Windows Inspired
Confessions of a Software Developer

« Everything You Need to Know About Command Lines for Windows Programs
How a Windows Program Splits Its Command Line Into Individual Arguments »

A Better Way To Understand Quoting and Escaping of Windows Command Line Arguments

I'd like to have an argument, please
— Michael Palin

This post presents a better way to understand the quoting and escaping of Windows command line arguments.

A Problem Scenario

In order for you to appreciate why it's important to understand the things we're discussing, I'm going to start with a typical problem scenario that illustrates the kind of things that can go wrong.

We know from the Microsoft documentation for CreateProcess that command lines are split into individual arguments based on the location of spaces on the command line. We're told to enclose arguments that contain spaces between a pair of double quote characters. The typical description of how whitespace (space and tab) and quotes are dealt with while parsing a command line uses terms like *double-quoted string* and *inside-* or *outside a quoted part*. Even the Microsoft description of how a command line is parsed says *a string surrounded by double quotation marks ("string") is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.*

We're led to believe that analyzing a command line is just a matter of looking for matching pairs of quote characters that enclose individual arguments or embedded quotes. It sounds simple enough and we're pretty sure we understand how it works. But still, occasionally we come across an example we can't explain, or (more likely), some command line breaks an existing program or script.

Suppose we have a program, CreateDocs.exe, that generates documentation for a set of C++ source files. It's driven by a batch file that accepts a starting directory and an optional "VERBOSE" switch. The batch file first passes the starting directory to the program, then generates a list of subdirectories which it also passes one at a time to the program. Our batch file faithfully quotes each directory in case it contain spaces. Recall that %~1 is the first batch argument (%1) with any existing quotes removed, so ["%~1"] unconditionally double quotes the string whether it was quoted or not on the command line:

```
1  @echo off
2  REM %1 is the root of the directory tree to process.
3  REM %2 is the optional string VERBOSE
```

Search Site

Recent Posts
- How to Step Through an Executable Program's Startup Code in Visual Studio
- How to Add a Debugging Console to a Windows GUI Application
- How a Windows Program Splits Its Command Line Into Individual Arguments
- A Better Way To Understand Quoting and Escaping of Windows Command Line Arguments
- Everything You Need to Know About Command Lines for Windows Programs
- Humble Beginnings
- Welcome to Windows Inspired

Categories
- Tools and Technologies
- Just For Fun
- Tutorials
- Welcome

Credits

```
4  cls
5  setlocal
6  echo.
7  IF '%1'=='' echo No Path Specified& goto:eof
8  set ROOT_PATH=%~1
9
10 REM Process root directory:
11 CreateDocs "%ROOT_PATH%" %2
12
13 REM Process subdirectories:
14 FOR /F %%S IN ('dir /ad /b "%ROOT_PATH%"') DO (
15    CreateDocs "%%S" %2
16    )
17 echo.
18 goto:eof
```

Once in a while the script neglects to process files in the starting directory. We discover that it fails whenever the user appends a trailing backslash to the specified directory. The backslash is interpreted by CreateDocs as an escape for the closing double quote. So instead of receiving [SomeDirectory\] in argv[1] when processing [”SomeDirectory\” VERBOSE], it receives [SomeDirectory” VERBOSE]. If you don’t already understand, you will later see why this is happening. It may seem a little mysterious that the FOR loop works correctly, but that’s only because cmd.exe parses things differently than the executable does.

Although we now know what the problem is, we can’t just tell our users not to append a trailing backslash. *Someone* will get it wrong! To fix this in CreateDocs would require some relatively complex logic. We could, for example, detect that argv[0] contains a [”] at the end of a valid path (and the directory exists), possibly followed by [VERBOSE].

