CMPSC 311 - Introduction to Systems Programming

Static/Dynamic Linking, C Preprocessor, and Make

Professors:

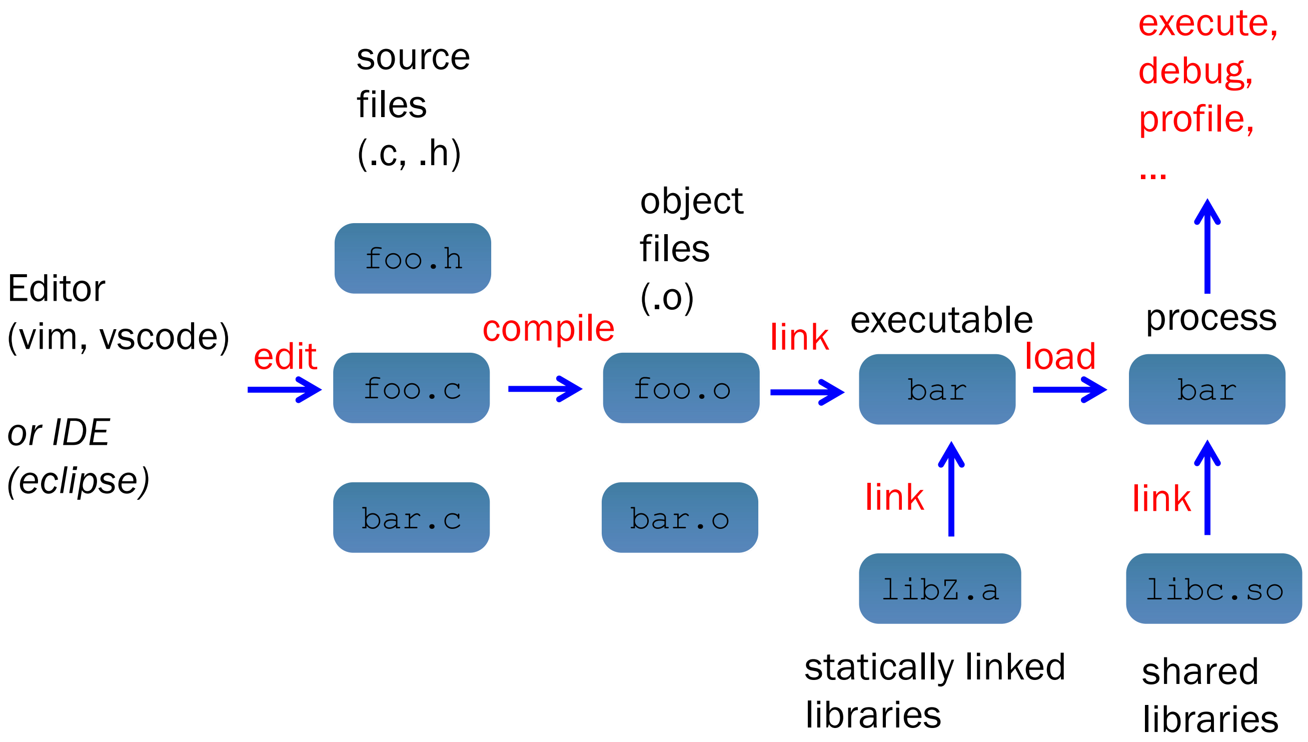
Suman Saha

(Slides are mostly by *Professor Patrick McDaniel* and *Professor Abutalib Aghayev*)

International Obfuscated C Code Contest

The International Obfuscated C Code Contest is a computer programming contest for the most creatively obfuscated C code. Held annually, it is described as "celebrating [C's] syntactical opaqueness". The winning code for the 27th contest, held in 2020, was released in July 2020. Wikipedia

C workflow



Editor
(vim, vscode)

or IDE
(eclipse)

edit

source
files
(.c, .h)

foo.h

foo.c

bar.c

compile

object
files
(.o)

foo.o

bar.o

link

executable

bar

link

libZ.a

statically linked
libraries

load

process

bar

execute,
debug,
profile,
…

link

libc.so

shared
libraries

Building a program 101

- There are two phases of building a program, compiling and linking
  - gcc is used to build the program
  - ld can be used to link the program (or gcc)

