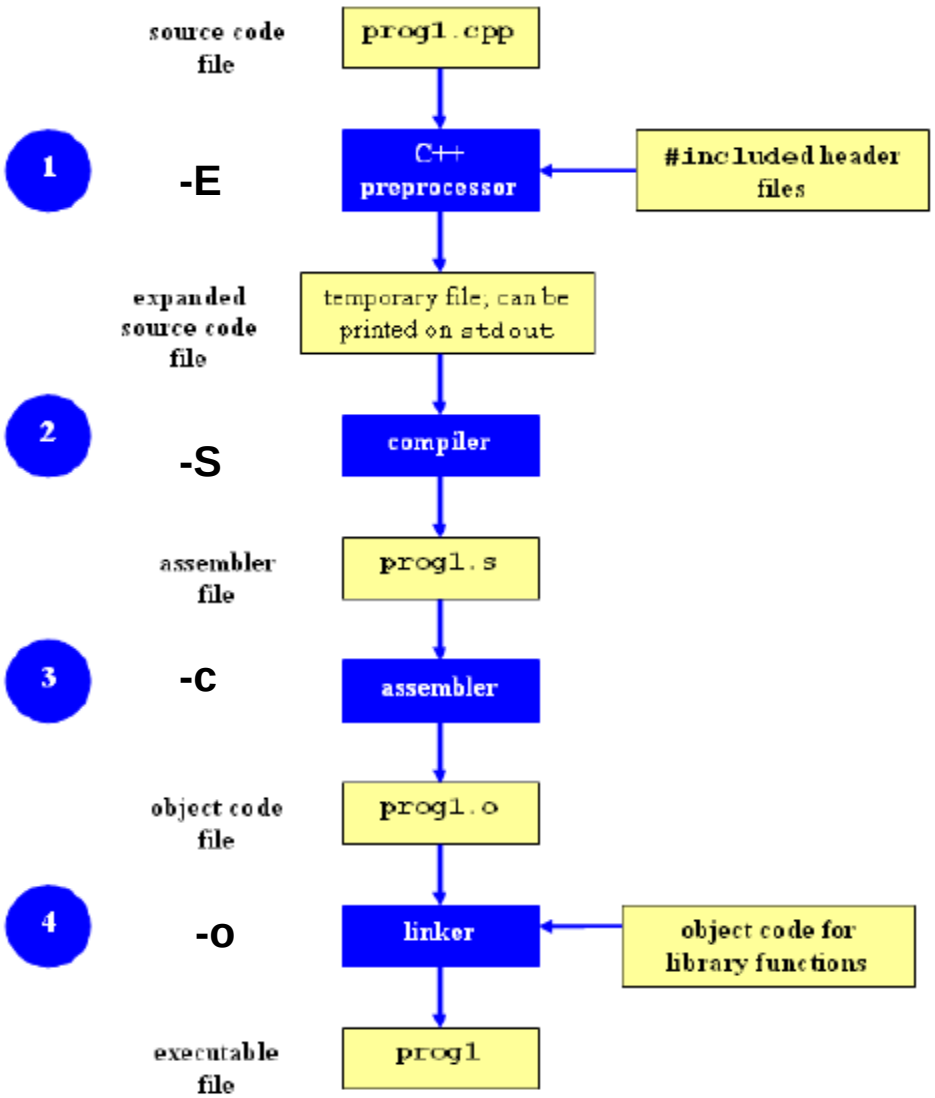
Introduction to Make

CS 35L
Spring 2020 – Week3

Compilation Process



- Preprocessor: handles include.h, #define.h , Strip out comments. Adds these expanded files to the program
- Compiler: Translates C to assembly. Produces .S file
- Assembler: translates assembler file to object .o file (not executable)
- Linker: produces executable

- Ex: g++ -c file.cpp (creates file.o)

C++ Header (.h) files

- Has a .h extension
- Contains Classes, function prototypes
- The implementation of the class goes in the .cpp file

Example .h\.cpp file

Num.h: defines classes and Function prototypes

Num.cpp: program that performs the Num function. includes Num.h

main.cpp: main program.
-Includes Num.h
- Uses the Num program

```
File: Num.h
class Num {
        private:
                int num;
        public:
                Num(int n);
                int getNum();
};
```

