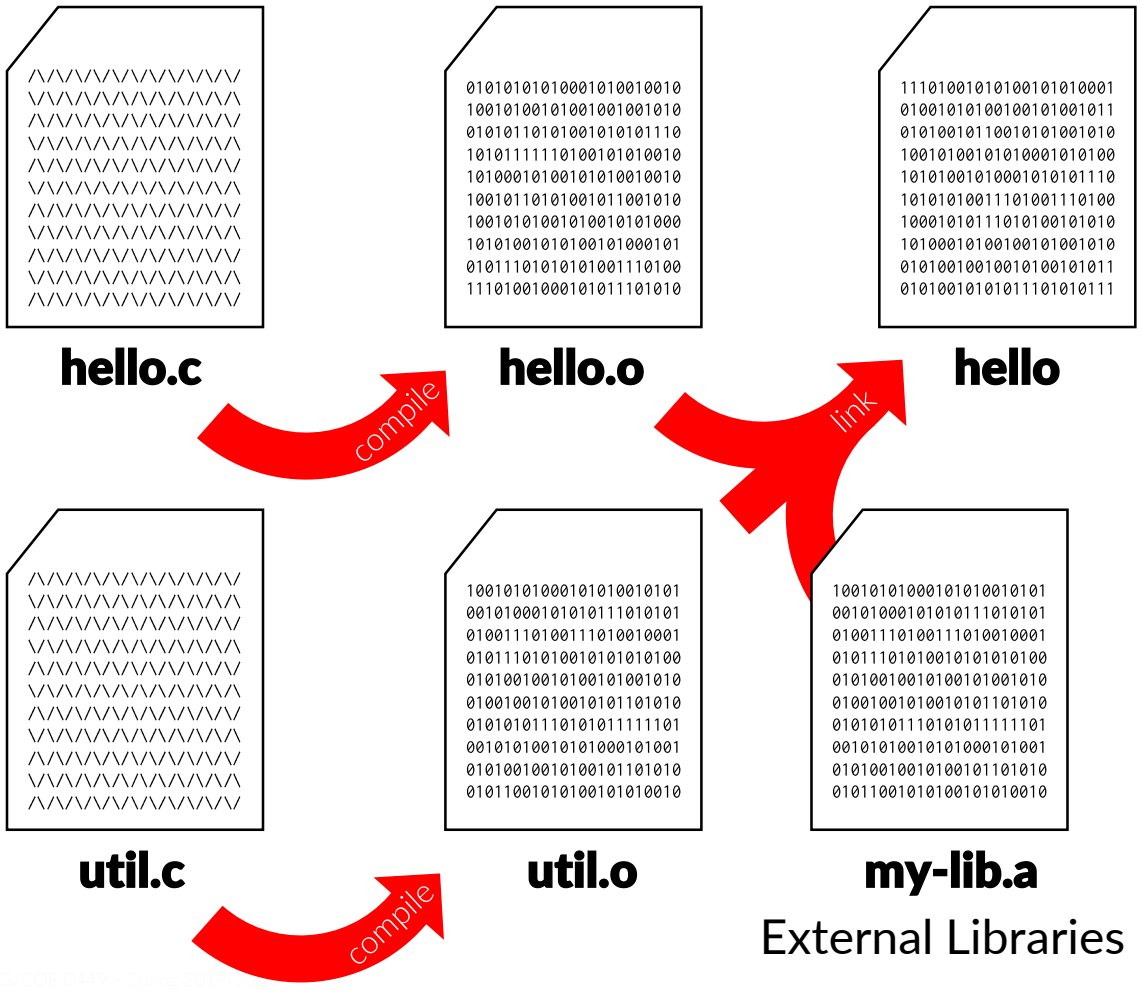
Static Libraries (`*.a` files)



- If you want to share your library with others…
- Instead of creating an executable, you can package together all of the `*.o` files into a single archive (`.a` file)
- You can use the `ar` program on Linux for this.

```
ar rs my-lib.a util.o tree.o
```

Compilation: Simple Overview – Redux



- We can use my-lib.a in place of the object files we need.
- The `*.a` file is just a container for a set of object files. Essentially, it is just a kind of zip file of object files.
- These object files get copied into our executable… not very efficient! Hmm!

