Windows Programming/Programming CMD

In Windows NT (XP, Vista, 7, 8, 10,...) one is able to write batch files that are interpreted by the Command Prompt (cmd.exe). They can be used to automate file-system tasks such as backups or basic installations and can be used with other command-line utilities as well. The batch files can be considered to be a simple scripting language with logic and jumps. The advantages of using batch files are the ease of writing them, the ability to edit the files without compiling anything, their cross-compatibility across Windows NT Operating Systems and their inherent ability to manipulate file systems due to their basis on MS-DOS. Batch file scripts are not case-sensitive, although strings and data are. The file is controlled by a list of commands separated into lines which are run like normal commands would be at the Command Prompt, although some functionality is different. Batch files can be run from Windows Explorer but the console used to display them closes automatically at the end of the batch file, so a command at the end that prevents instantaneous exit is needed for any remaining output to be read before the console window closes. Although the batch files are able to manipulate their environment, such as color settings and environment variables, the changes are all temporary, as in a standard Command Prompt session. Color settings, however, are retained on later editions of Windows NT. In order to try to learn about batch files, it is useful to understand Command Prompt commands. See: Guide to Windows commands.

Batch File

The script is kept inside a batch file, with the extension .bat or .cmd. Although .bat is more recognisable, as it was used in the MS-DOS environment that preceded the Command Prompt, the Command Prompt's interpretations of batch files is very different to the manner of interpreting DOS batch files, and .cmd files are only interpreted by the Command Prompt, so using the .cmd extension prevents mis-use in older environments.

Execution starts at the top of the file and ends at the end. When the end of the file is reached, the file exits to the Command Prompt if it was invoked from there, or the console window closes if it was invoked from Windows Explorer or the START command.

ECHO Command

Typically, batch files start with the 'echo off' command, which stops the input and prompt from being displayed during execution, so only the command output is displayed. The '@' symbol prevents a command from having input and its prompt displayed. It is used on the 'echo off' command to prevent that first command from displaying the input and prompt:

```
@ECHO OFF
```

In order to print lines, the ECHO command is used again, but this time with a text parameter other than 'off':

```
ECHO.Hello World!
```

The period ('.') is used to prevent confusion with the attempt to output ON or OFF rather than turn input and prompt displaying on and off. The last line in a code should then be:

```
ECHO ON
```

'Echo on' will turn the input/prompt display back on, just in case the program exits to the Command Prompt where without the command 'echo on', there will be no visible prompt left for use.

HINT: Starting with Windows XP, the ECHO ON command is optional. The command interpreter automatically enables it after the BAT file terminates.

Hello World Example

Using the code above, we can make a hello world program like so:

```
@ECHO OFF
ECHO Hello World!
ECHO ON
```

Comments

In batch files there are two ways of writing comments. Firstly there is the form:

```
REM Comment here.
```

This form is included as it was in the MS-DOS batch file script. The other form is this:

```
::Comment here.
```

This form is generally favoured, for being faster to execute and write, and also for being easy to differentiate from normal commands. For this type of comment only two double-colons ('::') are needed and the comment ends at the end of the line. Batch files have no multi-line comment types.

You can also add a comment to the end of the command:

```
command &::Comment here.
```

Variables

In batch scipting, all variables are strings stored in system memory. Variables are assigned using the SET command. Variables assigned in this manner persist in the System environment alongside predefined Global System variables until the environment in which they are defined is terminated, either through the use of the ENDLOCAL command or the closing of the command prompt session. Care should be taken to avoid 'overwriting' Global environment variables such as 'PATH' 'TIME' 'DATE', as other programs can be dependant on them having their 'proper' value.

