# Automatic Variables (GNU make)

---

### 10.5.3 Automatic Variables

Suppose you are writing a pattern rule to compile a '`.c`' file into a '`.o`' file: how do you write the '`cc`' command so that it operates on the right source file name? You cannot write the name in the recipe, because the name is different each time the implicit rule is applied.

What you do is use a special feature of `make`, the *automatic variables*. These variables have values computed afresh for each rule that is executed, based on the target and prerequisites of the rule. In this example, you would use '`$@`' for the object file name and '`$<`' for the source file name.

It's very important that you recognize the limited scope in which automatic variable values are available: they only have values within the recipe. In particular, you cannot use them anywhere within the target list of a rule; they have no value there and will expand to the empty string. Also, they cannot be accessed directly within the prerequisite list of a rule. A common mistake is attempting to use `$@` within the prerequisites list; this will not work. However, there is a special feature of GNU `make`, secondary expansion (see Secondary Expansion), which will allow automatic variable values to be used in prerequisite lists.

Here is a table of automatic variables:

`$@`

The file name of the target of the rule. If the target is an archive member, then '`$@`' is the name of the archive file. In a pattern rule that has multiple targets (see Introduction to Pattern Rules), '`$@`' is the name of whichever target caused the rule's recipe to be run.

`$%`

The target member name, when the target is an archive member. See Using `make` to Update Archive Files. For example, if the target is `foo.a(bar.o)` then '`$%`' is `bar.o` and '`$@`' is `foo.a`. '`$%`' is empty when the target is not an archive member.

`$<`

The name of the first prerequisite. If the target got its recipe from an implicit rule, this will be the first prerequisite added by the implicit rule (see Using Implicit Rules).

`$?`

The names of all the prerequisites that are newer than the target, with spaces between them. If the target does not exist, all prerequisites will be included. For prerequisites which are archive members, only the named member is used (see Using `make` to Update Archive Files).

'`$?`' is useful even in explicit rules when you wish to operate on only the prerequisites that have changed. For example, suppose that an archive named `lib` is supposed to contain copies of several object files. This rule copies just the changed object files into the archive:

```
lib: foo.o bar.o lose.o win.o
        ar r lib $?
```

`$^`

The names of all the prerequisites, with spaces between them. For prerequisites which are archive members, only the named member is used (see Using make to Update Archive Files). A target has only one prerequisite on each other file it depends on, no matter how many times each file is listed as a prerequisite. So if you list a prerequisite more than once for a target, the value of `$^` contains just one copy of the name. This list does **not** contain any of the order-only prerequisites; for those see the '`$|`' variable, below.

`$+`

This is like '`$^`', but prerequisites listed more than once are duplicated in the order they were listed in the makefile. This is primarily useful for use in linking commands where it is meaningful to repeat library file names in a particular order.

`$|`

The names of all the order-only prerequisites, with spaces between them.

`$*`

The stem with which an implicit rule matches (see How Patterns Match). If the target is `dir/a.foo.b` and the target pattern is `a.%.b` then the stem is `dir/foo`. The stem is useful for constructing names of related files.

In a static pattern rule, the stem is part of the file name that matched the '`%`' in the target pattern.

In an explicit rule, there is no stem; so '`$*`' cannot be determined in that way. Instead, if the target name ends with a recognized suffix (see Old-Fashioned Suffix Rules), '`$*`' is set to the target name minus the suffix. For example, if the target name is '`foo.c`', then '`$*`' is set to '`foo`', since '`.c`' is a suffix. GNU `make` does this bizarre thing only for compatibility with other implementations of `make`. You should generally avoid using '`$*`' except in implicit rules or static pattern rules.

If the target name in an explicit rule does not end with a recognized suffix, '`$*`' is set to the empty string for that rule.

Of the variables listed above, four have values that are single file names, and three have values that are lists of file names. These seven have variants that get just the file's directory name or just the file name within the directory. The variant variables' names are formed by appending '`D`' or '`F`', respectively. The functions `dir` and `notdir` can be used to obtain a similar effect (see Functions for File Names). Note, however, that the '`D`' variants all omit the trailing slash which always appears in the output of the `dir` function. Here is a table of the variants:

'`$(@D)`'

The directory part of the file name of the target, with the trailing slash removed. If the value of '`$@`' is `dir/foo.o` then '`$(@D)`' is `dir`. This value is `.` if '`$@`' does not contain a slash.

'`$(@F)`'

The file-within-directory part of the file name of the target. If the value of '`$@`' is `dir/foo.o` then '`$(@F)`' is `foo.o`. '`$(@F)`' is equivalent to '`$(notdir $@)`'.

'`$(*D)`'
'`$(*F)`'

The directory part and the file-within-directory part of the stem; `dir` and `foo` in this example.

‘$(%D)’
‘$(%F)’

The directory part and the file-within-directory part of the target archive member name. This makes sense only for archive member targets of the form *archive*(*member*) and is useful only when *member* may contain a directory name. (See Archive Members as Targets.)

‘$(<D)’
‘$(<F)’

The directory part and the file-within-directory part of the first prerequisite.

‘$(^D)’
‘$(^F)’

Lists of the directory parts and the file-within-directory parts of all prerequisites.

‘$(+D)’
‘$(+F)’

Lists of the directory parts and the file-within-directory parts of all prerequisites, including multiple instances of duplicated prerequisites.

‘$(?D)’
‘$(?F)’

Lists of the directory parts and the file-within-directory parts of all prerequisites that are newer than the target.

Note that we use a special stylistic convention when we talk about these automatic variables; we write “the value of ‘$<’”, rather than “the variable <” as we would write for ordinary variables such as `objects` and `CFLAGS`. We think this convention looks more natural in this special case. Please do not assume it has a deep significance; ‘$<’ refers to the variable named < just as ‘$(CFLAGS)’ refers to the variable named `CFLAGS`. You could just as well use ‘$(<)’ in place of ‘$<’.