Executables and Linking

CS449 Spring 2016

Remember External Linkage Scope?

```
#include <stdio.h>
int global = 0;
void foo();
int main()
{
  foo();
  printf("global=%d\n", global);
  return 0;
}
```

<main.c>

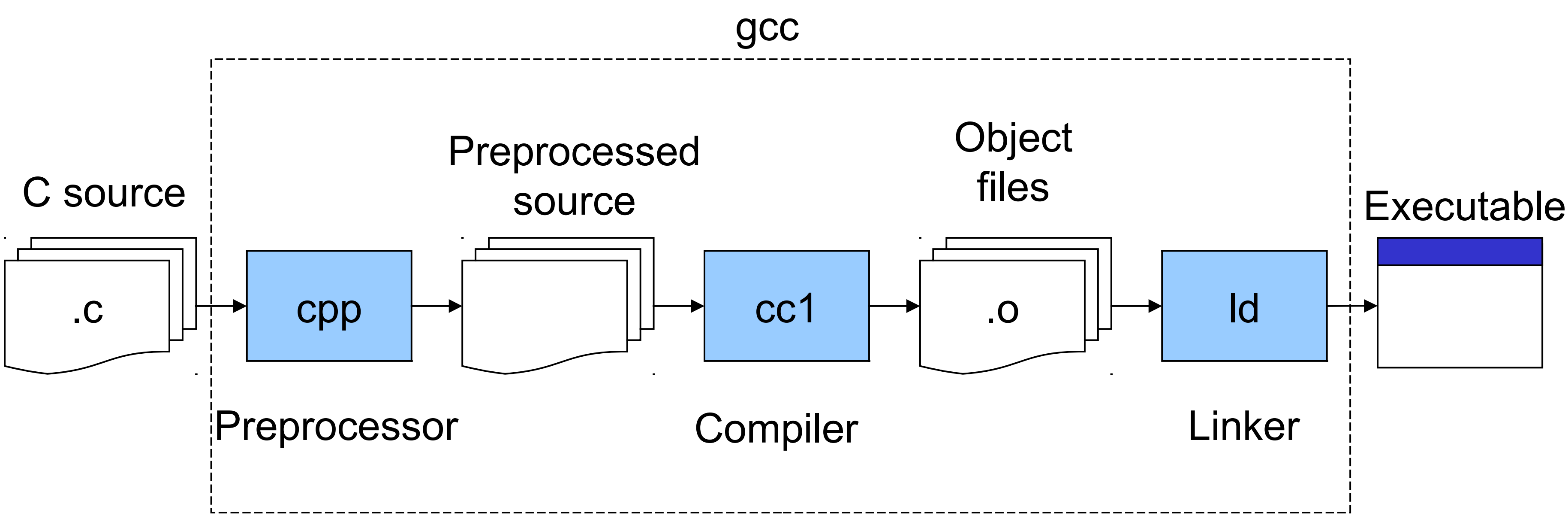
```
extern int global;
void foo() { global += 10; }
```

