CS 33

Linkers

gcc Steps

1) Compile
   - to start here, supply .c file
   - to stop here: `gcc -S` (produces .s file)
   - if not stopping here, gcc compiles directly into a .o file, bypassing the assembler
2) Assemble
   - to start here, supply .s file
   - to stop here: `gcc -c` (produces .o file)
3) Link
   - to start here, supply .o file

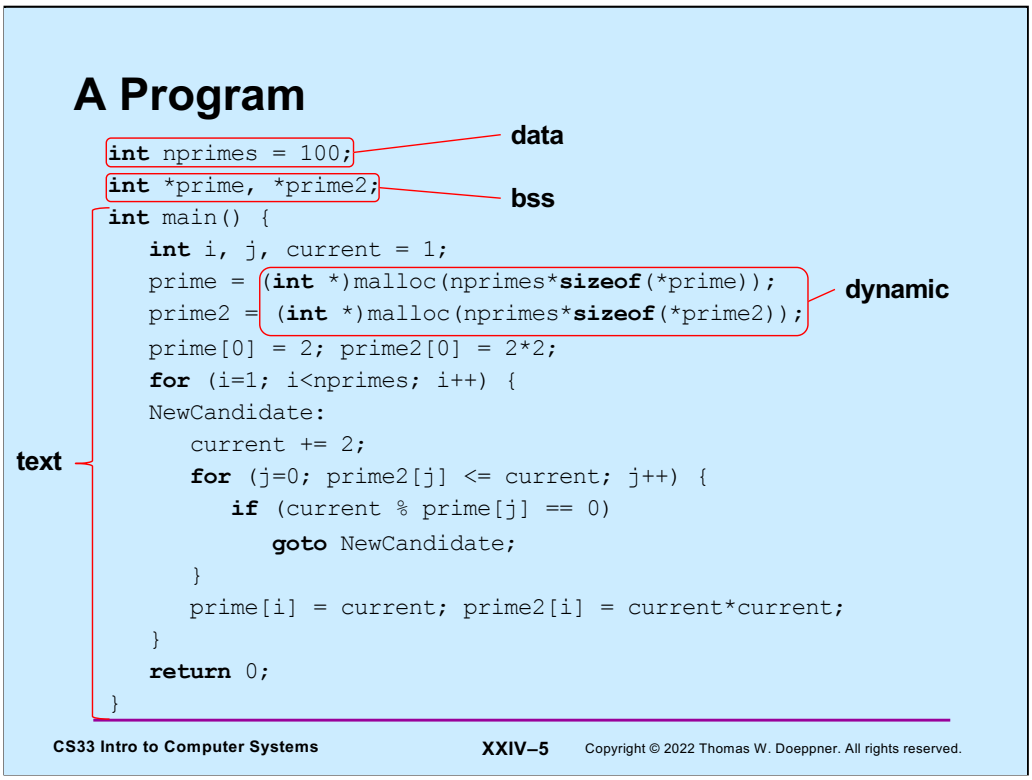
The Linker

- An executable program is one that is ready to be loaded into memory
- The linker (known as ld: /usr/bin/ld) creates such executables from:
  - object files produced by the compiler/assembler
  - collections of object files (known as libraries or archives)
  - and more we'll get to soon ...

The technology described here is current as of around 1990 and is known as static linking. We discuss static linking first, then move on to dynamic linking (in a few weeks), which is commonplace today.

Linker's Job

- Piece together components of program
  - arrange within address space
    - » code (and read-only data) goes into text region
    - » initialized data goes into data region
    - » uninitialized data goes into bss region
- Modify address references, as necessary



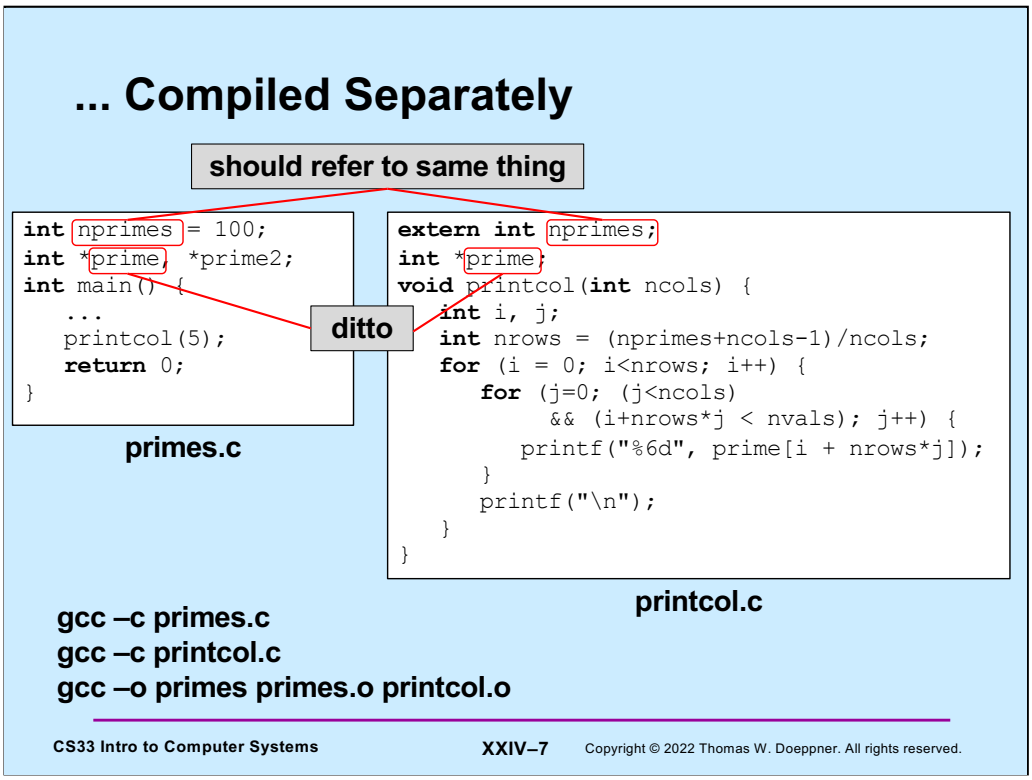
The code is an implementation of the "sieve of Eratosthenes", an early (~200 BCE) algorithm for enumerating prime numbers. The idea is to iterate through the positive integers. 2 is the first prime number. 3 is prime, since it's not divisible by 2. 4 is not prime, since it is divisible by 2. 5 is not prime, since it's not divisible by any of the primes discovered so far (5 is less than the largest's square). This continues ad infinitum.

