

OPerating Systems Laboratory *OPSLab*

PART 4.1

Programming in the UNIX Environment (basic ingredients)

Prof. Marco Autili
University of L'Aquila

C/C++ Compiler

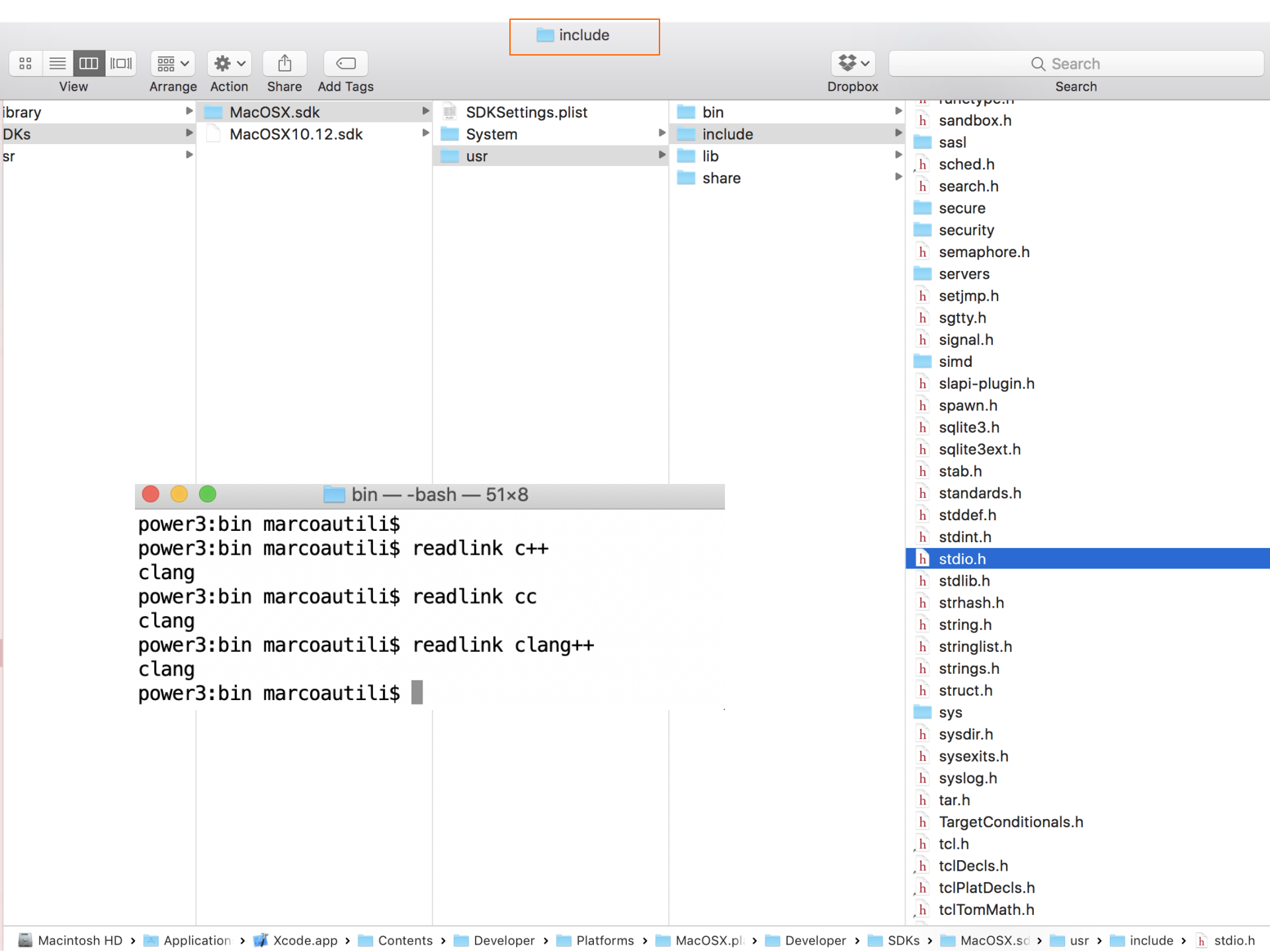
- `gcc --version` or `cc --version` or `g++ --version` or `clang++ --version`
- Historically, `cc` is the C compiler
 - On systems with the GNU C compilation system, the C compiler is `gcc` and `cc` is usually linked to `gcc` (`clang` in my case)

```
bin — -bash — 120x27
power3:~ marcoautili$
power3:~ marcoautili$ gcc --version
Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr --with-gxx-include-dir=/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.14.sdk/usr/include/c++/4.2.1
Apple LLVM version 10.0.0 (clang-1000.11.45.5)
Target: x86_64-apple-darwin18.2.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
power3:~ marcoautili$
power3:~ marcoautili$ cd /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
power3:bin marcoautili$
power3:bin marcoautili$ ls -la
total 195896
drwxr-xr-x  81 root  wheel   2592 Oct 31 10:54 .
drwxr-xr-x   8 root  wheel   256 Oct 31 10:54 ..
-rwxr-xr-x   1 root  wheel 33920 Oct 20 03:23 ar
-rwxr-xr-x   1 root  wheel 28000 Oct 20 03:23 as
-rwxr-xr-x   1 root  wheel 18176 Oct 20 03:23 asa
-rwxr-xr-x   1 root  wheel 212208 Oct 20 03:23 bison
-rwxr-xr-x   1 root  wheel 150048 Oct 20 03:23 bitcode_strip
lrwxr-xr-x   1 root  wheel    5 Sep 21 2017 c++ -> clang ←
-rwxr-xr-x   1 root  wheel 23152 Oct 20 03:23 c89
-rwxr-xr-x   1 root  wheel 23248 Oct 20 03:23 c99
lrwxr-xr-x   1 root  wheel    5 Sep 21 2017 cc -> clang ←
-rwxr-xr-x   1 root  wheel 78705232 Oct 20 03:23 clang ←
lrwxr-xr-x   1 root  wheel    5 Sep 21 2017 clang++ -> clang ←
-rwxr-xr-x   1 root  wheel 120064 Oct 20 03:23 cmpdylib
```

In my MacOS machine, "header files" are located here (as part of the Xcode installation)

(see next slide)

Try also `cd /usr/bin` followed by `ls -la`



Clang compiler

- **Clang** is a **C**, **C++**, and **Objective-C** compiler which encompasses **preprocessing**, **parsing**, **optimization**, **code generation**, **assembly**, and **linking**
- The clang executable is actually a **small driver** which controls the overall execution of other tools such as the compiler, assembler and linker. **Typically, you do not need to interact with the driver, but you transparently use it to run the other tools**
- Depending on which high-level mode setting is passed, Clang will stop before doing a full link
- While Clang is highly integrated, it is important to understand the stages of compilation, to understand how to invoke it

