Computer Architecture
CSCI 4350

Linking

Kwangsung Oh

kwangsungoh@unomaha.edu

http://faculty.ist.unomaha.edu/kwangsungoh

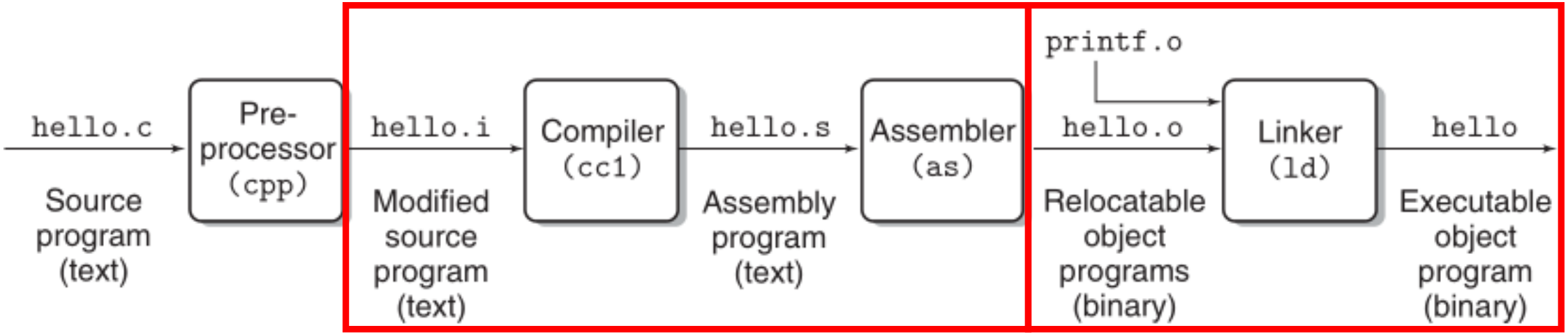
University of Nebraska Omaha

Compilation Process

code/intro/hello.c

```
1   #include <stdio.h>
2
3   int main()
4   {
5       printf("hello, world\n");
6   }
```

code/intro/hello.c



A Single Large Source Code

```
int printf(…) {
        thousands lines of code
}

int scanf (…) {}
…


int main() {
        int a;
        …
        printf(…)
}
```

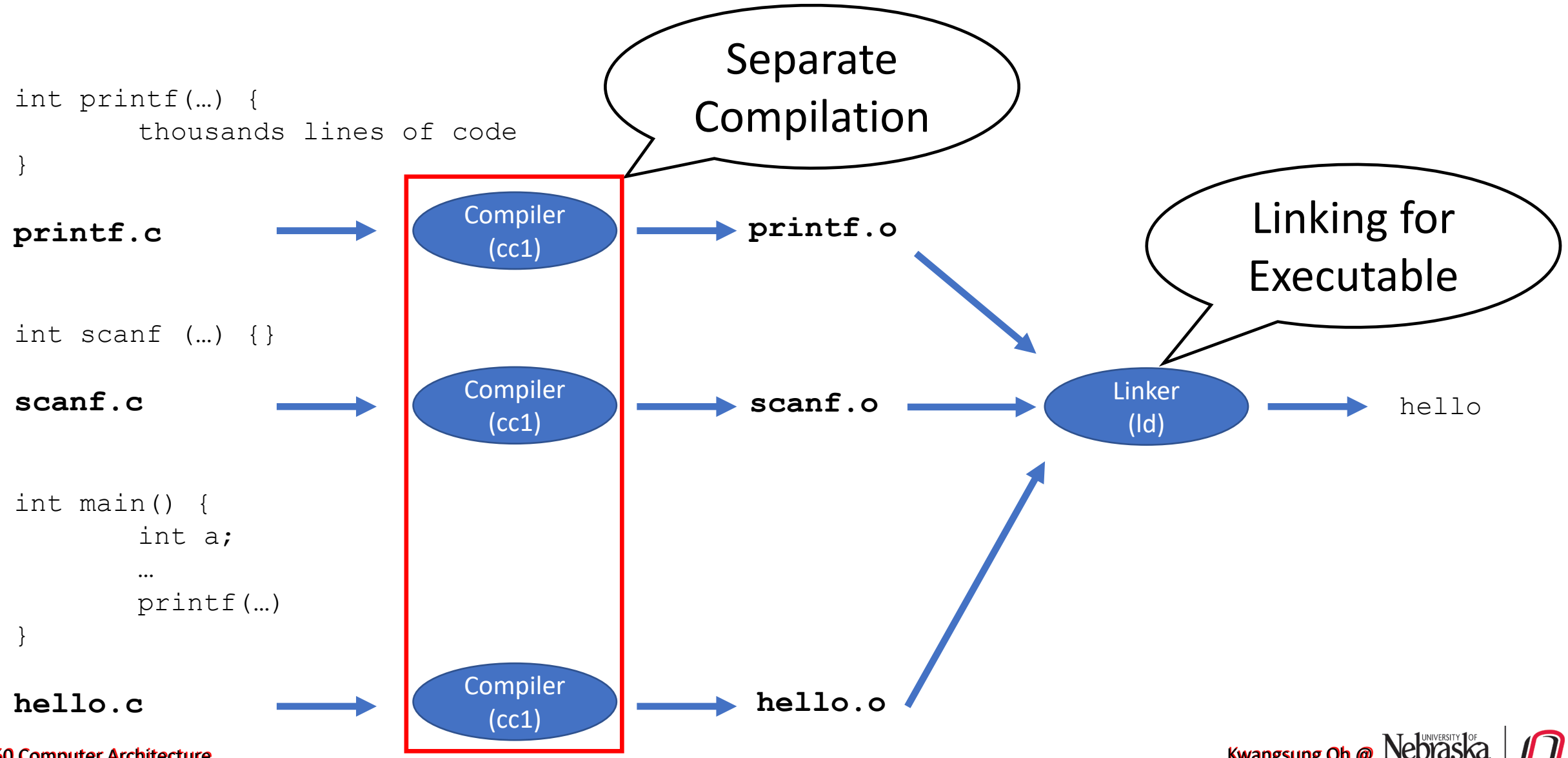
Problems

- Efficient – Code changes
- Modularity – Common functions

Solution

- Separate compilation and linker

A Better Approach



Complier Driver

- Coordinating all steps (compiling and linking)
  - E.g., GCC (GNU Compiler Collection)
  - Preprocessor (cpp), compiler (cc1), assembler (as), and linker (ld)
  - Passing various options for each phases
- Example
  - Generating executable hello from print.c, scan.c, and hello.c
  - **gcc – O2 –v –o hello print.c scan.c hello.c**
    1. cpp [args] print.c /tmp/cca07630.i
    2. cc /tmp/cca07630.i –O2 [args] –S –o /tmp/cca07630.s
    3. as [args] –o /tmp/cca076301.o /tmp/cca07630.s <Similar process for rests>
    4. ld –o hello [system obj files] /tmp/cca076301.o /tmp/cca076302.o /tmp cca076303.o

Linker

- Linking
  - Merging various pieces of code and data (*.o) into a single executable file that can be loaded into memory and executed
- When?
  - *Compile time*: when codes are compiled - static
  - *Load time*: when the program is loaded - dynamic
  - *Run time*: when the program is running - dynamic

Linker Tasks

- Resolving External Reference
  - External reference – symbol in another object file
  - Symbol resolution – exact one symbol in an executable
- Relocating symbols
  - Relative location (in *.o) -> absolute location (in *.exe)
  - Update references based on new symbol locations
  - E.g., a(); // reference to symbol a
  - int * p = &x; // reference to symbol x

Linker Benefits

- Modularity
  - Merging small pieces
  - Libraries of common functions e.g., stdlib, stdio …
- Time efficiency
  - No need to compile all source files
  - Compiling only source file changed and relink
- Space efficiency
  - Common functions in libraries
  - Containing only functions that they use

Static Linker

- Linking at compile time
  - Collecting all relocatable object files (*.o)
  - Using a collection of relocatable object files and command line arguments
- Generating executable file (*.exe)
- Tasks
  - Symbol resolution – exact one symbol
  - Relocation – relocate text and data

Object Files

- Relocatable object file (*.o)
  - Containing binary code and data that can be combined with other object files - **compilers** and **assemblers**
- Executable object file (*.exe)
  - Containing binary code and data that can be loaded into memory and executed - **linker**
- Shared object files (*.so, *.dll)
  - Containing binary code and data that can be linked at either *load* or *run* time

Windows applications on Linux?