We opt instead to add logic to our script to detect and remove the trailing backslash:

```
1  @echo off
2  REM %1 is the root of the directory tree to process.
3  REM %2 is the optional string VERBOSE
4  cls
5  setlocal
6  echo.
7  IF '%1'=='' echo No Path Specified& goto:eof
8  set ROOT_PATH=%~1
9
10 REM Strip off trailing backslash:
11 IF [%ROOT_PATH:~-1,1%]==[\] set ROOT_PATH=%ROOT_PATH:~0,-1%
12
13 REM Process root directory:
14 CreateDocs "%ROOT_PATH%" %2
15
16 REM Process subdirectories:
17 FOR /F %%S IN ('dir /ad /b "%ROOT_PATH%"') DO (
18    CreateDocs "%%S" %2
19    )
20 echo.
21 goto:eof
```

We congratulate ourselves on our cleverness and we use the script successfully for months.

Then one day we change our build process. We want to be able to debug using binaries built on different development machines. Because the source code is installed in different places on different machines (something we *don’t* want to change), the debugger can’t always find the .pdb file because it’s specified in the executable image with an absolute path. We could fix this various ways, but we decide to use a fixed drive letter in the build scripts and use *subst* to map the root of the source code tree, wherever it is on a particular machine, to the root of this drive.

Once again we start to notice that our script occasionally (but not always) fails.

What’s happening *now*?

The answer is that our script was smart, but not smart enough. The problem this time turns out to be caused by the fact that we are now

specifying a root directory (ex. [x:\]). When the batch file removes the trailing backslash it generates [x:] which (for a root directory only) is not the same as *with* the backslash. The former specifies the default working

Most of the time our users open a dedicated console for running this tool and never do any other work there. In that case c:\ is the same as c:. But occasionally someone switches to the *subst*ed drive and goes to a subdirectory to do some other work. This makes c: different from c:\ and on those infrequent occasions the script fails.

We fix it by adding a special case to *not* remove the trailing backslash for the special case of a root directory explicitly specified.

This example was not intended to teach you what to do, but to give a reasonable example of how problems can creep in.

The above batch file still has some issues that I will leave unresolved for now (the "IF" statements may fail with certain strings containing double quote characters).

The Problem With the Existing Way of Looking at Things

Consider the following set of partial command lines:

```
["Argument With Spaces"]

[Argument" "With" "Spaces]

["Argument "With" Spaces"]

[Argument" With Sp"aces]

["Argument With Spaces]

["Ar"g"um"e"n"t" W"it"h Sp"aces""]
```

Before continuing, take a few moments and try to pick out the outer and the embedded quoted strings.

You may be surprised to learn that *all* of these are interpreted *exactly the same way* by the command line parser— as a single argument: [Argument With Spaces].

But how can all of these possibly generate the same argument? One of the quotes is not even closed!

The command line parser rules must be either inconsistent, incomprehensible, or just plain stupid.

But the problem isn't with the rules, but simply that the conventional way of looking at things is wrong. We need a different way of analyzing command lines that's easy to apply and always works.

(for an even more extreme and humorous example take a look at 50 Ways to Say Hello)

A Better Way To Look at Things

We come now to a very important concept.

> Double quote characters in a command line string have no relation to the boundaries between arguments and do not necessarily enclose arguments. Each individual double quote character *by itself* acts simply as a switch to enable or disable the recognition of space as a divider between arguments.

Attempting to find pairs of double quote characters that enclose meaningful

chunks of text is possibly *the biggest conceptual mistake* that people make when looking at a complicated command line.

This point is extremely important (possibly the most important concept in how a command line is received and processed by an executable program.

Each of the command line parsers we will consider (parse_cmdline, CommandLineToArgvW, and cmd.exe) has a set of characters that in some contexts it considers *special* (examples are whitespace, double quote, caret, and backslash) and which cause some action to be taken when one of them is encountered— such as beginning a new argument (and removing the special character).

In other contexts the parser treats these same characters like regular text. So at any given time the parser is in one of two states— recognizing (i.e., *interpreting*) special characters, or *ignoring* them.

The Parser States Named

In order to be explicit when referring to these two states in the text, I invented names for them. I call the first state *InterpretSpecialChars* and the second state *IgnoreSpecialChars*. They correspond to what some writers refer to as being *outside* or *inside* a double quoted string. I purposely avoided giving them names containing *quote* or *quoting* because doing so reinforces the misconception that double quote characters somehow delineate arguments. The colors shown are used later to show the parser state in images I created to demonstrate how example command lines are parsed.