Compiling a program

- You will run a command to compile

```
gcc <source code> [options]
```

- Interesting options
  - -c (tells the compiler to just generate the object files)
  - -Wall (tells the compiler to show all warnings)
  - -g (tells the compiler to generate debug information)
  - -o <filename>.o (write output to file)
- An example

```
gcc hello.c -c -Wall -g -o hello.o
```

Linking a program

- You will run a command to link

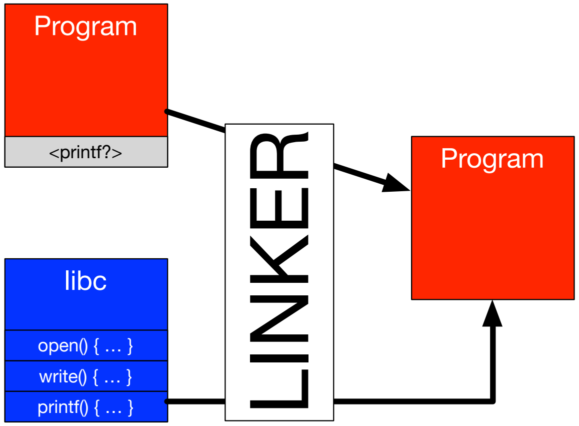
```
gcc <object files> [options]
```

- Interesting options
  - -l<lib> (link with a library)
  - -g (tells the compiler to generate debug information)
  - -o <filename> (write output to file)
- An example,

```
gcc hello.o goodbye.o -g -lmyexample -o hello
```

What is a “static” library?

- A library is a collection of related code and functions that are “linked” against a C program.
  - The library “exports” “symbols”
  - You program object code has “unresolved symbols”
  - The linker pulls chunks of the library containing those symbols and places them into the program
  - The program is done when all the pieces are resolved
  - It is called “static” linking because this is done at link time



Building a static library

PennState

- A statically linked library produces object code that is inserted into program at link time.
  - You are building an “archive” of the library which the linker uses to search for and transfer code into your program.

```
ar rcs lib<libname>.a <object files>
```

  - To run the command, use
- Library naming: with very few exceptions all static libraries are named `lib???.a`, e.g.,

```
ar rcs libmyexample.a a.o b.o c.o d.o
```

- You link against the name of the library, not against the name of the file in which the library exists (see linking)

Building a static library

- A statically linked library produces object code that is inserted into program at link time.
  - You are building an “archive” of the library which the linker uses to search for and transfer code into your program.
  - To run the

R - replace existing code with the objects passed
C - create the library if needed
S - create an index for “relocatable code”

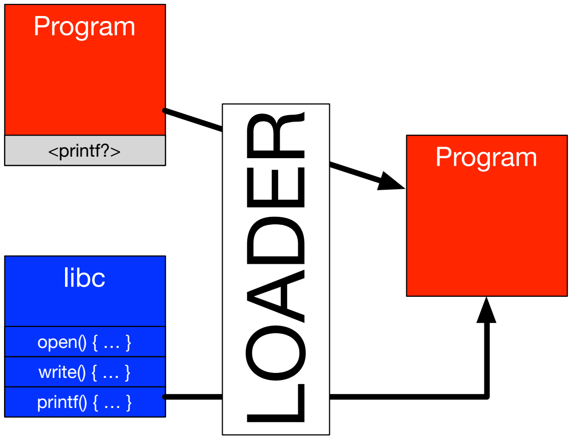
- Library nami amed lib???.a, e.g.,

```
ar rcs libmyexample.a a.o b.o c.o d.o
```

- You link against the name of the library, not against the name of the file in which the library exists (see linking)

What is a “dynamic” library?

