CS 33 Week 10

Section 1G, Spring 2015
Prof. Eggert (TA: Eric Kim)
v1.0

Announcements

- Lab 4 was due Wednesday
  - Last chance to submit: tonight (Friday), 11:55 PM
- Final Exam on Thursday (June 11th)
  - Break a leg!

Final Exam

- Thursday, June 11, 2015, 8:00am-11:00am
  - Come ~10 minutes early.
- Final Examination Code: 15
- **Location**: Ackerman Grand Ballroom
- Cumulative (emphasis on post-MT2)

http://smbc-comics.com/index.php?id=2999

Study Resources

TA pages:

Eric: www.eric-kim.net/cs33_page

Uen-Tao: http://www.seas.ucla.edu/~uentao/

He posts his lecture notes too!

Brandon: http://www.cs.ucla.edu/~brandonwu/

Textbook!

Google

Overview

- Linking
- Exceptions/Errors

Compilation Pipeline

```
cpp code.c code.i // C Preprocessor -> code.i
gcc -S code.i     // Compile C -> code.s
gcc -c code.s     // Assemble code.s -> code.o
gcc code.o        // Link objfile to make executable
```

Demo: Show compilation of simple C program

Object Files

```
$ gcc -c code.c    // Compile: outputs objfile code.o
```

Aka "Relocatable Object Files".

On unix systems, .o files are in ELF format ("Executable and Linkable Format").

Are binary objects that contain several sections:

header, text, rodata, data, bss, symtable, rel_text, rel_data, debug, line, strtab, footer

Object Files: readelf

Cool trick: Use the `readelf` program to inspect object files!

```
$ readelf -a code.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 code.c
...
```

Object Files: readelf

Can also use readelf on executable files:

```
$ readelf -a a.out

$ readelf -s code.o // only show symbol table
```

Linking

```
gcc code.o          // Link objfile to make executable
```

Generates an executable. Two main steps:
(1) Symbol resolution
(2) Relocation

Linking: Demo

Demo: Show how to use CSAPP functions in my C programs.

Linking and header files (.h)

```
// file1.c
#include <stdio.h>
#include "mything.h"

int main() {
    float r=fn(3, PI_);
    printf("%f\n",r);
    return 0;
}
```

```
// mything.h
double PI_;
float fn(int,double);
```

```
// mything.c
double PI_ = 3.1415;
float fn(int x, double y) {
    return x + 2*y;
}
```

Question: How to create executable?

Answer:

$ gcc -c file1.c
$ gcc -c mything.c
$ gcc -o file1 file1.o mything.o

Linking and header files (.h)

```
// file1.c
#include <stdio.h>
#include "mything.h"

int main() {
    float r=fn(3, PI_);
    printf("%f\n",r);
    return 0;
}
```

```
// mything.h
double PI_;
float fn(int,double);
```

```
// mything.c
double PI_ = 3.1415;
float fn(int x, double y) {
    return x + 2*y;
}
```

Question: Why did we not have to mention mything.h?

Answer: C preprocessor copied body of mything.h into file1.c, ie check out:

$ cpp file1.c file1.i

Linking and header files (.h)

```
// file1.c
#include <stdio.h>
#include "mything.h"

int main() {
    float r=fn(3, PI_);
    printf("%f\n",r);
    return 0;
}
```

```
// mything.h
double PI_;
float fn(int,double);
```

```
// mything.c
double PI_ = 3.1415;
float fn(int x, double y) {
    return x + 2*y;
}
```

Demo: Let's look at the generated symbol tables for file1.o and mything.o
In file1.o, we see that PI_ is declared as COM (uninitialized global var), and fn is declared as UNDEF.

Linking and header files (.h)

```
// file1.c
#include <stdio.h>
#include "mything.h"
int main() {
    float r=fn(3, PI_);
    printf("%f\n",r);
    return 0;
}
```

Question: Why does file1.c compile, but file1b.c not compile? What step of compilation process does file1b.c fail?

```
// file1b.c
#include <stdio.h>
int main() {
    float r=fn(3, PI_);
    printf("%f\n",r);
    return 0;
}
```

Answer: In file1b.c, the symbols PI_,fn are not declared anywhere. This is caught by the assembler, ie:

$ gcc -S file1b.c

ERROR

Linking and header files (.h)

```
// file1.c
#include <stdio.h>
#include "mything.h"
int main() {
    float r=fn(3, PI_);
    printf("%f\n",r);
    return 0;
}
```

```
// mything.h
double PI_;
```

```
// mything.c
double PI_ = 3.1415;
float fn(int x, double y) {
    return x + 2*y;
}
```

Question: What if we removed declaration of fn() in mything.h?

Answer: Still compiles! But get wrong answer: C assumes int return value for undeclared functions that are found at link time: http://stackoverflow.com/questions/9780930/correct-answer-before-return-incorrect-after-return

(Really weird!)

