Final C Details, Build Tools (make)

CSE 333 Summer 2018

Instructor: Hal Perkins

Teaching Assistants:

Renshu Gu William Kim Soumya Vasisht

Administrivia

Timetable (for planning ahead)

- Exercise 5 posted yesterday, due Monday
- Monday:
  - Lecture: File I/O, intro to system calls, overview of POSIX (system) library (lecture may jump around a little to get to things on time)
  - Exercise 6 out, due Thursday morning (instead of Wed. because…)
- No class Wednesday; 4th of July holiday
- Thursday:
  - Exercise 6 due morning
  - HW1 due evening
  - Sections: reading file system directories using POSIX I/O
    - Exercise 7 based on that out, due following Monday

Administrivia

- Homework 1 due on Thursday (7/5)
  - Watch that `hashtable.c` doesn't violate the modularity of `ll.h`
  - Watch for pointer to local (stack) variables
  - Use a debugger (*e.g.* `gdb`) if you're getting segfaults
  - If things don't work, try writing smaller tests to isolate bugs
  - Advice: clean up "to do" comments, but leave "step #" markers for graders
  - Late days: don't tag `hw1-final` until you are really ready
  - Extra Credit: if you add unit tests, put them in a new file and adjust the Makefile

A useful gdb trick

- gdb has a simple full-screen mode
  - `gdb -tui` <other command-line parameters>
  - <demo>

- Works great! When it works.
  - OK on attu, workstations
  - Broken on VM where an older gdb version is installed. To get the latest version that supports `-tui` run this command:

```
sudo yum -y install devtoolset-4-jsoup \
  devtoolset-4-gdb devtoolset-4-guava \
  devtoolset-4-lpg-java-compat devtoolset-4-sat4j
```

(all on one line with no `\`s), then restart your vm

Lecture Outline

- **Header Guards and Preprocessor Tricks**
- Visibility of Symbols
  - `extern`, `static`
- Make and Build Tools

An #include Problem

- What happens when we compile `foo.c`?

```
struct pair {
  int a, b;
};
```

pair.h

```
#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);
```

util.h

```
#include "pair.h"
#include "util.h"

int main(int argc, char** argv) {
  // do stuff here
  ...
  return 0;
}
```

foo.c

An `#include` Problem

- What happens when we compile `foo.c`?

```
bash$ gcc -Wall -g -o foo foo.c
In file included from util.h:1:0,
                 from foo.c:2:
pair.h:1:8: error: redefinition of 'struct pair'
 struct pair { int a, b; };
        ^
In file included from foo.c:1:0:
pair.h:1:8: note: originally defined here
 struct pair { int a, b; };
        ^
```

- `foo.c` includes `pair.h` twice!
  - Second time is indirectly via `util.h`
  - Struct definition shows up twice
    - Can see using `cpp`

Header Guards

- A commonly-used C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

```
#ifndef _PAIR_H_
#define _PAIR_H_

struct pair {
  int a, b;
};

#endif   // _PAIR_H_
```

pair.h

```
#ifndef _UTIL_H_
#define _UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // _UTIL_H_
```

util.h

Other Preprocessor Tricks

- A way to deal with "magic constants"

```c
int globalbuffer[1000];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * 3.1415;
  *area = rad * 3.1415 * 3.1415;
}
```

Bad code
(littered with magic constants)

```c
#define BUFSIZE 1000
#define PI 3.14159265359

int globalbuffer[BUFSIZE];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * PI;
  *area = rad * PI * PI;
}
```