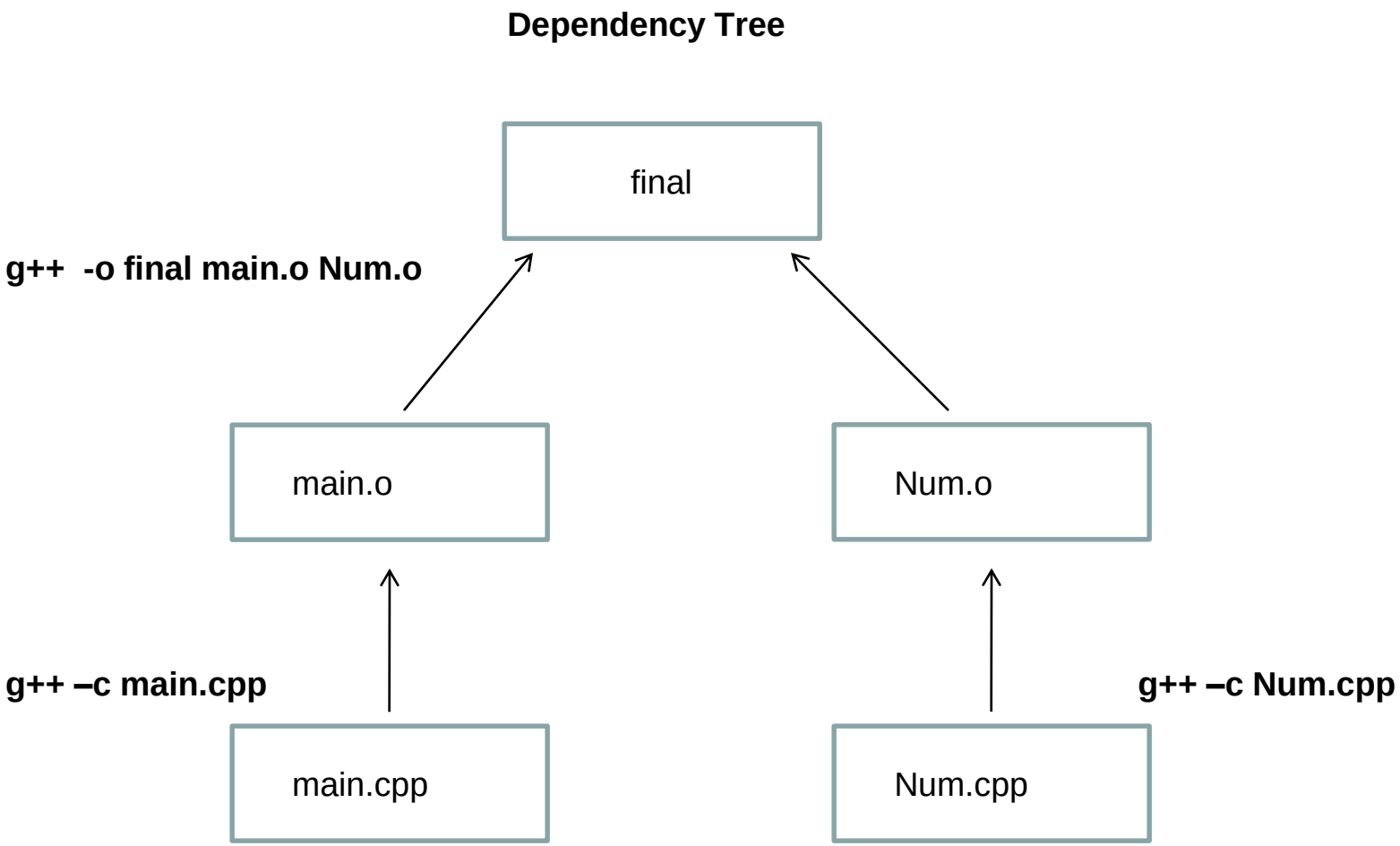
```
File: Num.cpp
#include "Num.h"
Num::Num() : num(0) { }
Num::Num(int n): num(n) {}
int Num::getNum() {
        return num;
}
```

```
File: main.cpp
#include <iostream>
#include "Num.h"
using namespace std;
int main() {
        Num n(35);
        cout << n.getNum() << endl;
        return 0;
}
```

Compiling files

- For any changes in Num.h both Num.ccp and main.cpp will have to be compiled.
- **Best practice is to compile programs separate and then link them.**
- Including compiled programs inside other programs is not recommended:
  - Will slow down compiling process
  - Requires main program to compile if changes occur in the program that was included in it.

