



The dwarf planet Makemake

Examples

An **excellent** place to find a couple example Makefiles (ones that follow the content in these slides) is:

`~rjchaney/Classes/cs333/src/make`

Look at both `Makefile` and `Makefile_vars`

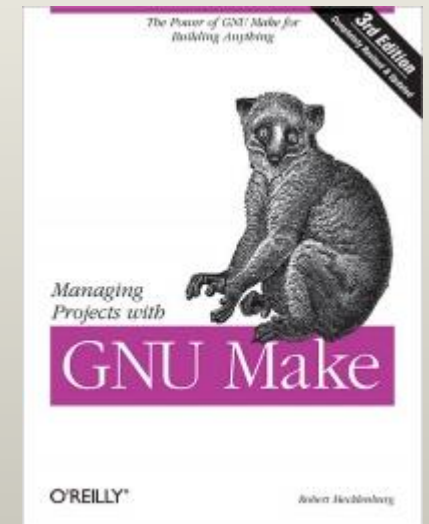
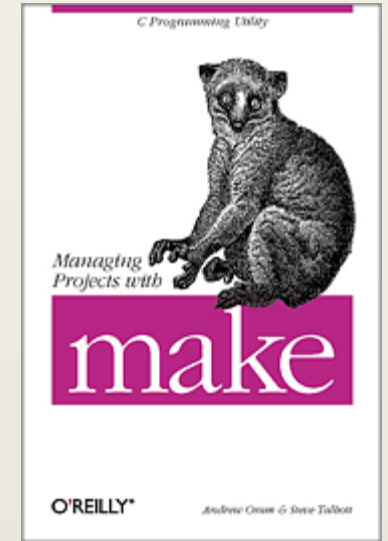
You'll need to create a `Makefile` to support your **C code** labs and assignments.

I have placed links to a some web resources in Canvas (Resources for `make`).

There are some books on `make` through the main Library and the GNU `make` book is **freely available** from O'Reilly as a pdf.

The GNU `make` manual is also online.

<https://www.gnu.org/software/automake/manual/make.pdf>



The 4 Stages of Compilation of a C Program

preprocessor

- Expansion of header file
- Substitute macros and inline functions
- Comments are removed

The C PreProcessor stage is often called cpp.

compiler

- Generates assembly language
- Verification of function usage and prototypes

- Generates machine code instructions
- Generates relocatable object file

assembler

- Binds necessary libraries
- Generates executable program

linker

You will see this on an exam.

Common Command Line Options for gcc

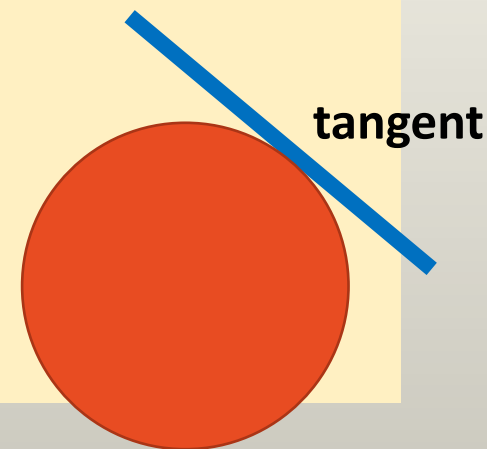
There are a couple command line options for the `gcc` compiler that you are going to be using.

The `man` page for `gcc` is a tad bit lengthy (18,000+ lines), so I'm going to call out 3 options:

`-O`

`-g`

`-c`



Command Line Options for gcc

`-o file`

Place **output** from the `gcc` compiler in file `file`. This applies to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

If `-o` is not specified, the default is to put an executable file in file named `a.out`, the object file for `source.c` is `source.o`, its assembler file is `source.s`, a precompiled header file is `source.c.gch`, and all preprocessed C source on standard output.

With the `-o` option, you specify the name of the output that you want to create. We **want** to be specific.

Command Line Options for gcc

-g

Produce **debugging** information in the operating system's native format (stabs, COFF, XCOFF, or DWARF 2). GDB can work with this debugging information.

Debugging with GDB is not quite like using Visual Studio, but it can be a lot better than just using print statements in your code.

You will be using this command line option.



Command Line Options for gcc

-c Compile or assemble the source files, but do not link.

The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix `.c`, `.i`, `.s`, etc., with `.o`.

Unrecognized input files, not requiring compilation or assembly, are ignored.



This is the `gcc` option to use to create the object file (the `.o`) from the `.c` file. The name given to the `.o` file will be the same as the name of the `.c` file, with the `.o` substituted for `.c`.

The `make` utility in Unix is one of the original tools designed by S. I. Fieldman of AT&T Bell labs in 1977. There are several/many versions.

We will be using the GNU version of `make`, the default for Linux.

GNU `make` has a LOT of options and capability. We will actually only use a small portion of that.



`make` is a program building tool which runs on Unix, Linux, and other various like operating systems.

You can even use `make` for Visual Studio development!!!

`make` allows you to describe build **dependencies**

between source code and object code or executables. It also allows you to **describe** how to bring dependencies **up to date**.



- When you are building a small program, with only a couple `.c` and `.h` files, a `Makefile` may not seem necessary.
- When you have a large program with many `.c` and `.h` files, a `Makefile` is a requirement. Especially when you have multiple developers working on the project.
- When you build a very large project that is made from many (maybe hundreds) files, you only want to rebuild the portions that have changed since the last build.



