# CSC 374/407: Computer Systems II

Lecture 2

Joseph Phillips
De Paul University

2014 January 3

# Reading

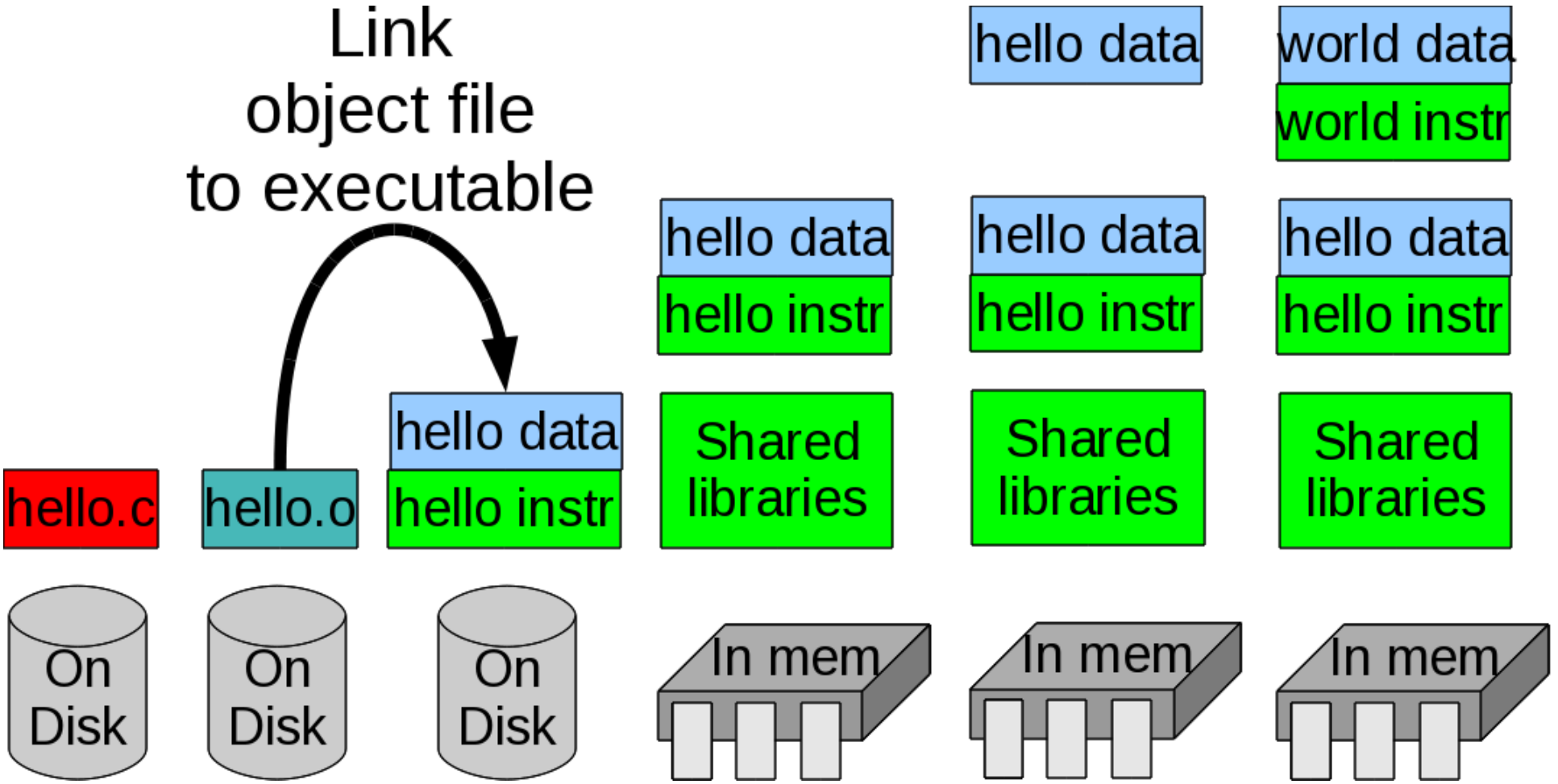
- Bryant & O'Hallaron "*Computer Systems, 2nd Ed.*"
  - Chapter 7: Linking
- Hoover "*System Programming*"
  - Program Management 6
  - Libraries 8.1-8.3, 8.6, 8.7

# Topics

- Linkers
- ELF
- Dynamic Linking

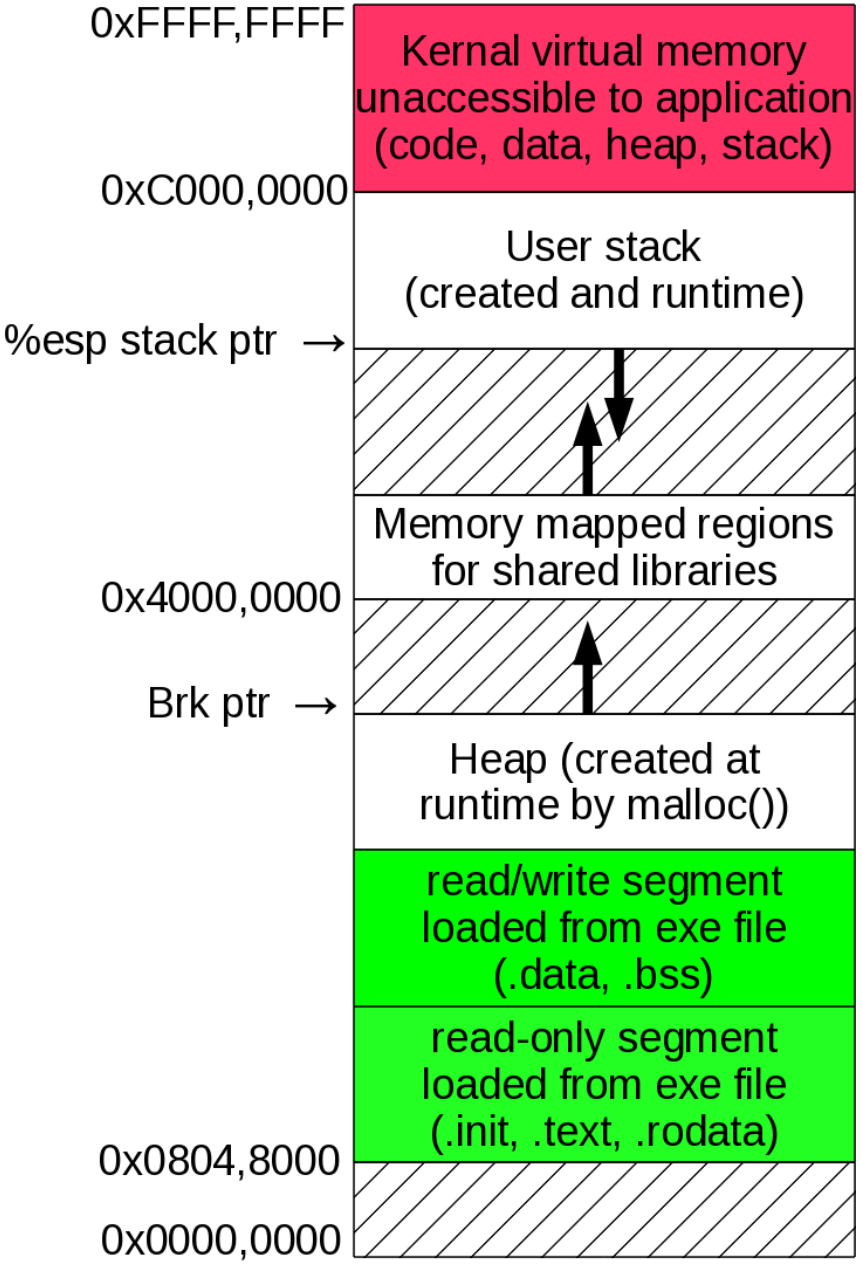
# Today's topic (in time)