Linking two object files

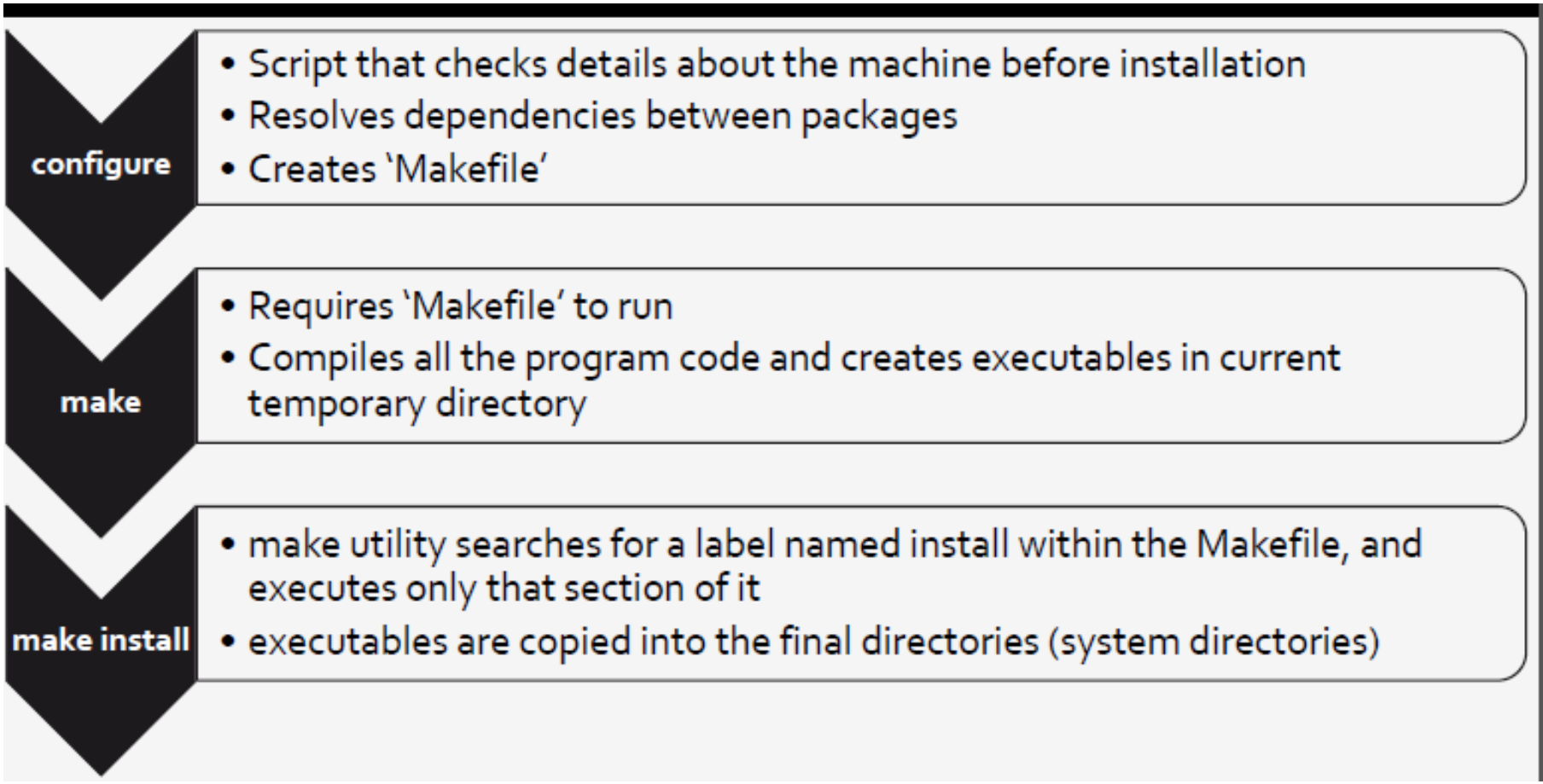


Make

- **Make** is a build automation tool that automatically builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program.
- It manages large software projects, offering a level of automation.
- Compiles files and keeps them up-to-date
- Offers efficient compilation: Only files that need to be recompiled