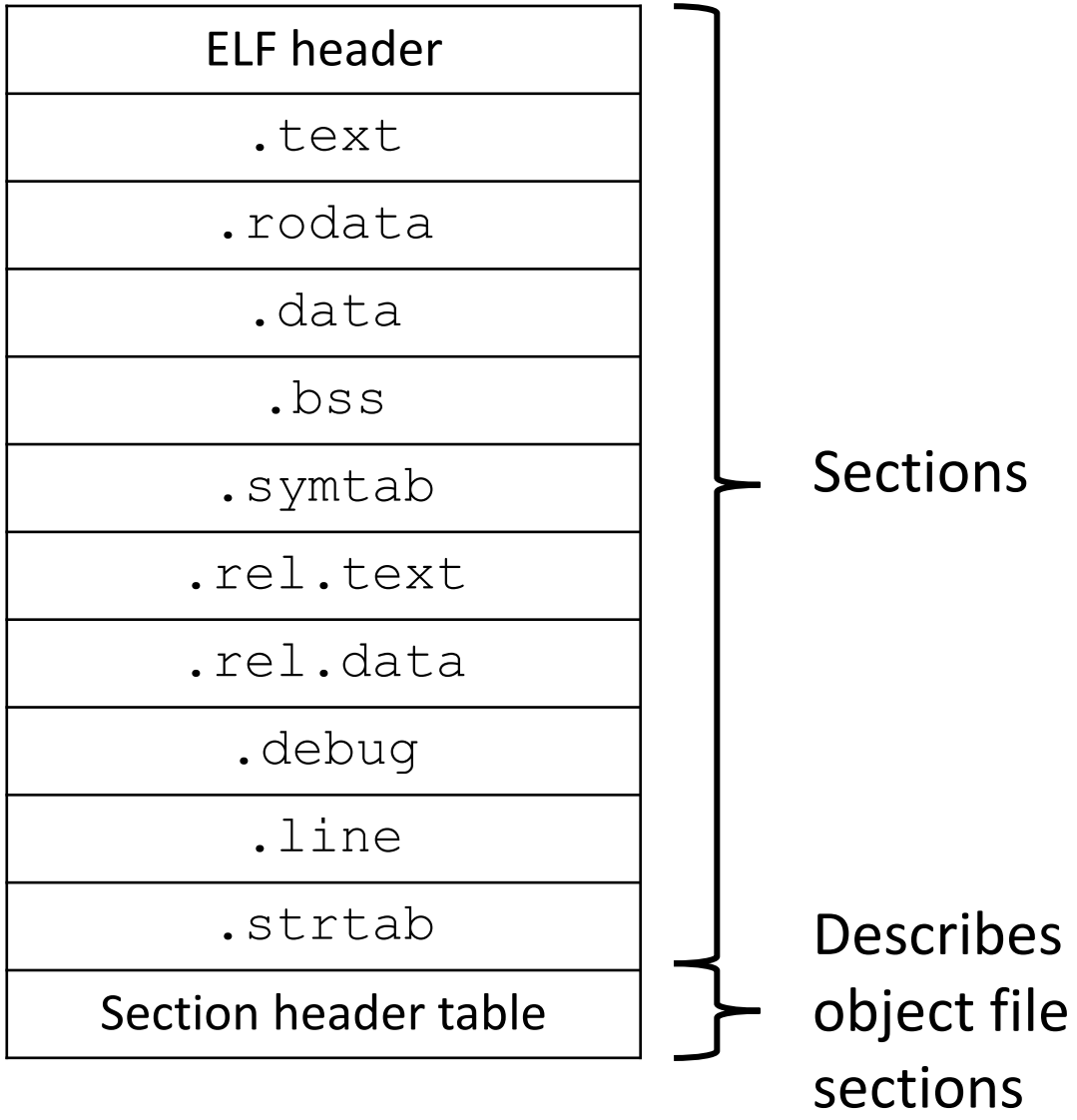
- Linux and Windows – x86-64
- Different **object file formats**
  - a.out – early Unix systems
  - COFF (Common Object File Format) – early system V
  - PE (Portable Executable) – Windows systems
  - ELF (Executable and Linkable) – Linux systems

ELF (Executable and Linkable Format)

- **Standard** binary format
  - Derives from Unix
  - Later adopted by BSD Unix variants and Linux
- One unified format for all object files
  - *.o, *.so, and executable files
- Better support for shared libraries than old a.out

Executable and Linkable Format

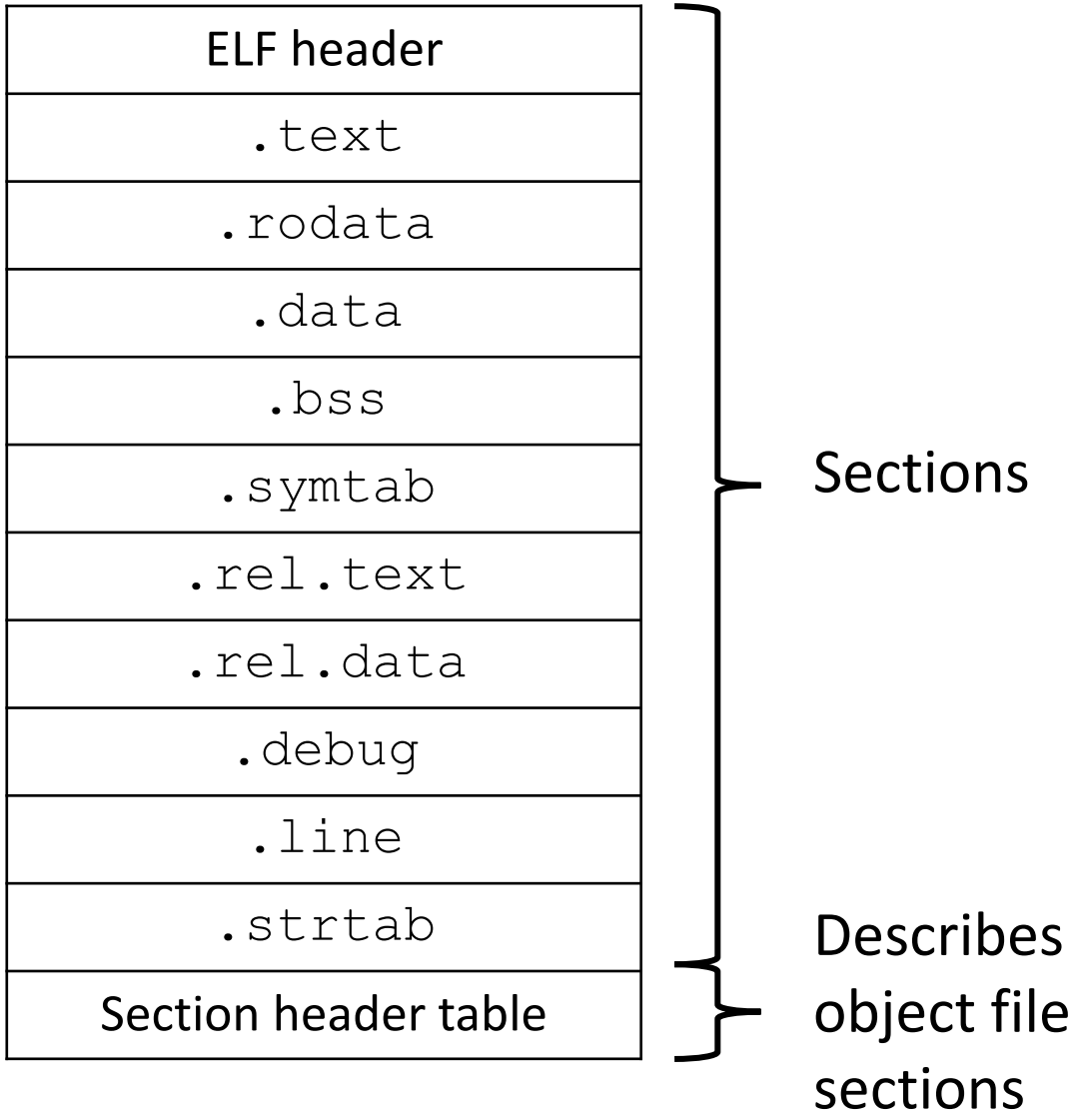
- ELF header
  - Word size
  - Byte ordering
  - Machine type (e.g., IA32)
  - ELF header size
  - Object file type
    relocatable, executable, or shared
  - Offset of the section header
  - Number of entries
- Section header table
  - Section locations and sizes
  - A fixed size entry for each



Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.

Executable and Linkable Format

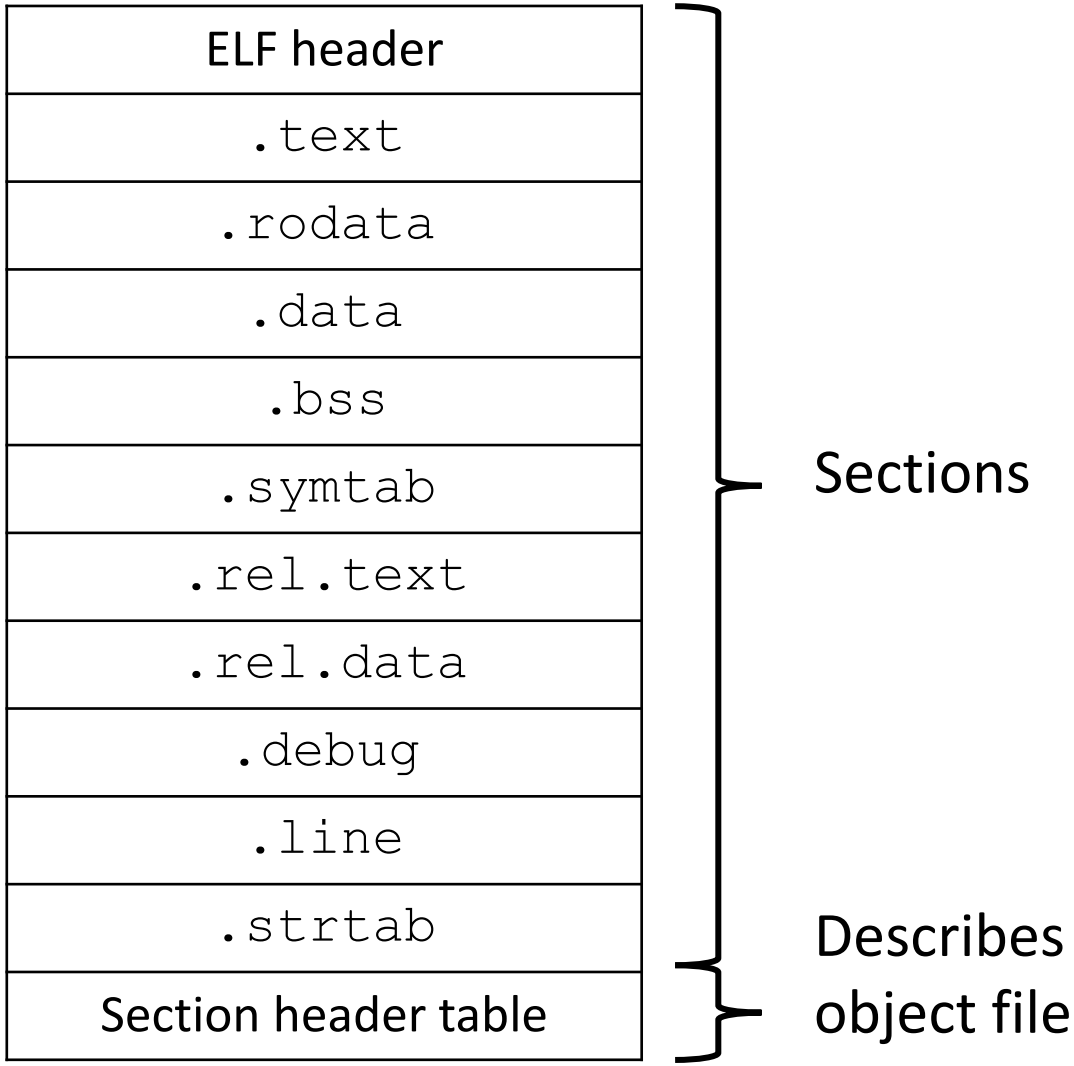
- .text section
  - Machine instructions
- .data section
  - Initialized global variables
  - Static local variables
- .bss (Block storage start) section
  - Uninitialized global variables
- .symtab
  - Symbol table
  - Functions and global variables



Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.

ELF Format

- .rel.text section
  - Relocation info for .text section
- .rel.data section
  - Relocation info for .data section
- .debug section (gcc –g)
  - Symbolic debugging information
- .line section
  - Mapping line # in source code and machine instruction in the .text section
- .strtab section
  - String table for .symtab and .debug



Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.

Life and Scope

- Life
  - If object (variable) is *still in memory* of the process?
- Scope
  - If the object (variable) is *visible (accessible) at current position*?
- Possible – an object is live but not visible, when?
- Impossible – an object is visible but not live

Local Variables' Life and Scope

- Variables defined in functions
- Stored in dynamic memory area (***Stack***)
- Life
  - Began when a function is called
  - Removed when a function is completed
- Scope
  - Visible only within a function where it is defined
  - What if ***static***?

```
void A () {
        static int i = 0;
}
```

Global Variables' Life and Scope

- Variables defined outside of functions
- Stored in static memory area (***data***)
- Life
  - Began when a program starts
  - Removed when a program completes
- Scope
  - Visible throughout the entire program
  - What if ***static***?

```
static int i = 0;
void A () {
}
```

Symbol Types

- *Global symbols*
  - Accessible from other modules
  - *Non-static* functions and variables
- *External global symbols*
- *Local symbols*
  - ***Static*** C functions and ***static*** variables
  - Local ***static*** variables
  - Local linker symbols ≠ local program variables

ELF Symbol Table

```
typedef struct {
      int name;          /* string table offset */
      int value;         /* section offset, or VM address */
      int size;          /* object size in bytes */
      char type:4,       /* data, func, section, or src file name (4 bits) */
           binding:4;    /* local or global (4 bits) */
      char reserved;     /* unused */
      char section;      /* section header index, ABS, UNDEF, */
                         /* or COMMON */
} Elf_Symbol;
```


Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.

