CS 241: Systems Programming
Lecture 29. Static Libraries

Fall 2025

Prof. Stephen Checkoway

What are some reasons we use libraries (crates)?

A. Select any option on your clicker

Multiple forms of code reuse

Source code reuse

- Distribute source code that can be included in many programs

Binary code reuse

- Distribute binary code that can be linked into programs
- Static libraries: code linked in at compile time (actually link time)
- Dynamic libraries: code linked in at runtime

Code compilation model

Source code goes in, object file comes out

In C:

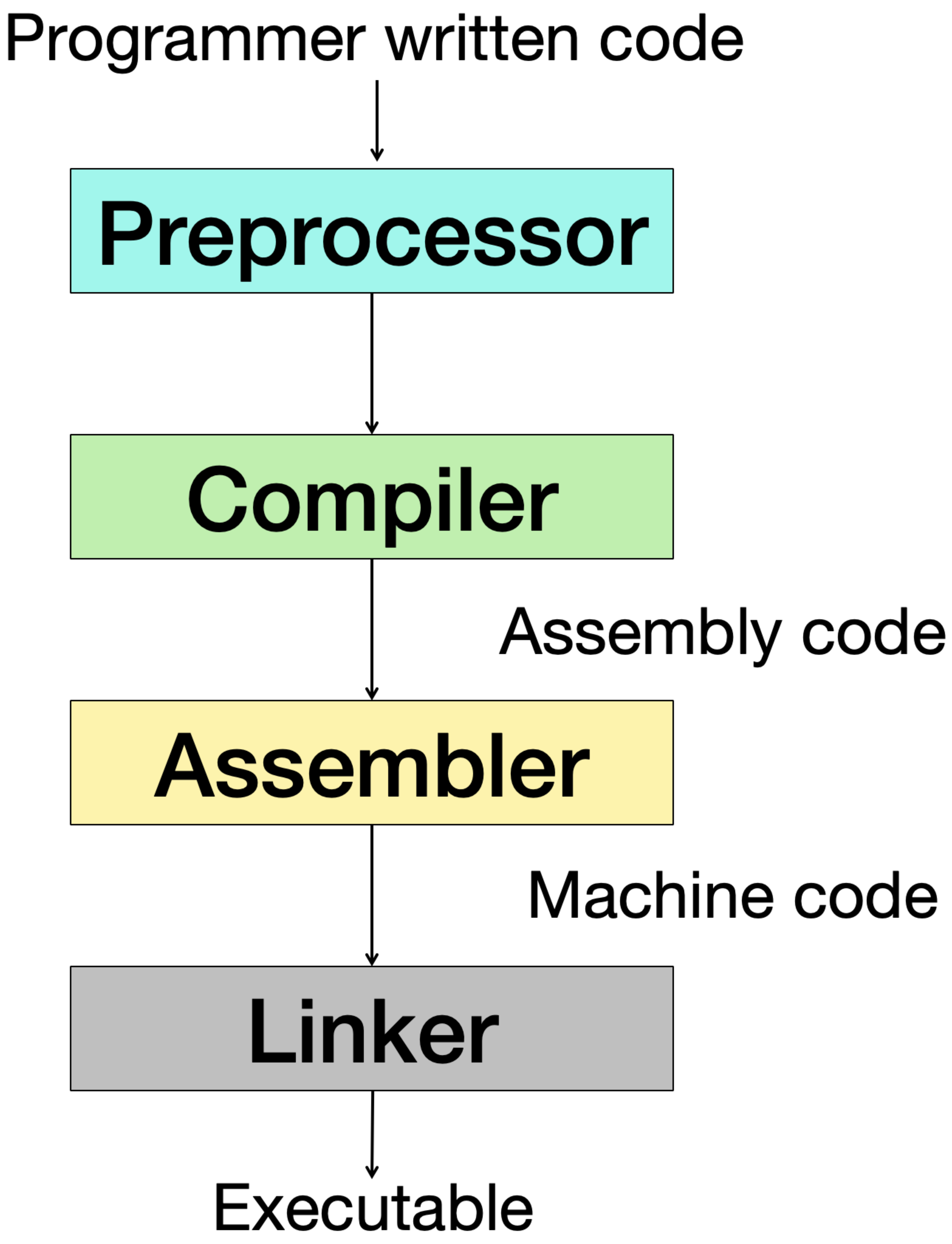
- foo.c -> foo.o

In Rust:

