Linking

Alan L. Cox

alc@cs.rice.edu

Some slides adapted from CMU 15.213 slides

x86-64 Assembly

Brief overview last time

- Lecture notes and x86-64 assembly overview available on course web page

You will not write any x86-64 assembly

- Need to be able to recognize/understand code fragments
- Need to be able to correlate assembly code to source code

More assembly examples today

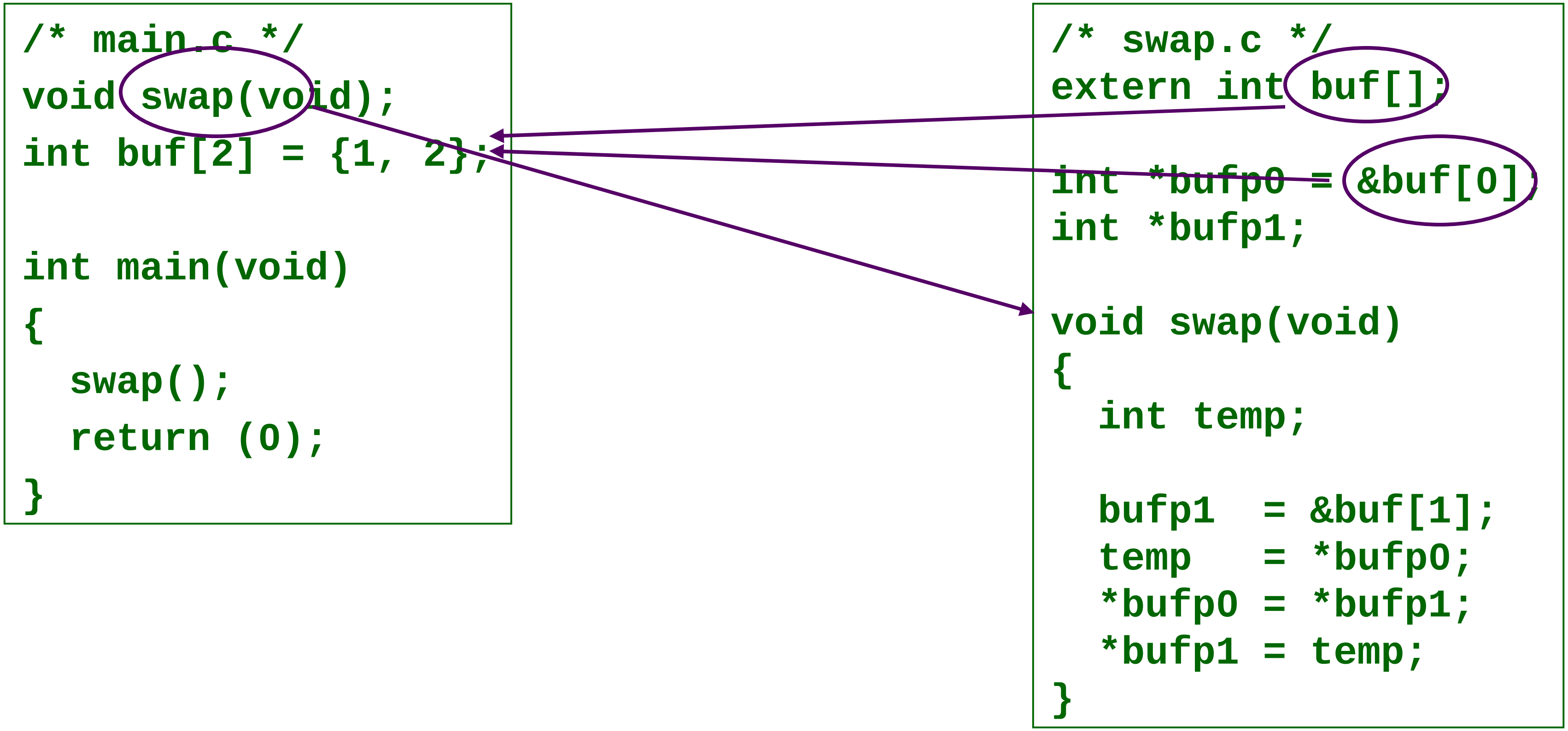
Linking

Linking: collecting and combining various pieces of code and data into a single file that can be loaded into memory and executed

Why learn about linking?

- It will help you build large programs
- It will help you avoid dangerous program errors
- It will help you understand how language scoping rules are implemented
- It will help you understand other important systems concepts (that are covered later in the class)
- It will enable you to exploit shared libraries

Example Program



```
/* main.c */
void swap(void);
int buf[2] = {1, 2};

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

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

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

void swap(void)
{
  int temp;

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

Compilation

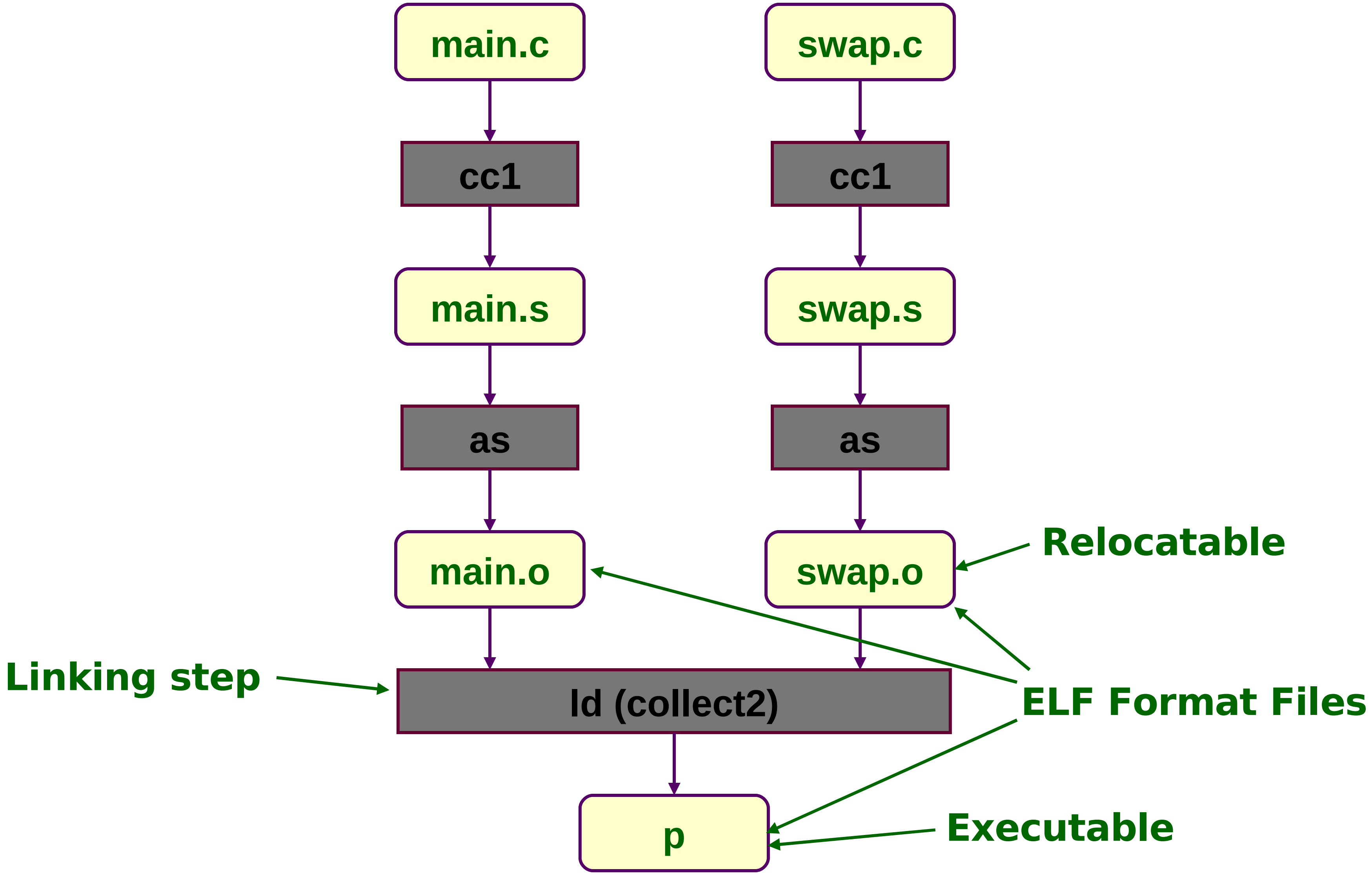
Compiler: .c to .s

Assembler: .s to .o

