# Object Modules

- The binary output of compiling (and optionally combining) one or more source modules is called an object module.
  - An object file is such a module stored as a binary file in a system environment.
  - E.g., .o files for C source; .class files for Java source; .exe, .dll, .so, .a, .app files; etc.

# Object Modules

- The binary output of compiling (and optionally combining) one or more source modules is called an object module.
  - An object file is such a module stored as a binary file in a system environment.
  - E.g., .o files for C source; .class files for Java source; .exe, .dll, .so, .a, .app files; etc.
- From a data perspective, an object file is simply a self-describing sequence of bytes, according to some file format specification.
  - Generally ISA/OS-specific and therefore non-portable: a.out ("assembler output", original Unix), PE ("Portable Executable", Windows), Mach-O ("Mach Object", macOS), ELF ("Executable and Linkable Format", Linux).
  - "Fat binaries" are executable object files that have been expanded with native code for multiple instruction sets and/or operating systems, which can therefore be run on multiple ISA/OS combinations.

# Types of Object Modules

- Relocatable object module
  - Contains binary code and data in a form suitable for combining.
  - Cannot be executed in isolation.

```
gcc -o prog prog1.c prog2.c -lm
```

# Types of Object Modules

- Relocatable object module
  - Contains binary code and data in a form suitable for combining.
  - Cannot be executed in isolation.
- Executable object module
  - Contains binary code and data in a form suitable for being loaded into memory and executed.

```
gcc -o prog prog1.c prog2.c -lm
```

# Types of Object Modules

- Relocatable object module
  - Contains binary code and data in a form suitable for combining.
  - Cannot be executed in isolation.
- Executable object module
  - Contains binary code and data in a form suitable for being loaded into memory and executed.
- Shared object module
  - A special type of relocatable object module that can be loaded into memory and linked dynamically, either at program load time or during program execution.

```
gcc -o prog prog1.c prog2.c -lm
```

# ELF Relocatable Object File Format

| | | |
|---|---|---|
| ELF header (16 B) | | Bootstrapping information for file |
| `.text` | | Machine code of compiled module |
| `.rodata` | | Read-only data (e.g., `printf` format strings, jump tables) |
| `.data` | | Initialized global / static variables |
| `.bss` | | Uninitialized static variables + those initialized to 0 |
| `.symtab` | | Symbol table |
| `.rel.text` | | List of `.text` locations that need to be modified |
| `.rel.data` | | List of `.data` locations that need to be modified |
| `.debug` | optional | Debugging symbol table |
| `.line` | optional | Mapping between source line #s and `.text` instructions |
| `.strtab` | | String table for symbols in `.symtab`, `.debug`, and section names |
| Section Header Table | | Fixed-size entries describing each section |

# Introduction to Linking: Source Files

main.c

```
int sum(int*, int);

int array[2] = {1,2};

int main(void) {
    int val = sum(array, 2);
    return val;
}
```

- The symbol `sum` is declared.
- The symbol `sum` is not defined.
- The symbol `sum` is referenced.

# Introduction to Linking: Source Files

main.c

```
int sum(int*, int);

int array[2] = {1,2};

int main(void) {
    int val = sum(array, 2);
    return val;
}
```

- The symbol `sum` is declared.
- The symbol `sum` is not defined.
- The symbol `sum` is referenced.

sum.c

```
int sum(int *a, int n) {
    int s = 0;
    for (int i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

- The symbol `sum` is defined.

# Introduction to Linking: Source Files

main.c

```
int sum(int*, int);

int array[2] = {1,2};

int main(void) {
    int val = sum(array, 2);
    return val;
}
```

sum.c

```
int sum(int *a, int n) {
    int s = 0;
    for (int i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

- The symbol `sum` is declared.
- The symbol `sum` is not defined.
- The symbol `sum` is referenced.

