CS 4400
Computer Systems

LECTURE 14

Linking

Object files and format

Symbols and symbol tables

`static` Attribute

- The `static` attribute has two jobs:
  - case 1: `static` inside a function
  - case 2: `static` outside a function
- These two uses are related; the first implies the second

static Attribute, case 1

- Suppose we want to count the number of times a particular function is invoked. Why won't the following work?

```
void my_function() {
     int count = 0;
     printf("invoked %d times\n", ++count);
}
```

static Attribute, case 1

- A `static` var's storage is allocated for the entire program.
  - a `static` variable is initialized only once (defaults to zero)

```
void my_function() {
     static int count = 0;
     printf(“invoked %d times\n”, ++count);
}
```

`static` Attribute, case 2

- `static` functions and variables
  - may be referenced only by code within the same file
- Like `private` in C++/Java, the `static` attribute hides variable and function definitions inside modules

`static` Attribute, case 2

- In C, any global variable or function declared *without* the `static` attribute is public and can be accessed by other modules
- In C, any global variable or function declared *with* the `static` attribute is private to that module

Linking

- ***Linking***
  - Collecting and combining code and data into a single file that can be executed
- Linkers enable ***separate compilation***.
  - Changing one module
  - Recompile the module
  - Relink the application (not recompile the other modules).

Linking

- Linking can be performed at:
  - ***compile time***—when the code is translated into machine code
  - ***load time***—when the program is copied into memory
  - ***run time***—during execution

Why Care About Linking

- Linkers help you build large programs
  - compilers don't scale to huge codes, linkers do
- Helps you avoid dangerous programming errors
  - Isolate code and data
- Helps you understand language scoping rules
  - difference in global and local vars, how to handle `static`
- Enables you to exploit shared libraries
  - shared libraries and dynamic linking are increasingly important