Better code

Macros

- You can pass arguments to macros

```
#define ODD(x) ((x) % 2 != 0)

void foo() {
  if ( ODD(5) )
    printf("5 is odd!\n");
}
```

cpp →

```
void foo() {
  if ( ((5) % 2 != 0) )
    printf("5 is odd!\n");
}
```

- Beware of operator precedence issues!
  - Use parentheses

```
#define ODD(x) ((x) % 2 != 0)
#define WEIRD(x) x % 2 != 0

ODD(5 + 1);

WEIRD(5 + 1);
```

cpp →

```
((5 + 1) % 2 != 0);

5 + 1 % 2 != 0;
```

Conditional Compilation

- You can change what gets compiled:

```
#ifdef TRACE
#define ENTER(f) printf("Entering %s\n", f);
#define EXIT(f)  printf("Exiting  %s\n", f);
#else
#define ENTER(f)
#define EXIT(f)
#endif

// print n
void pr(int n) {
  ENTER("pr");
  printf("\n = %d\n", n);
  EXIT("pr");
}
```

ifdef.c

Defining Symbols

- Besides `#define`s in the code, preprocessor values can be given as part of the `gcc` command:

```
bash$ gcc -Wall -g -DTRACE -o ifdef ifdef.c
```

- assert can be controlled the same way – defining `NDEBUG` causes assert to expand to “empty”
  - It’s a macro – see `assert.h`

```
bash$ gcc -Wall -g -DNDEBUG -o faster useassert.c
```

Lecture Outline

- Header Guards and Preprocessor Tricks
- **Visibility of Symbols**
  - **`extern`, `static`**
- Make and Build Tools

Namespace Problem

- If I define a global variable named "counter" in one C file, is it visible in another C file in my program?
  - Yes, if you use external linkage
    - The name "counter" refers to the same variable in both files
    - The variable is *defined* in one file and *declared* in the other(s)
    - When the program is linked, the symbol resolves to one location
  - No, if you use internal linkage
    - The name "counter" refers to different variable in each file
    - The variable must be *defined* in each file
    - When the program is linked, the symbols resolve to two locations

External Linkage

- `extern` makes a *declaration* of something externally-visible

```
#include <stdio.h>

// A global variable, defined and
// initialized here in foo.c.
// It has external linkage by
// default.
int counter = 1;

int main(int argc, char** argv) {
  printf("%d\n", counter);
  bar();
  printf("%d\n", counter);
  return 0;
}
```

foo.c

```
#include <stdio.h>

// "counter" is defined and
// initialized in foo.c.
// Here, we declare it, and
// specify external linkage
// by using the extern specifier.
extern int counter;

void bar() {
  counter++;
  printf("(b): counter = %d\n",
         counter);
}
```

bar.c

Internal Linkage

- `static` (in the global context) restricts a definition to visibility within that file

```
#include <stdio.h>

// A global variable, defined and
// initialized here in foo.c.
// We force internal linkage by
// using the static specifier.
static int counter = 1;

int main(int argc, char** argv) {
  printf("%d\n", counter);
  bar();
  printf("%d\n", counter);
  return 0;
}
```

foo.c

```
#include <stdio.h>

// A global variable, defined and
// initialized here in bar.c.
// We force internal linkage by
// using the static specifier.
static int counter = 100;

void bar() {
  counter++;
  printf("(b): counter = %d\n",
         counter);
}
```

bar.c

Function Visibility

bar.c

```
// By using the static specifier, we are indicating
// that foo() should have internal linkage.  Other
// .c files cannot see or invoke foo().
static int foo(int x) {
  return x*3 + 1;
}

// Bar is "extern" by default.  Thus, other .c files
// could declare our bar() and invoke it.
int bar(int x) {
  return 2*foo(x);
}
```

main.c

```
#include <stdio.h>

extern int bar(int x);

int main(int argc, char** argv) {
  printf("%d\n", bar(5));
  return 0;
}
```

Linkage Issues

- Every global (variables and functions) is `extern` by default
  - Unless you add the `static` specifier, if some other module uses the same name, you'll end up with a collision!
    - Best case: compiler (or linker) error
    - Worst case: stomp all over each other

- It's good practice to:
  - Use `static` to "defend" your globals
    - Hide your private stuff!
  - Place external declarations in a module's header file
    - Header is the public specification

