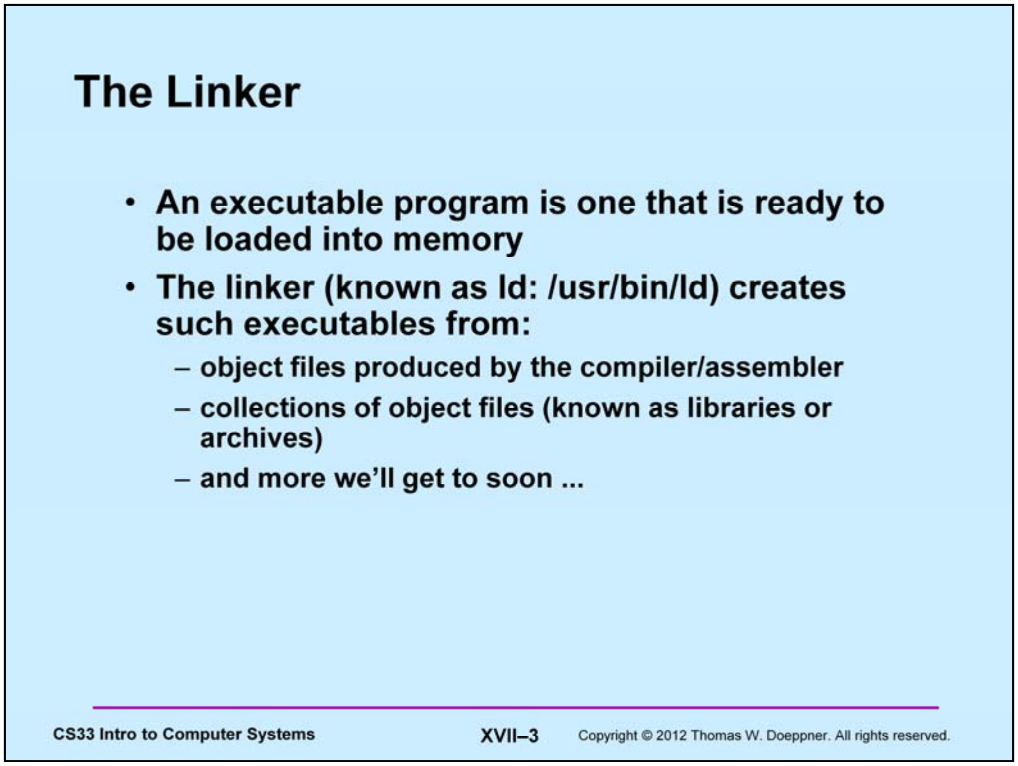
CS 33

Linkers and Loaders

gcc Steps

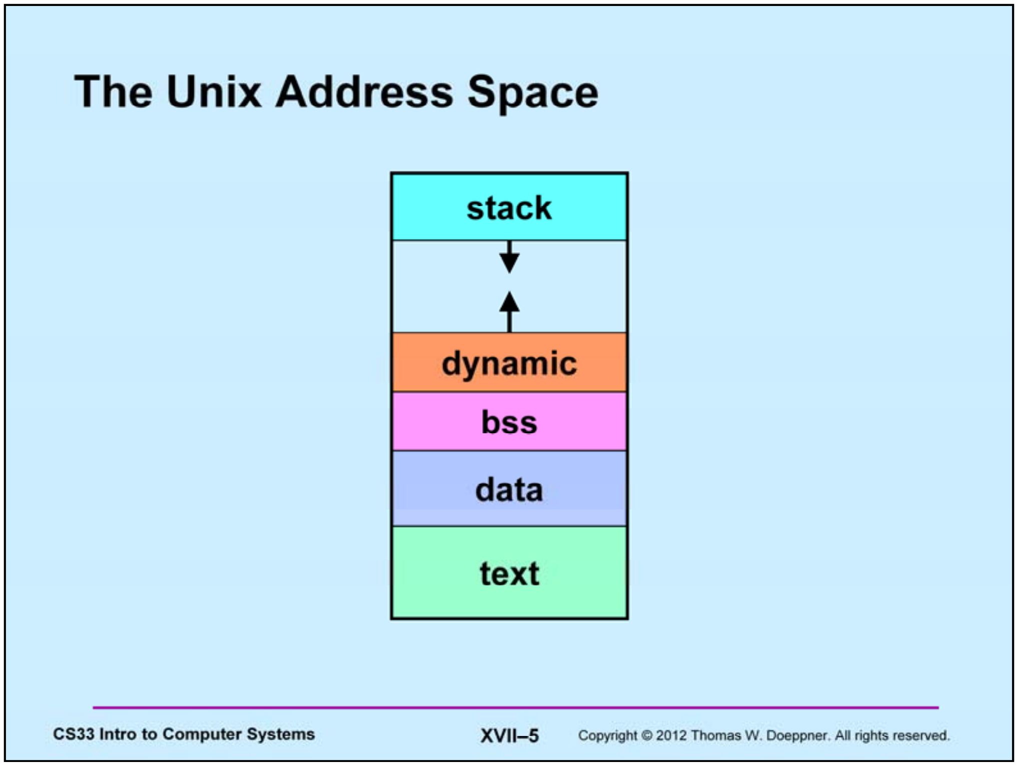
1) **Compile**
   - to start here, supply .c file
   - to stop here: `gcc -S` (produces .s file)
   - (if not stopping here, gcc compiles directly into machine code, bypassing the assembler)
2) **Assemble**
   - to start here, supply .s file
   - to stop here: `gcc -c` (produces .o file)
3) **Link**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



The 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.

Processes and Address Spaces

- The *process*
  - OS entity representing a computation
    - » sort of a virtual computer
    - » encapsulates
      - storage
        - containing code and data
          - address space
          - open files
      - identity



A Unix process's address space appears to be three regions of memory: a read-only *text* region (containing executable code); a read-write region consisting of initialized *data* (simply called data), uninitialized data (*BSS*—a directive from an ancient assembler (for the IBM 704 series of computers), standing for Block Started by Symbol and used to reserve space for uninitialized storage), and a *dynamic area*; and a second read-write region containing the process's user *stack* (a standard Unix process contains only one thread of control).

The first area of read-write storage is often collectively called the data region. Its dynamic portion grows in response to *sbrk* system calls. Most programmers do not use this system call directly, but instead use the *malloc* and *free* library routines, which manage the dynamic area and allocate memory when needed by in turn executing *sbrk* system calls.

The stack region grows implicitly: whenever an attempt is made to reference beyond the current end of stack, the stack is implicitly grown to the new reference. (There are system-wide and per-process limits on the maximum data and stack sizes of processes.)

