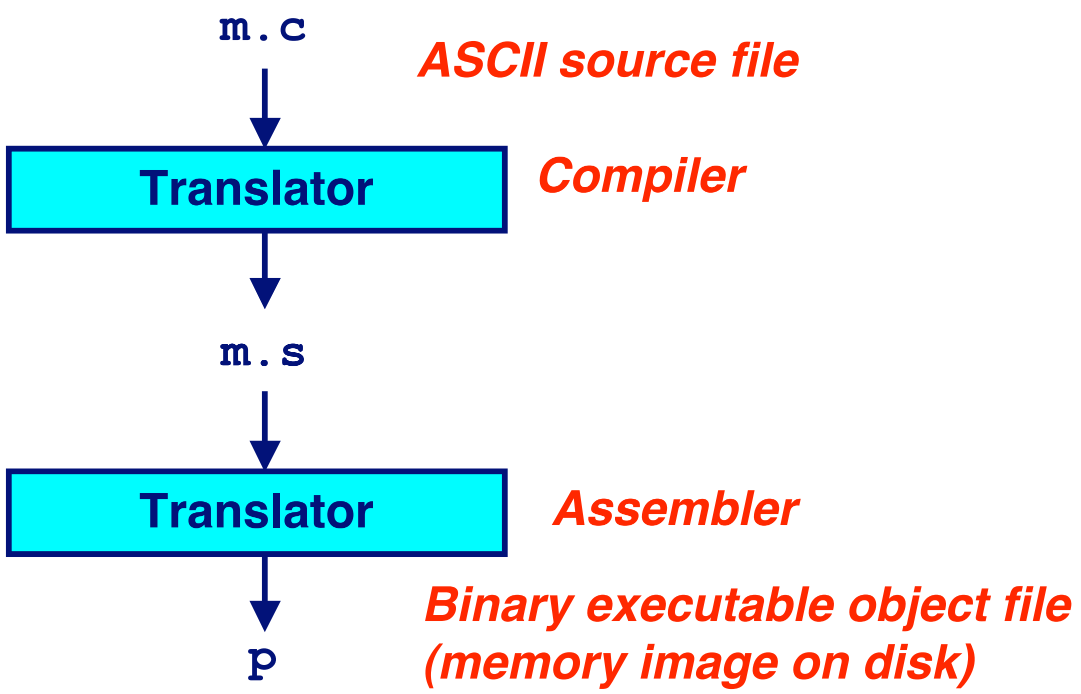
Systems I

Linking I

Topics

- Assembly and symbol resolution
- Static linking

A Simplistic Program Translation Scheme



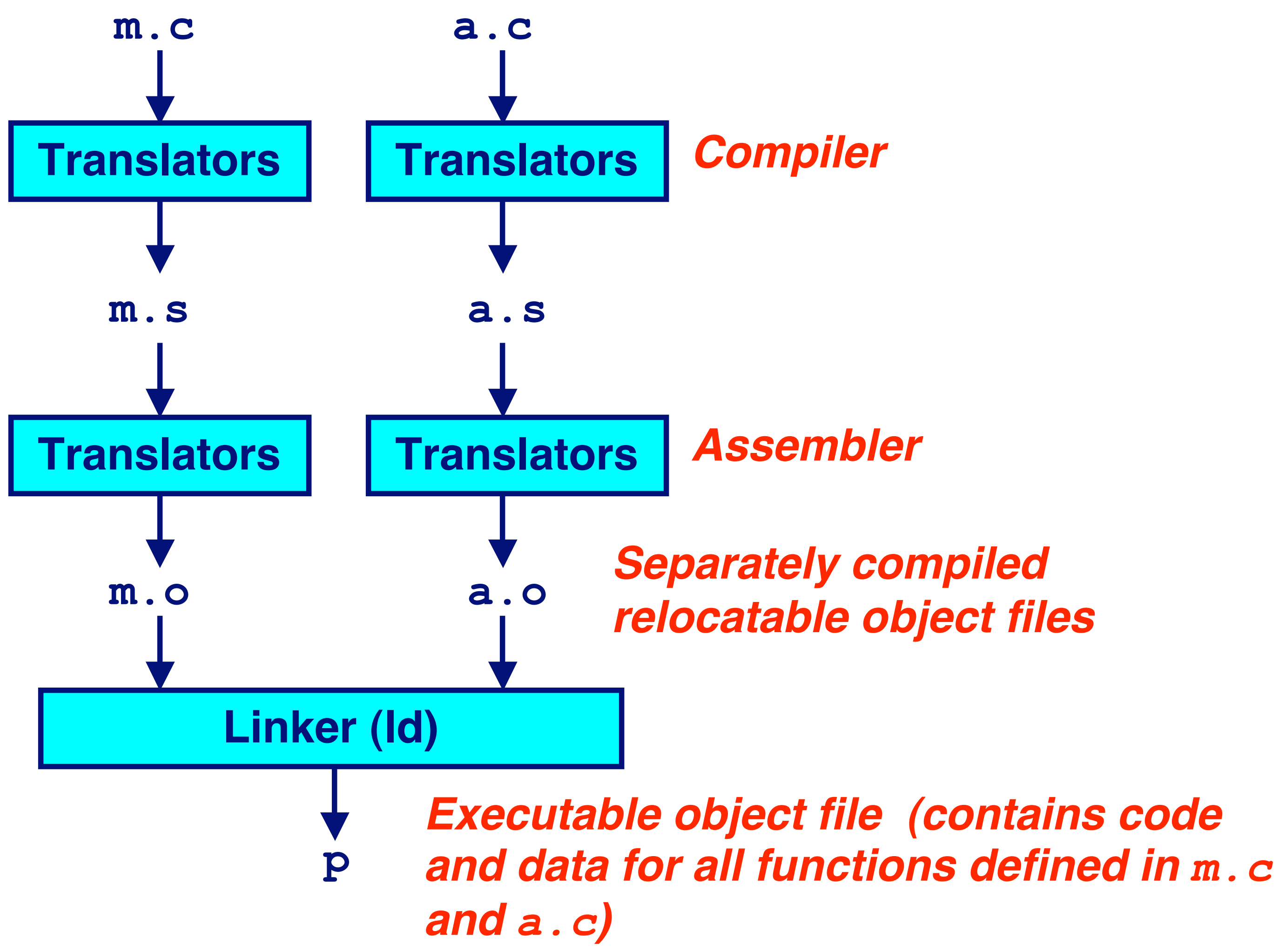
Problems:

- **Efficiency: small change requires complete recompilation**
- **Modularity: hard to share common functions (e.g. `printf`)**

Solution:

- ***Static linker (or linker)***

A Better Scheme Using a Linker


m.c
a.c
Translators
Translators
Compiler
m.s
a.s
Translators
Translators
Assembler
m.o
a.o
Separately compiled relocatable object files
Linker (ld)
p
Executable object file (contains code and data for all functions defined in m.c and a.c)

Translating the Example Program

Compiler driver **coordinates all steps in the translation and linking process.**

- Typically included with each compilation system (e.g., `gcc`)
- Invokes preprocessor (`cpp`), compiler (`cc1`), assembler (`as`), and linker (`ld`).
- Passes command line arguments to appropriate phases

Example: create executable `p` from `m.c` and `a.c`:

```
bass> gcc -O2 -v -o p m.c a.c
cpp [args] m.c /tmp/cca07630.i
cc1 /tmp/cca07630.i m.c -O2 [args] -o /tmp/cca07630.s
as [args] -o /tmp/cca076301.o /tmp/cca07630.s
<similar process for a.c>
ld -o p [system obj files] /tmp/cca076301.o /tmp/cca076302.o
bass>
```

Compiling/Assembling

C Code

```
double sum_loop(int val) {
  int sum = 0;
  double pi = 3.14;
  int i;

  for(i=3; i<=val; i++) {
    sum = sum + i;
  }
  return sum+pi;
}
```

Obtain with command

```
gcc -O -S sum_loop.c
```

Produces file `code.s`

Generated Assembly

```
sum_loop:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %ecx
        movl    $0, %edx
        cmpl    $2, %ecx
        jle     .L4
        movl    $0, %edx
        movl    $3, %eax
.L5:
        addl    %eax, %edx
        addl    $1, %eax
        cmpl    %eax, %ecx
        jge     .L5
.L4:
        pushl   %edx
        fildl   (%esp)
        leal    4(%esp), %esp
        faddl   .LC0
        popl    %ebp
        ret
.LC0:
        .long   1374389535
        .long   1074339512
```

Role of the Assembler

Translate assembly code into machine code

- Compiled or hand-generated

Translate data into binary codes (using directives)

Resolve symbols

- Translate into relocatable offsets

Error check

- Syntax checking
- Ensure that constants are not too large for fields

Where did the labels go?