## Linking object files into executable files



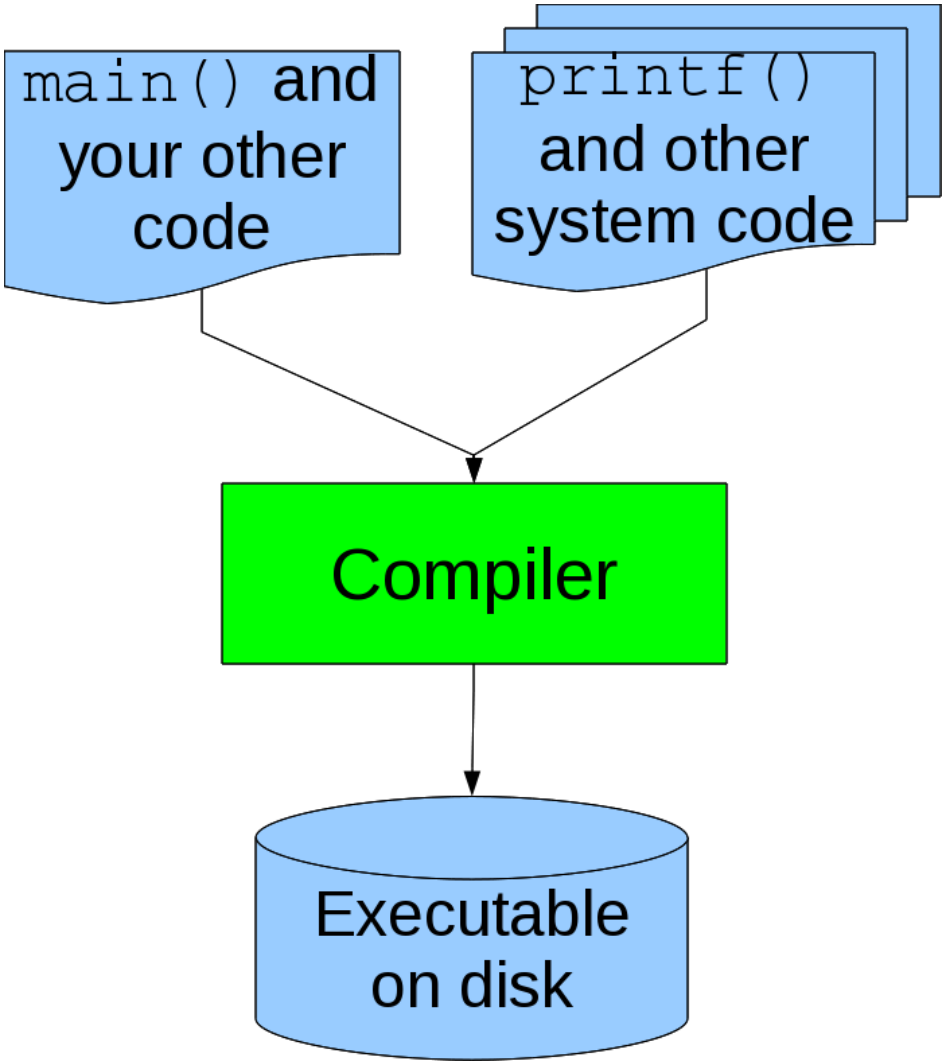
# Today's topic (in space)

- Parts of the program the compiler actually has to create



# How would *you* write a compiler?

- The direct approach:
- One program to compile ***everything***:
  - ***Everything*** includes your code (naturally!)
  - ***Everything*** includes standard library code too (***hmm?***)

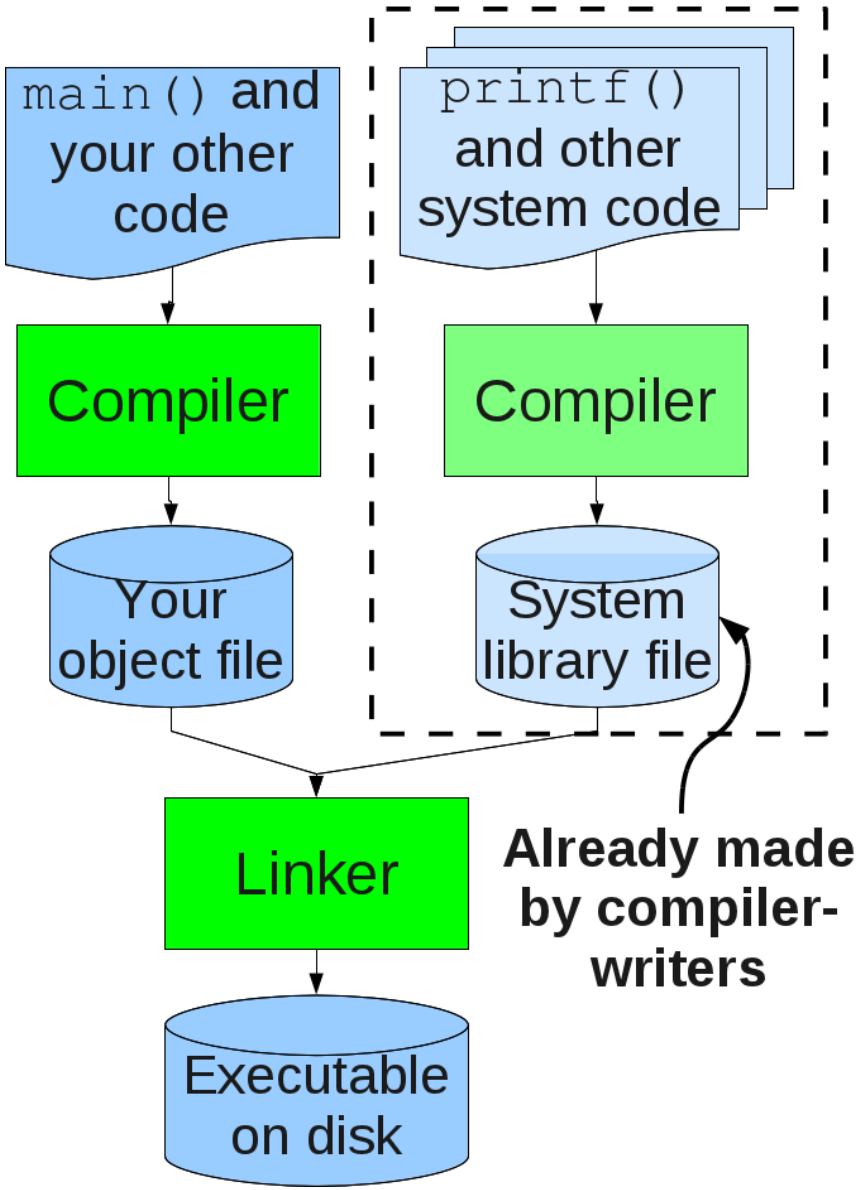


# *What?!? Re-compiling library code again and again?!?*

- Sounds like a bad idea because
  1. ***It's a waste of time***: mostly it doesn't change.
  2. ***So much for hiding implementation***: When given easy-access to the source code some folks just can't stop themselves from trying to “optimize” their own code after seeing how the standard library code actually works. ***Even worse*** they can start to muck with it . . .
- So, let's give them the library code in a ***post-compiled*** but ***pre-integrated*** into a program . . .

# Enter the linker!

- Compile standard code into libraries.
- Distribute libraries to user-programmers
- Programmers compile their own code to object files
- ***Linkers combine object files + libraries into executable files***

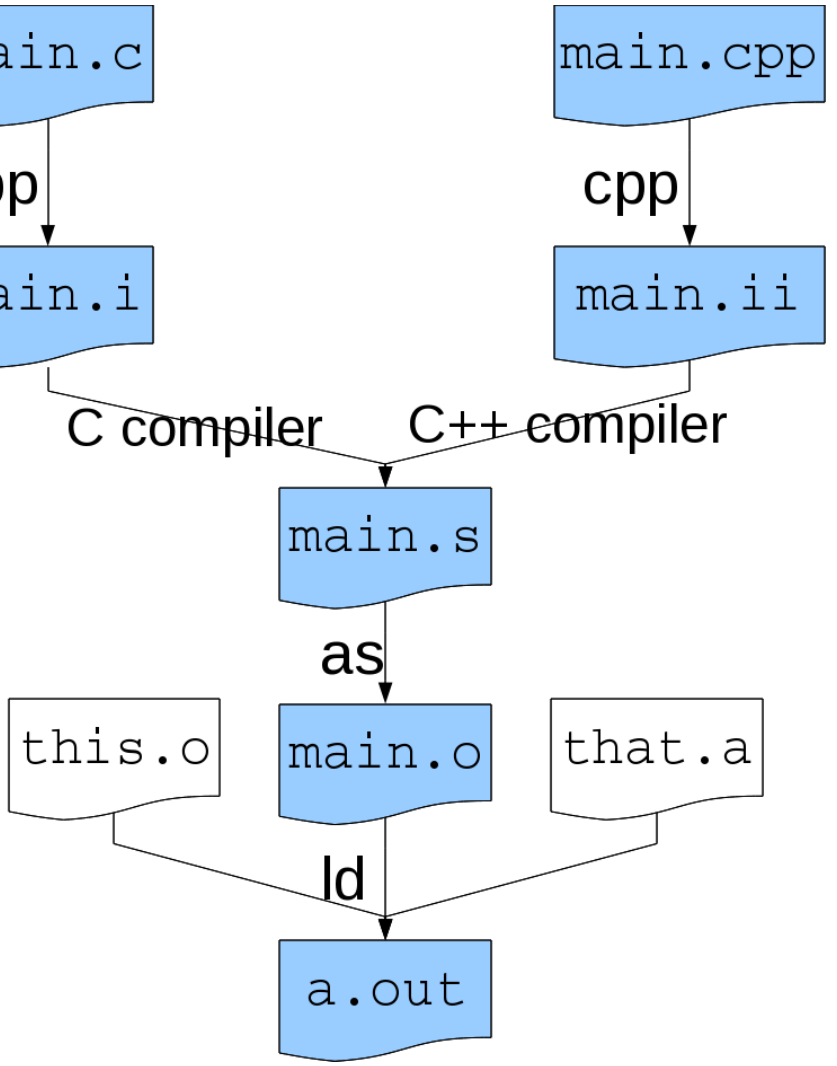


# Linkers give you . . .

- **Modularity**
  - You can write your program as small files can compile them individually
  - You can use the libraries of others or build your own
- **Efficiency**
  - ***Time***: No need to re-compile everything if change one small file
  - ***Space***: Can incorporate from libraries only code that's actually used