The **malloc** function allocates storage within the dynamic region. We discuss it in detail in an upcoming lecture.

... with Output

```
int nprimes = 100;
int *prime, *prime2;
int main() {
   ...
   printcol(5);
   return 0;
}

void printcol(int ncols) {
   int i, j;
   int nrows = (nprimes+ncols-1)/ncols;
   for (i = 0; i<nrows; i++) {
      for (j=0; (j<ncols) && (i+nrows*j < nvals); j++) {
         printf("%6d", prime[i + nrows*j]);
      }
      printf("\n");
   }
}
```


What this program actually does isn't all that important for our discussion. However, it prints out the vector of prime numbers in multiple columns.



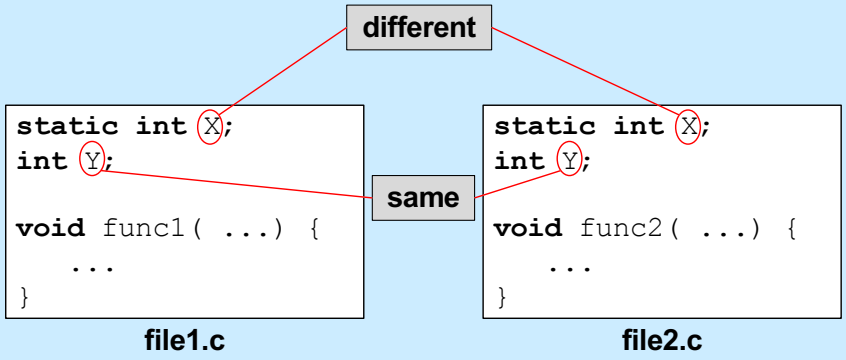
In the first two invocations of gcc, the “-c” flag tells it to compile the C code and produce an object (“.o”) file, but not to go any further (and thus not to produce an executable program). In the third invocation, gcc invokes the ld (linker) program to combine the two object files into an executable program. As we discuss soon, it will also bring in code (such as printf) from libraries.

Global Variables

- Initialized vs. uninitialized
  - initialized allocated in *data* section
  - uninitialized allocated in *bss* section
    - » implicitly initialized to zero
- File scope vs. program scope
  - *static* global variables known only within file that declares them
    - » two of same name in different files are different
    - » e.g., `static int X;`
  - non-static global variables potentially shared across all files
    - » two of same name in different files are same
    - » e.g., `int X;`

BSS is a mnemonic from an ancient assembler (not as ancient as Eratosthenes) and stands for “block started by symbol”, a rather meaningless phrase. The BSS section of the address space is where all uninitialized global and static local variables are placed. When the program starts up, this entire section is filled with zeroes.

Scope



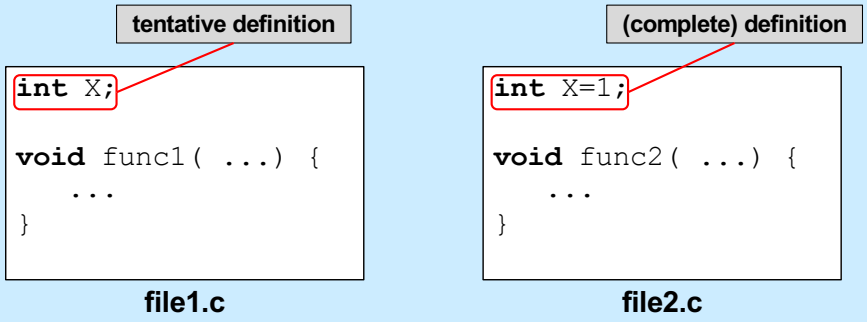
Static Local Variables

```
int *sub1() {
  int var = 1;
  …
  return &var;
  /* amazingly illegal */
}
```

```
int *sub2() {
  static int var = 1;
  …
  return &var;
  /* (amazingly) legal */
}
```


Static local variables have the same scope as other local variables, but their values are retained across calls to the procedures they are declared in. Like global variables, uninitialized static local variables are stored in the BSS section of the address space (and implicitly initialized to zero), initialized static local variables are stored in the data section of the address space.

Reconciling Program Scope (1)



Where does X go?
What's its initial value?

- **tentative definitions overridden by compatible (complete) definitions**
- **if not overridden, then initial value is zero**

X goes in the data section and has an initial value of 1. If file2.c did not exist, then X would go in the bss section and have an initial value of 0. Note that the textbook calls tentative definitions "weak definitions" and complete definitions "strong definitions". This is non-standard terminology and conflicts with the standard use of the term "weak definition," which we discuss shortly.

Reconciling Program Scope (2)

```
int X=2;

void func1( ...) {
    ...
}
```

file1.c

```
int X=1;

void func2( ...) {
    ...
}
```

file2.c

What happens here?

In this case we have conflicting definitions of X — this will be flagged (by the ld program) as an error.

Reconciling Program Scope (3)

```
int X=1;

void func1( ...) {
    ...
}
```

file1.c

```
int X=1;

void func2( ...) {
    ...
}
```

file2.c

Is this ok?

No; it is flagged as an error: only one file may supply an initial value.

Reconciling Program Scope (4)

```
extern int X;

void func1( ...) {
    ...
}
```

file1.c

```
int X=1;

void func2( ...) {
    ...
}
```

file2.c

What's the purpose of "extern"?