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

A Program

```
int nprimes = 100;
int *prime, *prime2;
int main() {
   int i, j, current = 1;
   prime = (int *)malloc(nprimes*sizeof(*prime));
   prime2 = (int *)malloc(nprimes*sizeof(*prime2));
   prime[0] = 2; prime2[0] = 2*2;
   for (i=1; i<nprimes; i++) {
   NewCandidate:
      current += 2;
      for (j=0; prime2[j] <= current; j++) {
         if (current % prime[j] == 0)
            goto NewCandidate;
      }
      prime[i] = current; prime2[i] = current*current;
   }
   return 0;
}
```

data

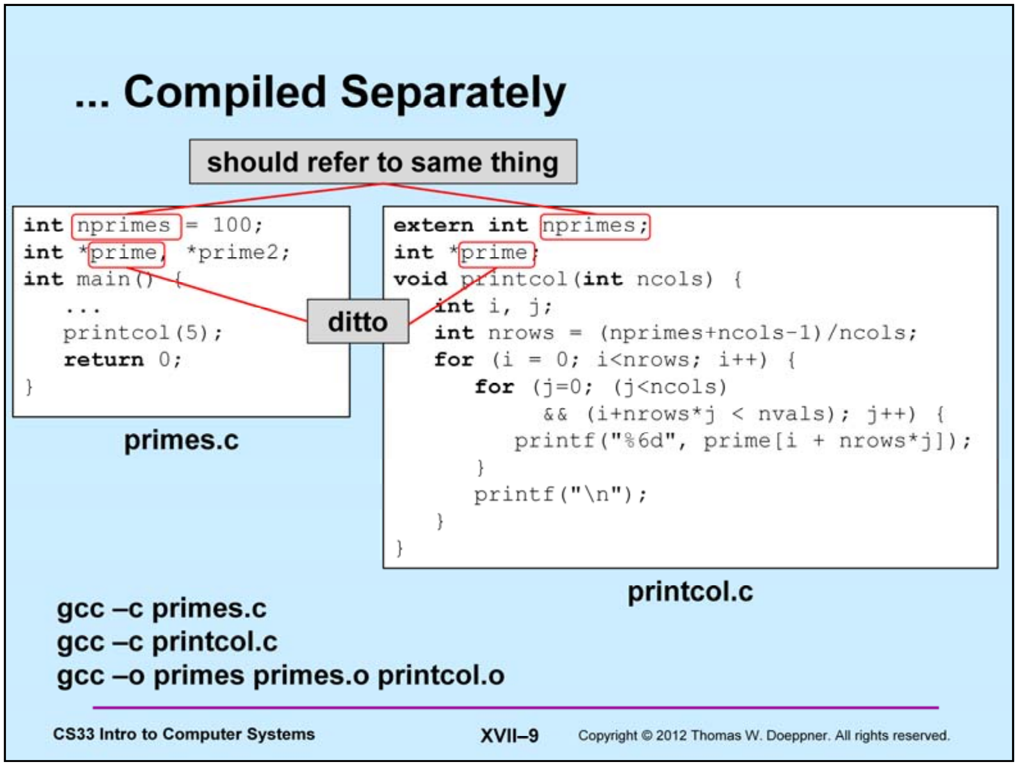
bss

dynamic

text

... 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");
   }
}
```



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;`

Scope

different

```c
static int X;
int Y;

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

file1.c

same

```c
static int X;
int Y;

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

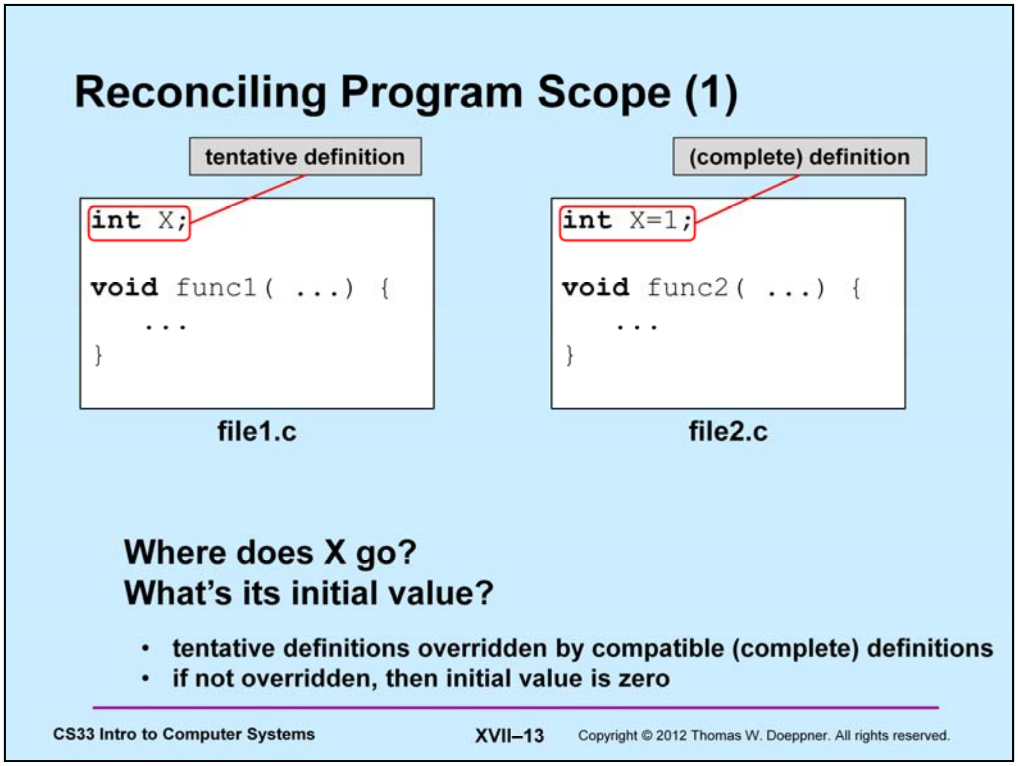
file2.c

Static Local Variables

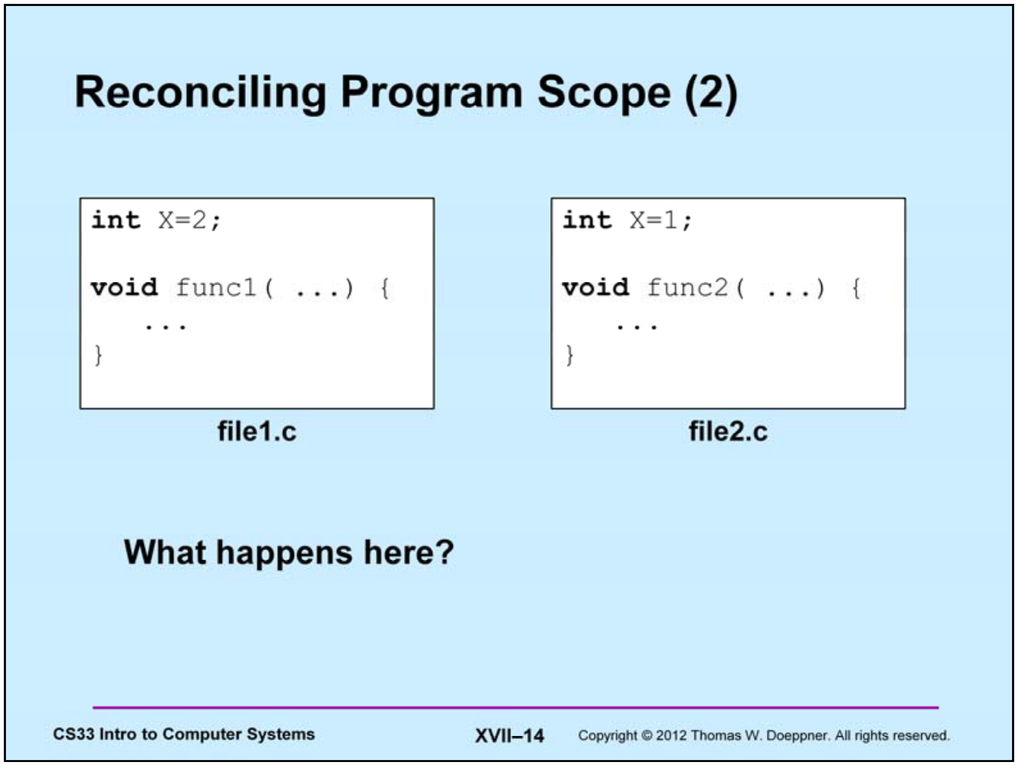
```
int *sub1() {
  int var;
  …
  return &var;
  /* amazingly illegal */
}
```