# There's a linker under the hood?

- **You** "*But all I have to say is* `gcc main.c`"
- **Prof Joe** "*Yes, 'cuz it does a number of things behind your back*:"
  - `cpp`: the C preprocessor
  - The C/C++ compiler proper
  - `as`: the assembler
  - `ld`: the linker



# Let's try!

- You can tell `gcc` and g++ to stop along the way:
  - `gcc -E test.c # stop after preprocessing`
    - Output is post-processed C code to `stdout`
  - `gcc -S test.c # stop after compiling`
    - Output is assembly code in `test.s`
  - `gcc -c test.c # stop after assembling`
    - Output is object file `test.o`
  - `gcc test.c  # Go ***all the way*** baby!`
    - Output is executable file `a.out`

# How do Linkers Spend Their Time?

To merge object files they must:

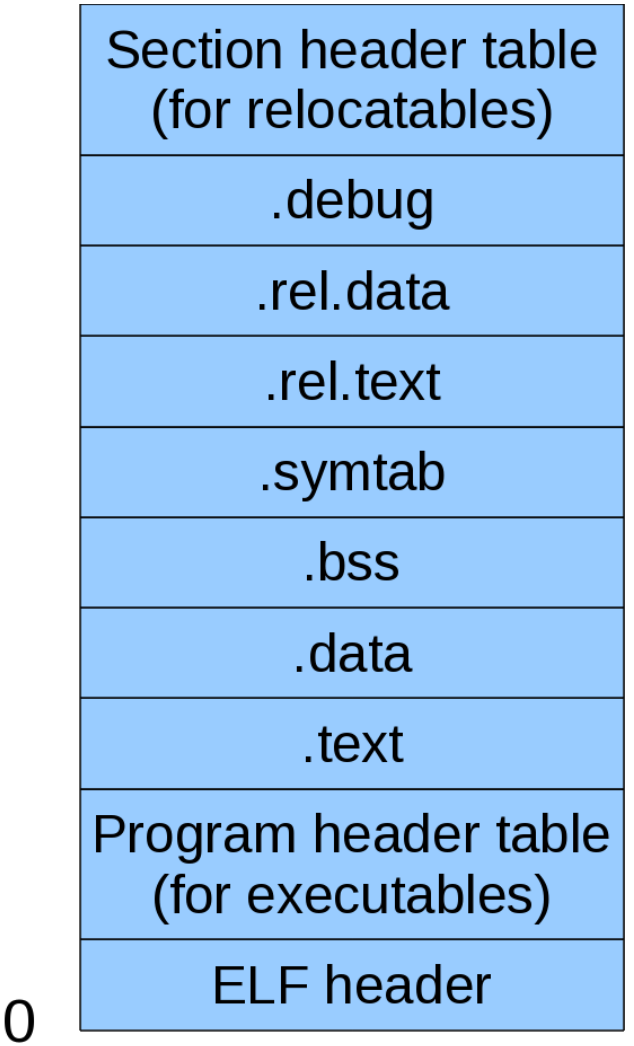
- Resolve “external references:”

```
extern int externallyDefinedVar;
extern int externallyDefinedFunc(int);
int        locallyDefinedGlobalVar;
int main ()
{
   . . .
  externallyDefinedVar=externallyDefinedFnc(5);
}
```

- Move all symbols from several object files to distinct places in executable
- Update symbol locations
- “Symbol” means either named code or data

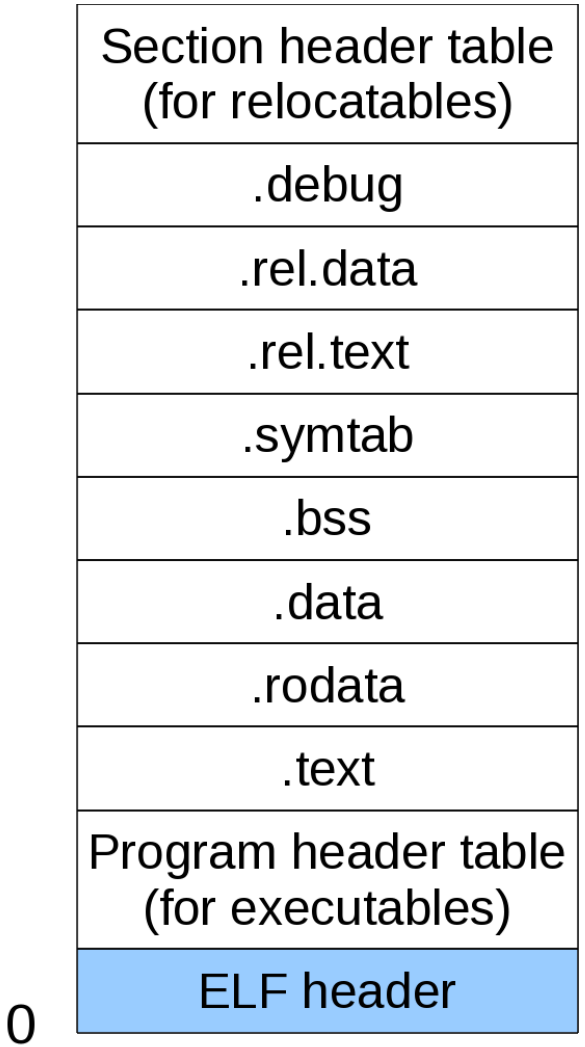
# Executable and Link Format (ELF)

- Assembler, Linker and Program Loader all need one format to describe object files and executables
- ELF derived from AT&T Unix System V and BSD
- Unifies description of:
  - Relocatable object files (.o)
  - Executables
  - Shared object files (.so)



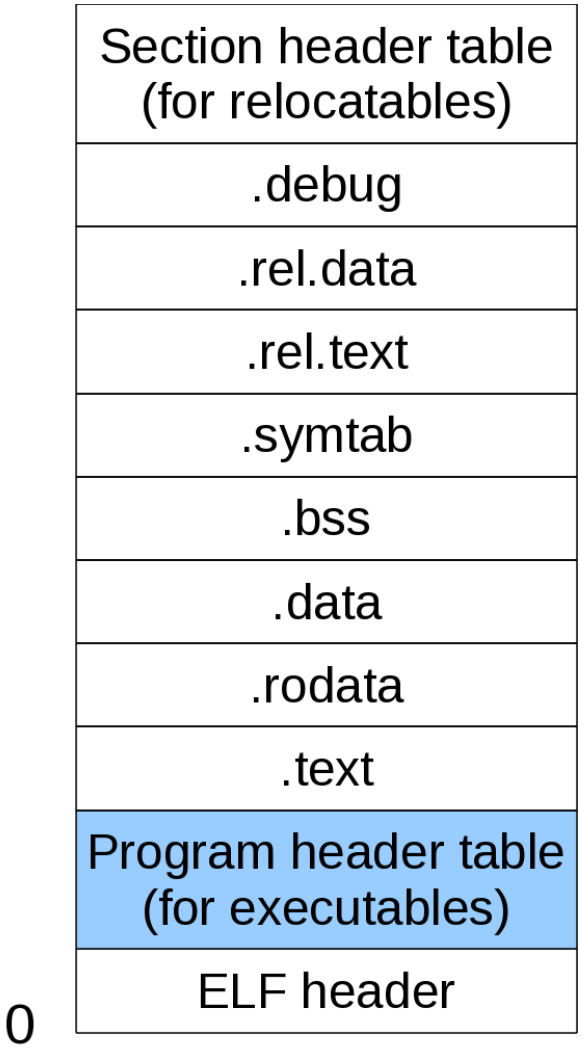
# What's in an ELF?

- ELF header:
  - “Magic number” (Am I for Intel 386? Sparc Solaris? Something else?)
    - Important for shared disk systems
  - Byte ordering
    - Big or little endian?
  - Type
    - .o?  Executable?  .so?



