/unc/comp211

Systems Fundamentals

Make & Makefiles

!

Go ahead and open both:

1. PollEv.com/compunc
2. Terminal - start a session, cd into lecture - pull today's materials with git pull origin master

Outline

- Build Systems, make, and Makefiles

Build Systems

- Machine code program files are complex digital artifacts to produce
  - Many tools are required to compile high level programs into machine code

- Tools in a compilation process may include:
  - linters to check and fix deviations from a style guide
  - running of test harnesses to verify lack of regressions
  - compilation of source code to an intermediate representation
  - compilation of intermediate representations to machine code

- Carrying out each step manually is tedious and error prone.

- During development only small parts of a program change.
  - Why repeat the whole process from scratch when most steps have same results?

Enter: `make` and Makefiles (1976 - Bell Labs)

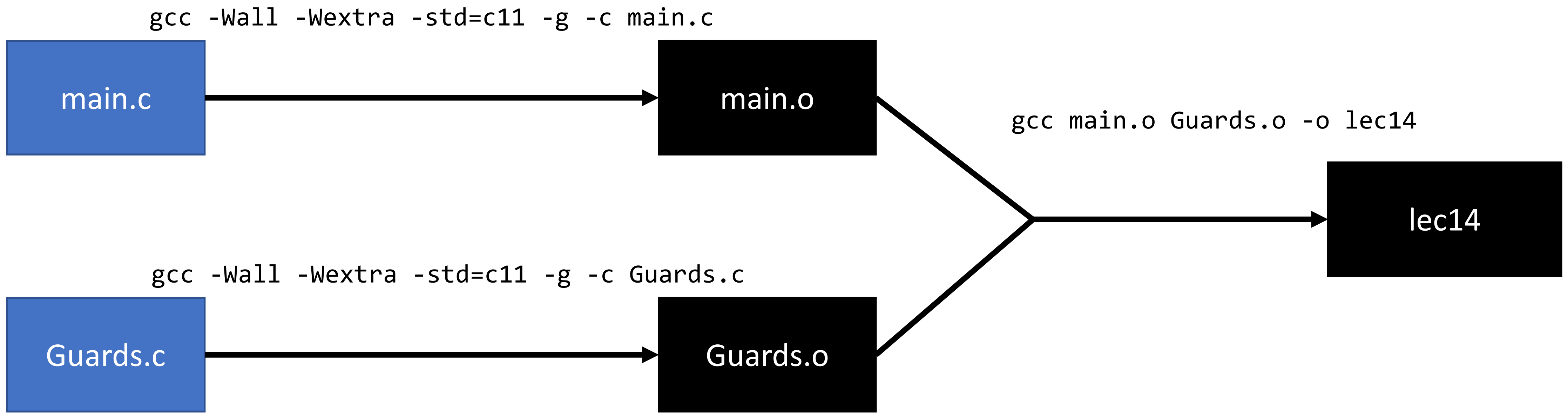
"Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, cc *.o was therefore unaffected).

As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make **that weekend**.

Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff."

— *Stuart Feldman*

Representing a Build Process



```
gcc -Wall -Wextra -std=c11 -g -c main.c
gcc -Wall -Wextra -std=c11 -g -c Guards.c
gcc main.o Guards.o -o lec14
```

- A Makefile describes the structure of a build process
  - The ***nodes*** of the graph are ***files*** and the ***edges*** are ***build steps***

- If a node/target is determined to be missing, make backtracks to the missing prerequisites and execute the commands of each edge in order. Additionally, if a file that produces another file is newer than the file it produces, make will automatically rebuild it.

- This is another example application directed acyclic graphs, partial ordering, and topological sort!

Makefile - Rule Syntax

```
<target-file>: <prerequisite-file>*
[tab-character]<recipe-to-produce-target-from-prereqs>*
```

Example Rules:

```
lec14: main.o Guards.o
	gcc main.o Guards.o -o lec14

Guards.o: Guards.c
	gcc -Wall -Wextra -std=c11 -g -c Guards.c

main.o: main.c
	gcc -Wall -Wextra -std=c11 -g -c main.c
```

The `make` Build System's Big Idea

- In a `Makefile`, you specify each step's:
  1. **Prerequisite** Source files
  2. **Recipe** of **Shell Command**(s) that use prerequisite files to produce target file
  3. The **Target** file produced by the recipe

- **`make`** reads the Makefile and then figures out which target files are missing or outdated and run *only* the commands needed to build exactly those targets.
  - Early steps in a build will run a commands taking ***source*** files to produce ***target*** files.
  - Later rules use earlier ***target*** files as ***source*** files to produce additional ***target*** files.

