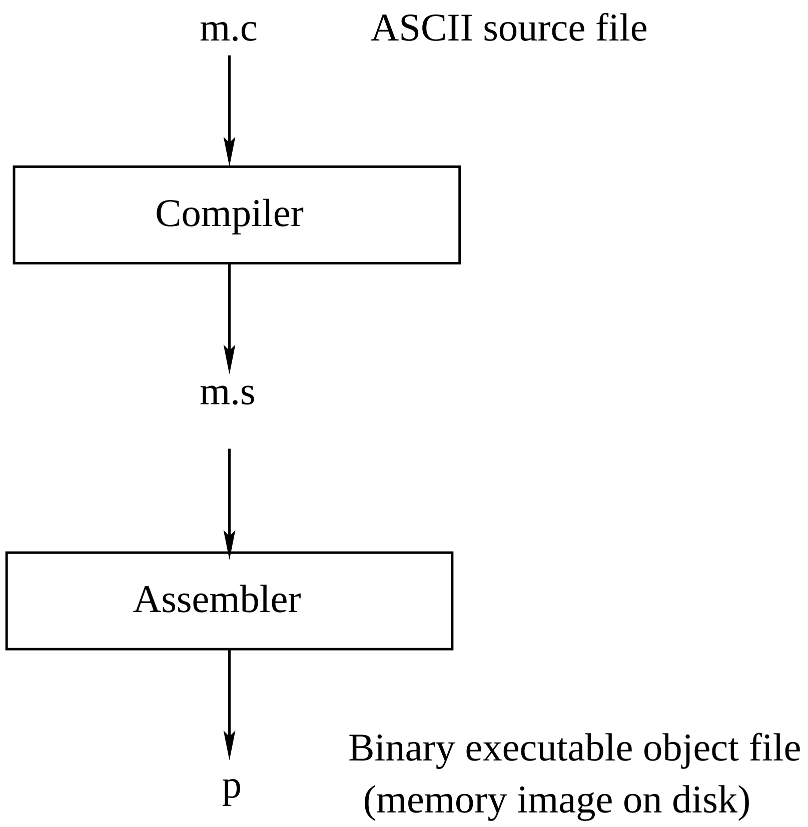
CS429: Computer Organization and Architecture