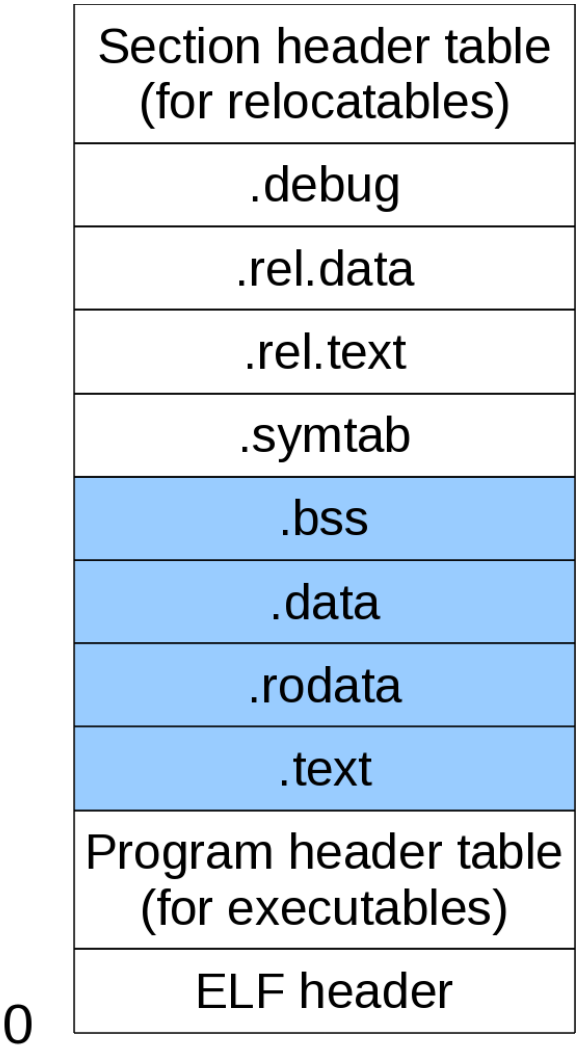
# What's in an ELF?

- Program header:
  - OS-specific info like
    - Page size
    - Virtual address mem segments
    - Segment sizes



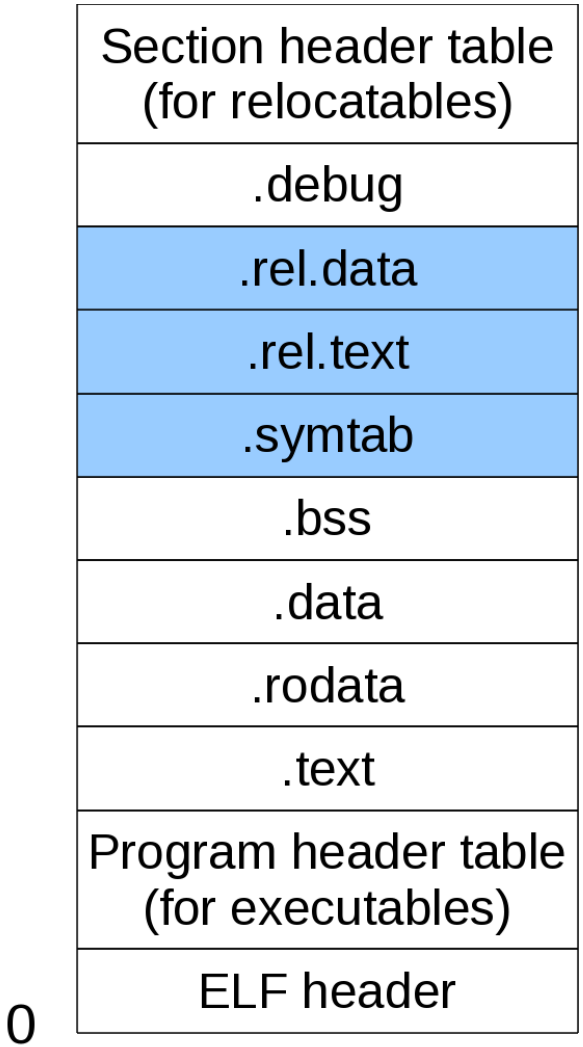
# What's in an ELF?

- `.text`
  - Code (functions)
- `.rodata`
  - ***R***ead-***O***nly consts (*e.g.* strings)
- `.data`
  - Global and static vars initialized to other than 0
- `.bss`
  - Global and static vars that start out 0
  - (Why not in `.data`?)



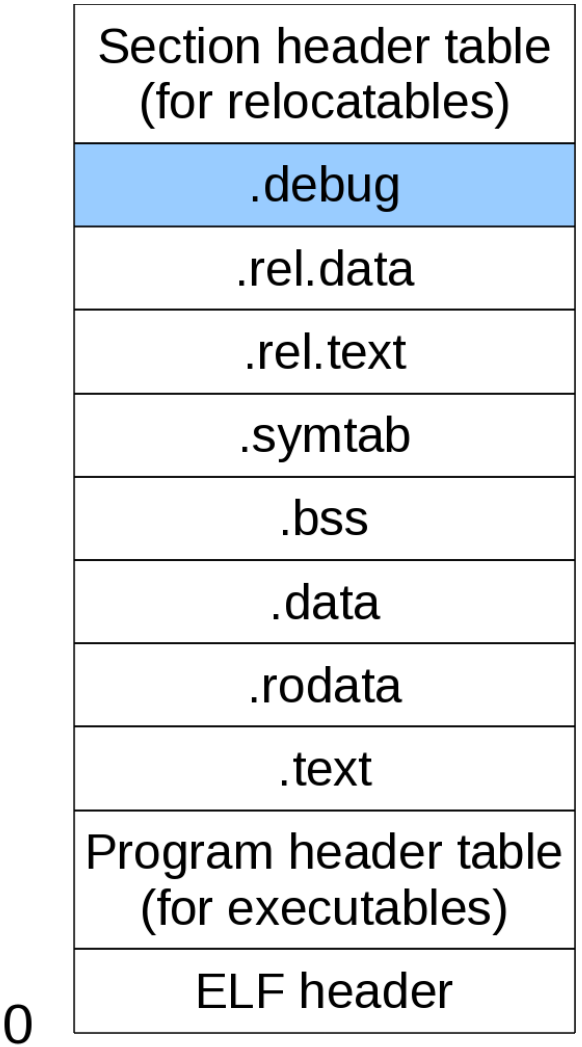
# What's in an ELF?

- `.symtab`
  - Symbol table holds names and locations of functions, variables and sections
- `.rel.text`
  - Relocatable info for .text (which `.text` addresses need to be changed and how)
- `.rel.data`
  - Ditto for `.data`



# What's in an ELF?

- .debug
  - Information for gdb debugger
  - Generated when compile with -g option



# Don't be shy, let's look in an ELF

```
/* Sample program hello.c */
#include <stdlib.h>
#include <stdio.h>

int     OS_EVERYTHING_OKAY = 0;

void    helloWorld     ()
{
  puts("Hello world!");
}

int     main ()
{
  helloWorld();
  return(OS_EVERYTHING_OKAY);
}
```

```
[jphillips@localhost lecture2]$ gcc -c hello.c # makes hello.o
[jphillips@localhost lecture2]$ gcc hello.c -o hello
```

# Peering inside an ELF

- `readelf -h <file>`
  - Prints header info of `.o` `.so` and execs

```
[jphillips@localhost lecture2]$ readelf -h hello
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:               0x80483b0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          2480 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

# Peering inside an ELF

- `readelf -S <file>`
  - Prints info on sections

```
[jphillips@localhost lecture2]$ readelf -S hello
There are 30 section headers, starting at offset 0x9b0:

Section Headers:
  [Nr] Name              Type     Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL     00000000 000000 000000 00      0   0  0
  . . . .
  [20] .jcr              PROGBITS 08049694 000694 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC  08049698 000698 0000e0 08  WA  6   0  4
  [22] .got              PROGBITS 08049778 000778 000004 04  WA  0   0  4
  [23] .got.plt          PROGBITS 0804977c 00077c 00001c 04  WA  0   0  4
  [24] .data             PROGBITS 08049798 000798 000008 00  WA  0   0  4
  [25] .bss              NOBITS   080497a0 0007a0 000008 00  WA  0   0  4
  [26] .comment          PROGBITS 00000000 0007a0 000114 00      0   0  1
  [27] .shstrtab         STRTAB   00000000 0008b4 0000fc 00      0   0  1
  [28] .symtab           SYMTAB   00000000 000e60 000450 10     29  46  4
  [29] .strtab           STRTAB   00000000 0012b0 000250 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

