Environment variable expansion occurs when the command is read

devblogs.microsoft.com/oldnewthing/20060823-00

August 23, 2006

Raymond Chen

On the command line (and in batch files), environment variable expansion occurs when the command is read. This sounds obvious at first, but it has its own consequences.

In the online documentation for `SET`, one such consequence is spelled out:

```
set VAR=before
if "%VAR%" == "before" (
    set VAR=after
    if "%VAR%" == "after" @echo If you see this, it worked
)
```

would never display the message, since the `%VAR%` in *both* “ `if` ” statements is substituted when the first “ `if` ” statement is read, since it logically includes the body of the “ `if` “, which is a compound statement.

In other words, the “ `if` ” command is not complete until the closing parenthesis is read. You can see this if you type the commands interactively:

```
C:\>set VAR=before
C:\>if "%VAR%" == "before" (
More? set VAR=after
More? if "%VAR%" == "after" @echo If you see this, it worked
More? )
C:\>
```

Notice that the “ `if` ” command didn’t execute until you closed the parenthesis; the command interpreter kept prompting “More?” to collect the body of the “ `if` “. This means that everything you type as the body of the “ `if` ” is evaluated **before the “ `if` ” condition or any of the lines in the body are evaluated**. It’s as if you had typed

```
C:\>if "before" == "before" (
More? set VAR=after
More? if "before" == "after" @echo If you see this, it worked
More? )
```

Note that this is different from most UNIX shells, which do not expand environment variables until the enclosing command is executed. For example,

```
$ var=before
$ var=after; echo $var
after
```

Notice that the `$x` is not expanded until the `echo` command's arguments are being computed. The analogous commands in the Windows command interpreter result in something quite different:

```
C:\>set VAR=before
C:\>set VAR=after & echo %VAR%
before
```

That's because the command interpreter expanded the environment variables at the time the line was read (not at the time the line is executed), yielding

```
set VAR=after & echo before
```

As a result, the old value of `VAR` is echoed. Some people treat this as a feature, allowing them to "restore" a variable without having to save it anywhere:

```
set VAR=newvalue & call helper.cmd & set VAR=%VAR%
```

This command sets the `VAR` variable to a new value, calls `helper.cmd` (which presumably uses the value of the `%VAR%` variable to control its behavior), then magically restores the variable to its original value since the `%VAR%` is expanded early, producing the old value.

But what if you want the variable to be expanded at execution time rather than at parse time? For that, you use "delayed expansion", which is enabled by the `/V` command line option or by using the `SETLOCAL ENABLEDELAYEDEXPANSION` command in a batch file.

```
C:\> copy con "%TEMP%\helper.cmd"
SETLOCAL ENABLEDELAYEDEXPANSION
set VAR=before
set VAR=after & echo immediate:%VAR%, delayed:!VAR!
ENDLOCAL
^Z
        1 file(s) copied.
C:\> "%TEMP%\helper.cmd"
C:\>SETLOCAL ENABLEDELAYEDEXPANSION
C:\>set VAR=before
C:\>set VAR=after   & echo immediate:before, delayed:!VAR!
immediate:before, delayed:after
C:\>ENDLOCAL
```

Immediate expansion is performed with percent signs, whereas delayed expansion is performed with exclamation points.

Why is immediate expansion the default? Because prior to Windows NT, that was the only type of expansion supported by the command interpreter. Retaining immediate expansion as the default preserved backwards compatibility with existing batch files. (The original command interpreter was written in assembly language. You really didn't want to be too clever or it would make your brain hurt trying to maintain the code. An interpreter loop of the form "Read a line, expand environment variables, evaluate" was therefore simple and effective.)

Armed with this understanding of immediate versus delayed expansion, perhaps you can explain what is really going on here. (Hint: It has nothing to do with `ERRORLEVEL`.)

Raymond Chen

Follow