The "extern" means that this file will be using X, but it depends on some other file to provide a definition for it, either initialized of uninitialized. If no other file provides a definition, then ld flags an error.

If the "extern" were not there, i.e., if X were declared simply as an "int" in file1.c, then it wouldn't matter if no other file provided a definition for X — X would be allocated in bss with an implicit initial value of 0.

Note: this description of extern is how it is implemented by gcc. The official C99 standard doesn't require this behavior, but merely permits it. It also permits "extern" to be essentially superfluous: its presence may mean the same thing as its absence.

The C11 standard more-or-less agrees with the C99 standard. Moreover, it explicitly allows a declaration of the form "extern int X=1;" (i.e., initialization), which is not allowed by gcc.

For most practical purposes, whatever gcc says is the law ...

Default Values (1)

```
float seed = 1.0;

int PrimaryFunc(float arg) {
  ...
  SecondaryFunc(arg + seed);
  ...
}

void SecondaryFunc(float arg) {
  ...
}
```


Default Values (2)

```
float seed = 2.0; /* want a different seed */

int main() {
  ...
  PrimaryFunc(floatingValue);
  ...
}

void SecondaryFunc(float arg) {
  /* would like to override default version */
  ...
}
```


The code in this slide will use the code in the previous slide, however, we would like to override the previous slide's definitions of **seed** and **SecondaryFunc**. The linker will not allow this and would flag "duplicate-definition" errors.

Default Values (3)

```
__attribute__((weak)) float seed = 1.0;

int PrimaryFunc(float arg) {
  ...
  SecondaryFunc(arg + seed);
  ...
}

void __attribute__((weak)) SecondaryFunc(float arg) {
  ...
}
```


By defining **seed** and **SecondaryFunc** to be **weak** symbols, we can indicate that they may be overridden. If there is no other definition for a weak symbol, the "weak" definition will be used. Otherwise, the other definition will be used.

Does Location Matter?

```
int main(int argc, char *[]) {
   return(argc);
}
```

```
main:
  pushq %rbp    ; push frame pointer
  movq  %rsp, %rbp     ; set frame pointer to point to new frame
  movl  %edi, %eax     ; put argc into return register (eax)
  movq  %rbp, %rsp     ; restore stack pointer
  popq  %rbp    ; pop stack into frame pointer
  ret          ; return: pops end of stack into rip
```


This rather trivial program references memory via only rsp and rip (rbp is set from rsp). Its code contains no explicit references to memory, i.e., it contains no explicit addresses.

Location Matters …

```
int X=6;
int *aX = &X;

int main() {
    void subr(int);
    int y=*aX;
    subr(y);
    return(0);
}

void subr(int i) {
    printf("i = %d\n", i);
}
```


We don't need to look at the assembler code to see what's different about this program: the machine code produced for it can't simply be copied to an arbitrary location in our computer's memory and executed. The location identified by the name **aX** should contain the address of the location containing **X**. But since the address of **X** will not be known until the program is copied into memory, neither the compiler nor the assembler can initialize **aX** correctly. Similarly, the addresses of **subr** and **printf** are not known until the program is copied into memory — again, neither the compiler nor the assembler would know what addresses to use.

Coping

- Relocation
  - modify internal references according to where module is loaded in memory
  - modules needing relocation are said to be *relocatable*
    - » which means they *require* relocation
  - the compiler/assembler provides instructions to the linker on how to do this

A Revised Version of Our Program

```
extern int X;
int *aX = &X;
int Y = 1;

int main() {
    void subr(int);
    int y = *aX+Y;
    subr(y);
    return(0);
}
```

main.c

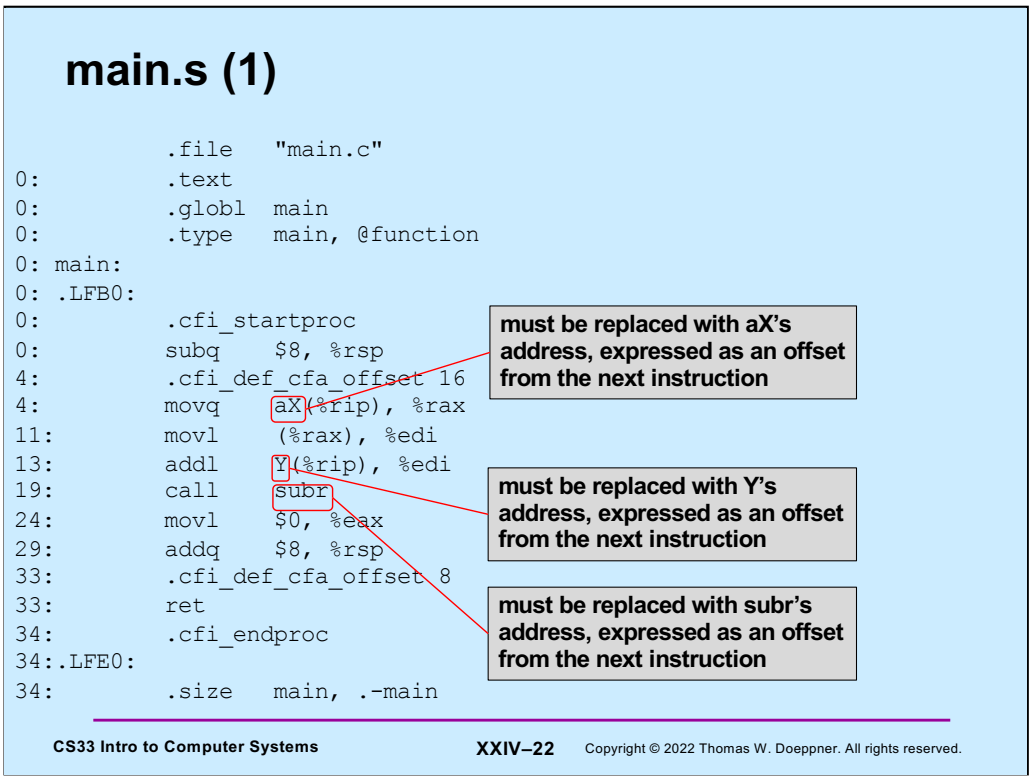
```
#include <stdio.h>
int X;

void subr(int XX) {
    printf("XX = %d\n", XX);
    printf("X = %d\n", X);
}
```