These stages are →

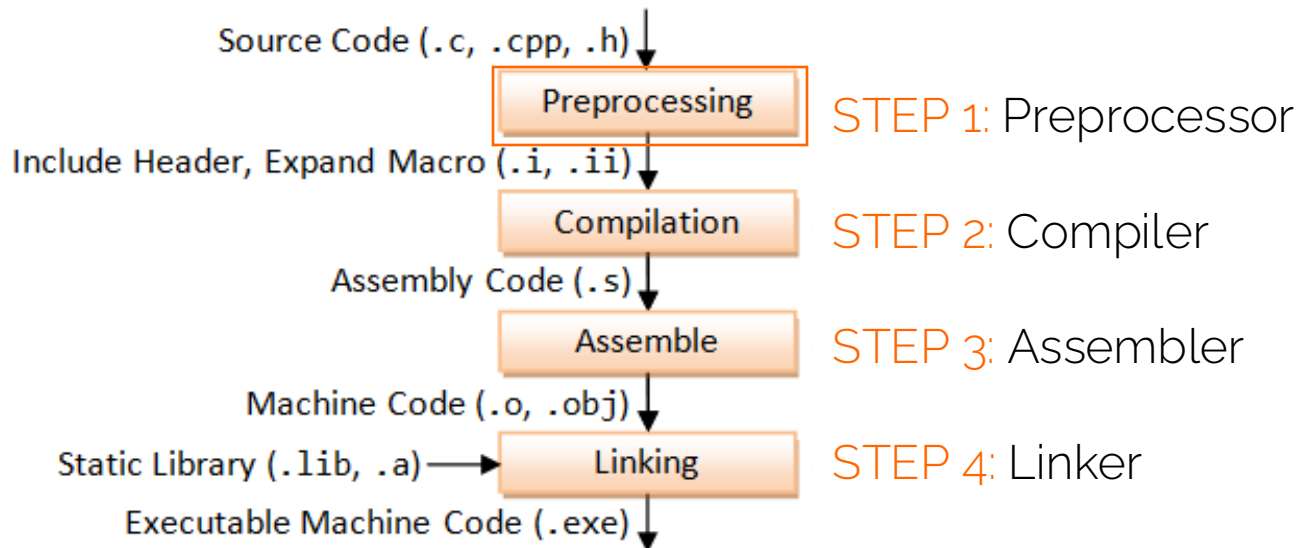
<https://it.wikipedia.org/wiki/Objective-C>

Preprocessing

Preprocessing

This stage handles tokenization of the input source file, macro expansion, #include expansion and handling of other preprocessor directives

The output of this stage is typically a ".i" (for C), ".ii" (for C++), ".mi" (for Objective-C), or ".mii" (for Objective-C++) file



Preprocessing

See the examples into the folder "PART 4.1 – Compilation"

```
power3:compilation marcoautili$
```

```
power3:compilation marcoautili$ cat define.c
```

```
#define TEST "Hello, World!" ←
```

```
const char str[] = TEST; ←
```

```
power3:compilation marcoautili$
```

```
power3:compilation marcoautili$ gcc -E define.c
```

```
# 1 "define.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 361 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "define.c" 2
```

```
const char str[] = "Hello, World!"; ←
power3:compilation marcoautili$ █
```

The **-E** option causes gcc to:

1. run the preprocessor
2. display the expanded output
3. exit (**without compiling** the resulting source code)

The preprocessor inserts lines recording the source file and line numbers in the form, e.g., **# line-number "source-file"**, to aid in debugging and allow the compiler to issue error messages referring to this information

The value of the **macro TEST** is substituted directly into the output
(→ **macro expansion**)

Preprocessing

```
power3:compilation marcoautili$ cat hello.c
#include <stdio.h>

int
main (void)
{
    printf ("Hello, world!\n");
    return 0;
}

power3:compilation marcoautili$ gcc -E hello.c
# 1 "hello.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 361 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "hello.c" 2

# 1 "/Applications/Xcode.app/Contents/Developer/F
```

- The ability to see the preprocessed source files can be useful for examining the effect of system **header files**, and finding declarations of system functions
- The program hello.c includes the header file '**stdio.h**' to obtain the declaration of the function **printf**
- the produced output is quite long...

Try yourself

The preprocessed system header files usually generate a lot of output. This can be redirected to a file, or saved more conveniently using the **gcc -save-temps** option

```
power3:compilation marcoautili$ gcc -c -save-temps hello.c
power3:compilation marcoautili$ ls
define.c      hello.bc      hello.c      hello.i      hello.o      hello.s
power3:compilation marcoautili$
```

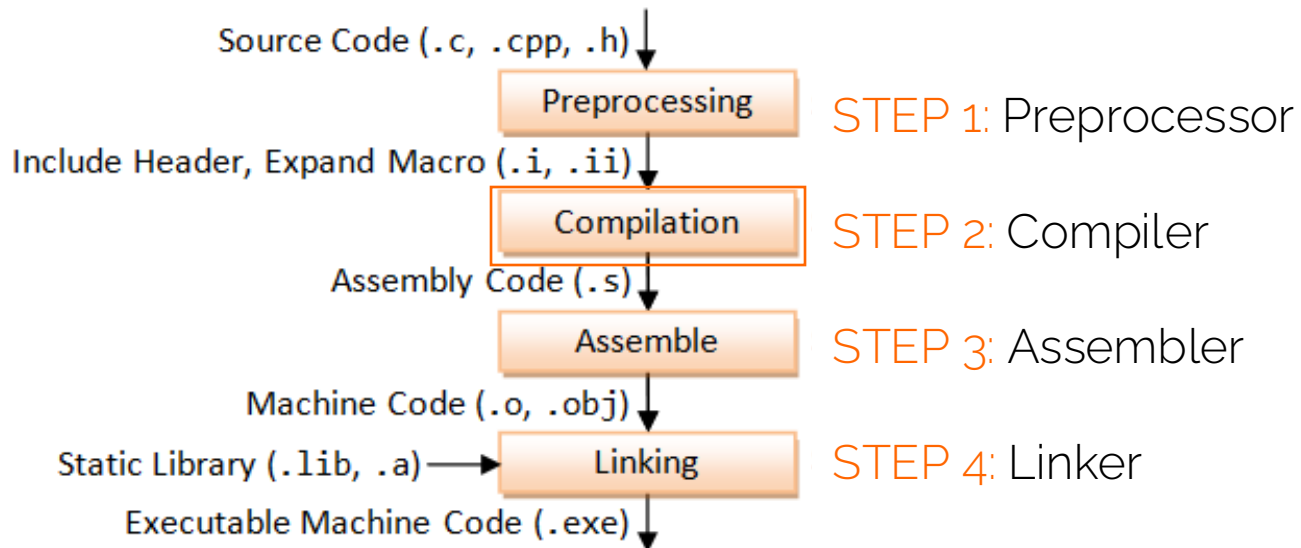
The **-c** option runs the phases:

- preprocessing
- compilation
- assemble

Compilation (cont'd)