Compiling from scratch

- **./configure --prefix=some-path** (absolute path)
  make
  make install (built programs will be copied to some-path)



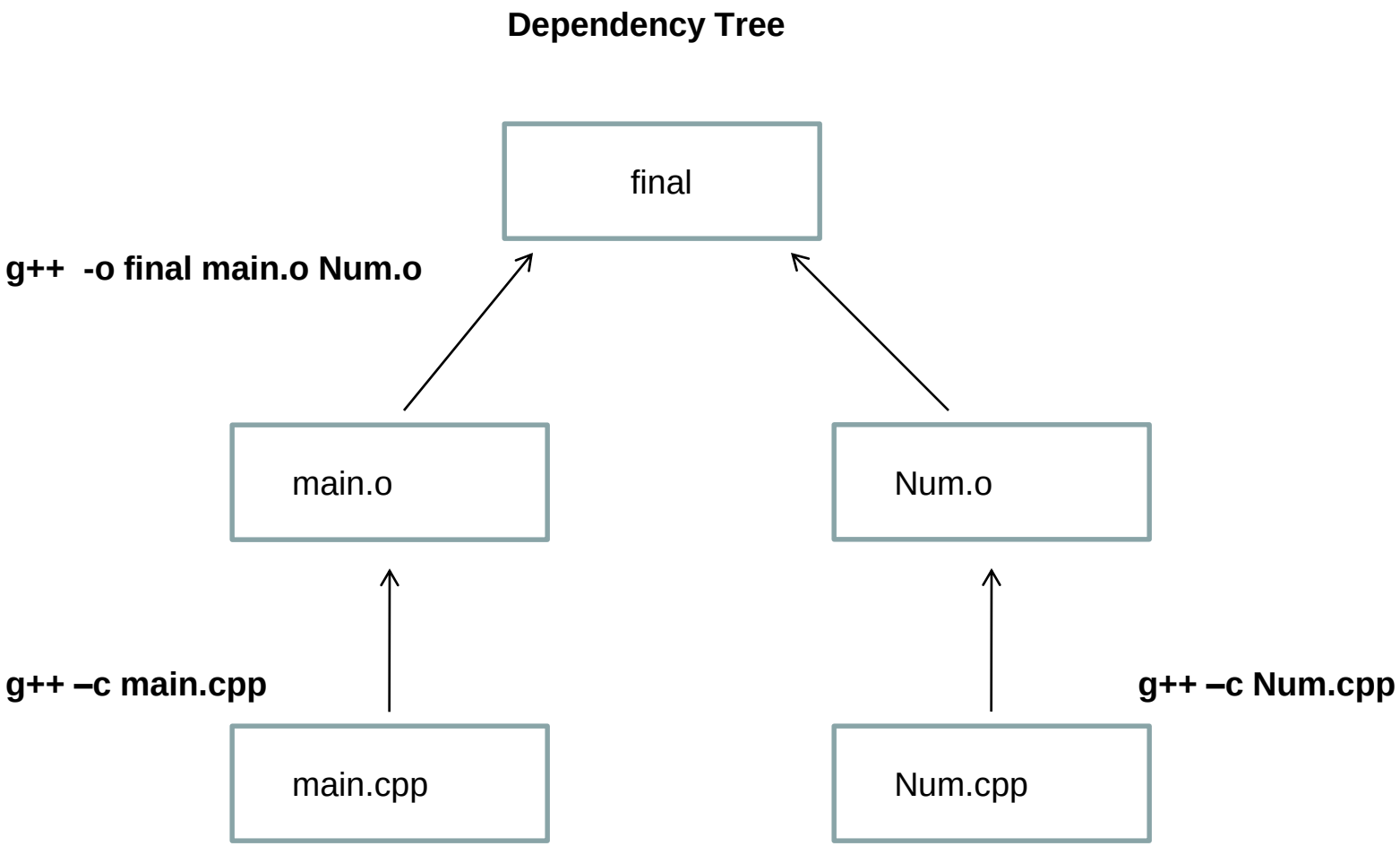
Configure script

- *https://thoughtbot.com/blog/the-magic-behind-configure/make/make-install*
- Designed to aid in developing a program to be run on a wide number of different computers
- `configure` is application specific
  - software provides it's own configure script
- Creates the Makefile
  - Can change default behavior with options
  - `./configure --` help for more info

Makefile and make

- We need a file that instructs make how to compile and link a program. Called a Makefile.
- The make program allows you to use macros, which are similar to variables to codify how to compile a set of source code.
  - Macros are assigned as BASH variable:
    - CFLAGS= -O -systype bsd43
    - LIBS = "-lncurses -lm –lsdl”
- Makefile is invoked with
  - make <target_name>

How does Makefile help here?