- The use of the command 'GOTO :EOF' in an environment where the command 'SETLOCAL' has created a child environment implies and effects an 'ENDLOCAL' command.

```
SET name=John Smith
```

This command creates an environment variable called name, and sets its value to the string "John Smith". White space on either side of the first '=' sign is included in the assignment of both the variables reference name ('name' in this instance) and the variables value [Variable names can contain whitespace!]. Trailing whitespace in a variables value is also assigned, which can cause unexpected failure in some scripts if not detected. For this reason, it is recommended to doubleqoute the assignment of variables:

```
SET "name=John Smith"
```

Variables are generally used within other commands by expanding the reference string with '%' characters marking the beginning and end of the variable reference string:

```
ECHO %name%
```

In batch scripting, variables that are expanded using '%' expansion are read [parsed] by the interpreter **prior** to the line or block being executed. As such, this means the command will execute with the value the variable held prior to the commencement of that section of code. This is especially significant where code blocks are used and the variables value is reassigned within that code block. A code block in batch scripting occurs when multiple commands are chained together on the same line using '&' to concatenate the commands; as well as within any parenthesised code such as 'IF' statements and 'FOR' loops.

An example that demonstrates the problem this poses:

```
@Echo off
Set "count=0"
For /L %%n in (1 1 3)Do (
 Set /A count+=1
 Echo %count%
)
Echo %count%
```

Output:

```
0
0
0
3
```

There are 2 methods for resolving this. The first and most commonly used is to Enable Delayed Expansion, which allows variables to be expanded using '!' (delayed) expansion:

```
@Echo off
Set "count=0"
SETLOCAL EnableExtensions EnableDelayedExpansion
For /L %%n in (1 1 3)Do (
 Set /A count+=1
 Echo !count!
)
Echo %count%
```

Output:

```
1
2
3
3
```

Using Delayed Expansion results in the variable being expanded during execution instead of when the command is initially parsed by the interpreter.

The 2nd method of expanding a variables current value during code blocks is to use the 'CALL' command to effectively reset the parser during execution of the command so the current value is read.

```
@Echo off
Set "count=0"
For /L %%n in (1 1 5)Do (
 Set /A count+=1
 Call Echo %%count%%
)
Echo %count%
```

The call method is significantly slower, However is useful for dealing with strings that may include '!' characters that the interpreter would attempt to parse as a variable when delayed expansion is enabled. It is not suitable for strings containing Carets '^' as the call command doubles Carets that occur in the command string that follows the Call.

The use of Delayed expansion allows 'Associated' variables to be defined using a common reference name with unique indexes to emulate arrays. This is possible because while Delayed Expansion enabled, expansion of variables occurs in two steps. First, '%' variables are expanded, then '!' variables are expanded. Example:

```
@Echo off
 Set "str[1]=one,three,five"
 Set "str[2]=two,four,six"
 Setlocal EnableExtensions EnableDelayedExpansion
rem for each in 1 2
 For %%i in (1 2)Do (
rem reset sub index variable
  Set "{i}=0"
rem for each in variable str[index]
  For %%G in (!str[%%i]!)Do (
rem increment sub index count
   Set /A {i}+=1
rem define element '{i}' from str[index] to str[index][subindex]
   Set "str[%%i][!{i}!]=%%G"
  )
 )
Set Str
Goto :Eof
```

Output:

```
str[1]=one,three,five
str[1][1]=one
str[1][2]=three
str[1][3]=five
str[2]=two,four,six
str[2][1]=two
str[2][2]=four
str[2][3]=six
```

Input

The set command can also be used for input:

```
SET /P var=Enter a value for var:
```

This command displays "Enter a value for var:" and when the user enters the data, var is given that value.

Be aware, if the user presses enter without entering anything then the value in var is unchanged, so for the sake of a prompt it is often best to give a default value, or clear the value for the variable first if it has been used before:

```
SET var=
SET /P var=Enter a value for var:
```

Below is an example:

```
@ECHO OFF
SET /P answer= Enter name of file to delete:
DEL /P %answer%
ECHO ON
```

This batch file gets the name of a file to delete and then uses the DEL command with the prompt parameter '/P' to ask the user if they're sure they want to delete the file.