Loaders

You should always stretch before you run – OSes do this, too.

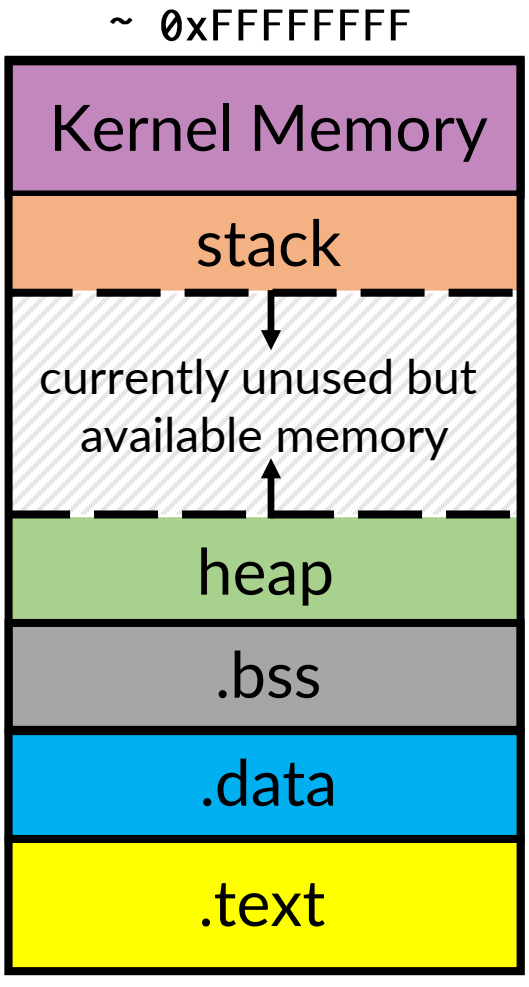
The Operating System

- How does your ELF executable actually run?
- There needs to be some system software to unpack the executable into memory.
- That system software is a **loader** and it is part of an **operating system**.

Memory Segments – Deeper dive!

- The ELF executable defines several segments:
  - `.text` – The code segment (machine code)
  - `.data` – The data segment (program data)
  - `.rodata` – The read-only data segment (constants)
  - `.bss` – Uninitialized data segment ("zero" data)

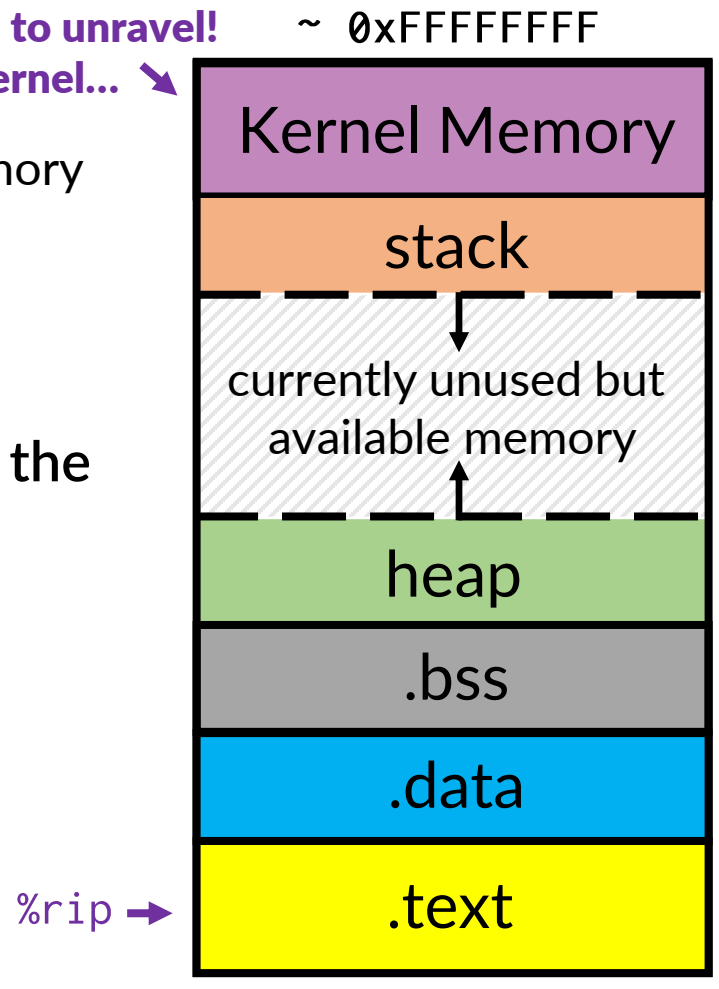
- The `.bss` segment is a special segment for all data that starts as `0` or `NULL`.
  - (Its name is Block Started by Symbol which is a historic nonsense name. Sigh!)
  - It is often an optimization: the executable does not need to store a whole bunch of zeros.
  - Hmm… the operating system must then allocate a bunch of zeros. Is that fast?? (We'll get there)