- A dynamic library is a collection of related code and functions that are “resolved” at run time.
  - The library “exports” “*symbols*”
  - You program object code has “*unresolved symbols*” in the code
  - The loader pulls chunks of the library containing those symbols and places them into the process
  - The symbols are resolved when the process is started or later during execution
  - It is called “dynamic” because it can be done any time …



Building a dynamic library

- A dynamically linked library produces object code that is inserted into program at execution time.
  - You are building a loadable version of the library which the loader uses to launch the application

```
gcc -shared -o libmyexample.so a.o b.o c.o d.o
```

  - To run the command, pass to gcc using "-shared", e.g.,
- Important: all object files to be placed in library must have been compiled to position-independent code (PIC)
  - PIC is not dependent on any being located at any predefined location in memory
  - e.g., uses relative jumps, so does not matter where it is loaded at execution time
- Naming: same as before, only with `.so` extension

Building a dynamic library

- A dynamically linked library produces object code that is inserted into program at execution time.
  - You are building a loadable version of the library which the loader uses to launch the application

```
gcc -shared -o libmyexample.so a.o b.o c.o d.o
```

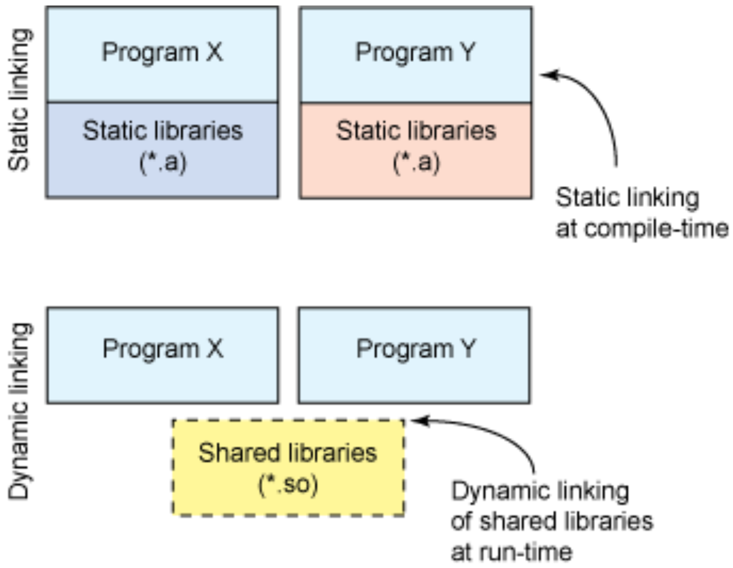
  - To run the com

```
gcc a.c -fpic -c -Wall -g -o a.o
```

- Important: all object files to be placed in library must have been compiled to position-independent code (PIC)
  - PIC is not dependent on any being located at any predefined location in memory
  - e.g., uses relative jumps, so does not matter where it is loaded at execution time
- Naming: same as before, only with `.so` extension

Static vs. Dynamic Linking

- Pros of static linking
  - Resolve all implementation issues at link time
  - Avoids problems of API versioning
  - Linker can optimize library code at link time
- Pros of dynamic linking
  - Reduced executable
  - System updates



The C Preprocessor

- The preprocessor processes input source code files for commands that setup the compile environment specific to that execution.
  - The programmer uses "#" directives to indicate what he wants that program to do.
  - We have seen one of these before, "#include"
  - There are more …

#include

- The `#include` directive tells the compiler to include data from some other data file.
  - `#include “foo.h”` - this tells the compiler to look in the local
    - (used for application programming)
  - `#include <foo.h>` - tells the compiler to look at the default directories and any provided by the command line.
- The `gcc -I<path>` option
  - tells the compiler to look in a specific directory for include files (that are using the <....h> approach)
  - Generally speaking, systems programming uses the <> approach to have better control over what is being included from where.