```
UNIX% gcc -v -O -g -o p main.c swap.c
…
cc1 -quiet -v main.c -quiet -dumpbase main.c -mtune=generic
  -auxbase main -g -O -version -o /tmp/cchnheja.s
as -V -Qy -o /tmp/ccmNFRZd.o /tmp/cchnheja.s
…
cc1 -quiet -v swap.c -quiet -dumpbase swap.c -mtune=generic
  -auxbase swap -g -O -version -o /tmp/cchnheja.s
as -V -Qy -o /tmp/ccx8FECg.o /tmp/ccheheja.s
…
collect2 --eh-frame-hdr –m elf_x86_64 --hash-style=gnu -dynamic-
  linker /lib64/ld-linux-x86-64.so.2 -o p crt1.o crti.o
  crtbegin.o –L<..snip..> /tmp/ccmNFRZd.o /tmp/ccx8FECg.o –lgcc
  --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed
  -lgcc_s --no-as-needed crtend.o crtn.o
```

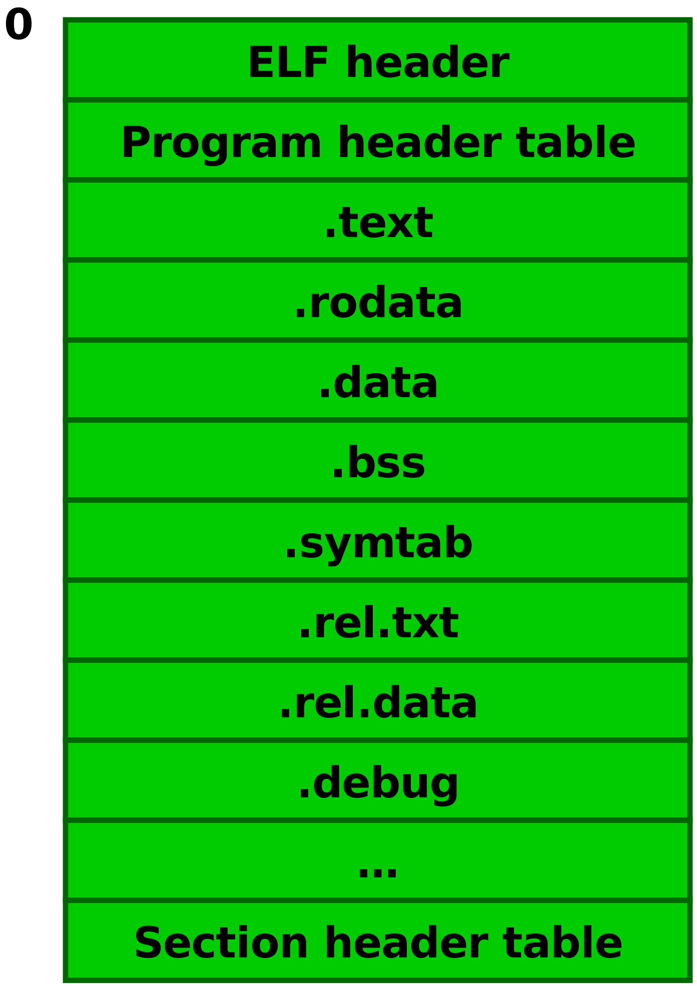
Linker: .o to executable

Compilation



ELF File Format

Order & existence of segments is arbitrary, except ELF header must be present and first



0

ELF header

Program header table

.text

.rodata

.data

.bss

.symtab

.rel.txt

.rel.data

.debug

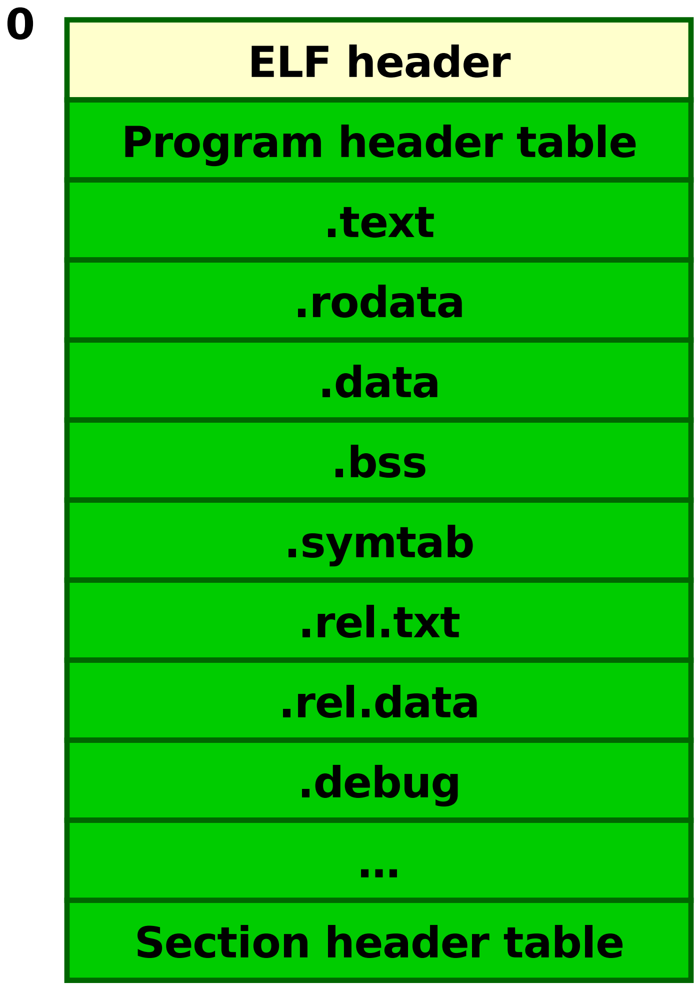
...

Section header table

ELF Header

Basic description of file contents:

- File format identifier
- Endianness
- Alignment requirement for other sections
- Location of other sections
- Code’s starting address
- …



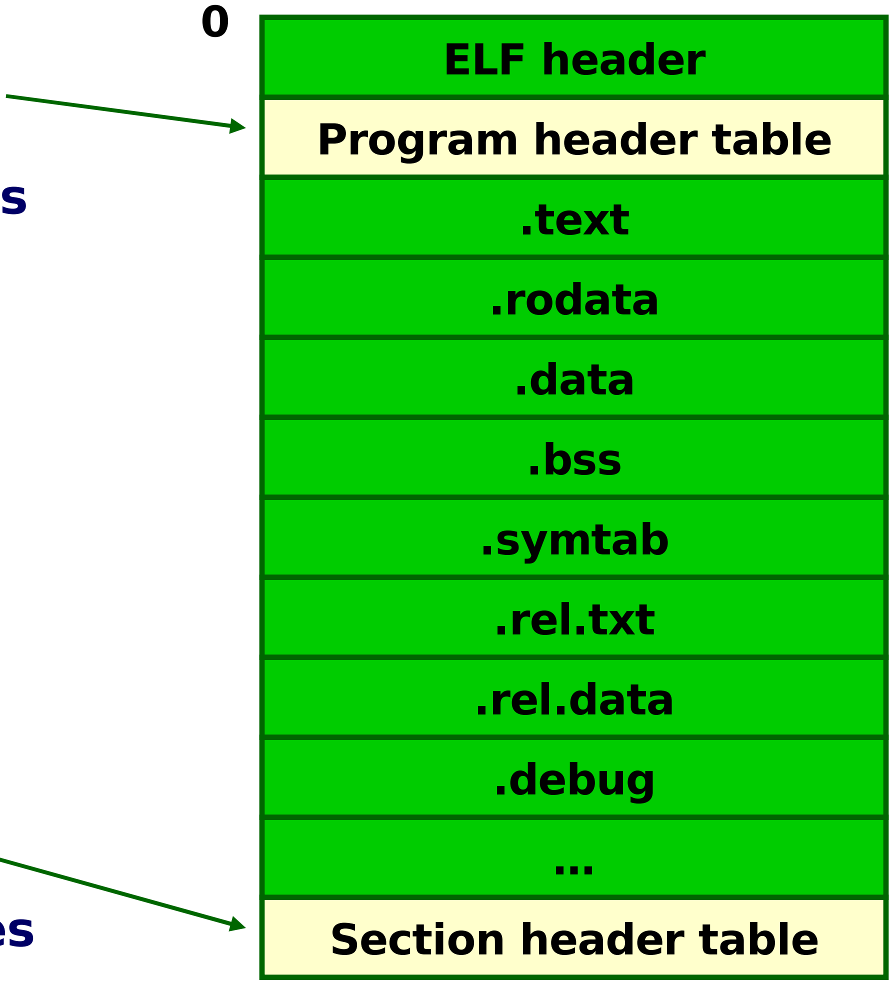
Program and Section Headers

Info about other sections necessary for loading

- Required for executables & libraries

Info about other sections necessary for linking

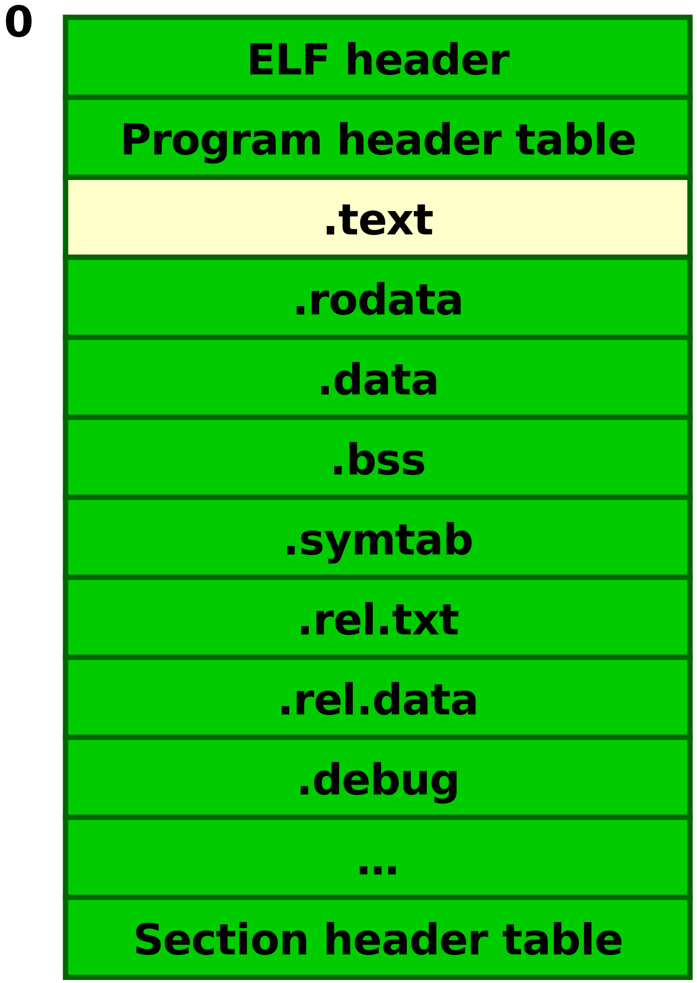
- Required for relocatables



Text Section

Code

- read-only



Data Sections

Static data

- initialized, read-only
- initialized, read/write
- uninitialized, read/write (BSS = "Block Started by Symbol" pseudo-op for IBM 704)

Initialized

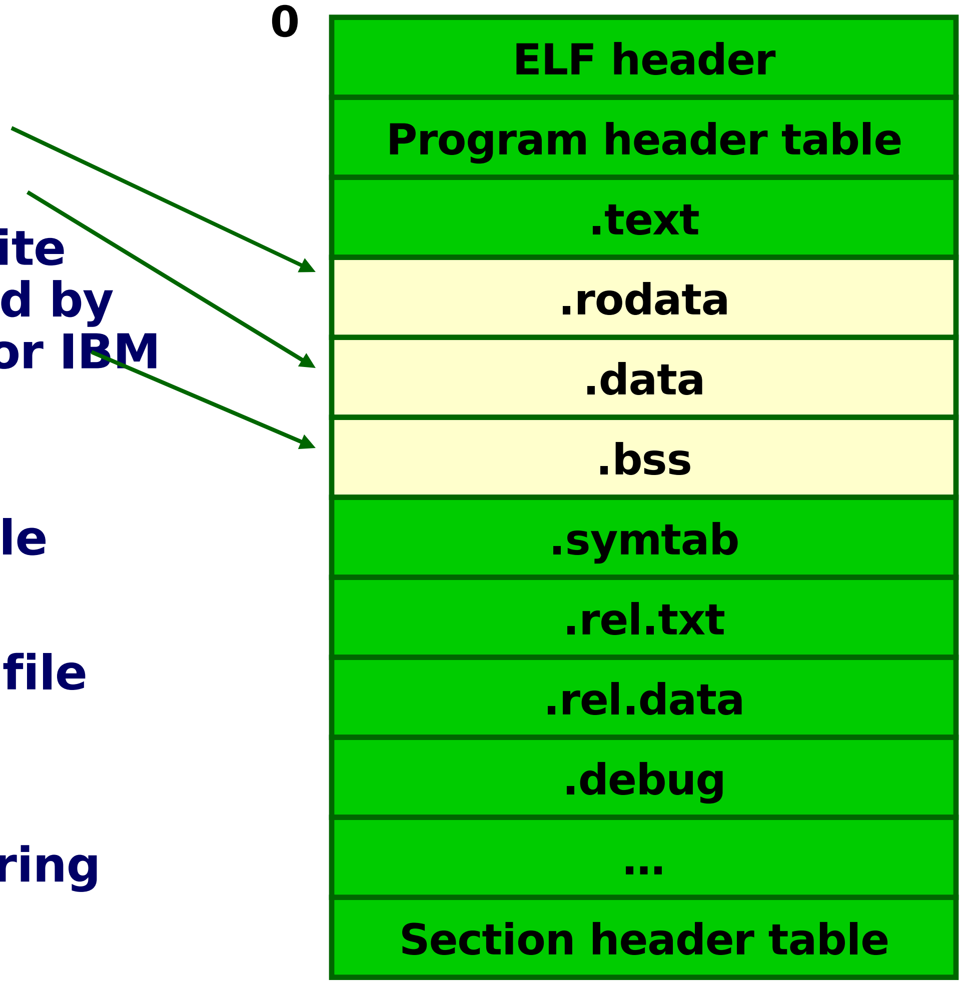
- Initial values in ELF file

Uninitialized

- Only total size in ELF file

Writable distinction enforced at run-time

- Why? Protection; sharing