Example Program

```
/* main.c */
void swap();

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

int main() {
        swap();
        return 0;
}
```

```
/* swap.c */
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap(){
        int temp;
        bufp1 = &buf[1];
        temp = *bufp0;
        *bufp0 = *bufp1;
        *bufp1 = temp;
}
```

ELF Symbol Examples

```
/* main.c */
void swap();

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

int main() {
        swap();
        return 0;
}
```

Num:	Value	Size	Type	Bind	Ndx	Name
8:	0	8	OBJECT	GLOBAL	3	buf
9:	0	21	FUNC	GLOBAL	1	main
10:	0	0	NOTYPE	GLOBAL	UND	swap

```
/* swap.c */
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap(){
        int temp;
        bufp1 = &buf[1];
        temp = *bufp0;
        *bufp0 = *bufp1;
        *bufp1 = temp;
}
```

Num:	Value	Size	Type	Bind	Ndx	Name
8:	0	4	OBJECT	GLOBAL	3	bufp0
9:	0	0	NOTYPE	GLOBAL	UND	buf
10:	8	4	OBJECT	GLOBAL	COM	bufp1
11:	0	59	FUNC	GLOBAL	1	swap

Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.

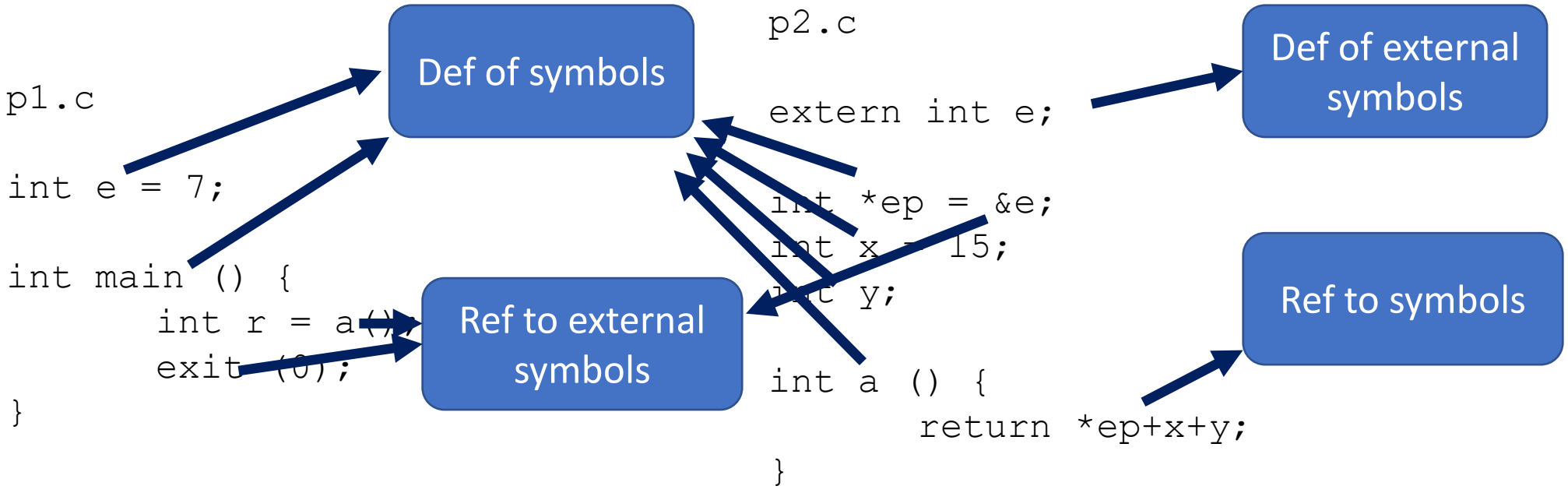
Additional Example

```
<swap.c>
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

static void incr () {
    static int count=0;
    int i=0;
    count++;
}

void swap () {
    int temp;

    incr();
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

buf:	external global, undefined	
bufp0:	global,	.data
bufp1:	local,	.bss
count:	local,	.data
incr:	local,	.text
swap:	global,	.text

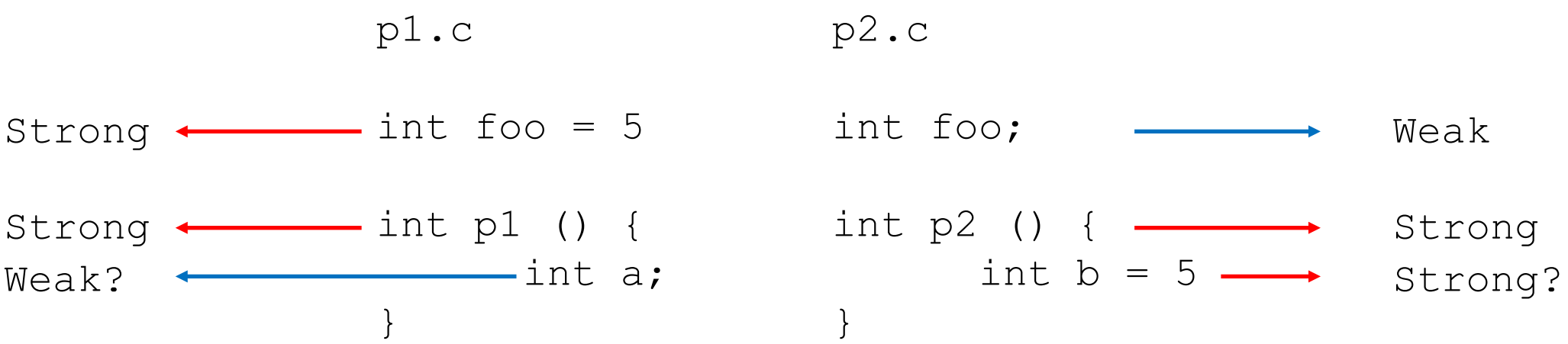
Resolving External References

- ***Symbols*** are lexical entities (functions and variables)
- Each symbol has a ***value*** (typically a memory address)
- Code consists of symbol ***definitions*** and ***references***



Strong and Weak Symbols

- Symbols are either *strong* and *weak*
  - Strong – Functions and initialized global variables
  - Weak – uninitialized global variables
  - Complier exports global symbols to the assembler (weak or strong)
  - Assembler encode symbols information in .symtab



Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.

Symbol Resolution

- **Rule 1** – Multiple strong symbols are not allowed
- **Rule 2** – Given a strong symbol and multiple weak symbols, choose the strong symbol
- **Rule 3** – Given multiple weak symbols, choose any of the weak symbols

Linker Symbol Examples

p1.c	p2.c	Symbol Resolution
`int x;` `p1() {}`	`p1 () {}`	Link time error. There are two strong symbols.
`int x;` `p1() {}`	`int x;` `p2() {}`	Two weak symbols. Is this what you really want?

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		Two weak symbols. If x in p1.c is chosen, y may be overwritten.

`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		y will be overwritten.

`int x=7;` `p1() {}`	`int x;` `p2() {}`	x in p2.c will refer x in p1.c.

Relocation

- After the symbol resolution
  - Linker associates each symbol reference in the code with exactly one symbol definition
  - Linker knows the exact sizes of the code and data sections

1. Relocating sections and symbol definitions
   - Merging all sections of the same type
   - Assigning run-time memory address
2. Relocating symbol reference within sections
   - Update External references to point to the correct address

Relocation Example

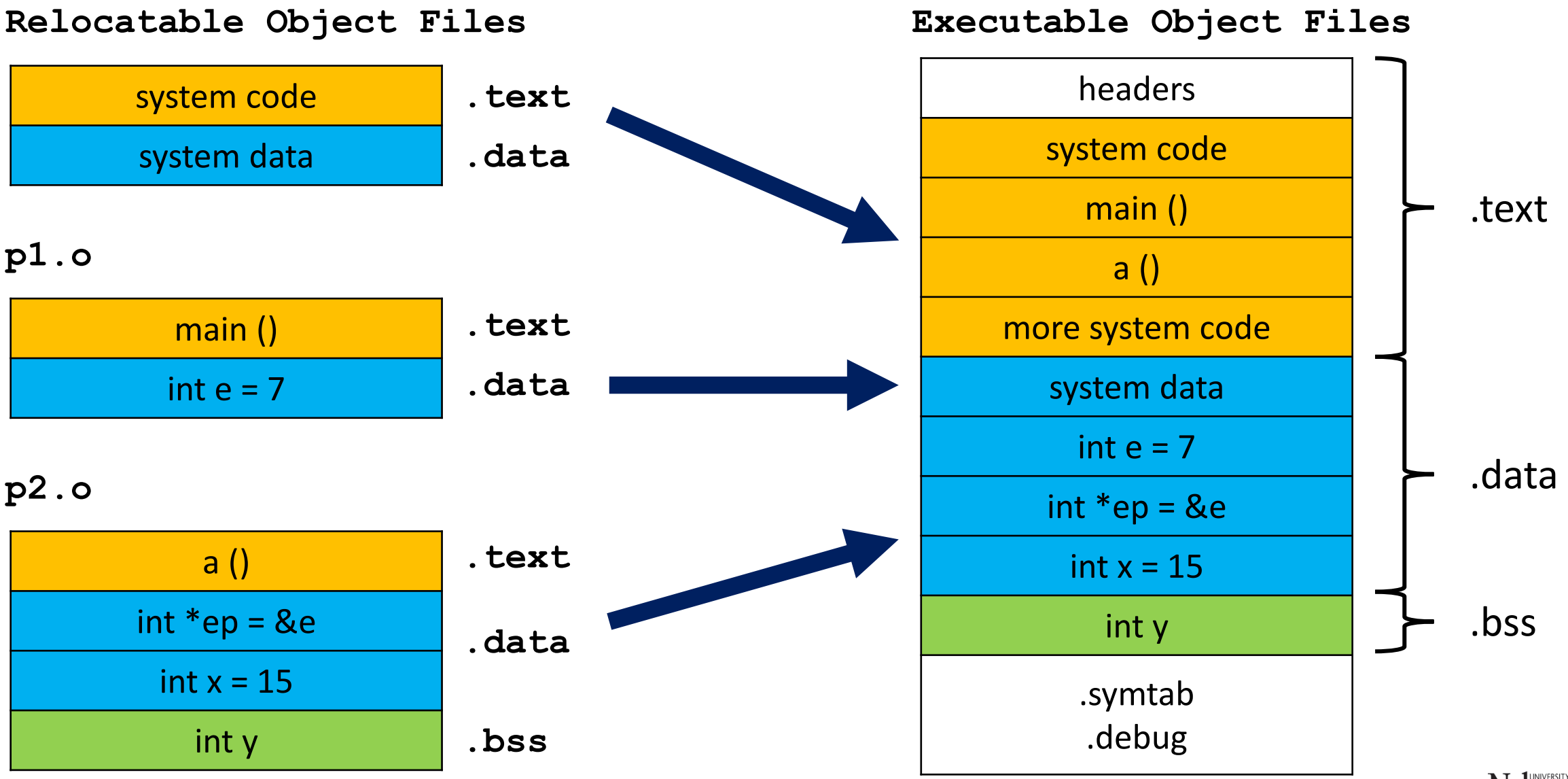
p1.c