Running a program

1. Take the ELF executable.
   - This defines each segment and where in memory it should go.
2. Place the `.text` segment into memory.
3. Place the `.data` segment into memory.
4. Write the number of zeroes specified to the `.bss` segment.
5. Allocate the stack and assign the stack pointer (`%rsp`)
6. Jump to the entry point address (the location of the `_start` symbol)
   - `_start` will call main after initializing the C runtime and the heap.

Our lie starts to unravel!
We have a kernel... ↘



Some .bss BS I've dealt with…

- Forgetting to zero the .bss segment is… very interesting.
  - If you write an OS, and forget this, then you get loops that don't work write.
  - Because now variables that were equal to `0` are now random garbage.

C (main.c)

This goes into the .bss because it is zero

```c
static int count = 0;
static int things[10] = {0};

int sumThings() {
  int ret = 0;
  for (int i = 0; i < counter; i++) {
    ret += things[i];
  }
  return ret;
}
```

This does not go into the .bss because it is not a symbol.

That's it???

- Pretty much! However, let's make it more flexible.

- Our linking so far is **static linking** where all of the code goes into the executable. Duplicate code from **static libraries** is copied in.
  - Not very space efficient. Duplicates code most programs are using! (libc)
  - What if we "shared" the code external to the executable?

- For **dynamic linking** we will think about loading not just the executable, but library code as well. A **shared library**.
  - The OS loader must load the program into memory and also take on the task of loading library code.
  - It then must do the "mad-libs" replacing references in the program to point to where in memory the library code was loaded. Tricky!

Dynamic Linking

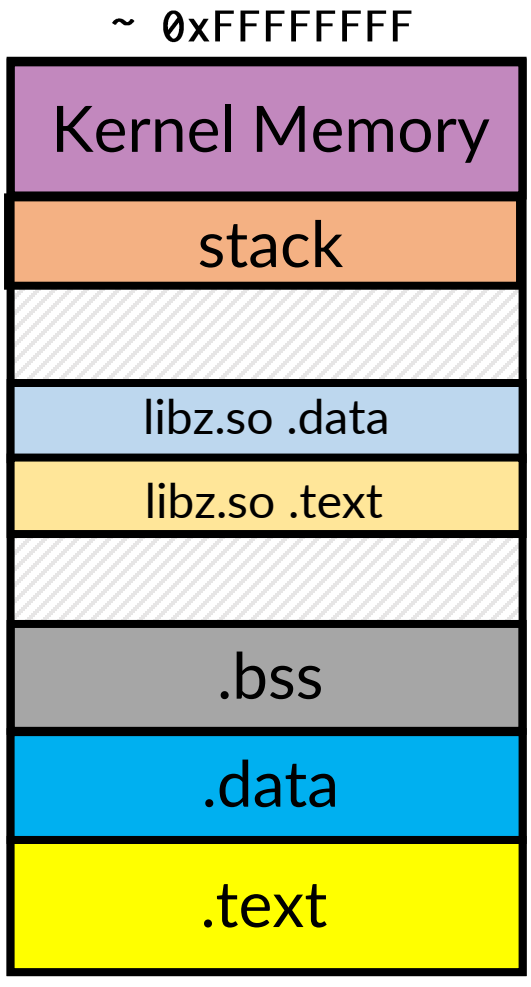
Linking but… yanno… animated.

Code that can be loaded… anywhere?

