Makefiles

Manolis Koubarakis

The Utility `make`

- The `make` utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
- It is useful when we write large programs that are contained in more than one file.

Preparing and Running `make`

- In order to use `make`, you should create a file named **`Makefile`**. This file describes the relationships among files in your program and provides commands for updating each file.
- In a program, typically, the **executable** file is updated from **object files**, which in turn are made by compiling **source files**.
- Once a suitable makefile exists, each time you change some source files, the simple shell command

```
make
```

  suffices to perform all necessary recompilations.
- The `make` program uses the **makefile** in the current directory and the **last-modification times** of the files to decide which of the files need to be updated. For each of those files, it issues the **recipes** recorded in the makefile.

Rules

- A simple makefile consists of **rules** with the following syntax:

```
target … : prerequisites …
        recipe
        …
        …
```

- A **target or a goal** is usually the name of a file that is generated by a program. Examples of targets are executable or object files. A target can also be the name of an action to carry out, such as 'clean'.
- A **prerequisite** is a file that is used as input to create the target. A target often depends on several files.
- A **recipe** is an action that `make` carries out. A recipe may have more than one command.
- Prerequisites are **optional**. For example, the rule containing the delete command associated with the target 'clean' does not have prerequisites.
- A **rule** explains how and when to remake certain files which are the targets of the particular rule. `make` carries out the recipe on the prerequisites to create or update the target.

Example

```
pqsort : sorting.o PQImplementation.o
	gcc sorting.o PQImplementation.o -o pqsort

sorting.o :sorting.c PQInterface.h PQTypes.h
	gcc -c sorting.c

PQImplementation.o :PQImplementation.c PQInterface.h
PQTypes.h
	gcc -c PQImplementation.c

clean:
	rm pqsort sorting.o PQImplementation.o
```

Mind the tabs!!!

Comments

- In this example makefile, the **targets** include the executable file `pqsort`, and the object files `sorting.o` and `PQImplementation.o`.
- The **prerequisites** are files such as `sorting.c` and `PQInterface.h` and `PQTypes.h`
- **Recipes** include commands like `gcc -c sorting.c` and `gcc -c PQImplementation.c`

- A **recipe** may follow each line that contains a target and prerequisites. These recipes say how to update the target file. **Important**: A **tab character** must come at the beginning of every line that contains a recipe to distinguish recipes from other lines in the makefile.
- The target `clean` is not a file, but merely the name of an action. Notice that `clean` is not a prerequisite of any other rule. Consequently, `make` never does anything with it unless you tell it specifically. Note also that the rule for `clean` does not have any prerequisites, so the only purpose of the rule is to run the specified recipe. Targets that do not refer to files but are just actions are called **phony targets**.

How `make` is invoked

- For the previous example, after some of the source `.c` files have changed, and we would like to create a new executable, we just write `make` on the command line.

How `make` Processes a Makefile

- When `make` is called, it reads the makefile in the current directory and starts processing the first rule of the makefile. In our case, this is the rule for the executable file `pqsort`.
- However, before `make` can process this rule, it should process the rules that update the files on which `pqsort` depends i.e., `sorting.o` and `PQImplementation.o.`
- These in turn depend on files such as `sorting.c`, `PQInterface.h` and `PQTypes.h` which are not the targets of any rule so the recursion stops here.

Variables

- **Variables** allow a text string to be defined once and substituted in multiple places later.
- For example, it is standard practice for every makefile to have a variable named `objects`, which is defined to be a list of all object file names.
- We can define this variable by writing

```
objects=sorting.o PQImplementation.o
```

- Then the variable can be used in the makefile using the notation `$(variable)`.

Example (cont'd)

```
objects=sorting.o PQImplementation.o

pqsort : $(objects)
	gcc $(objects) -o pqsort

sorting.o :sorting.c PQInterface.h PQTypes.h
	gcc -c sorting.c

PQImplementation.o :PQImplementation.c PQInterface.h
PQTypes.h
	gcc -c PQImplementation.c

clean:
	rm pqsort $(objects)
```

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.
- `make` has an **implicit rule** for updating a `.o` file from a correspondingly named `.c` file using a `cc -c` command (not `gcc`).

- So we can write our example as follows.

Example (cont’d)

```
objects=sorting.o PQImplementation.o

pqsort : $(objects)
      gcc $(objects) -o pqsort

sorting.o : PQInterface.h PQTypes.h
PQImplementation.o : PQInterface.h PQTypes.h

clean:
      rm pqsort $(objects)
```

Rules for Cleaning the Directory

- We can use makefiles to do other things except compiling programs. For example, we can have a recipe that deletes all the object files and executables so that the directory is clean.
- In our example, this is done by the following rule:

```
clean:
      rm pqsort $(objects)
```

- `clean` here is called a **phony target**.

- To avoid problems with files with the name clean in the same directory, you can write the above rule as follows:

```
.PHONY clean
clean:
      rm pqsort $(objects)
```

Rules for Cleaning the Directory (cont'd)

- You can execute the above rule by executing the shell command

```
make clean
```

Readings

- These slides were created by copying (sometimes verbatim!) material from the manual http://www.gnu.org/software/make/manual/make.html .
- Read this manual for more information (just reading Chapter 2 will suffice).