```
int *sub2() {
  static int var;
  …
  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. They are stored in the data section of the address space.



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 another use of the term "weak definition," which we discuss shortly.



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.

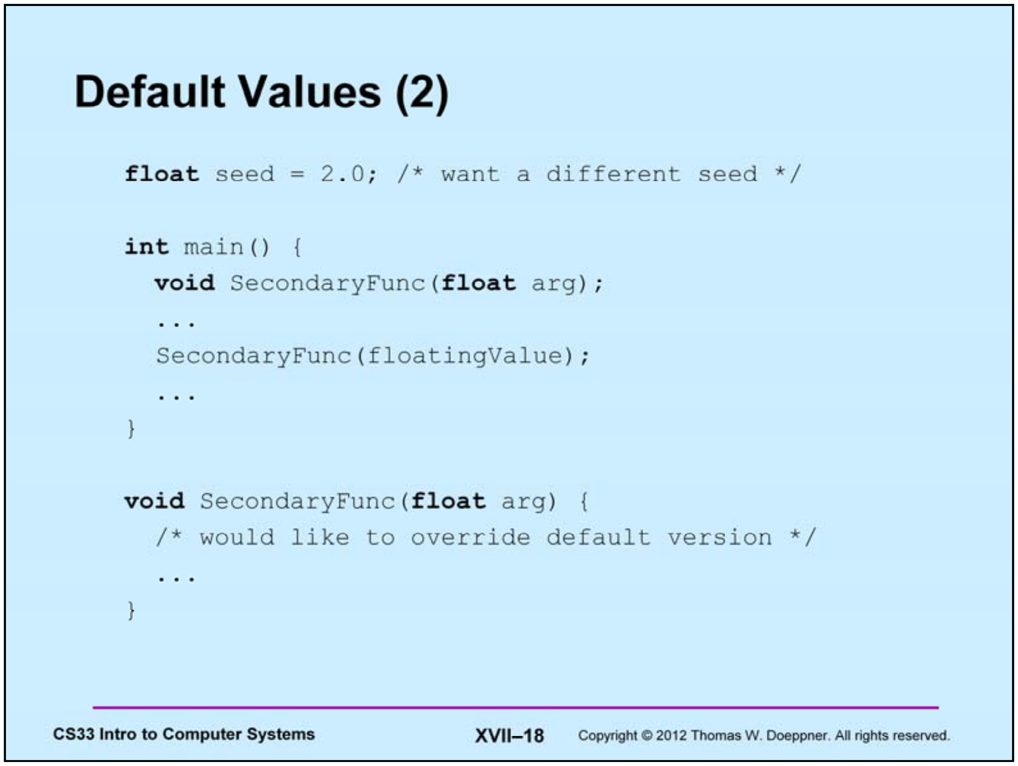
Note: this description of extern is how it is implemented by gcc. The official C99 standard doesn't require this behavior, but merely permits it. It also permits "extern" to be essentially superfluous: its presence may mean the same thing as its absence.

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(int arg) {
  void SecondaryFunc(float arg);
  ...
}

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



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 would not allow this and would flag "duplicate-definition" errors.

Default Values (3)

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

int PrimaryFunc(int arg) {
  void SecondaryFunc(float arg);
  ...
}

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:
  pushl %ebp    ; push frame pointer
  movl  %esp, %ebp     ; set frame pointer to point to new frame
  movl  8(%ebp), %eax  ; put argc into return register (eax)
  movl  %ebp, %esp     ; restore stack pointer
  popl  %ebp    ; pop stack into frame pointer
  ret           ; return: pops end of stack into eip
```


This rather trivial program references memory via only esp and eip (ebp is set from esp). 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 Revision

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

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

main.c

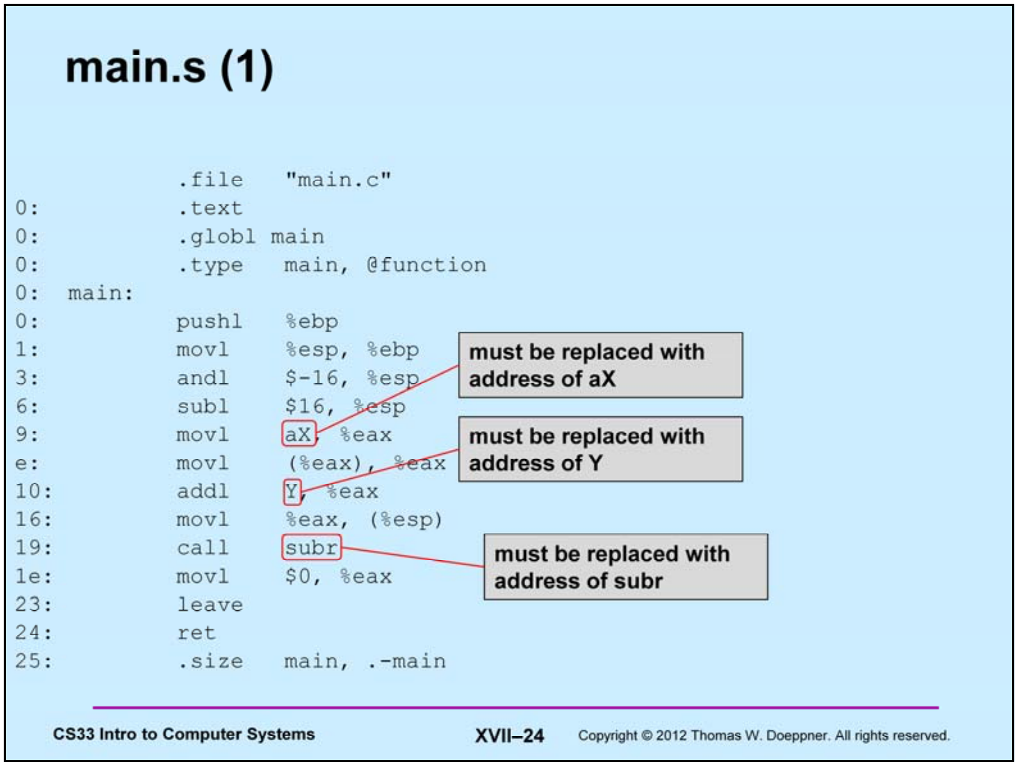
```
#include <stdio.h>
int X;