- lib.rs -> library_name-hash.o
- main.rs -> bin_name-hash.o
- bin/foo.rs -> foo-hash.o

Linking step combines object files and libraries into a final executable (or library)

Creating an Executable



Programmer written code

Preprocessor

Compiler

Assembly code

Assembler

Machine code

Linker

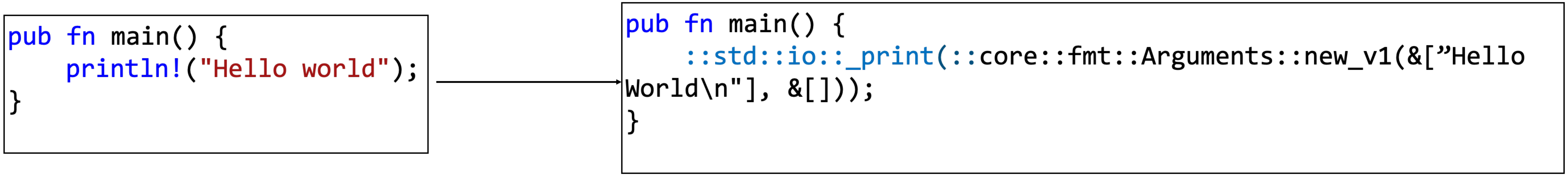
Executable

Preprocessor: expands macros

All macros in Rust end in ! - what are some examples of macros we've used?

A. Select any option on your clicker

Preprocessor



```
pub fn main() {
    println!("Hello world");
}
```

```
pub fn main() {
    ::std::io::_print(::core::fmt::Arguments::new_v1(&[”Hello
World\n"], &[]));
}
```