```
int e = 7;

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

p2.c

```
extern int e;

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

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

Relocating Sections



Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.

Relocation Info (x86-64)

p1.c

```
int e = 7;

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

Disassembly of section .text:

0000000000000000 <main>:

0:	55	push	%rbp
1:	48 89 e5	mov	%rsp,%rbp
4:	48 83 ec 10	sub	$0x10,%rsp
8:	b8 00 00 00 00	mov	$0x0,%eax
d:	e8 00 00 00 00	callq	12 <main+0x12>
12:	89 45 fc	mov	%eax,-0x4(%rbp)
15:	bf 00 00 00 00	mov	$0x0,%edi
1a:	e8 00 00 00 00	callq	1f <main+0x1f>

Disassembly of section .data:

0000000000000000 <e>:

0: 07 00 00 00

Relocation Info (x86-64)

p2.c

```
extern int e;

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

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

objdump

Disassembly of **section .text**:

0000000000000000 <a>:

```
   0:  55                      push   %rbp
   1:  48 89 e5                mov    %rsp,%rbp
   4:  48 8b 05 00 00 00 00    mov    0x0(%rip),%rax
   b:  8b 10                   mov    (%rax),%edx
   d:  8b 05 00 00 00 00       mov    0x0(%rip),%eax
  13:  01 c2                   add    %eax,%edx
  15:  8b 05 00 00 00 00       mov    0x0(%rip),%eax
  1b:  01 d0                   add    %edx,%eax
  1d:  5d                      pop    %rbp
  1e:  c3                      retq
```

Disassembly of **section .data**:

0000000000000000 <ep>:

```
   0:  00 00 00 00
```

0000000000000008 <x>:

```
   8:  0f 00 00 00
```

After Linked .text

Disassembly of section .text:

0000000000**4004f6** <main>:

```
4004f6:     55                  push   %rbp
4004f7:     48 89 e5            mov    %rsp,%rbp
4004fa:     48 83 ec 10         sub    $0x10,%rsp
4004fe:     b8 00 00 00 00      mov    $0x0,%eax
400503:     e8 0d 00 00 00      callq  400515 <a>
400508:     89 45 fc            mov    %eax,-0x4(%rbp)
40050b:     bf 00 00 00 00      mov    $0x0,%edi
400510:     e8 cb fe ff ff      callq  4003e0 <exit@plt>
```

0000000000**400515** <a>:

