7 Things You Should Know About Make

*October 26, 2014 by Alexey Shmalko   
Tags:*[*make*](http://www.alexeyshmalko.com/tags/make/)

[Make](http://en.wikipedia.org/wiki/Make_(software)) is a simple and powerful tool for automatically building any files out of others. However, some programmers experience issues writing makefiles and reinvent things without knowing some basic Make things.

How make works

By default, make starts with the first target. This is called the default goal.

Make reads the makefile in the current directory and begins by processing the first rule. But before make can fully process this rule, it must process the rules for the files that rule depends on. Each of this files is processed according to its own rules.

Actually, that’s recursive algorithm for each target:

1. Find a rule to build that target. If there is no rule to build the target, make fails;
2. For each prerequisite of the target, run this algorithm with that prerequisite as the target;
3. If either the target does not exist, or if any prerequisite’s modification time is newer than the target’s modification time, run the recipe associated with the target. If the recipe fails, (usually) make fails.

Assignment types

Make supports variables to ease writing makefiles. They are assigned with one of the following operators: =, ?=, :=, ::=, +=, !=. The difference between them is the following:

* = assigns a deferred value to a variable. That means that value of the variable will be computed every time variable is used. Be aware of that when assigning the result of shell command—shell command will be executed every time variable is read.
* := and ::= are essentially the same. Such assignment computes variable value once and just stores it. Simple and powerful. This type of assignment should be your default choice.
* ?= works as := if the variable was not defined, otherwise does nothing.
* += is append operator. The right-hand side is considered immediate if the variable was previously set with := or ::=, and deferred otherwise.
* != is a shell assignment operator. The right-hand side is evaluated immediately and handed to the shell. The result is stored in the variable named on the left.

Pattern rules

If you have a lot of files that have the same rule, you can easily define a pattern rule that will be used for all matching targets. A pattern rule looks like an ordinary rule, except that its target contains the character ‘%’. The target is considered a pattern for matching file names; the ‘%’ can match any non-empty substring.

In my blog directory I have the next Makefile:

all: \

build/random-advice.html \

build/proactor.html \

build/awesome\_skype\_fix.html \

build/ide.html \

build/vm.html \

build/make.html \

build/%.html: %.md

Markdown.pl $^ > $@

$^ is an automatic variable that means dependencies while $@ means target; so this rule just passes my markdown files through the converter. To see more details on how to write pattern rules and more automatic variables [refer to the manual](http://www.gnu.org/software/make/manual/make.html#Pattern-Rules).

Default implicit rules

GNU Make has a set of default rules. So that in lots of cases you don’t have to write explicit rules. The list includes, but not limited to, rules for compiling C, C++, assembler programs and linking them. The full list is available at [make manual](https://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html).

It’s possible to don’t have Makefile at all. For example, you can just save source code of a program in the file called hello.c and then just invoke make hello. Make will automatically compile hello.o from hello.c and then link it in hello for you.

The recipes are defined in the form $(CC) $(CPPFLAGS) $(CFLAGS) -c. That makes it possible to change the rule by changing variables. To compile source files with clang just add the following line: CC := clang. In the directory I’m using for saving my small test programs I have a tiny Makefile:

CFLAGS := -Wall -Wextra -pedantic -std=c11

CXXFLAGS := -Wall -Wextra -pedantic -std=c++11

Wildcarding and functions

To compile all C and C++ source files in current directory, use the following code for dependencies: $(patsubst %.cpp,%.o,$(wildcard \*.cpp)) $(patsubst %.c,%.o,$(wildcard \*.c)).

wildcard searches for all files matching the pattern and patsubst replaces appropriate file extension with .o.

There are lots of functions for transforming text in make. They are called in form $(function arguments).

For the full list of functions refer to [the manual](http://www.gnu.org/software/make/manual/make.html#Functions).

Note that space after comma is considered as part of an argument. That may cause unexpected results for some functions, so I recommend not put space after comma at all.