In `~rchaney/Classes/cs201/src/make` there are some sample files that will give you some practice with reading and writing Makefiles.

I like to make the first letter of my Makefile a capital M.

By default, when you run the `make` command it will look for a file called `GNUmakefile`, `makefile`, or `Makefile`, in that order.

You run make by typing `'make'` on the command line.



```
// The hellomake.c file
```

```
#include <stdio.h>  
#include "hellomake.h"
```

Both C files include
the `hellomake.h`
file.

```
int main( void )  
{  
    // call a function in another file  
    myPrintHelloMake();  
  
    return(0);  
}
```

```
// The hellofunc.c file
```

```
#include <stdio.h>  
#include "hellomake.h"
```

```
void myPrintHelloMake( void )  
{  
    printf("Hello makefile!\n");  
  
    return;  
}
```

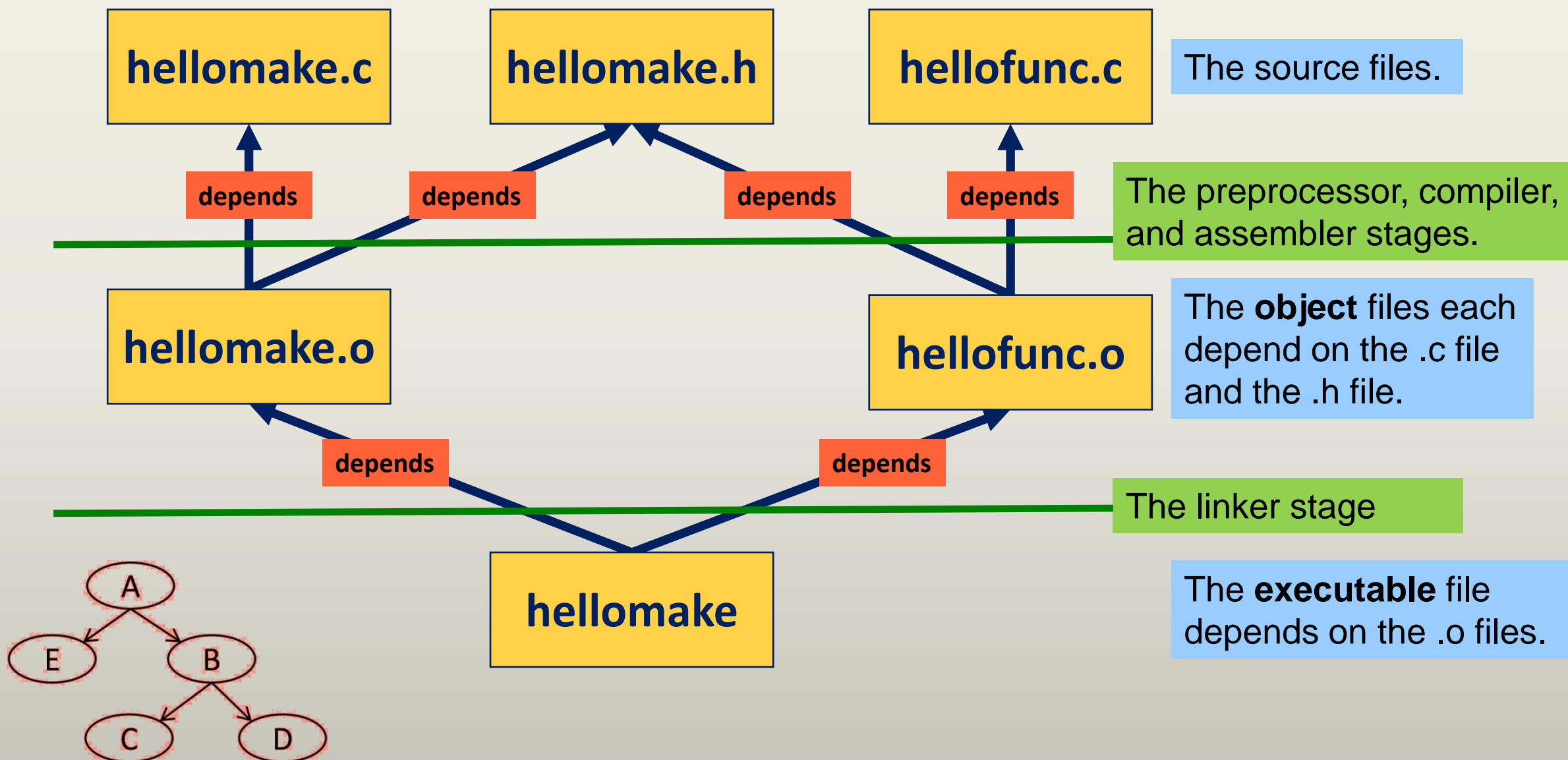
I want to build the program
`hellomake` using the 3 files
shown here.

```
// The hellomake.h file.
```

```
void myPrintHelloMake( void );
```



The File Dependency Tree



“**Depends on**” means that the target (dependent file or target) **must be rebuilt** any time one of the **prerequisite** files is **newer**.

“**Newer**” is based on timestamp of the file.