```
400515:     55                  push   %rbp
400516:     48 89 e5            mov    %rsp,%rbp
400519:     48 8b 05 38 04 20 00    mov    0x200438(%rip),%rax
                                # 600958 <ep>
400520:     8b 10               mov    (%rax),%edx
400522:     8b 05 38 04 20 00   mov    0x200438(%rip),%eax
                                # 600960 <x>
400528:     01 c2               add    %eax,%edx
40052a:     8b 05 38 04 20 00   mov    0x200438(%rip),%eax
                                # 600968 <__TMC_END__>
400530:     01 d0               add    %edx,%eax
400532:     5d                  pop    %rbp
400533:     c3                  retq
400534:     66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
40053b:     00 00 00
40053e:     66 90               xchg   %ax,%ax
```

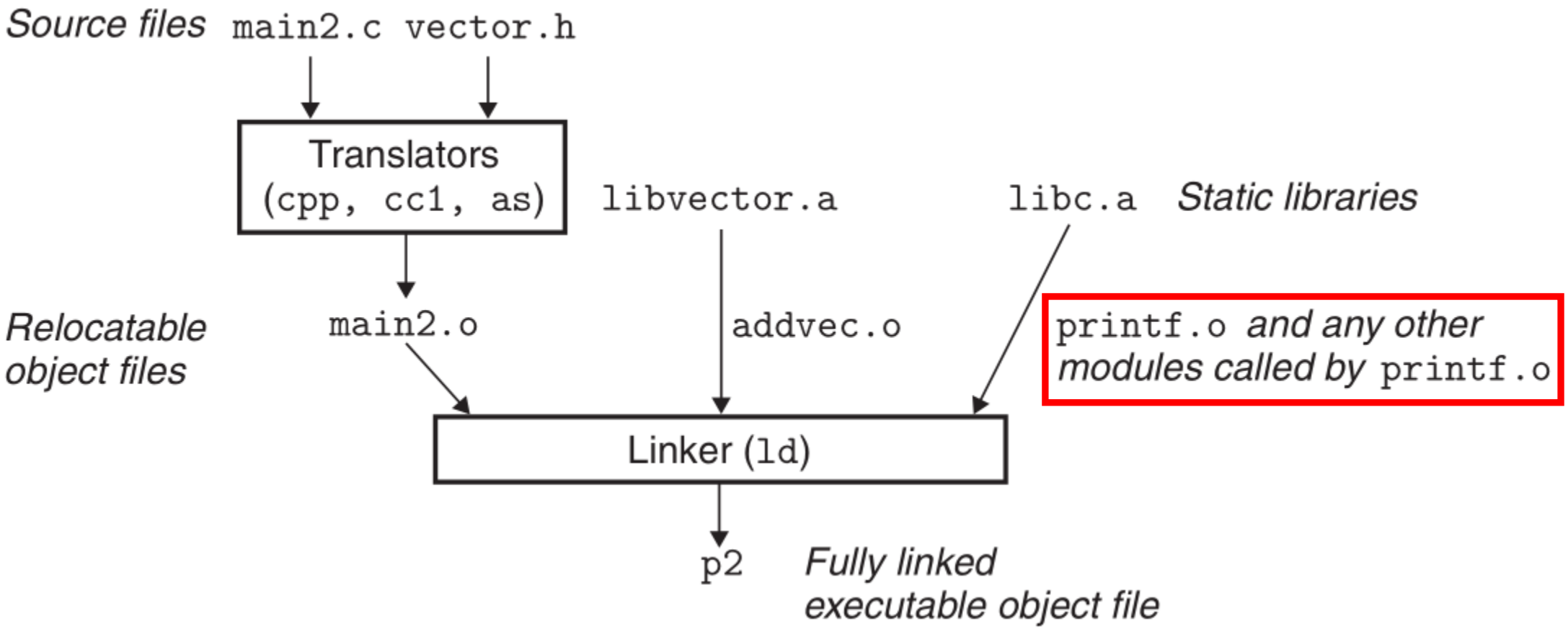
Linking with Static Libraries

- Static library
  - A package of related object modules
  - Copying only the object modules that referenced
  - libc.a: ANSI C standard C library (printf …)
  - libm.a: ANSI C math library
- from “gcc main.c /usr/lib/printf.o /usr/lib/scanf.o …”
- to “gcc main.c /usr/lib/libc.a”
- “ar rcs libxxx.a p2.o swap.o”

Why Static Libraries?

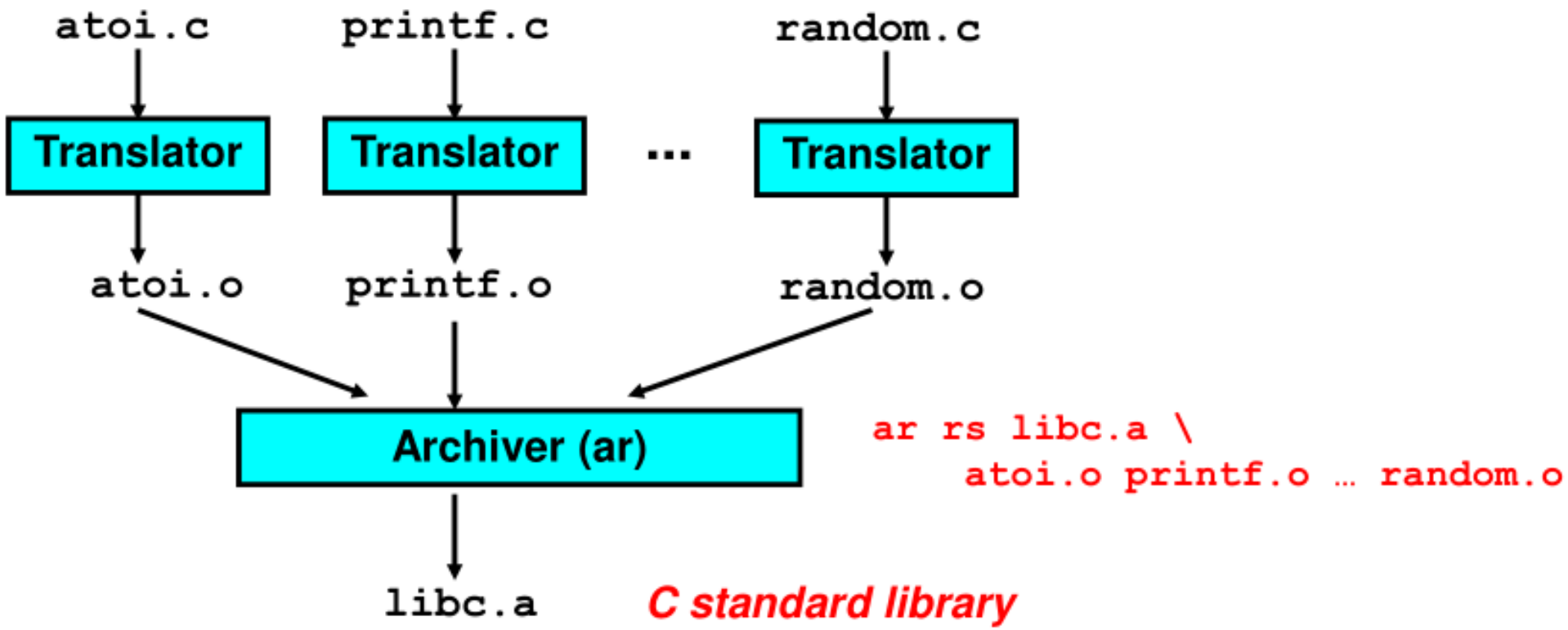
- Functions commonly used by programmers
  - I/O, math, string …
- Putting all functions in a single file
  - Space and time inefficient
- or putting each function to separate files