Linking I

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: April 23, 2014 at 13:31

A Simplistic Translation Scheme

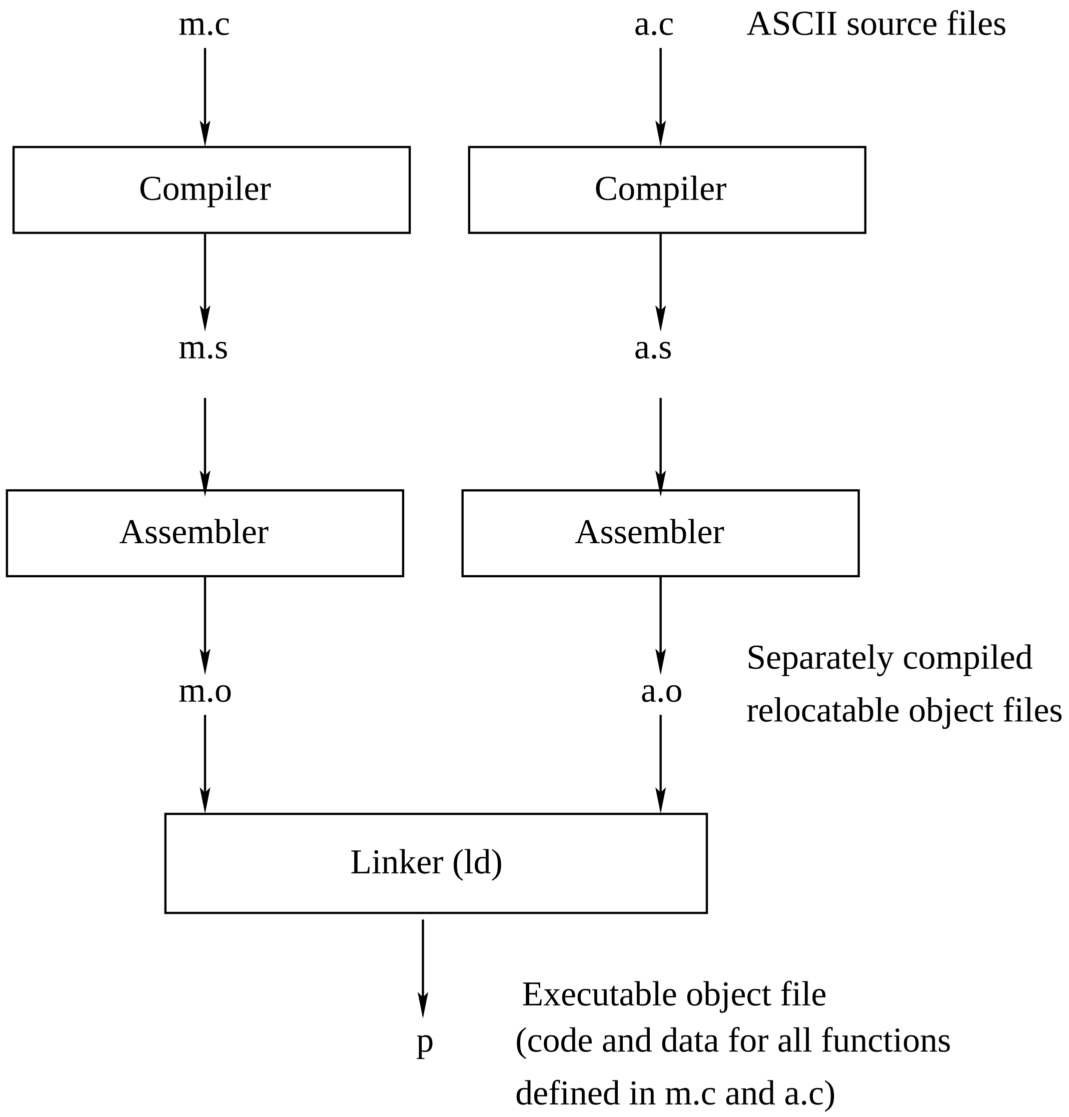


Problems:

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

Solution: Static linker (or linker).

Better Scheme Using a Linker



Linking is the process of combining various pieces of code and data into a single file that can be *loaded* (copied) into memory and executed.

Linking could happen at:

- compile time;
- load time;
- run time.

Linking

A *linker* takes representations of separate program modules and combines them into a single *executable*.

This involves two primary steps:

1. *Symbol resolution:* associate each symbol reference throughout the set of modules with a single symbol definition.
2. *Relocation:* associate a memory location with each symbol definition, and modify each reference to point to that location.

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 the preprocessor (cpp), compiler (cc1), assembler (as), and linker (ld).
- Passes command line arguments to the appropriate phases

Example: Create an executable p from m.c and a.c:

```
> 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
>
```