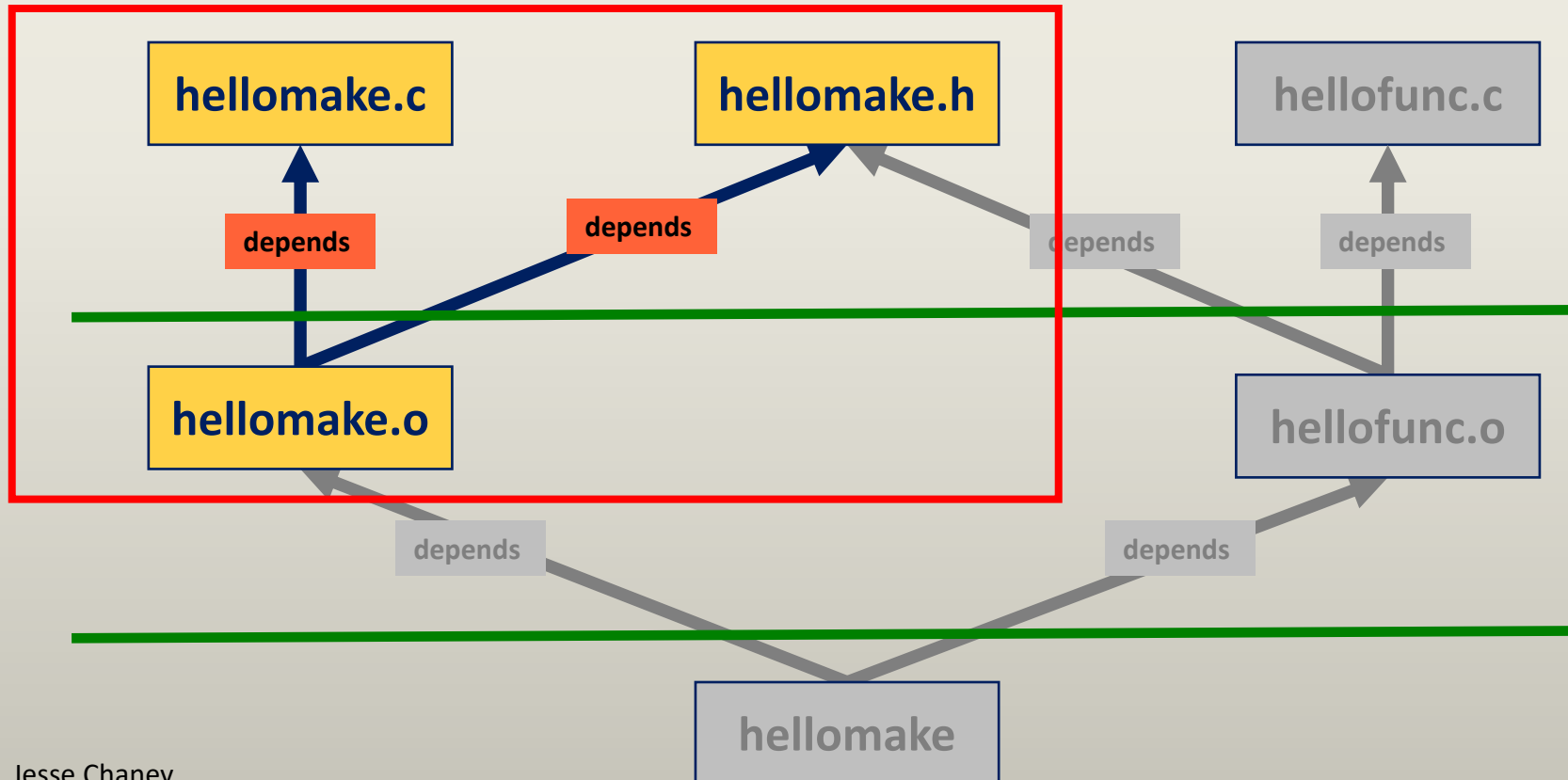
A more recent timestamp is newer than an less-recent timestamp.

Today is more recent than yesterday.



