make

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

- GNU make utility
  - A tool that makes it easy for you to describe how to compile programs to build C/C++ applications
  - Reads a description of a project from a **`Makefile`**
    - You need to describe all the files and their dependencies in the Makefile
- To use the "make" utility, just type "make" at the command prompt
  - By default, make will read and process a **`Makefile`** in your current directory.
  - Otherwise, specify the file name with **`-f`** option
    - **`make -f your_make_file`**

Makefile

A **Makefile** specifies a set of compilation rules

- in terms of *targets* (such as executables) and their *dependencies* (such as object files and source files)
- The "make" utility goes through the Makefile and follows the chain of dependencies until it reaches the end of the chain and then begins backing out executing the commands found in each target's rule
- **make** looks at the time stamp for each file in the chain and compiles from the point that is required to bring every file in the chain up to date
  - The "**make**" utility compiles only those source files that have been changed and the modules that depend upon them

Makefile Format

```
target: dependencies
→a tabcommand
```

- dependencies: names of files depended by the target, i.e:

  ```
  Hello:    Hello.cpp
  ```

- Rule/command: (the how) to construct the target from the dependencies, such as

  ```
  g++ -o Hello Hello.cpp
  ```

- **The first character pre the command line must be a tab**

  The space between the beginning of the line to the command MUST be the tab, not white an empty spaces

A Simple Makefile

- Note: the line index is not part of the Makefile

```
1 #Makefile
2 GCC = gcc
3 OBJS  = foo.o bar.o baz.o
4 CLFAGS = -Wall -O2
5 LDLIBS = -L./ -lbar

6 prog: $(OBJS)
        $(GCC) -o prog $(OBJS) $(LDLIBS)
7 foo.o: foo.c
        $(GCC) $(CFLAG) -c foo.c
8 bar.o: bar.c
        $(GCC) $(CFLAG) -c bar.c
9 baz.o: baz.c
        $(GCC) $(CFLAG) -c baz.c

10.PHONY: install clean
11 install:
        install -m 755 foo $HOME/local/bin
12 clean:
        rm *.o; rm foo
```

Makefile Explained

- L1: comment
- L2-L5: define variables (macro) OBJS, LDLIBS, GCC
- L6-L9: definition of **compilation rules**
  - It states that target `prog` depends on (or is built from) the object files whose names are contained in variable **`OBJS`** (called **dependency** list)
  - command line after that tells how to build the target from the dependency list, **the first character in the command line must be a tab**
- L10, .PHONY:
  - tells make that install and clean are not target files to avoid a conflict with a file of the same name
  - And there is no dependents for the phony target, so it will always be executed when the target is requested
- L11-12: install and clean are phony targets

Phony Targets

- A phony target is one that is not really the name of a file
- It is just a name for a recipe to be executed when you make an explicit request.
- There are two reasons to use a phony target
  - to avoid a conflict with a file of the same name,
  - to improve performance.
- Command: **`make clean`**
  - Will execute the recipe: **`rm *.o; rm fo`**
- Command: **`make install`**
  - Will execute the recipe: **`install –m 755 foo $HOME/local/bin`**
    - **`install:`** **copy files and set attributes**

For more detail:

- https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html

“Implicit Rules”

- Built-In Rules
  - Compiling C programs file.o

    `$(CC) $(CFLAGS) -c`

  - Compiling C++ programs
    - `$(CXX) $(CXXFLAGS) -c`
- Pattern Rules (prefix %)
  - `%.o:%.c` How to make a .o file from a .c file

https://www.gnu.org/software/make/manual/html_node/Implicit-Rules.html#Implicit-Rules

Automatic variables--special symbols

```
.c.o:
    $(GCC) $(CFLAG) -c $<
```

```
%.o: %.c
    $(GCC) $(CFLAG) -c $<
```

```
foo: $(OBJS)
    $(GCC) -o $@ $(OBJS) $(LDLIBS)
```

- **The inference rule**
  - `.s1.s2:` describes how to build a target that is appended with `.s2` with a prerequisite that is appended with `.s1`.
- `.SUFFIXES: .o:.c`
- `$*` ➔ current target without the extension
- `$<` ➔ is a dependent file (full name of the prerequisite file)
- `$@` ➔ represents the full target name of the current target

“make” Inference Rules

```
6 foo.o: foo.c
        $(GCC) $(CFLAG) -c foo.c
7 bar.o: bar.c
        $(GCC) $(CFLAG) -c bar.c
8 baz.o: baz.c
        $(GCC) $(CFLAG) -c baz.c
```

```
.c.o:
    $(GCC) $(CFLAG) -c $<
```

Look lines 7-9, the rules to make OBJECT file, they are very similar

```
<filename>.o : <filename>.c
$(GCC) $(CFLAG) -c <filename>.c
```

```
6 foo: $(OBJS)
        $(GCC) -o foo $(OBJS) $(LDLIBS)
```

```
5 foo: $(OBJS)
        $(GCC) -o $@ $(OBJS) $(LDLIBS)
```

File: Makefile with variables (macros)

```
#Makefile, the indent in the rules are always a TAB
PROGRAM       := qemployee
CXX           := c++
SRC           := employee.cpp main.cpp
OBJS          := employee.o main.o
LIBDIRS       := ../lib
INCLUDEDIRS   := ../include
LBFLAGS       := -L$(LIBDIRS)
CXXFLAGS      := -Wall -O2 -I$(INCLUDEDIRS)
$(PROGRAM):$(OBJS)
        $(CXX)  $(OBJS) -o $(PROGRAM) $(LBFLAGS)
employee.o: employee.cpp
        $(CXX) $(CXXFLAGS) -c employee.cpp -o employee.o
main.o: main.cpp
        $(CXX) $(CXXFLAGS) -c main.cpp  -o main.o
clean:
        rm *.o; rm employee
install: empolyee
       install -m 755 employee $HOME/local/bin
.PHONY: install clean
```

References

- An Introduction to GCC

  http://www.network-theory.co.uk/docs/gccintro/index.html

- GNU Debugger

  http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html

- GNU “make”

  http://www.gnu.org/software/make/manual/make.html