Parsing and Semantic Analysis

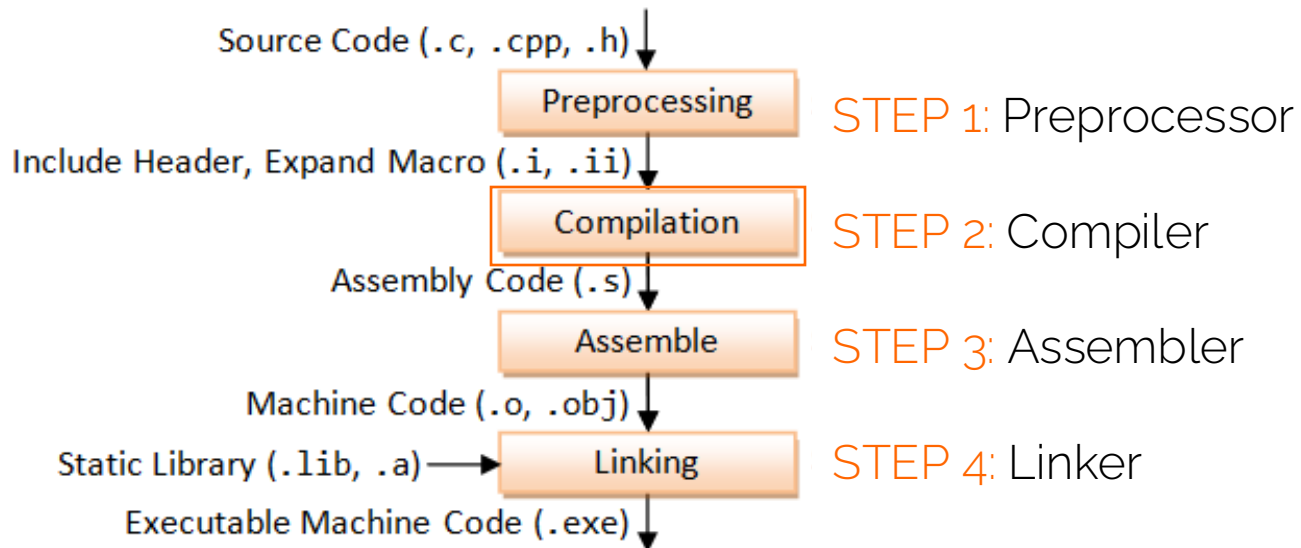
- This stage parses the file, **translating preprocessor tokens** into a **parse tree**
- Once in the form of a parse tree, it applies **semantic analysis** to compute **types** for expressions as well and determine whether the **code is well formed**
- This stage is responsible for generating most of the **compiler warnings** as well as **parsing errors**
- **The output** of this stage is an "**Abstract Syntax Tree**" (AST)



Compilation (cont'd)

Code Generation and Optimization

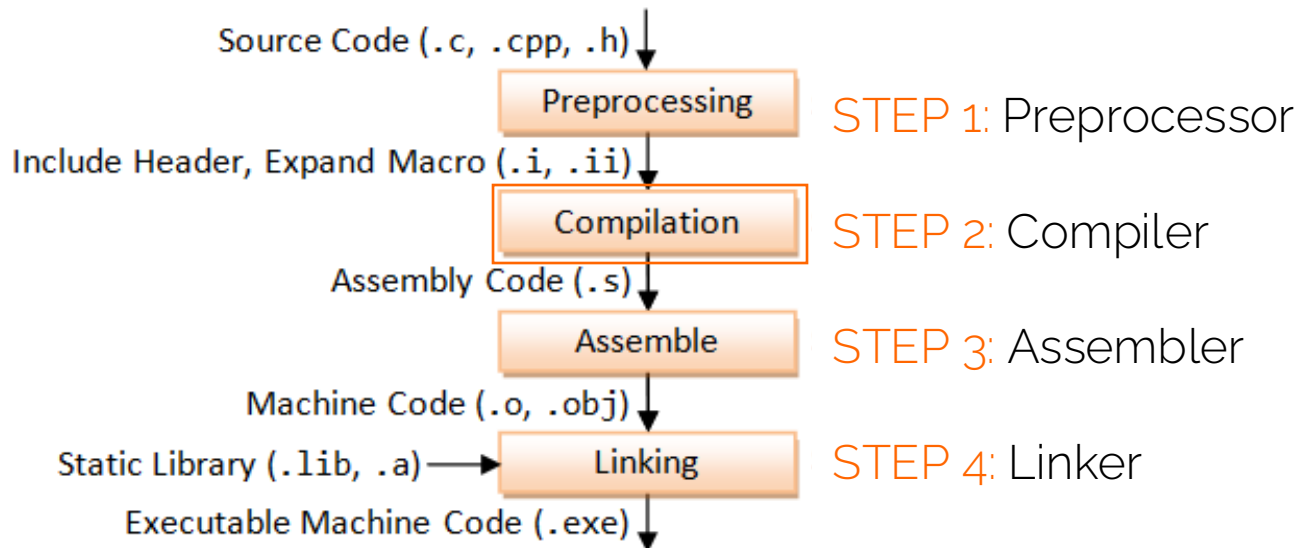
- This stage translates the AST into low-level intermediate code (known as "LLVM IR") and ultimately to assembly code
- This phase is also responsible for optimizing the generated code and handling target-specific code generation...



Compilation (cont'd)

Code Generation and Optimization

- The output of this stage is typically called a ".s" or "assembly" file
- Clang also supports the use of an integrated assembler, in which the code generator produces object files directly (this avoids the overhead of generating the ".s" file and of calling the target assembler)



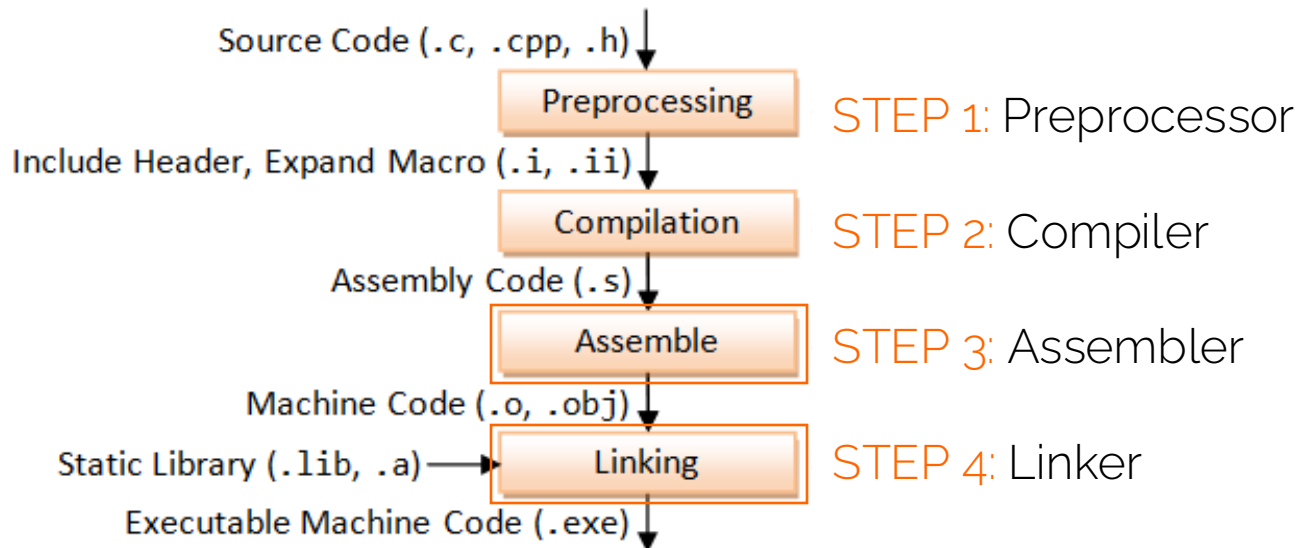
Assemble and linking

Assembler

- This stage runs the target assembler to translate the output of the compiler into a target object file. The output of this stage is typically called a ".o" file or "object" file

