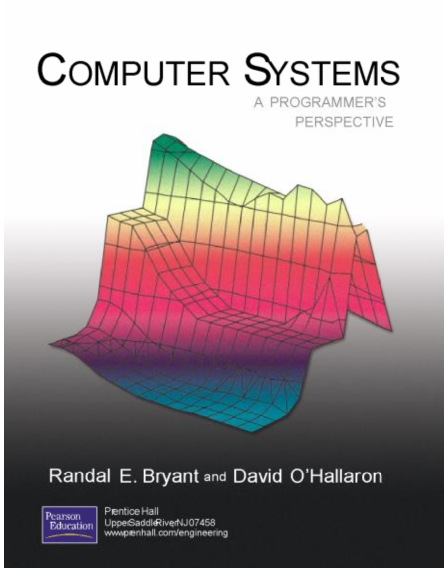
Linking - Unificação

http://csapp.cs.cmu.edu/public/lectures/class15.ppt

Topicos

- Scope
- Static linking
- Object files
- Static libraries
- Loading
- Dynamic linking of shared libraries


COMPUTER SYSTEMS
A PROGRAMMER'S PERSPECTIVE
Randal E. Bryant and David O'Hallaron
Pearson Education
Prentice Hall
Upper Saddle River NJ 07458
www.prenhall.com/engineering

Âmbito da Validade (Scope)

scope duma variável = Âmbito da Validade

= As secções de código onde a variável está “”valida””

```
int x;                      Variavel Global
Tipo funcao( int y )   y local – scope é a função
{
    extern int a;       variavel global
                         (memoria atribuida numa outra ficheiro
    static int b=1;     variavel local a função
                          (mas não guardado no stack)
                           inicialização é feita apenas uma vez
    int z;               variavle local cujo scope é o bloco
                         onde se econtra - neste caso a função toda
    while (z)
    {
       float a,y        variaveis a e y – local a este bloco
       z=z-y+x;         variavel  y aqui é do tipo float
    }
    ………….
    z = a+y;
}
```

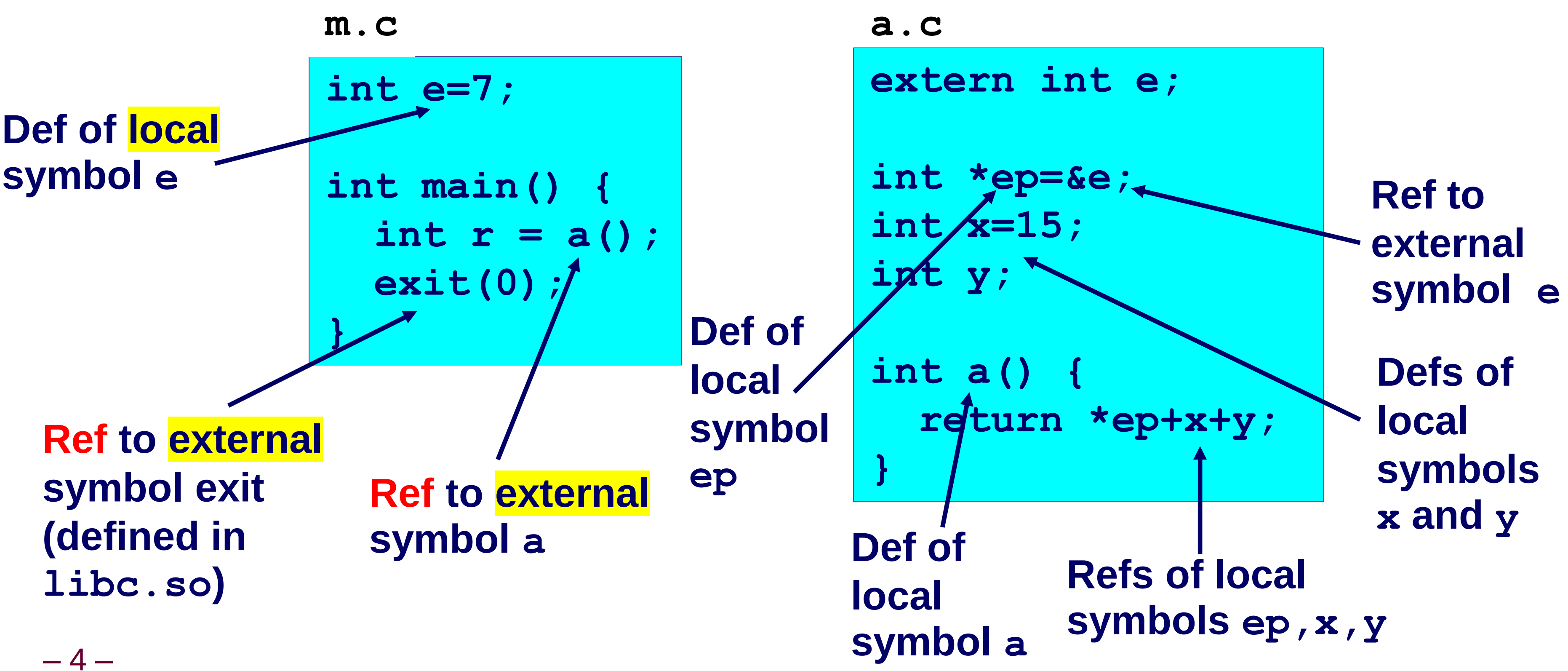
ANSII C versus GNU/WIN extensions

```
Declarações
{
    Declarações
    Instruções
}
```

```
Declarações
{
    Declarações
    Instruções
    Declarações
    Instruções
}

for(int i=0..
```

Symbols

- *Symbols* are lexical entities that name functions , variables , constants
- Each symbol has a *value* (typically a memory address).
- Code consists of symbol *definitions* and *references*.
- References can be either *local* or *external*.