subr.c

```
gcc –o prog –O1 main.c subr.c
```


Note that what we did, in order to obtain what's in the next few slides, was:

gcc –S –O1 main.c subr.c
gcc –c main.s subr.s
gcc –o prog main.o subr.o



Note that a symbol's value is the location of what it refers to. The compiler/assembler knows what the values (i.e., locations) of **aX** and **Y** are relative to the beginning of this module's data section (next slide), but has no idea what **subr**'s value is. It is the linker's job to provide final values for these symbols, which will be the addresses of the corresponding C constructs when the program is loaded into memory. The linker will adjust these values to obtain the locations of what they refer to relative to the value of register rip when the referencing instructions are executed.

One might ask why these locations are referred to using offsets from the instruction pointer (also known as the program counter), rather than simply using their addresses. The reason is to save space: the addresses would be 64 bits long, but the offsets are only 32 bits long.

The ".file" directive supplies information to be placed in the object file and the executable of use to debuggers — it tells them what the source-code file is.

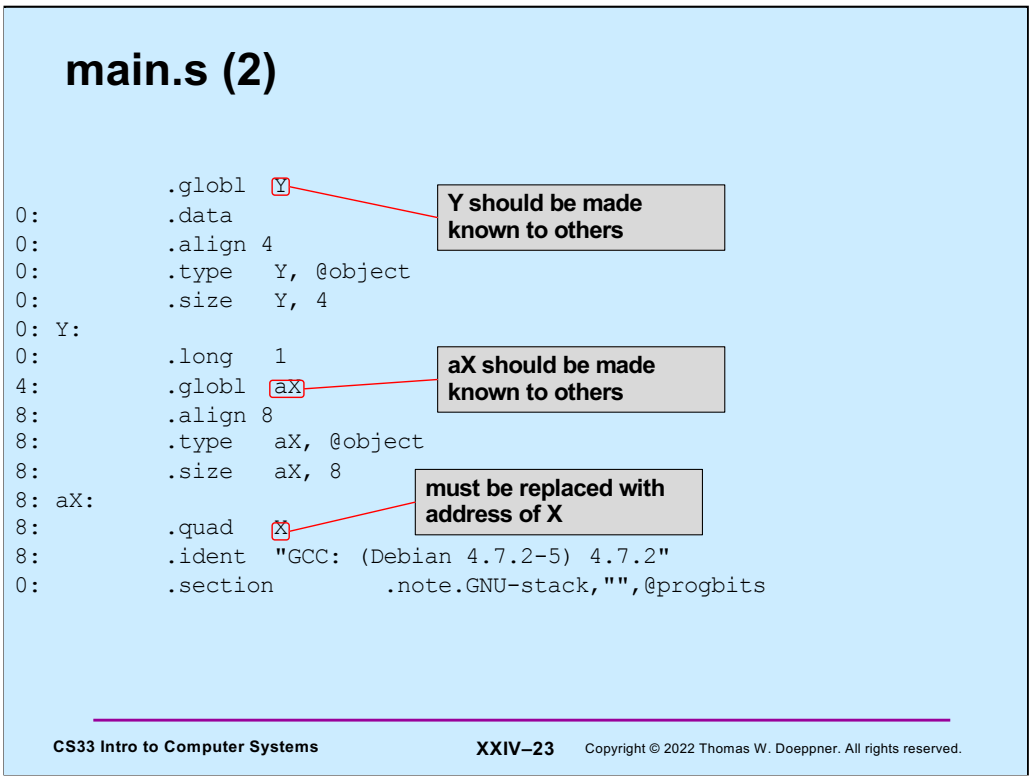
The ".globl" directive indicates that the symbol, defined here, will be used by other modules, and thus should be made known to the linker.

The ".type" directive indicates how the symbol is used. Two possibilities are function and object (meaning a data object).

The " .size" directive indicates the size that should be associated with the given symbol.

The directives starting with ".cfi_" are there for the sake of the debugger. They generate auxiliary information stored in the object file (but not executed) that describes the relation between the stack pointer (%rsp) and the beginning of the stack frame. Thus

they compensate for the lack of a standard frame-pointer register (%esp for IA32). In particular, they emit data going into a table that is used by a debugger (such as gdb) to determine, based on the value of the instruction pointer (%rip) and the stack pointer, where the beginning of the current stack frame is.



The symbol **X**'s value is, at this point, unknown.

The ".data" directive indicates that what follows goes in the data section.

The ".long" directive indicates that storage should be allocated for a long word.

The ".quad" directive indicates that storage should be allocated for a quad word.

The ".align" directive indicates that the storage associated with the symbol should be aligned, in the cases here, on 4-byte and 8-byte boundaries (i.e., the least-significant two bits and three bits of their addresses should be zeroes).