I originally considered calling these *InterpretWhitespace* and *IgnoreWhitespace*, but I wanted to use the same state names when discussing cmd.exe where the state governs the interpretation of a different set of special characters, not whitespace.

> The key to understanding any command line, no matter how complex, is to pay attention to which of these two states the parser is in at any given time and understanding what causes the state to change.

The Last Example Explained

Let's examine the last example in the above list to see how it is parsed, character by character, from left to right. The *special* character that we will see either *interpreted* or *ignored*, depending on the parser state, is the space character.

You'll see that the parser state is IgnoreSpecialChars when both the spaces are read, so they do not cause a new argument to be started.

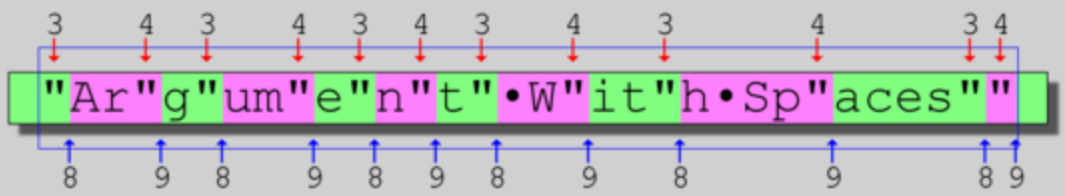
Each line below represents a single character read from the command line. The first column is the actual character read, followed by the parser state before processing the character, the action that was taken, and the value of the partial argument after processing the character.

Here's the command line again:

```
["Ar"g"um"e"n"t" With Sp"aces""]

Char read: State when char read:    Action:                       Argument (after processing char):
           InterpretSpecialChars    Start                         []
    "      InterpretSpecialChars    Go to IgnoreSpecialChars      []
    A      IgnoreSpecialChars       Add char [A]                  [A]
    r      IgnoreSpecialChars       Add char [r]                  [Ar]
    "      IgnoreSpecialChars       Go to InterpretSpecialChars   [Ar]
    g      InterpretSpecialChars    Add char [g]                  [Arg]
    "      InterpretSpecialChars    Go to IgnoreSpecialChars      [Arg]
    u      IgnoreSpecialChars       Add char [u]                  [Argu]
```

```
m      IgnoreSpecialChars      Add char [m]                [Argum]
"      IgnoreSpecialChars      Go to InterpretSpecialChars [Argum]
e      InterpretSpecialChars   Add char [e]                [Argume]
"      IgnoreSpecialChars      Go to InterpretSpecialChars [Argumen]
t      InterpretSpecialChars   Add char [t]                [Argument]
"      InterpretSpecialChars   Go to IgnoreSpecialChars    [Argument]
       IgnoreSpecialChars      Add char [ ]                [Argument ]
W      IgnoreSpecialChars      Add char [W]                [Argument W]
i      IgnoreSpecialChars      Add char [i]                [Argument Wi]
t      IgnoreSpecialChars      Add char [t]                [Argument Wit]
h      IgnoreSpecialChars      Add char [h]                [Argument With]
       IgnoreSpecialChars      Add char [ ]                [Argument With ]
S      IgnoreSpecialChars      Add char [S]                [Argument With S]
p      IgnoreSpecialChars      Add char [p]                [Argument With Sp]
"      IgnoreSpecialChars      Go to InterpretSpecialChars [Argument With Sp]
a      InterpretSpecialChars   Add char [a]                [Argument With Spa]
c      InterpretSpecialChars   Add char                    [Argument With Spac]
e      InterpretSpecialChars   Add char [e]                [Argument With Space]
s      InterpretSpecialChars   Add char [s]                [Argument With Spaces]
"      InterpretSpecialChars   Go to IgnoreSpecialChars    [Argument With Spaces]
"      IgnoreSpecialChars      Go to InterpretSpecialChars [Argument With Spaces]
```