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;
}
```

Def of local symbol e

Ref to external symbol exit (defined in libc.so)

Ref to external symbol a

Def of local symbol ep

Def of local symbol a

Refs of local symbols ep,x,y

Ref to external symbol e

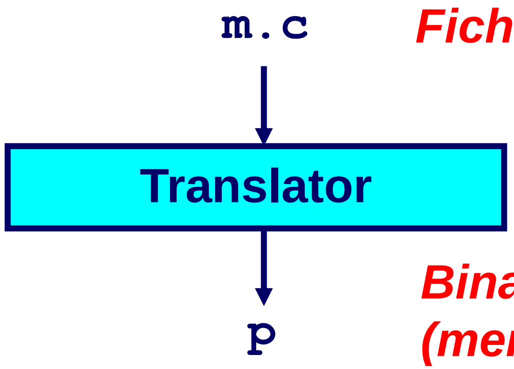
Defs of local symbols x and y

static

```
include <stdio.h>
int fib(int n, int flag)
{
   static int counter=0;
   if (flag==0) {
                printf("Report: chamadas %d\n",counter);
                return 0;
   }
   counter++;
   if (n<2)
         return 1;
   return
        fib(n-1,1)+fib(n-2,1);
}
main()
{
   int n,f;
   scanf("%d",&n);
   printf("fib(%d)=%d\n",n,fib(n,1));
   fib(0,0);
}
```

```
[crocker@penhas linker]$ ./a.out
6
fib(6)=13
Report: chamadas 25
```

O Porquê de Linker ! Compilição Simples



`m.c`

Ficheiro de código fonte, ASCII

Translator

`p`

Binary executable object file (memory image on disk)

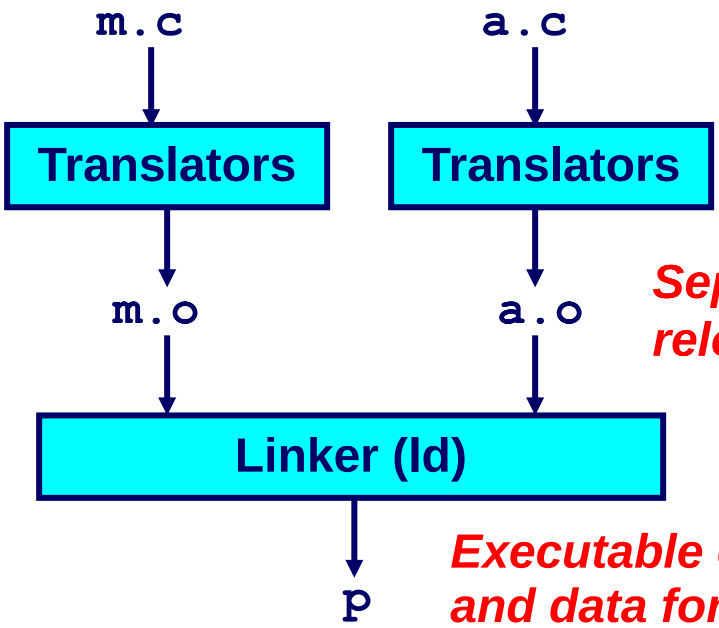
Problemas:

- **Eficácia: pequena mudança implica recompilação completa**
- **Modularity: difícil de partilhar funções comuns (e.g. `printf`)**

Solução:

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

Melhoraria : Utilização dum Linker



Separately compiled relocatable object files

Executable object file (contains code and data for all functions defined in `m.c` and `a.c`)

O que faz um Linker ?

1.Merges object files

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

2.What Does Merge Mean ?

- Resolves(1-1 Mapping) the 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.
  - Recall that Instructions contain References : to code or data
    - » code: `y=a();         /* reference to code symbol a */`
    - » data: `int *xp=&x;   /* reference to data symbol x */`

Porquê 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.
  - Executable files and running programs contain only code for the functions they actualy use.

Executable and Linkable Format (ELF)

Executable and Linkable Format (ELF)

Um formato binário padrão para ficheiro objectos: ELF binaries

Um único formato para

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

História

- Os primeiros Sistemas Unix (Bell-Labs) `a.out` format !!!
- COFF- common object file format (Unix system V AT&T)
- BSD Unix e Linux COFF → ELF

Outros

- PE – Windows Portable Execution Format
- MACH-O Macintosh
- etc

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

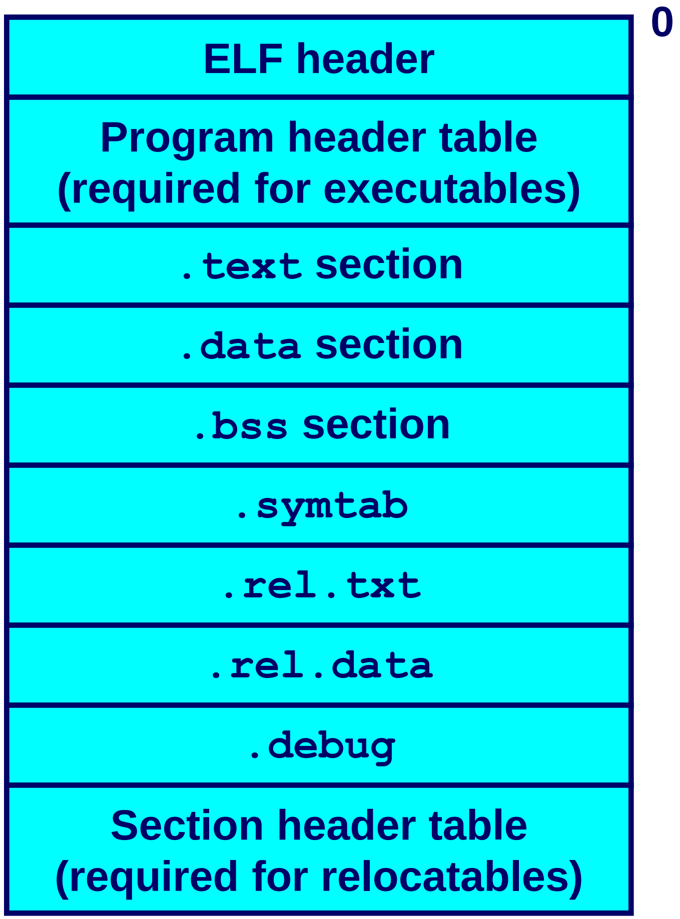
- Code

`.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

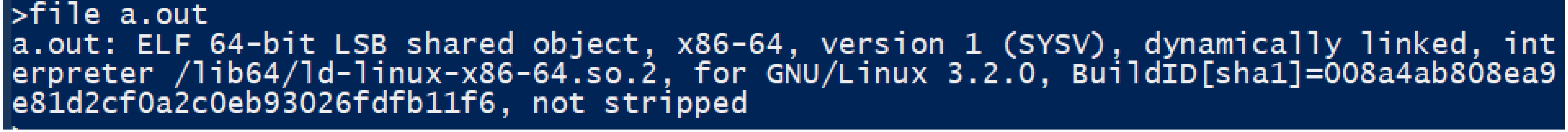
`.symtab`

`.rel.txt`

`.rel.data`

`.debug`

Section header table
(required for relocatables)



```
>file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, int
erpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=008a4ab808ea9
e81d2cf0a2c0eb93026fdfb11f6, not stripped
```

```
[crocker@penhas pagina]$ cc -o ola ola.c
[crocker@penhas pagina]$ objdump -af ola

Header       : file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080485c0
```