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

subr.c

```
gcc -o -O1 prog main.c subr.c
```


Note that what we actually 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 its location. The symbols *main*, *aX*, and *Y* have tentative values of zero, since the compiler/assember don't know otherwise. 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 ".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.

main.s (2)

```
         .globl   aX
0:       .data
0:       .align   4
0:       .type    aX, @object
0:       .size    aX, 4
0:  aX:
0:       .long    X
4:       .globl   Y
4:       .align   4
4:       .type    Y, @object
4:       .size    Y, 4
4:  Y:
4:       .long    1
8:       .ident   "GCC: (Debian 4.4.5-8) 4.4.5"
0:       .section        .note.GNU-stack,"",@progbits
```

must be replaced with address of X

Y should be made known to others

The symbol *X*'s value is, at this point, also 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 ".align" directive indicates that the storage associated with the symbol should be aligned, in the cases here, on 4-byte boundaries (i.e., the least-significant two 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 in all cases and indicates that the program should have a non-executable stack.

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.

subr.s (2)

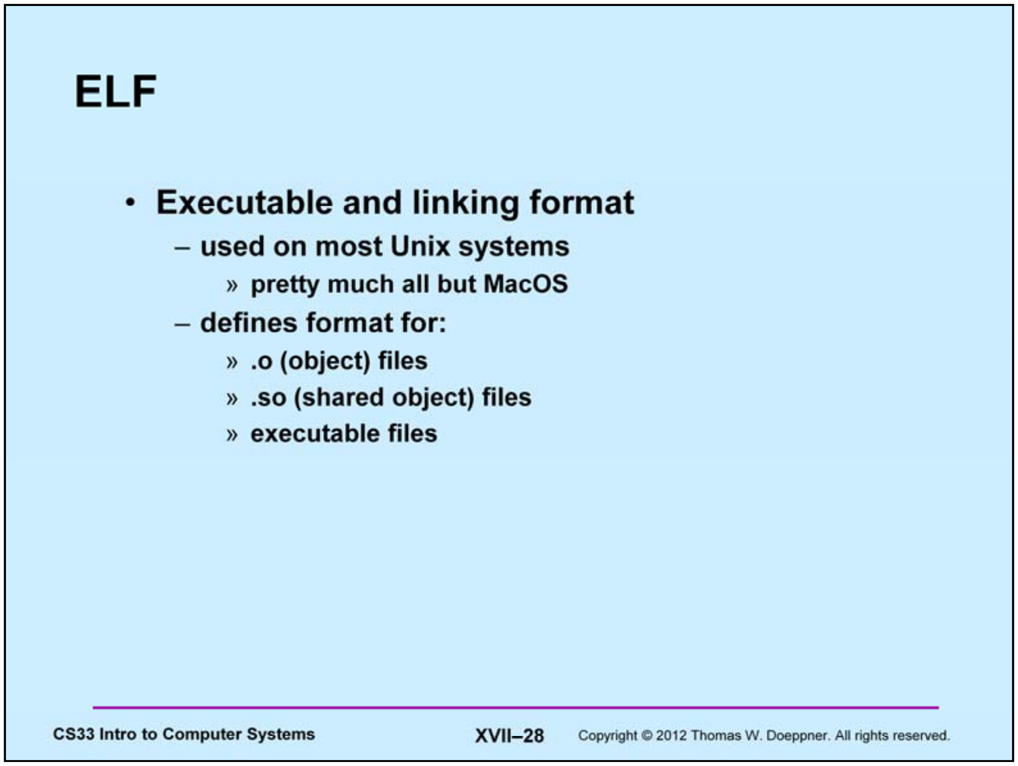
```
0:          .text
0:          .globl subr
0:          .type   subr, @function
0:   subr:
0:          pushl   %ebp
1:          movl    %esp, %ebp
3:          subl    $24, %esp
6:          movl    8(%ebp), %eax
9:          movl    %eax, 4(%esp)
d:          movl    $.LC0, (%esp)
14:         call    printf
19:         movl    X, %eax
1e:         movl    %eax, 4(%esp)
22:         movl    $.LC1, (%esp)
29:         call    printf
2e:         leave
2f:         ret
30:         .size   subr, .-subr
0:          .comm   X,4,4
4:          .ident  "GCC: (Debian 4.4.5-8) 4.4.5"
0:          .section        .note.GNU-stack,"",@progbits
```

must be replaced with .LC0's address

must be replaced with .LC1's address

must be replaced with printf's address

The ".comm" directive indicates here that four bytes of four-byte aligned storage is 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, its terminology wins (at least here).



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
  - two in common use on x86
    - » 32-bit absolute addresses
      - used for data references
    - » 32-bit PC-relative addresses
      - offset from current value of eip
      - used for procedure references

main.o (1)

```
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:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          208 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         11
  Section header string table index: 8
```


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

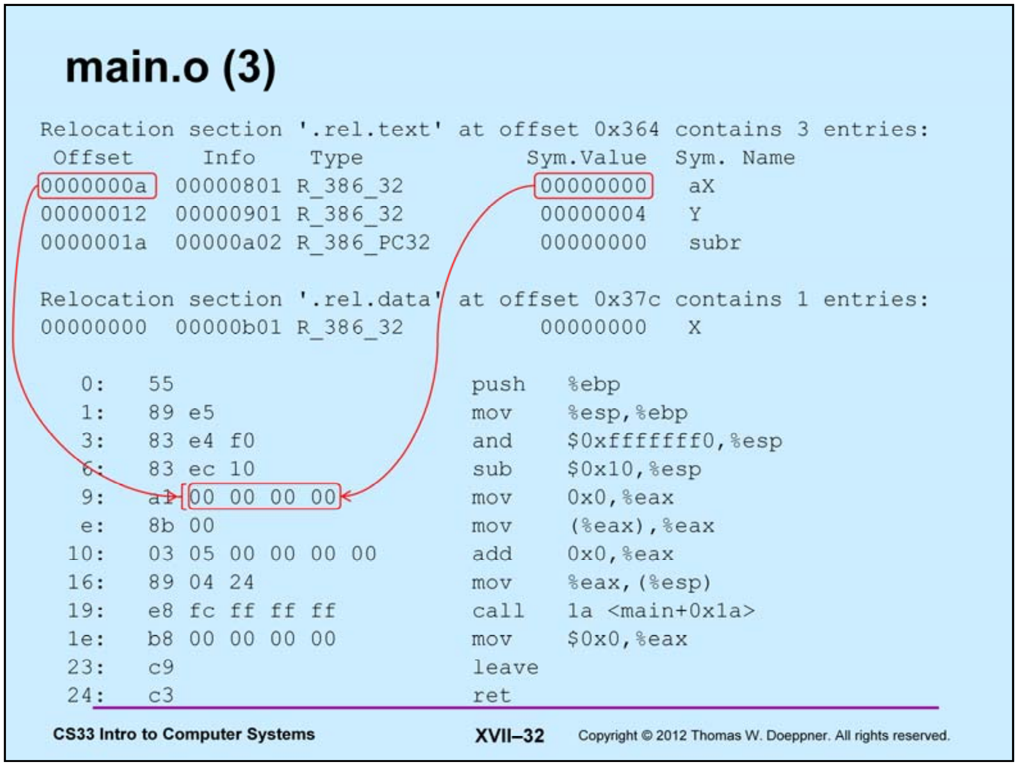
main.o (2)