Static Confusion…

- C has a *different* use for the word "`static`": to create a persistent *local* variable
  - The storage for that variable is allocated when the program loads, in either the `.data` or `.bss` segment
  - Retains its value across multiple function invocations
  - Confusing! Don't use!! (But you may see it ☹)

```
void foo() {
  static int count = 1;
  printf("foo has been called %d times\n", count++);
}

void bar() {
  int count = 1;
  printf("bar has been called %d times\n", count++);
}

int main(int argc, char** argv) {
  foo(); foo(); bar(); bar(); return 0;
}
```

static_extent.c

Additional C Topics

- Teach yourself!
  - **man pages** are your friend!
  - String library functions in the C standard library
    - `#include <string.h>`
      - strlen(), strcpy(), strdup(), strcat(), strcmp(), strchr(), strstr(), …
    - `#include <stdlib.h>` or `#include <stdio.h>`
      - atoi(), atof(), sprint(), sscanf()
  - How to declare, define, and use a function that accepts a variable-number of arguments (`varargs`)
  - `union`s and what they are good for
  - `enum`s and what they are good for
  - Pre- and post-increment/decrement
  - Harder: the meaning of the "`volatile`" storage class

Lecture Outline

- Header Guards and Preprocessor Tricks
- Visibility of Symbols
  - `extern`, `static`
- **Make and Build Tools**

make

- `make` is a classic program for controlling what gets (re)compiled and how
  - Many other such programs exist (*e.g.* `ant`, `maven`, IDE "projects")

- `make` has tons of fancy features, but only two basic ideas:
  1) Scripts for executing commands
  2) Dependencies for avoiding unnecessary work

- To avoid "just teaching `make` features" (boring and narrow), let's focus more on the concepts…

Building Software

- Programmers spend a lot of time "building"
  - Creating programs from source code
  - Both programs that they write and other people write

- Programmers like to automate repetitive tasks
  - Repetitive: gcc -Wall -g -std=c11 -o widget foo.c bar.c baz.c
    - Retype this every time: 😭
    - Use up-arrow or history: 😐 (still retype after logout)
    - Have an alias or bash script: 🙂
    - Have a Makefile: 😊 (you're ahead of us)

"Real" Build Process

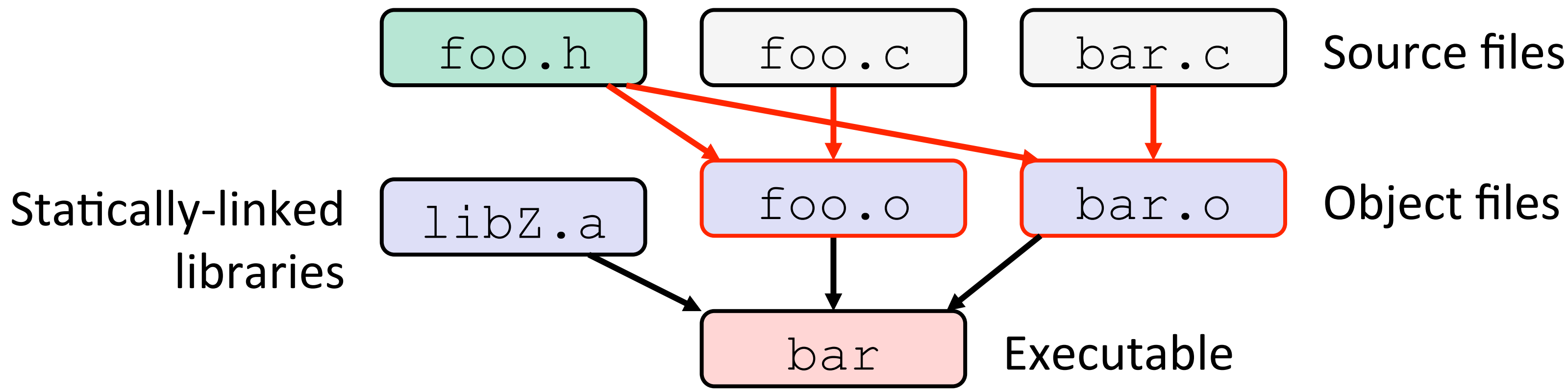
- On larger projects, you can't or don't want to have one big (set of) command(s) that redoes everything every time you change anything:
  1) If `gcc` didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
  2) If source files have multiple output (*e.g.* javadoc), you'd have to type out the source file name multiple times
  3) You don't want to have to document the build logic when you distribute source code
  4) You don't want to recompile everything every time you change something (especially if you have 10^5-10^7 files of source code)
- A script can handle 1-3 (use a variable for filenames for 2), but 4 is trickier