- How? Virtual memory



0

ELF header

Program header table

.text

.rodata

.data

.bss

.symtab

.rel.txt

.rel.data

.debug

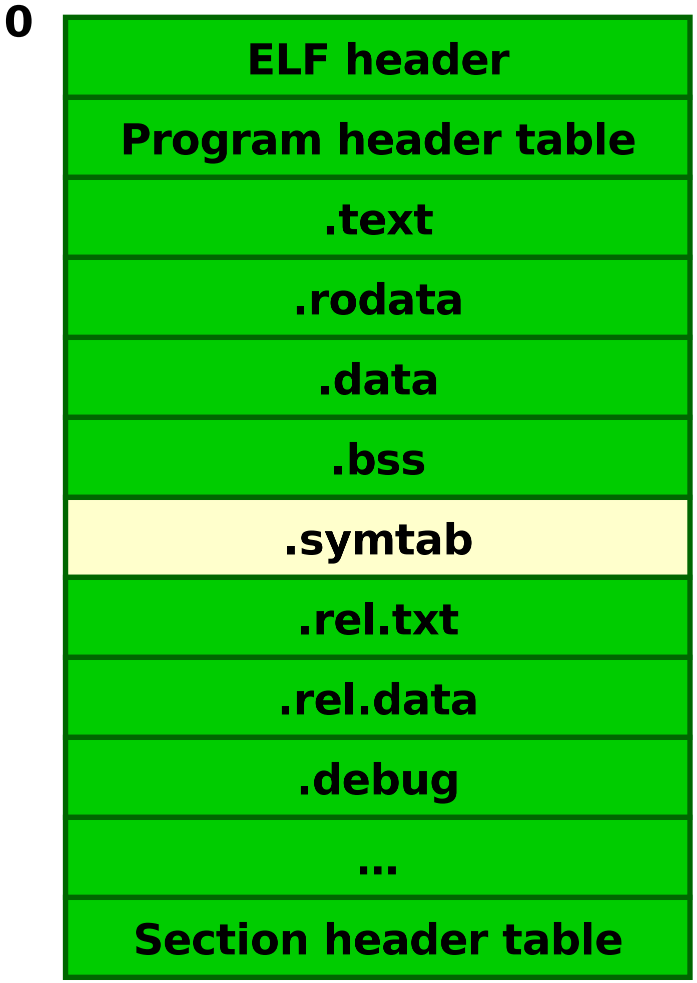
...

Section header table

Symbol Table

Describes where global variables and functions are defined

- **Present in all relocatable ELF files (not just in files compiled for debugging)**



0

ELF header

Program header table

.text

.rodata

.data

.bss

.symtab

.rel.txt

.rel.data

.debug

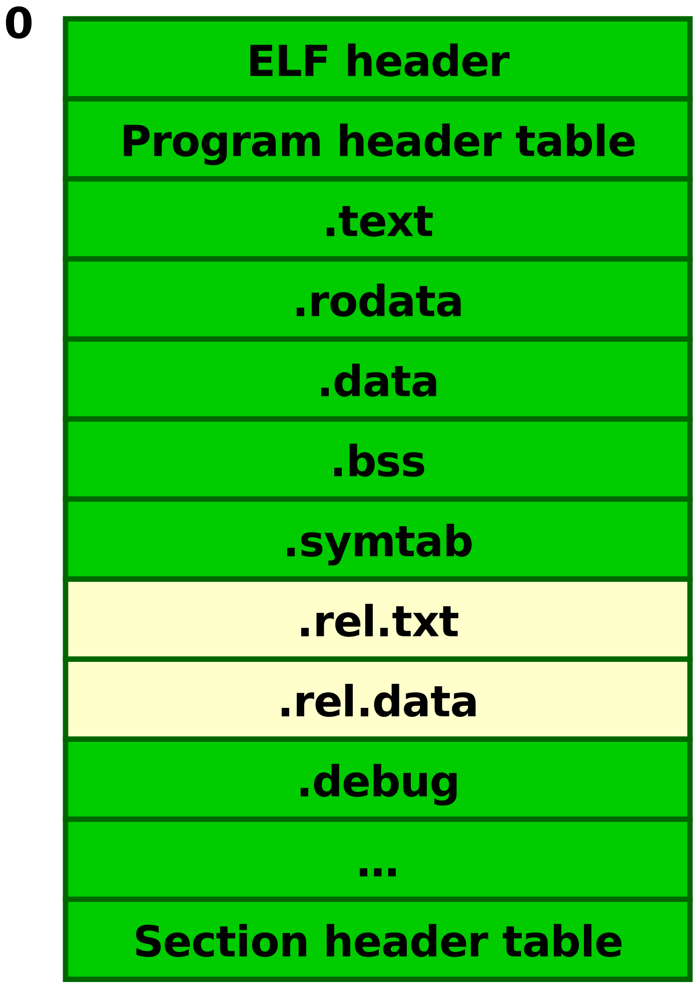
...

Section header table

Relocation Information

Describes where and how labels are used

- Allows object files to be easily relocated



0

ELF header

Program header table

.text

.rodata

.data

.bss

.symtab

.rel.txt

.rel.data

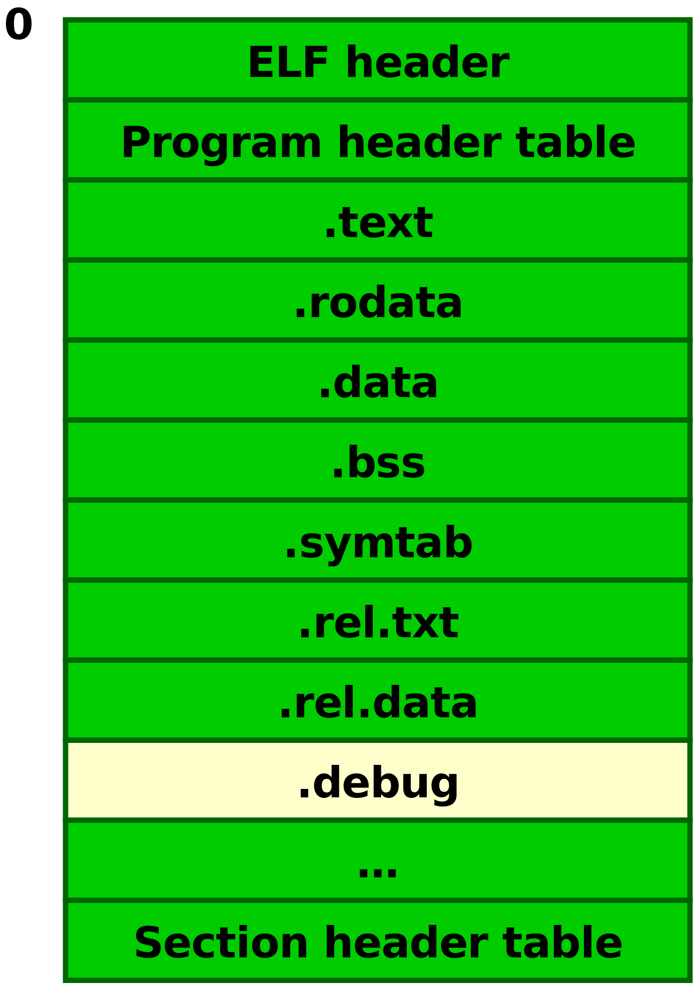
.debug

…

Section header table

Debug Section

Relates source code to the object code within the ELF file



0

ELF header

Program header table

.text

.rodata

.data

.bss

.symtab

.rel.txt

.rel.data

.debug

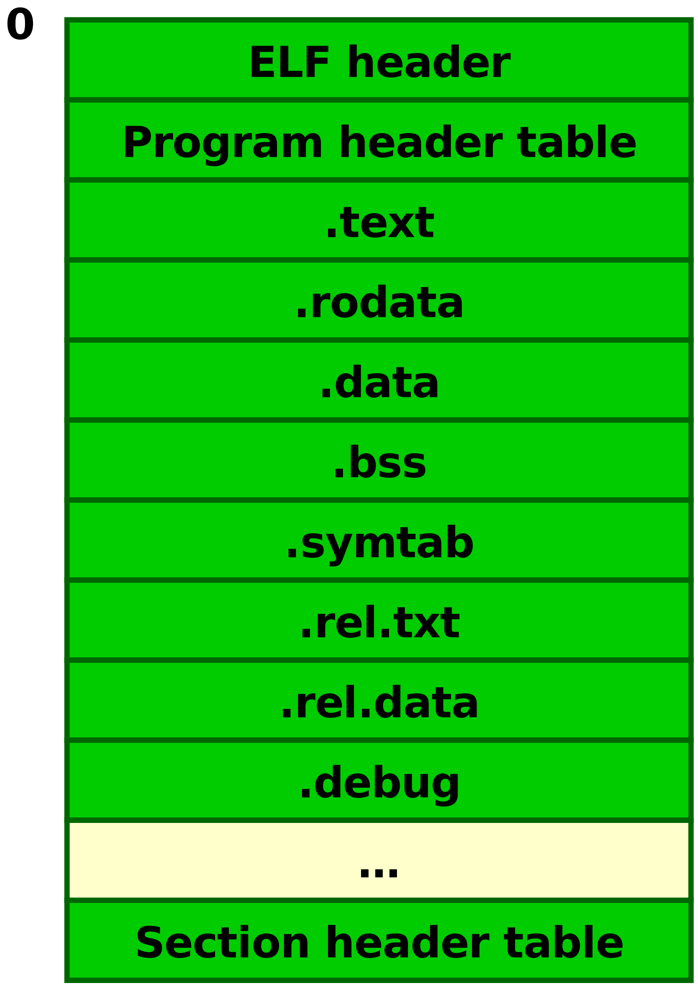
...

Section header table

Other Sections

Other kinds of sections also supported, including:

- Other debugging info
- Version control info
- Dynamic linking info
- C++ initializing & finalizing code



0

ELF header

Program header table

.text

.rodata

.data

.bss

.symtab

.rel.txt

.rel.data

.debug

...

Section header table

Linker Symbol Classification

Global symbols

- Symbols defined by module *m* that can be referenced by other modules
- C: non-`static` functions & global variables

External symbols

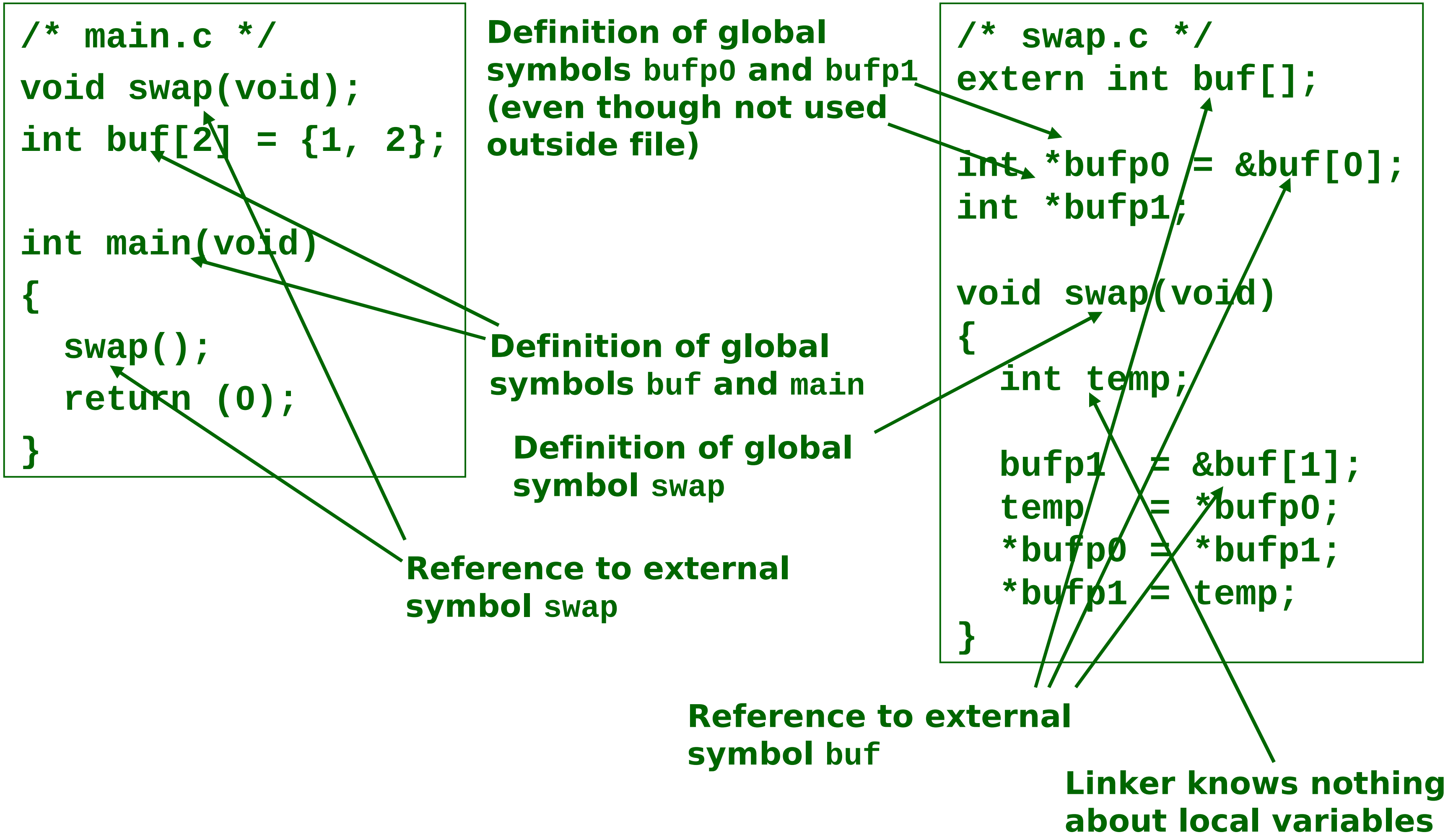
- Symbols referenced by module *m* but defined by some other module
- C: `extern` functions & variables

Local symbols

- Symbols that are defined and referenced exclusively by module *m*
- C: `static` functions & variables

Local linker symbols ≠ local program variables!

Linker Symbols