Mismatched function prototypes

For fun reading, check out this post that explores what happens when a function prototype disagrees with the actual function definition:

http://stackoverflow.com/questions/15137702/function-prototype-in-header-file-doesnt-match-definition-how-to-catch-this

Static vs Dynamic Linking

Two ways to link a program, each with its pros and cons.

Static Linking

Specify object file at compilation time. Ex: Here, mything.o is statically linked to file1:

```
$ gcc file1.o mything.o
```

For larger projects:

```
$ gcc -static vlc.o libpng.a libmpg.a libmp4.a libmp3.a …
```

Creating Libraries: ar

Suppose I have two files: add.c, mult.c

Can package their .o files into a single .a file:

```
$ gcc -c add.c mult.c
$ ar rcs libek.a add.o mult.o
```

(libek.a is basically add.o concatenated with mult.o)

Can use it later during linking:

```
$ gcc -o main main.c ./libek.a
```

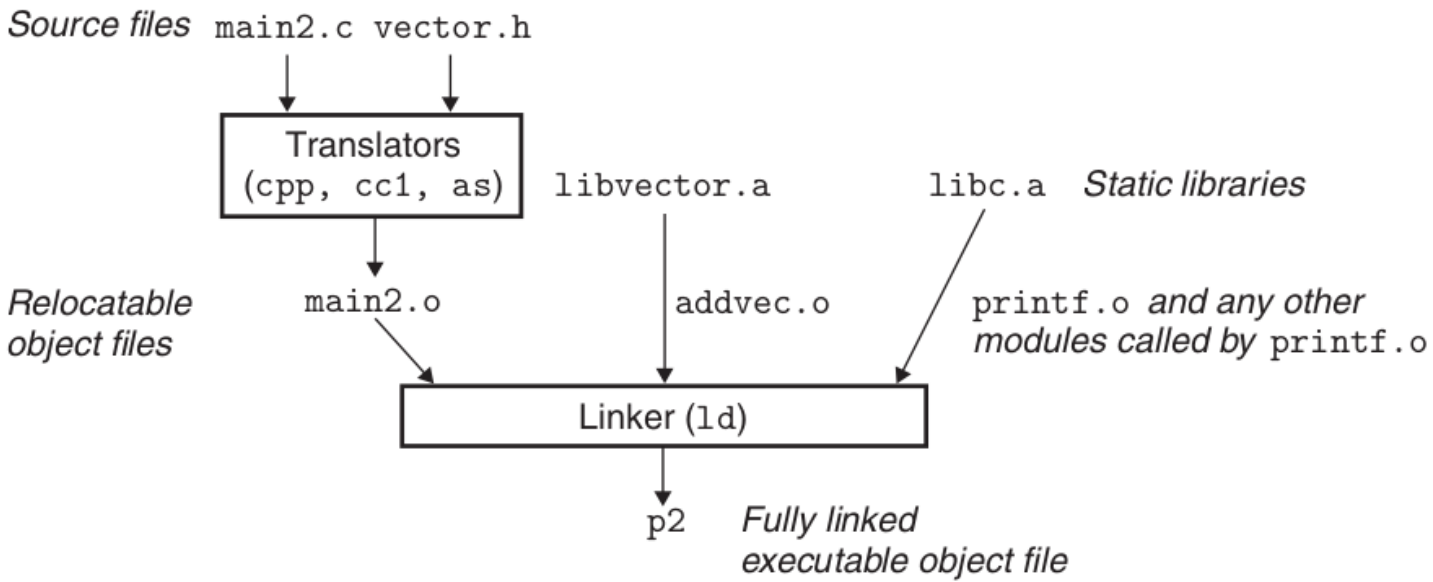


Figure 7.7 **Linking with static libraries.**

Demo [week10_slink]: add.c, mult.c, main.c, etc.
./main is statically linked

Question: Why is ./main much larger than ./mains?

Static Linking: Pros/Cons

Pros

- Consistent behavior across systems.
- Simple!

Cons

- Larger executable size. Ex: libc.a is ~8MB!
- If a library has an update (libmp3.a), then we have to re-link the project to include the updated library.
- For common libs (ie libc), each process has a copy in memory. Wasteful!
- Can't share libraries with other processes (discuss later)

Dynamic Linking

Static: Load all libraries into executable during compile time.

Dynamic: Load library(s) into executable while loading program!

Has a number of benefits.

Shared Libraries: .so and .dll

By design, shared (ie dynamic) libraries are loaded during runtime, when OS loads program into memory.