32-bit, absolute address

```
Relocation section '.rel.text' at offset 0x364 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000000a  00000801 R_386_32          00000000   aX
00000012  00000901 R_386_32          00000004   Y
0000001a  00000a02 R_386_PC32        00000000   subr
```

32-bit, PC-relative address

```
Relocation section '.rel.data' at offset 0x37c contains 1 entries:
00000000  00000b01 R_386_32          00000000   X

   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:   a1 00 00 00 00          mov    0x0,%eax
   e:   8b 00                   mov    (%eax),%eax
  10:   03 05 00 00 00 00       add    0x0,%eax
  16:   89 04 24                mov    %eax,(%esp)
  19:   e8 fc ff ff ff          call   1a <main+0x1a>
  1e:   b8 00 00 00 00          mov    $0x0,%eax
  23:   c9                      leave
  24:   c3                      ret
```

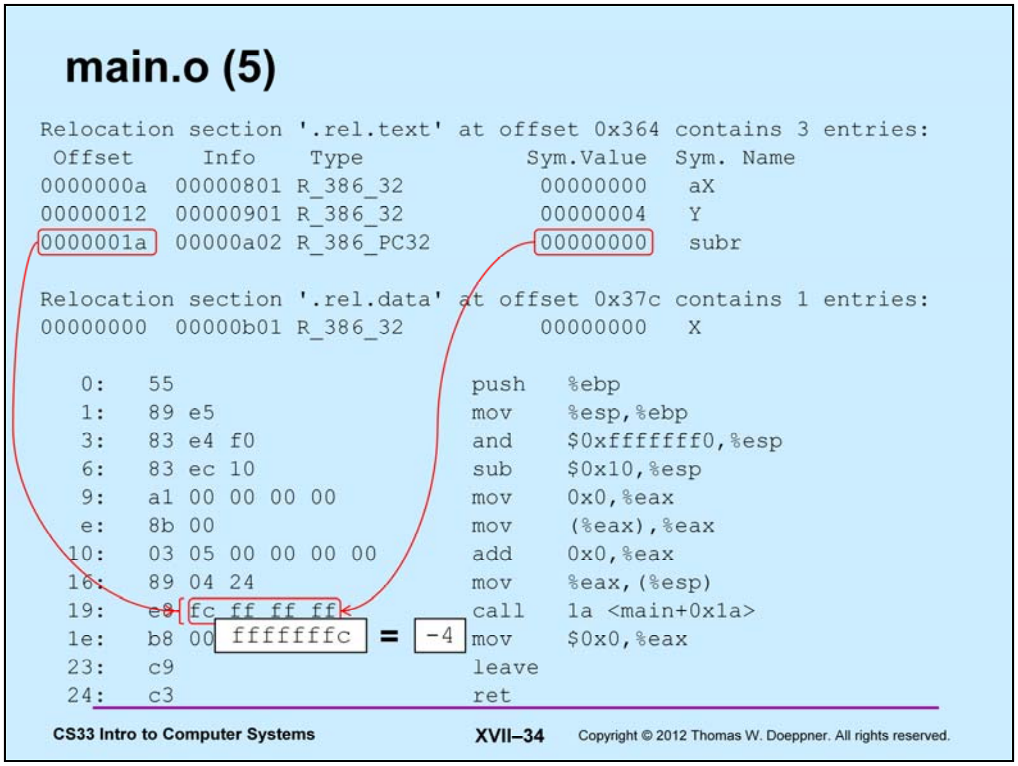



The first relocation instruction specifies that offset 0x0a of the text region should be updated by adding to it the value ultimately associated with symbol aX. This value will be, of course, the address of where aX is located in the data region.

main.o (4)

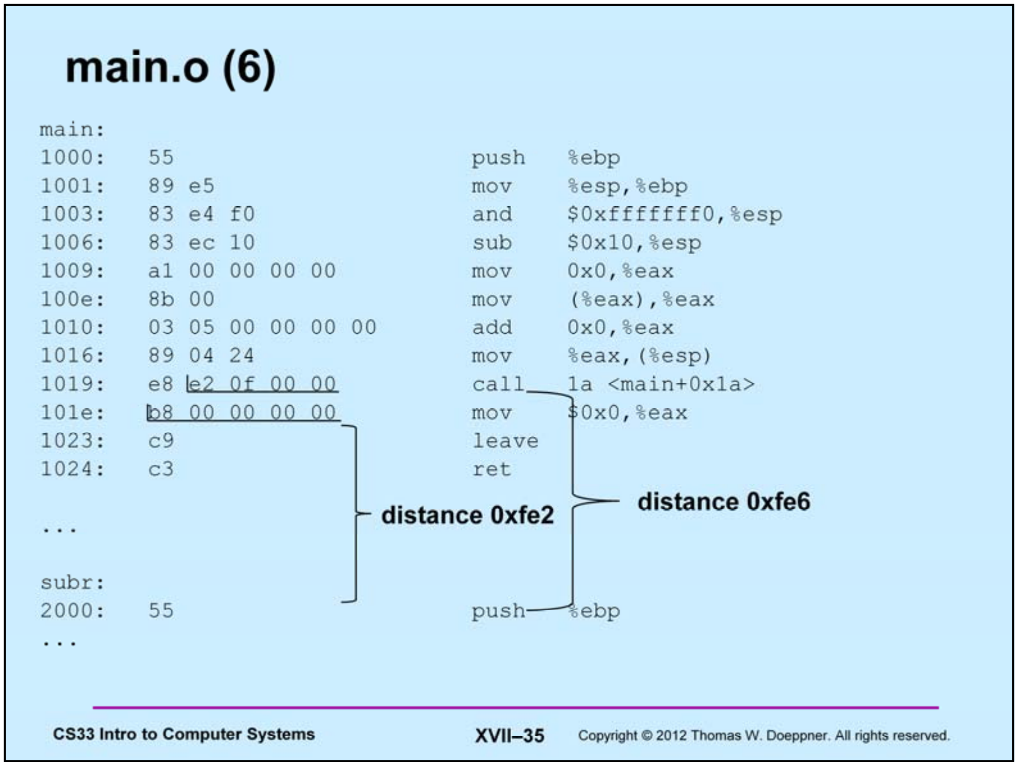
```
Relocation section '.rel.text' at offset 0x364 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000000a  00000801 R_386_32          00000000   aX
00000012  00000901 R_386_32          00000004   Y
0000001a  00000a02 R_386_PC32        00000000   subr

Relocation section '.rel.data' at offset 0x37c contains 1 entries:
00000000  00000b01 R_386_32          00000000   X

   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:   a1 00 00 00 00          mov    0x0,%eax
   e:   8b 00                   mov    (%eax),%eax
  10:   03 05 00 00 00 00       add    0x0,%eax
  16:   89 04 24                mov    %eax,(%esp)
  19:   e8 fc ff ff ff          call   1a <main+0x1a>
  1e:   b8 00 00 00 00          mov    $0x0,%eax
  23:   c9                      leave
  24:   c3                      ret
```


The next relocation instruction specifies that offset 0x12 of the text region should be updated by adding to it the value ultimately associated with symbol Y. This value will be, of course, the address of where Y is located in the data region. Note that the value given in “Sym. Value” (symbol value: 4) is the relative location of Y within the data section contributed by main.



