CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska
Winter 2010
Lecture 16 – Tools: linker, build scripts, make

Where We Are

- We are done with Linux, shell scripts, and C

- We are in the middle of learning about tools
  - Already completed: preprocessor, debugger, svn
  - Today: libraries, linker, and make

- Still to come: C++ and software engineering

Goal for Today

- At the end of today, you should understand
  - The sequence of operations involved in building an executable and what happens at each step
  - The goal of makefiles
  - Be comfortable writing simple makefiles

- This is not the end of the story
  - Much more to makefiles than what we will show
  - After this class, you should also learn about autoconf, automake, and cmake

Example

- Program composed of two modules
  - Queue module: `queue.c, queue.h`
    - Does not have a `main`
  - Main program: `main-queue.c` (uses queue)

Reminder: Header Files

header : A C file whose only purpose is to be included.

- generally a filename with the `.h` extension
- holds shared variables, types, and function declarations

key ideas:

- every ***name*`.c`** intended to be a module has a ***name*`.h`**
- ***name*`.h`** declares all global functions/data of the module
- other `.c` files that want to use the module will #include "***name*`.h`"

some conventions:

- `.c` files never contain global function prototypes
- `.h` files never contain definitions (only declarations)
- never `#include` a `.c` file (only `.h` files)

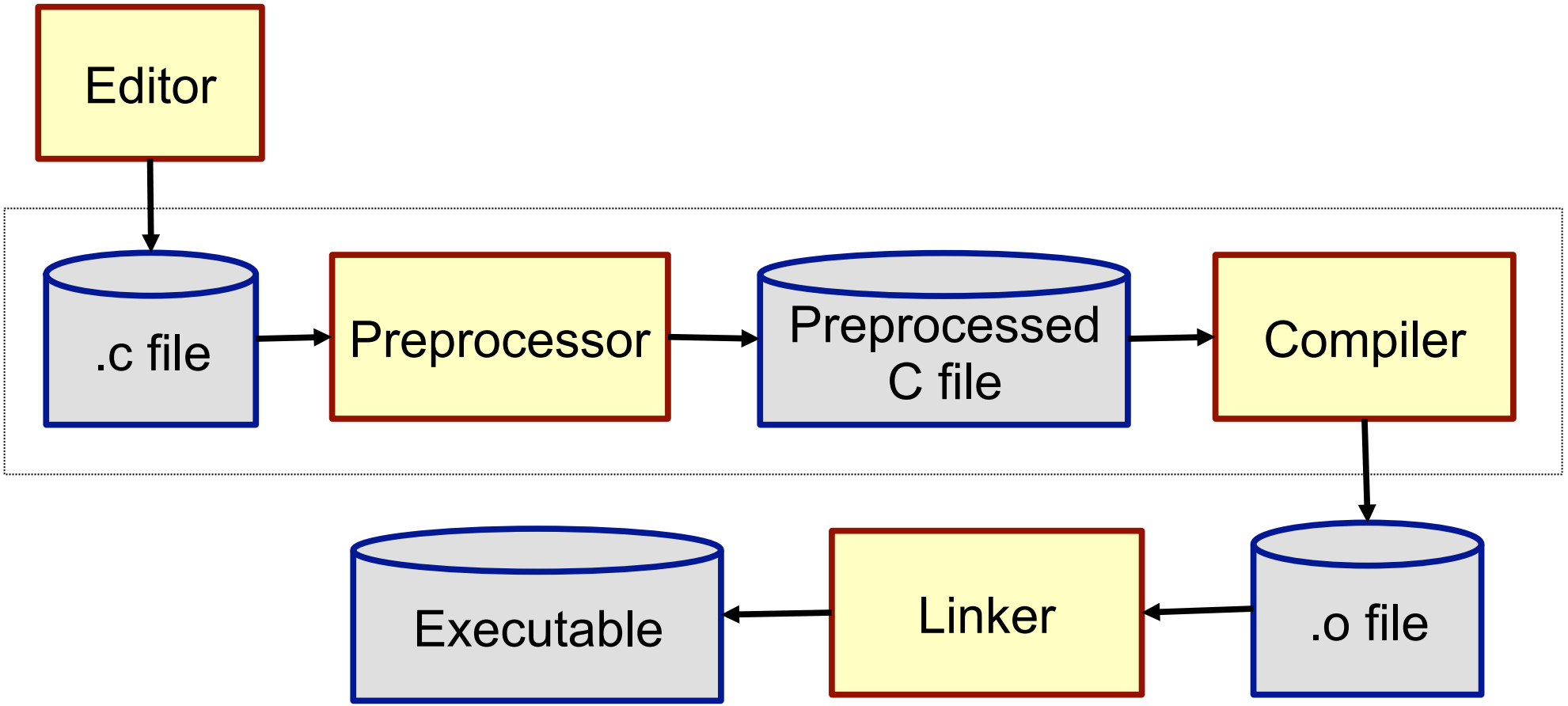
Back to our Example

- Program composed of two modules
  - Queue module: `queue.c, queue.h`
  - Main program: `main-queue.c` (uses queue)

- `main-queue.c` uses queue
  - For this reason it will `#include "queue.h"`
  - Now, it has enough information to be compiled by itself

Steps Involved in Creating a C Program

- Review from last lecture



Compiling our Example

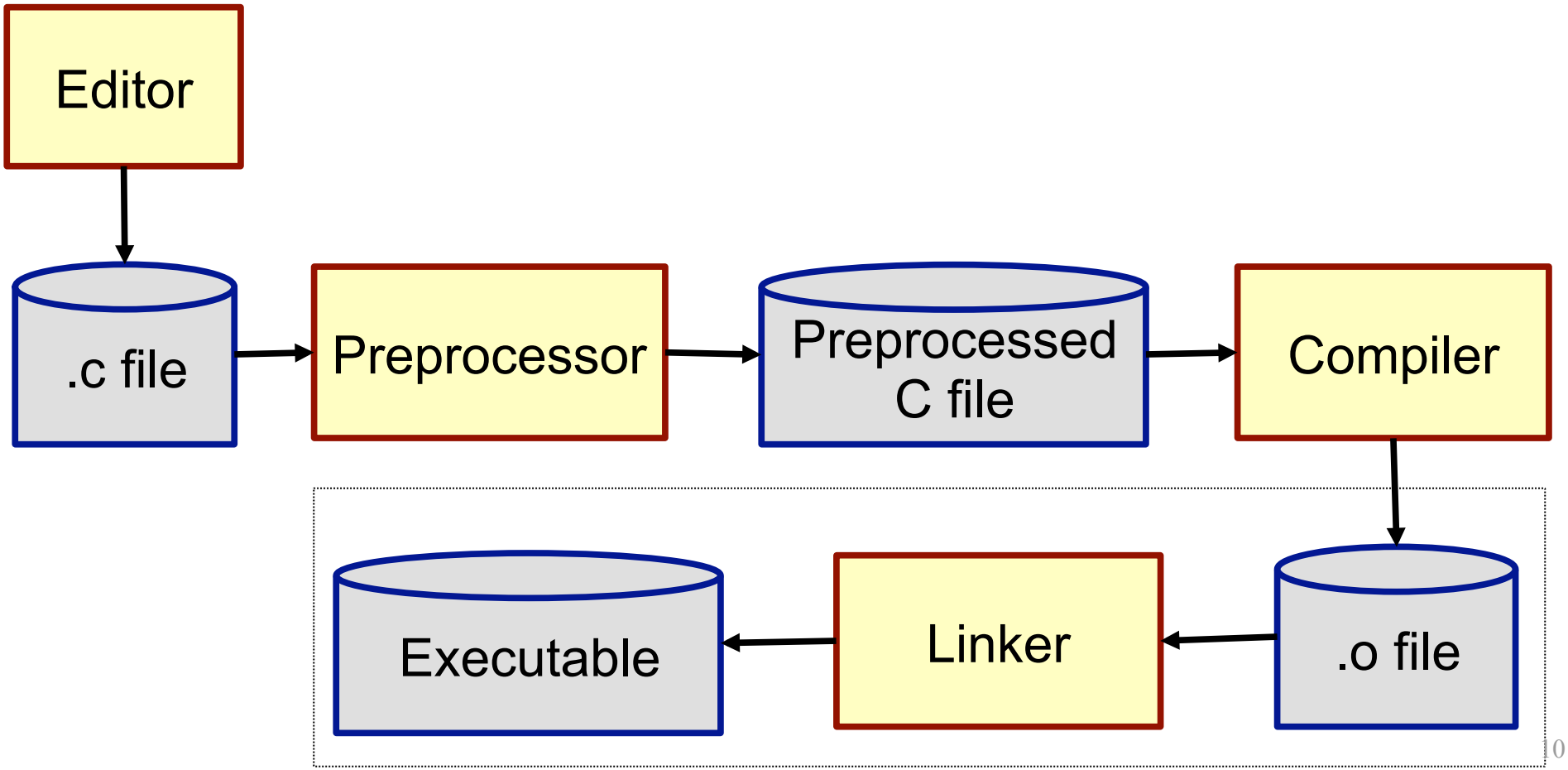
- Program composed of two modules
  - Queue module: `queue.c, queue.h`
  - Main program: `main-queue.c` (uses queue)
- Step 1&2: Preprocess and compile each `.c` file
  - Use option `-c` to produce the `.o` file
  - **Create `queue.o` and `main-queue.o`**
  - `gcc -Wall -g -c queue.c`
  - `gcc -Wall -g -c main-queue.c`