- **`make`** was designed for C projects but has many other applications
  - This is evidence of a *good abstraction.* Does it generalize beyond intent?

Makefile Variables

- It is good practice to define variables at the top of your Makefile

- Definition Syntax: **`VARIABLE=value`**
  - The value is a string that terminates at the end of the line
  - The variable name does not need to be all caps, but they often are

- Common variables:
  - CC is the name of the C compiler to compile with
    ```
    CC=gcc
    ```
  - CC_FLAGS are the compiler options
    ```
    CC_FLAGS=-Wall -Wextra -std=c11 -g
    ```

- Usage Syntax: **`${VARIABLE}`**
  ```
  Guards.o: Guards.c
      ${CC} ${CC_FLAGS} -c Guards.c
  ```

- The above example expands to:
  ```
  Guards.o: Guards.c
      gcc  -Wall -Wextra -std=c11 -g -c Guards.c
  ```

Hands-on

- In today's Makefile, change the topmost rule for lec14 from:

```
lec14: main.o Guards.o Point.o Path.o
      gcc -Wall -Wextra -std=c11 -g main.o Guards.o Point.o Path.o -o lec14 -lm
```

- To:

```
${TARGET}: ${OBJ_FILES}
      ${CC} ${CC_FLAGS} ${OBJ_FILES} -o ${TARGET} ${LINK_MATH}
```

- Save and try running **make**. Check-in when project is building.

Automatic Variables

- Recipes can use automatic variables to reference target or prerequisite file(s)

- Automatic variables are available:

  `${@}` - The ***target*** of the rule

  `${^}` - The names of ***all prerequisite files***

  `${<}` - The names of the ***first prerequisite*** file

  `${?}` - The names of all ***prerequisite files*** that are ***newer*** than the target

- More automatic variables exist, too. See the full documentation:

https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables

Hands-on

- Change the rule we just changed to use automatic variables:

```
${TARGET}: ${OBJ_FILES}
      ${CC} ${CC_FLAGS} ${OBJ_FILES} -o ${TARGET} 
${LINK_MATH}
```

- To:

```
${TARGET}: ${OBJ_FILES}
      ${CC} ${CC_FLAGS} ${^} -o ${@} ${LINK_MATH}
```

- Save and try running **make clean** followed by **make** again. Check-in when project is building.

Running **make** with specific goals

- The Default Goal of a Makefile is its first target

- You can change the goal of **make** by naming a specific target instead

- For example:

```
$ make main.o    # sets goal to target main.o
$ make Guards.o  # sets goal to target Guards.o
```

- This is useful in two cases:
    1. When your build process is slow, and you want to focus on a sub-target
    2. When you want to run recipes with *phony* targets (next slide!)

Phony Rules

- Some recipes of a Makefile intentionally *do not* produce their target...
  - ...they're **phony**!

- Common task: **clean** up (delete) files generated by the build process.

```
.PHONY: clean
clean:
    rm -f ${OBJ_FILES} ${TARGET}
```

- Phony rules are used to automate common tasks in development that are *outside* of the compilation process. Other examples besides clean:
  - test - run unit / functional test suites
  - run - build and run program
  - debug - build program and run gdb

Hands-on - A Phony run Rule

- After the clean rule, add the following:

```
.PHONY: run
run: ${TARGET}
	./${TARGET}
```

- Try: make run, make clean, make run.

- Check-in when project is building.