```
/* main.c */
void swap(void);
int buf[2] = {1, 2};

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

Definition of global symbols `bufp0` and `bufp1` (even though not used outside file)

Definition of global symbols `buf` and `main`

Definition of global symbol `swap`

Reference to external symbol `swap`

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

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

void swap(void)
{
  int temp;

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

Reference to external symbol `buf`

Linker knows nothing about local variables

Linking: Symbols

```
/* main.c */
void swap(void);
int buf[2] = {1, 2};

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

What’s missing?

- swap – where is it?

main is a 19-byte function located at offset 0 of section 1 (.text)

buf is a 8-byte object located at offset 0 of section 3 (.data)

swap is referenced in this file, but is undefined (UND)

use `readelf –S` to see sections

```
UNIX% gcc -O -c main.c
UNIX% readelf -s main.o

Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
…
     8: 0000000000000000    19 FUNC    GLOBAL DEFAULT    1 main
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND swap
    10: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 buf
```

Linking: Symbols

```
/* swap.c */
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

void swap(void)
{
  int temp;

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

What's missing?

- **buf – where is it?**

[illegible]
[illegible] alignment requirement

```
Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000    38 FUNC    GLOBAL DEFAULT    1 swap
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND buf
    10: 0000000000000008     8 OBJECT  GLOBAL DEFAULT  COM bufp1
    11: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 bufp0
```

Name Mangling

Other languages (i.e. Java and C++) allow overloaded methods

- Functions then have the same name but take different numbers/types of arguments
- How does the linker disambiguate these symbols?

Generate unique names through *mangling*

- Mangled names are compiler dependent
- Example: class “`Foo`”, method “`bar(int, long)`”:
    - `bar__3Fooil`
    - `_ZN3Foo3BarEil`
- Similar schemes are used for global variables, etc.