```
[Oracle]$ readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048300
  Start of program headers:          52 (bytes into file)
  Start of section headers:          2104 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (byte
```

ELF Object File Format (cont)

`.symtab` section

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

`.rel.text` section

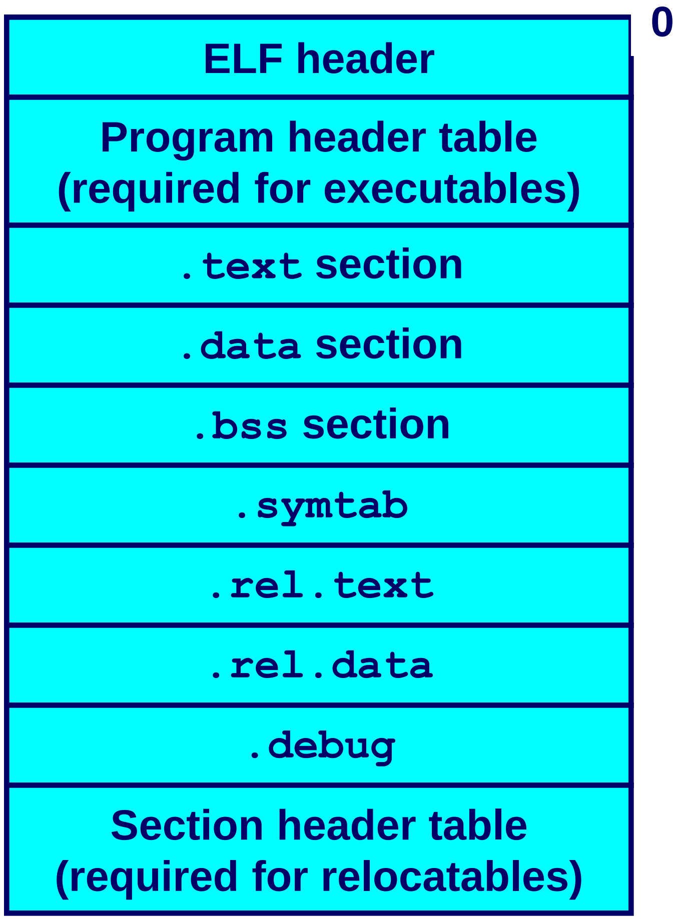
- Relocation info for `.text` section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

`.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)

Tabelas de Símbolos

Cada ficheiro objecto relocatável tem uma tabela de símbolos

Exemplo

```
/* bar.c */
int x = 2 ;
int g;
int f()
{
    extern int zz;
    static int y;
    zz=zz+fonde();
    y=x+y+g+zz;
    return y;
}
```

Tabela de Símbolos

Num maquina linux – usando

`readelf -s bar.o`

ou

`nm bar.o`

```
[crocker@penhas linker]$ nm bar.o
00000000 T f
00000004 C g
         U fonde
00000000 D x
00000004 b y.1283
         U zz
```

D initialized data section
b Uninitialized (local) data section BSS
U undefined
T symbol in text section
C Common symbols are uninitialized data.

Exemplo

```
[rtems@VirtualRTEMS sisops]$ cc -c ola.c
[rtems@VirtualRTEMS sisops]$ nm ola.o
00000000 T main
         U puts
[rtems@VirtualRTEMS sisops]$ readelf -s ola.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS ola.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    6
     8: 00000000    54 FUNC    GLOBAL DEFAULT    1 main
     9: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
```

```
#include <stdio.h>
#define valor = 6
int main()
{
  int x = 1;
  x = x +6;
  puts("ola\n");
  return 1;
}
```

Example C Program

a.c

```
extern int e;

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

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

m.c

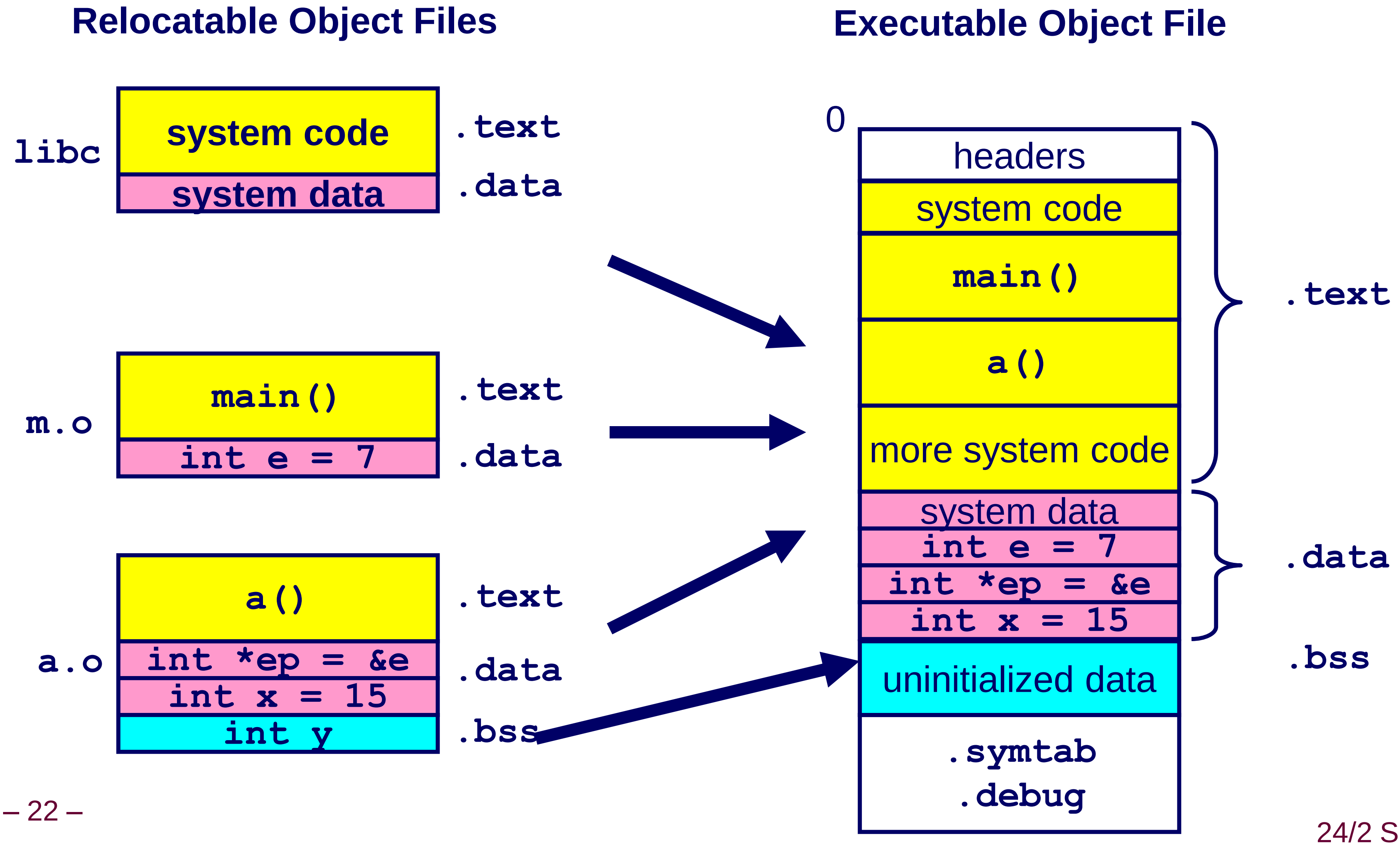
```
int e=7;

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