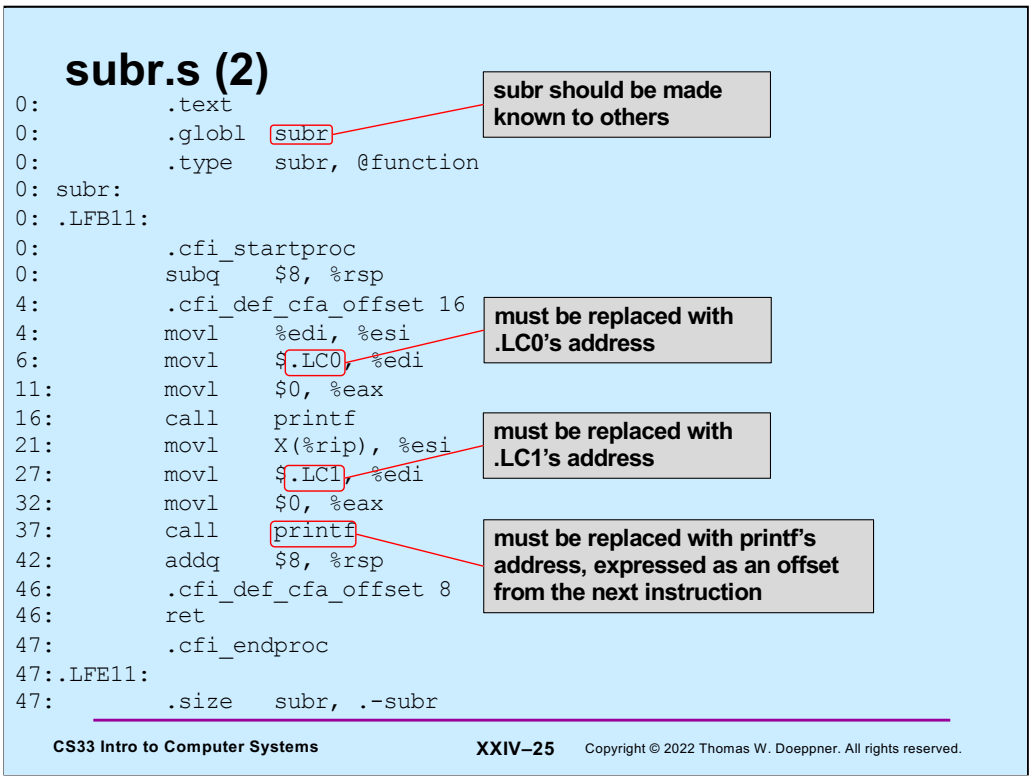
The ".ident" directive indicates the software used to produce the file and its version.

The ".section" directive used here is supplied by gcc by default and indicates that the program should have a non-executable stack (this is important for security purposes).

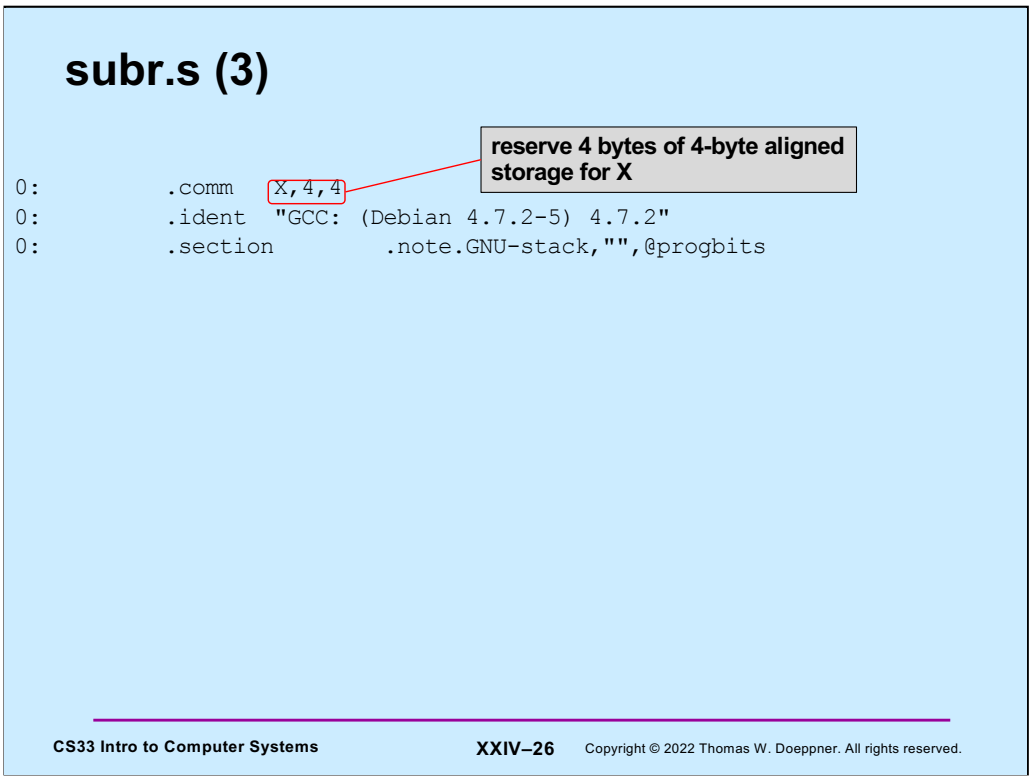
subr.s (1)

```
          .file   "subr.c"
0:        .section        .rodata.str1.1,"aMS",@progbits,1
0: .LC0:
0:        .string "XX = %d\n"
9: .LC1:
9:        .string "X = %d\n"
```


The ".section" directive here indicates that what follows should be placed in read-only storage (and will be included in the text section). Furthermore, what follows are strings with a one-byte-per-character encoding that require one-byte (i.e., unrestricted) alignment. This information will ultimately be used by the linker to reduce storage by identifying strings that are suffices of others.



Note that the compiler has generated **movl** instructions (copying 32 bits) for copying the addresses of .LC0 and .LC1: it's assuming that both addresses will fit in 32 bits (in other words, that the text section of the program will be less than 2^{32} bytes long — probably a reasonable assumption.



The ".comm" directive indicates here that four bytes of four-byte aligned storage are required for X in BSS. "comm" stands for "common", which is what the Fortran language uses to mean the same thing as BSS. Since Fortran predates pretty much everything (except for Eratosthenes), its terminology wins (at least here).