The Goal of the Linker

- Compiled code (`.o` file) is not "runnable"
- We have to link it with other code to make an executable
  - Where is the code for `printf` and `malloc`?
  - Where is the code for the queue module?
  - We only included the header files...
  - Need to find that code and put it in executable
  - That is what the linker does
- Normally, gcc/g++ hides this from you

Linking Step

- Linker transforms compiled code (.o files) into executable programs



Example

- Program composed of two modules
  - Queue module: `queue.c, queue.h`
  - Main program: `main-queue.c` (uses queue)
- Step 1&2: Preprocess and compile each `.c` file
  - Create `queue.o` and `main-queue.o`
  - `gcc -Wall -g -c queue.c`
  - `gcc -Wall -g -c main-queue.c`
- Step 3: Link files together to create executable
  - `gcc -o main main-queue.o queue.o`

Linking Overview

- If a C/C++ file uses but does not define a function (or global variable), then the `.o` has "undefined references"
  - Note: declarations do not count, only definitions
- Linker takes multiple .o files and "patches them" to include the references
  - Literally moves code and changes instructions like function calls
- Executable has no unresolved references
- Linker is called `ld`, but we will not invoke it directly. We will use `gcc`

Static Linking

- Puts all necessary code into executable
  - The `.o` files are no longer needed after linking

- Note: use option `-static` to also force the use of static linking for standard libraries

- Example: our queue test program
  - `gcc -static -o main main-queue.o queue.o`
  - (try linking with and without the `-static` option and see the difference in size of your executable)

Creating a Static Library

- To distribute the code for a module, it is convenient to put it all inside a library

- Let's try to put the code for the queue and stack inside a library

Creating a Static Library

- Create with `ar` (stands for "archiver")
  - `ar rc libdata.a queue.o stack.o`
  - Creates a static library named `libdata.a` and puts copies of object files `queue.o` and `stack.o` in it
  - If `libdata.a` exists, adds or replaces files in it
- Index the archive: `ranlib libdata.a`
  - Same as running `ar` with option `-s`
  - Improves performance during linking
  - Order inside the archive will no longer matter

Static Linking with Library

- Linking with library `libdata.a`

```
gcc -o main main-queue.o -L. -ldata
gcc -static -o main main-queue.o -L. -ldata
```

- Gcc will automatically link your executable with
  - `libgcc.a`
  - `libc.a` for C
  - `libstdc++.a` for C++

- Compile/link with option `-v` to see details

Static Linking Step-by-Step

- Begin: “Set of needed undefined functions” empty
- Action for .o file:
  - Include code in result
  - Remove all defined functions from set
  - Add to set all functions used but not yet defined
- Action for .a file: For each .o in order
  - If defines a needed function, proceed as above
  - Else skip
- End: If set of needed undefined functions empty, create executable, else error

