Introduction to making Makefiles

Johannes Franken
`<jfranken@jfranken.de>`

`make` is an interpreter for Makefiles. A Makefile is much like a shell script with additional directions, which qualify **`make`** to focus on the *required* lines. This will save you a lot of time.

Unfortunately common documentation stresses on compiling C programs. This guide shows how to use **`make`** for workaday tasks, that have nothing to do with the C programming language.

Contents

What's in those Makefiles?

A Makefile basically consists of

- Definitions of variables and functions,
  e.g. **`myshell=bash`**
- Comments,
  e.g. **`# converting postscript to pdf`**
- Includes,
  e.g. **`-include Makefile.local`**
- Rules,
  e.g. **`%.pdf: %.ps; -@ps2pdf $<`**

Explicit rules

Basics

The generic anatomy of a rule is as follows:

```
target [ more targets] :[:] [ prerequisites ] [; commands]
[  <tab>  commands ]
[  <tab>  commands ]
      ...
```

Everything in square brackets is optional.

Rules that have got commands are called *explicit*, those which haven't are called *implicit*. This chapter is about *explicit* rules.

Mind the tab key (**`<tab>`**) prefacing the commands past the first line.

Use a semicolon to stuff multiple commands into one single line, or a backslash to join split lines.

Example

The following Makefile:

```
hello:
        @echo hello \
        world

diskfree:; df -h /
```

defines two rules (with targets **`hello`** and **`diskfree`**) and (explicitly...) assigns each of them a short command. Now we can specifically call those commands:

```
$ make hello
hello world
$ make diskfree
df -h /
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda6              43G   37G  3.6G  92% /
```

BTW: The at-sign (`@`) in front of the command prevents **`make`** from announcing the command it's going to do.

Observation 1: If you don't tell **`make`** a target, it will simply do the first rule:

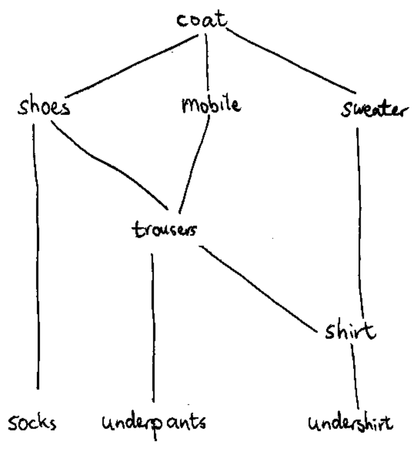
```
$ make
hello world
```

Observation 2: If you tell **`make`** multiple targets, it will do them all in order:

```
$ make diskfree hello
df -h /
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda6              43G   37G  3.6G  92% /
hello world
```

Prerequisites

When getting dressed every morning, the order of installations should result from the following dependency tree:



It makes sure you're already wearing socks and trousers before getting into the shoes. We pronounce:

Socks and trousers are prerequisites for shoes.

and minute for the Makefile:

```
shoes: socks trousers
        @echo put on shoes
```

Since **`make`** will substitute the target's name wherever it reads **`$@`**, we can generalize to:

```
shoes: socks trousers
        @echo put on $@
```

If you pursue this method for the full tree in one-lined notation, you will get the following Makefile:

```
.PHONY: coat shoes mobile sweater socks\
        trousers shirt pants undershirt

# target    prerequisite          command
# ---------------------------------------------------
coat:       shoes mobile sweater;  @echo put on $@
shoes:      socks trousers;        @echo put on $@
mobile:     trousers;              @echo put on $@
sweater:    shirt;                 @echo put on $@
socks:      ;                      @echo put on $@
trousers:   pants shirt;           @echo put on $@
shirt:      undershirt;            @echo put on $@
pants:      ;                      @echo put on $@
undershirt: ;                      @echo put on $@
```

BTW: The **`.PHONY`**-directive tells **`make`** which target names have nothing to do with potentially existing files of the same name. The next example will work out on that.

The result looks quite promising:

```
$ make coat | nl
     1  put on socks
     2  put on pants
     3  put on undershirt
     4  put on shirt
     5  put on trousers
     6  put on shoes
     7  put on mobile
     8  put on sweater
     9  put on coat
```

Just in case I just wanted to get into my sweater and everything what’s neccessary for it:

```
$ make sweater | nl
     1  put on undershirt
     2  put on shirt
     3  put on sweater
```

BTW: That **`|nl`** doesn’t belong to **`make`**. This little unix tool just numbers **`make`**’s output lines.