# Peering inside an ELF

- `readelf -s <file>`
  - Prints symbol table

```
[jphillips@localhost lecture2]$ readelf -s hello
Symbol table '.dynsym' contains 7 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     2: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
     3: 00000000   438 FUNC    GLOBAL DEFAULT  UND
__libc_start_main@GLIBC_2.0 (2)
     4: 00000000   399 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.0 (2)
     5: 08048394  1020 FUNC    GLOBAL DEFAULT  UND
__gxx_personality_v0@CXXABI_1.3 (3)
     6: 08048588     4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used

Symbol table '.symtab' contains 69 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08048134     0 SECTION LOCAL  DEFAULT    1
     2: 08048148     0 SECTION LOCAL  DEFAULT    2
     . . .
```

# Peering inside an ELF

- objdump -d -j <section> <file>
  - Disassembles (“-d”) <section> (“-j”) of <file>

```
[jphillips@localhost lecture2]$ objdump -d -j .text hello.o
hello.o:     file format elf32-i386

Disassembly of section .text:

00000000 <helloWorld>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
   d:   e8 fc ff ff ff          call   e <helloWorld+0xe>
  12:   c9                      leave
  13:   c3                      ret

00000014 <main>:
  14:   8d 4c 24 04             lea    0x4(%esp),%ecx
  18:   83 e4 f0                and    $0xfffffff0,%esp
  1b:   ff 71 fc                pushl  -0x4(%ecx)
  1e:   55                      push   %ebp
  1f:   89 e5                   mov    %esp,%ebp
  21:   51                      push   %ecx
  22:   83 ec 04                sub    $0x4,%esp
     . . .
```

# Now we know all about ELFs, *Put the linker to work!*

```
/* one.c */

#include <stdlib.h>
#include <stdio.h>

int     start   = 0;
int     stop    = 100;

extern  int sum ();

int     main    ()
{
  printf("Sum from %d to %d is %d\n",
          start,stop,sum()
         );
  return(EXIT_SUCCESS);
}
```

- **Which `one.o` section has:**
  - **`main()`?**
  - **`stop`?**
  - **`start`?**
  - **The string `"Sum from %d to %d is %d\n"`?**
  - ***How can we show this?***

# Now we know all about ELFs, *Put the linker to work!*

```
/* two.c */

extern  int     start;
extern  int     stop;

int *addrOfStop = &stop;

int     sum     ()
{
  int   i;
  int   total   = 0;

  for  (i = start;  i <= *addrOfStop;  i++)
    total += i;

  return(total);
}
```

- **Which `two.o` section has:**
  - **`sum()`?**
  - **`addrOfStop`?**
  - **`i` and `total`?**
    - ***Think carefully!***
  - ***How can we show this?***

# Linking to make an executable

```
[jphillips@localhost]$ g++ -c one.c
[jphillips@localhost]$ g++ -c two.c
[jphillips@localhost]$ g++ -o oneTwo one.o two.o
```

- Where in oneTwo are:
  - `start` and `stop`? What are their values?
  - `addrOfStop`? What is it's value?
  - `main()` and `sum()`?

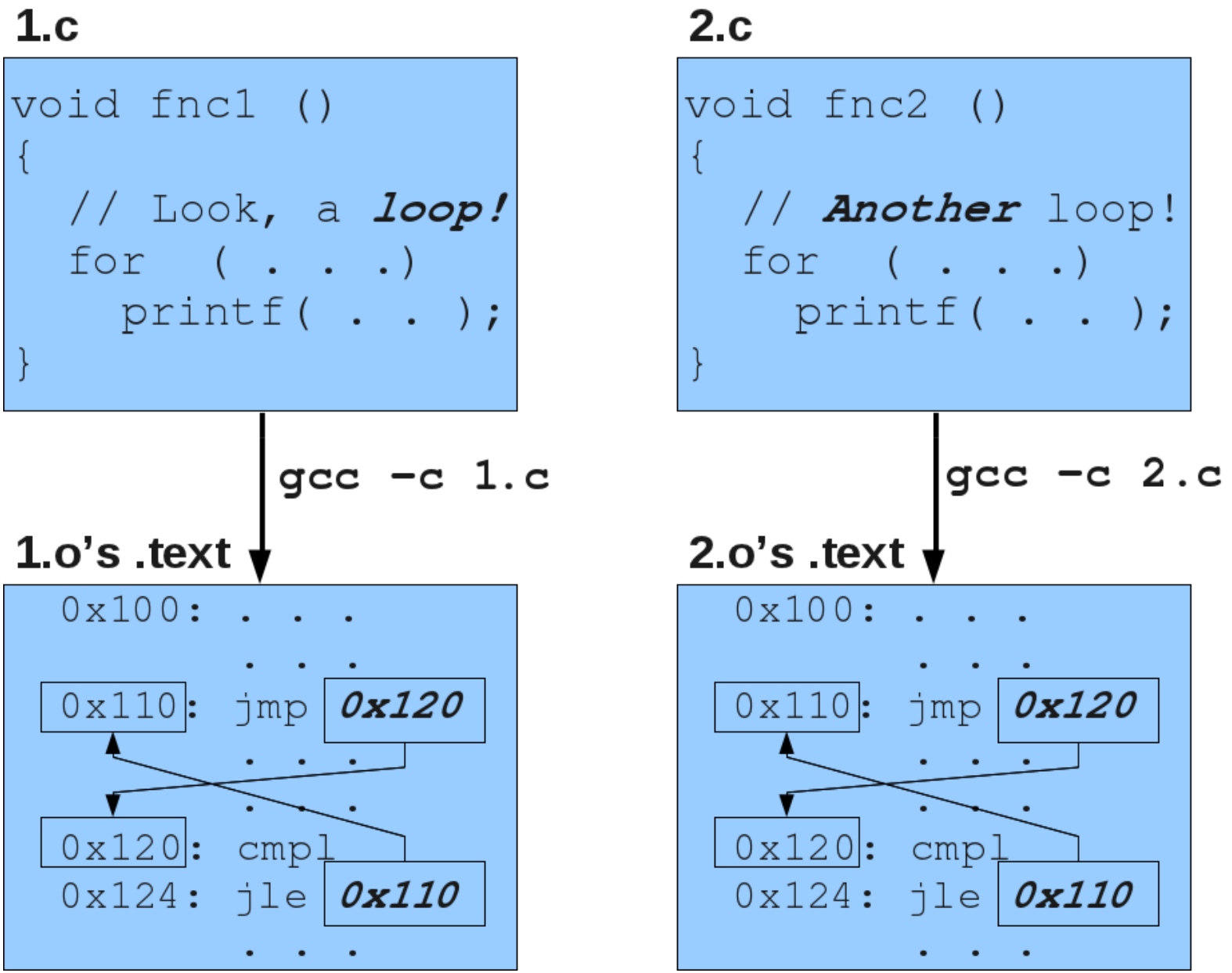
# Your turn!

- Write a program separated into three files:
  - `first.c`: Has two characters (`begin` and `end`) and `main()` which calls `enterBeginEnd()` and `printFromBeginToEnd()`.
  - `second.c`: Has `enterBeginEnd()`, which asks for a first character (`begin`), and then asks for a second character (`end`). It must ensure `end` has greater or equal ASCII value.
    - HINT: `char array[SIZE]; printf("Enter..."); fgets(array,SIZE,stdin); begin = array[0];`
  - `third.c`: Has `printFromBeginToEnd()` which prints out the characters from `begin` to `end` and their ASCII values in decimal. (Use a simple `for`-loop)

# Resolving references

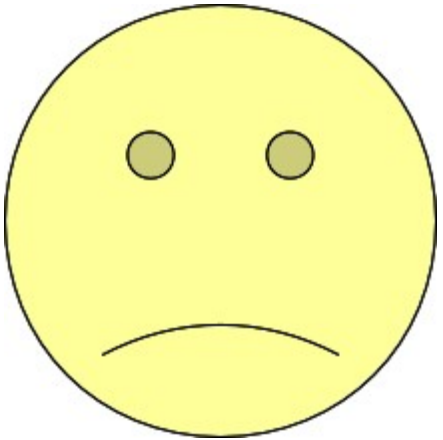
- The linker ***works*** for its money
  - It puts the all the `.text`'s together, all the `.data`'s together, *etc*.
  - It fills in the addresses of pointers, function calls, *etc*.
- I wonder how we should refer to jmp and call addresses?
  - ***Absolute addresses***?
  - ***Relative addresses?***