Quiz 1

```
int X;
int func(int arg) {
  static int Y;
  int Z;

  ...

}
```

Which of *X*, *Y*, *Z*, and *arg* would the compiler know the addresses of at compile time?

a) all
b) just *X* and *Y*
c) just *arg* and *Z*
d) none

ELF

- **Executable and linking format**
  - **used on most Unix systems**
    - » **pretty much all but Mac OS**
  - **defines format for:**
    - » **.o (object) files**
    - » **.so (shared object) files**
    - » **executable files**

Complete documentation for ELF (much more than you'd ever want to know) can be found at http://refspecs.linuxbase.org/elf/elf.pdf.

Doing Relocation

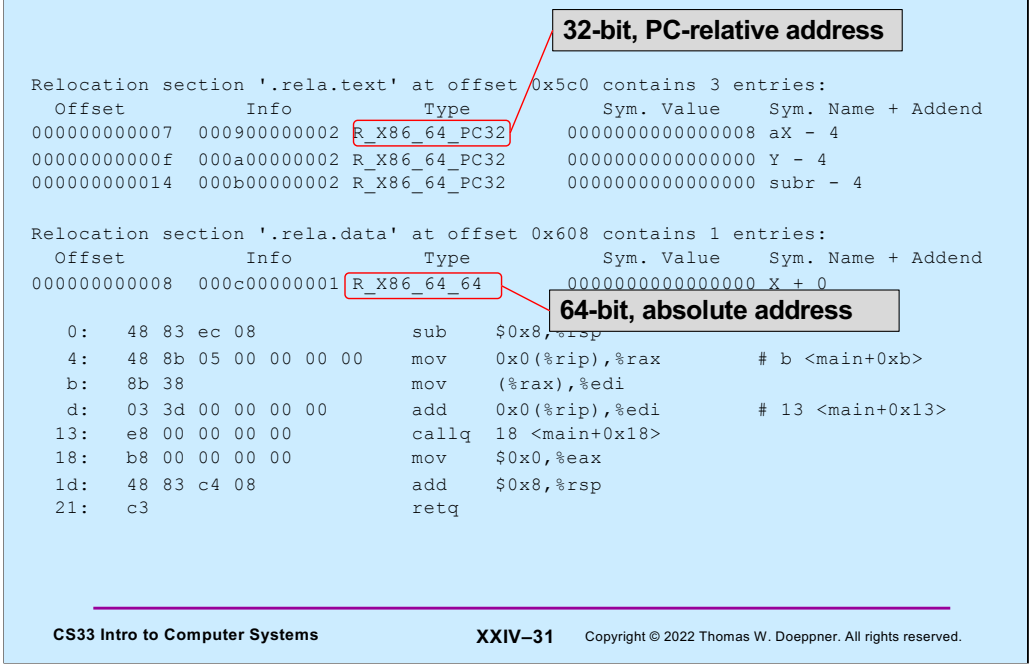
- Linker is provided instructions for updating object files
  - lots of ways addresses can appear in machine code
  - three in common use on x86-64
    - » 32-bit absolute addresses
      - used for text references
    - » 64-bit absolute addresses
      - used for data references
    - » 32-bit PC-relative addresses
      - offset from current value of rip
      - used for text and data references

main.o (1)

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          296 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

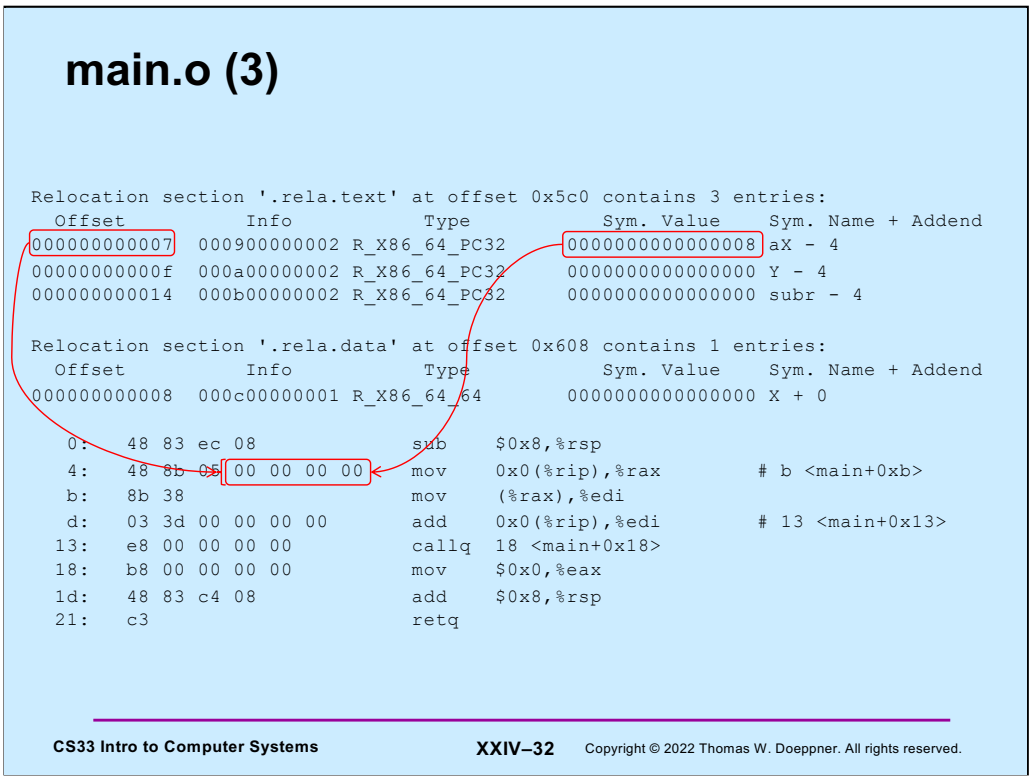

In this and the next few slides we examine the contents of the object files. This information was obtained by using the program “readelf”.

main.o (2)




```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000007  000900000002 R_X86_64_PC32     0000000000000008 aX - 4
00000000000f  000a00000002 R_X86_64_PC32     0000000000000000 Y - 4
000000000014  000b00000002 R_X86_64_PC32     0000000000000000 subr - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000008  000c00000001 R_X86_64_64       0000000000000000 X + 0

   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # b <main+0xb>
   b:   8b 38                   mov    (%rax),%edi
   d:   03 3d 00 00 00 00       add    0x0(%rip),%edi        # 13 <main+0x13>
  13:   e8 00 00 00 00          callq  18 <main+0x18>
  18:   b8 00 00 00 00          mov    $0x0,%eax
  1d:   48 83 c4 08             add    $0x8,%rsp
  21:   c3                      retq
```


The first relocation section above contains information about the text portion of the program – executable code. The second relocation section contains information about the data portion of the program.