Referencing existing files

Task: In the last example **make** assumed there always was no prerequisite satisfied at invocation time, means you have to be naked before getting dressed. Well, I prefer having a mug of coffee after installing the basics and before putting on the uncomfortable articles. Now, how can I allow the following sequence to work?

```
$ make shirt trousers socks
$ drink coffee
$ make coat
```

Basic approach: Ascertaining the steps to the coat must be split in two parts. For this to work, **make** has to save the state after the first run. So I create an empty file after finishing each target, with the filename equal to the target's name. When I now take out the **.PHONY**-line, **make** will conclude from an existing file to a satisfied target. Exactly speaking, it will compare the timestamps of target's and prerequisite's files. See next chapter for details on this.

Solution: The following Makefile:

```
# target    prerequisite           command
# ----------------------------------------------------------
coat:       shoes mobile sweater;  @echo put on $@; touch $@
shoes:      socks trousers;        @echo put on $@; touch $@
mobile:     trousers;              @echo put on $@; touch $@
sweater:    shirt;                 @echo put on $@; touch $@
socks:      ;                      @echo put on $@; touch $@
trousers:   pants shirt;           @echo put on $@; touch $@
shirt:      undershirt;            @echo put on $@; touch $@
pants:      ;                      @echo put on $@; touch $@
undershirt: ;                      @echo put on $@; touch $@
```

will do the job:

```
$ make shirt trousers socks | nl
     1  put on undershirt
     2  put on shirt
     3  put on pants
     4  put on trousers
     5  put on socks
$ ls
trousers    undershirt   shirt   socks  pants
$ # drink coffee
$ make coat | nl
     1  put on shoes
     2  put on mobile
     3  put on pullover
     4  put on coat
$ ls
shirt     coat      shoes  undershirt   mobile
trousers  pullover  socks  pants
```

Implicit rules

I can split each rule into two parts:

- An *implicit* rule stating the prerequisits, and
- an *explicit* one for the commands.

Any rules in the last example lead to the same commands and differed in their prerequisites only. For those rules that have no prerequisites (e.g. **`socks`**) I don't even need an implicit rule. The explicit ones can be pooled, because of their commands being all identic. Thus I get a shorter and pretty clear Makefile:

```
# An explicit rule assigns the commands for several targets
coat shoes mobile sweater socks trousers\
shirt pants undershirt: ;  @echo put on $@; touch $@

# Implicit rules state the prerequisites
coat:      shoes mobile sweater
shoes:     socks trousers
mobile:    trousers
sweater:   shirt
trousers:  pants shirt
shirt:     undershirt
```

User-defined variables and functions

Use **`=`** or **`:=`** to assign values to variables, depending on if potentially contained variables and functions should be expanded at *using* or *declaration* time. To retrieve the stored value, write **`$(myvar)`**. To have **`make`** executing the value (like a function), write **`$(call myvar)`**.

```
# Declaration of a variable
articles = coat shoes mobile sweater socks\
        trousers shirt pants undershirt

# An explicit rule assigns the commands for several targets
$(articles) :; @echo put on $@; touch $@

# Implicit rules state the prerequisites
coat:      shoes mobile sweater
shoes:     socks trousers
mobile:    trousers
sweater:   shirt
trousers:  pants shirt
shirt:     undershirt

# Additional feature
.PHONY: naked
naked:      ; @-rm $(articles)
```

BTW: The minus sign (**`-`**) in front of the **`rm`** causes **`make`** to ignore any errors that occure while doing **`rm`**. This way a **`make naked coat`** will end at the coat, even for people who're already naked at starting time.

Combining multiple Makefiles

Divide and konquer! For heavy projects (e.g. compiling the Linux kernel) you need giant Makefiles. They will become much clearer, when you split them into separate, smaller parts.

When **`make`** encounters an **`include`**-command, it will stop processing the current Makefile, read the included Makefile and then continue where it left off. If you don't want it to abort when the included Makefile's missing, just say **`-include Makefile(s)`**.

Example:

```
# tying the cravat
-include Makefile.tie
```

Double-colon rules

The following Makefile erroneously defines a target **`socks`** twice:

```
socks: ; @echo get into left sock
socks: ; @echo get into right sock
```

`make` will show up with a warning message and run the last target's command only:

```
$ make
Makefile:2: warning: overriding commands for target ‘socks’
Makefile:1: warning: ignoring old commands for target ‘socks’
get into right sock
```

Particularly when working with includes, you might want **`make`** to run the commands of any rule with a certain target name. For this to work, you just need to duplicate the colon:

```
socks:: ; @echo get into left sock
socks:: ; @echo get into right sock
```

It will work as expected:

```
$ make
get into left sock
get into right sock
```

Comparing timestamps

Here's a little Makefile I use for converting Postscript-files (**`tiger.ps`**) into PDF-files (**`tiger.pdf`**):

```
tiger.pdf: tiger.ps; ps2pdf $<
```

`make` will only call **`ps2pdf`** when **`tiger.pdf`** is non-existing or *older* than **`tiger.ps`**. BTW: **`make`** will automatically substitute the filename of the prerequisite (**`tiger.ps`**) for **`$<`**.