  - More efficient but burdensome work to link separately
- Static libraries (*. archive files)
  - Aggregating related relocatable objects into a single file
  - Resolving external references by looking symbols in archives

Linking with Static Libraries



Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.

Creating Static Libraries



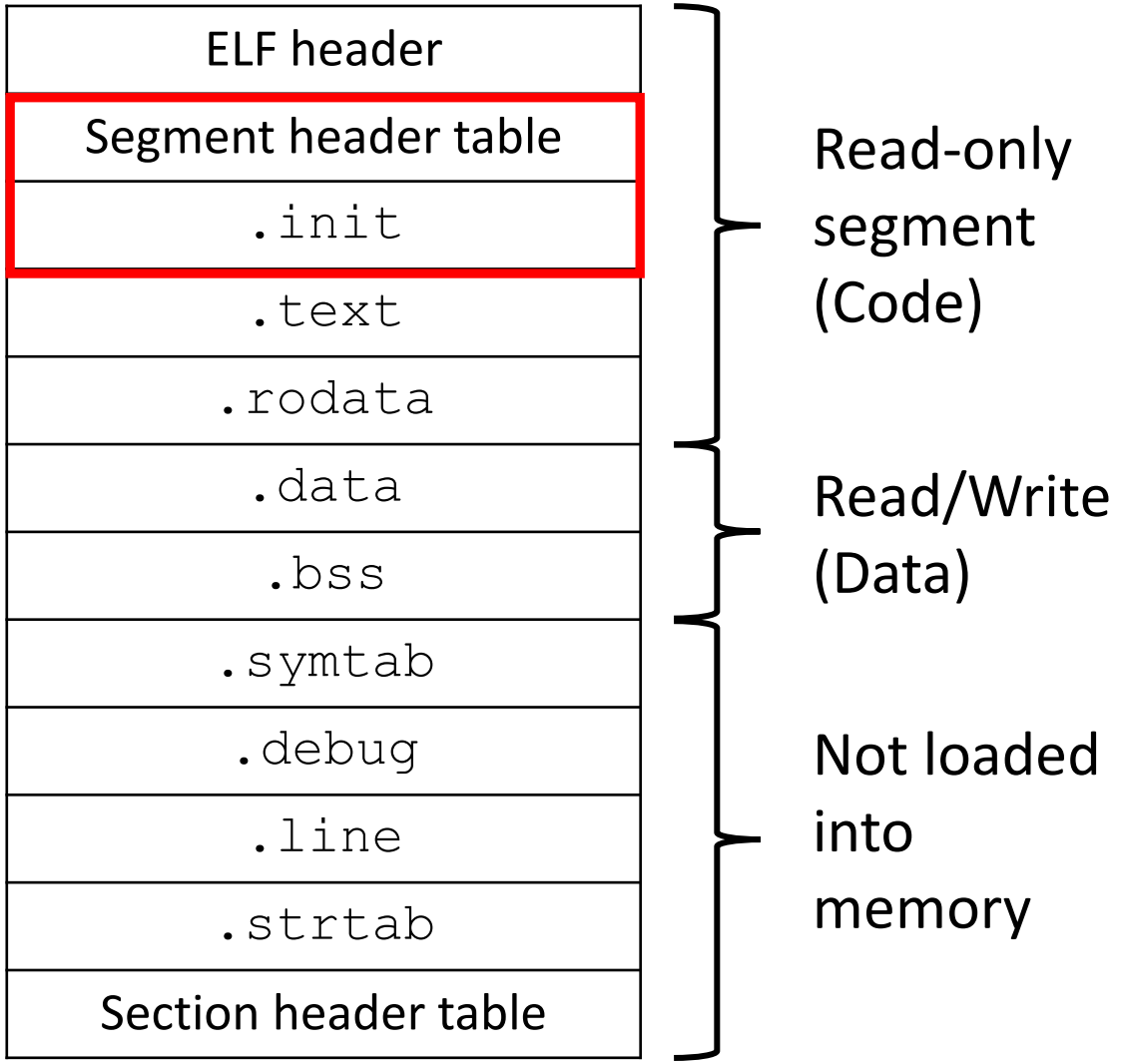
Incremental updates

Executable Object File

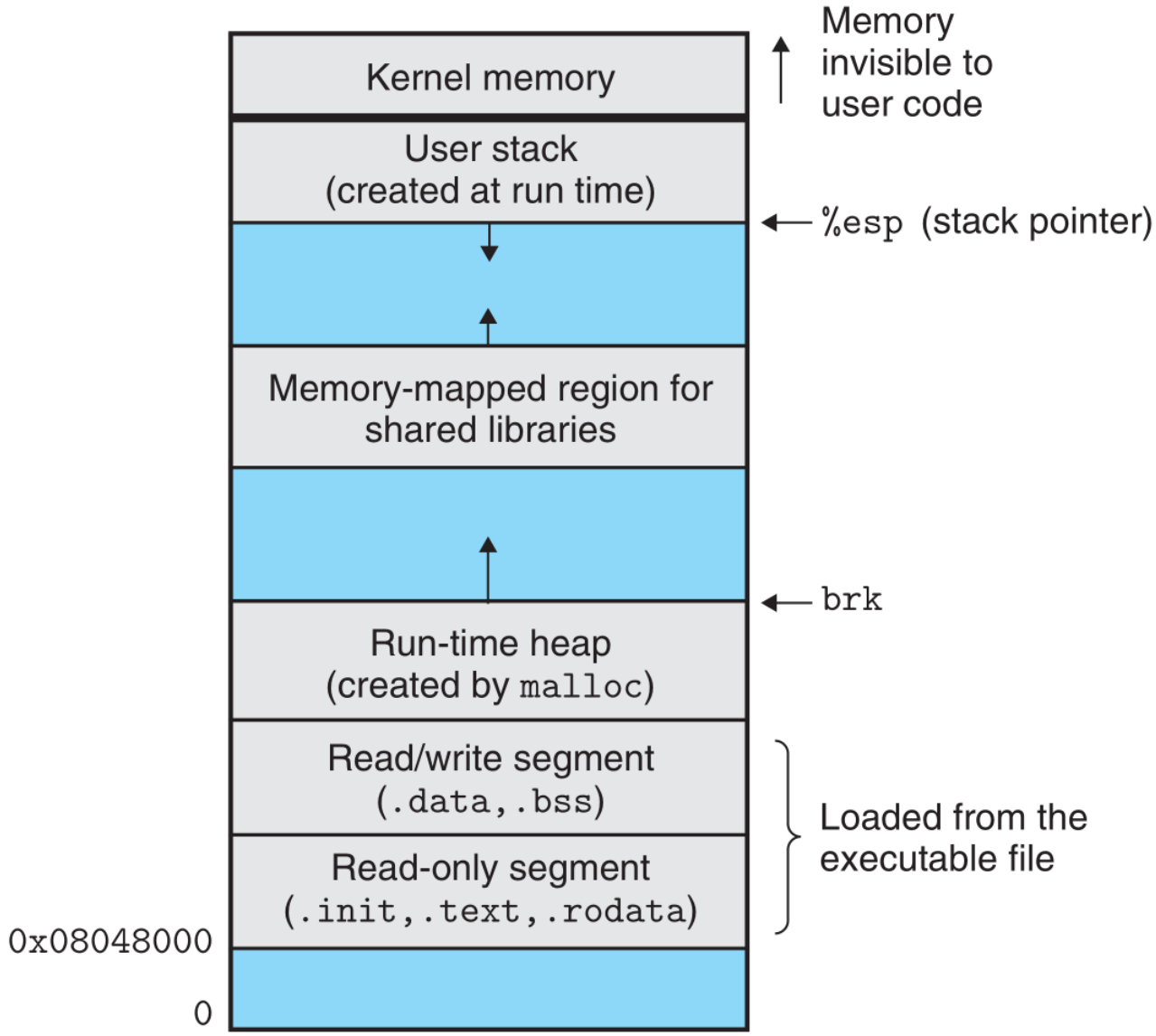
- After linking with static libraries
  - Ready to be loaded into memory and execute
- Executable object file format
  - Similar to relocatable object file format
  - ELF header includes programs' entry point (address of 1st instruction)
  - .init section defines a small function called _init
  - No relocation information (no need to be relocated)

Executable Object Format

- Fully linked (relocated)
  - no .rel sections needed
- Segment header Table
  - Page size
  - Virtual and physical address of memory segments (sections)
  - Segment sizes
- .init
  - Called by programs' initialization code



Linux Run-time Memory Image



Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.

Startup Routines for C Programs

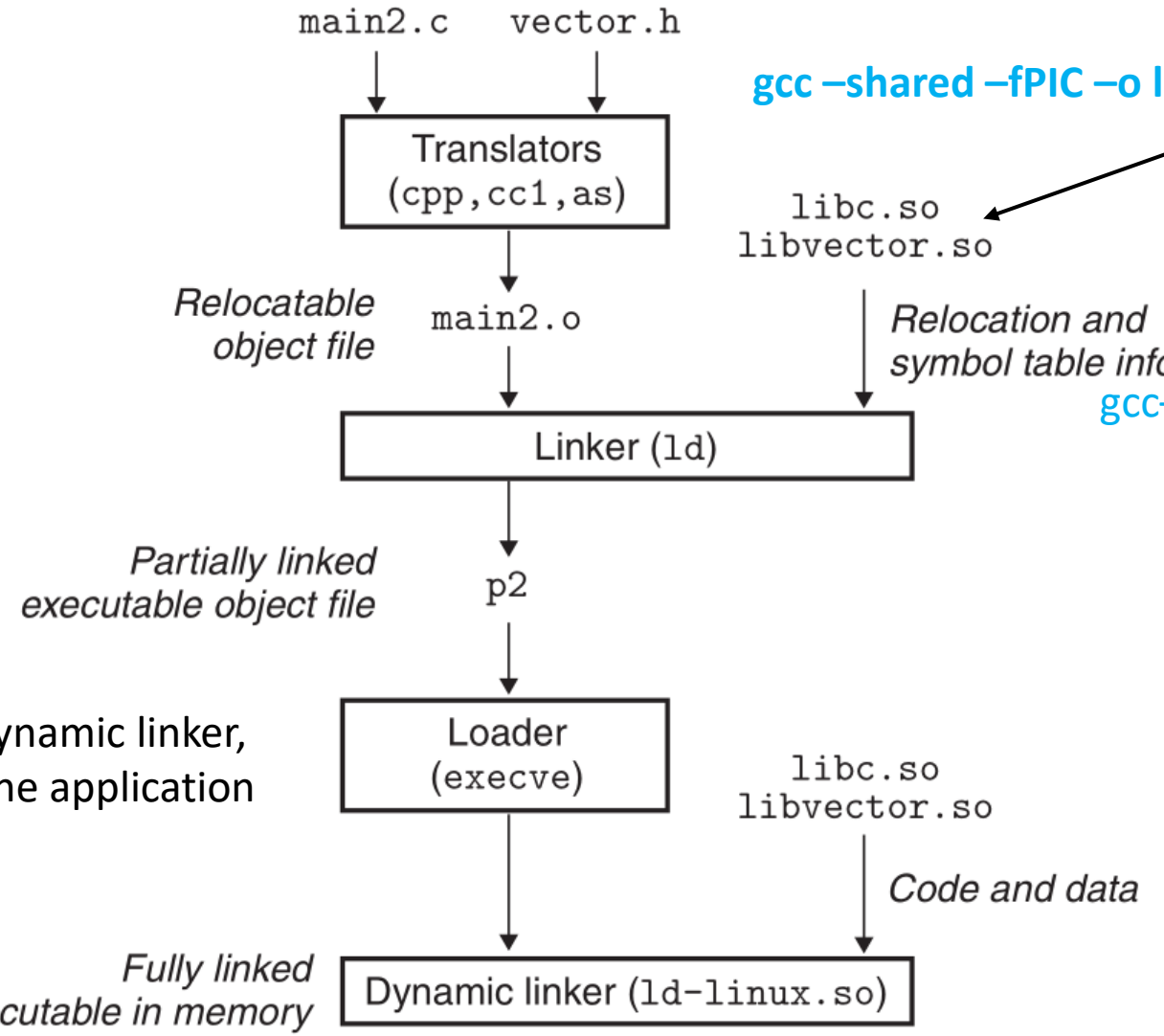
- The loader creates the memory image by copying chunks of the executable object files into code and data segments
- The loader jumps to the program's entry points (_start symbol)
- The startup code (_start) is defined in crt1.o

