Section 1: gcc and make

Meghan Cowan
ex00

• Due this Friday Jan 5\textsuperscript{th} by 11:15am
• Comparing performance between:
  • C
  • Optimized C
  • Java
• Questions?
Quick C Refresher

**foo.h**

```c
#ifndef __FOO_H__
#define __FOO_H__
#define PI 3.14159

void foo();
void fee();

#endif // __FOO_H__
```

**foo.c**

```c
#include "foo.h"
#include <stdio.h>

void foo() {
    printf("Foo!\n");
}

void fee() {
    printf("%f\n", PI);
}
```

**bar.c**

```c
#include "foo.h"

int main() {
    foo();
    fee();
}
```

Header files (usually)
- Function declarations
- Macro declarations

Source files (usually)
- Function definitions
- 1 main function
C workflow

Source files (.c, .h)
Code you write

Compile

foo.h
foo.c
bar.c

Object files

foo.o
bar.o

Link

Executable
Code you run on hardware

a.out

Library files

libc.a
<table>
<thead>
<tr>
<th>Compiling</th>
<th>Linking</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Creates object files</td>
<td>• Creates an executable</td>
</tr>
<tr>
<td>• Parses C code, generates assembly, and invokes assembler to produce object files.</td>
<td>• Links object files and libraries by replacing references to undefined symbols to their definitions</td>
</tr>
<tr>
<td>• Errors:</td>
<td>• Errors:</td>
</tr>
<tr>
<td>• Syntax errors</td>
<td>• Symbols not defined</td>
</tr>
<tr>
<td>• References to symbols (ex. functions) not declared</td>
<td></td>
</tr>
</tbody>
</table>
gcc: The C multitool

• Preprocess: expand macros (later) with -E
  • gcc -E ex.c -o out

• Compile: translate C into assembly with -S
  • gcc -S ex.c -o ex.s

• Compile: translate C into machine code with -c
  • gcc -c ex.c -o ex.o

-o <filename> ➔
put output in <filename>
Compiling with gcc

Translate source code into assembly

```
gcc -S foo.c -o foo.s
```

Translate assembly into machine code

```
as foo.s -o foo.o
```

Directly translate source code to machine code

```
gcc -c foo.c -o foo.o
```
gcc: The C multitool

• Preprocess:
  • gcc -E ex.c -o out

• Compile:
  • gcc -S ex.c -o ex.s
  • gcc -c ex.c -o ex.o

• Link: creates executable. By default does preprocessing, compilation, assembly, and linking
  • gcc -o ex ex.c (from source file)
  • gcc -o ex ex.o (from object file)
  • gcc -o ex ex.S (from assembly file)
C workflow

gcc -c foo.h foo.c bar.c

gcc -o a.out foo.o bar.o

gcc -c foo.h foo.c bar.c

gcc -o a.out foo.o bar.o

Automatically linked by gcc

Source files

foo.h

foo.c

bar.c

Compile

Object files

foo.o

bar.o

Link

Executable

a.out

Library files

libc.a

Automatically linked by gcc
Common gcc flags

- **g**  Produce debugging information for GDB
- **Ox**  Optimize (ex. **-O3**)
- **std=xxx**  Specify the standard (ex. **-std=c11**)
- **Wall**  enable warnings
- **-l**  Search for a library when linking

Many more! Run **man gcc**
ex00 - What does this do?

gcc -std=c11 -Wall -g ex00.c -o ex00
Make

make is a classic program for controlling what gets (re) compiled and how. Many other such programs exist (e.g., ant, maven, “projects” in IDEs, ...)

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

A makefile contains a bunch of triples

target: sources
  command

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

Syntax gotchas:
The colon after the target is required
Command lines must start with a TAB NOT SPACES
You can actually have multiple commands (executed in order); if one command spans lines you must end the previous line with \
Make Variables

You can define variables in a Makefile. Example:

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

Why?

Easy to change things Can change on make command line (CFLAGS=g)
More variables

It’s also common to use variables to hold list of filenames:

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

clean is a convention: remove any generated files, to “start over” and have just the source
Using make

```make
make -f nameOfMakefile Target
```

More commonly using defaults: `make`
- If no `-f` specified, use a file named `Makefile`
- If not target specified, use the first one in the file
Lots of Crazy Characters

- $@$ for target
- $^$ for all sources
- $<$ for left-most source
- % for pattern matching

Examples:

```plaintext
widget: foo.o bar.o
 $(CC) $(CFLAGS) -o $@ $^$
foo.o: foo.c foo.h bar.h
 $(CC) $(CFLAGS) -c $<
```