```
bash-4.1$ nm m.o
         U a
00000000 D e
         U exit
00000000 T main
```

```
bash-4.1$ nm a.o
00000000 T a
         U e
00000000 D ep
00000004 D x
00000004 C y
```

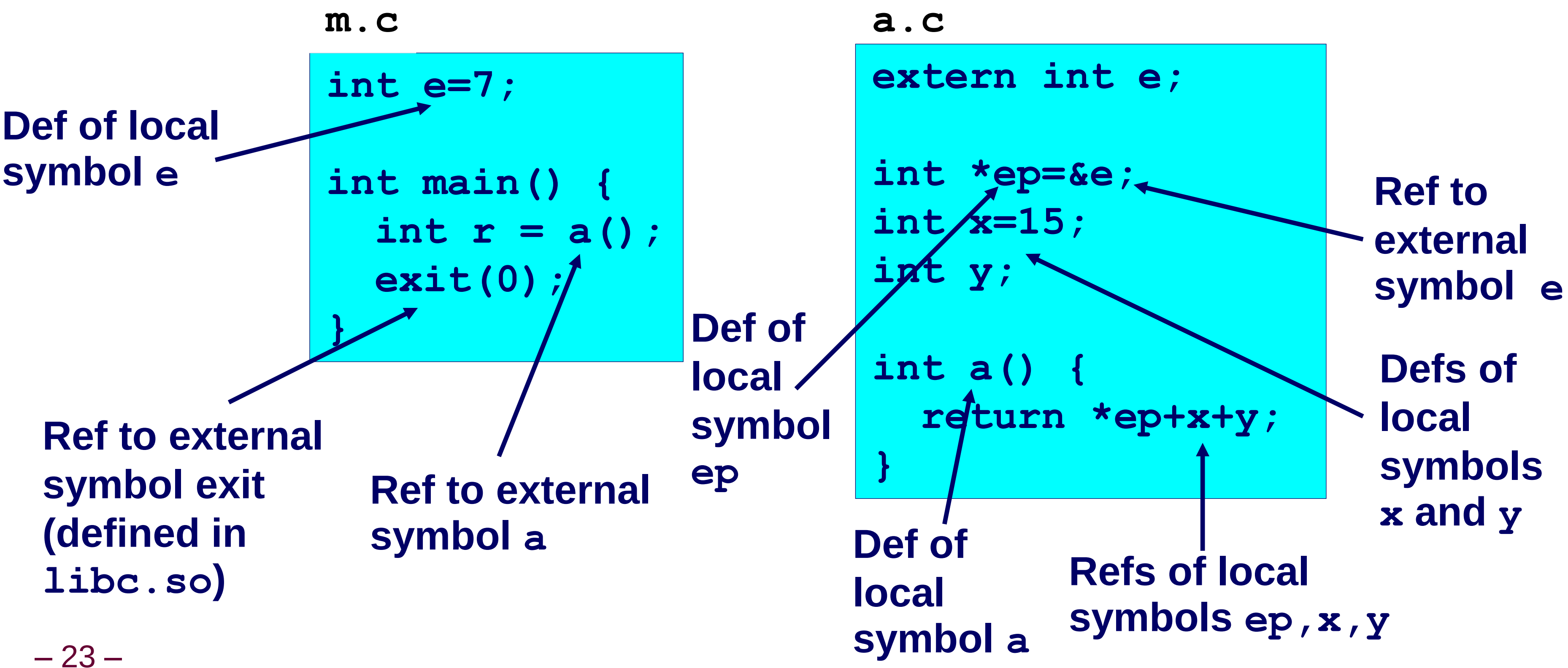
D initialized data section
b Uninitialized (local) data section BSS
U undefined
T symbol in text section
C Common symbols are uninitialized data.

Merging Relocatable Object Files into an Executable Object File



Relocating Symbols and Resolving External References

- *Symbols* are lexical entities that name functions and variables.
- Each symbol has a *value* (typically a memory address).
- Code consists of symbol *definitions* and *references*.
- References can be either *local* or *external*.



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;
}
```

Def of local symbol e

Ref to external symbol exit (defined in libc.so)

Ref to external symbol a

Def of local symbol ep

Ref to external symbol e

Defs of local symbols x and y

Def of local symbol a

Refs of local symbols ep,x,y

m.o Relocation Info

linux(objdump) Mac (otool) .NET Debug Windows-Disassembly.

m.c

```
int e=7;

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

```
Disassembly of section .text:

00000000 <main>: 00000000 <main>:
   0:   55                pushl  %ebp
   1:   89 e5             movl   %esp,%ebp
   3:   e8 fc ff ff ff    call   4 <main+0x4>
                          4: R_386_PC32    a
   8:   6a 00             pushl  $0x0
   a:   e8 fc ff ff ff    call   b <main+0xb>
                          b: R_386_PC32    exit
   f:   90                nop
```

```
Disassembly of section .data:

00000000 <e>:
   0:   07 00 00 00
```

a.o Relocation Info (.data)

a.c

```
extern int e;

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

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

```
Disassembly of section .data:

00000000 <ep>:
   0:   00 00 00 00
                          0: R_386_32     e
 00000004 <x>:
   4:   0f 00 00 00
```

a.o Relocation Info (.text)