Recompilation Management

- The "theory" behind avoiding unnecessary compilation is a "dependency dag" (**d**irected, **a**cyclic **g**raph)

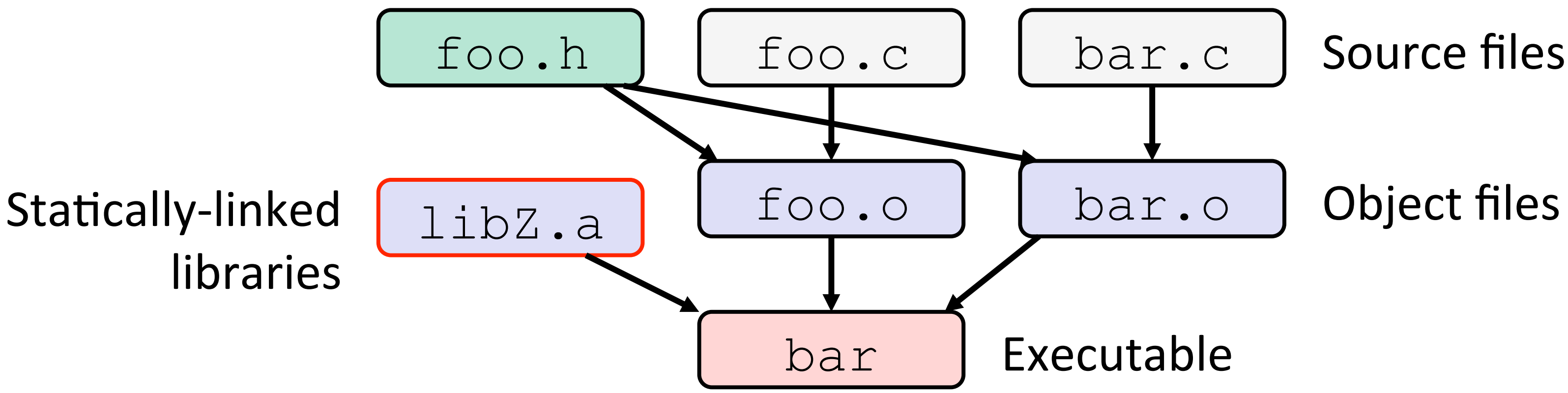
- To create a target t, you need sources s_1, s_2, …, s_n and a command c that directly or indirectly uses the sources
  - It t is newer than every source (file-modification times), assume there is no reason to rebuild it
  - Recursive building: if some source s_i is itself a target for some other sources, see if it needs to be rebuilt…
  - Cycles "make no sense"!