.so: "Shared Object" [unix]

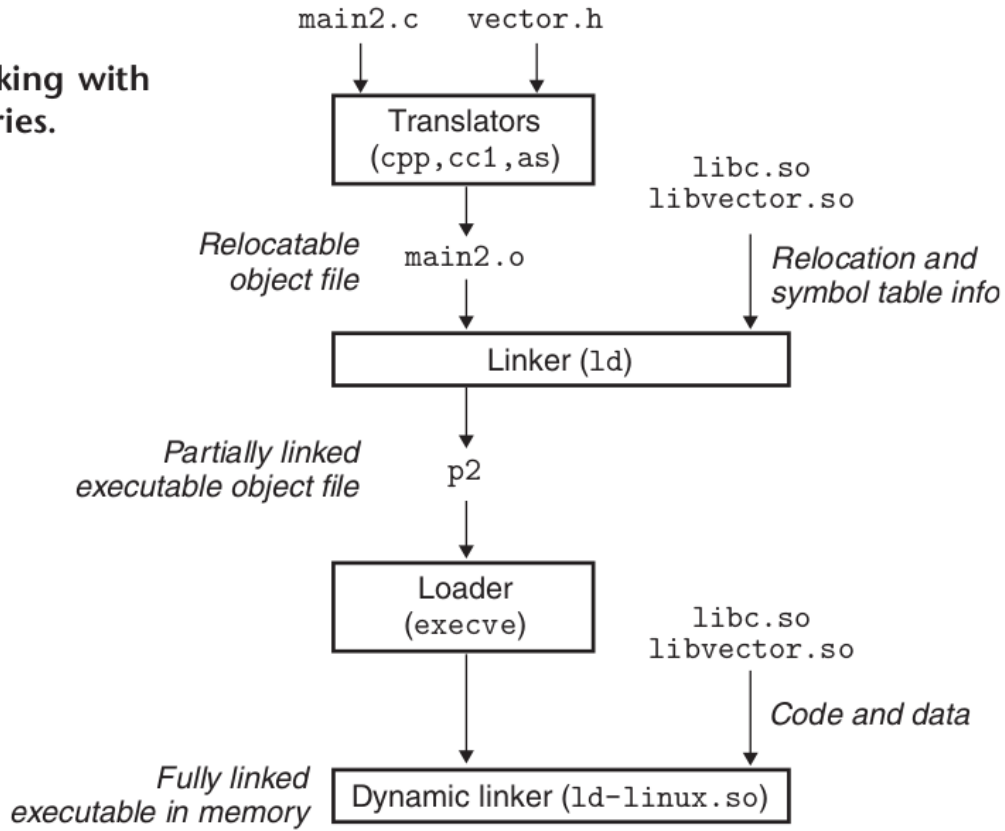
.dll: "Dynamic Link Library" [windows]

(In unix, can create shared libraries with gcc!)

Dynamic Linker

As OS loads program into memory, if it notices program refers to dynamic library (.interp, . dynsym, etc), then it will invoke dynamic linker.

Figure 7.15
Dynamic linking with shared libraries.



"DLL Hell"

Shared libs are now decoupled from executable.

Scenario: I download VLC 2.1.0, which assumed libmpeg2 (v1.1). Then, I upgrade libmpeg2 to v1.2, because 1.2 is much faster and less buggy.

Problem: VLC might now be broken!

DLL Hell!

Dynamic Linking: Pros/Cons

Pros

Smaller executable.

Processes can share libraries, save memory!

Cons

Possible instabilities ("DLL Hell")

More complicated behavior during load time (due to dynlinker).

Demo: Static vs Dynamic Linking

Check: week10_slink/*

Things to check:

1. mains, maind
2. How to create static/shared libs
3. Relative sizes of mains, maind
4. readelf/objdump --reloc -d of mains/maind

Object File (Contents)

(We only consider x86 ELF format)

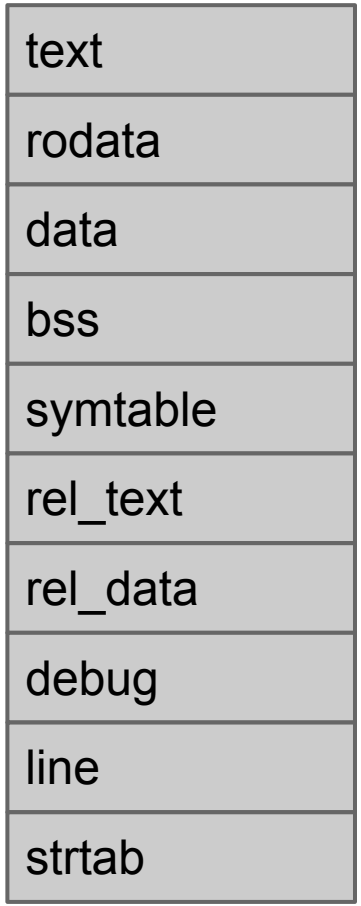
Contains several sections in binary format.

```
$ gcc -c -o myobjfile.o main.c
```

(relocatable) object file! Not an executable yet!