Compiling/Assembling

C Code

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

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

Obtain with command:

```
gcc -O -S sum.c
```

Produces file code.s

```
sum:
        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 (compiled or hand generated) into machine code.
- Translate data into binary code (using directives).
- Resolve symbols—translate into relocatable offsets.
- Error checking:
    - Syntax checking;
    - Ensure that constants are not too large for fields.

Where Did the Labels Go?

Disassembled Object Code

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

Byte relative offsets for jle and jge:

- jge: 13 bytes forward
- jge: 9 bytes backward (two's complement of 0xf7)

Relocatable absolute address:

- faddl: 0x8048450

How Does the Assembler Work?

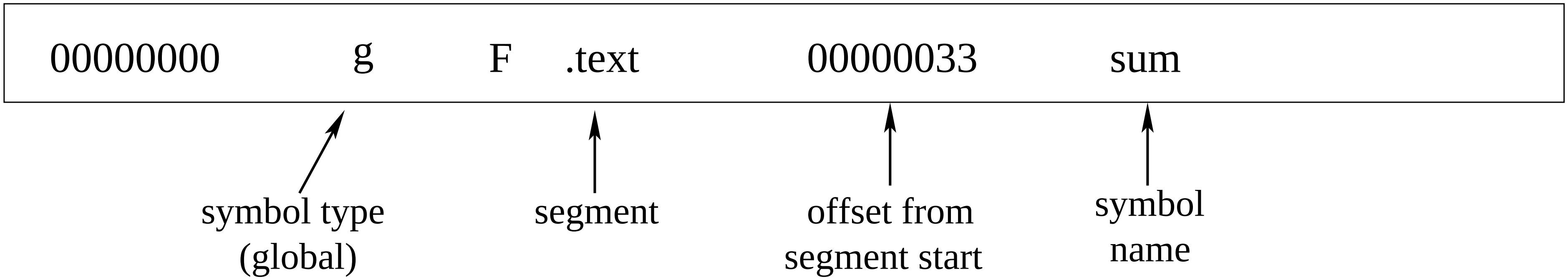
One Pass

- 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

symbol type (global)

segment

offset from segment start

symbol name