Compiler

Converts high-level language to assembly language

Compiler

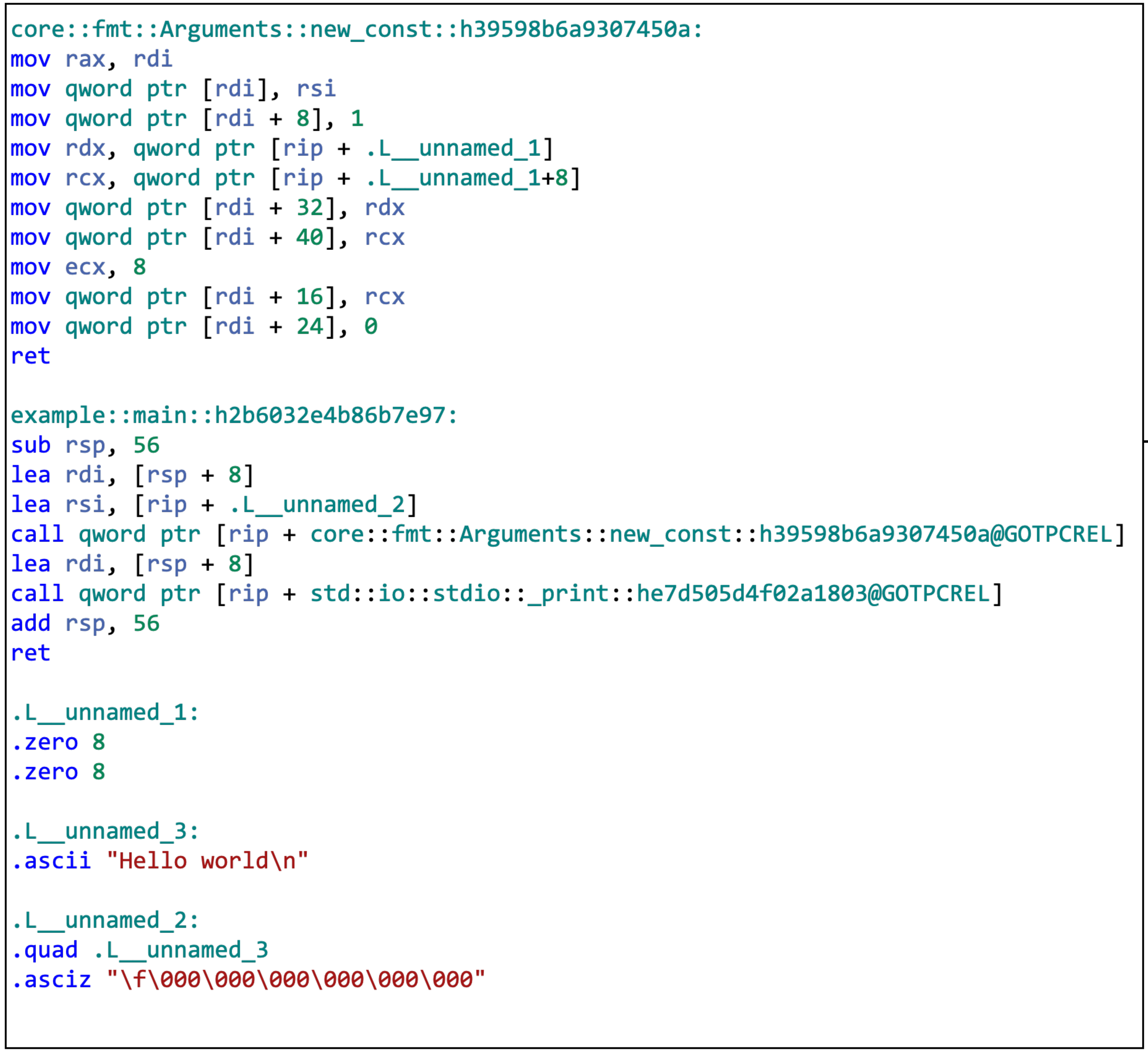
```
pub fn main() {
    ::std::io::_print(::core::fmt
::Arguments::new_v1(&[”Hello
World\n"], &[]));
}
```

```
core::fmt::Arguments::new_const::h39598b6a9307450a:
mov rax, rdi
mov qword ptr [rdi], rsi
mov qword ptr [rdi + 8], 1
mov rdx, qword ptr [rip + .L__unnamed_1]
mov rcx, qword ptr [rip + .L__unnamed_1+8]
mov qword ptr [rdi + 32], rdx
mov qword ptr [rdi + 40], rcx
mov ecx, 8
mov qword ptr [rdi + 16], rcx
mov qword ptr [rdi + 24], 0
ret

example::main::h2b6032e4b86b7e97:
sub rsp, 56
lea rdi, [rsp + 8]
lea rsi, [rip + .L__unnamed_2]
call qword ptr [rip + core::fmt::Arguments::new_const::h39598b6a9307450a@GOTPCREL]
lea rdi, [rsp + 8]
call qword ptr [rip + std::io::stdio::_print::he7d505d4f02a1803@GOTPCREL]
add rsp, 56
ret

.L__unnamed_1:
.zero 8
.zero 8

.L__unnamed_3:
.ascii "Hello world\n"

.L__unnamed_2:
.quad .L__unnamed_3
.asciz "\f\000\000\000\000\000\000"
```

Assembler

Converts assembly language to machine language

Assembler



```
core::fmt::Arguments::new_const::h39598b6a9307450a:
mov rax, rdi
mov qword ptr [rdi], rsi
mov qword ptr [rdi + 8], 1
mov rdx, qword ptr [rip + .L__unnamed_1]
mov rcx, qword ptr [rip + .L__unnamed_1+8]
mov qword ptr [rdi + 32], rdx
mov qword ptr [rdi + 40], rcx
mov ecx, 8
mov qword ptr [rdi + 16], rcx
mov qword ptr [rdi + 24], 0
ret

example::main::h2b6032e4b86b7e97:
sub rsp, 56
lea rdi, [rsp + 8]
lea rsi, [rip + .L__unnamed_2]
call qword ptr [rip + core::fmt::Arguments::new_const::h39598b6a9307450a@GOTPCREL]
lea rdi, [rsp + 8]
call qword ptr [rip + std::io::stdio::_print::he7d505d4f02a1803@GOTPCREL]
add rsp, 56
ret

.L__unnamed_1:
.zero 8
.zero 8

.L__unnamed_3:
.ascii "Hello world\n"

.L__unnamed_2:
.quad .L__unnamed_3
.asciz "\f\000\000\000\000\000\000"
```