Linker

- This stage runs the target linker to merge multiple object files into an executable or dynamic library. The output of this stage is typically called an "a.out", ".dylib" or ".so" file ("so" stands for shared object)



Clang Static Analyzer

- The **Clang Static Analyzer** is a tool that scans source code to try to find bugs through code analysis
- This tool uses many parts of Clang and is built into the same driver
- See <http://clang-analyzer.llvm.org> for more details on how to use the static analyzer

Compilation process notes

Remember that

- In Bash or Bourne shell, the default PATH does not include the current working directory. Hence, you may need to prefix the working directory (`./`) to the command
 - **Windows** include the current directory in the PATH automatically
 - **Unix** systems do not - you need to include the current directory explicitly in the PATH
- You may need to include the file extension, i.e., `./a.exe`
- In some Unix systems, the output file could be `a.out` or simply `a`
- You need to assign executable file-mode (x) to the executable file `a.out`
 - e.g., type `chmod a+x filename` add executable file-mode to all users

* https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

Introducing the “make” utility

The “make” utility in Unix is one of the original tools designed by S. I. Fieldman of AT&T Bell labs – 1977

- What is “make”?
 - It is designed to allow programmers to easily and efficiently compile large complex programs with many components
- As an alternative to make, you may place the commands to compile a program in, e.g., a Unix script but this will cause ALL modules to be compiled every time
- The “make” utility allows us to only compile those that have changed and the modules that depend upon them
- We will refer to C programs, since they are part of our course, but you can use make with any programming language whose compiler can be run with a shell command
- Indeed, make is not limited to programs
 - You can use it to support any task where some files must be updated automatically from others whenever the others change

* Unix Makefile notes from David A. Gaitros (Florida State University)
<https://www.cs.fsu.edu/departement/faculty/gaitrosd/>

How does it work?

In Unix, when you type the command “make” the operating system looks for a file called either “**makefile**” or “**Makefile**”

- There are exceptions to this, but we will assume that you will always have the “makefile” or “Makefile” file resident in the directory of where your program resides

This file contains a **series of directives** that tell the “make” utility **how to** compile your program and in **what order**

Each file can be associated with a list of other files by which it is dependent

- This is called **a dependency line**
- If any of the associated files have been recently modified, the make utility will **execute a directive command just below the dependency line**

The “make” utility is recursive

- For instance, if a very “low-level” utility is the only thing changed, it could cause all of the modules within a program to be re-compiled
- After the utility finishes the file, it goes through and checks all of the dependencies again to make sure all are up to date

Makefile rules

- A simple makefile consists of "rules" with the following shape

targets ... : prerequisites ...

recipe /command

...

targets ... : prerequisites ...

recipe /command

...

- Targets are usually names of files that are generated by the *recipe/command*; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as "clean"
 - (More details on Phony Targets at https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html.)
- A *prerequisite* is a file that is used as input to create the target; a target often depends on several files
- A *recipe/command* is an action that is carried out. A rule may have more than one command, each on its own line. Note that you need to put a *tab character* at the beginning of every command line and an *empty line* between one rule and another

[* https://www.gnu.org/software/make/manual/make.html#Introduction](https://www.gnu.org/software/make/manual/make.html#Introduction)

Makefile rules

- Usually, a command is placed in a rule with prerequisites and serves to create a target file whenever any of the prerequisites change
- However, for example, a rule containing the delete command associated with the target “*clean*” does not have prerequisites
- A rule, then, explains how and when to (*re*)*make* certain files which are the targets of the particular rule
- Make carries out the commands on the prerequisites to create or update the target. A rule can also explain how and when to carry out an action

Simple Example

```
hello: main.o factorial.o hello.o
```

```
    g++ main.o factorial.o hello.o -o hello
```

```
main.o: main.cpp
```

```
    g++ -c main.cpp
```

Remember that the `-c` option tells `gcc` to only run the `preprocess`, `compile`, and `assemble` steps, hence producing only up to the Machine Code `.o`

```
factorial.o: factorial.cpp
```

```
    g++ -c factorial.cpp
```

```
hello.o: hello.cpp
```

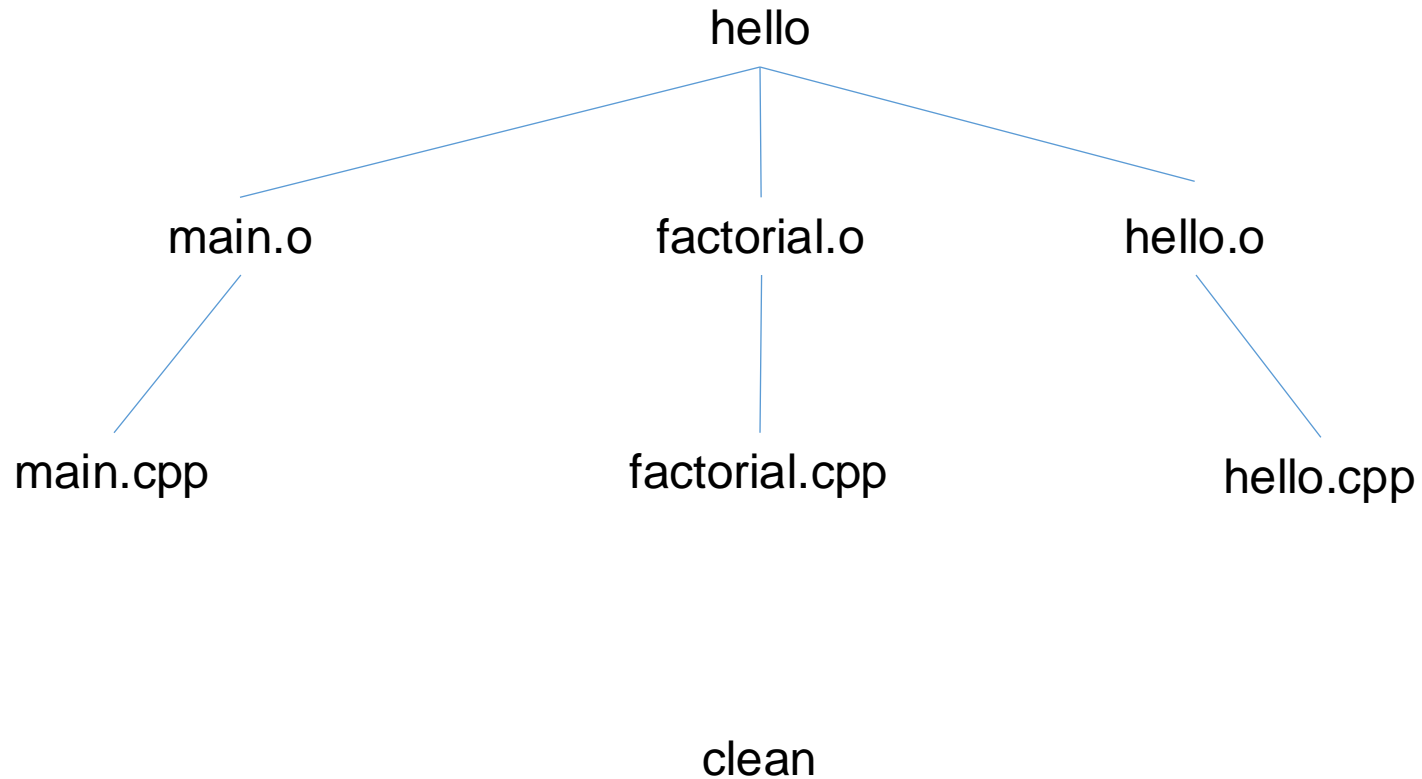
```
    g++ -c hello.cpp
```

```
clean:
```

```
    rm -rf *.o hello
```