Removing Repetitive Rules with Patterns (1 / 2)

- Notice the common structure to the rules right...
- When many rules take on a pattern where the target name is based on the prerequisite name you can use a **pattern rule**.

```
main.o: main.c
	${CC} ${CC_FLAGS} -c main.c

Point.o: Point.c
	${CC} ${CC_FLAGS} -c Point.c

Path.o: Path.c
	${CC} ${CC_FLAGS} -c Path.c

Guards.o: Guards.c
	${CC} ${CC_FLAGS} -c Guards.c
```

Pattern Rules (2 / 2)

- A **Pattern Rule** is one where the *target* contains a single % symbol

```
%.o: %.c
    ${CC} ${CC_FLAGS} -c ${^}
```

- If another rule's prerequisite(s) match a target pattern, then **implicit rules** are produced:
  - The matched part of the % in the target is also substituted in its prerequisites.
  - For example, with the rule above, since the top-level rule has Guards.o as a prerequisite, make generates an implicit rule of:

```
 Guards.o: Guards.c
      ${CC} ${CC_FLAGS} -c ${^}
```

- Automatic variables allow implicit recipes to use target **${@}** and prerequisite **${^}** values.

- Warning: A common misconception is that these rules are produced by make searching for matching prerequisite files. This is not the case. For a pattern rule to produce implicit rules some other rule's prerequisites must reference the pattern's target.

- Full Documentation: https://www.gnu.org/software/make/manual/html_node/Pattern-Rules.html

Hands-on - A Pattern Rule

- Remove the repetitive rules and replace them with a pattern rule:

```
%.o: %.c
	${CC} ${CC_FLAGS} -c ${^}
```

- Try: make clean, make run.

- Check-in when project is building.

`make` is goes far deeper than this tutorial

- As a 40-year old tool it has accumulated many capabilities

- Many features try to avoid redundancy and verbosity of the `Makefile`
  - The downside is this leads to cryptic, non-obvious Makefiles

- Special features to use `make` for building specific kinds of projects
  - i.e. C projects or archives

- Modern build systems like CMake will *generate a Makefile* specific to the system the project is being built on.
  - Eases portability between operating systems and versions.

- The documentation for **`make`** is generally very good:
  - https://www.gnu.org/software/make/manual/make.html

Many build tools are `make`-inspired

- Nothing stops you from using make for any project, but many ecosystems revolve around tooling custom suited for their environment.
- C/C++ – make, Cmake, bazel
- Rust – cargo
- Java – Ant, Maven, Gradle, bazel
- Node.js / JavaScript / TypeScript – npm, webpack, gulp, grunt
- Python – Scons, Waf

/unc/comp211

Systems Fundamentals

Make & Recursive Structs!

Go ahead and open both:

1. PollEv.com/compunc
2. Terminal - start a session, cd into lecture - pull today's materials with git pull origin master

Outline

- Build Systems, make, and Makefiles
- Recursive Structs
- Ownership in Memory Management
- Recursive to Iterative with Cursor Pointers

Build Systems

- Machine code program files are complex digital artifacts to produce
  - Many tools are required to compile high level programs into machine code

- Tools in a compilation process may include:
  - linters to check and fix deviations from a style guide
  - running of test harnesses to verify lack of regressions
  - compilation of source code to an intermediate representation
  - compilation of intermediate representations to machine code

- Carrying out each step manually is tedious and error prone.

- During development only small parts of a program change.
  - Why repeat the whole process from scratch when most steps have same results?

Enter: `make` and Makefiles (1976 - Bell Labs)

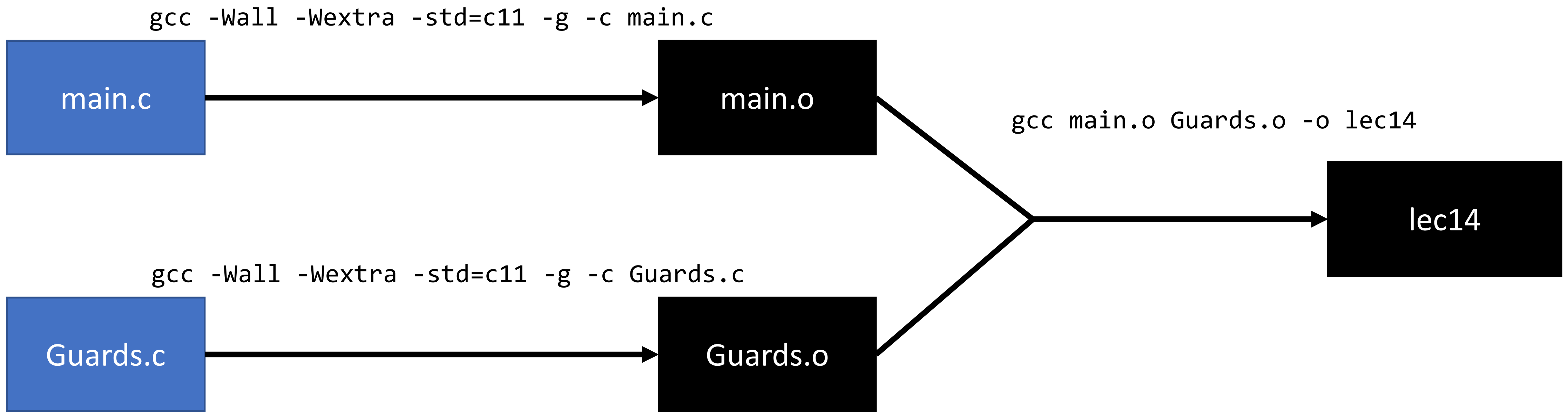
"Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, cc *.o was therefore unaffected).