Disassembled Object Code

```
08048334 <sum_loop>:
 8048334:       55                      push   %ebp
 8048335:       89 e5                   mov    %esp,%ebp
 8048337:       8b 4d 08                mov    0x8(%ebp),%ecx
 804833a:       ba 00 00 00 00          mov    $0x0,%edx
 804833f:       83 f9 02                cmp    $0x2,%ecx
 8048342:       7e 13                   jle    8048357 <sum_loop+0x23>
 8048344:       ba 00 00 00 00          mov    $0x0,%edx
 8048349:       b8 03 00 00 00          mov    $0x3,%eax
 804834e:       01 c2                   add    %eax,%edx
 8048350:       83 c0 01                add    $0x1,%eax
 8048353:       39 c1                   cmp    %eax,%ecx
 8048355:       7d f7                   jge    804834e <sum_loop+0x1a>
 8048357:       52                      push   %edx
 8048358:       db 04 24                fildl  (%esp)
 804835b:       8d 64 24 04             lea    0x4(%esp),%esp
 804835f:       dc 05 50 84 04 08       faddl  0x8048450
 8048365:       5d                      pop    %ebp
 8048366:       c3                      ret
```

Label Resolution

Disassembled Object Code

```
8048342:        7e 13                    jle     8048357 <sum_loop+0x23>
…
8048355:        7d f7                    jge     804834e <sum_loop+0x1a>
…
804835f:        dc 05 50 84 04 08        faddl   0x8048450
```

Byte relative offsets for jle and jge

- jle: 13 bytes forward
- jge: 9 bytes bytes backward (two's comp. of xf7)

Relocatable absolute address

- faddl x8048450

How does the assembler work

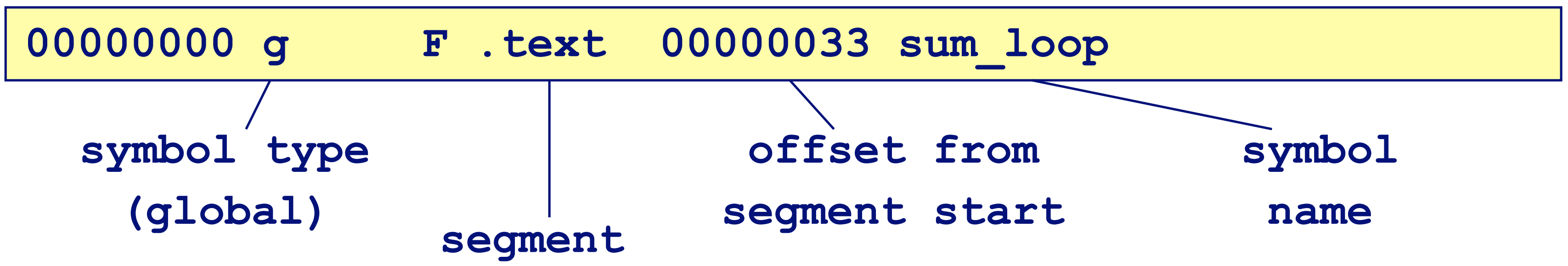
One pass

- Record label definitions
- When use is found, compute offset

Two pass

- Pass 1: scan for label instantiations - creates symbol table
- Pass 2: compute offsets from label use/def
- Can detect if computed offset is too large for assembly instruction

Symbol Table



```
00000000 g     F .text  00000033 sum_loop
```

Tracks location of symbols in object file

- Symbols that can be resolved need not be included
- Symbols that may be needed during linking must be included

What Does a Linker Do?

Merges object files

- Merges multiple relocatable (`.o`) object files into a single executable object file that can loaded and executed by the loader.

Resolves external references

- As part of the merging process, resolves external references.
  - *External reference*: reference to a symbol defined in another object file.

Relocates symbols

- Relocates symbols from their relative locations in the `.o` files to new absolute positions in the executable.
- Updates all references to these symbols to reflect their new positions.
  - References can be in either code or data
    - » code: `a();            /* reference to symbol a */`
    - » data: `int *xp=&x;   /* reference to symbol x */`

Why Linkers?

Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library

Efficiency

- Time:
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
- Space:
  - Libraries of common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.

Executable and Linkable Format (ELF)

Standard binary format for object files

Derives from AT&T System V Unix

- Later adopted by BSD Unix variants and Linux

One unified format for

- Relocatable object files (`.o`),
- Executable object files
- Shared object files (`.so`)

Generic name: ELF binaries

Better support for shared libraries than old `a.out` formats.

ELF Object File Format

Elf header

- Magic number, type (.o, exec, .so), machine, byte ordering, etc.

Program header table

- Page size, virtual addresses memory segments (sections), segment sizes.