Theory Applied to C



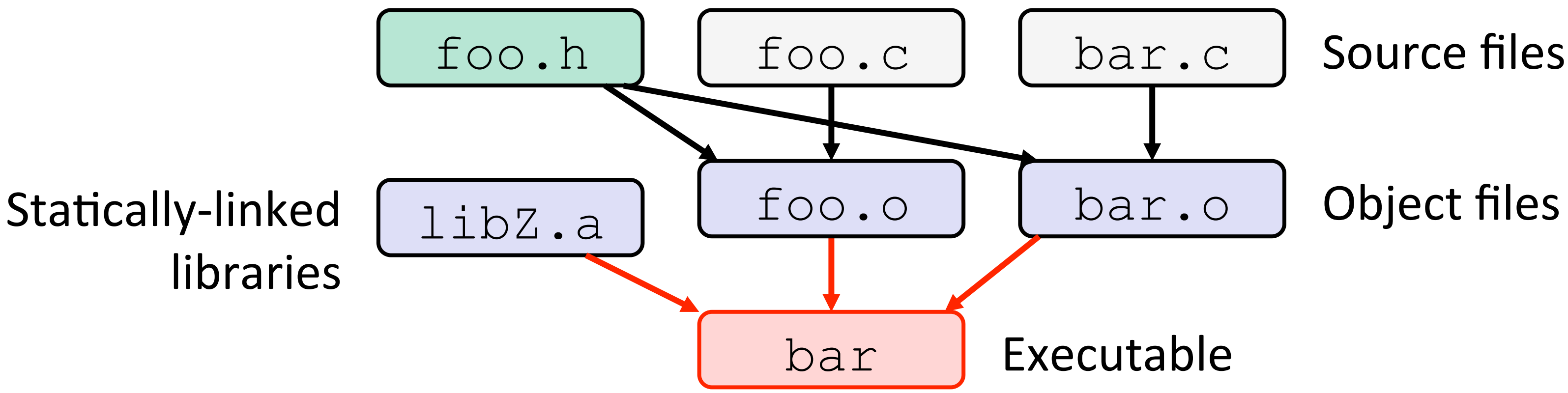
- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)

Theory Applied to C



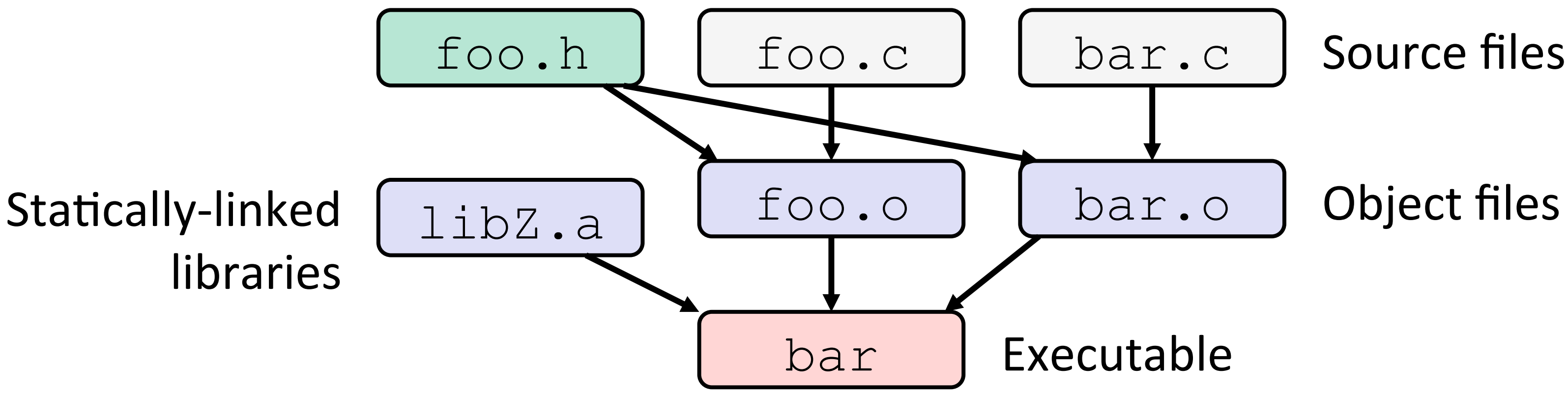
- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- An archive (library, `.a`) depends on included `.o` files

Theory Applied to C



- Compiling a .c creates a .o – the .o depends on the .c and all included files (.h, recursively/transitively)
- An archive (library, .a) depends on included .o files
- Creating an executable ("linking") depends on .o files and archives
  - Archives linked by -L<path> -l<name> (*e.g.* -L. -lfoo to get libfoo.a from current directory)

Theory Applied to C



- If one `.c` file changes, just need to recreate one `.o` file, maybe a library, and re-link
- If a `.h` file changes, may need to rebuild more
- Many more possibilities!