Flow Control

Conditionals

IF

The IF command can be used to create program logic in batch files. The IF command allows three basic checks, on the ERRORLEVEL, the equality of two strings, and the existence of a file or folder. The first check on the ERRORLEVEL will check to see if it is greater than or equal to a certain number:

```
IF ERRORLEVEL 5 ECHO.The ERRORLEVEL is at least 5.
```

For this style the first parameter is always ERRORLEVEL, and the second is the value it checks against. In this case, if the ERRORLEVEL is at least 5 then the command at the end of the line is executed, outputting the message "The ERRORLEVEL is at least 5.". The second form is a check between two strings:

```
IF "%str1%"=="Hello." ECHO.The strings are equal.
```

Here the first parameter is two strings either side of the double '=', symbolising a check to see if they are equal. If the variable str1 is exactly equal to "Hello.", a check which is case-sensitive, then "The strings are equal." is outputted. In the case that you wish to make the check case-insensitive you would rewrite it as following:

```
IF /I "%str1%"=="Hello." ECHO.The strings are equal.
```

Now, for example, str1 could contain "HELLO." but the check would still result in the command being executed at the end as the check is now case-insensitive. The final basic IF type is the existence check, to see if a file or folder exists.

```
IF EXIST myfile.txt TYPE myfile.txt
```

Here if the file "myfile.txt" exists in the current folder then the command `TYPE myfile.txt` is executed which displays the contents of "myfile.txt" in the console window.

All of the preceding examples have an optional NOT parameter that can be written after the IF which will execute the command at the end of the line if the condition is *not* true. For example:

```
IF NOT EXIST myfile.txt ECHO.File missing.
```

Which will output "File missing." if the file "myfile.txt" is not existent in the current folder. There are a few other IF types with command extensions, which can be seen with the `IF /?` command at the command prompt.

ELSE

The ELSE operator can be used with a combination of brackets to provide multi-line logical statements that provide an alternative set of commands if the condition is not true.

```
IF condition (
   commands to be executed if the condition is true
) ELSE (
   commands to be executed if the condition is false
)
```

Unlike some languages, in batch files the scripting requires that the lines `IF condition (`, `) ELSE (` and `)` are written very specifically like that. It is possible, however, to re-write it to use single-line outcomes all on one line:

```
IF condition (command if true) ELSE command if false
```

Below is an example of the ELSE operator in use:

```
@ECHO OFF
::Prompt for input.
SET /P answer=Enter filename to delete:
IF EXIST %answer% (
 DEL /P %answer%
) ELSE (
 ECHO.ERROR: %answer% can not be found in this folder!
)
ECHO ON
```

This batch file will delete a file, unless it doesn't exist in which case it will tell you with the message "ERROR: %answer% can not be found in this folder!".

Unlike in most computer languages, multiple multi-line IF...ELSE style statements can't be nested in batch files.

Jumps

You can control program flow using the GOTO statement. Batch files don't have all elements for structured programming scripting, however some elements of structured programming such as functions can be simulated. The simplest way of controlling program flow, however, is the GOTO statement which jumps to a specified label.

```
GOTO labelnam
```

This code will direct program flow to the label labelnam, which is found at the first occurrence of this line:

```
:labelnam
```

It is important to remember that labels only store 8 characters, so if a label is longer than 8 characters only the first 8 will be seen. This means the labels labelname1 and labelname2 can't be distinguished from each other as their only difference occurs past the first 8 characters. Although not strictly incorrect, it is better to avoid using label names longer than 8 characters to avoid these distinguishing problems easily.

Here is the example from earlier redesigned to loop until asked to stop:

```
@ECHO OFF

:prompt
::Clear the value of answer ready for use.
SET answer=
SET /P answer=Enter filename to delete (q to quit):

IF EXIST %answer% (
 DEL /P %answer%
 GOTO prompt
)
IF /I "%answer%"=="q" GOTO :EOF

::By this point an error must have occurred as all
::the correct entries have already been dealt with.
ECHO.ERROR: Incorrect entry!
GOTO prompt

ECHO ON
```

Take note of the command `GOTO :EOF`. This command will take the script to the end of the file and end the current batch script.