As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make **that weekend**.

Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff."

— *Stuart Feldman*

Representing a Build Process


gcc -Wall -Wextra -std=c11 -g -c main.c
main.c
main.o
gcc main.o Guards.o -o lec14
lec14
gcc -Wall -Wextra -std=c11 -g -c Guards.c
Guards.c
Guards.o

- A Makefile describes the structure of a build process
  - The ***nodes*** of the graph are ***files*** and the ***edges*** are ***build steps***

- If a node/target is determined to be missing, make backtracks to the missing prerequisites and execute the commands of each edge in order. Additionally, if a file that produces another file is newer than the file it produces, make will automatically rebuild it.

- This is another example application directed acyclic graphs, partial ordering, and topological sort!

Makefile - Rule Syntax

```
<target-file>: <prerequisite-file>*
[tab-character]<recipe-to-produce-target-from-prereqs>*
```

<u>Example Rules:</u>

```
lec14: main.o Guards.o
	gcc main.o Guards.o -o lec14

Guards.o: Guards.c
	gcc -Wall -Wextra -std=c11 -g -c Guards.c

main.o: main.c
	gcc -Wall -Wextra -std=c11 -g -c main.c
```

The `make` Build System's Big Idea

- In a `Makefile`, you specify each step's:
  1. **Prerequisite** Source files
  2. **Recipe** of **Shell Command**(s) that use prerequisite files to produce target file
  3. The **Target** file produced by the recipe

- **`make`** reads the Makefile and then figures out which target files are missing or outdated and run *only* the commands needed to build exactly those targets.
  - Early steps in a build will run a commands taking ***source*** files to produce ***target*** files.
  - Later rules use earlier ***target*** files as ***source*** files to produce additional ***target*** files.

- **`make`** was designed for C projects but has many other applications
  - This is evidence of a *good abstraction.* Does it generalize beyond intent?

Makefile Variables

- It is good practice to define variables at the top of your Makefile

- Definition Syntax: **`VARIABLE=value`**
  - The value is a string that terminates at the end of the line
  - The variable name does not need to be all caps, but they often are

- Common variables:
  - CC is the name of the C compiler to compile with

    ```
    CC=gcc
    ```
  - CC_FLAGS are the compiler options

    ```
    CC_FLAGS=-Wall -Wextra -std=c11 -g
    ```

- Usage Syntax: **`${VARIABLE}`**

```
Guards.o: Guards.c
	${CC} ${CC_FLAGS} -c Guards.c
```

- The above example expands to:

```
Guards.o: Guards.c
	gcc  -Wall -Wextra -std=c11 -g -c Guards.c
```

Hands-on

- In today's Makefile, change the topmost rule for lec14 from:

```
lec14: main.o Guards.o Point.o Path.o
	gcc -Wall -Wextra -std=c11 -g main.o Guards.o Point.o Path.o -o lec14 -lm
```

- To:

```
${TARGET}: ${OBJ_FILES}
	${CC} ${CC_FLAGS} ${OBJ_FILES} -o ${TARGET} ${LINK_MATH}
```

- Save and try running **make**. Check-in when project is building.