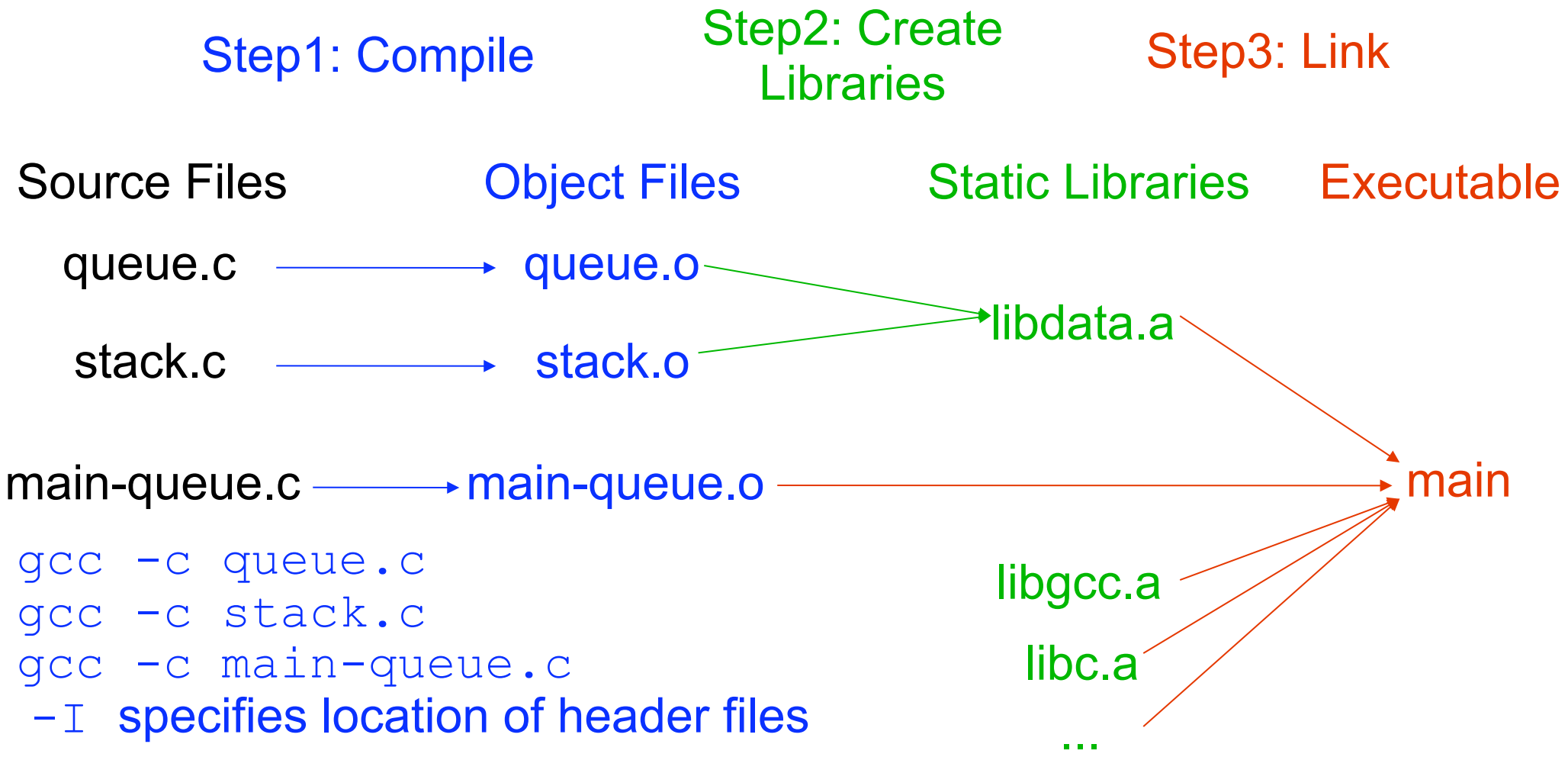
Library Gotchas

- Position of `-ldata` on command-line matters
  - Discover and resolve references in order
  - So typically list libraries after all object files
- Cycles
  - If two `.a` files need each other, you might need

```
-lfoo -lbar -lfoo ...
```

- If you include `math.h`, you'll need `-lm`
- Cannot have repeated function names

Summary of Building an Executable




```
gcc -c queue.c
gcc -c stack.c
gcc -c main-queue.c
```

`-I` specifies location of header files

```
ar rcs libdata.a stack.o queue.o
gcc -static -o main main-queue.o -L. -ldata
```