The first three relocation instructions are for the text associated with **main**. The first relocation instruction specifies that offset 0x07 of the text region should be updated by adding to it the PC-relative version of the address ultimately associated with symbol **aX**. This will be, of course, where **aX** is located in the data region. The field in the "Info" column encodes what's given more clearly in the next three columns. The "0009" identifies a field in the symbol table (not shown) that says the symbol's name is **aX** and that its value may be found at offset 0x08 (the "Sym. Value" column) in this module's contribution to the data section. The "0002" in the "Info" column says that the type of reference to **aX** is 32-bit PC-relative (the "Type" column).

To handle PC-relative addressing, the linker blindly assumes that the PC's value (the contents of register **rip**) is the address of the field within the instruction that's being modified (offset 7 in this example). Thus, for example, if the text section for **main** were loaded into memory at address 0x1000, the linker would assume that the value contained in register **rip** would be 0x1007, where the source operand of the first **mov** instruction is being located. If symbol **aX** is at, say, location 0x10008, the linker would modify the last four bytes of the **mov** instruction by replacing its contents with 0xf001 (= 0x10008 – 0x1001). However, by the time **rip** is used to access the source operand, it will already have been incremented to point to the next instruction (the second **mov**). If the PC-relative address of 0xf001 were actually used, it would point to four bytes beyond the location of **aX**. So, to correct for this, rather than use the value of symbol **aX** directly, the linker is instructed to use four less than this value (hence the "addend" of -4).

main.o (4)

```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000007  000900000002 R_X86_64_PC32     0000000000000008 aX - 4
00000000000f  000a00000002 R_X86_64_PC32     0000000000000000 Y - 4
000000000014  000b00000002 R_X86_64_PC32     0000000000000000 subr - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000008  000c00000001 R_X86_64_64       0000000000000000 X + 0

   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # b <main+0xb>
   b:   8b 38                   mov    (%rax),%edi
   d:   03 3d 00 00 00 00       add    0x0(%rip),%edi        # 13 <main+0x13>
  13:   e8 00 00 00 00          callq  18 <main+0x18>
  18:   b8 00 00 00 00          mov    $0x0,%eax
  1d:   48 83 c4 08             add    $0x8,%rsp
  21:   c3                      retq
```


The second relocation instruction specifies that offset 0x0f of the text region should be updated by adding to it the PC-relative version of the address ultimately associated with symbol **Y**. This will be, of course, where **Y** is located in the data region.

main.o (5)

```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000007  000900000002 R_X86_64_PC32     0000000000000008 aX - 4
00000000000f  000a00000002 R_X86_64_PC32     0000000000000000 Y - 4
000000000014  000b00000002 R_X86_64_PC32     0000000000000000 subr - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000008  000c00000001 R_X86_64_64       0000000000000000 X + 0

   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # b <main+0xb>
   b:   8b 38                   mov    (%rax),%edi
   d:   03 3d 00 00 00 00       add    0x0(%rip),%edi        # 13 <main+0x13>
  13:   e8 00 00 00 00          callq  18 <main+0x18>
  18:   b8 00 00 00 00          mov    $0x0,%eax
  1d:   48 83 c4 08             add    $0x8,%rsp
  21:   c3                      retq
```


The third relocation instruction specifies that offset 0x14 of the text region should be updated by adding to it the PC-relative version of the address ultimately associated with symbol **subr**. This will be, of course, where **subr** is located in the text region.

main.o (6)

```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000007  000900000002 R_X86_64_PC32     0000000000000008 aX - 4
00000000000f  000a00000002 R_X86_64_PC32     0000000000000000 Y - 4
000000000014  000b00000002 R_X86_64_PC32     0000000000000000 subr - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000008  000c00000001 R_X86_64_64       0000000000000000 X + 0

   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # b <main+0xb>
   b:   8b 38                   mov    (%rax),%edi
   d:   03 3d 00 00 00 00       add    0x0(%rip),%edi        # 13 <main+0x13>
  13:   e8 00 00 00 00          callq  18 <main+0x18>
  18:   b8 00 00 00 00          mov    $0x0,%eax
  1d:   48 83 c4 08             add    $0x8,%rsp
  21:   c3                      retq
```


The final relocation instruction, which is for the data associated with **main**, specifies that offset 0x08 of this module's contribution to the data region should be updated by adding to it the address of symbol **X**, once it's determined.

subr.o (1)

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          312 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```


subr.o (2)

```
Relocation section '.rela.text' at offset 0x5b0 contains 5 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000007  00050000000a R_X86_64_32       0000000000000000 .rodata.str1.1 + 0
000000000011  000a00000002 R_X86_64_PC32     0000000000000000 printf - 4
000000000017  000b00000002 R_X86_64_PC32     0000000000000004 X - 4
00000000001c  00050000000a R_X86_64_32       0000000000000000 .rodata.str1.1 + 9
000000000026  000a00000002 R_X86_64_PC32     0000000000000000 printf - 4

   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   89 fe                   mov    %edi,%esi
   6:   bf 00 00 00 00          mov    $0x0,%edi
   b:   b8 00 00 00 00          mov    $0x0,%eax
  10:   e8 00 00 00 00          callq  15 <subr+0x15>
  15:   8b 35 00 00 00 00       mov    0x0(%rip),%esi        # 1b <subr+0x1b>
  1b:   bf 00 00 00 00          mov    $0x0,%edi
  20:   b8 00 00 00 00          mov    $0x0,%eax
  25:   e8 00 00 00 00          callq  2a <subr+0x2a>
  2a:   48 83 c4 08             add    $0x8,%rsp
  2e:   c3                      retq
```

```
.rodata.str1.1:
XX = %d\n\0X = %d\n\0
```