#define/#undef

- The #define directive allows the user to create a definition symbol that gets search/replaced throughout
  - commonly used to define a constant that might be changed in the future, e.g., the sizes of arrays ...
  - The #undef directive undoes binding ...

```
#define NUMBER_ENTRIES 15

...

int main( void ) {

  // Declare your variables here
  float myFloats[NUMBER_ENTRIES];

  // Read float values
  for ( i=0; i<NUMBER_ENTRIES; i++ ) {
    scanf( "%f", &myFloats[i] );
  }
```

#define macros

- #define can also be used to create simple functions called macros

```
/* Defining a function to swap pairs of integers */
#define swap(x,y) {int temp=x; x=y; y=temp;}

int main( void ) {

  // Declare your variables here
  int i = 1, j =2;

  ...

  swap(i,j);
}
```

- Macros are not “called” as normal functions, but are replaced during preprocessing (during compilation)
  - Thus no function call overheads such as stack operations

Conditional Compilation

- You can conditionally compile parts of a program using the #if, #ifdef, and #ifndef directives

```
#define DEFINED

...

#if 0
/* This does not get compiled */
#else
/* This does get compiled */
#endif

#ifdef UNKNOWNVALUE
/* This does not get compiled */
#else
/* This does get compiled */
#endif

#ifndef DEFINED
/* This does not get compiled */
#else
/* This does get compiled */
#endif
```

```
/* A quick way to comment out code,
as typically used in doing debugging
and unit testing */

int main( void ) {

  // Declare your variables here
  float myFloats[NUMBER_ENTRIES];

#if 0
  // Read float values
  for ( i=0; i<NUMBER_ENTRIES; i++ )
{
    scanf( "%f", &myFloats[i] );
  }

  // Show the list of unsorted values
  printCharline( '*', 69 );
  printf( "Received and computed\n"
);
#endif

...
```

Make

- The make utility is a utility for building complex systems/program
  - figure out which parts of system are out of date
  - figure out what the dependencies are between objects
  - issue command to create the intermediate and final project files

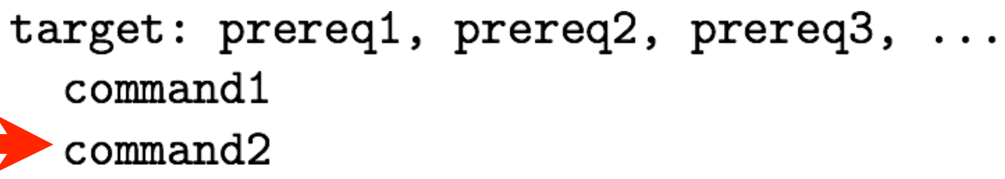
Note: being a good systems programmer requires mastering the make utility.

Make basics

- Each system you want to build has one (or more) files defining how to build, called the “Makefile”
  - A Makefile defines the things to build …
  - … the way to build them
  - … and the way they relate
- Terminology
  - Target - a thing to build
  - Prerequisites - things that the target depends on
  - Dependencies between files, e.g., a.o depends on a.c
  - Variables - data that defines elements of interest to the build
  - Rules - are statements of targets, prerequisites and commands

Makefile rules …

- Also known as a production, a rule defines how a particular item is built. The rule syntax is:



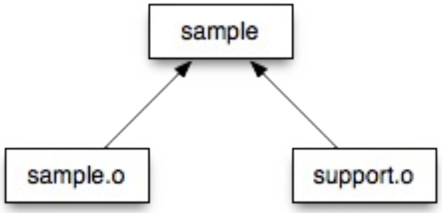
```
target: prereq1, prereq2, prereq3, ...
	command1
	command2
```

(*MUST BE TABBED OVER*)

- Where
  - target is thing to be built
  - prereq(1|2|3) are things needed to make the target
  - commands are the UNIX commands to run to make target
- **Key Idea**: run the (commands) to build (the target) when (any of the noted prerequisites are out of date)

Rule example

```
sample : sample.o support.o
  gcc sample.o support.o -o sample
```

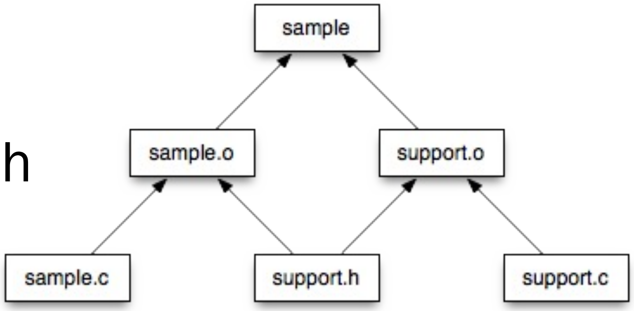


Dependency graph

What about the object files?