- The option `-f` is used to `remove the files without prompting for confirmation`, regardless of the file's permissions
- If the file does not exist, do not display a diagnostic message or modify the exit status to reflect an error
- The `-f` option overrides any previous `-i` options

Dependencies tree



Components of a Makefile

- Comments
- Rules
- Dependency Lines
- Shell Lines
- Macros
- Inference Rules

Comments

- A comment is indicated by the character “#” so that all text that appears after it will be ignored by the make utility until the end of line is detected
- Comments can start anywhere
 - There are more complex comment styles that involve continuation characters but, I suggest to start each new comment line with an # and avoid the more advanced features for now.
- Example

```
#  
  
# This is a comment  
  
projecte.exe : main.obj io.obj # this is also a comment
```

Rules

- Rules tell make **when** and **how** to make a file
- The format is as follows
 - A rule must have a dependency line and may have an action or shell line after it
 - The action line is executed if the dependency line is out of date

```
hello.o: hello.cpp
    g++ -c hello.cpp
```

- This shows **hello.o** as a “module” that **requires hello.cpp** as source code
- If the last modified **date of hello.cpp is newer than hello.o**, then the next line (shell line) is executed
- Together, **these two lines form a rule**

Dependency Lines

- The lines with a ":" are called **dependency lines**
 - targets are on the left
 - the (re)sources (prerequisites) needed to make the targets are on the right
- The **make process is recursive** in that it will check all dependency lines to make sure targets are not out of date before completing the build process
- It is **important to place all of the dependencies in a descending order** in the file
- Within a dependency line, more than one **target file may have the same prerequisites**

For instance, suppose that two files need a file called `bitvect.h`, the dependency line will be

```
main.obj this.obj: bitvect.h
```

Shell Lines

- The indented lines that follow each dependency line are called **shell lines**
- Shell lines **tell *make* how to build the target**
- A **target can have more than one shell line**
- Each shell line must be preceded by a tab
- After each shell line is executed, **make checks to see if it was completed without error**
- After each shell line is executed, Make checks the shell line **exit status**
 - Shell lines that returning an exit status of **zero** (0) means without error and **non-zero** if there is an error
 - The first shell line that **returns an exit status of non-zero** will cause the **make utility to stop and display an error**
 - You can override this by placing a **"-"** in front of the shell command, but I would not do this
 - `gcc -o my my.o mylib.o`

* Unix Makefile notes from David A. Gaitros (Florida State University)
<https://www.cs.fsu.edu/departments/faculty/gaitrosd/>

Macros

- A macro can be seen as a shorthand or alias used in the makefile
- Inside the file, to expand a macro, you have to place the "string" inside of `$()`
- The whole string is expanded during execution of the make utility

Examples of macros

- `HOME = /home/courses/cop4530/spring02`
- `CPP = $(HOME)/cpp`
- `TCPP = $(HOME)/tcpp`
- `PROJ = .`
- `INCL = -I $(PROJ) -I$(CPP) -I$(TCPP)`
- You can also define macros directly at the command line such as
 - `make DIR = /home/faculty/whalley/public_html/cop5622proj/lib2`
and this would take precedence over the one in the file

* Unix Makefile notes from David A. Gaitros (Florida State University)
<https://www.cs.fsu.edu/departments/faculty/gaitrosd/>

Inference Rules

- Inference rules are a method of generalizing the build process
- In essence, it is a wild card notation
- The "%" is used to indicate a wild card

Example

```
% .obj : % .c  
$(CC) $(FLAGS) -c $(.SOURCE)
```

- All .obj files have dependencies on all %c files of the same name

* Unix Makefile notes from David A. Gaitros (Florida State University)
<https://www.cs.fsu.edu/departments/faculty/gaitrosd/>

Makefile

hellomake.c	hellofunc.c	hellomake.h
<pre>#include <hellomake.h> int main() { // call a function in another file myPrintHelloMake(); return(0); }</pre>	<pre>#include <stdio.h> #include <hellomake.h> void myPrintHelloMake(void) { printf("Hello makefiles!\n"); return; }</pre>	<pre>/* example include file */ void myPrintHelloMake(void);</pre>

Normally, you would compile this collection of code by executing the following command

```
gcc -o hellomake hellomake.c hellofunc.c -I.
```

- This **compiles** the **two .c files** and names the resulting executable **hellomake**
- The **-I.** is specified so that gcc will look in the current directory (**.**) and will include **hellomake.h**
- The **-o <file>** option tells **gcc** to write the **output** to **<file>**

Without a **makefile**, the typical approach to the test/modify/debug cycle **is to use the up arrow in a terminal to go back to your last compile command** so you do not have to type it each time, especially once you've added a few more .c files to the mix

Makefile

Problem #1

- If you lose the compile command or switch computers you have to retype it from scratch, which is inefficient at best
- Of course, you may put it in a script, but ...

Problem #2

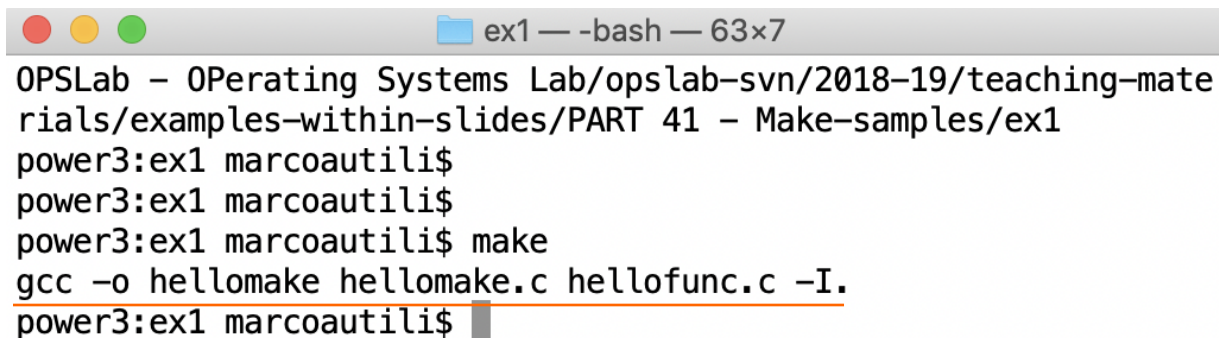
- ... if you are only making changes to one .c file, recompiling all of them every time is also time-consuming and inefficient

Makefile (example 1)

hellomake: hellomake.c hellofunc.c

gcc -o hellomake hellomake.c hellofunc.c -I.

