GNU make:
Short Introduction
What is MAKE?

• Simple but powerful build automation tool

• Helps us build executables and libraries from source code

• To achieve this it uses text files with specific syntax (Makefiles)
Why use MAKE?

- You can compile programs, build projects, install libraries and more without knowing the details of how it is done
- Not only for building but for every command or process you want to automate
Why use MAKE?

- Evaluates the order the files should be compiled
- Evaluates if files need to be compiled
Why use MAKE?

- Language independent

Not only for C or C++ *(misconception)*
Makefile simply specifies the shell commands to run
How does MAKE work?

- Reads a file with specific syntax (Makefile)
- Executes the commands in the file one at a time
- Echoes the commands to show you what is happening
- Terminates when everything is done or if any command returns a failure status
How to use MAKE?

- Simply run: `$ make`

- The program will look for a file named `makefile` or `Makefile` in the current directory

- If you want to use a different file or have multiple makefiles run: `$ make -f myfile`
How to use MAKE?

For more information:

$ man make
Makefiles
Makefile Syntax

• A rule in the makefile tells MAKE how to execute a series of commands in order to build a target file from source files

• It also specifies a list of dependencies of the target file

• If any dependencies have to be updated, does them first
• The general syntax of a Makefile target rule is:

```
target [target...] : [dependency ....]
[tab] [ command ...]
```
Rules

Example:

```
assignment1: translate.c
gcc -ansi -pedantic -Wall translate.c -o assignment1
```
Makefile Syntax

• To build our program we would run:
  $ make assignment1

• If make is executed without parameters it updates the first target listed in the makefile

• The make utility updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist
Makefile Syntax

• Lines starting with # are comments

```
# My first assignment!
assignment1: translate.c
    gcc -ansi -pedantic -Wall translate.c -o assignment1
```
• To suppress echoing the actual command, start with @

# My first assignment!
assignment1: translate.c
    gcc -ansi -pedantic -Wall translate.c -o assignment1

clean:
    @echo ‘Cleaning files...’
    @rm -f *.o *.out
Variables

• When makefiles are large and we want to make changes, it takes a lot of time (and errors) to change values that are hard-coded

• Solution: use Variables
Variables

- The simplest way to define a variable in a makefile is to use the `=` operator.

- A variable begins with a `$` and is enclosed within parentheses `(...)` or braces `{}`.
Variables

CFLAGS=-ansi -pedantic -Wall

# My first assignment!
assignment1: translate.c
  gcc $(CFLAGS) translate.c -o assignment1

clean:
  @echo ‘Cleaning files...’
  @rm -f *.o *.out
Variables

• Automatic Variables:
  
  $@$ : the target filename.
  $*$ : the target filename without the file extension.
  $<$ : the first prerequisite filename.
  $^$: the filenames of all the prerequisites, separated by spaces, discard duplicates.
  $^+$ : similar to $^$, but includes duplicates.
  $??$ : the names of all prerequisites that are newer than the target, separated by spaces.
Variables

CFLAGS=-ansi -pedantic -Wall

# My first assignment!
assignment1: translate.c
gcc $(CFLAGS) $< -o @$

clean:
  @echo 'Cleaning files...'
  @rm -f *.o *.out
Makefile Syntax

- A pattern rule looks like an ordinary rule, except that its target contains exactly one character `%`

- The `%` can match any substring of zero or more characters while other characters match only themselves

```plaintext
%.o: %.c lib.h
    gcc $(CFLAGS) -c $< -o $@
```
Example

# Author: Foivos S. Zakkak <zakkak@csd.uoc.gr>
CFLAGS=-ansi -pedantic -Wall

all: test1 test2

test1: main.o lib1.o
   gcc $(CFLAGS) main.o lib1.o -o test1

test2: main.o lib2.o
   gcc $(CFLAGS) $^ -o $@

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

clean:
   -rm *.o

distclean: clean
   -rm test1 test2

turnin: distclean
   turnin ask2@hy255 ../ask2
Notes

- **GNU make** is the standard implementation of make for Linux and OS X

- For Windows use GnuWin32 (?)

  *nmake is a command-line tool which is part of Visual Studio*

- Careful when using derivatives, they may have different behavior!!!
Notes

• Commands are executed in a sub-process shell

# This will not work. Different shells!
case1:
   cd src
   ls -l

# This will work. Commands use the same shell
case2:
   cd src; \
   ls -l
## Notes

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$ make</code></td>
<td>Use the default descriptor file, build the first target in the file</td>
</tr>
<tr>
<td><code>$ make assignment1</code></td>
<td>Use the default descriptor file, build the target with name “assignment1”</td>
</tr>
<tr>
<td><code>$ make -f myfile</code></td>
<td>Use the file “myfile”, build the first target in the file</td>
</tr>
<tr>
<td><code>$ make -f myfile assignment1</code></td>
<td>Use the file “myfile”, build the target with name “assignment1”</td>
</tr>
</tbody>
</table>
For more information:

- man make
- Second tutorial...
- https://www.gnu.org/software/make/