Linking Steps

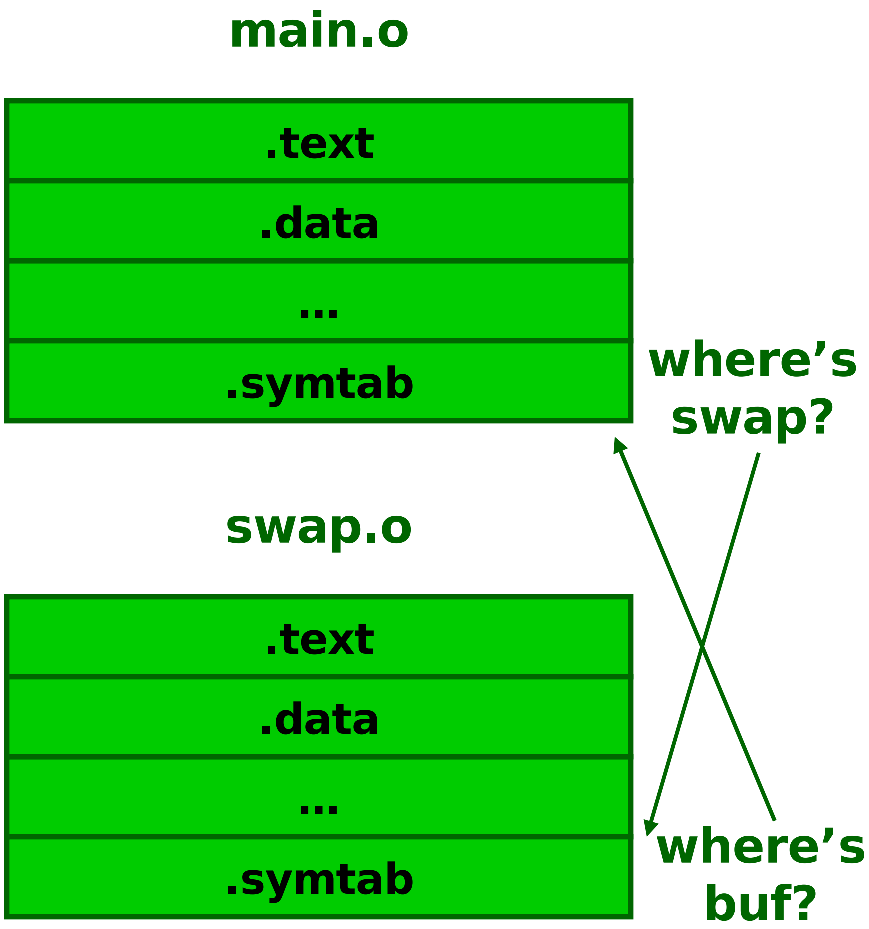
Symbol Resolution

- Determine where symbols are located and what size data/code they refer to

Relocation

- Combine modules, relocate code/data, and fix symbol references based on new locations

Symbol Resolution



main.o

.text

.data

…

.symtab

swap.o

.text

.data

…

.symtab

where’s swap?

where’s buf?

Undefined symbols must be resolved

- Where are they located
- What size are they?

Linker looks in the symbol tables of all relocatable object files

- Assuming every unknown symbol is defined once and only once, this works well

Linker Relocation

Once all symbols are resolved, must combine the input files

- Total code size is known
- Total data size is known
- All symbols must be assigned run-time addresses

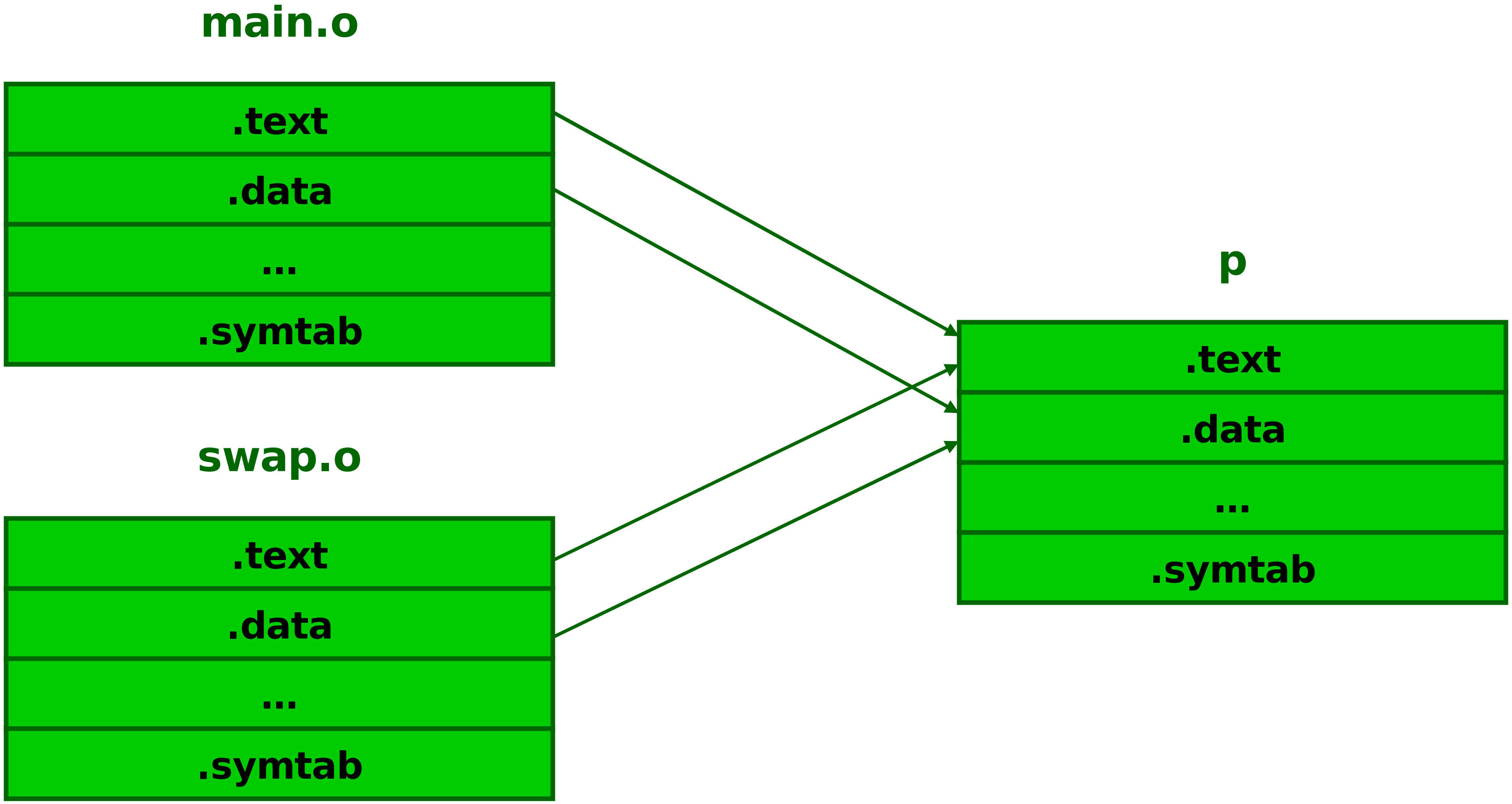
Sections must be merged

- Only one text, data, etc. section in final executable
- Final run-time addresses of all symbols are defined

Symbol references must be corrected

- All symbol references must now refer to their actual locations

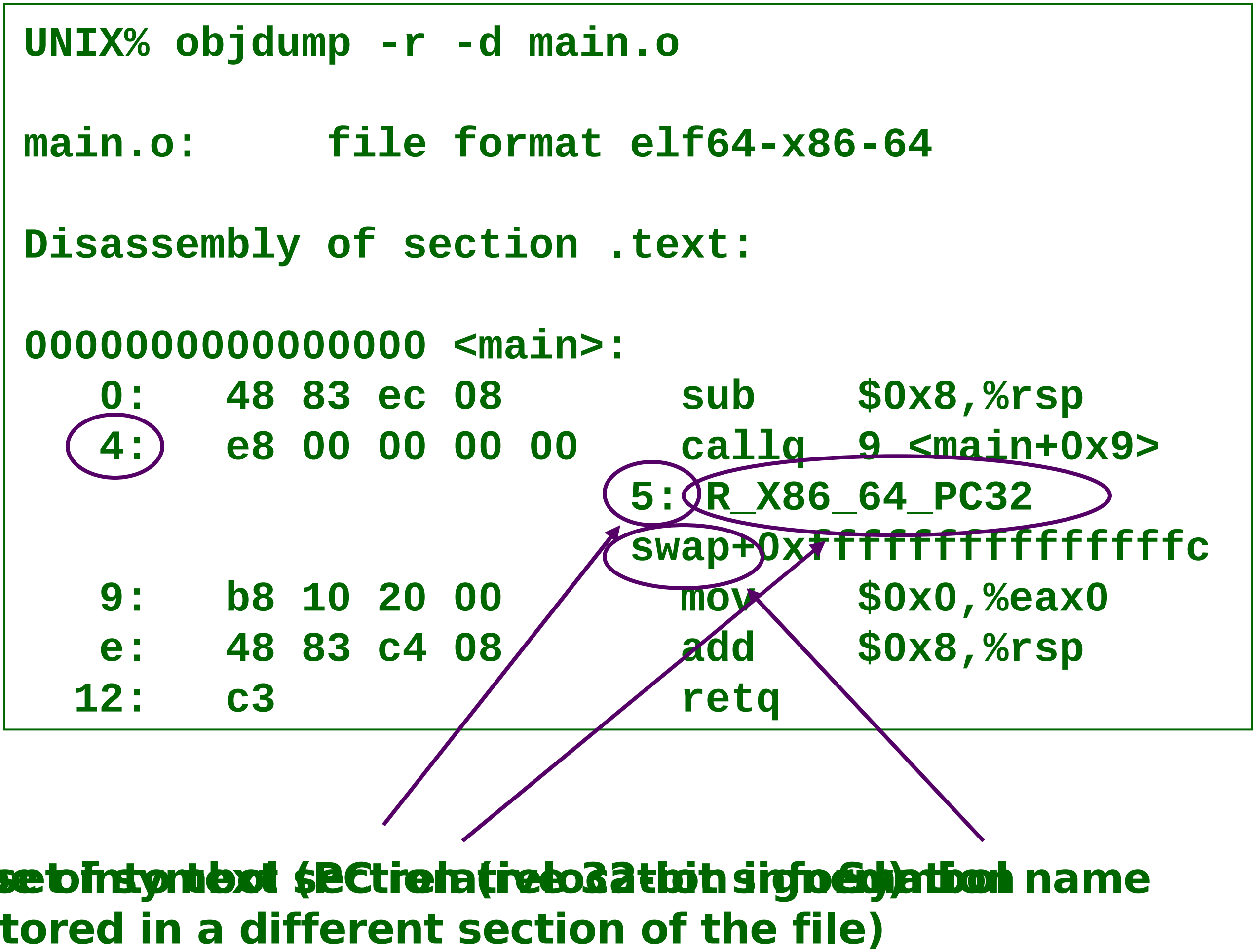
Relocation: Merging Files



Linking: Relocation

```
/* main.c */
void swap(void);
int buf[2] = {1, 2};

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

can also use `readelf –r` to see relocation information