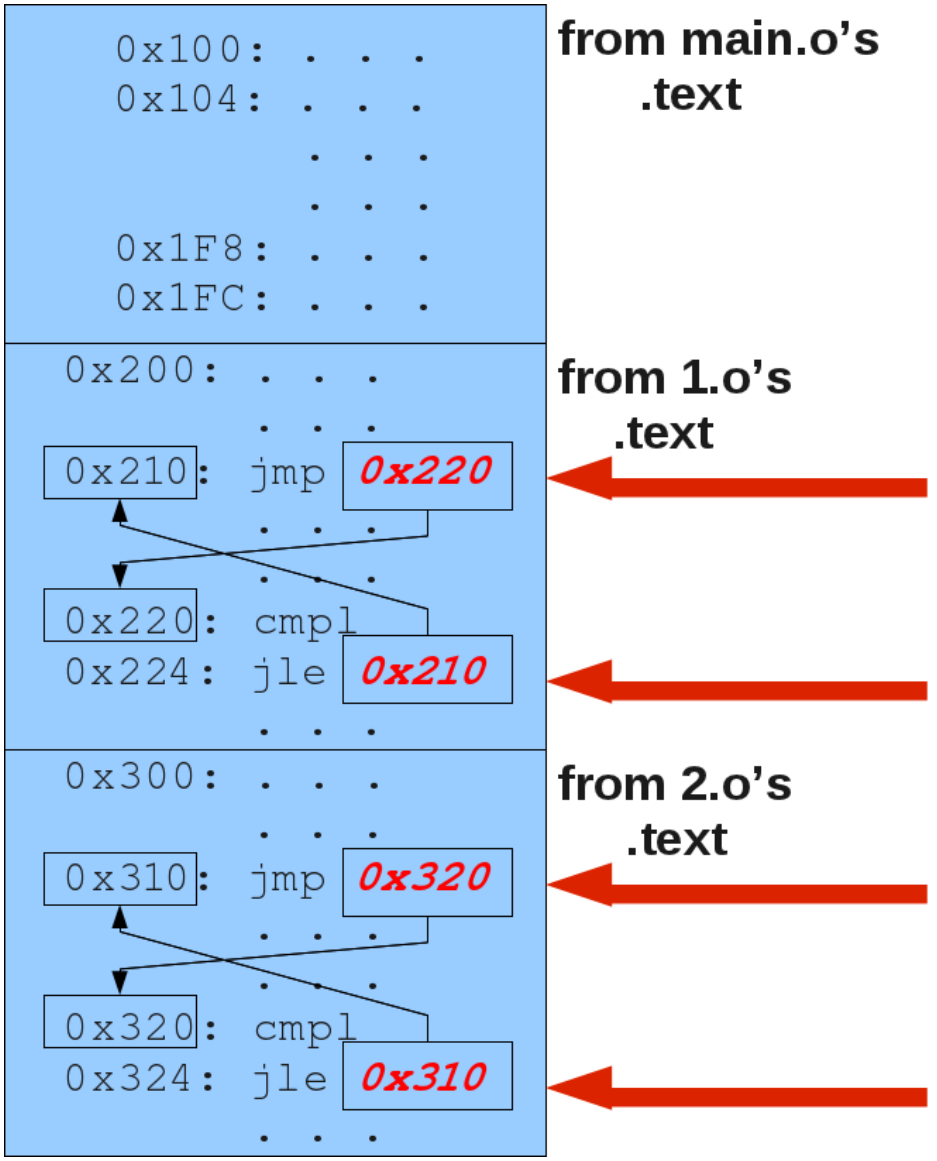
# Absolute addressing: compiling Hardcode addresses in `jmp` & `call`



# Absolute addressing: linking

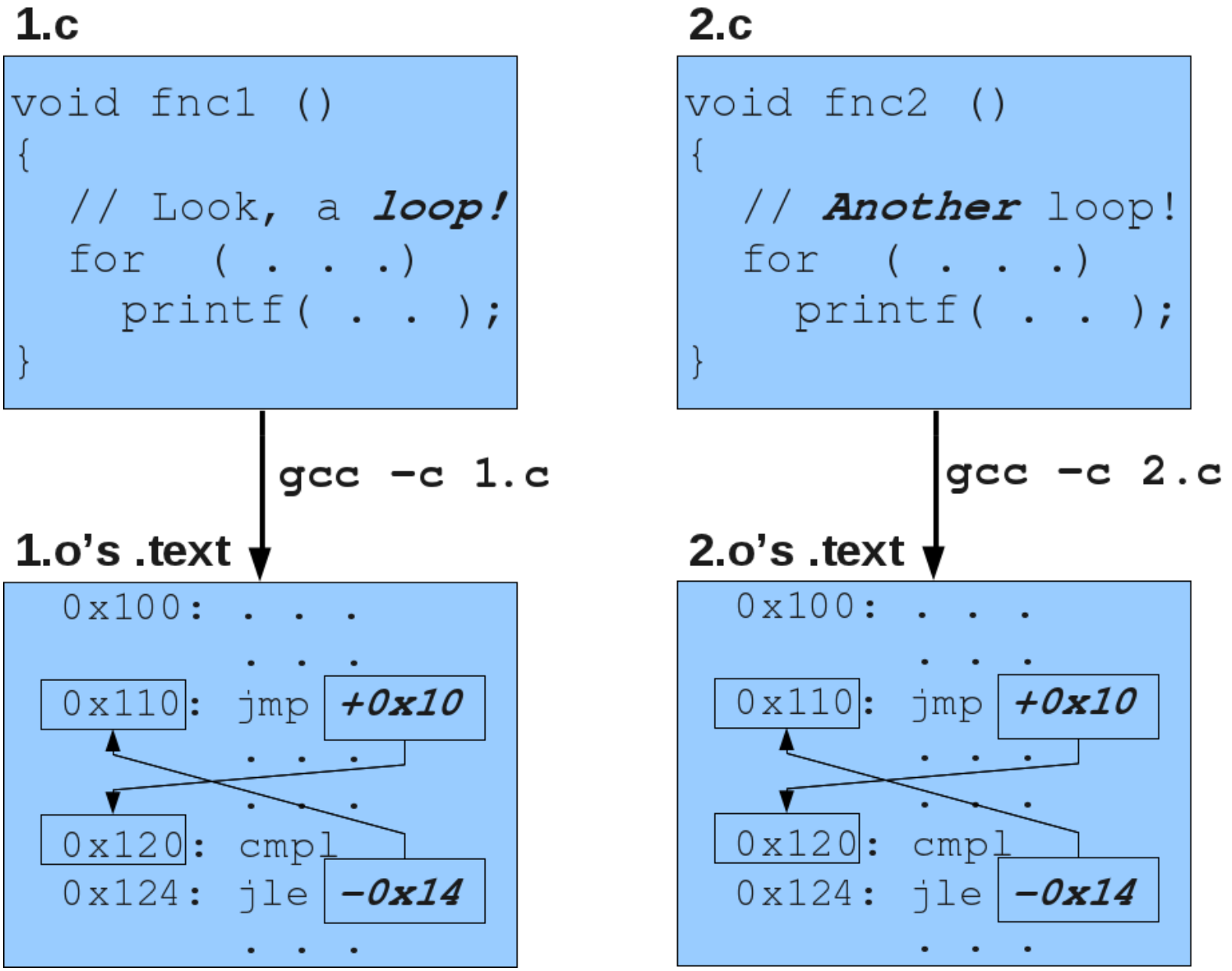
- Uh oh!
  - The linker now has **lots of work** to clean change the addresses of all jmps and calls, ***even within same source .texts!***
- Sad linker!