FOR Looping

Runs a specified command for each file in a set of files.

```
   FOR %variable IN (set) DO command [command-parameters]
```

```
 %variable  Specifies a single letter replaceable parameter.
 (set)      Specifies a set of one or more files.  Wildcards may be used.
 command    Specifies the command to carry out for each file.
 command-parameters
            Specifies parameters or switches for the specified command.
```

To use the FOR command in a batch program, specify %%variable instead of %variable. Variable names are case sensitive, so %i is different from %I.

Batch File Example:

```
for %%F IN (*.txt) DO @echo %%F
```

This command will list all the files ending in .txt in the current directory.

If Command Extensions are enabled, the following additional forms of the FOR command are supported:

```
   FOR /D %variable IN (set) DO command [command-parameters]
```

If set contains wildcards, then specifies to match against directory names instead of file names.

```
   FOR /R [[drive:]path] %variable IN (set) DO command [command-parameters]
```

Walks the directory tree rooted at [drive:]path, executing the FOR statement in each directory of the tree. If no directory specification is specified after /R then the current directory is assumed. If set is just a single period (.) character then it will just enumerate the directory tree.

```
   FOR /L %variable IN (start,step,end) DO command [command-parameters]
```

The set is a sequence of numbers from start to end, by step amount. So (1,1,5) would generate the sequence 1 2 3 4 5 and (5,-1,1) would generate the sequence (5 4 3 2 1)

```
   FOR /F ["options"] %variable IN (file-set) DO command [command-parameters]
   FOR /F ["options"] %variable IN ("string") DO command [command-parameters]
   FOR /F ["options"] %variable IN ('command') DO command [command-parameters]
```

or, if usebackq (or useback) option present:

```
   FOR /F ["options"] %variable IN ("file-set") DO command [command-parameters]
   FOR /F ["options"] %variable IN ('string') DO command [command-parameters]
   FOR /F ["options"] %variable IN (`command`) DO command [command-parameters]
```

(The purpose of usebackq is to use a fullname of file-set including the space.)

filenameset is one or more file names. Each file is opened, read and processed before going on to the next file in filenameset. Processing consists of reading in the file, breaking it up into individual lines of text and then parsing each line into zero or more tokens. The body of the for loop is then called with the variable value(s) set to the found token string(s). By default, /F passes the first blank separated token from each line of each file. Blank lines are skipped. You can override the default parsing behavior by specifying the optional "options" parameter. This is a quoted string which contains one or more keywords to specify different parsing options. The keywords are:

```
   eol=c           - specifies an end of line comment character
                     (just one)
   skip=n          - specifies the number of lines to skip at the
                     beginning of the file.
   delims=xxx      - specifies a delimiter set.  This replaces the
                     default delimiter set of space and tab.
   tokens=x,y,m-n  - specifies which tokens from each line are to
```

```
                  be passed to the for body for each iteration.
                  This will cause additional variable names to
                  be allocated.  The m-n form is a range,
                  specifying the mth through the nth tokens.  If
                  the last character in the tokens= string is an
                  asterisk, then an additional variable is
                  allocated and receives the remaining text on
                  the line after the last token parsed.
usebackq        - specifies that the new semantics are in force,
                  where a back quoted string is executed as a
                  command and a single quoted string is a
                  literal string command and allows the use of
                  double quotes to quote file names in
                  filenameset.
```

Some examples might help:

```
FOR /F "eol=; tokens=2,3* delims=, " %i in (myfile.txt) do @echo %i %j %k
```

would parse each line in myfile.txt, ignoring lines that begin with a semicolon, passing the 2nd and 3rd token from each line to the for body, with tokens delimited by commas and/or spaces. Notice the for body statements reference %i to get the 2nd token, %j to get the 3rd token, and %k to get all remaining tokens after the 3rd. For file names that contain spaces, you need to quote the filenames with double quotes. In order to use double quotes in this manner, you also need to use the usebackq option, otherwise the double quotes will be interpreted as defining a literal string to parse.