```
0x080480c0 <_start>:                    /* entry point */
        call _libc_init_first           /* startup code in .text */
        call _init                      /* startup code in .init */
                -> Initialization routines from .text and .init sections
        call atexit                     /* startup code in .text */
                -> Appends a list of routines that should be called when exit
        call main                       /* applications code */
                -> When completes, exit() is called to run routines registered by atexit
        call _exit                      /* return control to OS */
```

Why Shared Libraries?

- Disadvantages of static libraries
  - Duplicated common functions in many programs (e.g., printf)
  - Space inefficient for duplicated codes in text segment
  - Requirement of relinking all programs if a function changes
- Shared libraries (*.so or *.dll)
  - Dynamically loaded and linked at run-time
  - Exactly one shared library for a particular library
  - Sharing libraries in memory by different processes
  - By loader (ld-linux.so) at load-time
  - By user (dlopen() function) at run-time

Linking with Shared Libraries



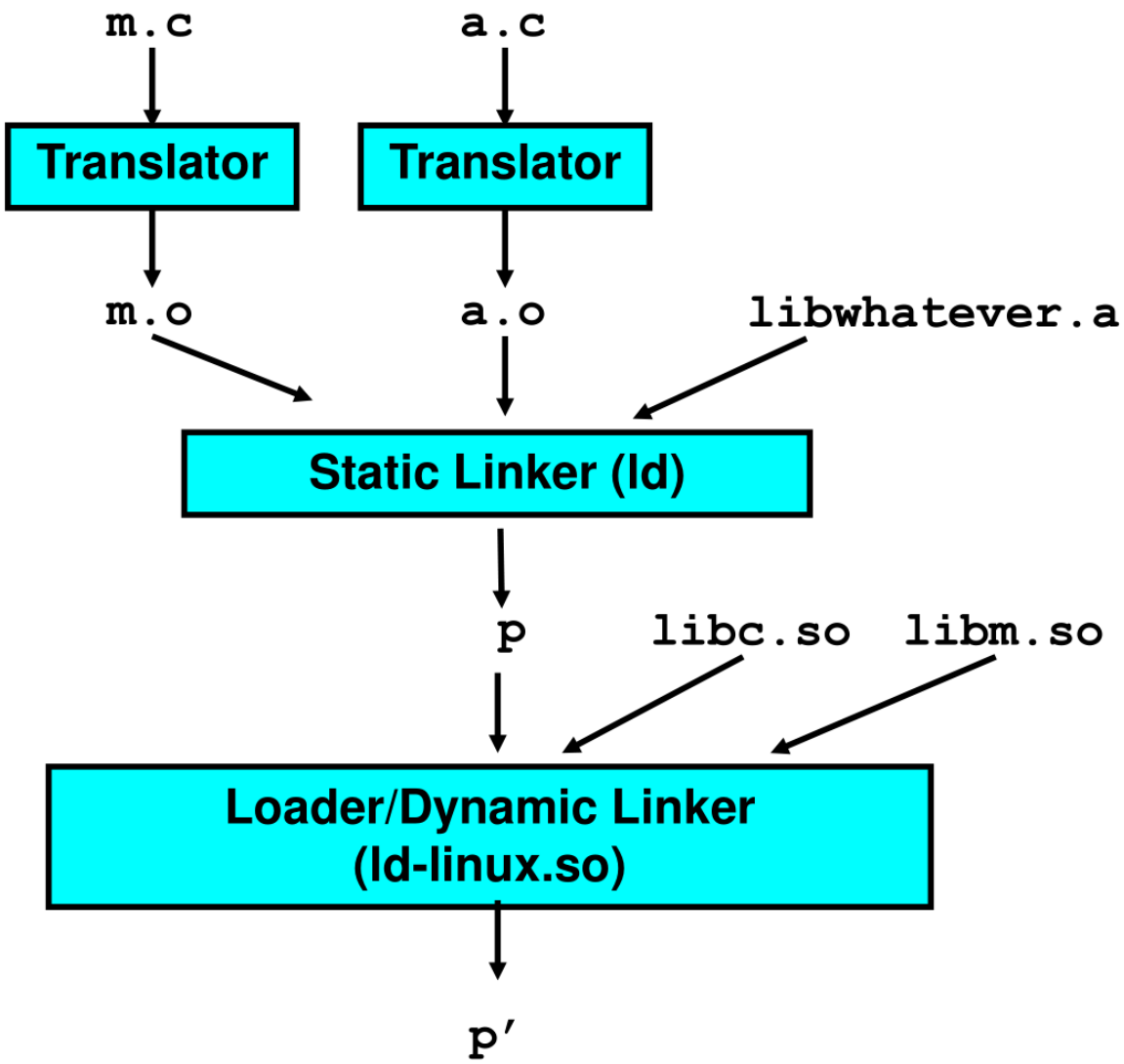
gcc –shared –fPIC –o libvector.so addvec.c multivec.c

gcc–o p2 ./libvector.so

Loader loads and runs the dynamic linker, and then passes control to the application

Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.

The Complete Picture



Copyright © 2011, 2003 by Randal E. Bryant and David R. O'Hallaron. All rights reserved.