

Linking: Basic Concepts & Static Linking

Introduction to Computer Systems
15th Lecture, Nov. 2, 2022

Instructors:

Class 1: Chen Xiangqun, Liu Xianhua

Class 2: Guan Xuetao

Class 3: Lu Junlin

Outline of Linking

- **Linking: combining object files into programs**
 - Object files
 - Linking mechanism
 - Symbols and symbol resolution
 - Relocation
- **Libraries**
- **Dynamic linking, loading & execution**
- **Library inter-positioning**

Why Linkers?

■ Reason 1: 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

Why Linkers? (cont)

■ Reason 2: Efficiency

- Time: Separate compilation
 - Change one source file, compile, and then relink.
 - No need to recompile other source files.
 - Can compile multiple files concurrently.
- Space: Libraries
 - Common functions can be aggregated into a single file...
 - **Option 1: *Static Linking***
 - Executable files and running memory images contain only the library code they actually use
 - **Option 2: *Dynamic linking***
 - Executable files contain no library code
 - During execution, single copy of library code can be shared across all executing processes

Why bother learning about linker?

- **Help you build large programs.**
 - Linking errors link missing modules, libraries or incompatible library may be baffling and frustrating.
- **Help you avoid dangerous errors.**
 - Linkers decisions on symbol reference solving may silently affect the correctness of your program.
- **Help you understand how language scoping rules implemented.**
 - Global vs. local names, What does *static* really means.
- **Help you understand other important systems concepts.**
 - Virtual memory, paging a memory mapping.
- **Enable you to exploit shared libraries.**

Example C Program

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

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

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

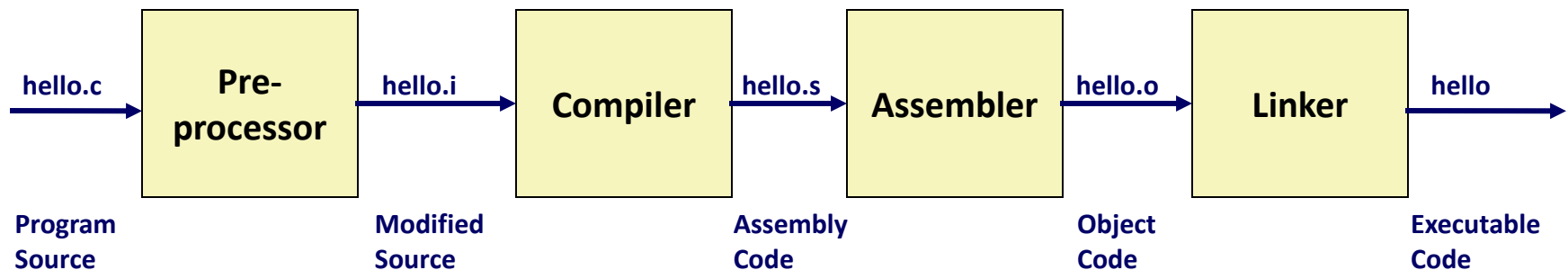
```
int sum(int *a, int n)
{
    int i, s = 0;

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

sum.c

Compiler Driver, GCC as an Example

- Gcc is the compiler driver in compilation toolchain.
- Gcc invokes several other compilation phases
 - cpp, the preprocessor
 - cc1, the compiler
 - as/gas, the assembler
 - ld, the linker
- What does each one do? What are their outputs?




Preprocessor

- First, *gcc* compiler driver invokes *cpp* to generate expanded source
 - Preprocessor just does text substitution/ gcc with option “-E”
 - Converts the C source file to another C source file
 - Expands “#” directives

```
#include <stdio.h>

#define FOO 4

int main(){
    printf("hello, world %d\n", FOO);
}
```



```
...
extern int printf (const char *__restrict __format,
    ...);
...

int main() {
    printf("hello, world %d\n", 4);
}
```


Directives of Preprocessor

■ Included files:

```
#include <foo.h>           /* /usr/include/... */
#include "bar.h"          /* within cwd */
```

■ Defined constants:

```
#define MAXVAL    40000000
```

- By convention, all capitals tells us it's a constant, not a variable.

■ Defined macros:

```
#define MIN(x,y)    ((x)<(y) ? (x) : (y))
```

■ Conditional compilation:

- Code you think you may need again.

■ Control your optimization flags

```
#pragma optimize ("-fno-common", on)
```

■ Extensions for your language or libraries

```
#pragma omp parallel for
```

Example: Conditional Compilation for Debugging

■ Debug print statements

- Include or exclude code using DEBUG condition and `#ifdef`, `#if` preprocessor directive in source code.

*`#ifdef DEBUG` or `#if defined(DEBUG)`
`#endif`*

- Set DEBUG condition via `gcc -D DEBUG` in compilation or within source code via `#define DEBUG`, More readable than commenting out.

```
#include <stdio.h>
int main() {
#ifdef DEBUG
    printf("Debug flag on\n");
#endif
    printf("Hello world\n"); return 0;
}
```

% gcc -o def def.c
% ./def
Hello world

% gcc -D DEBUG -o def def.c
% ./def
Debug flag on
Hello world

Example: Conditional Compilation for Portability

- Compilers with “built in” constants defined
- Use to conditionally include code

- Operating system specific code

```
#if defined(__i386__) || defined(WIN32) || ...
```

- Compiler-specific code

```
#if defined(__ICC) || defined(__INTEL_COMPILER)
```


- Processor-specific code

```
#if defined(__SSE__)
```

Compiler

- Next, *gcc* invokes *cc1* to generate assembly code
 - Translates high-level C code into assembly

```
...  
extern int printf (const char *__restrict __format,  
    ...);  
...  
int main() {  
    printf("hello, world %d\n", 4);  
}
```



```
    .section      .rodata  
    .LC0:  
        .string "hello, world %d\n"  
  
    .text  
main:  
    pushq    %rbp  
    movq     %rsp, %rbp  
    movl     $4, %esi  
    movl     $.LC0, %edi  
    movl     $0, %eax  
    call     printf  
    popq     %rbp  
    ret
```

Assembler

- Furthermore, *gcc* invokes *gas* to generate object code
 - Translates assembly code into binary object code

```
# readelf -a hello | grep rodata
[10] .rodata          PROGBITS          0000000000495d40  00095d40