<foo.c>

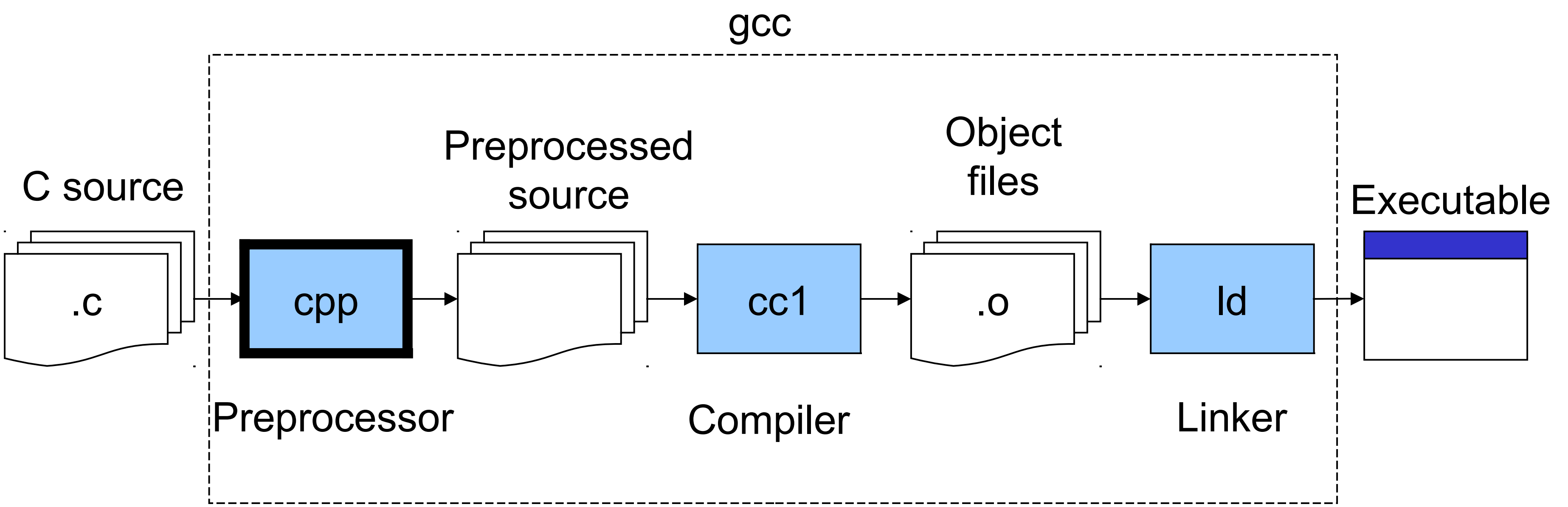
```
>> gcc ./main.c ./foo.c
>> ./a.out
global=10
```

- How did global in foo.c find global in main.c?
- How did foo() in main.c find foo() in foo.c?
- Where is the code for printf() and how does the call to printf() find it?
- What does a.out look like in the file system? How does it look like while executing in memory?

Compiler


gcc
C source
.c
cpp
Preprocessed
source
cc1
Object
files
.o
ld
Executable
Preprocessor
Compiler
Linker

Preprocessor



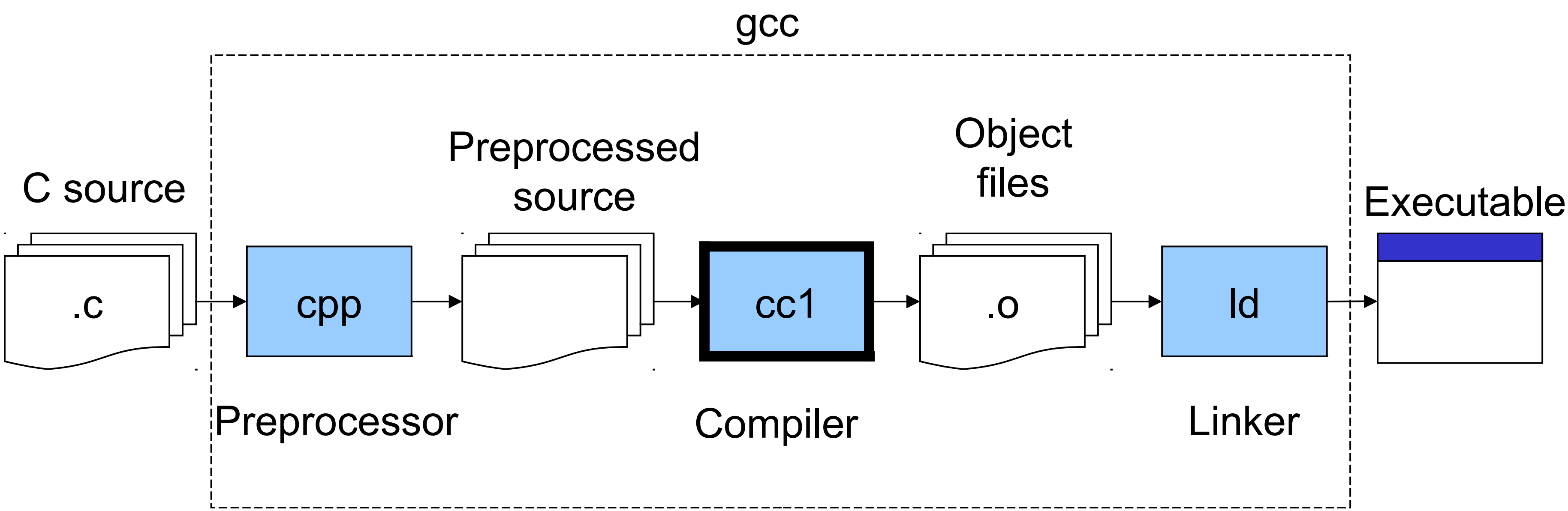
Preprocessor

- Input: A C source file with directives
- Output: A C source file after directives have been processed and comments stripped off.
- #include directive
  - `#include <...>` or `#include “...”`
  - Copies the contents of the file to source file
  - Contain function/variable declarations and type definitions
  - <...> searches for file under standard include path (usually /usr/include)
  - “....” searches for file under local directory or paths given as `–I` arguments (e.g. `gcc –I ~/local/include main.c`)