- **The main problem is this:**
  - Programs generally need to assume where in memory they live.
- **They refer to functions and data at particular addresses.**
  - The linker decides where those are, but they are then hard-coded in.

Where should this go??

- **We want to provide a single software library to multiple executables…**
  - We can't know ahead of time where that library *can* go in memory since programs are different sizes… they might need multiple libraries… etc.



Solution: relocatable code

- Let's allow code to refer to functions and/or data that may move.
- Essentially, the operating system plays the mad-lib game.
  - The ELF executable has a list of "relocatable entries"
  - The OS goes through them and fills them in according to where the external symbols are.

Linking to the libz.so dynamic library

C (`gcc -o compressor compressor.c -lz`)

```c
#include <zlib.h> // provides compressBound

int main(void) {
  long bufferSize = compressBound(10000);
  return 0;
}
```

We don't know where this function ultimately is...

Solution: relocatable code

- Let's allow code to refer to functions and/or data that may move.
- Essentially, the operating system plays the mad-lib game.
  - The ELF executable has a list of "relocatable entries"
  - The OS goes through them and fills them in according to where the external symbols are.

C (gcc -o compressor compressor.c -lz)

```c
#include <zlib.h> // provides compressBound

int main(void) {
  long bufferSize = compressBound(10000);
  return 0;
}
```

x86-64 (objdump -d compressor)

```
0000000000001139 <main>:
 1139: 55                   push   %rbp
 113a: 48 89 e5             mov    %rsp,%rbp
 113d: 48 83 ec 10          sub    $0x10,%rsp
 1141: bf 10 27 00 00       mov    $0x2710,%edi
 1146: e8 00 00 00 00       callq  ??? <???>
 114b: 48 89 45 f8          mov    %rax,-0x8(%rbp)
 114f: b8 00 00 00 00       mov    $0x0,%eax
 1154: c9                   leaveq
 1155: c3                   retq
```

Solution: relocatable code: Loading

- When the OS loads this executable… it will have a relocation entry that tells it to overwrite at byte `0x1147` the relative address of "compressBound"
- With this extra step, the OS loader is also providing dynamic linking.

C (`gcc -o compressor compressor.c -lz`)

```c
#include <zlib.h> // provides compressBound

int main(void) {
  long bufferSize = compressBound(10000);
  return 0;
}
```

0x114b + 0x5fe = 0x1749
(`callq` is relative to `%rip`)

In modern times, this makes use of a jump table called a Procedure Linkage Table (PLT).

x86-64 (`objdump -d compressor`)

```
0000000000001139 <main>:
 1139: 55                   push   %rbp
 113a: 48 89 e5             mov    %rsp,%rbp
 113d: 48 83 ec 10          sub    $0x10,%rsp
 1141: bf 10 27 00 00       mov    $0x2710,%edi
 1146: e8 fe 05 00 00       callq  1749 <compressBound>
 114b: 48 89 45 f8          mov    %rax,-0x8(%rbp)
 114f: b8 00 00 00 00       mov    $0x0,%eax
 1154: c9                   leaveq
 1155: c3                   retq
```

Taking a PIC, eating some PIE – Avoiding relocations

- In order to allow code to be resident anywhere in memory, the compiler must emit machine code that always uses relative addresses!