# readelf -a hello | grep -E "GLOBAL.* main"
1591: 0000000000401190      31 FUNC          GLOBAL DEFAULT      6 main

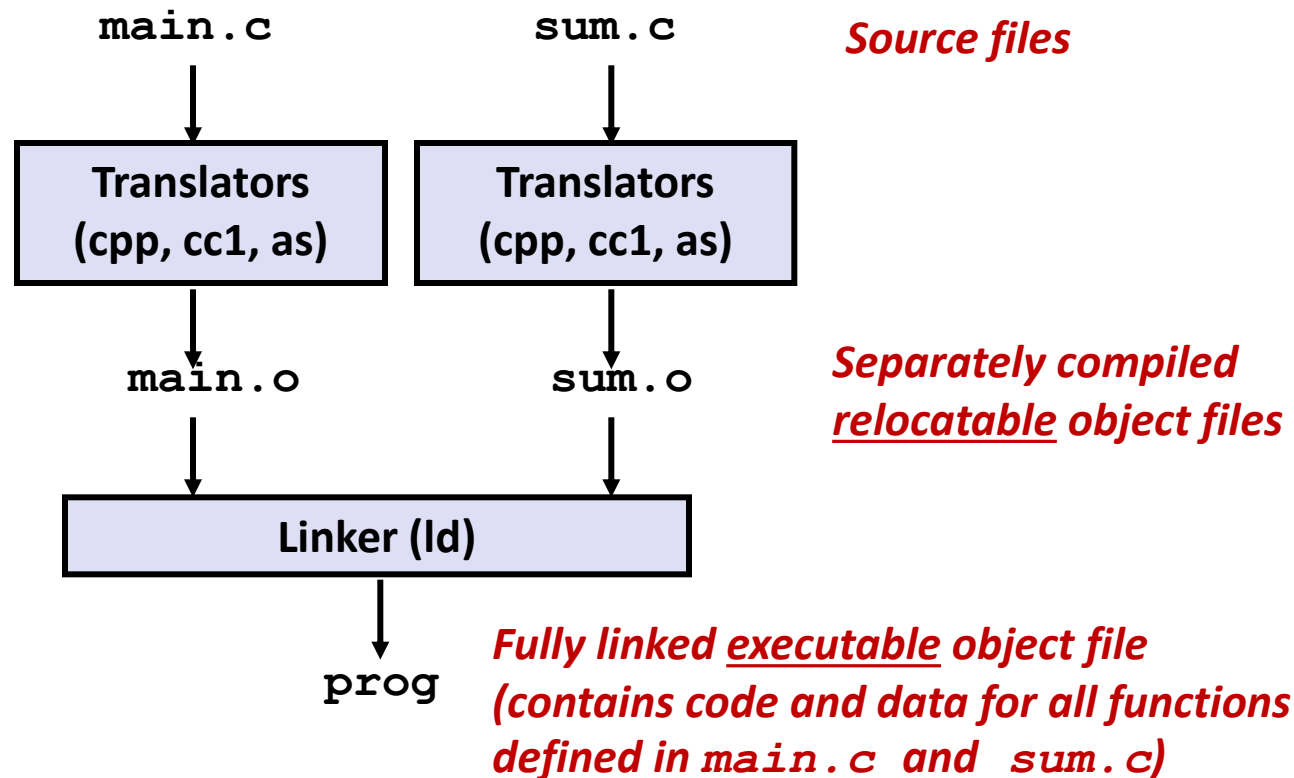
# readelf -x .rodata hello
Hex dump of section '.rodata':
0x00495d40 01000200 68656c6c 6f2c2077 6f726c64 ....hello, world
0x00495d50 2025640a 00464154 414c3a20 6b65726e %d..FATAL: kern

# objdump -d hello
0000000000401190 <main>:
401190:      55                push    %rbp
401191:      48 89 e5          mov     %rsp,%rbp
401194:      be 04 00 00 00    mov     $0x4,%esi
401199:      bf 44 5d 49 00    mov     $0x495d44,%edi
40119e:      b8 00 00 00 00    mov     $0x0,%eax
4011a3:      e8 d8 0e 00 00    callq   402080 <_IO_printf>
4011a8:      b8 00 00 00 00    mov     $0x0,%eax
4011ad:      5d                pop     %rbp
4011ae:      c3                retq
4011af:      90                nop
```

Linking

■ Programs are translated and linked using a *compiler driver*:

- `linux> gcc -Og -o prog main.c sum.c`
- `linux> ./prog`



What Do Linkers Do?

■ Step 1: Symbol resolution

- Programs define and reference *symbols* (global variables and functions):
 - `void swap() {...}` */* define symbol swap */*
 - `swap();` */* reference symbol swap */*
 - `int *xp = &x;` */* define symbol xp, reference x */*
- Symbol definitions are stored in object file (by assembler) in *symbol table*.
 - Symbol table is an array of entries.
 - Each entry includes name, size, and location of symbol.
- **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

Symbols in Example C Program

Declaration

```
int sum(int *a, int n);
int sum1(int *a, int n);

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

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

Definitions

```
int sum(int *a, int n)
{
    int i, s = 0;

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

sum.c

Reference

```
# gcc -c -o main.o main.c
# gcc -c -o sum.o sum.c
# nm main.o
0000000000000000 D array
0000000000000000 T main
                   U sum

# nm sum.o
0000000000000000 T sum
```

You may also try:
objdump -t main.o
objdump -t sum.o

Assembler will not create the entry for function declarations (sum1 in main.o)

What Do Linkers Do? (cont)

■ 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.

Let's look at these two steps in more detail....

Three Kinds of Object Files (Modules)

■ Relocatable object file (`.o` file)

- Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
 - Each `.o` file is produced from exactly one source (`.c`) file

■ Executable object file (`a.out` file)

- Contains code and data in a form that can be copied directly into memory and then executed.

■ Shared object file (`.so` file)

- Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
- Called *Dynamic Link Libraries* (DLLs) by Windows

Three Kinds of Object Files under Linux

```
# file sum.o main.o      Relocatable object file (.o file)
sum.o:  ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

```
# file main              Executable object file (a.out file)
main: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=0x34c39011eac6fd0ebae938e4087e788b28a4f6dd, not stripped
```

```
# ldd main
      linux-vdso.so.1 => (0x00007fff9dbfe000)
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4bef587000)
      /lib64/ld-linux-x86-64.so.2 (0x00007f4bef956000)
```

```
# file /lib/x86_64-linux-gnu/libc.so.6
/lib/x86_64-linux-gnu/libc.so.6: symbolic link to `libc-2.15.so'
```

```
# file /lib/x86_64-linux-gnu/libc-2.15.so  Shared object file (.so file)
/lib/x86_64-linux-gnu/libc-2.15.so: ELF 64-bit LSB shared object, x86-64,
version 1 (SYSV), dynamically linked (uses shared libs),
BuildID[sha1]=0x760efc6878e468a84b60e307a5bad802cbe2a480, for GNU/Linux
2.6.24, stripped
```

Object File Format

■ Unix like System

- Early Unix had a simple a.out format until early days of free Linux/BSD
- AT&T's 2nd try was COFF, still limited, but widely adopted with changes
- AT&T's third try was ELF, now used in almost all Unix systems

■ Early DOS and Windows had several limited formats

- Since the 32-bit era, Windows uses the PE (Portable Executable) format, partially derived from COFF
- OS X era Apple (including iOS, etc) uses a format named Mach-O

Although our discussion will focus on ELF, basic concepts are similar.