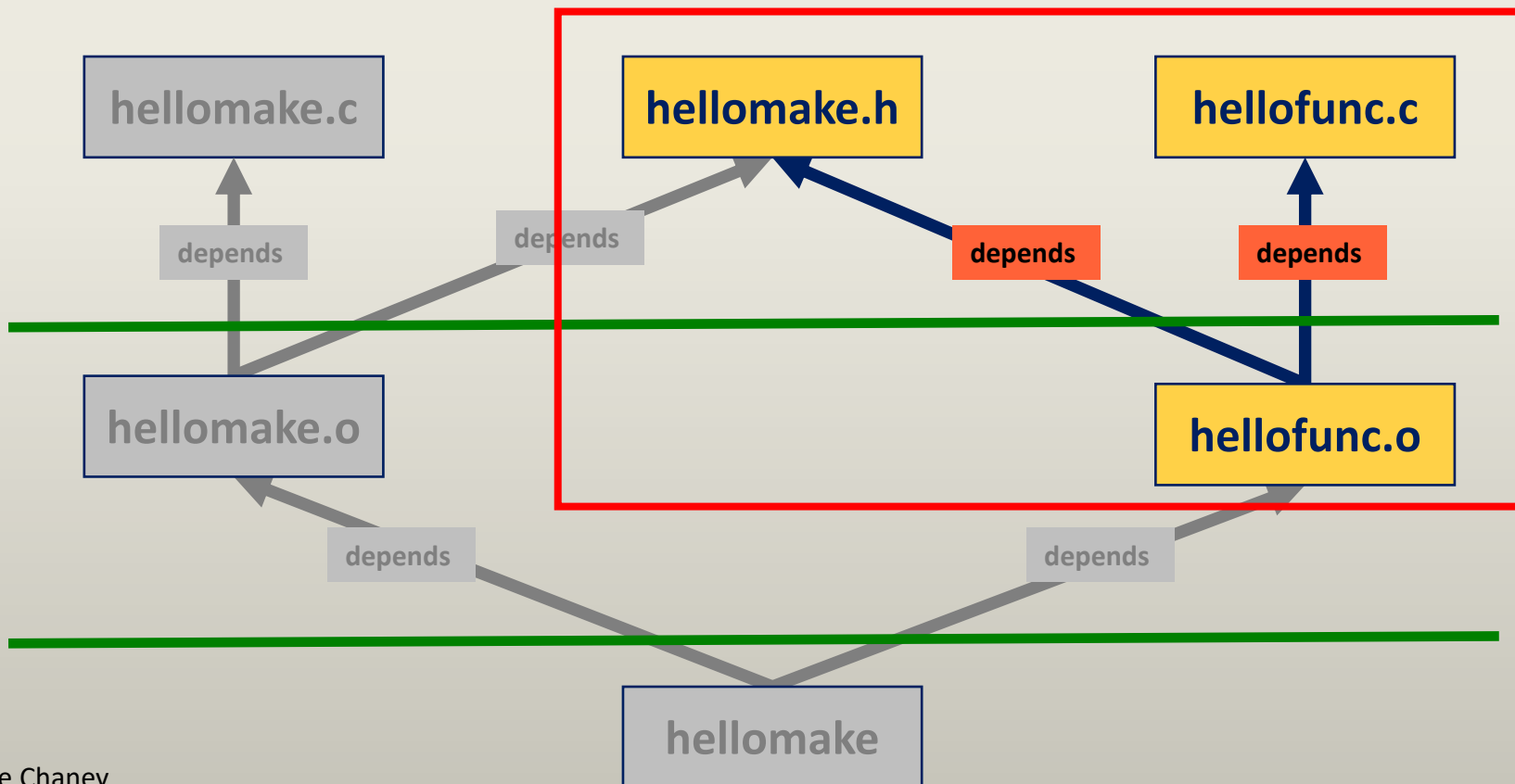
The File Dependency Tree

- The `hellomake.o` file **depends on** the `hellomake.c` file and the `hellomake.h` file.
- The `hellomake.c` file and the `hellomake.h` are **prerequisites** to the `hellomake.o` file.



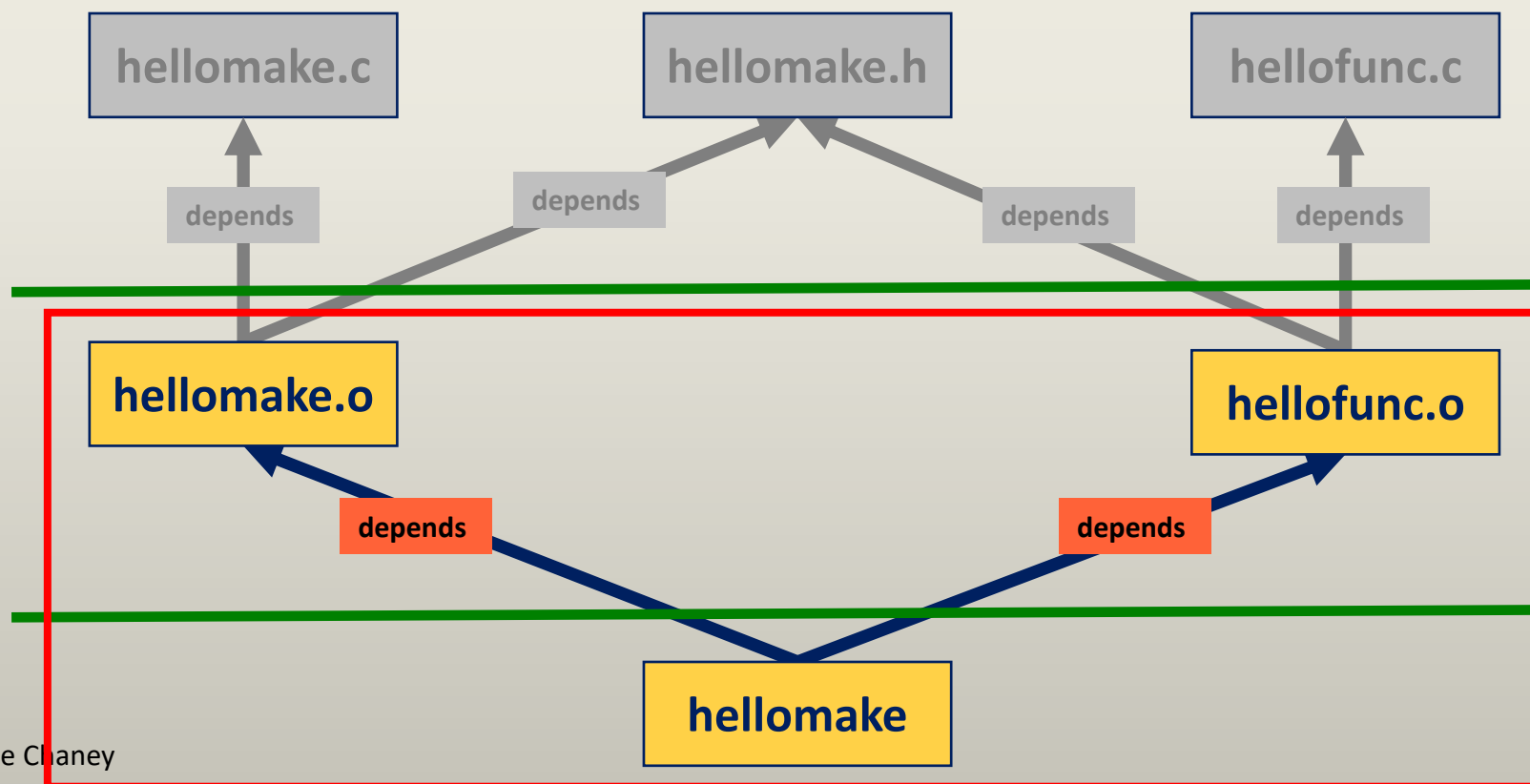
Anytime `hellomake.c` or `hellomake.h` are newer than the `hellomake.o` file, the `hellomake.o` file must be rebuilt.