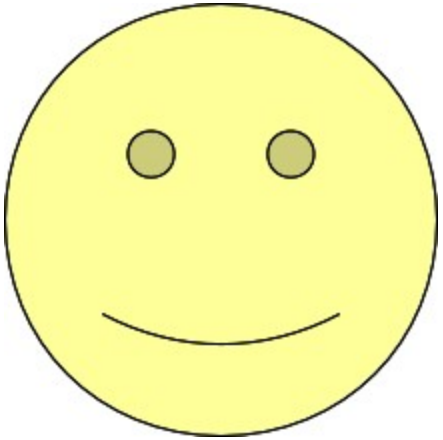
# Relative addressing: compiling

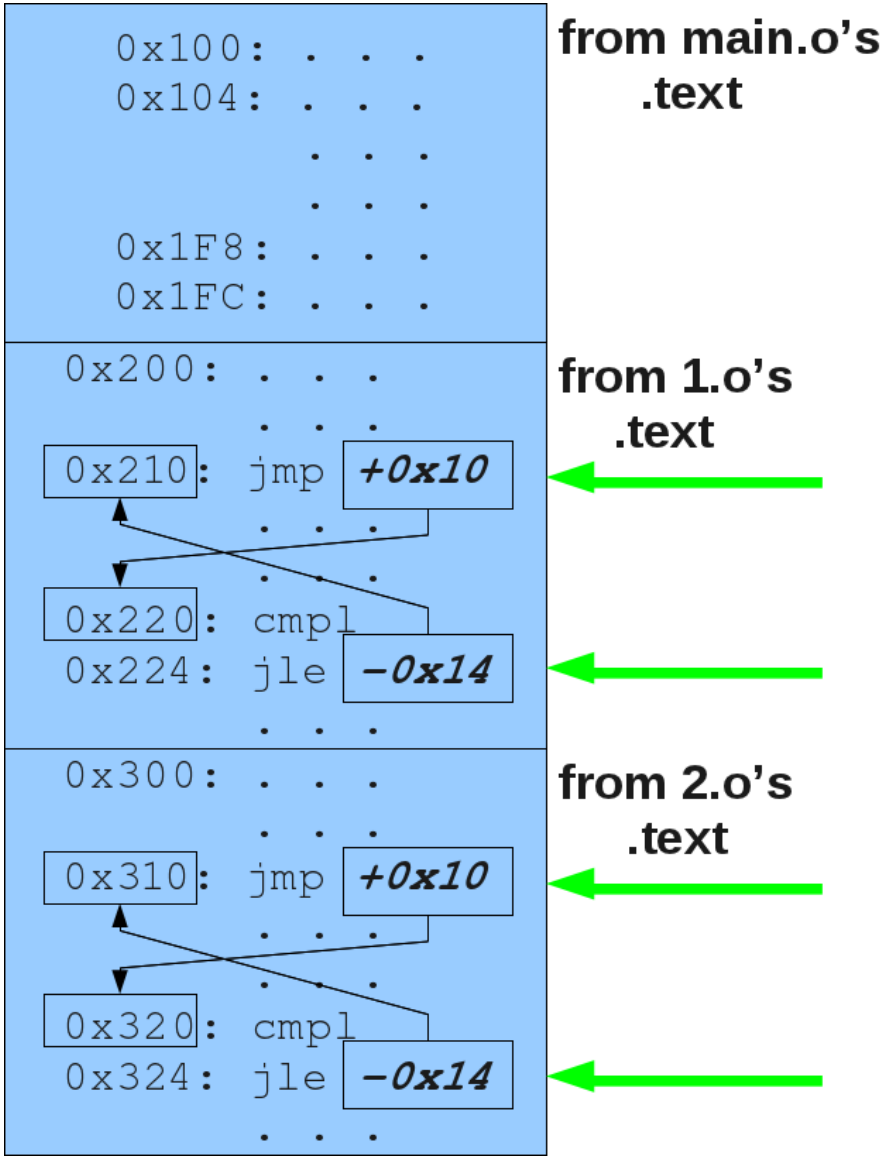
## Tell offset (what to add to `eip`) in `jmp` & `call`



# Relative addressing: linking

- Easy street!
  - The linker has ***nothing to do*** within same source .texts!
- Happy linker!





# Consider jmps in this loop:

```
#include <stdlib.h>
#include <stdio.h>

int main  ()
{
  int i;

  for  (i = 0;  i < 10;  i++)
    printf("%d ",i);

  printf("\n");
  return(EXIT_SUCCESS);
}
```

# Relative jumps, example 1:

```
080483f4 <main>:
 80483f4: 55                            push   %ebp
 80483f5: 89 e5                         mov    %esp,%ebp
 80483f7: 83 e4 f0                      and    $0xfffffff0,%esp
 80483fa: 83 ec 20                      sub    $0x20,%esp
 80483fd: c7 44 24 1c 00 00 00          movl   $0x0,0x1c(%esp)
 8048404: 00
 8048405: eb 1a                         jmp    8048421 <main+0x2d>
 8048407: b8 14 85 04 08                mov    $0x8048514,%eax
 804840c: 8b 54 24 1c                   mov    0x1c(%esp),%edx
 8048410: 89 54 24 04                   mov    %edx,0x4(%esp)
 8048414: 89 04 24                      mov    %eax,(%esp)
 8048417: e8 08 ff ff ff                call   8048324 <printf@plt>
 804841c: 83 44 24 1c 01                addl   $0x1,0x1c(%esp)
 8048421: 83 7c 24 1c 09                cmpl   $0x9,0x1c(%esp)
 8048426: 7e df                         jle    8048407 <main+0x13>
 8048428: c7 04 24 0a 00 00 00          movl   $0xa,(%esp)
 804842f: e8 d0 fe ff ff                call   8048304 <putchar@plt>
 8048434: b8 00 00 00 00                mov    $0x0,%eax
 8048439: c9                            leave
 804843a: c3                            ret
```

| | |
|---|---|
| Addr *after* jmp/call | 0x8048407 |
| + number in jmp/call | +0x000001A |
| Addr to which to go | 0x8048421 |

# You explain this one:

```
080483f4 <main>:
 80483f4:  55                          push   %ebp
 80483f5:  89 e5                       mov    %esp,%ebp
 80483f7:  83 e4 f0                    and    $0xfffffff0,%esp
 80483fa:  83 ec 20                    sub    $0x20,%esp
 80483fd:  c7 44 24 1c 00 00 00        movl   $0x0,0x1c(%esp)
 8048404:  00
 8048405:  eb 1a                       jmp    8048421 <main+0x2d>
 8048407:  b8 14 85 04 08              mov    $0x8048514,%eax
 804840c:  8b 54 24 1c                 mov    0x1c(%esp),%edx
 8048410:  89 54 24 04                 mov    %edx,0x4(%esp)
 8048414:  89 04 24                    mov    %eax,(%esp)
 8048417:  e8 08 ff ff ff              call   8048324 <printf@plt>
 804841c:  83 44 24 1c 01              addl   $0x1,0x1c(%esp)
 8048421:  83 7c 24 1c 09              cmpl   $0x9,0x1c(%esp)
 8048426:  7e df                       jle    8048407 <main+0x13>
 8048428:  c7 04 24 0a 00 00 00        movl   $0xa,(%esp)
 804842f:  e8 d0 fe ff ff              call   8048304 <putchar@plt>
 8048434:  b8 00 00 00 00              mov    $0x0,%eax
 8048439:  c9                          leave
 804843a:  c3                          ret
```

$$\begin{array}{r} 0x8048428 \\ +\quad 0xDF \\ \hline 0x8048407 \end{array}$$

*¿¿¿ How ???*

Try explain the math behind a call in program you just wrote

# Strong and weak symbols

- Strong symbols
  - Initialized global vars
  - Functions
- Weak symbols:
  - Uninitialized global vars
- Rules:
  1. Only one strong symbol
  2. Weak symbols over ridden by strong
  3. If all weak linker can choose arbitrarily

```
// Strong
int initialized = 0;

// Weak
int unInit;

// Strong
int someFnc()
{
 return(strong+weak);
}
```

# Happy Linker :) or Sad Linker :( ?

```
/* 1.c */

int var;

int someFnc()
{
  . . .
}
```

```
/* 2.c */

int someFnc()
{
  . . .
}
```

# Happy Linker :) or Sad Linker :( ?

```
/* 1.c */

int var;

int someFnc1()
{
  . . .
}
```

```
/* 2.c */

int var;

int someFnc2()
{
  . . .
}
```

# Happy Linker :) or Sad Linker :( ?

```
/* 1.c */

int var1;
int var2;

int someFnc1()
{
  . . .
}
```

```
/* 2.c */

double var1;

int someFnc2()
{
  . . .
}
```

# Happy Linker :) or Sad Linker :( ?

```
/* 1.c */

int var1 = 10;

int someFnc1()
{
  . . .
}
```

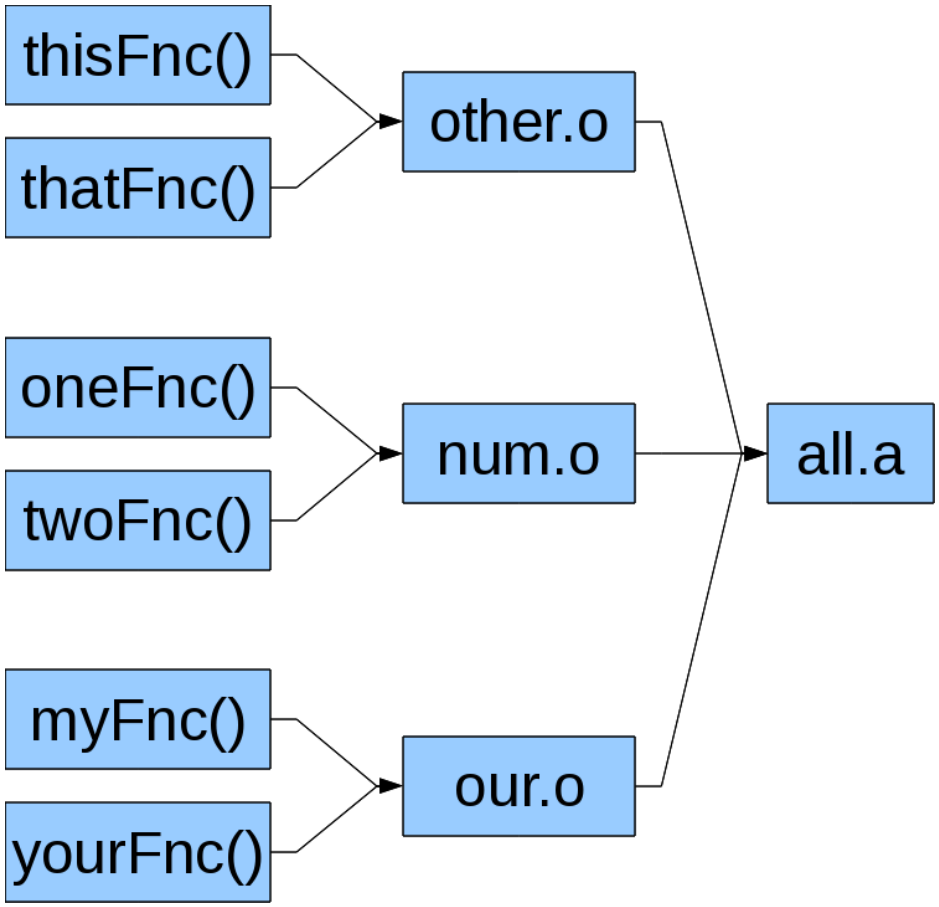
```
/* 2.c */

double var1;

int someFnc2()
{
  . . .
}
```