The symbol table tracks the location of symbols in the 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 be 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: `*xp = &x;           /* reference to symbol x */`

Why Linkers?

Modularity

- Programs can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions shared by multiple programs (e.g., math library, standard C library)

Efficiency

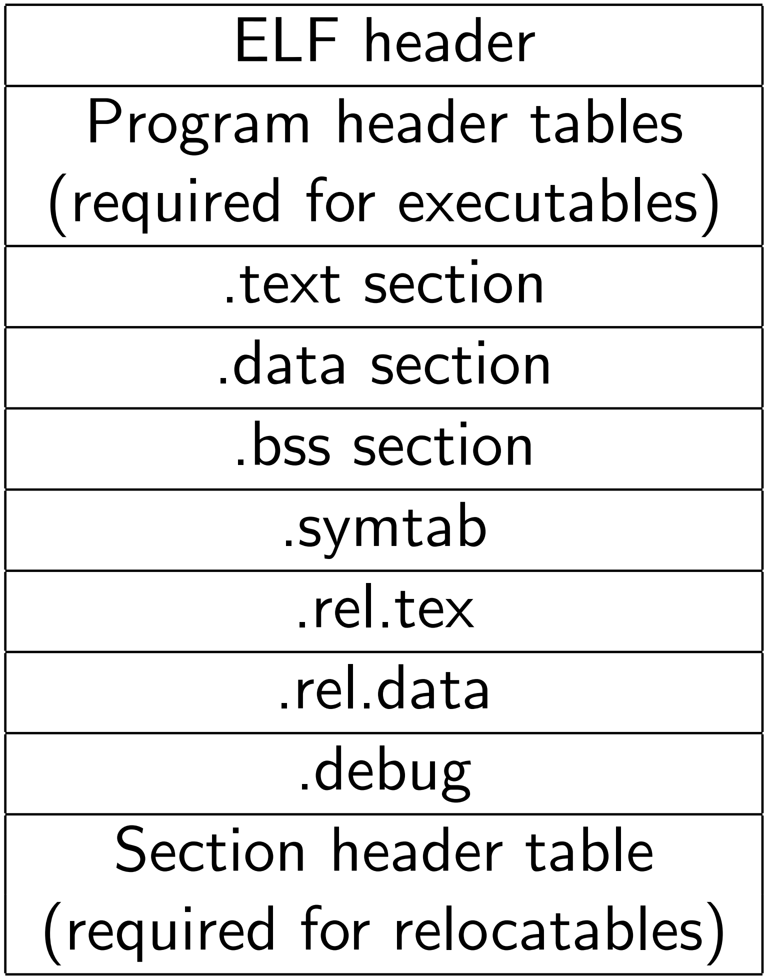
- Time:
  - Change one source file, recompile, and then relink.
  - No need to recompile other source files.
- Space:
  - Libraries of common functions can be aggregated into a single file.
  - Yet executable files and running machine images contain only code for the functions they actually use.

Executable and Linkable Format (ELF)

- Standard binary format for object files.
- Derives from AT&T System V Unix, and later adopted by BSD Unix variants and Linux.
- One unified format for:
  - Relocatable object files (.o),
  - Executable object files,
  - Shared object files (.so).
- The generic name is *ELF binaries*.
- Better support for shared libraries than the 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 of 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.