- The `hellofunc.o` file **depends on** the `hellofunc.c` file and the `hellomake.h` file.
- The `hellofunc.c` file and the `hellomake.h` are **prerequisites** to the `hellofunc.o` file.



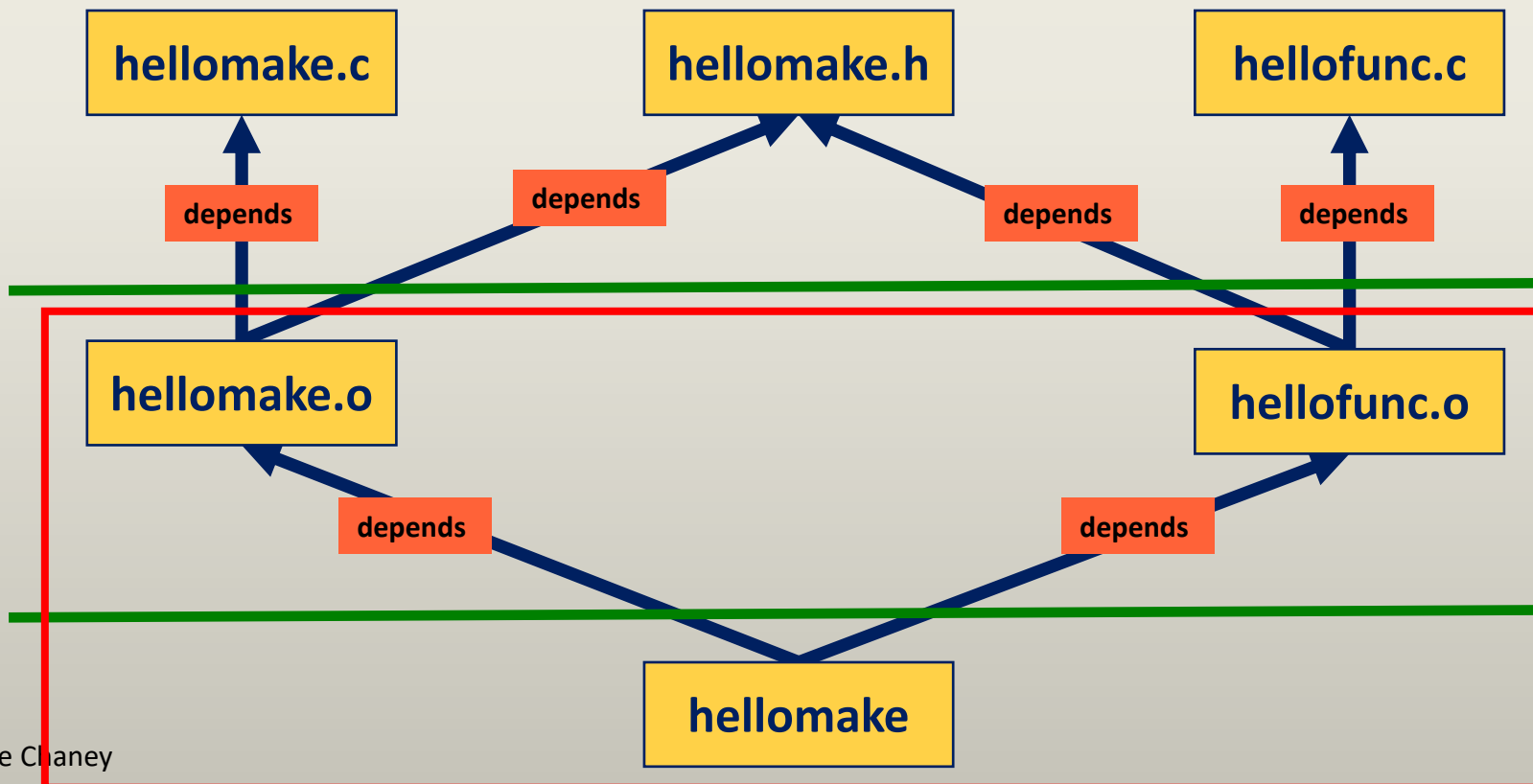
Anytime `hellofunc.c` or `hellomake.h` are newer than the `hellofunc.o` file, the `hellofunc.o` file must be rebuilt.

- The `hellomake` executable **depends on** the `hellomake.o` and `hellofunc.o` files.
- The `hellomake.o` and `hellofunc.o` files are **prerequisites** to the `hellomake` file.