Dynamic Linking

- Static linking has disadvantages
  - More disk space, more memory when programs run
- Instead can use
  - Shared libraries (extension `.so`)
    - Link in when program starts executing
    - Saves disk space and memory
  - Dynamically linked/loaded libraries (while running)
- To experiment, link `main` with no option or with `-static`, or `-shared-libgcc`
  - In between commands execute: `ldd main`
    - Prints shared library dependencies
  - And also check the size of `main`

Linking and Libraries Summary

- Main steps when building executable
  - Preprocessing (specific to C)
  - Compiling
  - Linking
- Process can get complex for large systems
  - Definitely don't want to do manually each time
  - Would like to automate the process... Makefile
- Know about potential problems. Learn how to solve them as you encounter them

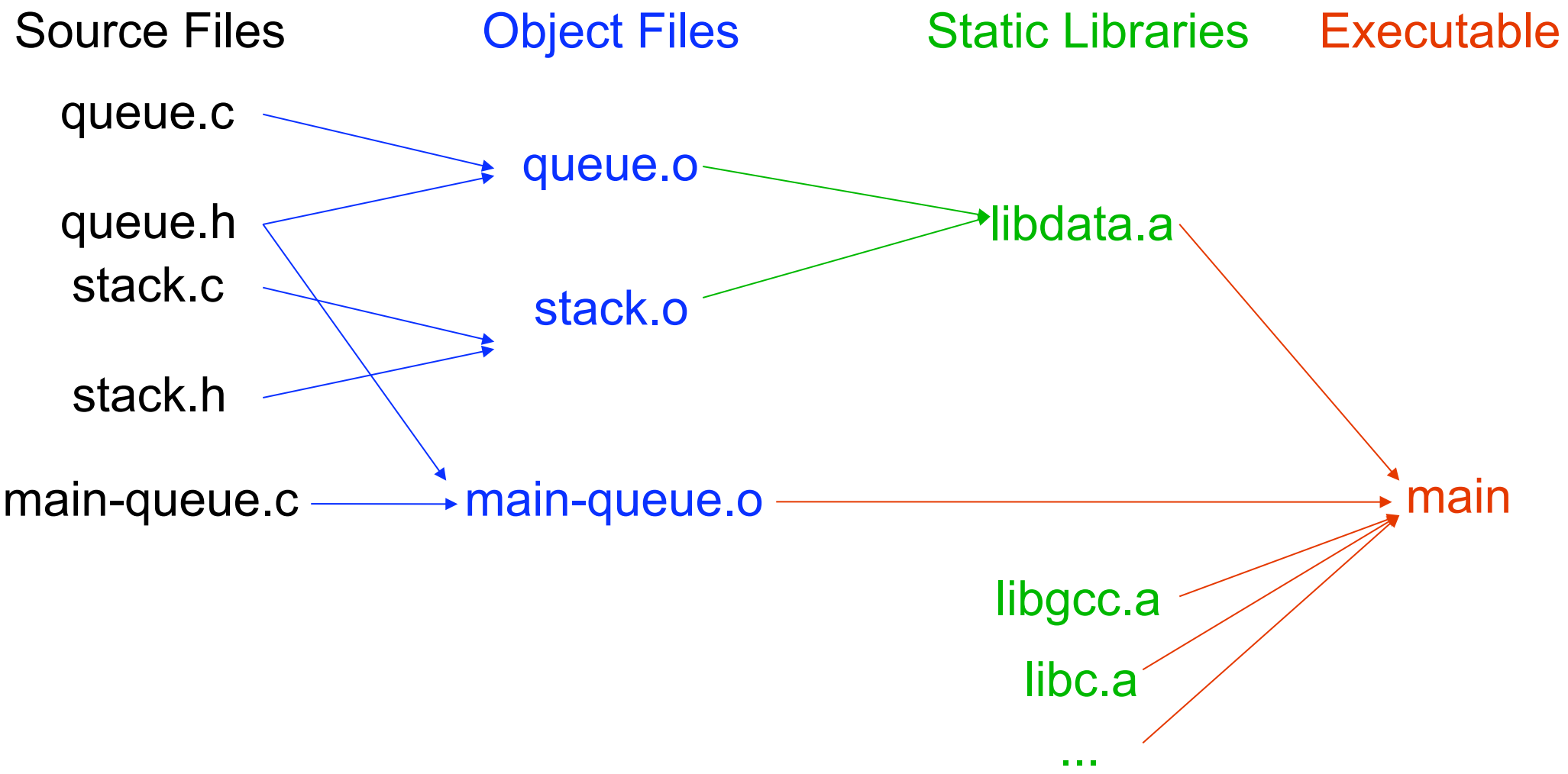
Make

- Two main goals
  - Automate the build process with a script
  - When a source file changes, rebuild only what is needed: keep track of *dependencies*
- Why?
  - Do not want to retype long, complex commands
  - Easier for others to build the system
  - Want to shorten build time
- Especially important for large systems