ELF header

Program header tables (required for executables)

.text section

.data section

.bss section

.symtab

.rel.tex

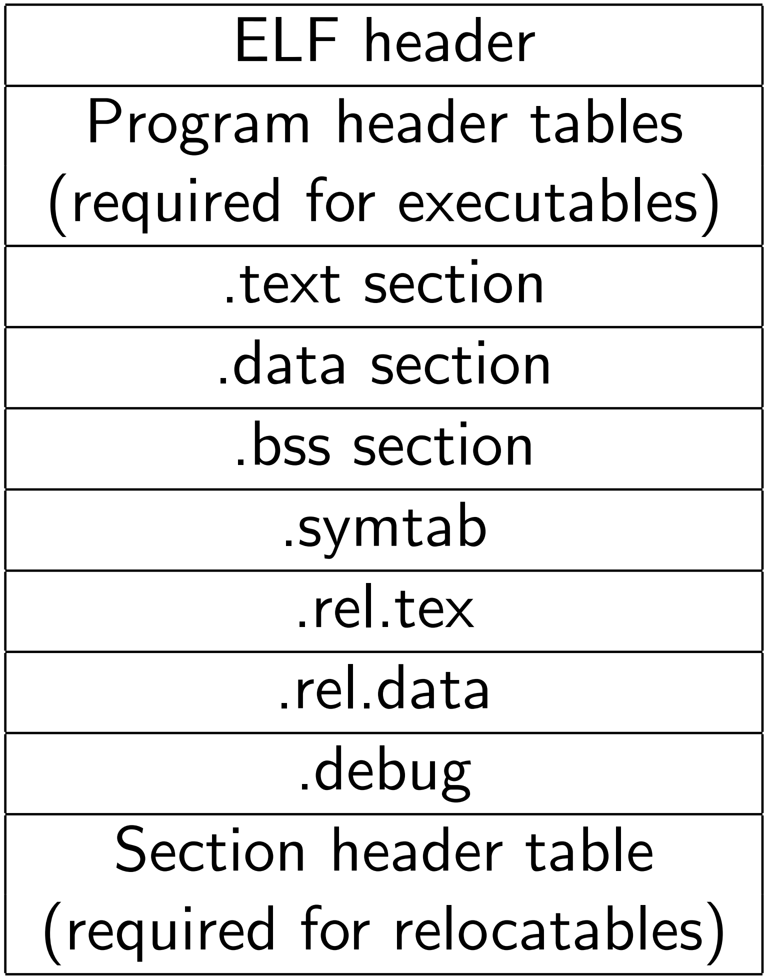
.rel.data

.debug

Section header table (required for relocatables)

ELF Object File Format (continued)

- `.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 needing modification in the merged executable
- `.debug` section
  - Info for symbolic debugging (`gcc -g`)



ELF header

Program header tables
(required for executables)

.text section

.data section

.bss section

.symtab

.rel.tex

.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`

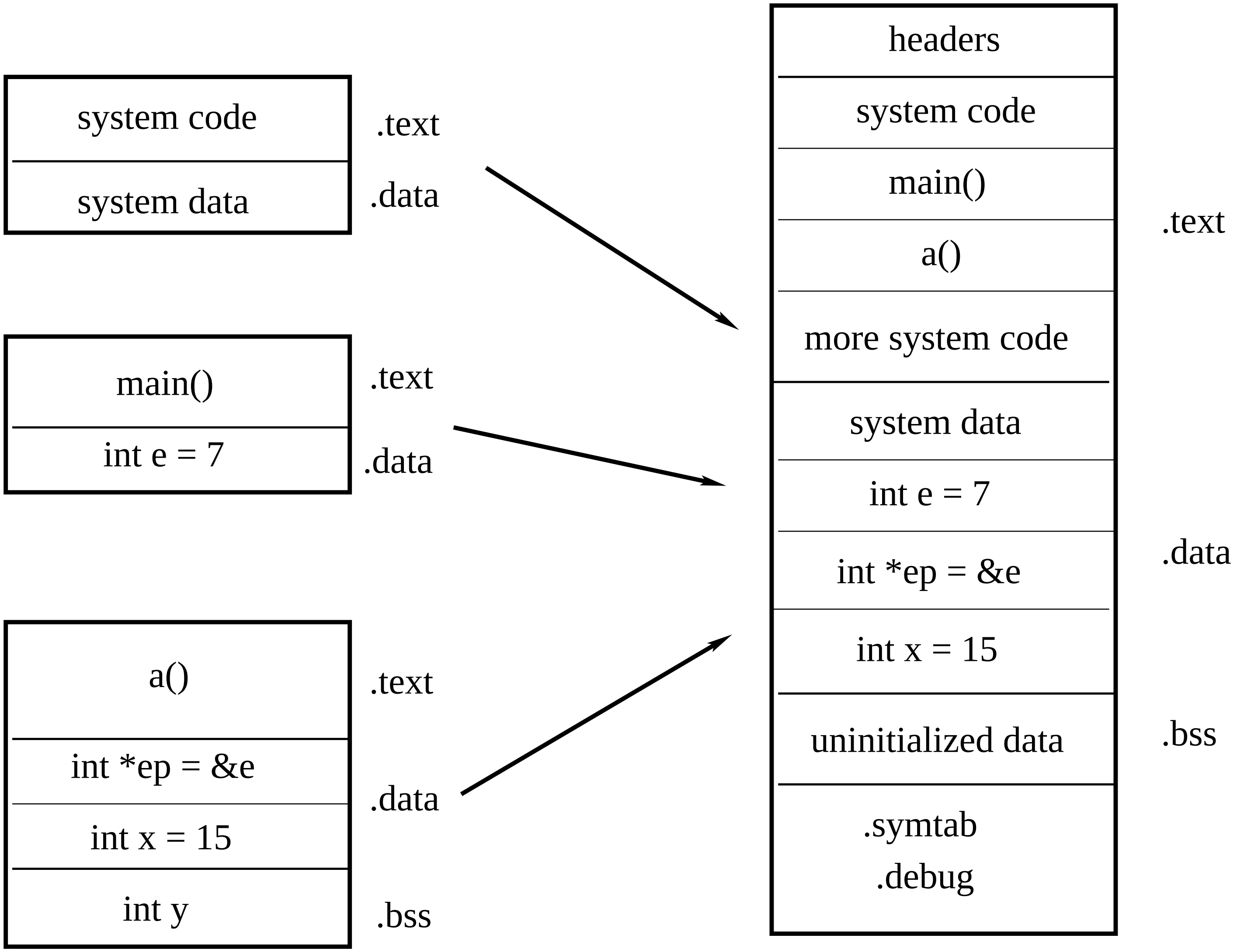
```
extern int e;

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

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

Merging Relocatable Object Files

Relocatable object files are merged into an executable by the Linker. Both are in ELF format.



Summary

This slideset:

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

Next time:

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