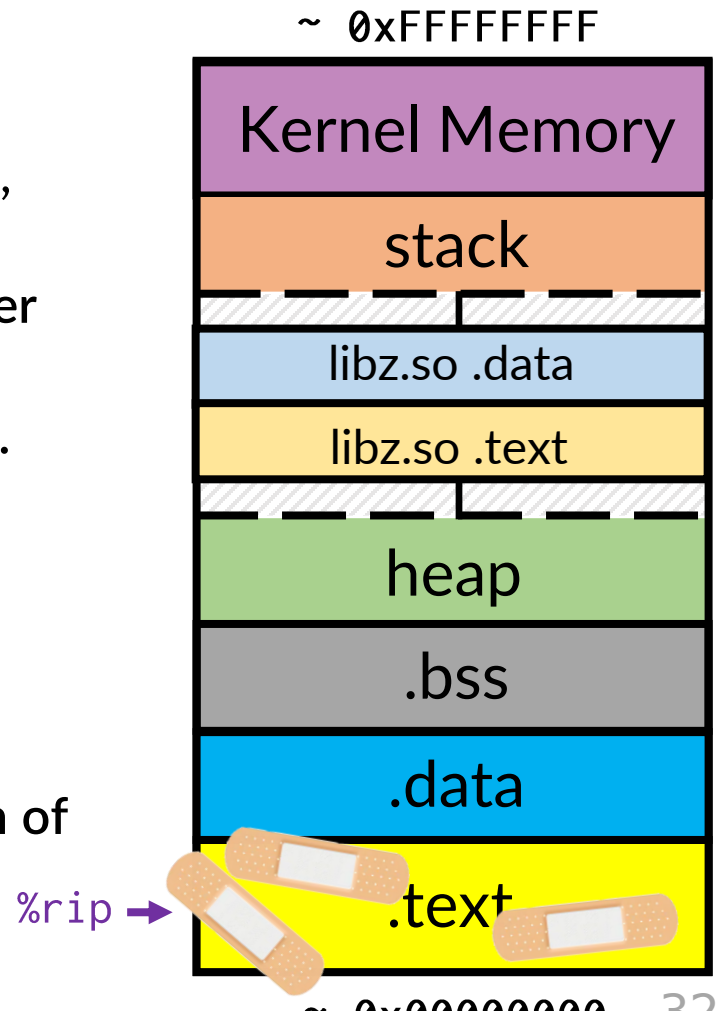
- This is called **position independent code** (or PIC).
- When your entire executable is made out of PIC, it is a **position independent executable** (or PIE)

- `gcc` will compile code this way when you specify the `-fPIC` flag.
  - You generally need this when creating dynamic libraries.

Who doesn't like pie???

Running a program - Redux

1. Take the ELF executable.
2. Place and initially prepare the `.text`, `.data`, `.bss` segments into memory.
5. Allocate the stack and assign the stack pointer (`%rsp`)
6. Repeatably load each required shared library.
   6a. Place .text and .data in memory
   6b. Rewrite .text sections by looking at the relocatable entries
   6c. Repeat for each library.
7. Jump to the entry point address (the location of the `_start` symbol)
   - `_start` will call main after initializing the C runtime and the heap.



Being lazy – Run-time loading

- Having the OS load every library at the start can delay the execution of a program.
  - What if your program rarely uses a library?
  - What if you want to expand the program while it is running?
    - Plugins are a good example.
- We can make use of an OS service to dynamically load libraries.
  - On Linux we have the `dlopen` and `dlsym` system functions.
  - Look at the documentation online and refer to examples.

C (`gcc -o compressor compressor.c -ldl`)

```c
#include <zlib.h>  // provides compressBound
#include <dlfcn.h> // provides dlopen, etc
#include <stdio.h> // provides fprintf and stderr

// Declares a pointer to a function that
// returns an int and takes an int as an argument
static int (*compressBound)(int);

int main(void) {char* error = NULL;
  void* handle = dlopen("libz.so", RTLD_LAZY);
  if (!handle) {
    fprintf(stderr, "%s\n", dlerror());
    return 1;
  }

  lazyCompressBound = dlsym(handle, "compressBound");
  error = dlerror();
  if (error) {
    fprintf(stderr, "%s\n", error);
    return 1;
  }

  long bufferSize = lazyCompressBound(10000);
  return 0;
}
```

Function pointers are very messy.

Prints to the screen's "error" buffer.

Uses the lazy-loaded function.

Investigating dynamic libraries

- **If you would like to see what dynamic libraries a program uses, you can use `readelf` or the `ldd` command.**
  - Cannot see the `dlopen / dlsym` lazy loaded libraries.

- `ldd ./my-program`

- `readelf -d ./my-program`

Linking, loading; static and dynamic… Whew!

- Linking is when we merge multiple pieces of executable code into one logical program.

- We link at various times:

  - At **compile-time**: using our normal `*.o` files and static libraries (`*.a`)

  - At **load-time**: our OS reads and loads the executable and loads dynamic libraries (`*.so`) at the same time, rewriting relocatable sections.

  - At **run-time**: our program uses system services (`dlopen`) to load dynamic libraries lazily.