Makefile

- **Makefile**: contains a list of rules. These rules tell the system what commands you want to be executed. Most times, these rules are commands to compile (or recompile) a series of files.

```
#  -*-  MakeFile  -*-          (defines this file as a Makefile)
# target:dependencies          (this is the dependency tree)
#     action                   (there is a Tab before action)

all:  final clean
final: main.o  Num.o
    g++  -o final main.o  Num.o
main.o: main.cpp
    g++  -c main.cpp
Num.o:Num.cpp
    g++  -c Num.cpp
clean: rm  –f  main.o  Num.o
```

rule

Standard “targets”

- People have come to expect certain targets in Makefiles. You should always browse first, but it's reasonable to expect that the targets all (or just make), install, and clean will be found

  - **`make`** - compile the default target

  - **`make all`** - should compile everything so that you can do local testing before installing things.

  - **`make install`** - should install things in the right places. But watch out that things are installed in the right place for your system.

  - **`make clean`** - should clean things up. Get rid of the executables, any temporary files, object files, etc.

Patching

- A patch is a piece of software designed to fix problems or update a program.

- It is a **diff** file that includes changes made to a file.

- The program that has the bug can be fixed by applying the patch to that program

- patch pnum < patch_file

If path is a/src/ls.c

p1: src/ls.c

Applying a Patch

diff Unified Format

- --- path/to/original_file
- +++ path/to/modified_file

- @@ -l,s +l,s @@
  - @@: beginning and end of a hunk
  - l: beginning line number
  - s: number of lines the change hunk applies to for each file
  - A line with a:
    - - sign was deleted from the original
    - + sign was added in the new file
    - ‘ ‘ stayed the same

Patching

- `cd` into directory patch considers pwd
- emacs patch_file: copy and paste the patch content
- `patch [options] [originalfile [patchfile]]`
- `patch -p`**`num`** `<patch_file`
- *`man patch`* to find out about p**num**
- BE AWARE: p**num** defaults to p1 if omitted
- `cd` into the coreutils-8.29 directory and type `make` to rebuild patched `ls.c`
- More patch command examples - link

How do you download a file

- Wget
- apt-get (Advanced Package Tool)
  - Debian Linux, Ubuntu Linux
- rpm (Redhat Package Management)
  - RedHat Linux

Unzipping -Tar commands

- `tar –cvf <tarfilename.tar> <target directories>` - creates tar file.
- `tar –tvf <tarfilename.tar>` - list tar file contents
- `tar –xvf <tarfilename.tar>` - extracts tar file
- Can add `–J` flag for .xz files
- USAGE:
  - Always create tarfile in target directory (relative file/directory names)
  - Always list tarfile before extracting (insure relative file names)
  - Always extact tarfile in target directory (relative file/directory names)
- Example:

```
tar –xJvf ~/bb-1_3a_tar.xz
```

Lab 3 (compiling coreutils)

- Download coreutils-8.29 to your home directory (use wget)
- Untar and unzip it
- Make a directory “coreutilsinstall” in your home directory (this is where the coreutils will be installed)
- Go to the coreutils-8.29 you just unzipped
- Read the INSTALL file; especially section on –prefix
- Run ./configure with the prefix flag that will install the files in the “coreutilsinstall” directory
- Compile using Make
- Then use Make install

Lab 3 (check the bug inside coreutils)

```
[User:-)@lnxsrv07 ~/cs35L/lab3/coreutils/bin]$ ls -l /bin/bash
-rwxr-xr-x 1 root root 960376 Jul  8  2015 /bin/bash


[User:-)@lnxsrv07 ~/cs35L/lab3/coreutils/bin]$ ./ls -l /bin/bash
-rwxr-xr-x 1 root root 960376 2015-07-08 04:11 /bin/bash
```

- run the ls command using ./ls not just ls, in order to use the newly built coreutils