The relocation section for **subr** includes entries for relocating the references to the strings passed to the calls to **printf**. For both references, the symbol name is ".rodata.str1.1", which refers to the section containing both strings: the first is at offset 0, the second at offset 9. Hence the addend value is used to indicate which string is being referenced.

Quiz 2

Consider the following 5-byte instruction:

```
ea 00 00 00 00
```

`ea` is the opcode for the call instruction with a 32-bit PC-relative operand.

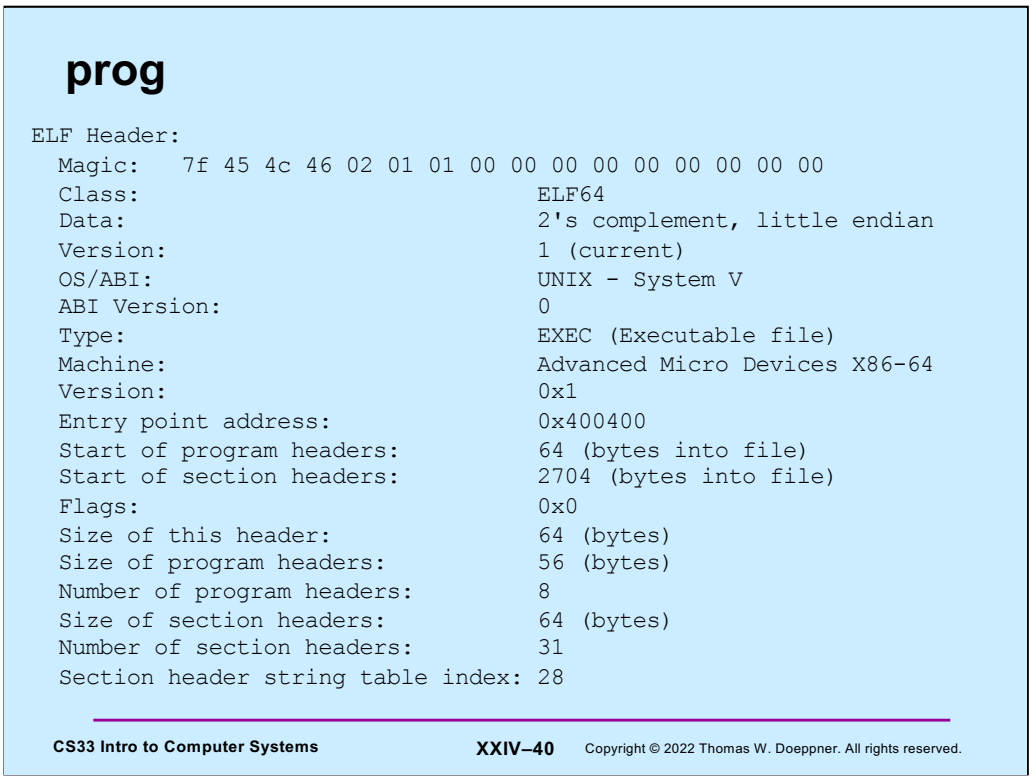
Suppose this instruction is at location 0x1000. To what location would control be transferred if the instruction were executed as is?

a) 0
b) 0x1000
c) 0x1001
d) 0x1005

printf.o

```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
0000000002d3  000b00000002 R_X86_64_PC32     0000000000000000 write - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
0000000000d3  000c00000001 R_X86_64_64       0000000000000000 StandardFiles + 0
```


To simplify our discussion a bit, the version of **printf** shown here is not what is really provided the C library, but is much simpler. Assume "StandardFiles" is an array of per-file information required by **printf** (and other I/O routines). **Printf** calls **write**, the system call that actually performs the write operation.



This is the ELF header from the final executable created for our fully linked program.

Final Result

Symbol	Value	Size	
_start	0x400400	0x60	text
main	0x400460	0x3f	
subr	0x4004a0	0x30	
printf	0x4004d0	0x12000	
write	0x4124d0	0x30	
.rodata	0x412500	0x9	
aX	0x413000	0x8	data
Y	0x413008	0x8	
StandardFiles	0x413010	0x1000	
X	0x414010	0x8	bss

The slide shows the final layout of the address space. (Though keep in mind that, as already mentioned, what's there for **printf** is simplified.) Note that a special entry "_start" has been added. This is what is actually called first. It then calls **main**. When **main** returns, it returns to **_start**, which then causes the process to terminate (by calling the operating system's "exit" function).

If you are exceptionally sharp-eyed, you might notice that .rodata refers to an area (within text) that's only 9 bytes long, but that the sum of the lengths of the two format strings passed to the two calls to **printf** in **subr** was 17 bytes. The linker actually determines that the second string is a suffix of the first, and thus it's only necessary to store the first (and thus the reference to the second string is a reference to the second character of the first).

One might ask why text starts at 0x400400 (= 4,195,328 in decimal) rather than at a much smaller value (such as 0). The answer is that there's other "stuff" at lower addresses, some of which we'll discuss later. However, it's important that nothing be at location zero (in fact, nothing should be in the first "page" of memory, which is either the first 4k bytes or the first 2M bytes on the x86-64, depending on how configured) — this is so that that page can be marked "inaccessible" and thus all attempts to use a zero (null) pointer will fail.