Anytime `hellomake.o` or `hellofunc.o` are newer than the `hellomake` file, the `hellomake` file must be rebuilt.

- We can actually go a step further and say that the **hellomake** program actually depends on: `hellomake.o`, `hellofunc.o`, `hellomake.c`, `hellofunc.c`, and `hellomake.h`



Anytime `hellomake.o`, `hellofunc.o`, `hellomake.c`, `hellofunc.c`, or `hellomake.h` are newer than the `hellomake` file, the `hellomake` file must be rebuilt.

Identifying Prereqs in a Makefile

Target (what we want to build)

A single colon separates the target from the prerequisite files.

Prerequisites/Depends on

hellomake : hellomake.o hellofunc.o

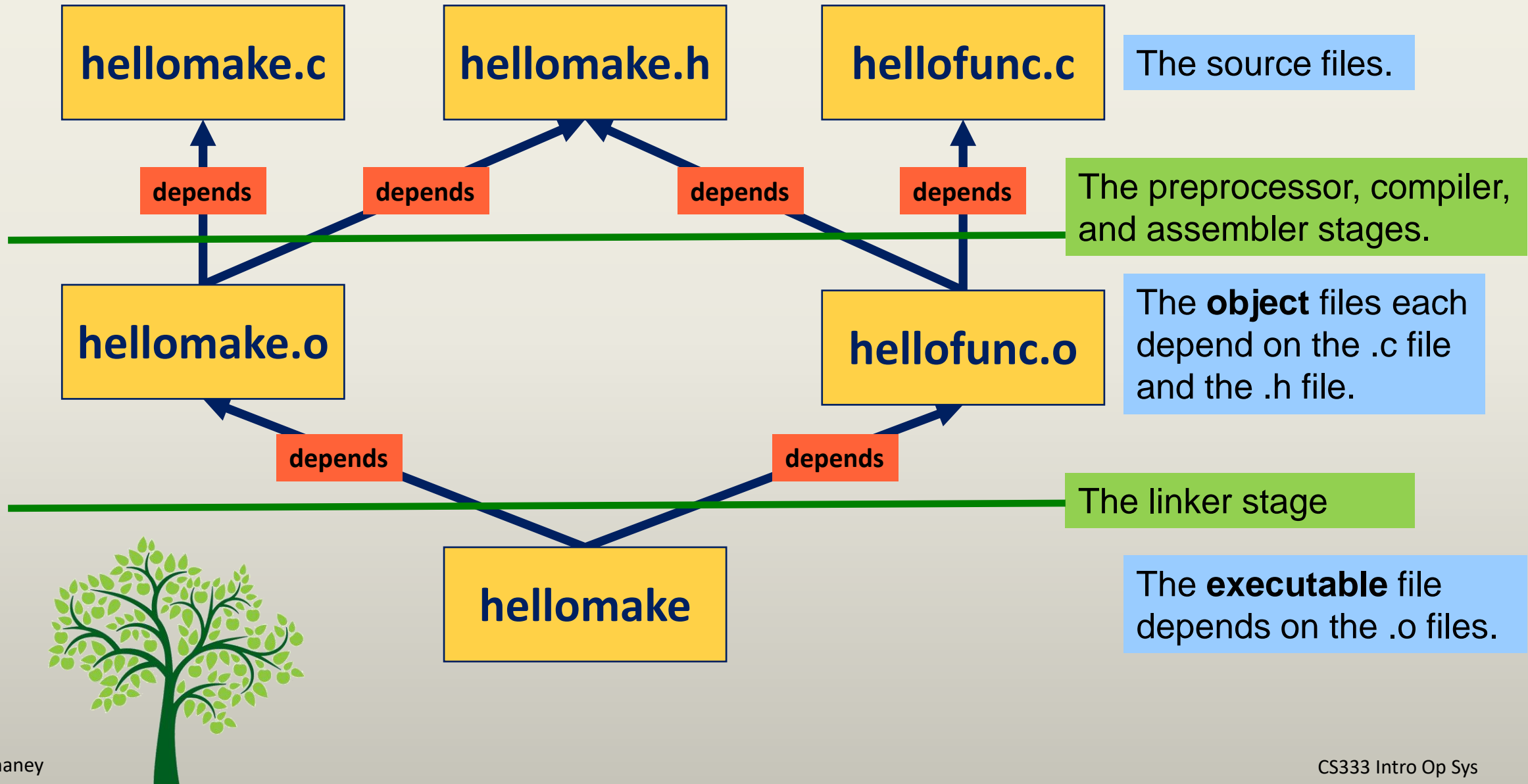
hellomake.o : hellomake.c hellomake.h

hellofunc.o : hellofunc.c hellomake.h

If any of the prerequisite files is **newer** than the target, the target must be rebuilt.



The File Dependency Tree



Rules



```
hellomake: hellomake.o hellofunc.o
gcc -o hellomake hellomake.o hellofunc.o
```

Target

Prerequisites

hellomake.o: hellomake.c hellomake.h

Command

gcc -c hellomake.c

```
hellofunc.o: hellofunc.c hellomake.h
gcc -c hellofunc.c
```

A **rule** is composed of the target, the prerequisites, and the command(s) used to build the target.

make is **VERY** finicky about
tab characters.



```
hellomake: hellomake.o hellofunc.o
```

```
[ ] gcc -o hellomake hellomake.o hellofunc.o
```

These **MUST**
be hard tab
characters!!!

```
hellomake.o: hellomake.c hellomake.h
```

```
[ ] gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
[ ] gcc -c hellofunc.c
```

Do **NOT** try and do indentation
before commands with spaces.