- Make with no arguments executes the first rule in the file
- By putting the list of .c files on which the command depends on the first line after the :, make knows that the rule hellomake needs to be executed if any of those files change
 1. Problem #1 is solved
 2. However, the system is still not being efficient in terms of (re-)compiling only the strictly needed files



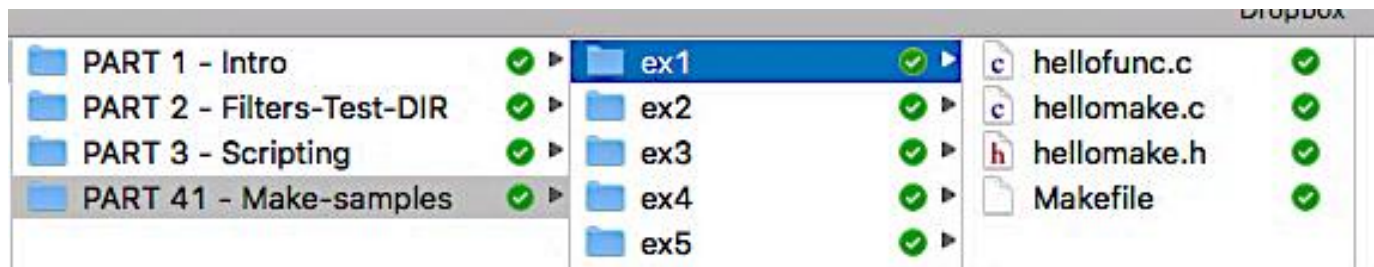
A terminal window titled "ex1 — -bash — 63x7" showing the execution of the Makefile rule. The window title bar has three colored circles (red, yellow, green) on the left. The terminal text is as follows:

```
OPSLab - OPerating Systems Lab/opslab-svn/2018-19/teaching-mate
rials/examples-within-slides/PART 41 - Make-samples/ex1
power3:ex1 marcoautili$
power3:ex1 marcoautili$
power3:ex1 marcoautili$ make
gcc -o hellomake hellomake.c hellofunc.c -I.
power3:ex1 marcoautili$
```

Makefile (example 1)

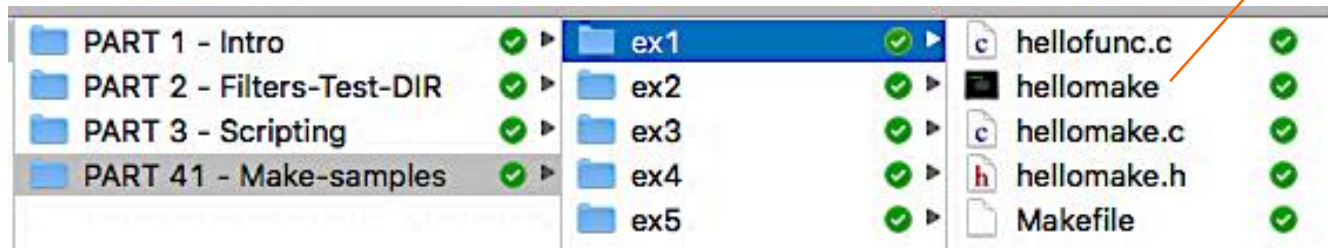
hellomake: hellomake.c hellofunc.c

gcc -o hellomake hellomake.c hellofunc.c -I.



```
OPSLab - OPerating Systems Lab/opslab-svn/2018-19/teaching-materials/examples-within-slides/PART 41 - Make-samples/ex1
power3:ex1 marcoautili$
power3:ex1 marcoautili$
power3:ex1 marcoautili$ make
gcc -o hellomake hellomake.c hellofunc.c -I.
power3:ex1 marcoautili$
```

Only the executable
has been created



Makefile (example 2)

```
CC=gcc
CFLAGS=-I.
hellomake: hellomake.o hellofunc.o
    $(CC) -o hellomake hellomake.o hellofunc.o $(CFLAGS)
```

- By putting the object files (hellomake.o and hellofunc.o) in the dependency list and in the rule, **make** knows (see next slide) it must:

- first, compile the .c versions individually
- and then, build the executable hellomake

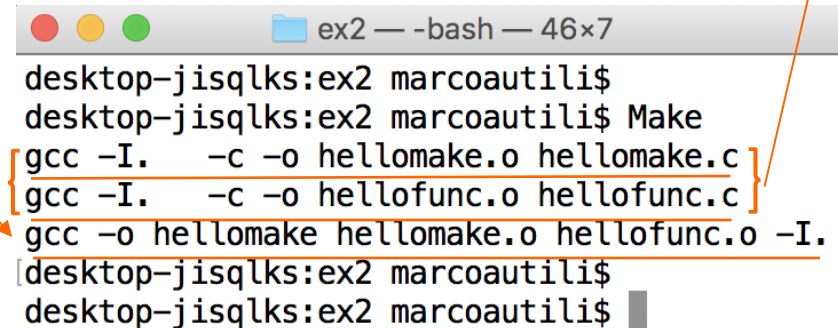
These two shell lines are **implicit**:
we did not specify them!

... next slide

However, there is one thing missing:

the dependency on the include files! ☹

- If you change **hellomake.h**, **make** would not recompile the .c files, even though they need to be
- In order to fix this, we need to tell make that all .c files depend on certain .h files
(ex. 3 will fix this issue)



```
desktop-jisqlks:ex2 marcoautili$
desktop-jisqlks:ex2 marcoautili$ Make
gcc -I. -c -o hellomake.o hellomake.c
gcc -I. -c -o hellofunc.o hellofunc.c
gcc -o hellomake hellomake.o hellofunc.o -I.
desktop-jisqlks:ex2 marcoautili$
desktop-jisqlks:ex2 marcoautili$
```

Remember that the **-c** option tells **gcc** to only run the **preprocess**, **compile**, and **assemble** steps, hence producing only up to the Machine Code **.o**

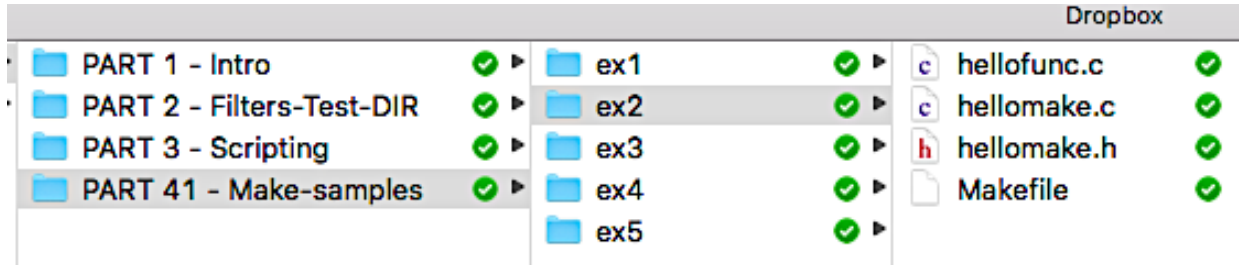
Makefile (example 2)