```
UNIX% objdump -r -d main.o

main.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   48 83 ec 08          sub    $0x8,%rsp
   4:   e8 00 00 00 00       callq  9 <main+0x9>
                        5: R_X86_64_PC32
                        swap+0xfffffffffffffffc
   9:   b8 10 20 00          mov    $0x0,%eax0
   e:   48 83 c4 08          add    $0x8,%rsp
  12:   c3                   retq
```

Offset into text section (relocation info is stored in a different section of the file)

Type of symbol (PC relative 32-bit signed)

Symbol name

Linking: Relocation

```
/* swap.c */
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
  int temp;

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

```
UNIX% objdump -r -D swap.o

swap.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <swap>:
   0:   48 c7 05 00 00 00 00 movq $0x0,0(%rip)
   7:   00 00 00 00
                    3: R_X86_64_PC32
                    bufp1+0xfffffffffffffff8
                    7: R_X86_64_32S buf+0x4

   <..snip..>
Disassembly of section .data:

0000000000000000 <bufp0>:
        ...
                        0: R_X86_64_64 buf
```

Need relocated address of buf[]

Need to initialize bufp0 with &buf[0] (== buf)

After Relocation

```
0000000000000000 <main>:
   0:    48 83 ec 08          sub    $0x8,%rsp
   4:    e8 00 00 00 00       callq  9 <main+0x9>
                        5: R_X86_64_PC32 swap+0xfffffffffffffffc
   9:    b8 10 20 00          mov    $0x0,%eax0
   e:    48 83 c4 08          add    $0x8,%rsp
  12:    c3                   retq
```

```
0000000000400448 <main>:
  400448:    48 83 ec 08          sub    $0x8,%rsp
  40044c:    e8 0b 00 00 00       callq  40045c <swap>
  400451:    b8 10 20 00          mov    $0x0,%eax0
  400456:    48 83 c4 08          add    $0x8,%rsp
  40045a:    c3                   retq
```

After Relocation

```
0000000000000000 <swap>:
   0:   48 c7 05 00 00 00 00 movq $0x0,0(%rip)
   7:   00 00 00 00
                    3: R_X86_64_PC32 bufp1+0xfffffffffffffff8
                    7: R_X86_64_32S  buf+0x4
   <..snip..>
0000000000000000 <bufp0>:
        ...
                    0: R_X86_64_64 buf
```

```
000000000040045c <swap>:
  40045c:   48 c7 05 01 04 20 00 movq $0x600848,2098177(%rip)
  400463:   48 08 60 00                               # 600868 <bufp1>
   <..snip..>
0000000000600850 <bufp0>:
  600850:   44 08 60 00 00 00 00 00
```

Problem: Undefined Symbols

```
UNIX% gcc -O2 -o p main.c
/tmp/cccpTy0d.o: In function `main’:
main.c:(.text+0x5): undefined reference to `swap’
collect2: ld returned 1 exit status
UNIX%
```

Missing symbols are not compiler errors

- **May be defined in another file**
- **Compiler just inserts an undefined entry in the symbol table**

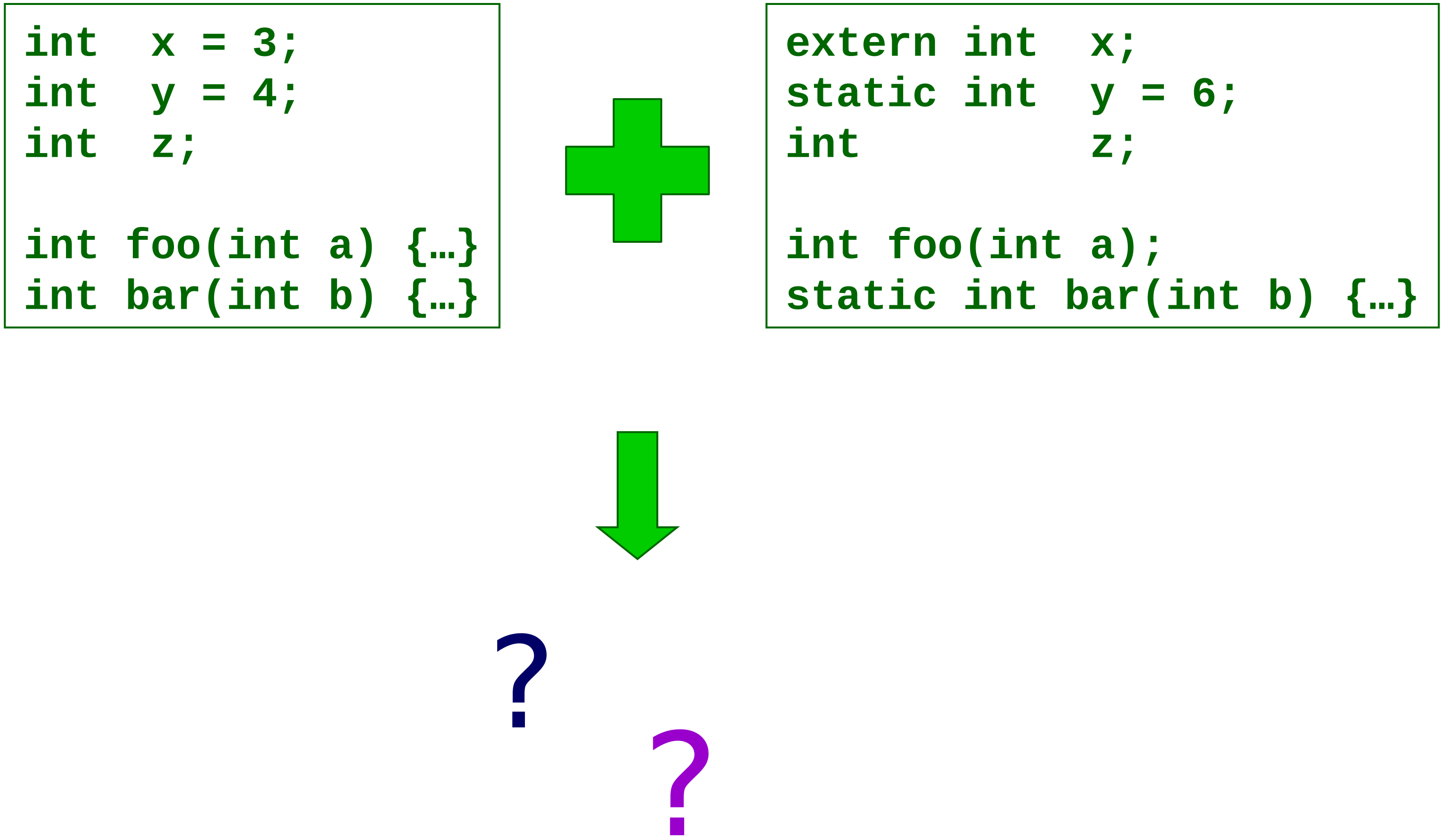
During linking, any undefined symbols that cannot be resolved cause an error

Problem: Multiply Defined Symbols

Different files could define the same symbol

- Is this an error?
- If not, which one should be used? One or many?

Linking: Example

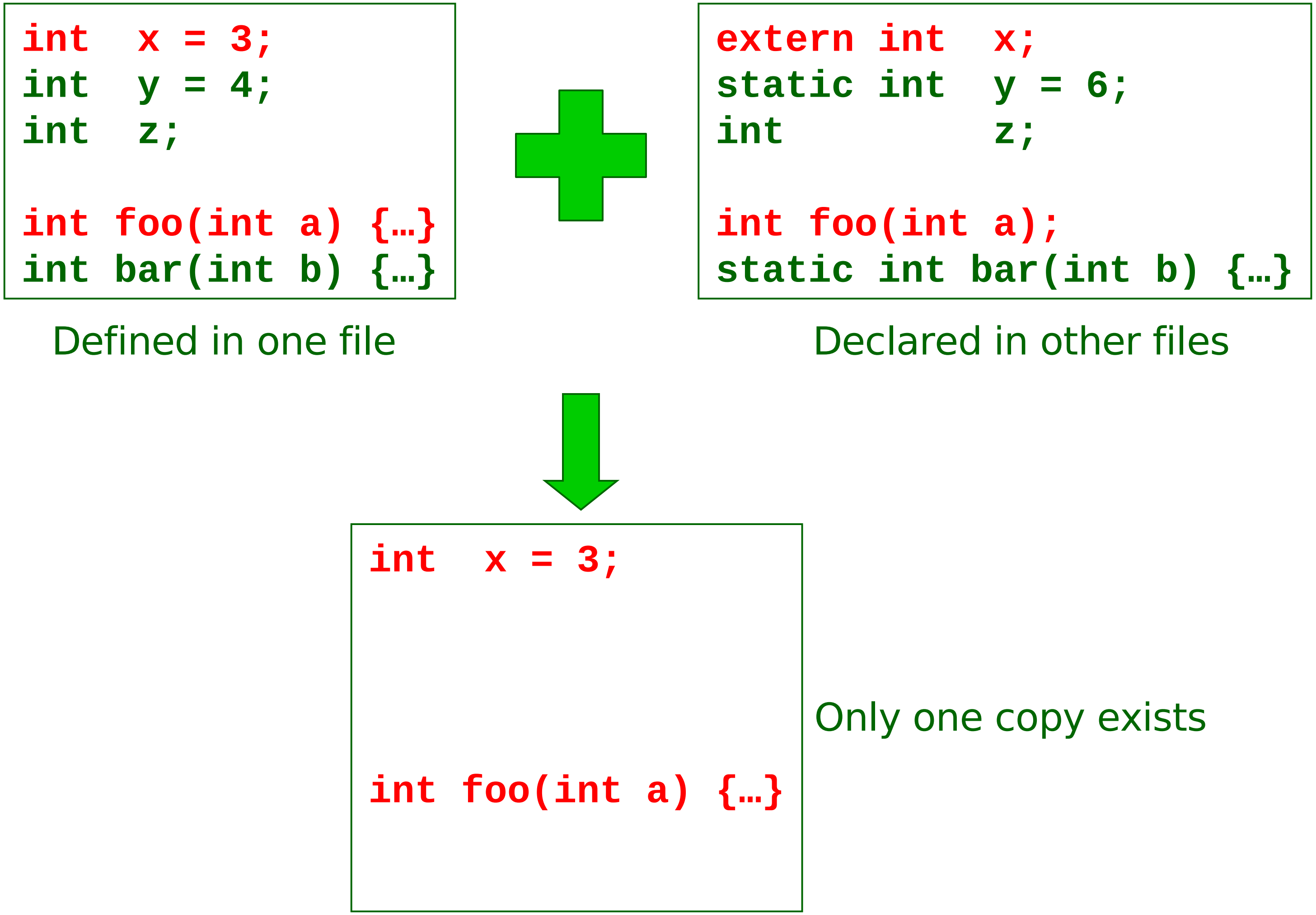


```
int   x = 3;
int   y = 4;
int   z;

int foo(int a) {…}
int bar(int b) {…}
```

```
extern int   x;
static int   y = 6;
int          z;

int foo(int a);
static int bar(int b) {…}
```