The next relocation is PC-relative and a bit more complicated. It specifies that the PC-relative location of subr should be added to the 32-bit value starting at location 0x1a. The idea is that what should go into the instruction is the offset, relative to the location of the instruction, of the procedure subr. However, on the x86 architecture, when this instruction is being executed, the PC (register eip) has already been set to point to the next instruction. Thus what we really want here is the address of subr relative to the instruction following this one. Rather than having to build into ELF some knowledge of the details of PC-relative addressing for each architecture, a bias of -4 is supplied as the initial address value in the instruction. The value that's shown in the slide, if converted from little-endian form, is fffffffc, which is the two's-complement notation for -4.

Thus what is added to the 32-bit value at location 0x1a is difference between the location where subr is ultimately loaded into memory and the location where this 32-bit portion of the call instruction is loaded into memory. For example, if main is loaded into memory at location 0x1000, then that portion of the call instruction is loaded at location 0x101a. If subr is loaded into memory at location 0x2000, then what is added to the value at 0x101a is 0x2000-0x101a, which is 0xfe6. What then appears is the instruction is the value 0xfe2. See the next slide.



Here we see the result of relocating the reference to subr at location 0x1a within main. We assume that main was loaded into memory at location 0x1000 and subr was loaded into memory at location 0x2000. The 32-bit value in location 0x101a should be the PC-relative address of subr, for when the PC refers to the instruction following the call instruction, i.e., when eip’s value is 0x101e.

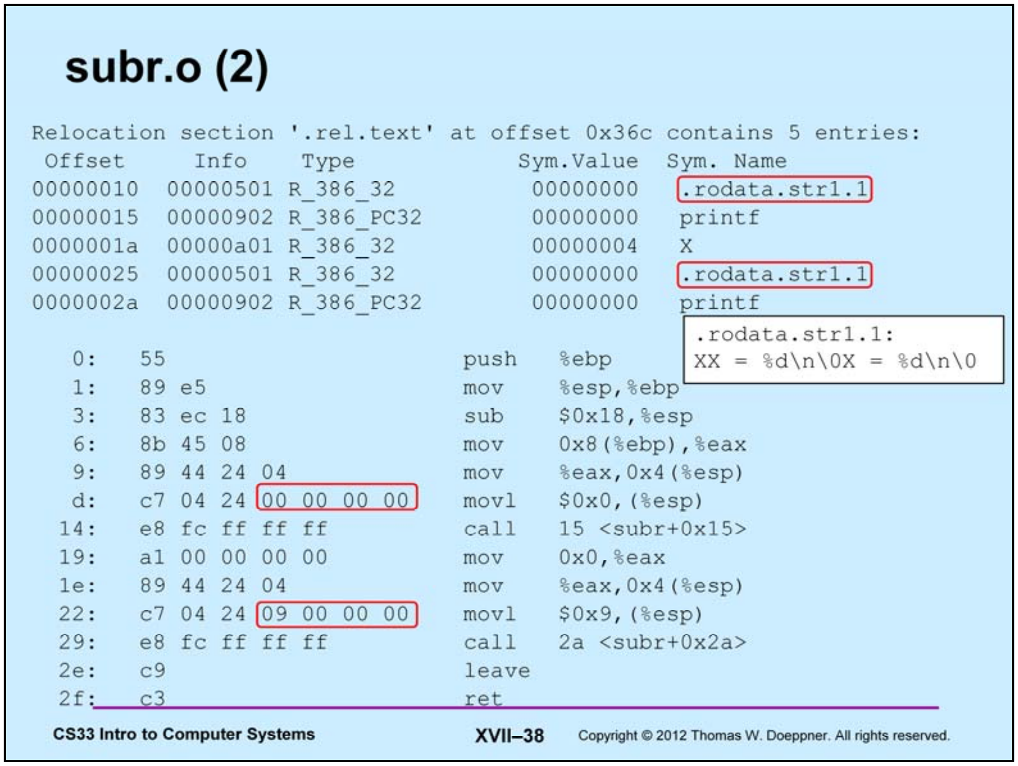
main.o (7)

```
Relocation section '.rel.text' at offset 0x364 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000000a  00000801 R_386_32          00000000   aX
00000012  00000901 R_386_32          00000004   Y
0000001a  00000a02 R_386_PC32        00000000   subr

Relocation section '.rel.data' at offset 0x37c contains 1 entries:
00000000  00000b01 R_386_32          00000000   X

   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:   a1 00 00 00 00          mov    0x0,%eax
   e:   8b 00                   mov    (%eax),%eax
  10:   03 05 00 00 00 00       add    0x0,%eax
  16:   89 04 24                mov    %eax,(%esp)
  19:   e8 fc ff ff ff          call   1a <main+0x1a>
  1e:   b8 00 00 00 00          mov    $0x0,%eax
  23:   c9                      leave
  24:   c3                      ret
```


The final relocation entry applies to the data region: the 32-bit value starting at offset 0 of this object file's contribution to the data region should have final value of symbol X added to it.

subr.o (1)

```
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:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          236 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         11
  Section header string table index: 8
```



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". However the initial value within the instruction is 0 for the first reference and 9 for the second (keep in mind that the values are in little-endian form). Thus after adding in the ultimate value of .rodata.str1.1, the first reference is to the beginning of .rodata.str1.1, and the second reference is to .rodata.str1.1+9. The slide shows the two strings within the .rodata.str1.1 section: the first starts at offset 0, the second starts immediately after the first at offset 9 (keep in mind that each string is terminated with a null ('\0') character).

printf.o

```
Relocation section '.rel.text' at offset 0x36c contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
000002d3  00000902 R_386_PC32        00000000   write
000000d3  00000a01 R_386_32          00000004   StandardFiles
```


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, a lower-level I/O routine that we discuss in a later lecture.

Final Result

Symbol	Value	Size	
_start	0x8048000	0x60	
main	0x8048060	0x25	
subr	0x8048085	0x30	text
printf	0x80480b5	0x12000	
write	0x805a0b5	0x30	
.rodata	0x805a0e5	0x9	
aX	0x805b000	0x4	
Y	0x805b004	0x4	data
StandardFiles	0x805b008	0x1000	
X	0x805c008	0x4	bss

The slide shows the final layout of the address space. 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" routine, which we discuss in a later lecture).