Here's what you get:

```
$ ls tig*
tiger.ps
$ make tiger.pdf
ps2pdf tiger.ps
$ ls tig*
tiger.pdf tiger.ps
$ make tiger.pdf
make: ‘tiger.pdf’ is up to date.
$
$ cp tiger2.ps tiger.ps
$ make tiger.pdf
ps2pdf tiger.ps
```

Patterns

Thanks to the pattern-notation I need not write a separate rule for each postscript file. Generalizing the last example:

```
%.pdf: %.ps; -ps2pdf $<
```

The **`make tiger.pdf`** will work as before. But now a simple **`make`** (no targets) won't work any more. **`make`** must be told a target's name.

```
$ make
make: *** No targets.  Stop.
$ make tiger.pdf
ps2pdf tiger.ps
```

The following Makefile will create the file **`tiger.pdf`** even when you call **`make`** without any parameters:

```
all  : tiger.pdf
%.pdf: %.ps; -ps2pdf $<
```

Now, this works fine for the tiger. What if I want to convert *any* files in the current directory? No problem - the **`$(wildcard)`** function can hand me a list of all **`*.ps`** files, and the **`$(patsubst)`**-function will change it to a list of PDF filenames:

```
all: $(patsubst %.ps,%.pdf,$(wildcard *.ps))
%.pdf: %.ps; -ps2pdf $<
```

Built-in functions

`$(subst from,to,text)`	Replace **`from`** with **`to`** in **`text`**.
`$(patsubst pattern,replace-ment,text)`	Replace words matching **`pattern`** with **`replacement`** in **`text`**.
`$(strip string)`	Remove excess whitespace characters from **`string`**.
`$(findstring find,text)`	Locate **`find`** in **`text`**.
`$(filter pattern...,text)`	Select words in **`text`** that match one of the **`pattern`** words.
`$(filter-out pattern...,text)`	Select words in **`text`** that do not match any of the **`pattern`** words.
`$(sort list)`	Sort the words in **`list`** lexicographically, removing dupli-cates.
`$(dir names...)`	Extract the directory part of each file **`name`**.
`$(notdir names...)`	Extract the non-directory part of each file **`name`**.
`$(suffix names...)`	Extract the suffix (the last dot and following characters) of each file **`name`**.
`$(basename names...)`	Extract the base name (name without suffix) of each file **`name`**.
`$(addsuffix suffix,names...)`	Append **`suffix`** to each word in **`names`**.
`$(addprefix prefix,names...)`	Prepend prefix to each word in **`names`**.
`$(join list1,list2)`	Join two parallel lists of words.
`$(word n,text)`	Extract the **`n`**th word (one-origin) of **`text`**.
`$(words text)`	Count the number of words in **`text`**.
`$(wordlist s,e,text)`	Returns the list of words in text from **`s`** to **`e`**.
`$(firstword names...)`	Extract the first word of **`names`**.
`$(wildcard pattern...)`	Find file names matching a shell file name pattern (not a '%' pattern).
`$(error text...)`	When this function is evaluated, **`make`** generates a fatal error with the message **`text`**.
`$(warning text...)`	When this function is evaluated, **`make`** generates a warning with the message **`text`**.
`$(shell command)`	Execute a shell command and return its output.
`$(origin variable)`	Return a string describing how the **`make`** variable **`variable`** was defined.
`$(foreach var,words,text)`	Evaluate **`text`** with **`var`** bound to each word in **`words`**, and concatenate the results.
`$(call var,param,...)`	Evaluate the variable var replacing any references to **`$(1),$(2)`** with the first, second, etc. param values.

Automatic variables

`$@`	The name of the target.
`$%`	The target member name, when the target is an archive member.
`$<`	The name of the first (or only) prerequisite.
`$?`	The names of all the prerequisites that are newer than the target, with spaces between them.
`$^` **`$+`**	The names of all the prerequisites, with spaces between them. The value of **`$^`** omits duplicate prerequisites, while **`$+`** retains them and preserves their order.
`$*`	The stem with which an implicit rule matches.

`$(@D)`
`$(@F)`
	The directory part and the file-within-directory part of **`$@`**

`$(*D)`
`$(*F)`
	The directory part and the file-within-directory part of **`$*`**

`$(%D)`
`$(%F)`
	The directory part and the file-within-directory part of **`$%`**

`$(<D)`
`$(<F)`
	The directory part and the file-within-directory part of **`$<`**

`$(^D)`
`$(^F)`
	The directory part and the file-within-directory part of **`$^`**

`$(+D)`
`$(+F)`
	The directory part and the file-within-directory part of **`$+`**

`$(?D)`
`$(?F)`
	The directory part and the file-within-directory part of **`$?`**

Links

- GNU **`make`** Manual