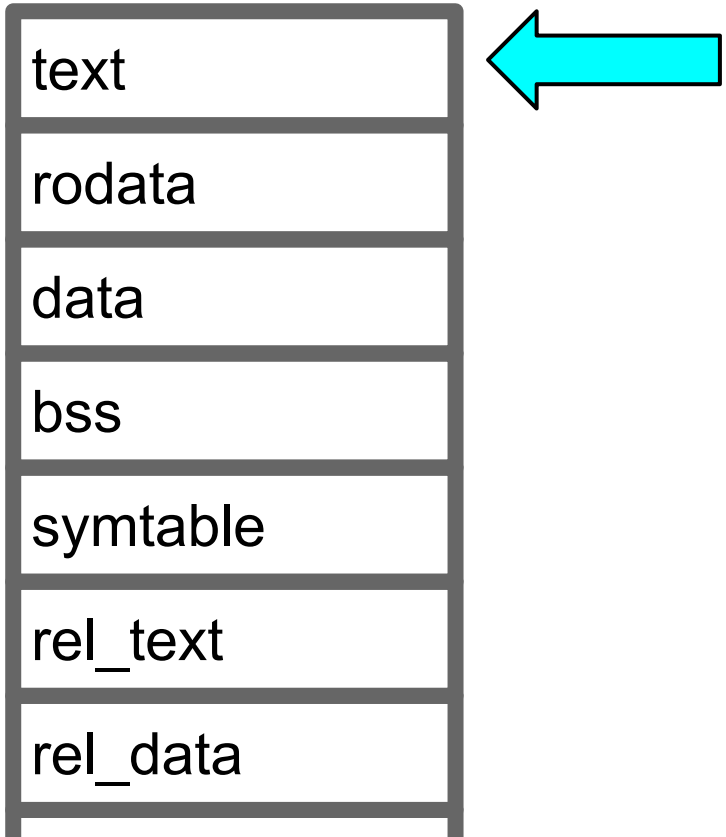
Object File (Contents)

Is actually empty!



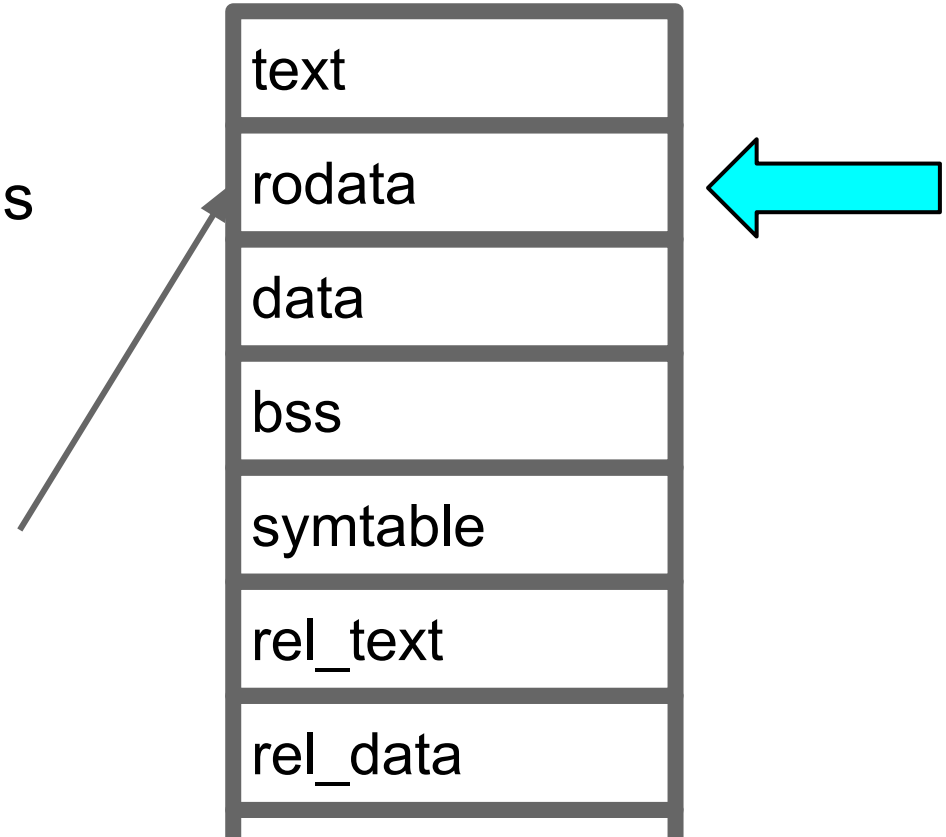
(Not shown: Header and Footer)

text: instructions for *this* module, ie any functions defined in this file.