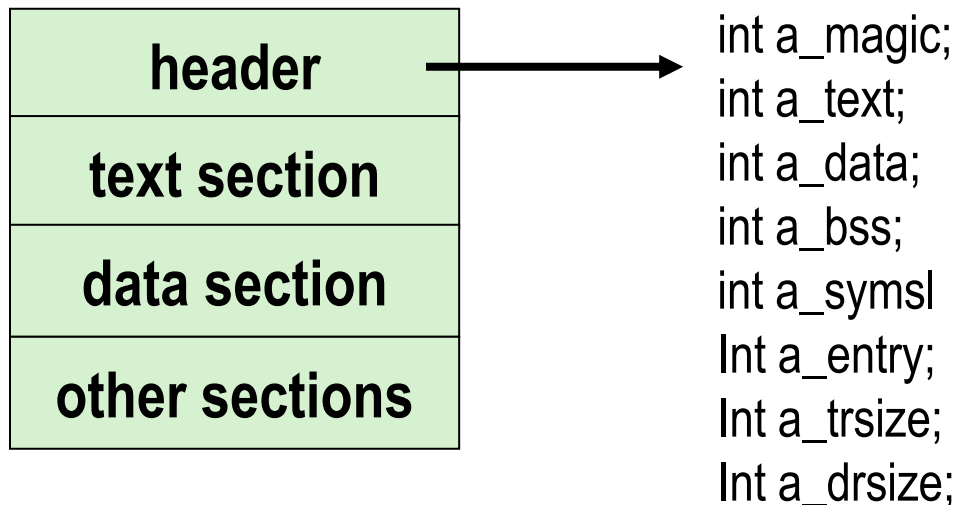
Simplest Object File Format (Cont.)

■ The NULL object format: DOS COM Files

- Only have binary code. Fits in a single segment. No relocation.
- DOS EXE file header has *relocation table*.
 $\text{program_base_address} + \text{segment_number} + \text{offset}$

■ Unix simple a.out format

- Can be linked to start at a fixed address. No relocation needed at load-time.
- Hard to support dynamic linking



Executable and Linkable Format (ELF)

- **Standard binary format for object files**
- **One unified format for**
 - Relocatable object files (`.o`),
 - Executable object files (`a.out`)
 - Shared object files (`.so`)
- **Generic name: ELF binaries**
- **First appeared in System V Release 4 Unix, c. 1989**
- **Linux switched to ELF c. 1995, BSD later at c. 1998-2000**

ELF Object File Format

■ ELF header

- Word size, byte ordering, file type (.o, exec, .so), machine type, etc.

■ Segment header table

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

■ .text section

- Code

■ .rodata section

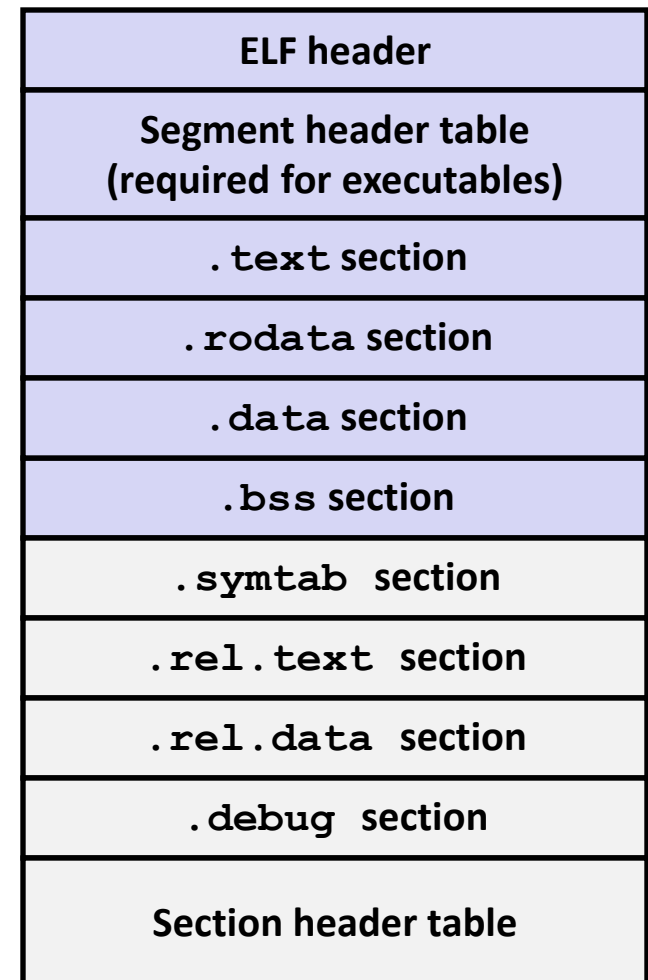
- Read only data: jump tables, string constants, ...

■ .data section

- Initialized global variables

■ .bss section

- Uninitialized global variables
- “Block Started by Symbol”
- “Better Save Space”
- Has section header but occupies no space



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**)
- **Section header table**
 - Offsets and sizes of each section

ELF header
Segment header table (required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.text section
.rel.data section
.debug section
Section header table

0

Parallel Views of a ELF File

- ***Program header table/Segments*** is used to build a process image (execute a program); relocatable files don't need it.
- Files used during linking must have a ***section header table/Sections***.

ELF Header
<i>Program header table optional</i>
Section 1
...
Section n
...
Section header table required

Linking View

ELF Header
Program header table required
Segment 1
Segment 2
Segment 3
...
<i>Section header table optional</i>

Execution View

Linker Symbols

■ Global symbols

- Symbols defined by module m that can be referenced by other modules.
- E.g.: non-**static** C functions and non-**static** global variables.

■ External symbols

- Global symbols that are referenced by module m but defined by some other module.

■ Local symbols

- Symbols that are defined and referenced exclusively by module m .
- E.g.: C functions and global variables defined with the **static** attribute.
- **Local linker symbols are *not* local program variables**

Symbol Table

- Symbol tables are built by assembler, using symbols exported by the compiler into the assembly language `.s` file.
- An ELF symbol table is contained in the `.symbol` section. It contains an array of entries.

source code of `glibc/elf/elf.h`

```
529 typedef struct
530 {
531     Elf64_Word st_name; /* Symbol name (string tbl index) */
532     unsigned char st_info; /* Symbol type and binding */
533     unsigned char st_other; /* Symbol visibility */
534     Elf64_Section st_shndx; /* Section index */
535     Elf64_Addr st_value; /* Symbol value */
536     Elf64_Xword st_size; /* Symbol size */
537 } Elf64_Sym;
```

Element of Symbol Table Structure

■ **st_name(name)**

- Byte offset into the string table, which holds the name of the symbols.

■ **st_value(value)**

- gives the alignment constraints for COMMON symbols.
- In other relocatable files, it holds a section offset for a defined symbol
- In executable and shared object files, st_value holds a virtual address.