If you are exceptionally sharp-eyed, you might notice that .rodata refers to an area 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 0x8048000 (=134,512,640 in decimal) rather than at a much smaller value (such as 0). For a 32-bit machine, 0x8048000 is around 3% of the way from 0 to the end of the 2^{32}-byte address space, which seems rather high up, particularly since nothing is ordinarily allocated at lower addresses. Apparently this value first appeared in the specification of an old version of Unix (in particular, in the "System V Application Binary Interface, Intel386 Architecture Processor Supplement — http://www.uclibc.org/docs/psABI-i386.pdf), which suggested (but did not require) that the stack be at the bottom of the address space, ranging from just before 0x8048000 down to 0, and that text start at 0x8048000. The Linux/gcc folks apparently took the suggestion concerning text, but stayed with the more typical Unix convention that the stack start at the top of the user-addressable portion of the address space. It's clearly not all that important where text begins. However, it is a very good idea for address 0 to be illegal, generating an exception if used — thus guaranteeing an exception whenever a null pointer is dereferenced.

Libraries

- Collections of useful stuff
- Incorporate items into your program
- Replace existing items with new stuff
- Often ugly …

Creating a Library

```
% gcc -c sub1.c sub2.c sub3.c
% ls
sub1.c        sub2.c        sub3.c
sub1.o        sub2.o        sub3.o
% ar cr libpriv1.a sub1.o sub2.o sub3.o
% ar t libpriv1.a
sub1.o
sub2.o
sub3.o
%
```


Using a Library

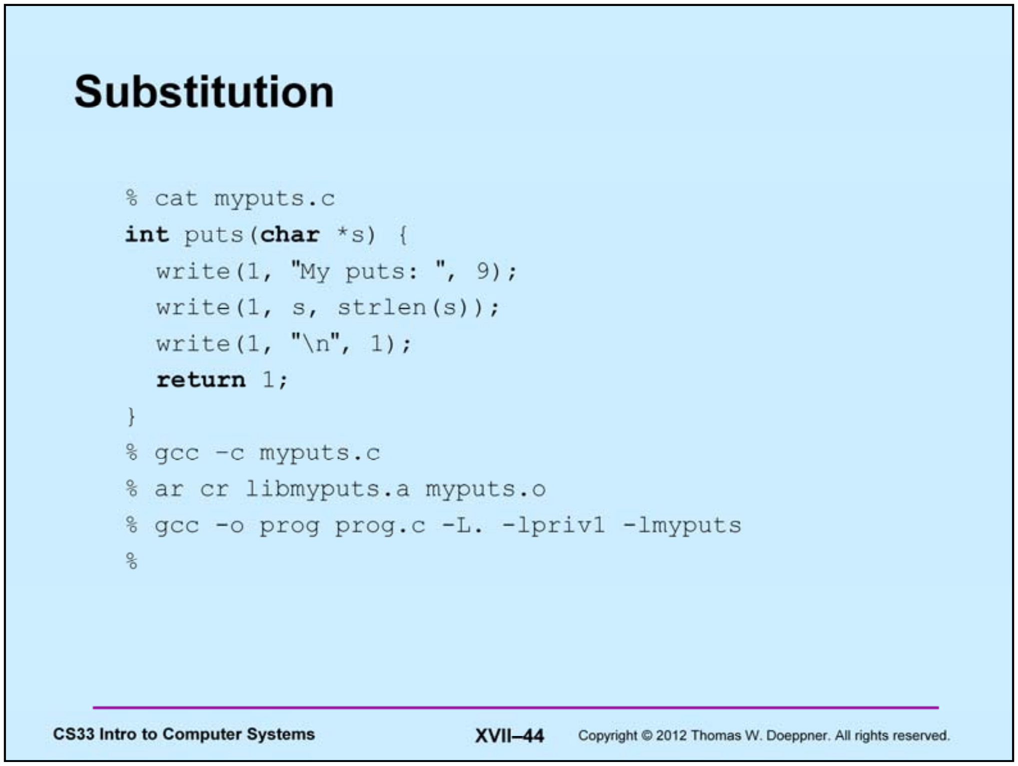
```
% cat prog.c
int main() {
  sub1();
  sub2();
  sub3();
}
% cat sub1.c
void sub1() {
  puts("sub1");
}
```

```
% gcc -o prog prog.c -L. -lpriv1
```

Where does *puts* come from?

```
% gcc -o prog prog.c -L. \
  -lpriv1 -L/lib -lc
```


The routine “puts” is from the standard-I/O library, just as printf is, but it’s far simpler. It prints its single string argument, appending a ‘\n’ (newline) to the end.



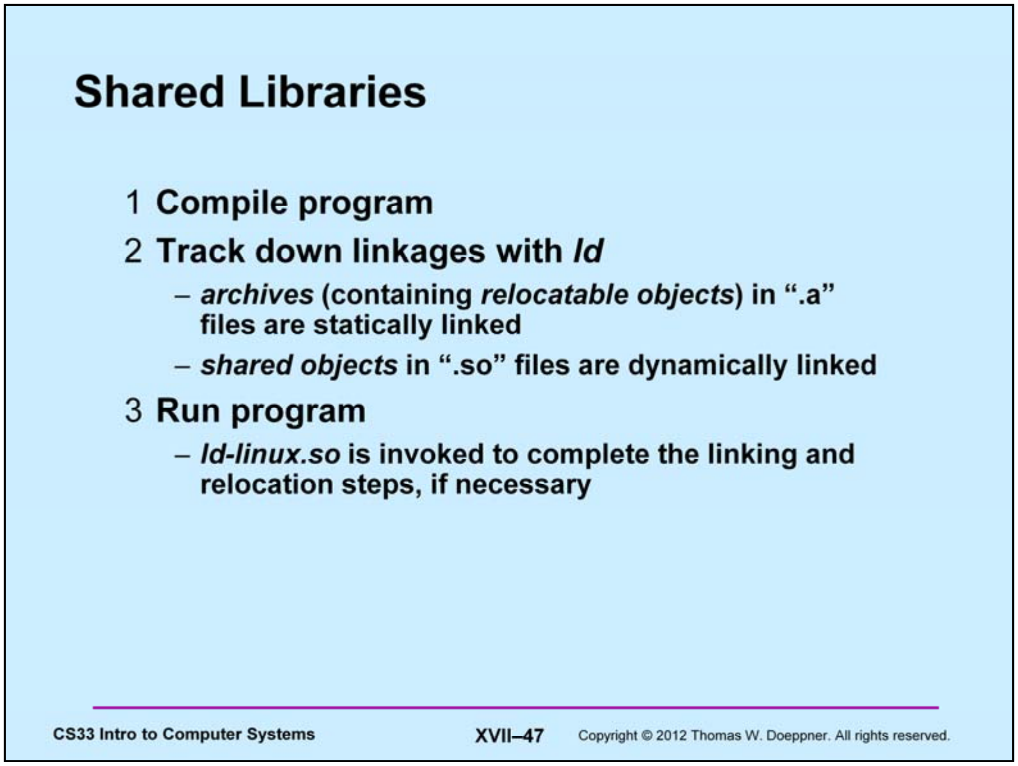
The routine "write" is a low-level I/O routine, used by puts, printf, and anything else that ultimately does output. Its first argument indicates where the data is going: "1" refers to "standard output", which is normally the display. The second argument is the data to be output, and the third argument is its length. We discuss all this in much more detail in a future lecture.