Example Program

```c
/* main.c */

void swap();

int buf[2] = {1, 2};

int main() {
  swap();

  return 0;
}
```

```c
/* swap.c */

extern int buf[];

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

void swap() {
  int temp;

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

- The program consists of two source files.
- Global variable `buf` is defined in `main.c` but visible in `swap.c`.
- Function `swap` swaps the two elements in array `buf`.

Declaration vs. Definition

- A **declaration** tells the compiler about a variable or function and its type, but does not create it
- A **definition** creates the variable or function
  - A definition is also a declaration

Declaration vs. Definition

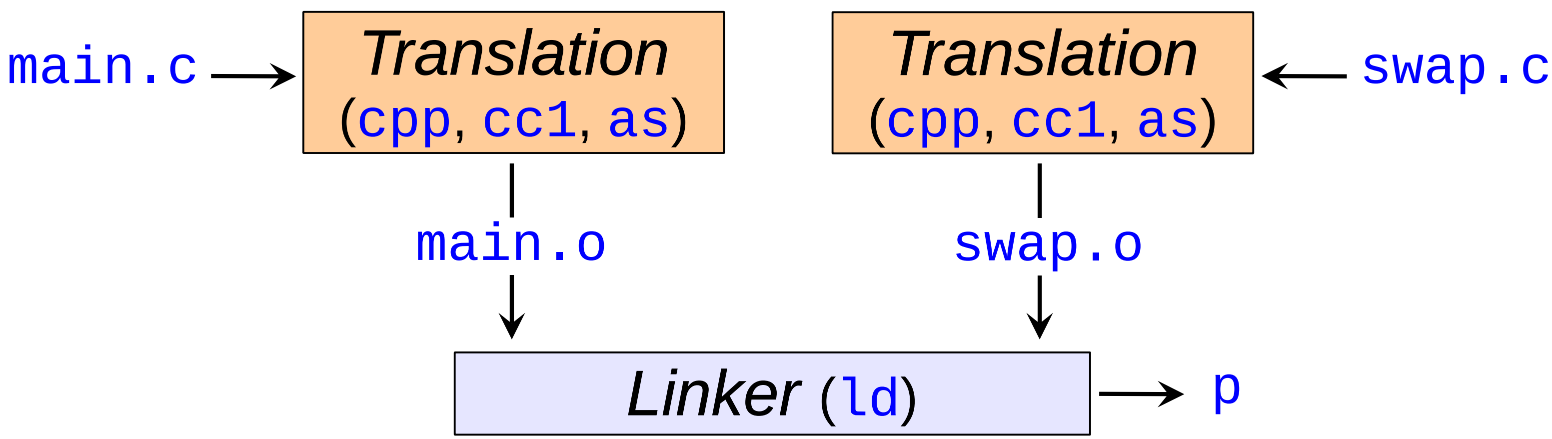
- In C (but not C++) you can use a variable or function without declaring it first
  - Variables are assigned default type int
  - Functions, by default, take no parameters and return int
  - ASSUME THESE ARE NEVER WHAT YOU WANT

Declaration vs. Definition

- `int i;` may be either a declaration or definition depending on context
  - Always a definition if an initializer exists or inside a function
- `extern int i;` is always just a declaration
- Declaration vs. definition is important both at compile time and also at link time

Compiler Driver

- A compiler driver invokes the language preprocessor, compiler, assembler, and linker.
- Invoke the driver:` > gcc -O2 -o p main.c swap.c`



- Run the executable p: `> ./p`
  – The shell invokes the loader, which copies code/data of p into memory and transfers control to beginning of program.

Object Files

- `main.o` and `swap.o` are ***relocatable object files***
- An object file is merely a collection of blocks of bytes.
  - some blocks contain program code
  - other blocks contain program data
  - yet other blocks contain info to guide the linker and loader

Object Files

- Three types of object files:
  - ***relocatable***—can be combined with other relocatable object files at compile time to create an executable object file
  - ***executable***—can be copied directly into memory and executed
  - ***shared***—special type of relocatable object file that can be loaded into memory and linked dynamically (load or run time)

Static Linking

- The Unix `ld` program is a ***static linker***.
  - input: a collection of relocatable object files (`main.o`, `swap.o`)
  - output: a fully-linked executable file (p)

Static Linking

- Object files define and reference symbols
  - *symbol resolution*—associates each symbol reference with exactly one symbol definition
- Compilers and assemblers generate code and data sections that start at address 0x0
  - *relocation*—associates a memory location with each symbol definition and modifies its references to point to this location

ELF Object File Format

`ELF header`	word size, byte order, object file type, offset/size of section header table
`.text`	machine code of the compiled program
`.rodata`	read-only data (e.g., printf format strings and jump tables)
`.data`	initialized global vars (recall that local vars are stored on the stack)
`.bss`	uninitialized global vars (no actual space, just a placeholder)
`.symtab`	symbol table (info about functions and global vars, unlike compiler's)
`.rel.text`	locs in .text that reference external fns or vars (linker must modify)
`.rel.data`	locs in .data whose initial value is address of external fns or vars
`.debug`	symbol table like compiler's (w/ locals), must compile with -g
`.line`	mapping of source code line #s to machine code, must compile with -g
`.strtab`	sequence of null-terminated char strings, for .symtab and .debug
`section header table`	locations and sizes of the various sections (fixed entry for each)

Symbols

- ***Global symbols***—defined by module *m* and can be referenced by other modules
  - e.g., C functions and globals defined without the static attribute
- ***Externals***—symbols referenced by *m*, but defined by some other module
  - e.g., C functions and variables defined in other modules
- ***Local symbols***—defined and referenced exclusively by module *m*
  - e.g., C functions and globals defined with the static attribute
  - does not include non-static locals (maintained on the stack)

Symbol Tables

- The `.symtab` section contains an array of entries, each with the following information about an object:
  - ***name***—offset into the string table, pointing to symbol's name
  - ***value***—offset from beginning of section where object is defined (relocatable) or an absolute run-time address (executable)
  - ***size***—number of bytes for the object
  - ***type***—data (`OBJECT`) or function (`FUNC`)
  - ***binding***—global or local
  - ***section***—index into section header table or special pseudosections (`ABS`—symbols that should not be relocated, `UND`—symbols that are referenced but not defined, `COM`—uninitialized, unallocated symbols)

Example: Symbol Tables

Num	Value	Size	Type	Bind	Ot	Ndx	Name
8:	0	8	OBJECT	GLOBAL	0	3	buf
9:	0	17	FUNC	GLOBAL	0	1	main
10:	0	0	NOTYPE	GLOBAL	0	UND	swap

last three entries of symbol table for `main.o` (displayed by the `readelf` tool)

Num:	Value	Size	Type	Bind	Ot	Ndx	Name
8:	0	4	OBJECT	GLOBAL	0	3	bufp0
9:	0	0	NOTYPE	GLOBAL	0	UND	buf
10:	0	39	FUNC	GLOBAL	0	1	swap
11:	4	4	OBJECT	GLOBAL	0	COM	bufp1

symbol table entries for `swap.o`

- `Ndx=1` denotes the `.text` section, `Ndx=3` the `.data` section.
- For `COM` symbols, `Value` gives the alignment and `Size` the max size.
- The first eight entries are local symbols that the linker uses internally.

Question

- Does it have a symbol table entry?
- If so, what is its type?
  - Local, global, or extern?
- Which module defines it?
- Which section does it occupy?