a.c

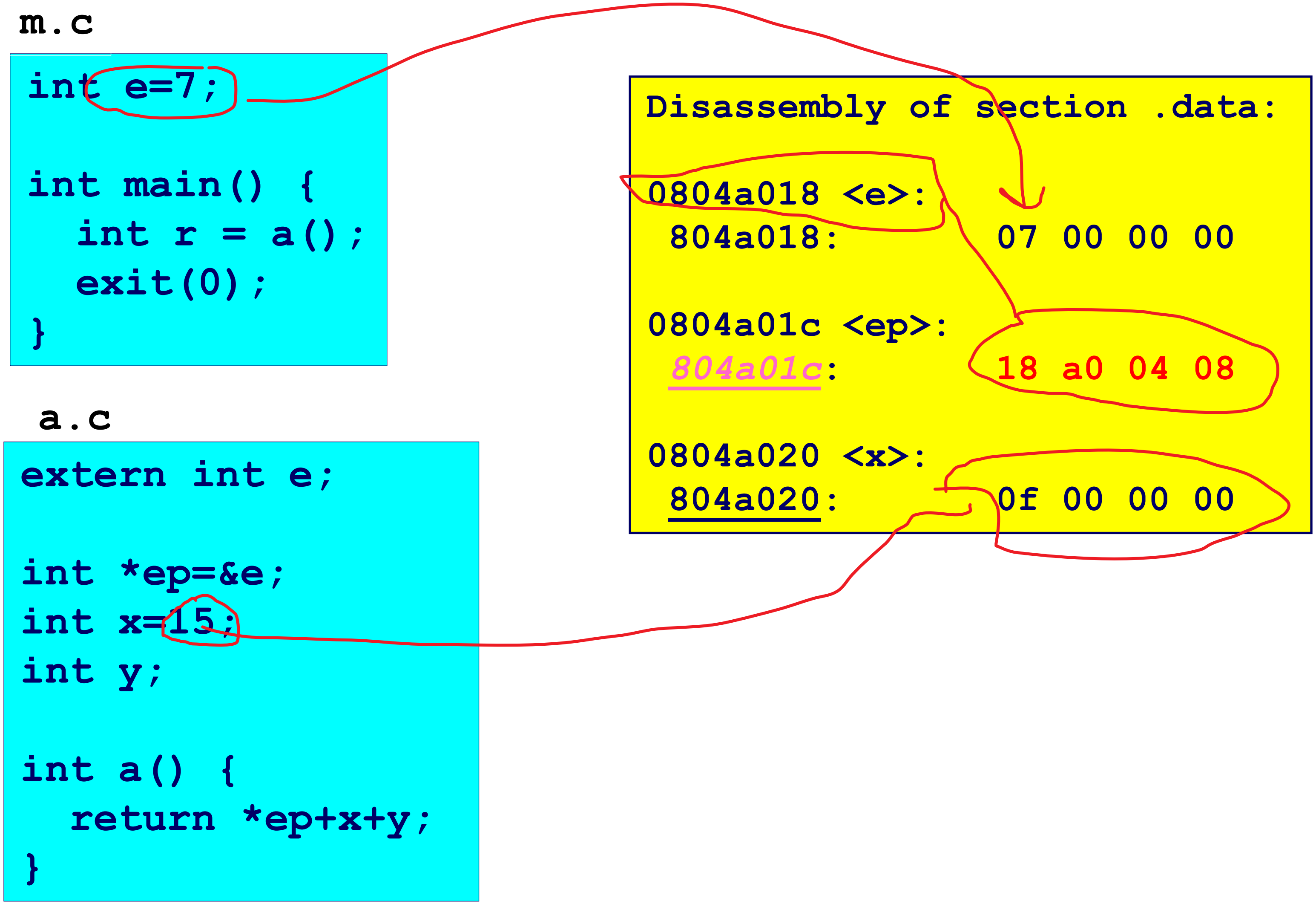
```
extern int e;

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

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

```
Disassembly of section .text:

00000000 <a>:
   0:   55                push1  %ebp
   1:   8b 15 00 00 00    movl   0x0,%edx
   6:   00
                           3: R_386_32      ep
   7:   a1 00 00 00 00    movl   0x0,%eax
                           8: R_386_32      x
   c:   89 e5             movl   %esp,%ebp
   e:   03 02             addl   (%edx),%eax
  10:   89 ec             movl   %ebp,%esp
  12:   03 05 00 00 00    addl   0x0,%eax
  17:   00
                           14: R_386_32     y
  18:   5d                popl   %ebp
  19:   c3                ret
```

Executable After Relocation and External Reference Resolution (.text)

```
08048530 <main>:
 8048530:       55                pushl  %ebp
 8048531:       89 e5             movl   %esp,%ebp
 8048533:       e8 08 00 00 00    call   8048540 <a>
 8048538:       6a 00             pushl  $0x0
 804853a:       e8 35 ff ff ff    call   8048474 <_init+0x94>
 804853f:       90                nop

08048540 <a>:
 8048540:       55                pushl  %ebp
 8048541:       8b 15 1c a0 04    movl   0x804a01c,%edx
 8048546:       08
 8048547:       a1 20 a0 04 08    movl   0x804a020,%eax
 804854c:       89 e5             movl   %esp,%ebp
 804854e:       03 02             addl   (%edx),%eax
 8048550:       89 ec             movl   %ebp,%esp
 8048552:       03 05 d0 a3 04    addl   0x804a3d0,%eax
 8048557:       08
 8048558:       5d                popl   %ebp
 8048559:       c3                ret
```

Executable After Relocation and External Reference Resolution(.data)



m.c

```
int e=7;

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

a.c

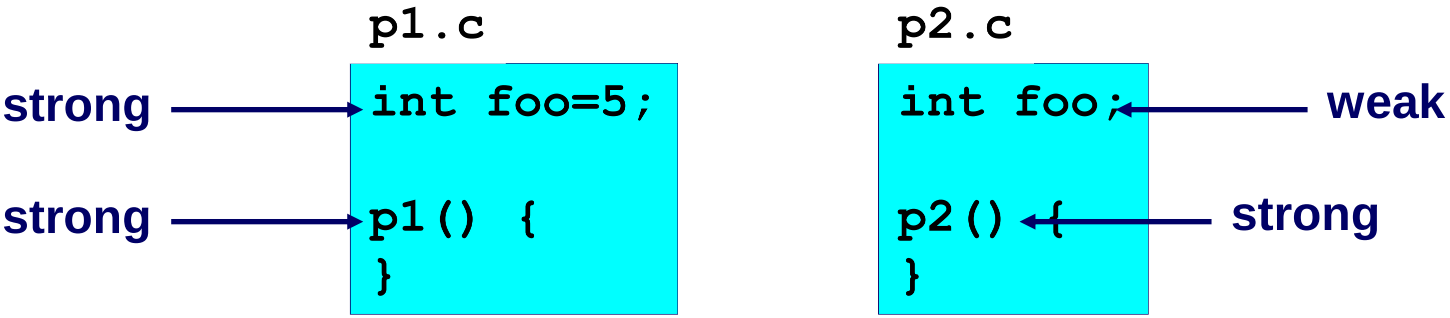
```
extern int e;

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

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

```
Disassembly of section .data:

0804a018 <e>:
 804a018:        07 00 00 00

0804a01c <ep>:
 804a01c:        18 a0 04 08

0804a020 <x>:
 804a020:        0f 00 00 00
```

Strong and Weak Symbols

Program symbols are either strong or weak

- *strong*: procedures and initialized globals
- *weak*: uninitialized globals



p1.c