A Problem

- printf is found to have a bug
  - perhaps a security problem
- All existing instances must be replaced
  - there are zillions of instances ...
- Do we have to re-link all programs that use printf?

Static vs. Dynamic Linking

- Static linking
  - needed items are included in executable
  - all necessary relocation is performed before run time
- Dynamic linking
  - not all needed items included in executable
  - some are found and linked at run time



Linux supports two kinds of libraries—static libraries, contained in *archives*, whose names end with ".a" (e.g. *libc.a*) and *shared* objects, whose names end with ".so" (e.g. *libc.so*). When *ld* is invoked to handle the linking of object code, it is normally given a list of libraries in which to find unresolved references. If it resolves a reference within a *.a* file, it copies the code from the file and statically links it into the object code. However, if it resolves the reference within a *.so* file, it records the name of the shared object (not the complete path, just the final component) and postpones actual linking until the program is executed.

If the program is fully bound and relocated, then it is ready for direct execution. However, if it is not fully bound and relocated, then *ld* arranges things so that when the program is executed, rather than starting with the program's main routine, a runtime version of *ld*, called *ld-linux.so*, is called first. *ld-linux.so* maps all the required libraries into the address space, completes the linkages, and then calls the main routine.

Creating a Shared Library (1)

```
% gcc -fPIC -c myputs.c
% ld -shared -o libmyputs.so myputs.o
% gcc -o prog prog.c -L. -lpriv1 -lmyputs
% ./prog
./prog: error while loading shared libraries:
libmyputs.so: cannot open shared object file: No
such file or directory
% ldd prog
        libmyputs.so => not found
        libc.so.6 => /lib/tls/i686/cmoc/libc.so.6
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2
```


The –fPIC flag tells gcc to produce "position-independent code," which is something we discuss in a later lecture. The ld command invokes the loader directly. The –shared flag tells it to created a shared object. In this case, it's creating it from the object file myputs.o and calling the shared object libmyputs.o.

The error occurs because we haven't indicated in the executable (prog) where ld-linux.so should look for shared objects. The ldd (list dynamic dependencies) command, which looks at all the shared objects referenced in the executable and prints out where they are found, shows us what the problem is.

Creating a Shared Library (2)

```
% gcc -o prog prog.c -L. -lpriv1 -lmyputs \
-Wl,-rpath .
% ldd prog
       libmyputs.so => ./libmyputs.so
       libc.so.6 => /lib/tls/i686/cmoc/libc.so.6
       /lib/ld-linux.so.2 => /lib/ld-linux.so.2
% ./prog
My puts: sub1
My puts: sub2
My puts: sub3
```


The “-Wl,-rpath .” flag tells the loader to indicate in the executable (prog) that ld-linux.so should look in the current directory (referred to as “.”) for shared objects. (The “-Wl” part of the flag tells gcc to pass the rest of the flag to the loader.)

Versioning

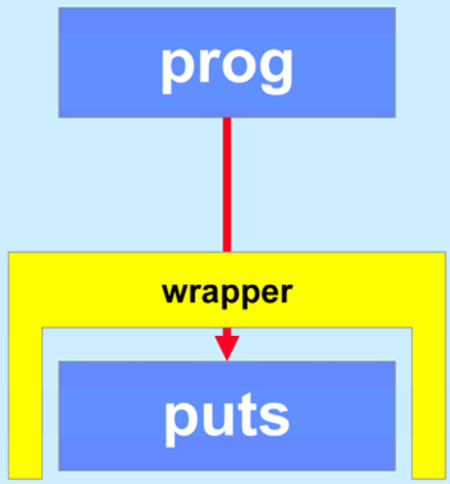
```
% gcc -fPIC -c myputs.c
% ld -shared -soname libmyputs.so.1 \
-o libmyputs.so.1 myputs.o
% ln -s libmyputs.so.1 libmyputs.so
% gcc -o prog1 prog1.c -L. -lpriv1 -lmyputs \
-Wl,-rpath .
% vi myputs.c
% ld -shared -soname libmyputs.so.2 \
-o libmyputs.so.2 myputs.o
% rm -f libmyputs.so
% ln -s libmyputs.so.2 libmyputs.so
% gcc -o prog2 prog2.c -L. -lpriv1 -lmyputs \
-Wl,-rpath .
```


Here we are creating two versions of libmyputs, in libmyputs.so.1 and in libmyputs.so.2. Each is created by invoking the loader directly via the "ld" command. The "-soname" flag tells the loader to include in the shared object its name, which is the string following the flag ("libmyputs.so.1" in the first call to ld). The effect of the "ln –s" command is to create a new name (its last argument) in the file system that refers to the same file as that referred to by ln's next-to-last argument. Thus, after the first call to ln –s, libmyputs.so refers to the same file as does libmyputs.so.1. Thus the second invocation of gcc, where it refers to –lmyputs (which expands to libmyputs.so), is actually referring to libmyputs.so.1.

Then we create a new version of myputs and from it a new shared object called libmyputs.so.2 (i.e., version 2). The call to "rm" removes the name libmyputs.so (but not the file it refers to, which is still referred to by libmyputs.so.1). Then ln is called again to make libmyputs.so now refer to the same file as does libmyputs.so.2. Thus when prog2 is linked, the reference to –lmyputs expands to libmyputs.so, which now refers to the same file as does libmyputs.so.2.

If prog1 is now run, it refers to libmyputs.so.1, so it gets the old version (version 1), but if prog2 is run, it refers to libmyputs.so.2, so it gets the new version (version 2). Thus programs using both versions of myputs can coexist.

Interpositioning



How To …

```
int __wrap_puts(const char *s) {
  int __real_puts(const char *);

  write(2, "calling myputs: ", 16);
  return __real_puts(s);
}
```

Compiling/Linking It

```
% cat tputs.c
int main() {
  puts("This is a boring message.");
  return 0;
}
% gcc -o tputs -Wl,--wrap=puts tputs.c myputs.c
% ./tputs
calling myputs: This is a boring message.
%
```

How To (Alternative Approach) …

```
#include <dlfcn.h>

int puts(const char *s) {
  int (*pptr)(const char *);

  pptr = (int(*)())dlsym(RTLD_NEXT, "puts");

  write(2, "calling myputs: ", 16);
  return (*pptr)(s);
}
```

What's Going On …

- gcc/ld
  - compiles code
  - does static linking
    - » searches list of libraries
    - » adds references to shared objects
- runtime
  - program invokes *ld-linux.so* to finish linking
    - » maps in shared objects
    - » does relocation and procedure linking as required
  - *dlsym* invokes *ld-linux.so* to do more linking

Delayed Wrapping

- LD_PRELOAD
  - environment variable checked by *ld-linux.so*
  - specifies additional shared objects to search (first) when program is started

Example

```
% gcc -o tputs tputs.c
% ./tputs
This is a boring message.
% setenv LD_PRELOAD ./libmyputs.so.1
% ./tputs
calling myputs: This is a boring message.
%
```

There's More ...

- **But first we have to discuss virtual memory**
  - coming up in a couple weeks