Preprocessor

- #define directive
  - Defines a macro (a symbol that is expanded to some other text)
  - `#define PI 3.14`
    - All instances of `PI` gets replaced by `3.14` in source
  - `#define AVG(a, b) (((a) + (b)) / 2)`
    - `AVG(2, 4)` in source gets replaced by `((2 + 4) / 2)`
  - No type checking
    - CPP is just a text translator; no concept of types
    - Will even work on `AVG(“Hello”, “World”)`

Compiler

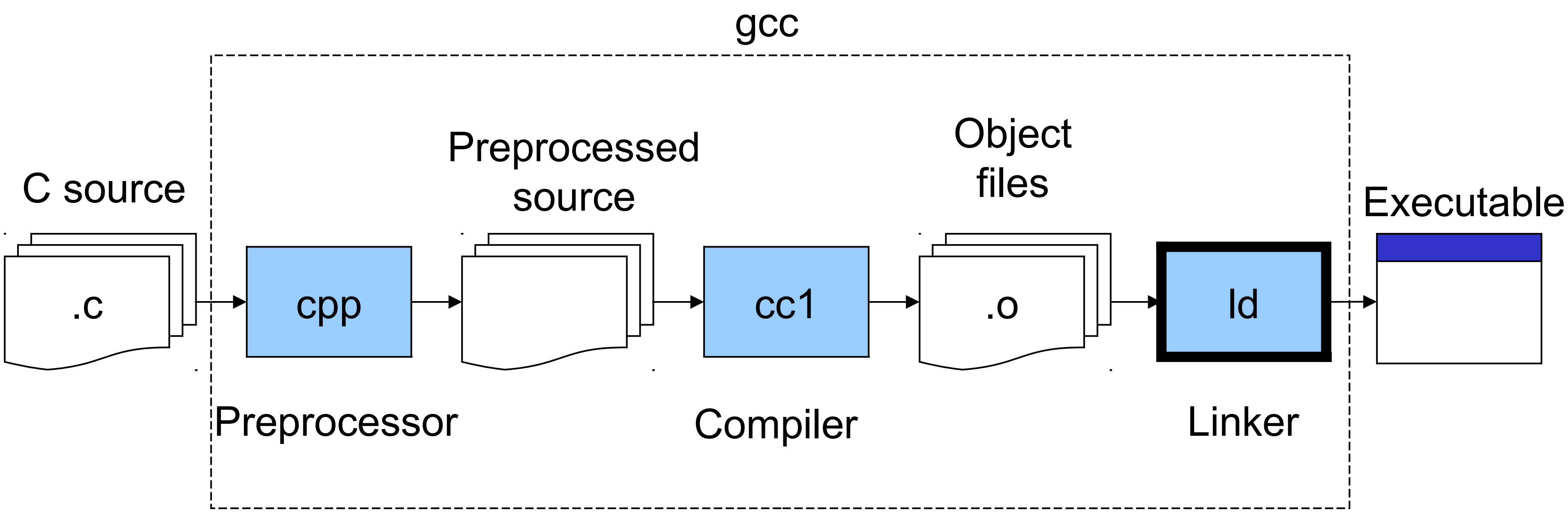

gcc
C source
.c
cpp
Preprocessor
Preprocessed
source
cc1
Compiler
Object
files
.o
ld
Linker
Executable

Compiler

- Input: A preprocessed C source file
- Output: An object file in machine language
- In the process it performs:
  - Syntax and type checking
  - Compiler optimizations to reduce execution time and memory footprint
    - E.g. `gcc –O2 main.c` (applies "level 2" optimizations)
  - Code generation for given machine
- One object file for one C source file
- The `–c` option produces an object file instead of an executable
- Regular `gcc main.c` command translates to the following:

`gcc –c main.c` (produces `main.o`)
`gcc main.o` (produces `a.out`)
`rm main.o`

Linker


gcc
C source
.c
cpp
Preprocessor
Preprocessed
source
cc1
Compiler
Object
files
.o
ld
Linker
Executable

Why Linkers?