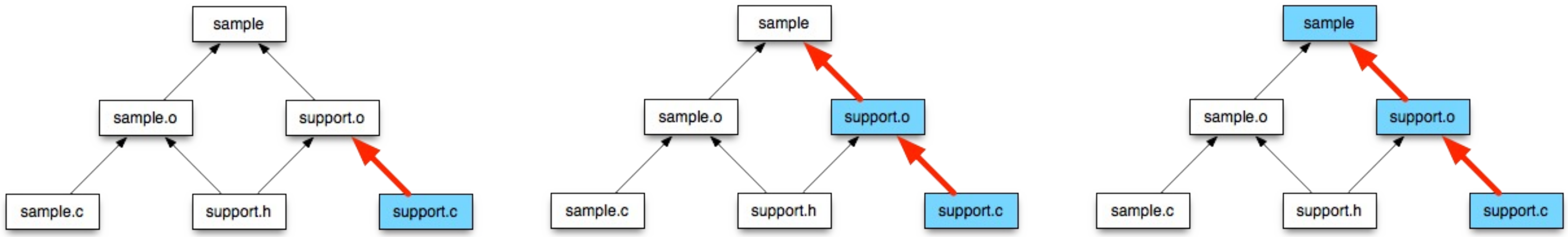
```
sample : sample.o support.o
  gcc sample.o support.o -o sample

sample.o : sample.c support.h
  gcc -c -Wall -I. sample.c -o sample.o

support.o : support.c support.h
  gcc -c -Wall -I. support.c -o support.o
```

Dependency graph



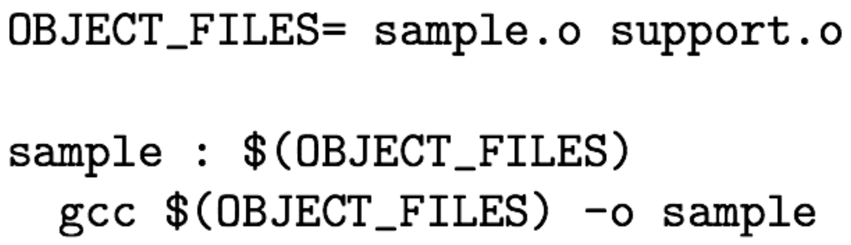
Running make

- To run make, just type `make` at the UNIX prompt.
  - It will open the `Makefile` (or `makefile`) and build any targets that are out of date
  - Thus … it will look at the dependency graph
  - E.g., consider if I edit `support.c` …



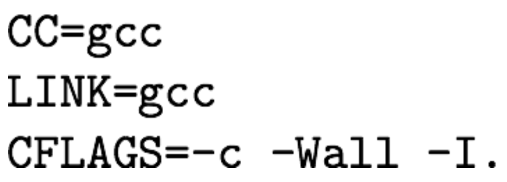
Variables

- Makefile variables allow you to replace some repetitive (and changing text) with others



```
OBJECT_FILES= sample.o support.o

sample : $(OBJECT_FILES)
  gcc $(OBJECT_FILES) -o sample
```

- Some standard variables include:



```
CC=gcc
LINK=gcc
CFLAGS=-c -Wall -I.
```

Built-in Variables

- Make supports a range of “special” variables that are used while evaluating each rule (called built-ins)
- Three of the most popular built-ins are
  - “$@” is the current rule target
  - “$^” is the prerequisite list
  - “$<“ is the first prerequisite