Recompilation Management

- The “theory” behind avoiding unnecessary compilation is a “dependency graph”
- To create target `t`, need
  - Sources $s_1, s_2, \ldots, s_n$
  - A command `a` that will create target from sources
- If `t` newer than all s_i, assume no reason to rebuild it
- Otherwise, recursive rebuild
  - If s_i is itself a target, check if need to rebuild it
  - If need to rebuild, use the given command `a`

Dependency Graph Example



Basic Idea Behind a Makefile

- Enables us to define targets & dependencies
- In form of triples: target, source, command(s)

```
target: sources (aka dependencies)
  command1
  command2
  ...
queue.o: queue.c queue.h
  gcc -Wall -c queue.c
```

- Warning: command lines must start with TAB
- If a command spans multiple lines, use \

Make

- On the command line

```
make -f nameOfMakefile target
```

- Defaults
  - If no `-f`, looks for a file named `Makefile`
  - If no target specified, uses first target in the file
- The make utility
  - Examines the dependency graph
  - Examines the file-modification times
  - Recursively decides what to rebuild
  - Note: make is **language independent** (java, c, latex)

Standard Targets

- all: make everything

```
all: main-queue main-stack
```

- clean: remove any generated files, to "start over" and have just the source

```
clean:
	rm -f *.o main-queue main-stack
```

- Phony targets: "all" and "clean" never exist

Variables

- We have seen the basics, now let's get more sophisticated with our Makefiles
- You can define variables in a Makefile

```
OBJ = main-stack.o stack.o

main-stack: $(OBJ)
  gcc -o main-stack $(OBJ)
```

- Help avoid error-prone duplications
  - List of object files
  - List of executables
- In make, variables are often called macros

Default Macros

- There exists a lot of default macros
- You must respect the naming conventions
- Override defaults in the Makefile

```
CC = gcc
CFLAGS = -Wall -g
queue.o: queue.c queue.h
  $(CC) $(CFLAGS) -c queue.c
```

- Override defaults with environment variables

```
export CFLAGS =”-Wall -g”
```

- View list of macros: `make -p`

Revenge of Funny Characters

- Internal macros
  - `$@` designates the current target
  - `$^` designates all prerequisites
  - `$<` designates left-most prerequisite
- Pattern rules

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

- Basic ones already defined
  - They are called implicit rules

Dependencies

- Our Makefile is starting to look quite elegant
- But, we are still listing dependencies manually
  - Keeping track of dependencies is hard
  - It is easy to forget some header files
- This is not make's problem
  - Make has no understanding of programming languages. It only understands rules
- Because this is error-prone, there are often language-specific tools that can keep track of dependencies for you

Dependency-Generator Example

- `gcc -MM [src files]`
  - Useful variants include `-M` and `-MG (man gcc)`
  - Automatically creates a rule for you
  - One approach, run via a phony `depend` target

```
depend: $(SRC)
  $(CC) -M $^ > .depend
```

  - Then `include` the resulting file in your Makefile

```
include .depend
```

- `makedepend` combines many of these steps
- Read more if you are interested in this topic

Installing Program from Source

- You don't need to know this for the class
- Typical four steps when installing software

`autoconf` (sometimes `setup` script instead)

```
configure --prefix=/where/to/install/
```

```
make
```

```
make install
```

- Configure script: defines variables needed in the Makefile, performs various checks before compiling
- Configure script has many options so try

```
configure --help
```

Readings

- Programming in C
  - Chapter 15 and Appendix C
- Make/Makefile tutorials
  - http://www.gnu.org/software/make/manual/make.html
  - http://www.eng.hawaii.edu/Tutor/Make/
- Extra references: man pages for gcc, ranlib, ar, ld
- In the future (no need to read for this class)
  - autoconf/automake: http://www.gnu.org/manual/
  - cmake: http://www.cmake.org/