`.text` section

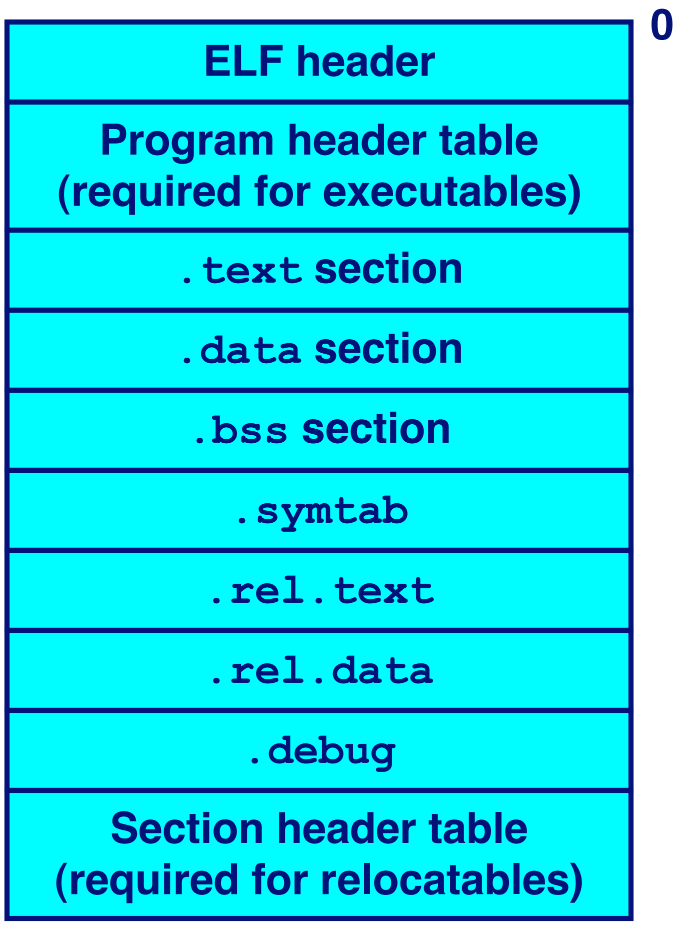
- Code

`.data` section

- Initialized (static) data

`.bss` section

- Uninitialized (static) data
- “Block Started by Symbol”
- “Better Save Space”
- Has section header but occupies no space



0
ELF header
Program header table (required for executables)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (required for relocatables)

ELF Object File Format (cont)

`.symtab` **section**

- Symbol table
- Procedure and static variable names
- Section names and locations

`.rel.text` **section**

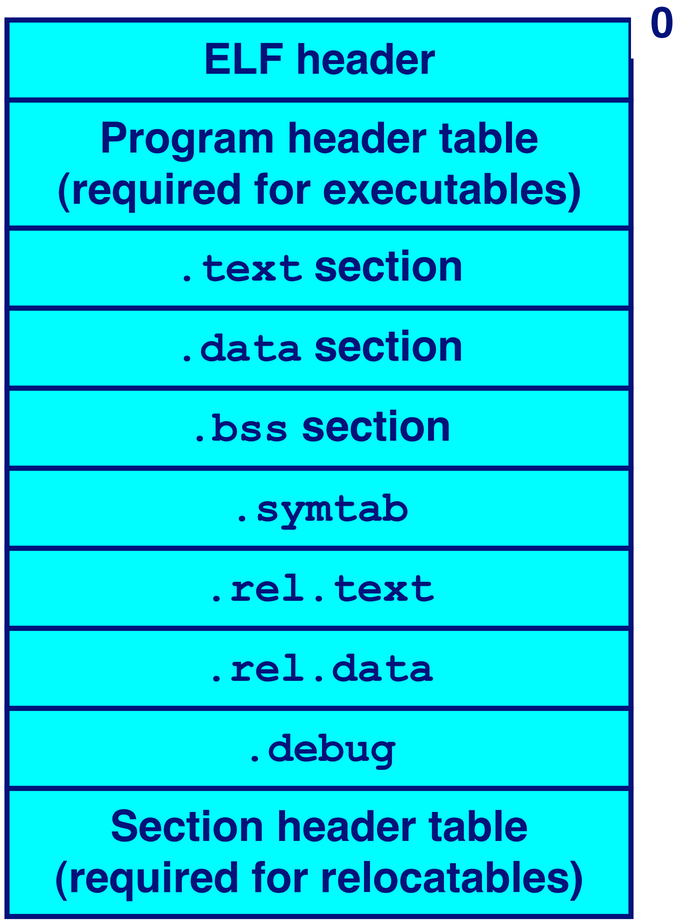
- Relocation info for `.text` section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