Note: Linking uses object files
Examples use source-level for convenience

Linking: Example



```
int   x = 3;
int   y = 4;
int   z;

int foo(int a) {…}
int bar(int b) {…}
```

Defined in one file

```
extern int   x;
static int   y = 6;
int          z;

int foo(int a);
static int bar(int b) {…}
```

Declared in other files

```
int   x = 3;




int foo(int a) {…}
```

Only one copy exists

Linking: Example

```
int  x = 3;
int  y = 4;
int  z;

int foo(int a) {…}
int bar(int b) {…}
```

+

```
extern int  x;
static int  y = 6;
int         z;

int foo(int a);
static int bar(int b) {…}
```

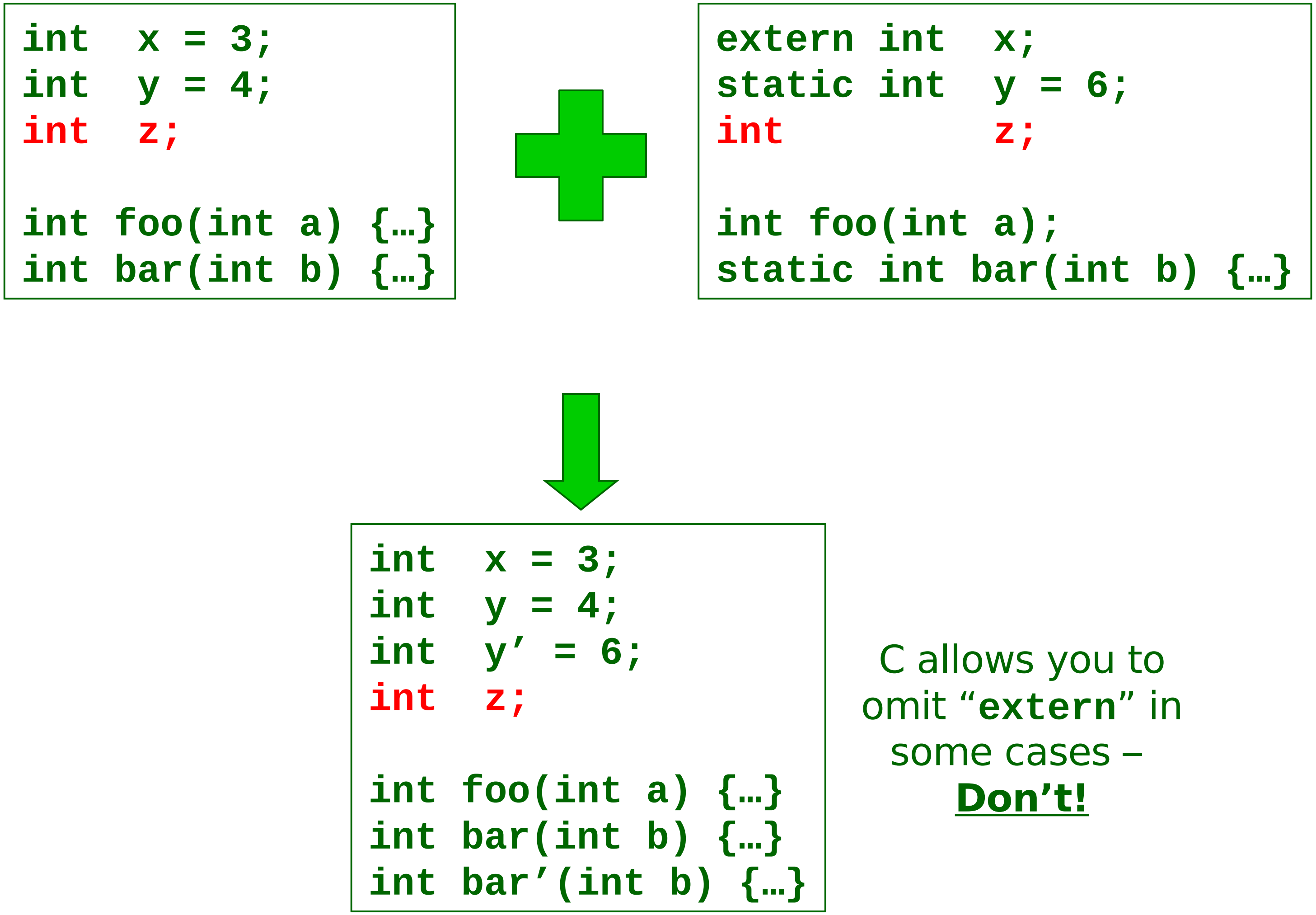
Private names not in symbol table.
Can’t conflict with other files’ names

↓

```
int  x = 3;
int  y = 4;
int  y’ = 6;


int foo(int a) {…}
int bar(int b) {…}
int bar’(int b) {…}
```

Renaming is a convenient source-level way to understand this

Linking: Example



```
int  x = 3;
int  y = 4;
int  z;

int foo(int a) {…}
int bar(int b) {…}
```

```
extern int  x;
static int  y = 6;
int         z;

int foo(int a);
static int bar(int b) {…}
```

```
int  x = 3;
int  y = 4;
int  y’ = 6;
int  z;

int foo(int a) {…}
int bar(int b) {…}
int bar’(int b) {…}
```

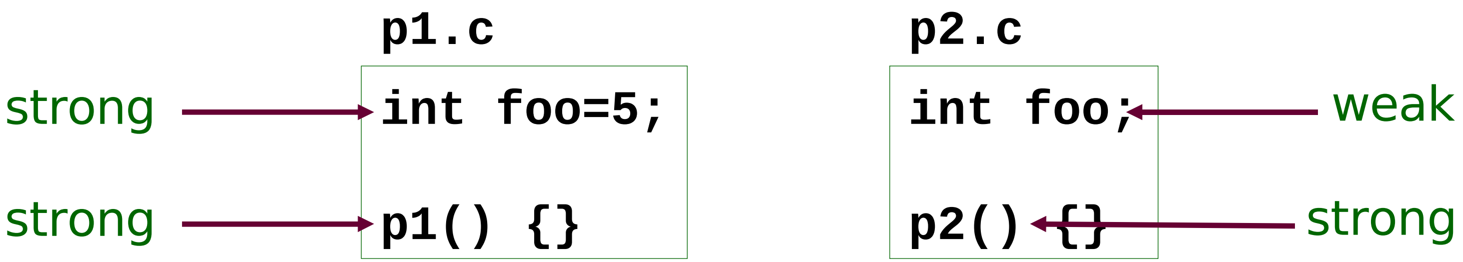
C allows you to omit “**extern**” in some cases – **Don’t!**

Strong & Weak Symbols

Program symbols are either strong or weak

strong **procedures & initialized globals**

weak **uninitialized globals**



p1.c

```
int foo=5;

p1() {}
```

p2.c

```
int foo;

p2() {}
```

Strong & Weak Symbols

A strong symbol can only appear once

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

- References to the weak symbol resolve to the strong symbol

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

Linker Puzzles: What Happens?

```
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** will overwrite **y**!
Nasty!

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

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

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

Nightmare scenario: replace **int** with a **struct** type, compile each file with different alignment rules

Libraries

How should functions commonly used by programmers be provided?

- Math, I/O, memory management, string manipulation, etc.
- 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 object files into their programs
  - More efficient, but burdensome on the programmer

Solution: static libraries (.a archive files)

- Multiple relocatable files + index → single archive file
- Only links the subset of relocatable files from the library that are used in the program
- Example: `gcc –o fpmath main.c float.c -lm`

Two Common Libraries

`libc.a` (the C standard library)

- 4 MB archive of 1395 object files
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- Usually automatically linked

`libm.a` (the C math library)

- 1.3 MB archive of 401 object files
- floating point math (sin, cos, tan, log, exp, sqrt, …)
- Use “`-lm`” to link with your program