■ **st_size(size)**

- A data object's size is the number of bytes contained in the object.
- Holds 0 if the symbol has no size or an unknown size.

■ **st_info (type & binding)**

- st_type is usually data or function.
- Binding field indicates whether the symbol is local, global or weak.

■ **st_shndx(section)**

- holds the relevant section header table index.
- Every symbol table entry is defined in relation to some section.

Pseudo Sections for Section Header Index (*st_shndx*)

- Every symbol table entry is defined in relation to some section. *st_shndx* holds the relevant section header table index.
- Some section indexes indicate special meanings.
- SHN_ABS(*st_shndx* == 0xffff1)
 - Absolute values for the corresponding reference.
 - Symbols are not affected by relocation.
- SHN_UNDEF(*st_shndx* == 0x0)
 - The symbol is undefined. When the link editor combines this object file with another that defines the indicated symbol, this file's references to the symbol will be linked to the actual definition..
- SHN_COMMON(*st_shndx* == 0xffff2)
 - Uninitialized data objects that are not yet allocated. The link editor will allocate space for the symbol at an address that is a multiple of *st_value*.
 - E.g. FORTRAN common block or unallocated C external variables.

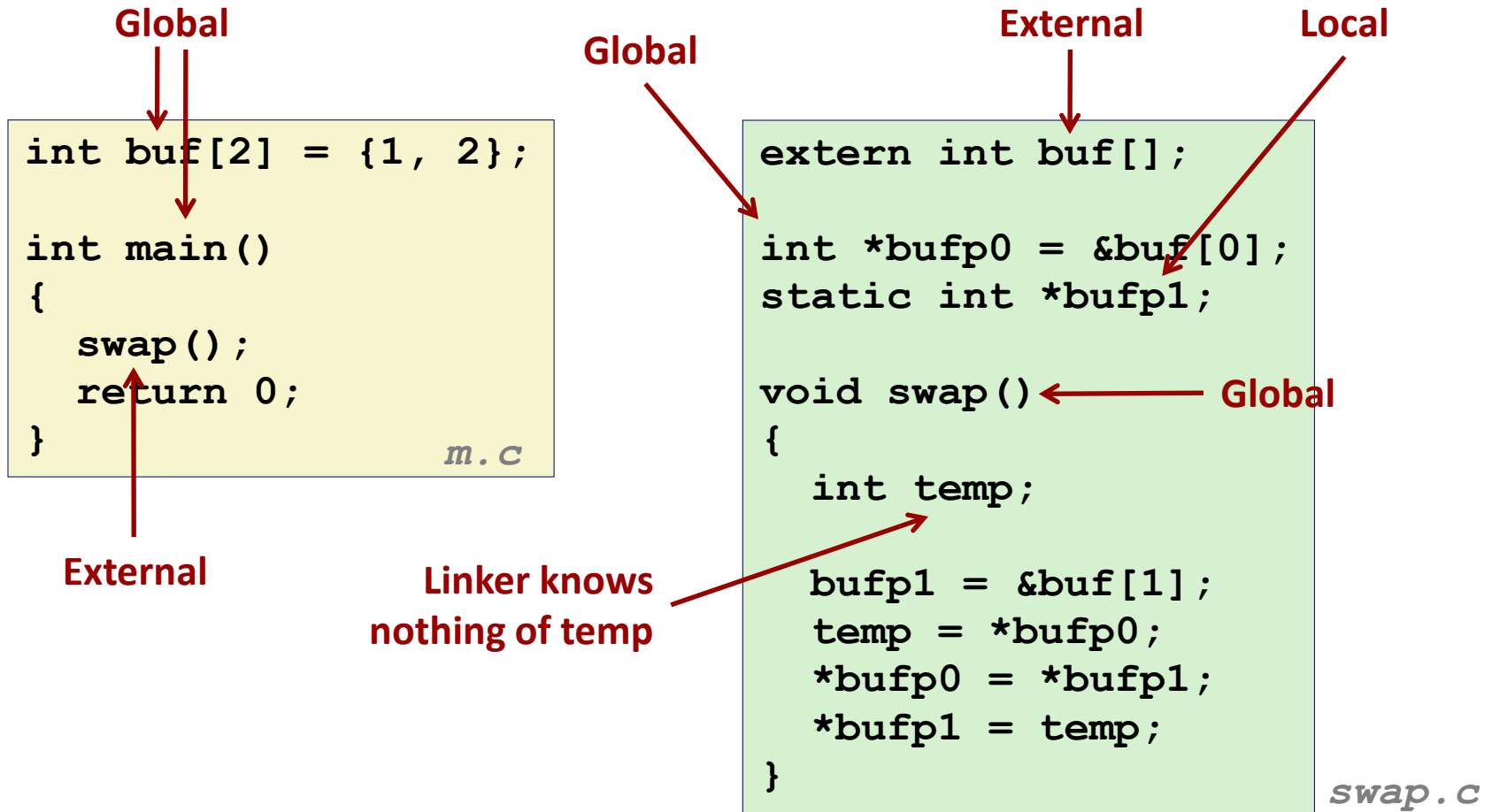
Section *COMMON* vs. Section *.bss*

- Distinction between *COMMON* and *.bss* is subtle.
- Modern versions of GCC assign symbols in relocatable object files to *COMMON* and *.bss* using the following conventions.

	Global Variables	Static Variables
Uninitialized	<i>COMMON</i>	<i>.bss</i>
Initialized to Zero	<i>.bss</i>	<i>.bss</i>
Initialized to Non-Zero	<i>.data</i>	<i>.data</i>

- Variables may be in *COMMON* or *.bss* in relocatable object files, both in *.bss* in executable files.

Symbol Table Entries (Fig. 7-5 in textbook)



Symbol Table Entries (Fig. 7-5 in textbook)

```
# objdump -r -d -t m.o | head -n 15
```

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

```
SYMBOL TABLE:
```

```
0000000000000000 1      df *ABS*  0000000000000000 m.c
0000000000000000 1      d  .text  0000000000000000 .text
0000000000000000 1      d  .data  0000000000000000 .data
0000000000000000 1      d  .bss   0000000000000000 .bss
0000000000000000 1      d  .note.GNU-stack0000000000000000 .note.GNU-stack
0000000000000000 1      d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 1      d  .comment      0000000000000000 .comment
0000000000000000 g      O  .data  0000000000000008 buf
0000000000000000 g      F  .text  0000000000000015 main
0000000000000000      *UND*  0000000000000000 swap
```

```
# readelf -s m.o
```

```
Symbol table '.symtab' contains 11 entries:
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	m.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	7	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
8:	0000000000000000	8	OBJECT	GLOBAL	DEFAULT	3	buf
9:	0000000000000000	21	FUNC	GLOBAL	DEFAULT	1	main
10:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	swap

Symbol Table Entries (Fig. 7-5 in textbook)

```
# objdump -r -d -t swap.o | head -n 15
```

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