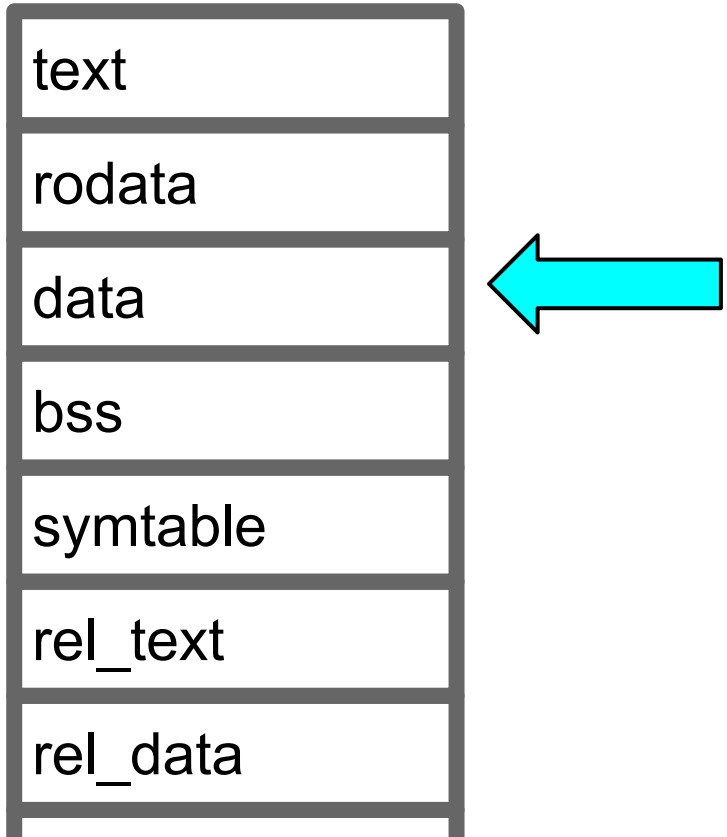
rodata: Read-only data. Example: constant strings declared in your programs.

```
int main() {
    char* msg = "Hello!\n";
    printf(msg);
    return 0;
}
```



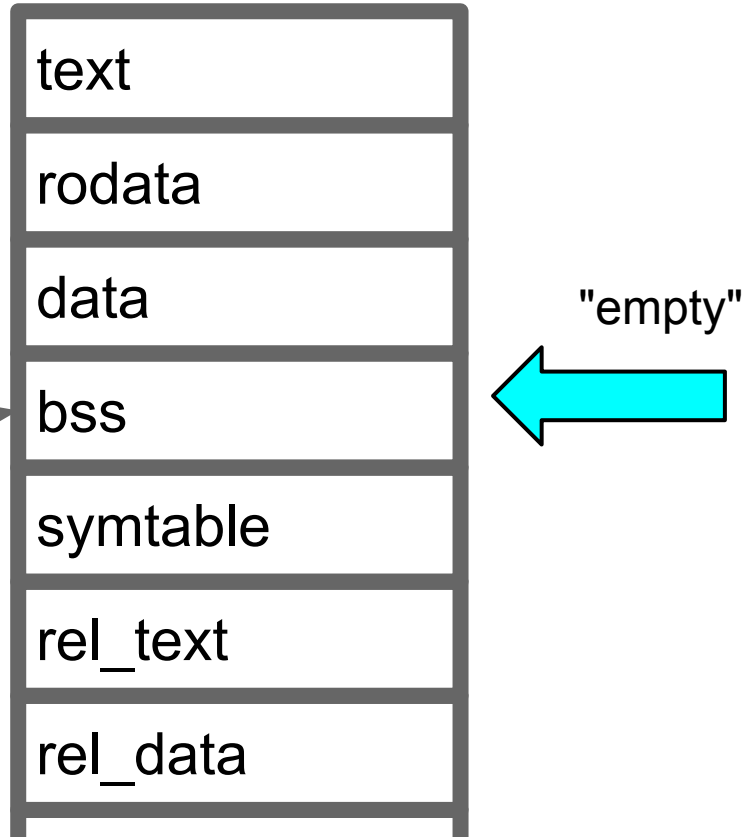
data: Read/write data that is **initialized** by user. Example: static initialized variables (ie file-level).

```
// code.c
int PI_APPROX = 3.1415
int answer;
...
int main() {
    …
}
```



bss: Read/write data that is **uninitialized** by user. Example: static uninitialized variables.

```
// code.c
int PI_APPROX = 3.1415
int answer;
...
int main() {
    …
}
```