You **must** have a target in your `Makefile` called **clean**.
The `clean` target will delete a three different kinds of files.

1. The executable file/files
2. The object `.o` files
3. Any editor droppings files that are left around (`#*` files from `vi` and `*~` files from `emacs`).

A target without any prerequisites. This target has 2 names, `clean` and `cls`. Either can be used.

```
clean cls:  
rm -f hellomake *.o *~ \#*
```

Note the `-f` option for `rm`. Why?



Each `Makefile` has a **default target**. The default target is the **first target** listed in the `Makefile`. Typically, that is a target called **a11**. When you don't specifically identify a target for `make`, it will run the default target.

The `a11` target should build all the executables in the `Makefile`.

A rule without any commands. It does have Prerequisites though.

```
a11: hellomake
```

A `Makefile` can build more than 1 executable.

I really like to ask this on exams.



We'll go through
this simple
Makefile in detail.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
gcc -c hellofunc.c
```

```
clean cls:
```


```
rm -f *.o hellomake *~ \#*
```

Notice the command line
options used with gcc.




```
all: hellomake
```


```
hellomake: hellomake.o hellofunc.o
```

```
 gcc -o hellomake hellomake.o hellofunc.o
```


```
hellomake.o: hellomake.c hellomake.h
```

```
 gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
 gcc -c hellofunc.c
```

```
clean cls:
```

```
 rm -f *.o hellomake *~ \#*
```

These **MUST** be
tab characters!!!



Make sure
there is at least
1 blank line
between rules.
The line should
not have any
spaces or tabs
in it.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
gcc -c hellofunc.c
```

```
clean cls:
```

```
rm -f *.o hellomake *~ \#*
```

```
all: hellomake
```

The default target (`all`) is what runs when you just type `make`.

```
hellomake: hellomake.o hellofunc.o
    gcc -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.c hellomake.h
    gcc -c hellomake.c

hellofunc.o: hellofunc.c hellomake.h
    gcc -c hellofunc.c

clean cls:
    rm -f *.o hellomake *~ \#*
```

The rule to build the executable file `hellomake`.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o  
    gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h  
    gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h  
    gcc -c hellofunc.c
```

```
clean cls:  
    rm -f *.o hellomake *~ \#*
```

The targets and rules to build the object files from the `.c` files.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o  
gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h  
gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h  
gcc -c hellofunc.c
```

```
clean cls:  
rm -f *.o hellomake *~ \#*
```

The `hellofunc.o` file depends on the `hellofunc.c` file and the `hellomake.h` file.

The `hellomake.o` file depends on the `hellomake.c` file and the `hellomake.h` file.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
    gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
    gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
    gcc -c hellofunc.c
```

```
clean cls:
```

```
    rm -f *.o hellomake *~ \#*
```

Clean up editor
droppings with the
clean target.

This is the order in which you'd typically place the entries in the Makefile.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
    gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
    gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
    gcc -c hellofunc.c
```

```
clean cls:
```

```
    rm -f *.o hellomake *~ \#*
```

We'll step through this `Makefile` how it would be executed if we simply type `make` on the command line and the object files and executable file do not exist.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
    gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
    gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```


```
    gcc -c hellofunc.c
```

```
clean cls:
```

```
    rm -f *.o hellomake *~ \#*
```

We type `make` without any target, it will execute the default target, `all`.

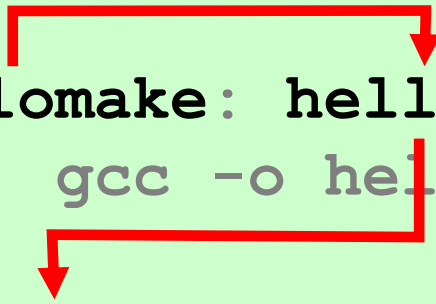
The `all` target will execute the `hellomake` target.


`all: hellomake`
`hellomake: hellomake.o hellofunc.o`
`gcc -o hellomake hellomake.o hellofunc.o``hellomake.o: hellomake.c hellomake.h`
`gcc -c hellomake.c``hellofunc.o: hellofunc.c hellomake.h`
`gcc -c hellofunc.c``clean cls:`
`rm -f *.o hellomake *~ \#*`

The `hellomake` rule will check the `hellomake.o` file, find it is out of date, and will run the `hellomake.o` target.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o  
           gcc -o hellomake hellomake.o hellofunc.o
```



```
hellomake.o: hellomake.c hellomake.h  
             gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h  
            gcc -c hellofunc.c
```

```
clean cls:  
    rm -f *.o hellomake *~ \#*
```

The `hellomake.o` rule will check the `hellomake.c` and `hellomake.h` files, finding they are newer than the target.

There is not a `hellomake.c` target, so the command for the `hellomake.o` rule will be executed, building the `hellomake.o` file.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
→ gcc -c hellomake.c
```



```
hellofunc.o: hellofunc.c hellomake.h
```

```
gcc -c hellofunc.c
```

```
clean cls:
```

```
rm -f *.o hellomake *~ \#*
```