# Static libraries (.a for “archives”)

- Link related object files
  - General C functionality `/usr/lib/libc.a`
  - Math-related (`sin()`, `cos()`, *etc*.) `/usr/lib/libm.a`
  - Cryptography `/usr/lib/libcrypt.a`
  - Also for strings, graphics, *etc*.



# Static libraries (.a for “archives”)

- What's in them? Find out:

```
$ ar t /usr/lib/libcrypt.a
crypt-entry.o
md5-crypt.o
md5.o
sha256-crypt.o
sha256.o
sha512-crypt.o
sha512.o
crypt.o
crypt_util.o
```

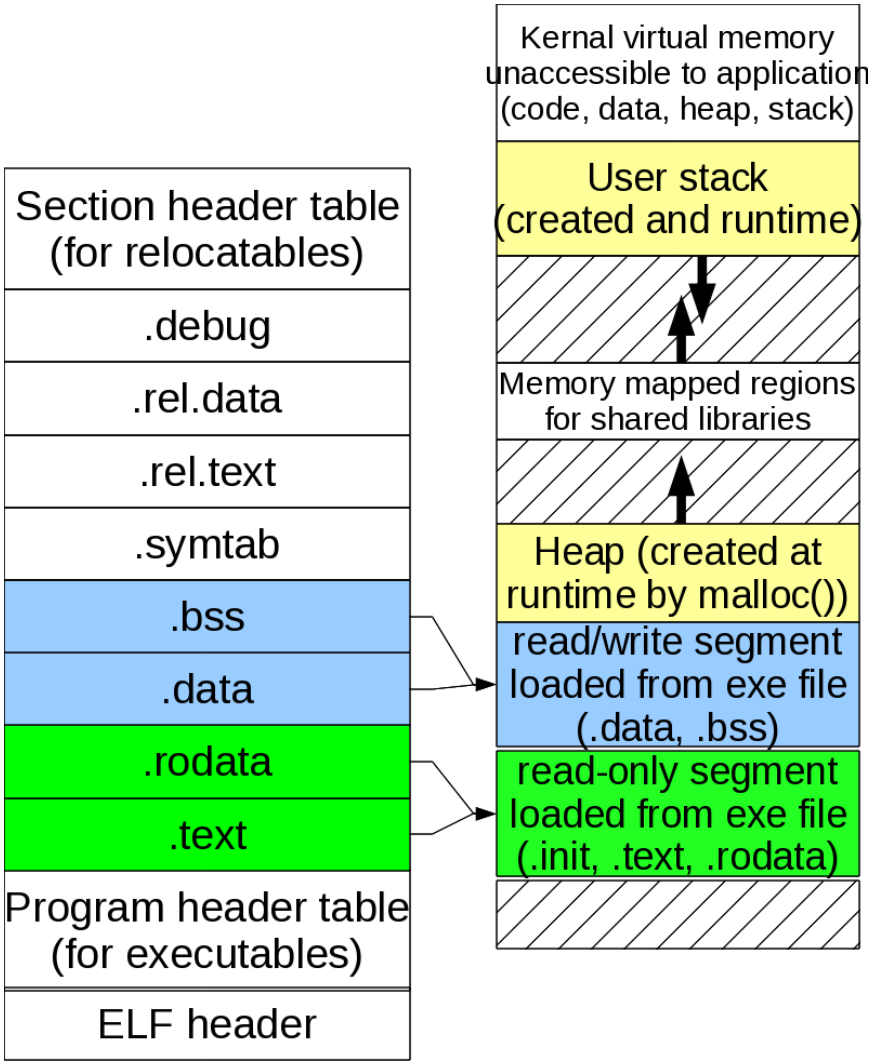
- How to make them?

```
$ ar rs all.a other.o num.o our.o
```

- How to use them?

```
$ gcc -o myExecutable main.o -lm
```

# Got an executable? *Load it!*

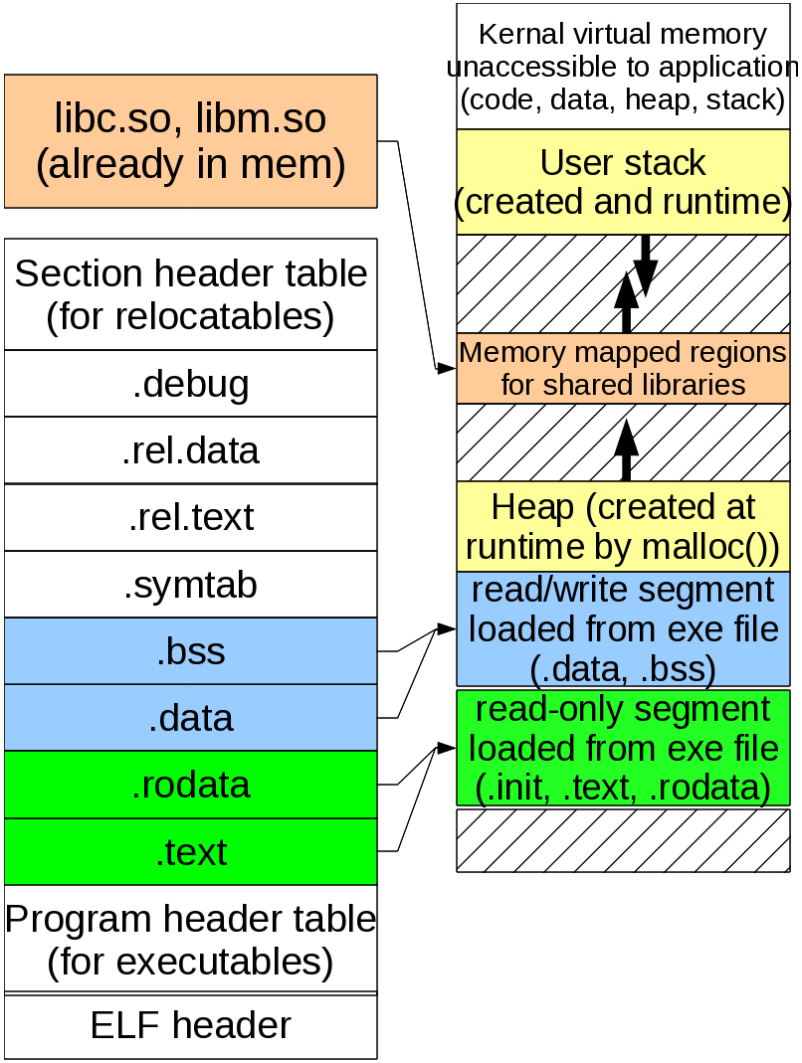



- `.text` to execute-only code pages
- `.rodata` to read-only data pages
- `.data` to read-write data pages
- `.bss` expanded to 0-initialized data pages
- Construct stack and heap pages

# Hey!  Aren't we wasting memory?

- Almost any program written in C includes `libc.a` code for `printf()`, `fgets()`, *etc*.
  - Wastes ***disk-space*** (many executables with same code)
  - (Even worse) wastes ***memory*** (memory, of course, is faster and more scarce)

# Solution: Dynamically Shared Libs




1. `gcc -c myFile.c`
2. `gcc -o myFile.o -lspecial`
3. When loading to execute, dynamically link with `libc.so`, `libm.so`, *etc*.

# Don't dynamic linking?

- Use `-static` flag to link `libc.a` instead of dynamically link with `libc.so`.
- But it will cost you though:

```
[jphillips@local]$ gcc -c hello.c
[jphillips@local]$ gcc -o hello hello.o
[jphillips@local]$ gcc -static -o helloStatic hello.c
[jphillips@local]$ ls -l hello helloStatic
-rwxrwxr-x 1    4990 2011-01-11 07:03 hello
-rwxrwxr-x 1  546491 2011-01-11 07:03 helloStatic
```

# Next time: *Loading and executing!*