```
SYMBOL TABLE:
```

```
0000000000000000 1      df *ABS*  0000000000000000 swap.c
0000000000000000 1      d  .text  0000000000000000 .text
0000000000000000 1      d  .data  0000000000000000 .data
0000000000000000 1      d  .bss   0000000000000000 .bss
0000000000000000 1      d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 1      d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 1      d  .comment 0000000000000000 .comment
0000000000000000 g      O  .data  0000000000000008 bufp0
0000000000000000      *UND* 0000000000000000 buf
0000000000000008      O *COM* 0000000000000008 bufp1
0000000000000000 g      F  .text  000000000000003c swap
```

```
# readelf -s swap.o | tail -n 11
```

1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	swap.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	7	
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	8	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
8:	0000000000000000	8	OBJECT	GLOBAL	DEFAULT	3	bufp0
9:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	buf
10:	0000000000000008	8	OBJECT	GLOBAL	DEFAULT	COM	bufp1
11:	0000000000000000	60	FUNC	GLOBAL	DEFAULT	1	swap

Step 1: Symbol Resolution

...that's defined here

Referencing
a global...

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

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

int main(int argc, char **argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

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

sum.c

Defining
a global

Linker knows
nothing of val

Referencing
a global...

...that's defined here

Linker knows
nothing of i or s

How Linker Resolves Duplicate Symbol Definitions (such as sum, array)?

Symbol Identification

Which of the following names will be in the symbol table of *symbols.o* ?

symbols.c:

```
int incr = 1;
static int foo(int a) {
    int b = a + incr;
    return b;
}

int main(int argc,
          char* argv[]) {
    printf("%d\n", foo(5));
    return 0;
}
```

Names:

- **incr**
- **foo**
- a
- argc
- argv
- b
- **main**
- **printf**
- "%d\n"

Can find this with readelf:

```
linux> readelf -s symbols.o
```

Local Symbols

■ Local non-static C variables vs. local static C variables

- local non-static C variables: stored on the stack
- local static C variables: stored in either `.bss`, or `.data`

```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}

int h() {
    return x += 27;
}
static-local.c
```

Compiler allocates space in `.data` for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x`, `x.1721` and `x.1724`.

Resolving Global Symbols

- When unable to find a definition for the reference symbol in any of its input modules, it prints an error message and terminates.

```
void foo(void) ;  
int main() {  
    foo();  
    return 0;  
}  
linkerror.c
```

```
# gcc -Wall -Og -S linkerror.c  
# as -o linkerror.o linkerror.s
```

Compiler and assembler runs without a hitch.

```
# gcc -Wall -Og -o linkerror linkerror.o  
linkerror.o: In function `main':  
linkerror.c:(.text+0x5): undefined reference to `foo'  
collect2: error: ld returned 1 exit status
```

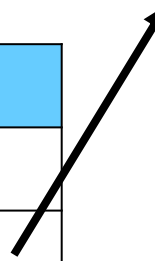
Linker terminates when it cannot resolve the reference to function foo.

Name Mangling

■ Override in C++/Java

Name	Referent
"A"	<class A>
"print"	<overload set>

Signature	Referent
void(int)	<print 1>
void(char)	<print 2>
void(String)	<print 3>



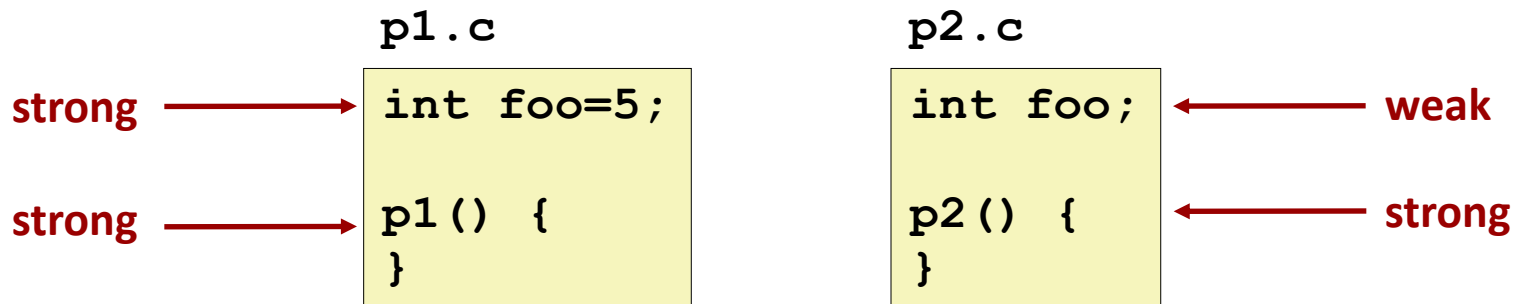
- name mangling encodes the function's signature(argument and return types) into a textual form.

```

void print(int i, float f)
=> "_Z5printf"           (g++)
=> "?print@@YAXHM@Z"    (msvc++)
void print(float f, int i)
=> "_Z5printfi"          (g++)
=> "?print@@YAXMH@Z"     (msvc++)
  
```

How Linker Resolves Duplicate Symbol Names

- Program symbols are either *strong* or *weak*
 - **Strong**: procedures and initialized global variables
 - **Weak**: uninitialized global variables
 - Or ones declared with specifier **extern**
- Compiler exports such kind of information and assembler encodes it implicitly in the symbol table of ELF files.



Linker's Symbol Rules

- **Rule 1: Multiple strong symbols are not allowed**
 - Each item can be defined only once
 - Otherwise: Linker error

- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
 - References to the weak symbol resolve to the strong symbol

- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
 - Can override this with `gcc -fno-common`

Convention between Compiler and Linker

- **Compiler may assign symbols to either COMMON or .bss**
 - Uninitialized global names is assigned to COMMON.
 - -fno-common GCC option force uninitialized global names to be assigned in .bss section.
 - `__attribute__((weak))` force global names to be weak.
- **Sometimes, linker allows multiple modules to define global symbols with the same name.**
 - The link editor honors the global definition and ignores the weak ones.
 - Similarly, the link editor honors the COMMON definition and ignores the weak ones.

What if you mess up?

```
int x=7;
p1() {}
```

```
extern int x;
p2() {}
```

Correct program.
Only one definition of **x**, **p1**, **p2**

```
int x=7;
p1() {}
```

```
int x=0;
p1() {}
```

Link error: two definitions of **x** and **p1**

```
int x;
p1() {}
```

```
int x;
p2() {}
```

Compiler-dependent. Might be considered either one or two definitions of **x**.

```
int x=7;
int y=5;
p1() {}
```

```
extern double x;
p2() {}
```

Undefined behaviour. No link error.
Writes to **x** in **p2** may overwrite **y**!