object file (.o)

```
0000000 facf feed 000c 0100 0000 0000 0002 0000
0000010 0012 0000 0738 0000 0085 00a0 0000 0000
0000020 0019 0000 0048 0000 5f5f 4150 4547 455a
0000030 4f52 0000 0000 0000 0000 0000 0000 0000
0000040 0000 0000 0001 0000 0000 0000 0000 0000
0000050 0000 0000 0000 0000 0000 0000 0000 0000
0000060 0000 0000 0000 0000 0019 0000 0228 0000
0000070 5f5f 4554 5458 0000 0000 0000 0000 0000
0000080 0000 0000 0001 0000 0000 0004 0000 0000
0000090 0000 0000 0000 0000 0000 0004 0000 0000
00000a0 0005 0000 0005 0000 0006 0000 0000 0000
00000b0 5f5f 6574 7478 0000 0000 0000 0000 0000
00000c0 5f5f 4554 5458 0000 0000 0000 0000 0000
00000d0 089c 0000 0001 0000 290c 0003 0000 0000
00000e0 089c 0000 0002 0000 0000 0000 0000 0000
00000f0 0400 8000 0000 0000 0000 0000 0000 0000
0000100 5f5f 7473 6275 0073 0000 0000 0000 0000
0000110 5f5f 4554 5458 0000 0000 0000 0000 0000
0000120 31a8 0003 0001 0000 0300 0000 0000 0000
0000130 31a8 0003 0002 0000 0000 0000 0000 0000
0000140 0408 8000 0000 0000 000c 0000 0000 0000
0000150 5f5f 6367 5f63 7865 6563 7470 745f 6261
0000160 5f5f 4554 5458 0000 0000 0000 0000 0000
0000170 34a8 0003 0001 0000 11d8 0000 0000 0000
0000180 34a8 0003 0002 0000 0000 0000 0000 0000
0000190 0000 0000 0000 0000 0000 0000 0000 0000
00001a0 5f5f 6f63 736e 0074 0000 0000 0000 0000
00001b0 5f5f 4554 5458 0000 0000 0000 0000 0000
```

Static libraries ("archives")

Nothing more than a collection of object files (.o) bundled together

A "foo" library composed of object files a.o, b.o, …, z.o

- Traditionally named `libfoo.a`
- Compile object files as normal, e.g.,

  ```
  $ rustc lib.rs --emit=obj
  ```
- Put them in an archive:

  ```
  $ ar crs libfoo.a a.o b.o … z.o
  ```

Rust static libraries

Rust libraries are distributed as source code

Compiling a Rust project causes each library to be built as a static library

- libc -> liblibc-73ce9a2ad47cacba.rlib

Rust's .rlibs are just standard archive files (although this is an implementation detail)

Linker

Combines object files into a single executable (or dynamic library)

Updates addresses of symbols now that files are combined

Symbols

Anything a module has a name for:

- Function
- Global variable
- Static variable

What are the symbols in this code?

```
const B: i32 = 10;

fn max(a: i32) -> i32 {
    if a > B {
        return a;
    }
    B
}

fn main() {
    let x: i32 = 11;
    let y = max(x);
    println!("{y}");
}
```

A. main, max

B. main, max, println (really std::io::_print)

C. B, main, max, println (really std::io::_print)

D. a, x, y, B, main, max, println (really std::io::_print)

Symbols

Symbols have

- a name — the identifier used in the program; and
- a value — an offset into a section (.text, .data, .bss, etc.)

Symbols

Symbols have

- a name — the identifier used in the program; and
- a value — an offset into a section (.text, .data, .bss, etc.)

```
$ readelf -s maze.o

Symbol table '.symtab' contains 59 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    45: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND free
    46: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND malloc
    47: 00000000000005e0   135 FUNC    GLOBAL DEFAULT    2 maze_free
    48: 0000000000000700   143 FUNC    GLOBAL DEFAULT    2 maze_get_cols
```

Symbols

Symbols have

- a name — the identifier used in the program; and
- a value — an offset into a section (.text, .data, .bss, etc.)

UND is undefined
2 is .text (in this case)

```
$ readelf -s maze.o

Symbol table '.symtab' contains 59 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    45: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND free
    46: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND malloc
    47: 00000000000005e0   135 FUNC    GLOBAL DEFAULT    2 maze_free
    48: 0000000000000700   143 FUNC    GLOBAL DEFAULT    2 maze_get_cols
```

What does a linker do?