```
int foo=5;

p1() {
}
```

p2.c

```
int foo;

p2() {
}
```

Linker’s Symbol Rules

Rule 1. A strong symbol can only appear once.

Rule 2. A weak symbol can be overridden by a strong symbol of the same name.

- references to the weak symbol resolve to the strong symbol.

Rule 3. If there are multiple weak symbols, the linker can pick an arbitrary one.

`Regras : Toma Nota` …

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 we really want?

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

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

Linker picks arbitary weak symbol x
Writes to `x` in `p2` might overwrite `y`! Evil!

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

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

Linker picks integer
Writes to `x` in `p2` will overwrite `y`!
Nasty!

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

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

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

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Code Libraries

Packaging Commonly Used Functions

- static clibraries
- dynamic libraries

Packaging Commonly Used Functions

How to package functions commonly used by programmers?

- Math, I/O, memory management, string manipulation, etc.

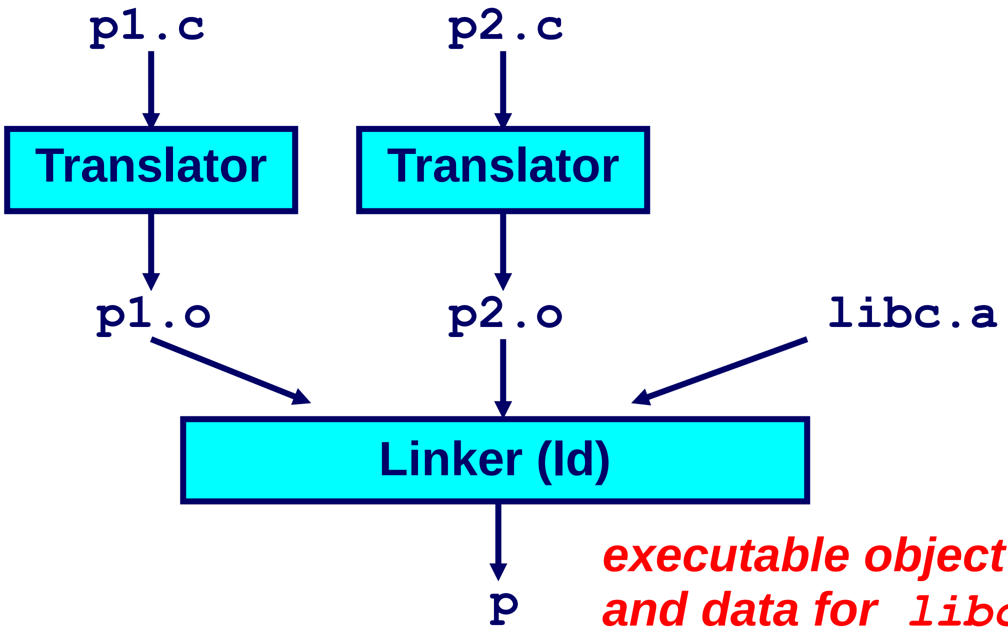
Awkward, given the linker framework so far:

- Option 1: Put all functions in a single source file
  - Programmers link big object file into their programs
  - Space and time inefficient
- Option 2: Put each function in a separate source file
  - Programmers explicitly link appropriate binaries into their programs
  - More efficient, but burdensome on the programmer

Solution: *static libraries* (`.a` archive files)

- Concatenate related relocatable object files into a single file with an index (called an archive).
- Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
- If an archive member file resolves reference, link into executable.

Static Libraries (archives)



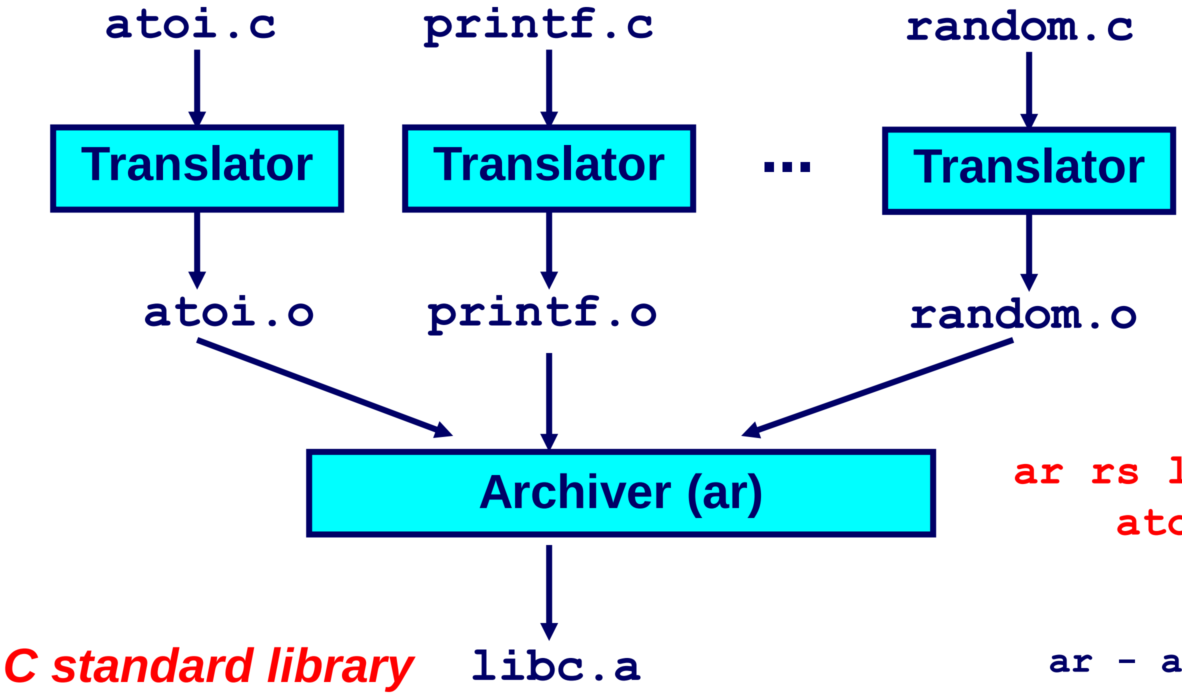
static library (archive) of relocatable object files concatenated into one file.

executable object file (only contains code and data for `libc` functions that are called from `p1.c` and `p2.c`)

Further improves modularity and efficiency by packaging commonly used functions [e.g., C standard library (`libc`), math library (`libm`)]

Linker selectively only the `.o` files in the archive that are actually needed by the program.

Creating Static Libraries



```
ar rs libc.a \
    atoi.o printf.o … random.o
```

```
ar - archiver
r - adicionar ou substituir
s - criar indice
```

Archiver **:**

- **permite actualizações pontuais:**
  - **Recompilar uma função modificada**
  - **Substituir ficheiro .o no arquivo**

Commonly Used Libraries

`libc.a` (the C standard library)

- 8 MB archive of 900 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

- 1 MB archive of 226 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Commonly Used Libraries

```
[penhas]$
ar -t /usr/lib/libc.a | sort | grep "^f"

[penhas]$ ar -t /usr/lib/libc.a | wc -l
1269

[penhas]$ ar -t /usr/lib/libm.a | wc -l
401
```

```
Mac/Darwin
libtool - replaces ar
otool -   replaces objdump, nm, ldd
```

```
fprintf.o
fpu_control.o
fputc.o
fputc_u.o
fputwc.o
fputwc_u.o
freelocale.o
fremovexattr.o
freopen64.o
freopen.o
fscanf.o
fseek.o
fseeko64.o
fseeko.o
fsetxattr.o
fstab.o
```

Using Static Libraries

Linker's algorithm for resolving external references:

- Scan .o files and .a files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.
- If any entries in the unresolved list at end of scan, then error.

Problem:

- Command line order matters!
- Moral: put libraries at the end of the command line.