```
char p1[]
= 0xC3;
```

```
extern void p1();
p2() { p1(); }
```

Undefined behaviour. No link error.
Call to **p1** may crash!

Linker checks for two definitions of one symbol.
Linker *does not* check types of references.

Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (**p1**)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** might overwrite **y**!
Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** might overwrite **y**!
Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same initialized variable.

Important: Linker does not do type checking.

Type Mismatch Example

```
extern long int x;  
  
int main(int argc,  
         char *argv[]) {  
    printf("%ld\n", x);  
    return 0;  
}
```

mismatch-main.c

```
double x = 3.14;
```

mismatch-variable.c

- Compiles without any errors or warnings
- What gets printed?

```
-bash-4.2$ ./mismatch  
4614253070214989087
```

Detecting the Type Mismatch Example

```
extern long int x;  
mismatch.h
```

```
#include "mismatch.h"  
  
int main(int argc,  
         char *argv[]) {  
    printf("%ld\n", x);  
    return 0;  
}  
mismatch-main.c
```

```
#include "mismatch.h"  
  
double x = 3.14;  
  
mismatch-variable.c
```

- Now we get an error ... from the *compiler*, not the linker.

mismatch-variable.c:3:8: conflicting types for 'x'

mismatch.h:1:17: previous declaration of 'x'

Rules for avoiding type mismatches

- Avoid global variables as much as possible
- Use **static** as much as possible
- Declare *everything* that's not **static** in a header file
 - Make sure to include the header file everywhere it's relevant
 - Including the files that define those symbols
- Always put **extern** on declarations in header files
 - Unnecessary but harmless for function declarations
 - Avoids the quirky behavior of extern-less global variables
- Always write **(void)** when a function takes no arguments
 - **extern void no_args(void) ;**
 - Leaving out the **void** means “I’m *not saying* what argument list this function takes.” Turns off argument type checking!

Use of *extern* in .h Files (#1)

c1.c

```
#include "global.h"

int f() {
    return g+1;
}
```

global.h

```
extern int g;
int f();
```

c2.c

```
#include <stdio.h>
#include "global.h"

int g = 0;

int main(int argc, char argv[]) {
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

Use of .h Files (#2)

c1.c

```
#include "global.h"

int f() {
    return g+1;
}
```

global.h

```
extern int g;
static int init = 0;
```

```
#else
    extern int g;
    static int init = 0;
#endif
```

c2.c

```
#define INITIALIZE
#include <stdio.h>
#include "global.h"

int main(int argc, char** argv) {
    if (init)
        // do something, e.g., g=31;
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

```
int g = 23;
static int init = 1;
```


Linking Example

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

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

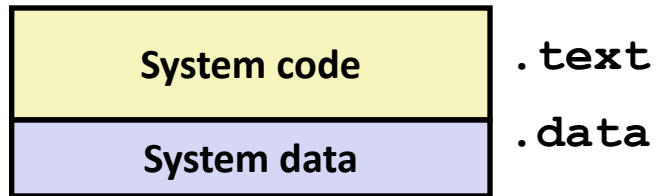
int main(int argc, char **argv)
{
    int val = sum(array, 2);
    return val;
}                                     main.c
```

```
int sum(int *a, int n)
{
    int i, s = 0;

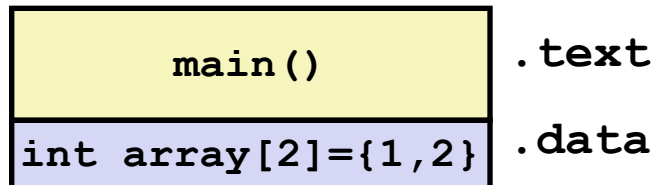
    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}                                     sum.c
```

Step 2: Relocation

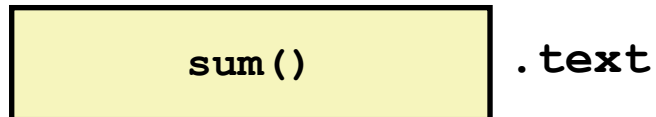
Relocatable Object Files



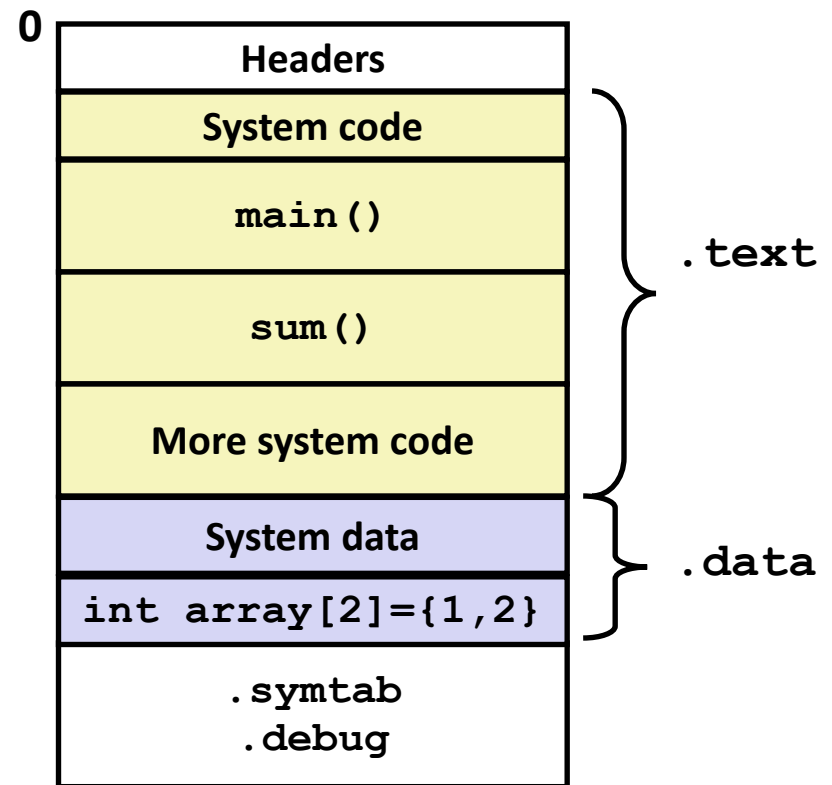
main.o



sum.o



Executable Object File



2-Step Relocation in Static Linking

■ Relocating sections and symbol definitions

- Merges all sections of the same type into a new aggregate section of the same type.
- Assigns run-time memory addresses to
 - The new aggregate section.
 - Each section defined by the input modules.
 - Each symbol defined by the input modules.

■ Relocating symbol references with sections

- Modifies every symbol reference in the bodies of the code and data sections so that they point to the correct run-time addresses.
- It relies on data structures in the relocatable modules known as **relocation entries**.