- Linkers allow the combination of multiple files into executables
  - object files and libraries (libraries are archives of object files)
- Modularity
  - Large program can be written as a collection of smaller files, rather than one monolithic mass
  - Can build libraries of common functions
    - e.g., Math library, C standard library
- Efficiency
  - Time:
    - Change one source file, compile, and then re-link
    - No need to recompile other source files
  - Space:
    - Libraries of common functions can be put in a single file and linked to multiple programs

What does a Linker do?

- Step 1: Symbol resolution
  - Programs define and reference symbols (variables and functions)
    ```
    void foo() {…} /* define symbol foo */
    foo(); /* reference symbol foo */
    int *p = &x; /* define p, reference x */
    ```
  - Symbol definitions are stored (by compilers) in a symbol table
    - Symbol table is an array of structs
    - Each entry includes name, type, size, and location of symbol
  - Linker associates each symbol reference with exactly one symbol definition
    - Local symbols or static global symbols resolved within single object file
    - Only symbols with external linkage need to be in symbol table

What does a Linker do?

- Step 2: Relocation
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable
  - Updates all references to these symbols to reflect their new positions using relocation tables
- Think of linkers as binary rewriters
  - Uses symbol tables and relocation tables to rewrite addresses while mapping object files to memory space
  - Treats the rest of the object file as a black box

Executables

- Includes everything needed to run on a system
  - Code
  - Data
  - Dependencies
  - Directions for laying out program in memory

- Self contained files that adhere to a standard format
  - Generating executables is the job of the linker

Older Executable Formats

- a.out (Assembler OUTput)
  - Oldest UNIX format
  - No longer commonly used
- COFF (Common Object File Format)
  - Older UNIX Format
  - No longer commonly used

Modern Executable Formats

- PE (Portable Executable)
  - Based on COFF
  - Used in 32- and 64-bit Windows
- ELF (Executable and Linkable Format)
  - Linux/UNIX
- Mach-O file
  - Mac

Header

- Every **a.out** formatted binary file begins with an exec structure:

```
struct exec {
   unsigned long   a_midmag; //magic number
   unsigned long   a_text; // size of text segment
   unsigned long   a_data; // size of initialized data
   unsigned long   a_bss;  // size of uninitialized data
   unsigned long   a_syms; // size of symbol table
   unsigned long   a_entry; // entry point of program
   unsigned long   a_trsize; // size of text relocation
   unsigned long   a_drsize; // size of data relocation
};
```