3) Switch to IgnoreSpecialChars
4) Switch to InterpretSpecialChars
8) Single double quote character- Enter IgnoreSpecialChars
9) Single double quote character- Return to InterpretSpecialChars

Quoting Defined

Enabling or disabling the recognition of special characters (switching between InterpretSpecialChars and IgnoreSpecialChars) is done by strategically placing double quote characters at specific locations on the command line— wherever you want the state to change. This is under the control of the person who writes the command line. The act of placing double quote characters for this purpose is how I define the term *quoting*. It is *not* the delineation of a piece of text by enclosing it in a pair of double quote characters.

Quoting is the placement of individual double quote characters on the command line to control the switching between InterpretSpecialChars and IgnoreSpecialChars.

Since double quote characters work individually and not in pairs, there are no such concept as a *dangling* (unclosed) or *mismatched* quote as discussed by other writers. The command line parser simply ends in one or the other of the two states. If there is an even number of un-escaped double quote characters on the command line (we will define *escape* later) the parser ends in InterpretSpecialChars. If there is an odd number of un-escaped double quote characters it ends in IgnoreSpecialChars.

What would traditionally be called a quoted string (text between two double quote characters) can span multiple arguments or be contained completely within an argument— more evidence that it's hopeless to use pairs of double quote characters to find the arguments!

The seemingly arcane parse rules in the Microsoft documentation actually make sense once you understand that they don't directly tell you how the command line is split into arguments, but merely describe what causes the parser to switch state.

By training your mind to scan a command line from left to right like the parser does, instead of trying to pick out the quoted *chunks*, you will have little problem understanding or correctly generating even the most

We'll talk more about this when we go over the specifics of the parsers later.

How a Program Receives it's Command Line

Everyone knows the purpose of command line arguments— to customize a particular run of a program or a script. This section covers how an executable program interprets the command line string received from CreateProcess. Elsewhere I will cover the behavior of cmd.exe. For now, keep the following in mind:

The way cmd.exe interprets the command line is different and completely independent of how an executable program interprets the command line. You must separate your thinking about what cmd.exe does from your thinking about what an executable program does.

From a programmer's perspective, a Windows application written in C or C++ begins execution in main or WinMain and makes available (as function parameters) either an array of individual arguments (main's argv[]) or a single command line string (WinMain's lpCmdLine). Both types of application also have available an alternate source for the same set of arguments contained in argv[]— the global variables __argc and __argv. These are filled in before main or WinMain is called so many programmers believe that their programs receive the command line already split up into individual arguments, perhaps by Windows itself. But that is not the case.

A program receives a single command line string which the program itself, not the operating system, splits into individual arguments.

This string normally consists of the executable name followed by the actual arguments. I will have a lot more to say about this, but for now the important thing to remember is there is just one command line string.

All programs executed under Windows are ultimately started by CreateProcess (or variants such as CreateProcessAsUser). CreateProcess is a complex subject and I'm only going to look at two of the parameters it accepts: the executable name of the program to start (lpApplicationName) and the command line string (lpCommandLine), both of which are optional. Obviously, you have to tell Windows what program to run, so if the first parameter is not supplied (is NULL) then the program name must be given at the start of the command line string. By convention, even if you *do* specify the program name in the first parameter, you're supposed to repeat it at the beginning of the command line (if you supply one), but this is not enforced and leaving it out can cause problems. If you don't supply a command line Windows creates one containing just the program name.

A program does not know how its name was specified and most programs blindly assume the first argument is the program name.

The first command line argument is usually interpreted differently than the rest of the command line or ignored altogether. A Windows GUI app will not even see the first argument (at least not in lpCmdLine). If it's something important it will get lost:

> [ImportantArgument-0 Argument-1] is received in lpCmdLine as just [Argument-1]

To avoid problems caused by special handling of the first argument, or the assumption that it is the executable name, always include the program name at the beginning of any command line string you supply to CreateProcess.