```
UNIX% ar t /usr/lib64/libc.a
…
fprintf.o
…
feof.o
…
fputc.o
…
strlen.o
…
```

```
UNIX% ar t /usr/lib64/libm.a
…
e_sinh.o
e_sqrt.o
e_gamma_r.o
k_cos.o
k_rem_pio2.o
k_sin.o
k_tan.o
…
```

Creating a Library

```
/* vector.h */
void addvec(int *x, int *y, int *z, int n);
void multvec(int *x, int *y, int *z, int n);
```

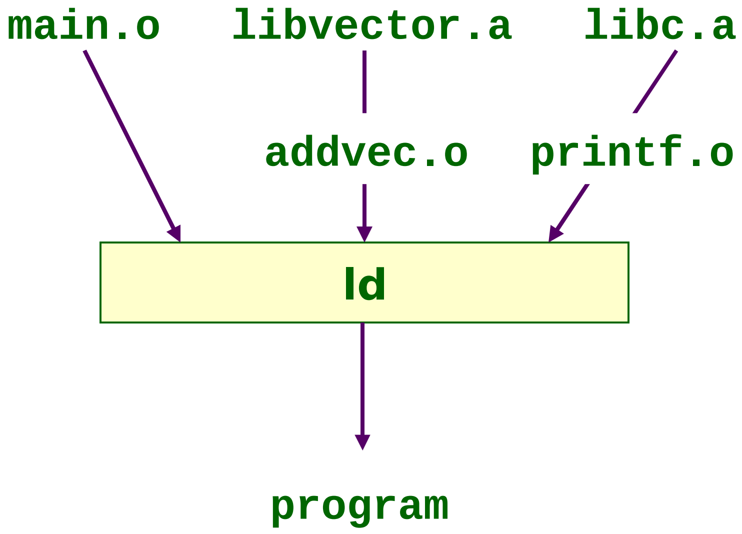
```
/* addvec.c */
#include “vector.h”
void addvec(int *x, int *y,
            int *z, int n)
{
  int i;

  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];
}
```

```
/* multvec.c */
#include “vector.h”
void multvec(int *x, int *y,
             int *z, int n)
{
  int i;

  for (i = 0; i < n; i++)
    z[i] = x[i] * y[i];
}
```

```
UNIX% gcc –c addvec.c multvec.c
UNIX% ar rcs libvector.a addvec.o multvec.o
```

Using a library

```
/* main.c */
#include <stdio.h>
#include “vector.h”

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(void)
{
  addvec(x, y, z, 2);
  printf(“z = [%d %d]\n”, z[0], z[1]);
  return (0);
}
```



```
UNIX% gcc –O2 –c main.c
UNIX% gcc –static –o program main.o ./libvector.a
```

How to Link: Basic Algorithm

Keep a list of the current unresolved references.

For each object file (.o and .a) in command-line order

- **Try to resolve each unresolved reference in list to objects defined in current file**
- **Try to resolve each unresolved reference in current file to objects defined in previous files**
- **Concatenate like sections (.text with .text, etc.)**

If list empty, output executable file, else error

Problem: Command line order matters! Link libraries last:

```
UNIX% gcc main.o libvector.a
UNIX% gcc libvector.a main.o
main.o: In function `main':
main.o(.text+0x4): undefined reference to `addvec'
```

Dynamic Libraries

Static	Dynamic
Linked at compile-time UNIX: foo.a	Linked at run-time UNIX: foo.so
Relocatable ELF File	Shared ELF File

What are the differences?

Static & Dynamic Libraries

Static

- Library code added to executable file
- Larger executables
- Must recompile to use newer libraries
- Executable is self-contained
- Some time to load libraries at compile-time
- Library code shared only among copies of same program

Dynamic

- Library code not added to executable file
- Smaller executables
- Uses newest (or smallest, fastest, …) library without recompiling
- Depends on libraries at run-time
- Some time to load libraries at run-time
- Library code shared among all uses of library

Static & Dynamic Libraries

Static

Creation

```
ar r libfoo.a bar.o baz.o
ranlib libfoo.a
```

Use

```
gcc –o zap zap.o -lfoo
```

Adds library’s code, data, symbol table, relocation info, …

Dynamic

Creation

```
gcc –shared –Wl,-soname,libfoo.so
 -o libfoo.so bar.o baz.o
```

Use

```
gcc –o zap zap.o -lfoo
```

Adds library’s symbol table, relocation info

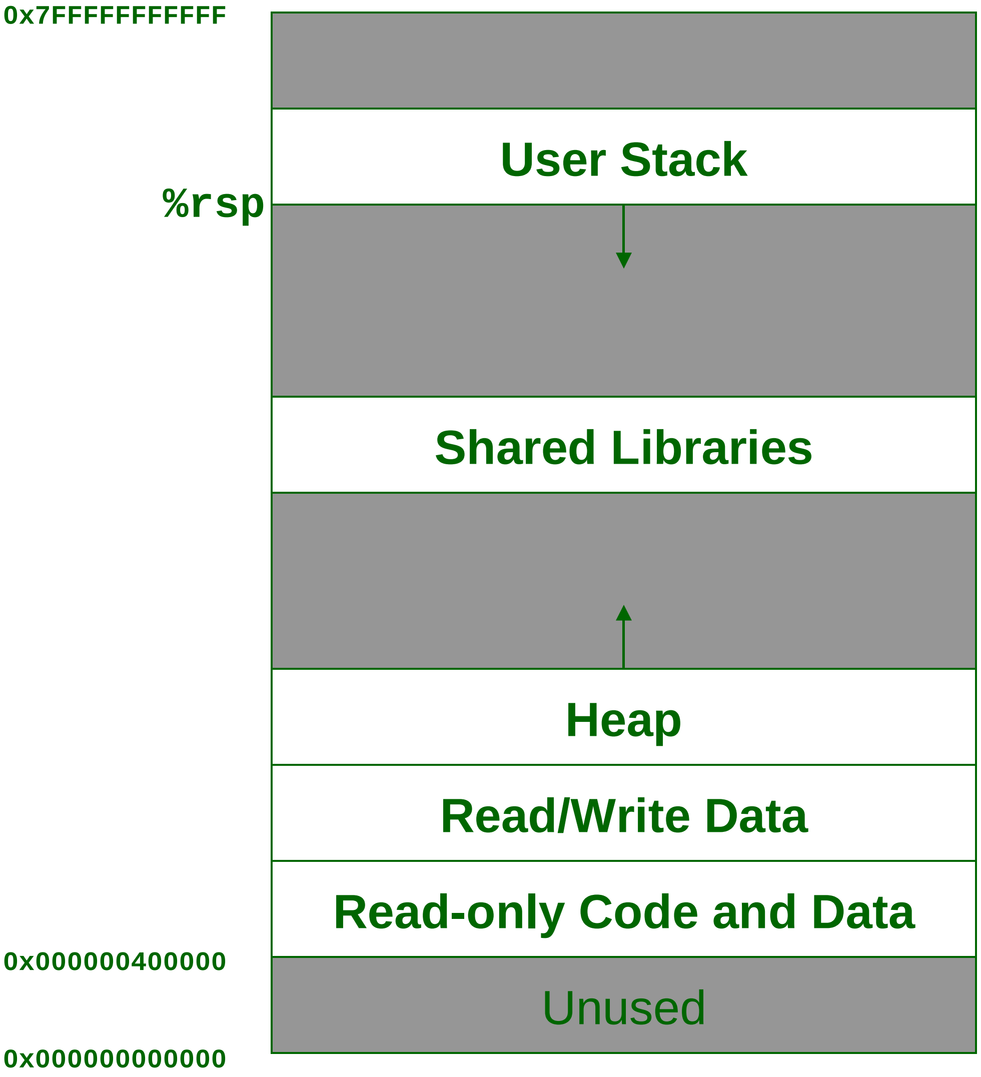
Loading

Linking yields an executable that can actually be run

Running a program

- `unix% ./program`
- Shell does not recognize "`program`" as a shell command, so assumes it is an executable
- Invokes the *loader* to load the executable into memory (any unix program can invoke the loader with the `execve` function – more later)

Creating the Memory Image (sort of…)



Create code and data segments

- Copy code and data from executable into these segments

Create initial heap segment

- Grows up from read/write data

Create stack

- Starts near the top and grows downward

Call dynamic linker to load shared libraries and relocate references

Starting the Program

Jump to program’s entry point (stored in ELF header)

- **For C programs, this is the `_start` symbol**

Execute `_start` code (from `crt1.o` – same for all C programs)

- `call __libc_init_first`
- `call _init`
- `call atexit`
- `call main`
- `call _exit`

Position Independent Code

- **Static libraries compile with unresolved global & local addresses**
  - **Library code & data concatenated & addresses resolved when linking**

Position Independent Code

By default (in C), dynamic libraries compile with <u>resolved</u> global & local addresses

- E.g., `libfoo.so` starts at 0x400000 in every application using it
- Advantage: Simplifies sharing
- Disadvantage: Inflexible – must decide ahead of time where each library goes, otherwise libraries can conflict

Position Independent Code

Can compile dynamic libraries with <u>unresolved</u> global & local addresses

- `gcc –shared –fPIC …`
- **Advantage: More flexible – no conflicts**
- **Disadvantage: Code less efficient – referencing these addresses involves indirection**

Library Interpositioning

Linking with non-standard libraries that use standard library symbols

- "Intercept" calls to library functions

Some applications:

- Security
  - Confinement (sandboxing)
  - Behind the scenes encryption
    - Automatically encrypt otherwise unencrypted network connections
- Monitoring & Profiling
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - malloc tracing
    - Detecting memory leaks
    - Generating malloc traces

Dynamic Linking at Run-Time

Application access to dynamic linker via API:

```
#include <dlfcn.h>

void
dlink(void)
{
  void *handle = dlopen(“mylib.so”, RTLD_LAZY);

  /* type */ myfunc = dlsym(handle, “myfunc”);
  myfunc(…);
  dlclose(handle);
}
```

Symbols resolved at first use, not now

Error-checking omitted for clarity

Next Time

Exceptions