%i is explicitly declared in the for statement and the %j and %k are implicitly declared via the tokens= option. You can specify up to 26 tokens via the tokens= line, provided it does not cause an attempt to declare a variable higher than the letter 'z' or 'Z'. Remember, FOR variables are single-letter, case sensitive, global, and you can't have more than 52 total active at any one time.

You can also use the FOR /F parsing logic on an immediate string, by making the filenameset between the parenthesis a quoted string, using single quote characters. It will be treated as a single line of input from a file and parsed.

Finally, you can use the FOR /F command to parse the output of a command. You do this by making the filenameset between the parenthesis a back quoted string. It will be treated as a command line, which is passed to a child CMD.EXE and the output is captured into memory and parsed as if it was a file. So the following example:

```
FOR /F "usebackq delims==" %i IN (`set`) DO @echo %i
```

would enumerate the environment variable names in the current environment.

In addition, substitution of FOR variable references has been enhanced. You can now use the following optional syntax:

```
%~I         - expands %I removing any surrounding quotes (")
%~fI        - expands %I to a fully qualified path name
%~dI        - expands %I to a drive letter only
%~pI        - expands %I to a path only
%~nI        - expands %I to a file name only
%~xI        - expands %I to a file extension only
%~sI        - expanded path contains short names only
%~aI        - expands %I to file attributes of file
%~tI        - expands %I to date/time of file
%~zI        - expands %I to size of file
%~$PATH:I   - searches the directories listed in the PATH
              environment variable and expands %I to the
              fully qualified name of the first one found.
              If the environment variable name is not
```

```
                 defined or the file is not found by the
                 search, then this modifier expands to the
                 empty string
```

The modifiers can be combined to get compound results:

```
    %~dpI       - expands %I to a drive letter and path only
    %~nxI       - expands %I to a file name and extension only
    %~fsI       - expands %I to a full path name with short names only
    %~dp$PATH:I - searches the directories listed in the PATH
                  environment variable for %I and expands to the
                  drive letter and path of the first one found.
    %~ftzaI     - expands %I to a DIR like output line
```

In the above examples %I and PATH can be replaced by other valid values. The %~ syntax is terminated by a valid FOR variable name. Picking upper case variable names like %I makes it more readable and avoids confusion with the modifiers, which are not case sensitive.

Pipes

this is mainly used to redirect the output of one program to another program

```
a | b
```

means execute "a" and what all output "a" gives to the console - give that as "b" s input

```
dir | find ".htm"
```

will give a list of file which contain ".htm" in their names

The output of a command as well as possible errors could be redirected also to files (Note the 2 in front of the >>):

```
ACommand >>TheOutputOfTheCommandLogFile.log 2>>TheErrorOutputOfTheCommandFile.log
```

Functions

Functions may be simulated by using labels to control flow of execution, and using an environment variable to return the resulting return value. Labels can be defined at any place in the script and do not need to be referenced. The code following the label will be executed when encountered. Code blocks identified by labels are usually most conveniently placed after the main script has exited (at the end of the file) so that they will not run accidentally, but will be reached only when targeted by a GOTO or CALL statement.

The subroutine construct works because of the following:

- Code identified by a label can be jumped to by using the GOTO built in command
- Using the CALL built in command with a label creates a new invocation of the command processor for the same script, with an implicit GOTO the label
- Using the expression GOTO :EOF closes the current command processor invocation (Same as EXIT /B except the latter allows to specify an ERRORLEVEL)