The `hellomake` rule will check the `hellofunc.o` file, find it is out of date, and will run the `hellofunc.o` target.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
gcc -c hellofunc.c
```

```
clean cls:
```

```
rm -f *.o hellomake *~ \#*
```

The `hellofunc.o` rule will check the `hellofunc.c` and `hellomake.h` files, finding it is out of date.

There is not a `hellofunc.c` target, so the command for the `hellofunc.o` rule will be executed, building the `hellofunc.o` file.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

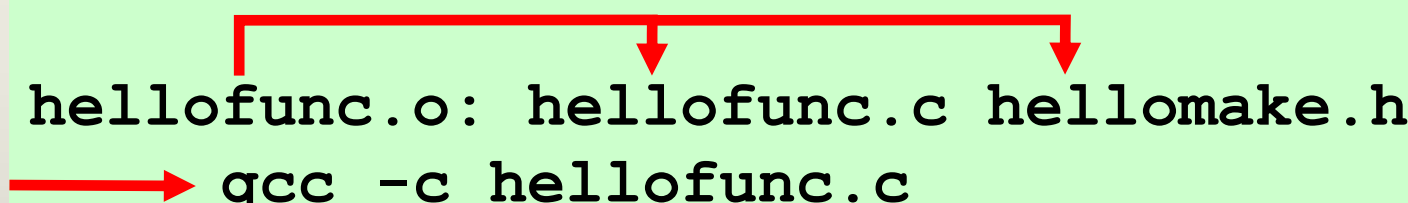
```
gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
→ gcc -c hellofunc.c
```



```
clean cls:
```

```
rm -f *.o hellomake *~ \#*
```

Now that the `hellomake.o` and `hellofunc.o` files are up to date, the command associated with `hellomake` will be run, producing the `hellomake` executable.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
→ gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
gcc -c hellofunc.c
```

```
clean cls:
```

```
rm -f *.o hellomake *~ \#*
```


The `hellomake` dependency and the `all` rule are now up to date. The `hellomake` executable has been built.

If you run `make` again right now, nothing will be rebuilt, as all targets are up to date.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
    gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
    gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
    gcc -c hellofunc.c
```

```
clean cls:
```

```
    rm -f *.o hellomake *~ \#*
```

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
gcc -o hellomake hellomake.o hellofunc.o
```


```
hellomake.o: hellomake.c hellomake.h
```

```
gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
gcc -c hellofunc.c
```

```
clean cls:
```



```
rm -f *.o hellomake *~ \#*
```

The executable, object files, and editor droppings will be deleted.

In order to run the target `clean`, you issue the command:
`make clean`

Since there are no dependencies, the command will simply be run.

- You put comments in your `Makefile` by using the `#` (octothorpe) character. Comments continue to the end of line.
- You can suppress showing the output from a command by putting a `@` as the first character in the command.
- **Don't do this. I will deduct points if you do.**

```
# This is a comment  
clean cls:
```

```
    @rm -f *.o hellomake *~ \#*
```

Don't do this!



You can define variables to use in your Makefile.

Creates 2 variables you can use in the Makefile.

```
CC = gcc -Wall -g  
PROG = hellomake
```

Yes, you can put spaces around the = operator in the assignment!

all: \$(PROG)

You **must** use braces **or** parentheses around variable names.

\$(PROG): hellomake.o hellofunc.o

\$(CC) -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.c hellomake.h

\$(CC) -c hellomake.c

hellofunc.o: hellofunc.c hellomake.h

\$(CC) -c hellofunc.c

clean cls:

rm -f *.o hellomake *~ \#*



var

Special make variables:

- `$@` - The name of the target of the rule (`hellomake.o`).
- `$<` - The name of the first prerequisite (`hellomake.c`).
- `^` - The names of all the prerequisites (`hellomake.c hellomake.h`).
- `$?` - The names of all prerequisites that are newer than the target.

This is NOT the same `$?` you use in your bash shell scripts.



In addition to just being part of the **cool crowd**, use of variables in your `Makefile` makes it easier to copy and reuse for other projects.

```
CC = gcc -Wall -g
PROG = hellomake
```

```
all: $(PROG)
```

```
$(PROG): hellomake.o hellofunc.o
```

```
$(CC) -o $@ $^
```

Special variables

The name of the target. The names of all the prerequisites.

```
hellomake.o: hellomake.c hellomake.h
```

```
$(CC) -c $<
```

Special variable

The name of the first prerequisite.

```
hellofunc.o: hellofunc.c hellomake.h
```

```
$(CC) -c $<
```

The name of the first prerequisite.

```
clean cls:
```

```
rm -f *.o hellomake *~ \#*
```



You can put other targets in your Makefile. The lazy among us might have variables and targets that look like these:

```
TAR_FILE = Lab3_${LOGNAME}.tar.gz
```

An environment variable.

```
tar: clean  
    rm -f $(TAR_FILE)  
    tar cvfa $(TAR_FILE) *. [ch] ?akefile
```

Notice that I use braces around environment variables and parenthesis around `make` variables. This is just my standard, not required.



Ever have your code “disappear” just before the due date? Revision control can help.

```
ci:
    if [ ! -d RCS ] ; then mkdir RCS; fi
    -ci -t-none -m"lazy-chicken" -l *.[ch] ?akefile
```

OR

```
git checkin chicken poultry:
    if [ ! -d .git ] ; then git init; fi
    git add *.[ch] ?akefile
    git commit -m "my lazy git commit comment"
```