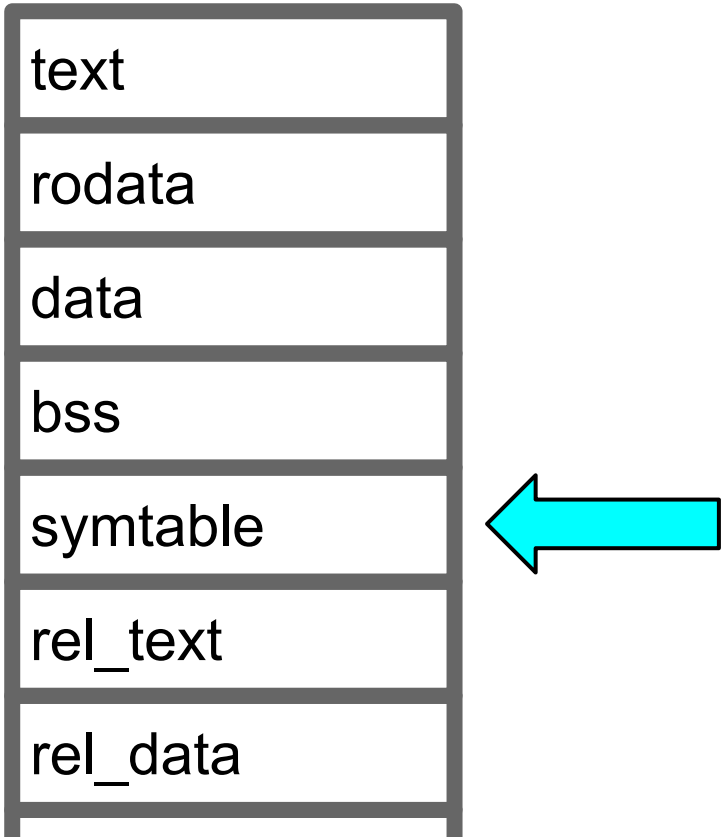
bss section: "Better Save Space"

bss section only contains length, ie # bytes of bss data there is.

Thus, object file does not actually set aside space for these uninitialized file-scope variables.

However, when the program is executed, the .bss section is allocated on the stack **zero'd out**.

symtable: Information about symbols:
(1) What symbols I define
(2) What symbols are undefined



symtable

```
// file1.c
#include <stdio.h>
extern float PI;

void main() {
    printf("pi=%f\n",PI);
}
```

```
// file2.c
float PI = 3.1415;
```

Demo: Show symtables for file1.o, file2.o

Question: What do the symtables look like for both file1.o and file2.o?

Answer: file1.o declares main(), but claims PI, printf are undefined.
file2.o declares PI.

symtable

Ndx: The section index. # can have diff meaning for different objfiles! 1: text

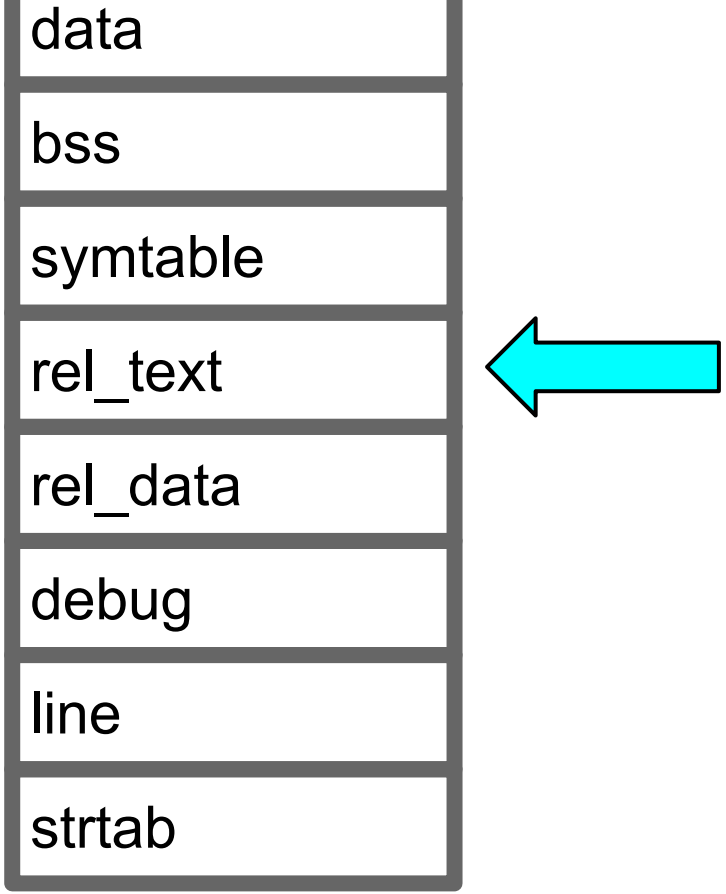
```
$ readelf -s file1.o
Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS file1.c
     ...
     9: 0000000000000000    38 FUNC    GLOBAL DEFAULT    1 main
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND PI
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

symtable

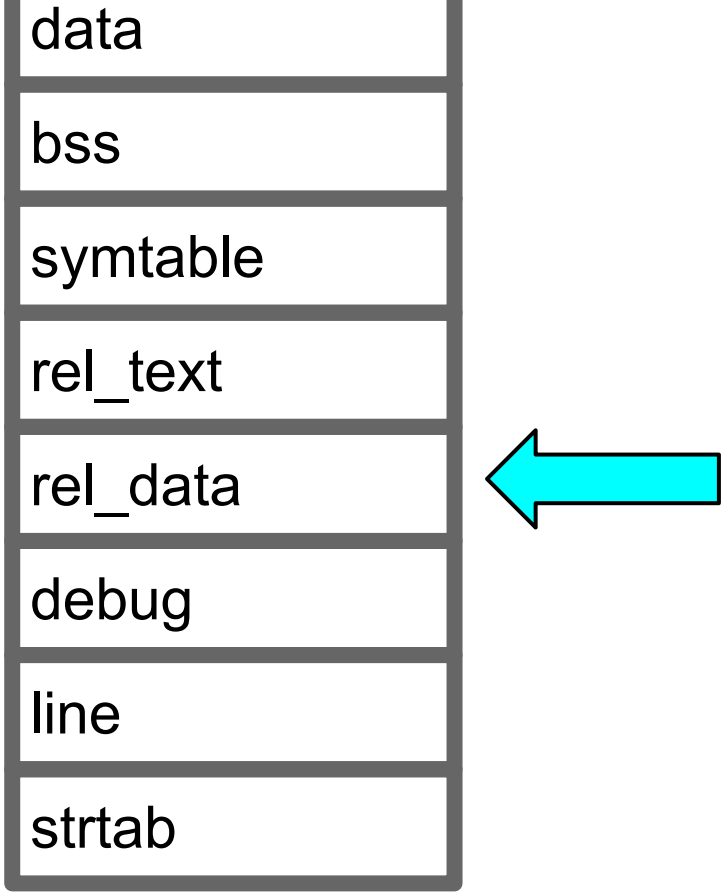
Ndx: The section index.
1: text 2: data 3: bss