Automatic Variables

- Recipes can use automatic variables to reference target or prerequisite file(s)

- Automatic variables are available:

  `${@}` - The ***target*** of the rule

  `${^}` - The names of ***all prerequisite files***

  `${<}` - The names of the ***first prerequisite*** file

  `${?}` - The names of all ***prerequisite files*** that are ***newer*** than the target

- More automatic variables exist, too. See the full documentation:

https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables

Hands-on

- Change the rule we just changed to use automatic variables:

```
${TARGET}: ${OBJ_FILES}
      ${CC} ${CC_FLAGS} ${OBJ_FILES} -o ${TARGET}
${LINK_MATH}
```

- To:

```
${TARGET}: ${OBJ_FILES}
      ${CC} ${CC_FLAGS} ${^} -o ${@} ${LINK_MATH}
```

- Save and try running **make clean** followed by **make** again. Check-in when project is building.

Running **make** with specific goals

- The Default Goal of a Makefile is its first target

- You can change the goal of **make** by naming a specific target instead

- For example:

```
$ make main.o     # sets goal to target main.o
$ make Guards.o   # sets goal to target Guards.o
```

- This is useful in two cases:
  1. When your build process is slow, and you want to focus on a sub-target
  2. When you want to run recipes with *phony* targets (next slide!)

Phony Rules

- Some recipes of a Makefile intentionally *do not* produce their target...
  - ...they're **phony**!

- Common task: **clean** up (delete) files generated by the build process.

```
.PHONY: clean
clean:
    rm -f ${OBJ_FILES} ${TARGET}
```

- Phony rules are used to automate common tasks in development that are *outside* of the compilation process. Other examples besides clean:
  - test - run unit / functional test suites
  - run - build and run program
  - debug - build program and run gdb

Hands-on - A Phony run Rule

- After the clean rule, add the following:

```
.PHONY: run
run: ${TARGET}
	./${TARGET}
```

- Try: make run, make clean, make run.

- Check-in when project is building.

Removing Repetitive Rules with Patterns (1 / 2)

- Notice the common structure to the rules right...
- When many rules take on a pattern where the target name is based on the prerequisite name you can use a **pattern rule**.

```
main.o: main.c
    ${CC} ${CC_FLAGS} -c main.c

Point.o: Point.c
    ${CC} ${CC_FLAGS} -c Point.c

Path.o: Path.c
    ${CC} ${CC_FLAGS} -c Path.c

Guards.o: Guards.c
    ${CC} ${CC_FLAGS} -c Guards.c
```

Pattern Rules (2 / 2)

- A **Pattern Rule** is one where the *target* contains a single % symbol

```
%.o: %.c
    ${CC} ${CC_FLAGS} -c ${^}
```

- If another rule's prerequisite(s) match a target pattern, then **implicit rules** are produced:
  - The matched part of the % in the target is also substituted in its prerequisites.
  - For example, with the rule above, since the top-level rule has Guards.o as a prerequisite, make generates an implicit rule of:

```
Guards.o: Guards.c
     ${CC} ${CC_FLAGS} -c ${^}
```

- Automatic variables allow implicit recipes to use target **${@}** and prerequisite **${^}** values.

- Warning: A common misconception is that these rules are produced by make searching for matching prerequisite files. This is not the case. For a pattern rule to produce implicit rules some other rule's prerequisites must reference the pattern's target.

- Full Documentation: https://www.gnu.org/software/make/manual/html_node/Pattern-Rules.html

Hands-on - A Pattern Rule

- Remove the repetitive rules and replace them with a pattern rule:

```
%.o: %.c
    ${CC} ${CC_FLAGS} -c ${^}
```

- Try: make clean, make run.

- Check-in when project is building.

make is goes far deeper than this tutorial

- As a 40-year old tool it has accumulated many capabilities

- Many features try to avoid redundancy and verbosity of the `Makefile`
  - The downside is this leads to cryptic, non-obvious Makefiles

- Special features to use `make` for building specific kinds of projects
  - i.e. C projects or archives

- Modern build systems like CMake will *generate a Makefile* specific to the system the project is being built on.
  - Eases portability between operating systems and versions.

- The documentation for **`make`** is generally very good:
  - https://www.gnu.org/software/make/manual/make.html

Many build tools are `make`-inspired

- Nothing stops you from using make for any project, but many ecosystems revolve around tooling custom suited for their environment.
- C/C++ – make, Cmake, bazel
- Rust – cargo
- Java – Ant, Maven, Gradle, bazel
- Node.js / JavaScript / TypeScript – npm, webpack, gulp, grunt
- Python – Scons, Waf