- The symbol `sum` is defined.

```
gcc –o run main.c sum.c
```

# Introduction to Linking: Object Files

```
main.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 g     O .data      0000000000000008 array
0000000000000000 g     F .text      0000000000000022 main
0000000000000000         *UND*      0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*      0000000000000000 sum

Contents of section .data:
 0000 01000000 02000000                    ........

Disassembly of section .text:

0000000000000000 <main>:
   0:       48 83 ec 18             sub    $0x18,%rsp
   4:       be 02 00 00 00          mov    $0x2,%esi
   9:       48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi # 10 <main+0x10>
            c: R_X86_64_PC32        array-0x4
  10:       e8 00 00 00 00          callq  15 <main+0x15>
            11: R_X86_64_PLT32      sum-0x4
  15:       89 44 24 0c             mov    %eax,0xc(%rsp)
  19:       8b 44 24 0c             mov    0xc(%rsp),%eax
  1d:       48 83 c4 18             add    $0x18,%rsp
  21:       c3                      retq
```

# Introduction to Linking: Object Files

```
main.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 g     O .data      0000000000000008 array
0000000000000000 g     F .text      0000000000000022 main
0000000000000000         *UND*      0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*      0000000000000000 sum

Contents of section .data:
 0000 01000000 02000000                    ........

Disassembly of section .text:

0000000000000000 <main>:
   0:       48 83 ec 18             sub    $0x18,%rsp
   4:       be 02 00 00 00          mov    $0x2,%esi
   9:       48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi # 10 <main+0x10>
            c: R_X86_64_PC32        array-0x4
  10:       e8 00 00 00 00          callq  15 <main+0x15>
            11: R_X86_64_PLT32      sum-0x4
  15:       89 44 24 0c             mov    %eax,0xc(%rsp)
  19:       8b 44 24 0c             mov    0xc(%rsp),%eax
  1d:       48 83 c4 18             add    $0x18,%rsp
  21:       c3                      retq
```

```
sum.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 g     F .text      000000000000004b sum

Disassembly of section .text:

0000000000000000 <sum>:
   0:       48 89 7c 24 e8          mov    %rdi,-0x18(%rsp)
            <more lines of code>
  4a:       c3                      retq
```

# Introduction to Linking: Executable File

```
run:     file format elf64-x86-64

Contents of section .data:
 201000 00000000 00000000 08102000 00000000  .......... .....
 201010 01000000 02000000                    ........

Disassembly of section .text:

00000000000005fa <main>:
 5fa:       48 83 ec 18             sub    $0x18,%rsp
 5fe:       be 02 00 00 00          mov    $0x2,%esi
 603:       48 8d 3d 06 0a 20 00    lea    0x200a06(%rip),%rdi # 201010 <array>
 60a:       e8 0d 00 00 00          callq  61c <sum>
 60f:       89 44 24 0c             mov    %eax,0xc(%rsp)
 613:       8b 44 24 0c             mov    0xc(%rsp),%eax
 617:       48 83 c4 18             add    $0x18,%rsp
 61b:       c3                      retq

000000000000061c <sum>:
 61c: 48 89 7c 24 e8       mov    %rdi,-0x18(%rsp)
            <more lines of code>
 666: c3                   retq
```

# What The Static Linker Does

- The (static) linker is solely concerns with the externally-visible symbols of the relocatable modules it is linking.
  - It is not concerned with handling local variables and other symbols that are not visible outside individual relocatable modules.

# What The Static Linker Does

- The (static) linker is solely concerns with the externally-visible symbols of the relocatable modules it is linking.
  - It is not concerned with handling local variables and other symbols that are not visible outside individual relocatable modules.
- The symbol table of a relocatable module provides information about the externally-visible symbols defined in it and the external symbols referenced by it.
  - These definitions and references have to be matched up. This task is called symbol resolution.

# What The Static Linker Does

- The (static) linker is solely concerns with the externally-visible symbols of the relocatable modules it is linking.
  - It is not concerned with handling local variables and other symbols that are not visible outside individual relocatable modules.
- The symbol table of a relocatable module provides information about the externally-visible symbols defined in it and the external symbols referenced by it.
  - These definitions and references have to be matched up. This task is called symbol resolution.
- The relocation entries of a relocatable module provide information about which symbol references in it need to be adjusted (and how) when combining multiple relocatable modules.
  - Each module is generated in its standalone local coordinate system.
  - These multiple local coordinate systems have to be combined correctly into a single (global) coordinate system in the final executable. This task is called relocation.