`.rel.data` **section**

- Relocation info for `.data` section
- Addresses of pointer data that will need to be modified in the merged executable

`.debug` **section**

- Info for symbolic debugging (`gcc -g`)


0
ELF header
Program header table (required for executables)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (required for relocatables)

Example C Program

m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

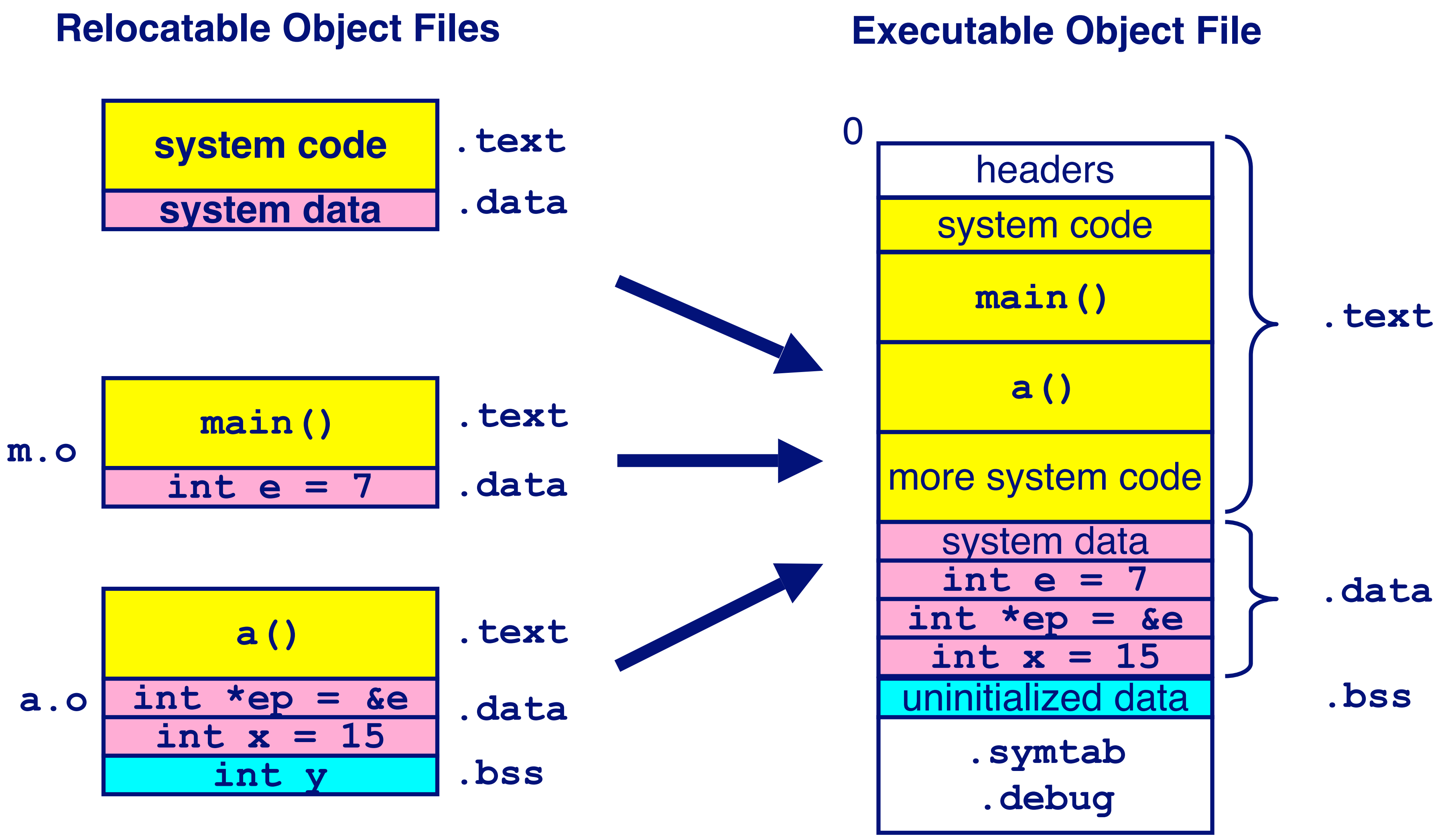
a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

Merging Relocatable Object Files into an Executable Object File



Summary

Today

- Compilation/Assembly/Linking
- Symbol resolution and symbol tables

Next Time

- Code and data relocation
- Loading
- Libraries
- Dynamically linked libraries