The Microsoft documentation for CreateProcess implies that including the program name at the start of lpCommandLine is optional, something only "C programmers generally" do. After we show how the parser gives special

you *always* specify the executable name as the first thing on the command line. An interesting thing to note (which we won't pursue) is that if you specify the executable name in the first argument, the PATH is not searched, but if you specify NULL for the first argument, the PATH *is* used. For security reasons, Microsoft recommends that you always supply the first argument to CreateProcess and always enclose the executable name at the beginning of the command line string in quotes, though this is strictly only necessary if the path contains spaces.

How the Command Line is Split

Programs generally split their command line by treating a sequence of one or more whitespace characters (space, tab) as the separator between arguments. I say *generally* because a program is free to do whatever it wants and there are no standards to guide us. There is not even universal agreement on the set of characters that constitute whitespace.

A program can interpret its command line any way it wants. There is no method to ensure with 100 percent certainty that a given command line is correct or guarantee how it will be broken up into individual components.

This may sound hopeless, but fortunately there's only a small set of facilities for parsing the command line that most programs use. Once you understand these, you will understand how the vast majority of programs behave.

A program that links with the Microsoft C/C++ runtime library automatically splits the command line by calling a function named parse_cmdline and passes the result to main in argc and argv, or to WinMain through the global variables __argc and __argv.

A Windows GUI program can access the command line string (minus the program name) through the lpCmdLine argument to WinMain, and any program can access the full command line, including the program name (or whatever else was specified at the start of the command line when CreateProcess was called) by calling the aptly named *GetCommandLine* function. The program can then split the resulting string explicitly either by calling CommandLineToArgvW or a custom command line parser.

The splitting rule used by both these parsers is simple: splitting always occurs on a whitespace boundary, and *only* when the parser state is InterpretSpecialChars.

Almost all programs use the arguments generated by either parse_cmdline or CommandLineToArgvW, so we will focus on these.

Getting a View on Things

In the next post I'll go into great detail about the algorithms used by parse_cmdline and CommandLineToArgvW to split a command line. We'll see some ugliness caused by a special Microsoft rule governing backslashes and we'll go over the special handling of the first argument. But first I'm going to give you some tools for looking at things (later I'll give you additional tools that may actually make your job easier).

Download DumpArgs Project
Download RunTest Project

Both of these are console programs.

The first utility is called DumpArgs and is used to see exactly how a given command line is split into arguments.

DumpArgs first displays the command that it received, both ANSI and Unicode in case there's a difference. Since it is a console app, it already has the command line split into arguments by parse_cmdline (in argv[]). The program retrieves the raw command line using GetCommandLine and splits for comparison.

Simply run it with any command line you want.

Example Run of DumpArgs

Command line:

DumpArgs "First NotSecond" Second!

Output:

```
Unicode Command Line (from GetCommandLineW): [dumpargs  "First NotSecond" Second!]


              00 01 02 03 04 05 06 07 | 08 09 0a 0b 0c 0d 0e 0f
              -------------------------------------------------
   00000000:  64 00 75 00 6d 00 70 00 | 61 00 72 00 67 00 73 00   d.u.m.p. | a.r.g.s.
   00000010:  20 00 20 00 22 00 46 00 | 69 00 72 00 73 00 74 00   _._.".F. | i.r.s.t.
   00000020:  20 00 4e 00 6f 00 74 00 | 53 00 65 00 63 00 6f 00   _.N.o.t. | S.e.c.o.
   00000030:  6e 00 64 00 22 00 20 00 | 53 00 65 00 63 00 6f 00   n.d."._. | S.e.c.o.
   00000040:  6e 00 64 00 21 00 00 00                             n.d.!...


ANSI Command Line (from GetCommandLineA): [dumpargs  "First NotSecond" Second!]


              00 01 02 03 04 05 06 07 | 08 09 0a 0b 0c 0d 0e 0f
              -------------------------------------------------
   00000000:  64 75 6d 70 61 72 67 73 | 20 20 22 46 69 72 73 74   dumpargs | __"First
   00000010:  20 4e 6f 74 53 65 63 6f | 6e 64 22 20 53 65 63 6f   _NotSeco | nd"_Seco
   00000020:  6e 64 21 00                                         nd!.


CommandLineToArgvW Found 3 Argument(s)
   arg 0   = [dumpargs]
   arg 1   = [First NotSecond]
   arg 2   = [Second!]


Command Line Arguments From argv Array (argc = 3):
   argv[0] = [dumpargs]
   argv[1] = [First NotSecond]
   argv[2] = [Second!]
```

I show the source code below, but you can download a zip archive containing a pre-built executable, source code, and a Visual Studio project.