```
bass> gcc -L. libtest.o -lmine
bass> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Shared Libraries

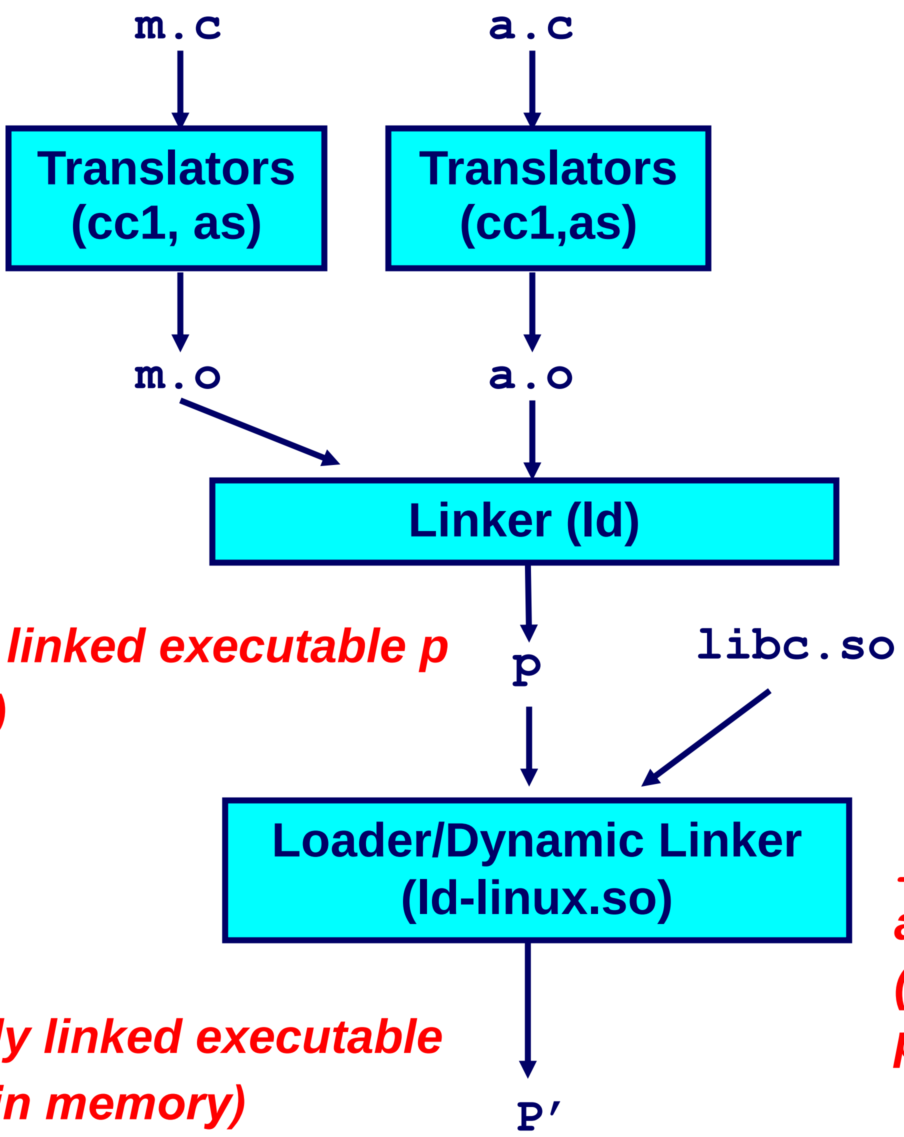
Static libraries have the following disadvantages:

- Potential for duplicating lots of common code in the executable files on a filesystem.
  - e.g., every C program needs the standard C library
- Potential for duplicating lots of code in the virtual memory space of many processes.
- Minor bug fixes of system libraries require each application to explicitly relink

Solution:

- ***Shared libraries*** (dynamic link libraries, DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.
  - Dynamic linking can occur when executable is first loaded and run.
    - » Common case for Linux, handled automatically by `ld-linux.so`.
  - Dynamic linking can also occur after program has begun.
    - » In Linux, this is done explicitly by user with `dlopen()`.
    - » Basis for High-Performance Web Servers.
  - Shared library routines can be shared by multiple processes.

Dynamically Linked Shared Libraries



`m.c`

`a.c`

Translators (cc1, as)

Translators (cc1,as)

`m.o`

`a.o`

Linker (ld)

Partially linked executable p (on disk)