Contents of an Executable

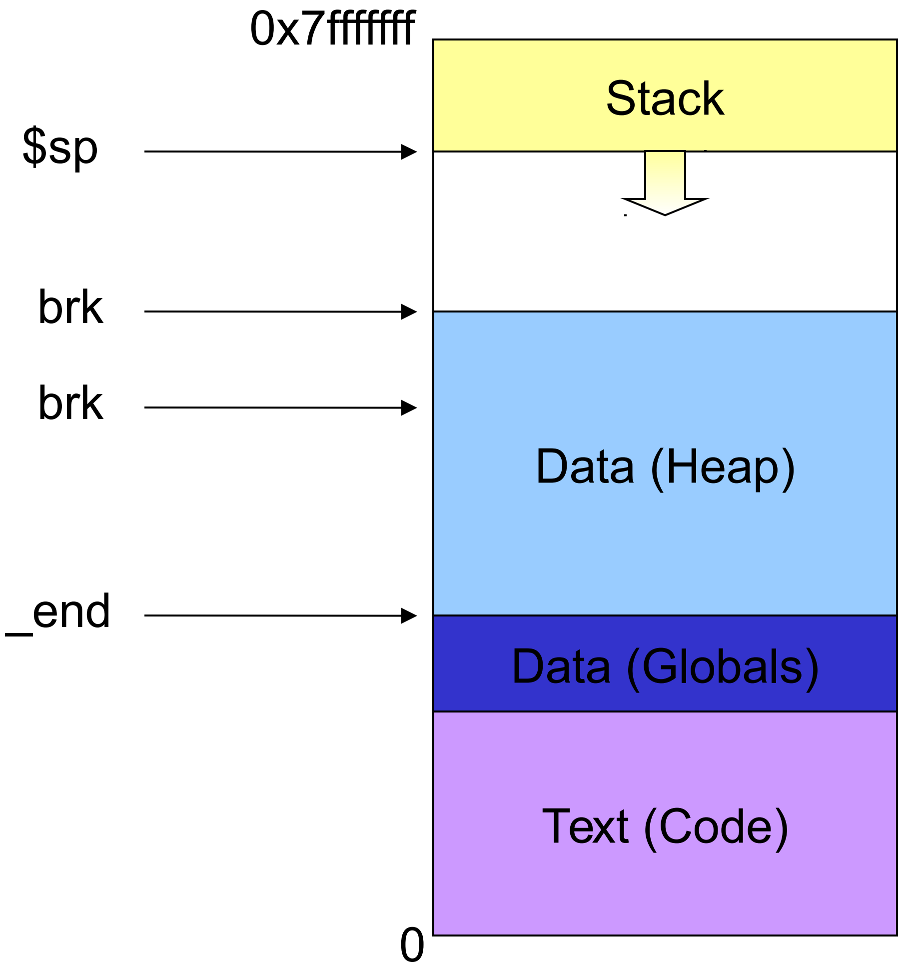
- exec header
- text segment
- data segment
- text relocations
- data relocations
- symbol table
- string table

Header

- Every **a.out** formatted binary file begins with an exec structure:

```
struct exec {
  unsigned long   a_midmag; //magic number
  unsigned long   a_text; // size of text segment
  unsigned long   a_data; // size of initialized data
  unsigned long   a_bss;  // size of uninitialized data
  unsigned long   a_syms; // size of symbol table
  unsigned long   a_entry; // entry point of program
  unsigned long   a_trsize; // size of text relocation
  unsigned long   a_drsize; // size of data relocation
};
```

Process's Address Space



Symbol Tables

```
#include <stdio.h>
int global = 0;
void foo();
int main()
{
  foo();
  printf("global=%d\n", global);
  return 0;
}
```

<main.c>

```
extern int global;
void foo() { global += 10; }
```

<foo.c>

```
>> gcc –c ./main.c ./foo.c
>> nm ./main.o
         U foo
00000000 B global
00000000 T main
         U printf
>> nm ./foo.o
00000000 T foo
         U global
```

- **nm** is a utility that shows the symbol table of object files and executables
- offset + symbol type + symbol name
- Symbol types: undefined (**U**), text (**T**), BSS (**B**).
- Symbols *foo* and *printf* are undefined (**U**) in main.o
- Symbol *global* is undefined (**U**) in foo.o

Symbol Tables

```
#include <stdio.h>
int global = 0;
void foo();
int main()
{
  foo();
  printf("global=%d\n", global);
  return 0;
}
```

<main.c>

```
extern int global;
void foo() { global += 10; }
```

<foo.c>

```
>> gcc –c ./main.c ./foo.c
>> nm ./a.out
[sic]
080483f0 T foo
08049684 B global
         U printf@@GLIBC_2.0
```

- Symbols *foo*, *global* defined and point to appropriate places in memory space
- Symbol *printf* still undefined
  - Will be defined at load time

Relocation Tables

```
#include <stdio.h>
int global = 0;
void foo();
int main()
{
  foo();
  printf("global=%d\n", global);
  return 0;
}
```

<main.c>

```
extern int global;
void foo() { global += 10; }
```

<foo.c>

```
>> gcc –c ./main.c ./foo.c
>> objdump –r ./main.o
[sic]
OFFSET   TYPE            VALUE
0000000a R_386_PC32      foo
00000010 R_386_32        global
00000015 R_386_32        .rodata
00000021 R_386_PC32      printf
```

- Offsets store location of reference to symbol
- For example, in main.o call to foo() will be rewritten by the linker at the relocation phase.

Relocation Tables

```
#include <stdio.h>
int global = 0;
void foo();
int main()
{
  foo();
  printf("global=%d\n", global);
  return 0;
}
```

<main.c>

```
extern int global;
void foo() { global += 10; }
```

<foo.c>

```
>> gcc –c ./main.c ./foo.c
>> objdump –R ./a.out
[sic]
OFFSET   TYPE              VALUE
[sic]
08049674 R_386_JUMP_SLOT   printf
```

- Relocation table of a.out contains printf since it has not yet been resolved (no printf definition in the source files).