It’s even possible to write your own functions with [call function](http://www.gnu.org/software/make/manual/make.html#Call-Function) and kind of parameterized templates with [eval function](http://www.gnu.org/software/make/manual/make.html#Eval-Function).

Search path

There is special make variable VPATH used as a PATH for all prerequisites. That is, in the VPATH variable, directory names are separated by colons or blanks. The order in which directories are listed is the order followed by make in its search. The rules may then specify the names of files in the prerequisite list as if they all existed in the current directory.

There is also a more fine-grained vpath directive. It allows you to specify the search path for every file matching the pattern. So, if you store all your headers in include directory, you can use the following line:

vpath %.h include

However, while make changes only prerequisite part of the rule and not the rule itself, you can’t rely on explicit file names in your rule. Instead, you must use *automatic variables* such as $^.

For more information on searching direcotries for prerequisites refer to [make manual](http://www.gnu.org/software/make/manual/make.html#Directory-Search).

Debugging Makefiles

There a couple of techniques to debug makefiles:

Printing

The first one is plain old printing. You can print the value of some expression using one of the following make functions: $(info ...) $(warning ...) $(error ...) Make will print the value of the expression once it passes through this line.

I believe you know how to use tracing.

### 6.5 Setting Variables

To set a variable from the makefile, write a line starting with the variable name followed by ‘=’, ‘:=’, or ‘::=’. Whatever follows the ‘=’, ‘:=’, or ‘::=’ on the line becomes the value. For example,

objects = main.o foo.o bar.o utils.o

defines a variable named objects. Whitespace around the variable name and immediately after the ‘=’ is ignored.

Variables defined with ‘=’ are recursively expanded variables. Variables defined with ‘:=’ or ‘::=’ are simply expanded variables; these definitions can contain variable references which will be expanded before the definition is made. See [The Two Flavors of Variables](https://www.gnu.org/software/make/manual/html_node/Flavors.html#Flavors).

The variable name may contain function and variable references, which are expanded when the line is read to find the actual variable name to use.

There is no limit on the length of the value of a variable except the amount of memory on the computer. You can split the value of a variable into multiple physical lines for readability (see [Splitting Long Lines](https://www.gnu.org/software/make/manual/html_node/Splitting-Lines.html#Splitting-Lines)).

Most variable names are considered to have the empty string as a value if you have never set them. Several variables have built-in initial values that are not empty, but you can set them in the usual ways (see [Variables Used by Implicit Rules](https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html#Implicit-Variables)). Several special variables are set automatically to a new value for each rule; these are called the automatic variables (see [Automatic Variables](https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables)).

If you’d like a variable to be set to a value only if it’s not already set, then you can use the shorthand operator ‘?=’ instead of ‘=’. These two settings of the variable ‘FOO’ are identical (see [The origin Function](https://www.gnu.org/software/make/manual/html_node/Origin-Function.html#Origin-Function)):

FOO ?= bar

and

ifeq ($(origin FOO), undefined)

FOO = bar

endif

The shell assignment operator ‘!=’ can be used to execute a shell script and set a variable to its output. This operator first evaluates the right-hand side, then passes that result to the shell for execution. If the result of the execution ends in a newline, that one newline is removed; all other newlines are replaced by spaces. The resulting string is then placed into the named recursively-expanded variable. For example:

hash != printf '\043'

file\_list != find . -name '\*.c'

If the result of the execution could produce a $, and you don’t intend what follows that to be interpreted as a make variable or function reference, then you must replace every $ with$$ as part of the execution. Alternatively, you can set a simply expanded variable to the result of running a program using the shell function call. See [The shell Function](https://www.gnu.org/software/make/manual/html_node/Shell-Function.html#Shell-Function). For example:

hash := $(shell printf '\043')

var := $(shell find . -name "\*.c")

As with the shell function, the exit status of the just-invoked shell script is stored in the .SHELLSTATUS variable.