`make` Basics

- A makefile contains a bunch of triples:

```
target: sources
← Tab → command
```

  - Colon after target is *required*
  - Command lines must start with a TAB, NOT SPACES
  - Multiple commands for same target are executed *in order*
    - Can split commands over multiple lines by ending lines with '\'

- Example:

```
foo.o: foo.c foo.h bar.h
	gcc -Wall -o foo.o -c foo.c
```

Using make

```
bash% make -f <makefileName> target
```

- Defaults:
  - If no `-f` specified, use a file named `Makefile`
  - If no `target` specified, will use the first one in the file
  - Will interpret commands in your default shell
    - Set `SHELL` variable in makefile to ensure

- Target execution:
  - Check each source in the source list:
    - If the source is a target in the Makefile, then process it recursively
    - If some source does not exist, then error
    - If any source is newer than the target (or target does not exist), run `command` (presumably to update the target)

make Variables

- You can define variables in a makefile:
  - All values are strings of text, no "types"
  - Variable names are case-sensitive and can't contain ':', '#', '=', or whitespace

- Example:

```
CC = gcc
CFLAGS = -Wall -std=c11
foo.o: foo.c foo.h bar.h
      $(CC) $(CFLAGS) -o foo.o -c foo.c
```

- Advantages:
  - Easy to change things (especially in multiple commands)
  - Can also specify on the command line (`CFLAGS=-g`)

More Variables

- It's common to use variables to hold list of filenames:

```
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
	gcc -o widget $(OBJFILES)
clean:
	rm $(OBJFILES) widget *~
```

- `clean` is a convention
  - Remove generated files to "start over" from just the source
  - It's "funny" because the target doesn't exist and there are no sources, but it works because:
    - The target doesn't exist, so it must be "remade" by running the command
    - These "phony" targets have several uses, such as "`all`"…

"all" Example

```
all: prog B.class someLib.a
      # notice no commands this time

prog: foo.o bar.o main.o
      gcc -o prog foo.o bar.o main.o

B.class: B.java
      javac B.java

someLib.a: foo.o baz.o
      ar r foo.o baz.o

foo.o: foo.c foo.h header1.h header2.h
      gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

Makefile Example

- "talk" program (find files on web with lecture slides)

`main.c` `speak.h` `speak.c` `shout.h` `shout.c`

Revenge of the Funny Characters

- Special variables:
  - **`$@`** for target name
  - **`$^`** for all sources
  - **`$<`** for left-most source
  - Lots more! – see the documentation

- Examples:

```
# CC and CFLAGS defined above
widget: foo.o bar.o
        $(CC) $(CFLAGS) -o $@ $^
foo.o: foo.c foo.h bar.h
        $(CC) $(CFLAGS) -c $<
```

And more…

- There are a lot of "built-in" rules – see documentation
- There are "suffix" rules and "pattern" rules
  - Example:

```
%.class: %.java
        javac $<  # we need the $< here
```

- Remember that you can put *any* shell command – even whole scripts!
- You can repeat target names to add more dependencies
- Often this stuff is more useful for reading makefiles than writing your own (until some day…)

Extra Exercise #1

- Write a program that:
  - Prompts the user to input a string (use `fgets()`)
    - Assume the string is a sequence of whitespace-separated integers (*e.g.* `"5555 1234 4 5543"`)
  - Converts the string into an array of integers
  - Converts an array of integers into an array of strings
    - Where each element of the string array is the binary representation of the associated integer
  - Prints out the array of strings

Extra Exercise #2

- Modify the linked list code from Lecture 5 Extra Exercise #1
  - Add static declarations to any internal functions you implemented in `linkedlist.h`
  - Add a header guard to the header file
  - Write a `Makefile`
    - Use Google to figure out how to add rules to the `Makefile` to produce a library (`liblinkedlist.a`) that contains the linked list code