Libraries

- Not all code in a program is what you wrote (e.g. printf in C standard library).
- Code written by others can be included in your own program.
- How?

Linking

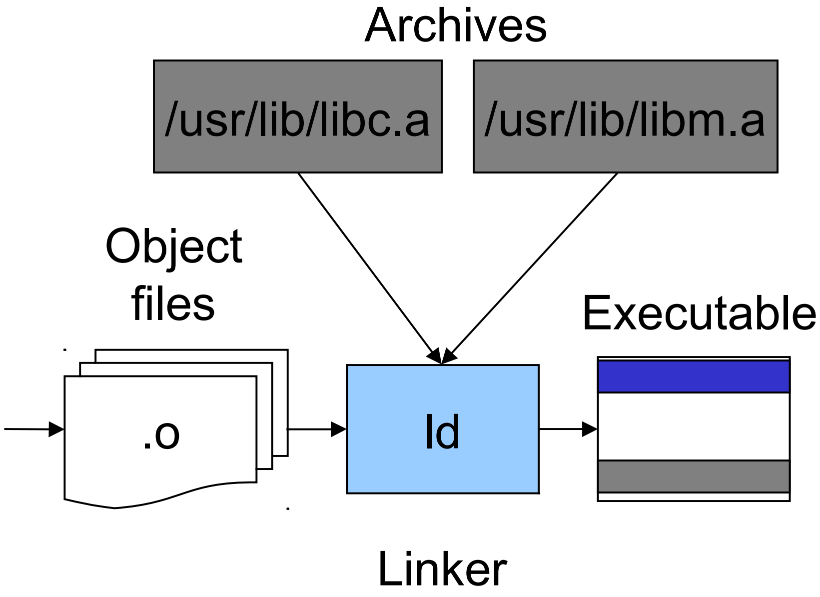
- Static Linking
  - Copy code into executable at compile time
  - Done by linker

- Dynamic Linking
  - Copy code into Address Space at load time or later
  - Done by link loader

Static Linking

```
#include <stdio.h>
#include <math.h>

int main() {
   printf("The sqrt of 9 is %f\n", sqrt(9));
   return 0;
}
```



Static Linking

```
#include <stdio.h>
int global = 0;
void foo();
int main()
{
  foo();
  printf("global=%d\n", global);
  return 0;
}
```

<main.c>

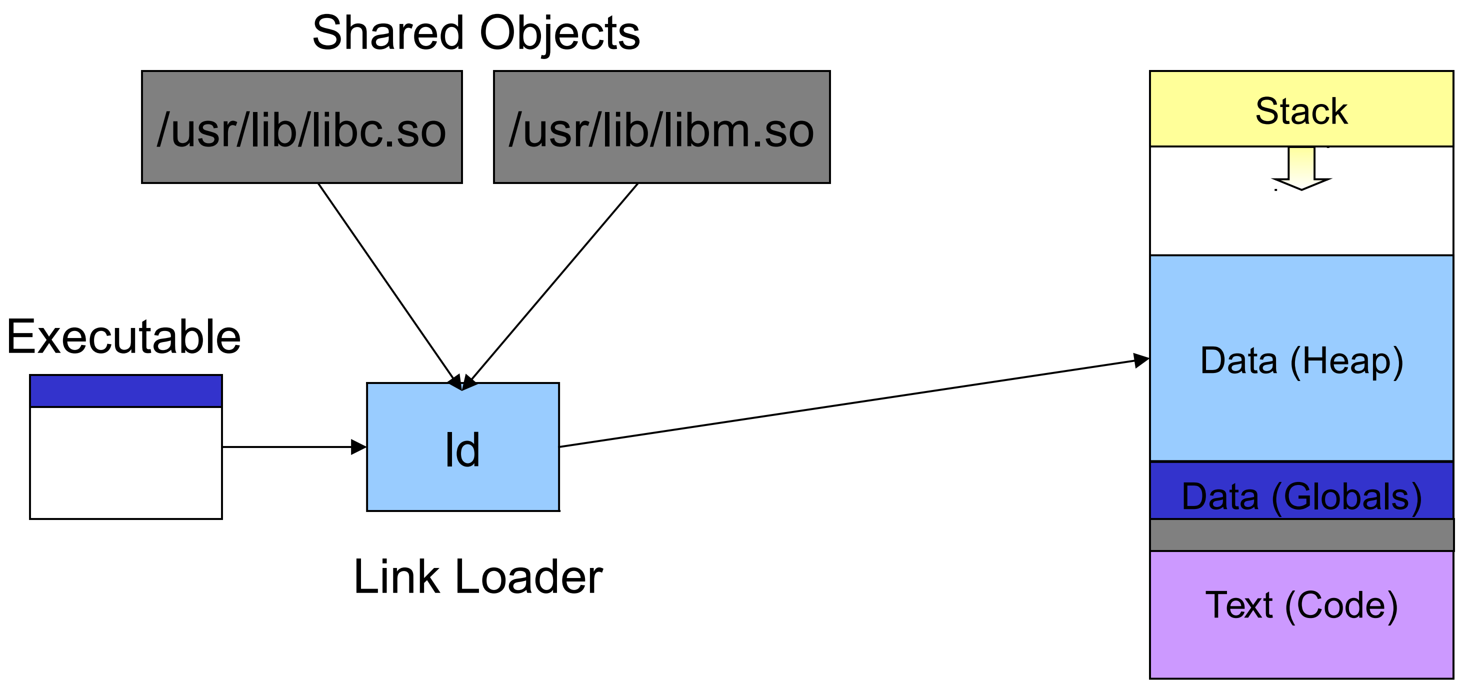
```
extern int global;
void foo() { global += 10; }
```