`p`

`libc.so`

Shared library of dynamically relocatable object files

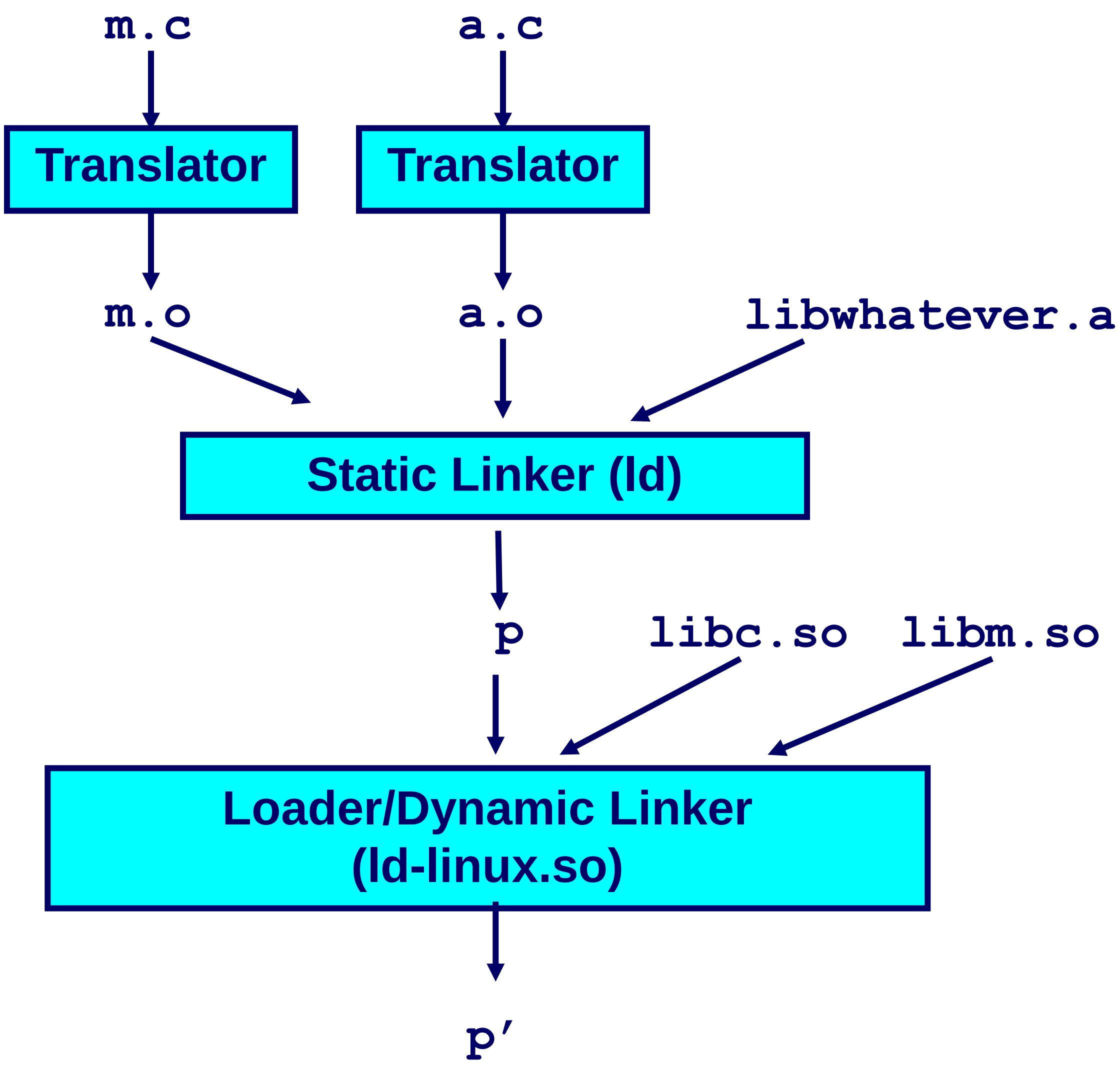
Loader/Dynamic Linker (ld-linux.so)

`libc.so` functions called by `m.c` and `a.c` are loaded, linked, and (potentially) shared among processes.

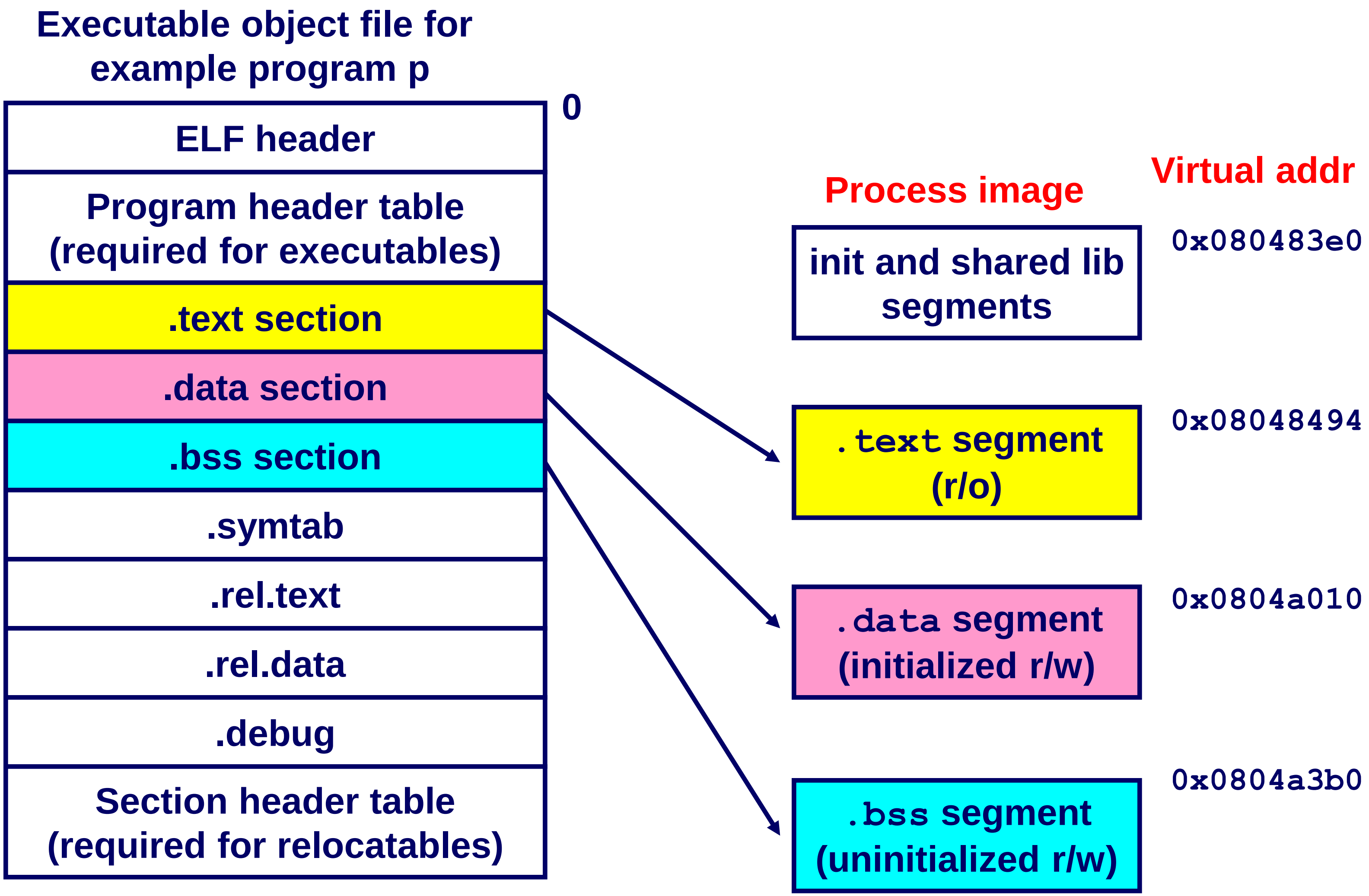
Fully linked executable p’ (in memory)

`p’`

The Complete Picture



Loading Executable Binaries


Executable object file for example program p
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)
Process image
Virtual addr
init and shared lib segments
0x080483e0
.text segment (r/o)
0x08048494
.data segment (initialized r/w)
0x0804a010
.bss segment (uninitialized r/w)
0x0804a3b0