Notes on the VS solution (these notes apply to both the DumpArgs project and the StartTest project, below):

- Just unzip the archive to an empty directory and open the .sln file.
- You can switch between ANSI and Unicode build by first clicking on the project in the Solution Explorer in Visual Studio and selecting *Project->DumpArgs Properties*. Under *Configuration Properties->General* you will see *Character Set* in the right pane. Select the one you want (counter-intuitively, you select *Multi-Byte Character Set* for an ANSI build).

```
1  // DumpArgs.cpp : Defines the entry point for the console app
2  //
3
4  #include "stdafx.h"
```

```
void APIENTRY DumpHex (unsigned char* Buffer, int Length) //
{
for (int BufIdx = 0; BufIdx < Length; ++BufIdx) {
   if ((BufIdx % 256) == 0) {
      _tprintf (_T("               ---------------------------
      }
   if ((BufIdx % 16) == 0) {
      // Print buffer offset at start of lines
      _tprintf (_T("   %08lx:  "), BufIdx);
      }
   _tprintf (_T("%02x "), Buffer[BufIdx]);
   // Output separator in middle (but not after last byte):
   if (((BufIdx % 16) == 7) && (BufIdx < (Length - 1))) {
        _tprintf (_T("| "));
        }
     if (((BufIdx % 16) == 15) || (BufIdx == (Length - 1))) {
        // Pad last line with spaces if necessary
        int Padding = 3 * (16 - ((BufIdx % 16) + 1));
        if (Padding > 21) {
         Padding += 2; // For where separator would have beer
         }
       while (Padding--) {
          _tprintf (_T(" "));
          }
       // Output printable characters
       _tprintf (_T("  "));
       for (int CharIdx = 16 * (BufIdx / 16); CharIdx <= BufIc
          _tprintf (_T("%c"), isprint(Buffer[CharIdx])?Buffer[
          // Output separator in middle (but not after last by
          if (((CharIdx % 16) == 7) && (CharIdx != BufIdx)) {
             _tprintf (_T(" | "));
             }
          }
       _tprintf (_T("\n"));
       }
   }
}

int _tmain(int argc, _TCHAR* argv[])
{
wchar_t* CommandLineUnicode = GetCommandLineW();
wprintf(L"Unicode Command Line (from GetCommandLineW): [%s]\n
DumpHex((unsigned char*)CommandLineUnicode, (2 * wcslen(Comma
_tprintf (_T("\n\n"));

char* CommandLineAnsi = GetCommandLineA();
printf("ANSI Command Line (from GetCommandLineA): [%s]\n\n",
DumpHex((unsigned char*)CommandLineAnsi, 1 + strlen(CommandLi
_tprintf (_T("\n\n"));

int NumArgs = 0;
wchar_t** Args = CommandLineToArgvW(CommandLineUnicode, &NumA
_tprintf (_T("CommandLineToArgvW Found %d Argument(s)\n"), Nu
for (int arg = 0; arg < NumArgs; ++arg) {
    wprintf (L"   arg %s%d   = [%s]\n", ((NumArgs >= 10) && (
   }

_tprintf (_T("\nCommand Line Arguments From argv Array (argc
for (int arg = 0; arg < argc; ++arg) {
    _tprintf (_T("   argv[%s%d] = [%s]\n"), ((argc >= 10) &&
   }
_tprintf (_T("\n"));
LocalFree(Args);

return 0;
}
```

The next utility, RunTest, lets you drive DumpArgs with specific command lines without worrying about how cmd.exe first mangles the command line.