<foo.c>

```
>> gcc -m32 -static ./main.c ./foo.c -lc
>> ls –l ./a.out
-rwxr-xr-x 1 wahn UNKNOWN1 639385 Sep 30 15:20 ./a.out
>> objdump –R ./a.out
objdump: ./a.out: not a dynamic object
objdump: ./a.out: Invalid operation
```

- `-static` tells gcc linker to do static linking
- `-lc` links in library libc.a (found in /usr/lib)
- Static linking copies over all libraries into binary (just like with object files)
  - That's why a.out is so large!

Dynamic Linking



Dynamic Linking

```
#include <stdio.h>
int global = 0;
void foo();
int main()
{
  foo();
  printf("global=%d\n", global);
  return 0;
}
```

<main.c>

```
extern int global;
void foo() { global += 10; }
```

<foo.c>

```
>> gcc –m32 ./main.c ./foo.c -lc
>> ls –l ./a.out
-rwxr-xr-x 1 wahn UNKNOWN1 4769 Sep 30 15:26 ./a.out
>> ldd ./a.out
        linux-gate.so.1 =>  (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x00bbd000)
        /lib/ld-linux.so.2 (0x00b9b000)
```

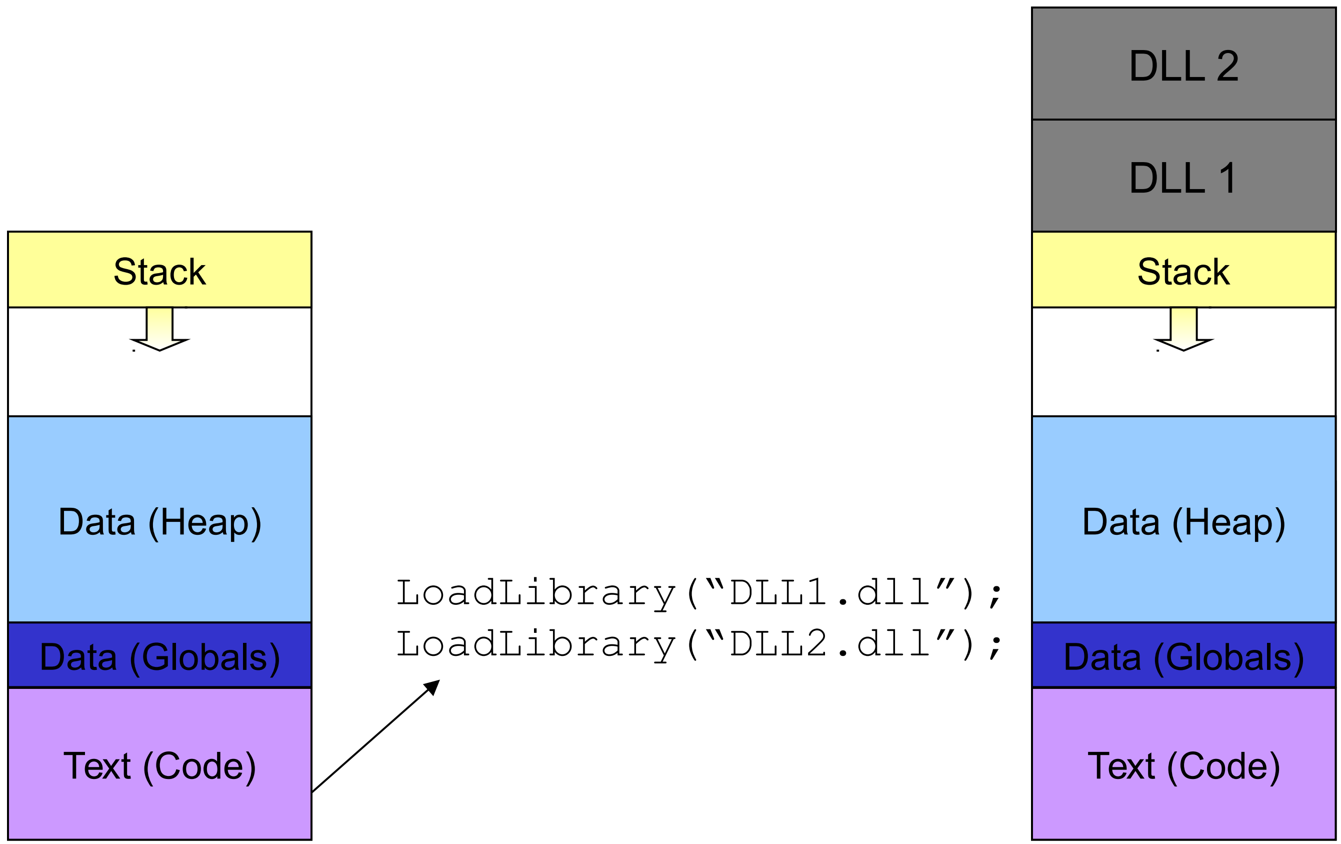
- By default gcc does dynamic linking
- Now ./a.out is much smaller!
- `-lc` marks dependency on libc.so
- ldd prints out shared library dependencies
  - Program will not start if not satisfied

Dynamic Linking Pros/Cons

- Pros
  - More efficient use of storage
  - More efficient use of memory
    - One copy of library can be mapped to multiple processes
  - Much easier to fix bugs – don’t have to statically relink every application
- Cons
  - Address resolution done at load time – slow
  - Versioning of multiple dynamic libraries (DLL Hell)
  - Security holes due to replacement of libraries