Letting make Deduce the Recipes

- It is not necessary to spell out the recipes for compiling the individual C source files, because make can figure them out:
 - it has an implicit rule for updating a '.o' file from a correspondingly named '.c' file using a 'cc -c' command
- That is why (in previous slide) make used the recipe `'gcc -l. -c -o hellomake.o hellomake.c'` to compile `hellomake.c` into `hellomake.o`. We can therefore omit the recipes from the rules for the object files
 - see Using Implicit Rules - <https://www.gnu.org/software/make/manual/make.html#Implicit-Rules>
- When a '.c' file is used automatically in this way, it is also automatically added to the list of prerequisites. We can therefore omit the '.c' files from the prerequisites

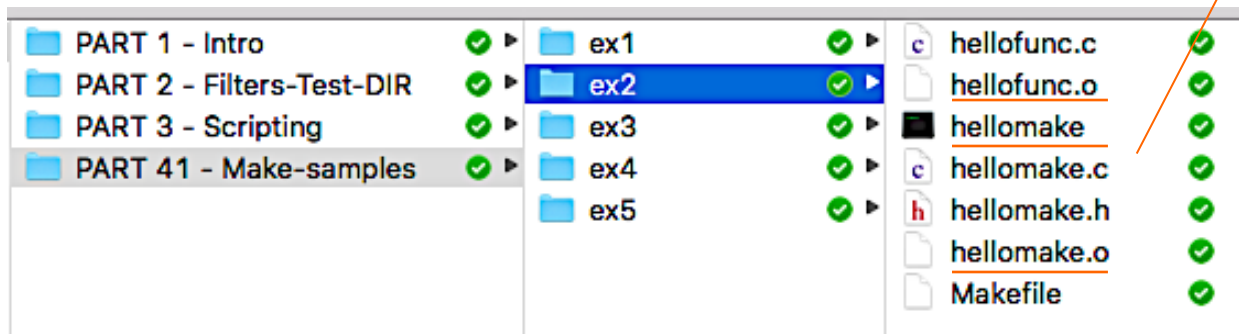
<https://www.gnu.org/software/make/manual/make.html#make-Deduces>

Makefile (example 2)

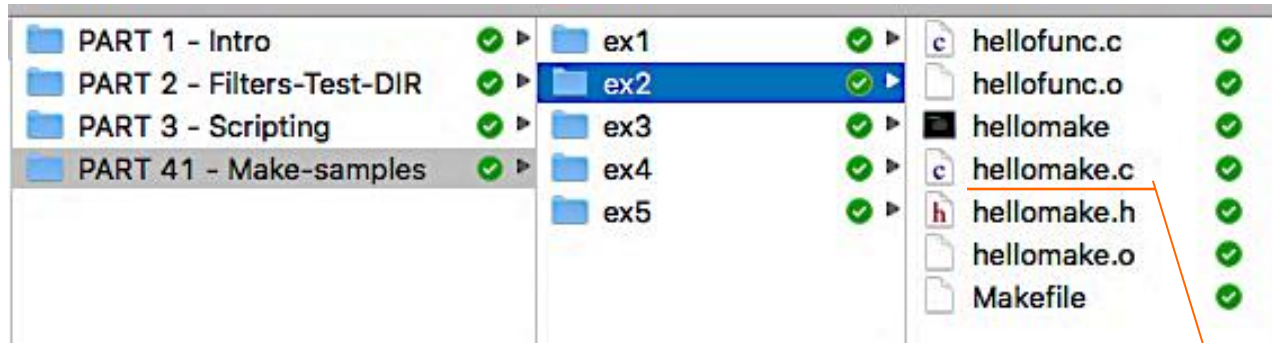


```
desktop-jisqlks:ex2 marcoautili$  
desktop-jisqlks:ex2 marcoautili$ Make  
gcc -I. -c -o hellomake.o hellomake.c  
gcc -I. -c -o hellofunc.o hellofunc.c  
gcc -o hellomake hellomake.o hellofunc.o -I.  
desktop-jisqlks:ex2 marcoautili$  
desktop-jisqlks:ex2 marcoautili$
```

- The .o files have been created through the first two (implicit) shell lines
- The executable has been created by the third one



Makefile (example 2)

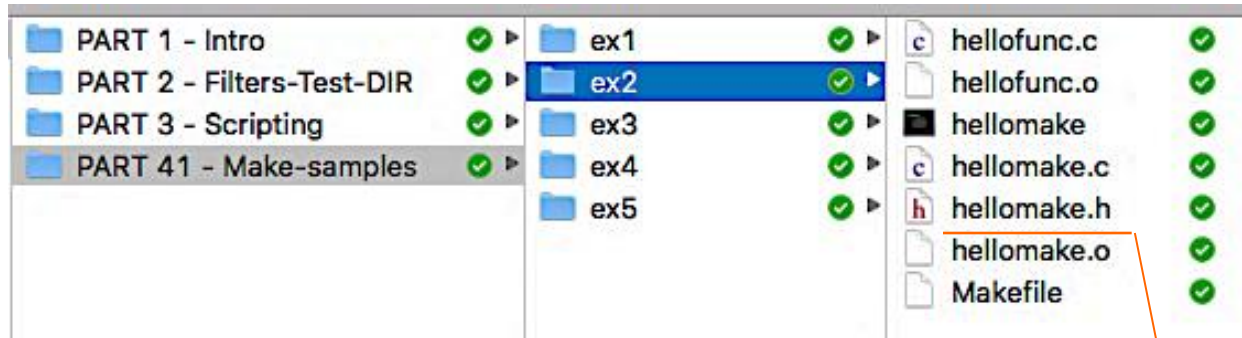


```
valintina-vaio:ex2 marcoautili$  
valintina-vaio:ex2 marcoautili$  
valintina-vaio:ex2 marcoautili$  
[valintina-vaio:ex2 marcoautili$ make  
[gcc -I. -c -o hellomake.o hellomake.c}  
gcc -o hellomake hellomake.o hellofunc.o -I.  
valintina-vaio:ex2 marcoautili$
```

If you modify **only** hellomake.c,
only it will be (**implicitly**) recompiled together
with the executable
Thus, **Problem #2** has also been solved

However

Makefile (example 2)



```
ex2 — -bash — 71x5
[valintina-vaio:ex2 marcoautili$
[valintina-vaio:ex2 marcoautili$
[valintina-vaio:ex2 marcoautili$ make
make: 'hellomake' is up to date.
valintina-vaio:ex2 marcoautili$
```

If you modify only hellomake.h,
make does not recompile anything ☹

The next example

- solves this problem by adding a dependency on the file .h, and
- generalizes the make file

Makefile (example 3)

CC=gcc

CFLAGS=-I.