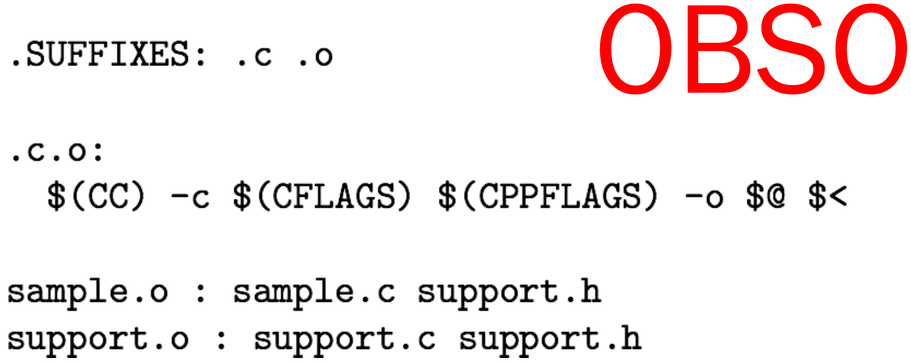
```
sample : $(OBJECT_FILES)
  $(CC) $^ -o $@

sample.o : sample.c support.h
  $(CC) $(FLAGS) sample.c -o $@

support.o : support.c support.h
  $(CC) $(FLAGS) support.c -o $@
```

- Built-ins are used to make builds cleaner …

Suffix rules

- A suffix rule defines a default way to generate a target from a type of prerequisites
  - You only need to define the dependency and not the commands for those files
  - Defining suffix rule:
    - Step 1: define the file types to be in suffix rules (.SUFFIXES)
    - Step 2: define the default productions
    - E.g.,

OBSOLETE



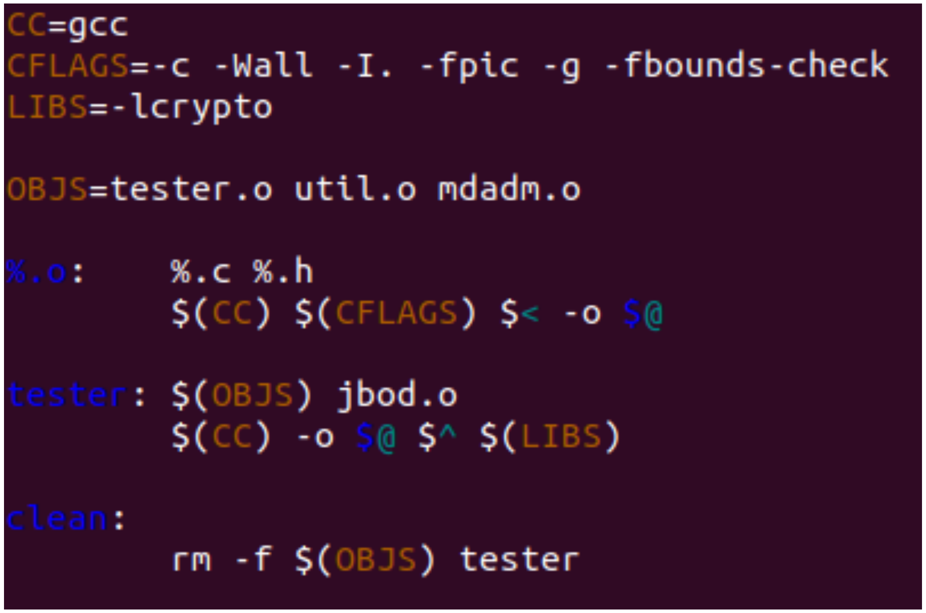
```
.SUFFIXES: .c .o

.c.o:
  $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

sample.o : sample.c support.h
support.o : support.c support.h
```

Pattern rules

- A pattern rule looks like an ordinary rule, except
  - the target contains exactly one ‘%’ character
  - ‘%’ can match any nonempty string, while other characters match themselves
  - prerequisites also use ‘%’ to show how their names relate to the target name
- The following rule says how to make file `foo.o` from file `foo.c`

```
%.o:    %.c %.h
        $(CC) $(CFLAGS) $< -o $@
```

Putting it all together …



```
CC=gcc
CFLAGS=-c -Wall -I. -fpic -g -fbounds-check
LIBS=-lcrypto

OBJS=tester.o util.o mdadm.o

%.o:    %.c %.h
        $(CC) $(CFLAGS) $< -o $@

tester: $(OBJS) jbod.o
        $(CC) -o $@ $^ $(LIBS)

clean:
        rm -f $(OBJS) tester
```