Dynamic Loading

and Dynamic Link Libraries



Function Pointers

- How do we call a function when we can’t be sure what address it’s loaded at?
- Need a level of indirection.
- Use a function pointer.

Dynamic Loading

```c
#include <dlfcn.h>
int main() {
  void *handle;
  int (*pf)(const char *format, ...);
  /* open the C standard library */
  handle = dlopen("libc.so.6", RTLD_LAZY);
  /* find the address of the printf function */
  pf = dlsym(handle, "printf");
  (*pf)("Hello world!\n");
  dlclose(handle);
  return 0;
}
```

```
>> gcc -ldl ./dlsym.c
>> ./a.out
Hello world!
```

Problem 1

- You have a commodity computer. There is a text file of size 4TB, written on the 12TB HDD. The file contains short integers only. Every line of the file contains exactly one number. The numbers are different.
- Propose a solution for sorting the file with minimal number of operations.

Problem 2

- Implement the command ***wc***. When run, the program should get 2 arguments. The first one is an option and the second one is a file name: *wc* <*option*> <*file_name*>.
- Option ‘***c’*** instructs the program to count the number of characters. Option ‘***w’*** instructs it to count the number of words and option ‘***l’*** – the number of lines.

Problem 3

- Write a function in C, which gets a string as input parameter and checks if it is a palindrome. The function should return 1, if the string is a palindrome and -1 otherwise.
- A **palindrome** is a word, phrase, number, or other sequence of characters which reads the same backward or forward.