Relocation Entries

- A *relocation entry* generates from reference with unknown location.

```
/* Relocation table entry with addend
(in section of type SHT_RELA). */
```

```
660 typedef struct
661 {
662     Elf64_Addr r_offset; /* Address */
663     Elf64_XWord r_info; /* Relocation type and symbol index */
664     Elf64_Sxword r_addend; /* Addend */
665 } Elf64_Rela;
673 #define ELF64_R_SYM(i) ((i) >> 32)
674 #define ELF64_R_TYPE(i) ((i) & 0xfffffffff)
```

- *r_offset* is the section offset of the reference that will be modified.
- *ELF_64_R_SYM* identifies the symbol that the reference should point to.
- *ELF_64_R_TYPE* tells the linker how to modify the new reference.
- *r_addend* is a constant used for offset adjustment in some kind of relocation.

Two Most Basic Relocation Types

■ R_X86_64_PC32

- Relocates a reference that uses a 32-bit PC-relative address.

■ R_X86_64_32/R_X86_64_32S

- Relocates a reference that uses a 32-bit absolute address.

```

for each section s {
  foreach relocation entry r {
    refptr = s + r.offset;  /* ptr to reference to be relocated */

    /* Relocate a PC-relative reference */
    if (r.type == R_X86_64_PC32) {
      refaddr = ADDR(s) + r.offset; /* ref's run-time address */
      *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
    }

    /* Relocate an absolute reference */
    if (r.type == R_X86_64_32)
      *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
  }
}

```

Relocation Entries

```
int array[2] = {1, 2};
int main(int argc, char** argv){
    int val = sum(array, 2);
    return val;
}
main.c
```

readelf -r main.o

Relocation section '.rela.text' at offset 0x560 contains 2 entries:

Offset	Info	Type	Sym.Value	Sym.Name+Addend
00000000000015	000800000000a	R_X86_64_32	0000000000000000	array + 0
0000000000001a	000a000000002	R_X86_64_PC32	0000000000000000	sum - 4

Relocation section '.rela.eh_frame' at offset 0x590 contains 1 entries:

Offset	Info	Type	Sym.Value	Sym.Name+Addend
00000000000020	0002000000002	R_X86_64_PC32	0000000000000000	.text + 0

offset

type

symbol name & addend

Totally 3 symbols to be relocated.

Relocation Entries (in main.o)

```
int array[2] = {1, 2};
int main(int argc, char** argv){
    int val = sum(array, 2);
    return val;
}
main.c
```

```
# readelf -r main.o
```

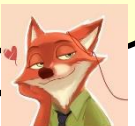
Relocation section '.rel.text' at offset 0x560 contains 2 entries:

Offset	Info	Type	Sym.Value	Sym.Name+Addend
00000000000015	000800000000a	R_X86_64_32	0000000000000000	array + 0
0000000000001a	000a000000002	R_X86_64_PC32	0000000000000000	sum - 4

Dear Linker,

Please patch the .rel.text section at offsets 0x15. Patch in a 32-bit value like following steps. When you determine the addr of .data, compute [addr of array] + [addend, which equals 0] and place the result at the prescribed place.

Sincerely,
Assembler



Relocation Entries (in main.o)

```
int array[2] = {1, 2};
int main(int argc, char** argv){
    int val = sum(array, 2);
    return val;
}
main.c
```

```
# readelf -r main.o
```

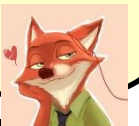
```
Relocation section '.rel.text' at offset 0x560 contains 2 entries:
```

Offset	Info	Type	Sym.Value	Sym.Name+Addend
00000000000015	000800000000a	R_X86_64_32	0000000000000000	array + 0
0000000000001a	000a000000002	R_X86_64_PC32	0000000000000000	sum - 4

Dear Linker,

Please patch the .rel.text section at offsets 0x1a. Patch in a 32-bit “PC-relative” value like following steps. When you determine the addr of sum, compute [addr of sum] + [addend, which equals -4] – [addr of section + offset] and place the result at the prescribed place.

Sincerely,
Assembler



Relocation Entries (in sum.o)

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

sum.c

# readelf -r sum.o			
Relocation section '.rela' at offset 0x4f8 contains 1 entries:			
Offset	Info	Type	Sym.Value
00000000000020	0002000000002	R_X86_64_PC32	0000000000000000
		Sym.Name+Addend	
		.text + 0	
offset		type	
		symbol name & addend	

1 symbol to be relocated (.text)

Original Object File of main.o

```
int array[2] = {1, 2};
int main(int argc, char** argv){
    int val = sum(array, 2);
    return val;
}
main.c
```

```
000000000000000000 <main>:
0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: 48 83 ec 20       sub     $0x20,%rsp
8: 89 7d ec          mov     %edi,-0x14(%rbp)
b: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
f: be 02 00 00 00   mov     $0x2,%esi
14: bf 00 00 00 00   mov     $0x0,%edi      # %edi = &array
15: R_X86_64_32 array # Relocation entry
19: e8 00 00 00 00   callq  1e <main+0x1e> # sum()
1a: R_X86_64_PC32 sum-0x4 # Relocation entry
1e: 89 45 fc          mov     %eax,-0x4(%rbp)
21: 8b 45 fc          mov     -0x4(%rbp),%eax
24: c9               leaveq
25: c3               retq
```

Source: objdump -r -d main.o

Original Object File of sum.o

```
int sum(int *a, int n){
```

```
000000000000000000 <sum>:
```

```

0:      55                                push    %rbp
1:      48 89 e5                          mov     %rsp,%rbp
4:      48 89 7d e8                       mov     %rdi,-0x18(%rbp)
8:      89 75 e4                          mov     %esi,-0x1c(%rbp)
b:      c7 45 fc 00 00 00 00             movl    $0x0,-0x4(%rbp)
12:     c7 45 f8 00 00 00 00             movl    $0x0,-0x8(%rbp)
19:     eb 1d                            jmp     38 <sum+0x38>
1b:     8b 45 f8                          mov     -0x8(%rbp),%eax
1e:     48 98                            cltq
20:     48 8d 14 85 00 00 00             lea     0x0(,%rax,4),%rdx
27:     00
28:     48 8b 45 e8                       mov     -0x18(%rbp),%rax
2c:     48 01 d0                          add     %rdx,%rax
2f:     8b 00                            mov     (%rax),%eax
31:     01 45 fc                          add     %eax,-0x4(%rbp)
34:     83 45 f8 01                       addl    $0x1,-0x8(%rbp)
38:     8b 45 f8                          mov     -0x8(%rbp),%eax
3b:     3b 45 e4                          cmp     -0x1c(%rbp),%eax
3e:     7c db                            jl      1b <sum+0x1b>
40:     8b 45 fc                          mov     -0x4(%rbp),%eax
43:     5d                                pop     %rbp
44:     c3                                retq

```

A Little Bit of Preparation for Linking

- Linking script can be used to configure your linking process.
- Start address of sections could be specified.