DEPS = hellomake.h

%.o: %.c \$(DEPS)

\$(CC) -c -o \$@ \$< \$(CFLAGS)

It will be resolved in

gcc -c -o hellomake.o hellomake.c -I.

gcc -c -o hellofunc.o hellofunc.c -I.

as before, but now explicitly!

hellomake: hellomake.o hellofunc.o

gcc -o hellomake hellomake.o hellofunc.o \$(CFLAGS)

- The macro **DEPS** specifies the **.h** files on which the **.c** files depend
- Each **.o** file depends upon its **.c** version and the **.h** files included in the **DEPS** macro
- The rule then says that to generate the **.o** file, make needs to compile the **.c** file.
 - **-o \$@** is the name of the target being generated. It says to put the output of the compilation of the **.c** files on the right of **:** in same named (corresponding) files on the left of **:**
 - **\$<** is the first prerequisite in the dependencies line

Makefile (example 3)

```
ex3 — -bash — 56x18
[valintina-vaio:ex3 marcoautili$
[valintina-vaio:ex3 marcoautili$
[valintina-vaio:ex3 marcoautili$ make
gcc -c -o hellomake.o hellomake.c -I.
gcc -c -o hellofunc.o hellofunc.c -I.
gcc -o hellomake hellomake.o hellofunc.o -I.
[valintina-vaio:ex3 marcoautili$
[valintina-vaio:ex3 marcoautili$
[valintina-vaio:ex3 marcoautili$ make
make: `hellomake' is up to date.
[valintina-vaio:ex3 marcoautili$
[valintina-vaio:ex3 marcoautili$
[valintina-vaio:ex3 marcoautili$ make
gcc -c -o hellomake.o hellomake.c -I.
gcc -c -o hellofunc.o hellofunc.c -I.
gcc -o hellomake hellomake.o hellofunc.o -I.
[valintina-vaio:ex3 marcoautili$
[valintina-vaio:ex3 marcoautili$
```

If "here" you modify **only**
hellomake.h,
all the files will be recompiled

Thus, **the problem with the .h file**
has also been solved 😊

Makefile (example 4)

As a final simplification, all of the **include files** should be listed as part of the macro **DEPS**, and all of the **object files** should be listed as part of the macro **OBJ**

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h
OBJ = hellomake.o hellofunc.o

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

hellomake: $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS)
```

```
desktop-jisqlks:ex4 marcoautili$
desktop-jisqlks:ex4 marcoautili$ Make
gcc -c -o hellomake.o hellomake.c -I.
gcc -c -o hellofunc.o hellofunc.c -I.
gcc -o hellomake hellomake.o hellofunc.o -I.
desktop-jisqlks:ex4 marcoautili$
desktop-jisqlks:ex4 marcoautili$
desktop-jisqlks:ex4 marcoautili$
```

$\$^$ is the entire right side of the ":"
i.e., $\$^$ evaluates to $\$(OBJ)$

The entire line will be resolved in
`gcc -o hellomake hellomake.o hellofunc.o -I.`

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

$hellomake: \$(OBJ)$
 $\$(CC) -o \$@ \$^ \$(CFLAGS)$

Summing up, considering the following:

mytarget: library.cpp main.cpp

$\$@$ evaluates to **mytarget**

$\$<$ evaluates to **library.cpp**

$\$^$ evaluates to **library.cpp main.cpp**

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

Makefile (example 5)

Makefile

```
1 INCDIR =./include
2 CC=gcc
3 CFLAGS=-I$(INCDIR)
4
5 OBJDIR=obj
6 # LIBDIR =./lib
7
8 # LIBS=-lmath # E.g., to link libmath.so
9
10 _DEPS = hellomake.h
11 DEPS = $(patsubst %, $(INCDIR)/%, $( _DEPS))
12
13 _OBJ = hellomake.o hellofunc.o
14 OBJ = $(patsubst %, $(OBJDIR)/%, $( _OBJ))
15
16
17 $(OBJDIR)/%.o: %.c $(DEPS)
18     $(CC) -c -o $@ $< $(CFLAGS)
19
20 hellomake: $(OBJ)
21     gcc -o $@ $^ $(CFLAGS) # $(LIBS)
22
23 .PHONY: clean
24
25 clean:
26     rm -f $(OBJDIR)/*.o *~ core $(INCDIR)/*~
27
```

clean is not genuine/real:
it is a "fake" target, i.e.,
clean is not a file

ex5 — -bash — 64×10

```
desktop-jisqlks:ex5 marcoautili$
desktop-jisqlks:ex5 marcoautili$ Make
gcc -c -o obj/hellomake.o hellomake.c -I./include
gcc -c -o obj/hellofunc.o hellofunc.c -I./include
gcc -o hellomake obj/hellomake.o obj/hellofunc.o -I./include #
desktop-jisqlks:ex5 marcoautili$
desktop-jisqlks:ex5 marcoautili$ Make clean
rm -f obj/*.o *~ core ./include/*~
desktop-jisqlks:ex5 marcoautili$
desktop-jisqlks:ex5 marcoautili$
```

1. Using folders for DEPS and OBJ

2. Cleaning up

- all .o files
- all possible backup files
- all possible core files

See next three slides

If you are sure that a file named
"clean" will never be in the folder,
you can also avoid to use .PHONY...

Functions for String Substitution and Analysis

`$(subst from, to, text)`

- Performs a textual replacement on the text `text`: each occurrence of `from` is replaced by `to`

For example, `$(subst ee, EE, feet on the street)` leads to `"fEEt on the strEEt"`

`$(patsubst pattern, replacement, text)`

- Finds `whitespace-separated words in text` that `match pattern` and replaces them with `replacement`. For example, `pattern` may contain a `'%'` which acts as a wildcard

In our case,

- `hellomake.h` will lead to `./include/hellomake.h`
`hellomake.o hellofunc.o` will lead to `./obj/hellomake.o ./obj/hellofunc.o`
- Summing up, each `.h files` listed in `_DEPS` will be searched for into `./include`, and each generated `.o files` will be placed into `./obj`

https://www.gnu.org/software/make/manual/html_node/Text-Functions.html

Cleaning up possible backup files

*~ means all files ending with ~

- Many Unix-like/Linux systems programs create backup files that end with ~
- On Unix-like/Linux systems *~ is like *.bak on Windows

For example

- The emacs and nano editors automatically save a backup copy of each file you edit
- When a file is saved, the old version gets saved using the file name with a tilde (~) added to the end

"core" files

- A core file might be created when **a program terminates unexpectedly**, due to a bug, or a violation of the operating system's or hardware's protection mechanisms
- The **operating system kills the program** and **creates a core** file that programmers can use to figure out what went wrong
- It contains a detailed description of the state that the program was in when it died
- If want to determine what program a core file came from, use the file command, like this:

\$ file core

- That will tell you the name of the program that produced the core dump. You may want to write the maintainer(s) of the program, telling them that their program dumped core