The structure of a subroutine call looks like this:

```
CALL :subroutine1 param1 param2 ...
ECHO %result% was returned from subroutine1

CALL :subroutine2 param1 param2 ...
ECHO %result% was returned from subroutine2

GOTO :EOF
REM The above line ends the main invocation of the command processor and so exits the script

:subroutine1
  SETLOCAL
  commands using parameters %1, %2, .... and setting %retval%
  ENDLOCAL & SET result=%retval%
  GOTO:EOF
  REM The above line ends the child invocation of the command processor and so returns to just after CALL
subroutine1 in the main script

:subroutine2
  SETLOCAL
  commands using parameters %1, %2, .... and setting %retval%
  ENDLOCAL & SET result=%retval%
  GOTO:EOF
  REM The above line ends the child invocation of the command processor and so returns to just after CALL
subroutine2 in the main script
```

Bat file with next content outputs "42" as result of execution

```
:: describes and calls function for multiplication with 2 arguments
@ECHO OFF
CALL :multiply 21 2
ECHO %result%

:multiply
SETLOCAL
set retval=0
set left=%1
set right=%2
:: use '/A' for arithmetic
set /A "retval=left*right"
ENDLOCAL & SET result=%retval%
GOTO :EOF
```

Note:

- Using CALL with a label or the expression CALL :EOF or EXIT /B requires command extensions to be enabled.
- A label is a line consisting of a valid name (not containing any separators such as space or semicolon) prefixed by a colon.

Command-Line Interfacing

let's say we want to call a program "MyProgram" from the command prompt. we type the following into our prompt (the .exe file extension is unnecessary):

```
C:\>myprogram.exe
```

And this will run the myprogram executable. Now, let's say we want to pass a few arguments to this program:

```
C:\>myprogram arg1 arg2 arg3
```

Now, if we go into the standard main function, we will have our argc and argv values:

```
int main(int argc, char *argv[])
```

Where:

```
argc = 4
argv[0] = "myprogram" (the name of the program - deduct 1 from argc to get the number of arguments)
argv[1] = "arg1"
argv[2] = "arg2"
argv[3] = "arg3"
```

This shouldn't come as a big surprise to people who have any familiarity with standard C programming. However, if we translate this to our WinMain function, we get a different value:

```
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrev, LPSTR CmdLine, int iCmdShow)
```

we will only have one value to accept the command line:

```
CmdLine = "myprogram arg1 arg2 arg3".
```

We can also use the function **GetCommandLine** to retrieve this string from any point in the application. If we want to parse this into a standard argv/argc pair, we can use the function **CommandLineToArgvW** to perform the conversion. It is important to note that CommandLineToArgvW only works on unicode strings.

When we return a value from our C program, that value gets passed to the CMD shell, and stored in a variable called "ERRORLEVEL". ERRORLEVEL is the only global variable that is not a string, and it can contain a number from 0 to 255. By convention, a value of zero means "success", while a value other then zero signifies an error.

Let's say we wanted to write a C program that returns the number of arguments passed to it. This might sound like a simple task in C, but it is difficult to accomplish in batch script:

```
int main(int argc, char *argv[])
{
   return (argc - 1);
}
```

And we will name this program "CountArgs.exe". Now, we can put this into a batch script, to pass it a number of arguments, and to print out the number passed:

```
countargs.exe %*
ECHO %ERRORLEVEL%
```

We can, in turn, call this script "count.bat", and run that from a command prompt:

```
C:\>count.bat arg1 arg2 arg3
```

and running this will return the answer: 3.

NOTE: Actually, this can be accomplished with a batch file without resorting to a C program, by simply using CMD delayed variable expansion via the "/V:ON" parameter:

```
/V:ON   Enable delayed environment variable expansion using ! as the
delimiter. For example, /V:ON would allow !var! to expand the variable
var at execution time.  The var syntax expands variables at input time,
which is quite a different thing when inside of a FOR loop.
```

Then use a simple batch file like the following to count parameters:

```
set COUNT=0
for %%x in (%*) do ( set /A COUNT=!COUNT!+1 )
echo %COUNT%
```

Or an easier way, without having to enable & use 'delayed environment expansion', would be to do the following:

```
set COUNT=0
for %%x in (%*) do set /A COUNT+=1
echo COUNT = %COUNT%
```

This returns the same answer as the C program example.

Console Control Handlers