- .text starts at 0xbabe00
- .data starts at 0xcafe00

- Using:

gcc a.lds -o m main.o sum.o

SECTIONS

```
{  
    .text 0x00BABE00:  
    {  
        *(.text)  
    }  
    . = ALIGN(0);  
    .data 0x00CAFE00:  
    {  
        *(.data)  
    }  
}
```

a.lds

`.text=0xbabe00``000000000000babe00 <_start>:``00000000000000000 <main>:`

```

0: 55
1: 48 89 e5
4: 48 83 ec 20
8: 89 7d ec
b: 48 89 75 e0
f: be 02 00 00 00
14: bf 00 00 00 00
19: e8 00 00 00 00
1e: 89 45 fc
21: 8b 45 fc
24: c9
25: c3

```

`main.o``00000000000000000 <sum>:`

```

0: 55
1: 48 89 e5
4: 48 89 7d e8
8: 89 75 e4
b: c7 45 fc 00 00 00 00
12: c7 45 f8 00 00 00 00
19: eb 1d
1b: 8b 45 f8
1e: 48 98
20: 48 8d 14 85 00 00 00
27: 00
28: 48 8b 45 e8
2c: 48 01 d0
2f: 8b 00
31: 01 45 fc
34: 83 45 f8 01
38: 8b 45 f8
3b: 3b 45 e4
3e: 7c db
40: 8b 45 fc
43: 5d
44: c3

```

`sum.o``...
0000000000babf18 <main>:`

```

babf18: 55
babf19: 48 89 e5
babf1c: 48 83 ec 20
babf20: 89 7d ec
babf23: 48 89 75 e0
babf27: be 02 00 00 00
babf2c: bf 10 fe fa 00
babf31: e8 0a 00 00 00
babf36: 89 45 fc
babf39: 8b 45 fc
babf3c: c9
babf3d: c3

```

`0000000000babf40 <sum>:`

```

babf40: 55
babf41: 48 89 e5
babf44: 48 89 7d e8
babf48: 89 75 e4
babf4b: c7 45 fc 00 00 00 00
babf52: c7 45 f8 00 00 00 00
babf59: eb 1d
babf5b: 8b 45 f8
babf5e: 48 98
babf60: 48 8d 14 85 00 00 00
babf67: 00
babf68: 48 8b 45 e8
babf6c: 48 01 d0
babf6f: 8b 00
babf71: 01 45 fc
babf74: 83 45 f8 01
babf78: 8b 45 f8
babf7b: 3b 45 e4
babf7e: 7c db
babf80: 8b 45 fc
babf83: 5d
babf84: c3

```

`executable`

Disassembly of section .data:
 0000000000cafe00 <__data_start>:

```

...
0000000000cafe10 <array>:
  cafe10: 00 00
  cafe12: 00 00
  cafe14: 02 00

```

`.data=0xcafe00`

`addr_of_array=0xcafe10`
 Using the value
 0xcafe10 to modify the
 content here

`addr_of_main=0xbabf18``addr_of_sum=0xbabf40``offset = 0x1a``addend = -4``refptr``= 0xbabf18 + 0x1a``= 0xbabf32``*refptr(content)``=0xbabf40-4-0xbabf32``=0x0a`

```
00000000000000000000 <_start>:
```

```
...
```

```
00000000000000000000 <main>:
```

```
babf18: 55
babf19: 48 89 e5
babf1c: 48 83 ec 20
babf20: 89 7d ec
babf23: 48 89 75 e0
babf27: be 02 00 00 00
babf2c: bf 10 fe ca 00
babf31: e8 0a 00 00 00
babf36: 89 45 fc
babf39: 8b 45 fc
babf3c: c9
babf3d: c3
```

```
00000000000000000000 <sum>:
```

```
babf40: 55
babf41: 48 89 e5
babf44: 48 89 7d e8
babf48: 89 75 e4
babf4b: c7 45 fc 00 00 00 00
babf52: c7 45 f8 00 00 00 00
babf59: eb 1d
babf5b: 8b 45 f8
babf5e: 48 98
babf60: 48 8d 14 85 00 00 00
babf67: 00
babf68: 48 8b 45 e8
babf6c: 48 01 d0
babf6f: 8b 00
babf71: 01 45 fc
babf74: 83 45 f8 01
babf78: 8b 45 f8
babf7b: 3b 45 e4
babf7e: 7c db
babf80: 8b 45 fc
babf83: 5d
babf84: c3
```

It differs with
linking order

```
00000000000000000000 <_start>:
```

```
...
```

```
00000000000000000000 <sum>:
```

```
babf18: 55
babf19: 48 89 e5
babf1c: 48 89 7d e8
babf20: 89 75 e4
babf23: c7 45 fc 00 00 00 00
babf2a: c7 45 f8 00 00 00 00
babf31: eb 1d
babf33: 8b 45 f8
babf36: 48 98
babf38: 48 8d 14 85 00 00 00
babf3f: 00
babf40: 48 8b 45 e8
babf44: 48 01 d0
babf47: 8b 00
babf49: 01 45 fc
babf4c: 83 45 f8 01
babf50: 8b 45 f8
babf53: 3b 45 e4
babf56: 7c db
babf58: 8b 45 fc
babf5b: 5d
babf5c: c3
babf5d: 00 00
```

```
00000000000000000000 <main>:
```

```
babf60: 55
babf61: 48 89 e5
babf64: 48 83 ec 20
babf68: 89 7d ec
babf6b: 48 89 75 e0
babf6f: be 02 00 00 00
babf74: bf 10 fe ca 00
babf79: e8 9a ff ff ff
babf7e: 89 45 fc
babf81: 8b 45 fc
babf84: c9
```

Recall: Parallel Views of a ELF File

- ***Program header table/Segments*** is used to build a process image (execute a program); relocatable files don't need it.
- Files used during linking must have a ***section header table/Sections***; other object files may or may not have one.

ELF Header
<i>Program header table optional</i>
Section 1
...
Section n
...
Section header table required

Linking View

ELF Header
Program header table required
Segment 1
Segment 2
Segment 3
...
<i>Section header table optional</i>

Execution View

Program Headers

- ELF executables are easy to load into memory, with contiguous chunks mapped to contiguous memory segments. It is described by the program header table.

# objdump -x m					
.....					
LOAD	off 0x000000000001abe00	vaddr 0x000000000000babe00	paddr 0x00000000000babe00	align 2**21	
	filesz 0x000000000000032c	memsz 0x000000000000032c	flags r-x		
LOAD	off 0x000000000002afe00	vaddr 0x000000000000cafe00	paddr 0x00000000000cafe00	align 2**21	
	filesz 0x0000000000000018	memsz 0x0000000000000020	flags rw-		
Off: offset in object file vaddr/paddr: memory address align: alignment requirement					
Filesz: segment size in obj memsz: segment size in mem flags: runtime permissions					

- .text starts from 0x00babe00 and .data starts from 0x00cafe00 as we specified before in the link script.
- According to the flags, .text is executable and .data is writable.

Loading Executable Object Files

Executable Object File

0	ELF header
	Program header table (required for executables)
	.init section
	.text section
	.rodata section
	.data section
	.bss section
	.symtab
	.debug
	.line
	.strtab
	Section header table (required for relocatables)