```
$ readelf -s file2.o
Symbol table '.symtab' contains 8 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS file2.c
     ...
     7: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 PI
```

rel_text: Locations in .text that must be relocated.
Ex: Any inst that calls an externally-defined function.



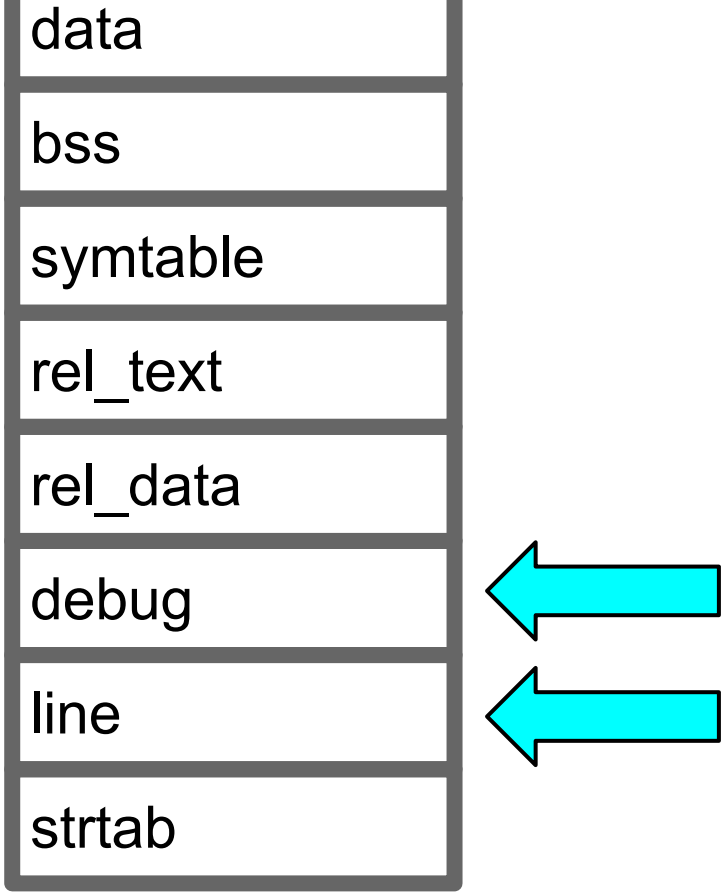
rel_data: Locations in .data that must be relocated.
Ex: Any initialized global variable whose initial value is the addr of a global variable or externally-defined function.



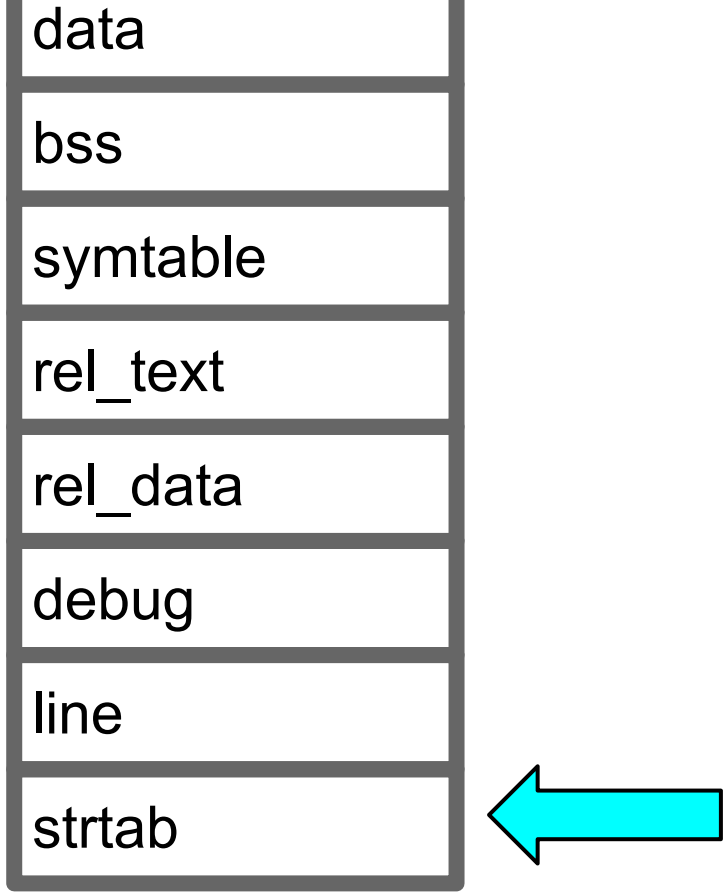
debug: Contains information for debugger.

line: Maps btwn machine code insts and C source code lines.

For both debug+line, must use -g compiler option.



strtab: String table. Contains the human-readable string for each symbol in symtable and debug.



Symbol Resolution, Relocation

During compilation, need to know

(1) What symbols are defined (and where)

(2) What run-time addresses to assign to each symbol

Key: Need to handle multiple files, each referring to each other.

Symbol Resolution