Symbol resolution

Relocation

Before we combine a bunch of files that reference the same variables/functions, we need exactly one definition for each variable/function, and every reference needs to point to that definition

Linking with static libraries

Linking with static libraries

The linker only includes object files from an archive which are "needed"

For example,

Linking with static libraries

The linker only includes object files from an archive which are "needed"

For example,

- `a.rs` defines `fn fun1();`

Linking with static libraries

The linker only includes object files from an archive which are "needed"

For example,

- `a.rs` defines `fn fun1();`
- `b.rs` defines `fn fun2();`

Linking with static libraries

The linker only includes object files from an archive which are "needed"

For example,

- `a.rs` defines `fn fun1();`
- `b.rs` defines `fn fun2();`
- `c.rs` defines `i32 blah;`

Linking with static libraries

The linker only includes object files from an archive which are "needed"

For example,

- `a.rs` defines `fn fun1();`
- `b.rs` defines `fn fun2();`
- `c.rs` defines `i32 blah;`
- `libfoo.rlib` contains `a.o`, `b.o`, and `c.o`

Linking with static libraries

The linker only includes object files from an archive which are "needed"

For example,

- `a.rs` defines `fn fun1();`
- `b.rs` defines `fn fun2();`
- `c.rs` defines `i32 blah;`
- `libfoo.rlib` contains `a.o`, `b.o`, and `c.o`
- If the program uses `fun1()` and `blah` but not `fun2()` in its `main.rs` then the linker will only include `a.o` and `c.o` in the final program

Defined/undefined symbols

Defined symbols have a value relative to a section in the object file (or binary)

Undefined symbols are references to symbols defined in other object files (or dynamic libraries)

Linking with static libraries

Linking with static libraries

The linker maintains a list of currently undefined symbols, initially empty

Linking with static libraries

The linker maintains a list of currently undefined symbols, initially empty

For each input files (objects and archives) from left-to-right

- If it's an object file, add the contents and symbols to the program
    - Remove defined symbols from the undefined symbol list
    - Add new undefined symbols to the undefined symbol list
- If it's an archive, perform the following until no new object files are added
    - If any object file in the archive defines a symbol in the undefined symbol list, add the object file from the archive as above

Linkers add object files from archives that define currently undefined symbols in a loop.

	a.o	b.o	foo.o	bar.o
Defined symbols	fun1	fun2 bar	main foo	bar
Undefined symbols	malloc free bar		bar fun1	

`libex.a` contains `a.o` and `b.o`.
`prog` is linked as

```
$ clang -o prog foo.o bar.o libex.a
```

Which object files are linked into `prog`?

A. `foo.o`, `bar.o`, `a.o`, and `b.o`

B. `foo.o`, `bar.o`, and `a.o`

C. `foo.o`, `bar.o`, and `b.o`

D. `foo.o`, `a.o`, and `b.o`

E. `foo.o`, and `bar.o`

Duplicate symbols are an error.

`libex.a` contains `a.o` and `b.o`.
`libbar.a` contains `bar.o`.
`prog` is linked as

```
$ clang -o prog foo.o libex.a \
  libbar.a
```

	a.o	b.o	foo.o	bar.o
Defined symbols	fun1	fun2 bar	main foo	bar
Undefined symbols	malloc free bar		bar fun1	

Which object files are linked into `prog`?

A. `foo.o`, `bar.o`, `a.o`, and `b.o`

B. `foo.o`, `bar.o`, and `a.o`

C. `foo.o`, `bar.o`, and `b.o`

D. `foo.o`, `a.o`, and `b.o`

E. Duplicate symbol error

Duplicate symbols are an error.

`libex.a` contains `a.o` and `b.o`.
`libbar.a` contains `bar.o`.
`prog` is linked as

```
$ clang -o prog foo.o libex.a bar.o
```

	a.o	b.o	foo.o	bar.o
Defined symbols	fun1	fun2 bar	main foo	bar
Undefined symbols	malloc free bar		bar fun1	

Which object files are linked into `prog`?

A. `foo.o`, `bar.o`, `a.o`, and `b.o`

B. `foo.o`, `bar.o`, and `a.o`

C. `foo.o`, `bar.o`, and `b.o`

D. `foo.o`, `a.o`, and `b.o`

E. Duplicate symbol error

Moral of the story

Specify your static libraries at the end of the link line

Dynamic libraries

Dynamic libraries are produced by the (program) linker and are combined at run time by the loader (dynamic linker)

We'll talk more about them next time!