```
extern int buf[];

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

void swap() {
  int temp;

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

- Symbols referenced in `swap.o`:
  - `bufp0`, `bufp1`, `buf`, `swap`, `temp`

Symbol Resolution

- The linker associates each symbol reference with exactly one definition from the symbol tables of its input relocatable object files.
  - trivial for a local symbol (one per module, unique name)
  - tricky for a global symbol

Symbol Resolution

- The compiler assumes `foo` is defined in some other module and generates a symbol table entry (leaving it for the linker to handle).

```
/* linkerror.c */
void foo();
int main() {
  foo();
  return 0;
}
```

```
> gcc -Wall -O2 linkerror.c
/tmp/ccYEn1m9.o(.text+0x7): In function `main':
: undefined reference to `foo'
collect2: ld returned 1 exit status
```

Multiply-Defined Symbols

- What if the same global symbol is defined by multiple object files?
  – Linker must report an error or choose one of the definitions.
  – Is this a problem for C++/Java overloaded methods?

Multiply-Defined Symbols

- The compiler exports symbols as ***strong*** (functions and initialized globals) or ***weak*** (uninitialized globals).
- Unix linkers use the following rules:
  1. Multiple strong symbols are not allowed.
  2. If multiple weak symbols and a strong symbol, choose strong.
  3. If multiple weak symbols, choose any one.

Example: Rule 1

```
/* foo1.c */
int main() {
  return 0;
}
```

```
/* bar1.c */
int main() {
  return 0;
}
```

```
> gcc foo1.c bar1.c
/tmp/ccvzRoJL.o(.text+0x0): In function `main':
: multiple definition of `main'
/tmp/ccepVLhT.o(.text+0x0): first defined here
```

Example: Rule 1

```c
/* foo2.c */
int x = 15213;
int main() {
  return 0;
}
```

```c
/* bar2.c */
int x = 15213;
void f() {}
```

```
> gcc foo2.c bar2.c
/tmp/ccXhFAzx.o(.data+0x0): multiple definition of `x'
/tmp/cccOqVLn.o(.data+0x0): first defined here
```

Example: Rule 2

```
/* foo3.c */
#include <stdio.h>

void f();

int x = 15213; /* strong */

int main() {
  f();
  printf(“x = %d\n”, x);
  return 0;
}
```

```
/* bar3.c */
int x; /* weak */

void f() {
  x = 15212;
}
```

```
> gcc foo3.c bar3.c
> ./a.out
x = 15212
```

- At run time, `f` changes the value of `x` from `15213` to `15212`.
- The linker gives no indication that it found multiple defs of `x` (unless Rule 1).

Example: Rule 3

```c
/* foo4.c */
#include <stdio.h>

void f();

int x; /* weak */

int main() {
  x = 15213;
  f();
  printf(“x = %d\n”, x);
  return 0;
}
```

```c
/* bar4.c */
int x; /* weak */

void f() {
  x = 15212;
}
```

```
> gcc foo4.c bar4.c
> ./a.out
x = 15212
```

Example: Rule 2

```
/* foo5.c */
#include <stdio.h>

void f();

int x = 15213; /* strong */
int y = 15212;

int main() {
  f();
  printf("x = 0x%x y = 0x%x \n", x, y);
  return 0;
}
```

```
/* bar5.c */
double x; /* weak */

void f() {
  x = -0.0;
}
```

Alignment of (8-byte) `x` in `bar5.c` overwrites memory locations for (4-byte) `x` and (4-byte) `y` in `foo5.c` with the double-precision floating-point representation of negative 0.

```
> gcc foo5.c bar5.c
/usr/bin/ld: Warning: alignment 4 of symbol `x' in
/tmp/ccY13dOq.o is smaller than 8 in /tmp/cc8VBPpA.o
> ./a.out
x = 0x0 y = 0x80000000
```

Question

$REF(\mathtt{x}.\mathtt{i}) \rightarrow DEF(\mathtt{x}.k)$ denotes that linking will associate any reference to *x* in module *i* to the definition of *x* in *k*.

```
/* Module 1 */
int main() {}
```

```
/* Module 2 */
int main;
int p2() {}
```

$REF(\mathtt{main.1}) \rightarrow DEF(\mathtt{??})$
$REF(\mathtt{main.2}) \rightarrow DEF(\mathtt{??})$

`main.1`, `main.2`, ERROR, or UNKNOWN?

```
/* Module 1 */
void main() {}
```

```
/* Module 2 */
int main = 1;
int p2() {}
```

$REF(\mathtt{main.1}) \rightarrow DEF(\mathtt{??})$
$REF(\mathtt{main.2}) \rightarrow DEF(\mathtt{??})$

`main.1`, `main.2`, ERROR, or UNKNOWN?

```
/* Module 1 */
int x;
void main() {}
```

```
/* Module 2 */
double x = 1.0;
int p2() {}
```

$REF(\mathtt{x.1}) \rightarrow DEF(\mathtt{??})$
$REF(\mathtt{x.2}) \rightarrow DEF(\mathtt{??})$

`x.1`, `x.2`, ERROR, or UNKNOWN?