```
$ gcc -c code.s
```

Idea: At **assemble** time, want to know:

(1) Which symbols do I define?

(2) Which symbols are defined in other files?

Basically: Creates the .**symtable** section of the object file.

Quick Example

```
// file1.c
#include <stdio.h>
extern int counter, var;
float PI = 3.1415;
int fn(x){ return x + var; }
int main() {
  printf("%f\n", fn(counter)+PI);
  return 0;
}
```

Question: Which symbols are completely resolved during symbol resolution ($ gcc -c file1.c)? Which symbols are declared as UNDEF?

Answer:
Resolved: PI, fn, main.
Undefined: counter, var

Strong vs Weak Symbols

Mechanism to handle multiple definitions of the same global symbol.

Basic: "strong" symbols trump "weak" symbols.

Can't have >1 "strong" symbols for same symbol (error).

For details, read Textbook (Ch. 7.6.1, pg 664).

Relocation

Occurs after Symbol Resolution.

At this point, linker knows the exact sizes of code/data sections.

Next: Assign **run-time addresses** to each symbol!

objdump

Can use objdump to view *unrelocated* symbols.

```
$ objdump -d --reloc file1.o
Disassembly of section .text:
00000000 <main>:
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    83 e4 f0                and    $0xfffffff0,%esp
   6:    83 ec 10                sub    $0x10,%esp
   9:    d9 05 00 00 00 00       flds   0x0
             b: R_386_32   PI
   f:    b8 00 00 00 00          mov    $0x0,%eax
             10: R_386_32  .rodata
  14:    dd 5c 24 04             fstpl  0x4(%esp)
  18:    89 04 24                mov    %eax,(%esp)
  1b:    e8 fc ff ff ff          call   1c <main+0x1c>
             1c: R_386_PC32printf
  20:    c9                      leave
  21:    c3                      ret
```

objdump

Contrast with fully-linked (relocated) executable:

```
$ objdump -d --reloc file1
Disassembly of section .text:
080483e4 <main>:
 80483e4: 55                      push   %ebp
 80483e5: 89 e5                   mov    %esp,%ebp
 80483e7: 83 e4 f0                and    $0xfffffff0,%esp
 80483ea: 83 ec 10                sub    $0x10,%esp
 80483ed: d9 05 14 a0 04 08       flds   0x804a014
 80483f3: b8 e0 84 04 08          mov    $0x80484e0,%eax
 80483f8: dd 5c 24 04             fstpl  0x4(%esp)
 80483fc: 89 04 24                mov    %eax,(%esp)
 80483ff: e8 fc fe ff ff          call   8048300 <printf@plt>
 8048404: c9                      leave
 8048405: c3                      ret
```

Relocation Algorithm

Relocation effectively performs the previous transformation, via a **fixed-point** algorithm.

For details on relocation algorithm itself, read Ch. 7.7 (pgs 672-677). Algorithm itself is on pg. 674 (Fig. 7.9).

Questions you should consider:

(1) Convince yourself that the algorithm always terminates.

(2) On lines 8,13, why do I have to add *refptr when computing *refptr?

For Fun: Vision Demos

(1) Hand Localization

(2) Handwriting Detection