You specify each command line in a file named CommandLines.txt which must be a plain ANSI text file (not Unicode) in the current directory. DumpArgs must also be in the current directory.

Put each command line on a separate line in CommandLines.txt *exactly as you want DumpArgs to receive it*. Empty lines are ignored and so are lines starting with a semicolon (in the first column), so you can include comments.

RunTest reads the each command line from CommandLines.txt and starts DumpArgs using CreateProcess, passing it the specified command line in the lpCommandLine parameter. It waits up to 5 seconds for DumpArgs to Finish. RunTest explicitly specifies DumpArgs.exe in lpApplicationName and does not insert *DumpArgs.exe* at the beginning of lpCommandLine so that

you can test what happens if you pass something other than the executable name as the first command line argument (for example, to see how the first argument is parsed differently).

download a zip archive containing pre-built executables (for both DumpArgs and RunTest), source code and a Visual Studio project for RunTest and a sample CommandLines.txt file.

```
// RunTest.cpp : Defines the entry point for the console appl
//
#include "stdafx.h"
/*
 * Very simple-minded read line function:
 *
 *    - Writes the next line into Buffer and returns true if
 *    - Trusts that Buffer and BufSize are both valid.
 *    - Always NULL terminates Buffer, unless Buffer is 0 byt
 *    - Works only with ANSI file (not Unicode).
 *    - Skips empty lines and lines that start with ";" (for
 *    - Silently truncates line if Buffer too small (but cons
 *
 */
bool ReadLine (HANDLE hFile, char* Buffer, int BufSize)
{
bool Result = false;
try {
   if (BufSize >= 1) {
      int BytesWritten = 0;
      char* BufPtr = Buffer;
      while (1) {
         char Char;
         DWORD BytesRead;
         if ((ReadFile(hFile, &Char, sizeof(char), &BytesRead
            if (Char == 0x0a) {
               if (BytesWritten > 0) {
                  if (Buffer[0] == ';') { // Ignore comments
                     Result = false;
                     BytesWritten = 0;
                     BufPtr = Buffer;
                     }
                  else {
                     break;
                     }
                  }
               }
            else if ((Char != 0x0d) && (BytesWritten++ < (Bu
               *BufPtr++ = Char;
               Result = true;
               }
            }
         else {
            break;
            }
         }
      *BufPtr = '\0';
      }
   }
catch (...)
   {
   Result = false;
   }
return Result;
}

int _tmain(int argc, _TCHAR* argv[])
{
HANDLE hFile = CreateFileA (".\\CommandLines.txt", GENERIC_RE
if (hFile == INVALID_HANDLE_VALUE) {
   _tprintf(_T("\n\nERROR: Could not find CommandLines.txt in
   return 1;
   }
while (1) {
   char CommandLine[1024];
   if (ReadLine (hFile, CommandLine, 1024) == false) {
      break;
      }
   // Create the process:
   STARTUPINFOA si;
   PROCESS_INFORMATION pi;
   memset (&si, 0, sizeof(si));
   memset (&pi, 0, sizeof(pi));
   GetStartupInfoA(&si);
   si.cb = sizeof(si);
   si.dwFlags = STARTF_USESHOWWINDOW;

   if (CreateProcessA (".\\DumpArgs.exe", CommandLine, NULL,
      // Successfully created the process. Wait for it to fin
      if (WaitForSingleObject(pi.hProcess, 5000) == WAIT_TIME
```

```
81              _tprintf (_T("Timed out waiting for process to exit\
82              }
83          }
84      else {
85          tprintf ( T("CreateProcess failed\n"));
88  CloseHandle (hFile);
89  return 0;
90  }
```

Now that you understand how a command line is given to a program, and have some tools for looking at things, we can begin next time to look closely at exactly how a command is parsed.

August 7, 2013, 8:43 PM
Trackback

Post a Comment

Name (optional)

E-Mail (will not be published) (optional)

Website

Submit

Entries (RSS) and Comments (RSS)