A brief tour of the console alias functions

devblogs.microsoft.com/oldnewthing/20160201-00

February 1, 2016

Raymond Chen

Today's Little Program exercises the console alias functions. These functions let you define console aliases which are active when a target program reads a line of text from the console. The alias is recognized when it is entered at the start of a line. (Therefore, a way to defeat an alias is to put a space in front of it.) More details about console aliases can be found in the documentation for `DOSKEY`.

The program we'll write has five commands:

`add program.exe alias "value"`
This defines a console alias for the specified program.

`delete program.exe alias`
This deletes a console alias definition.

`show program.exe alias`
This shows the current definition of an alias.

`showall program.exe`
This shows all aliases defined for the specified program.

`showexes`
This shows all programs that have aliases defined.

Let's dive in.

```
#define UNICODE
#define _UNICODE
#include <windows.h>
#include <iostream>
#include <vector>

void do_add(int argc, wchar_t **argv);
void do_delete(int argc, wchar_t **argv);
void do_show(int argc, wchar_t **argv);
void do_showall(int argc, wchar_t **argv);
void do_showexes(int argc, wchar_t **argv);

int __cdecl wmain(int argc, wchar_t **argv)
{
  auto command = argv[1];

  if (wcscmp(command, L"add") == 0) {
    do_add(argc, argv);
  } else if (wcscmp(command, L"delete") == 0) {
    do_delete(argc, argv);
  } else if (wcscmp(command, L"show") == 0) {
    do_show(argc, argv);
  } else if (wcscmp(command, L"showall") == 0) {
    do_showall(argc, argv);
  } else if (wcscmp(command, L"showexes") == 0) {
    do_showexes(argc, argv);
  }
  return 0;
}
```

The main program looks at the first command line argument and dispatches the rest of the work to the appropriate handler function. Now let's look at each of the handlers. Remember, Little Programs do little to no error checking.

```
void do_add(int argc, wchar_t **argv)
{
  auto program = argv[2];
  auto alias = argv[3];
  auto value = argv[4];
  if (AddConsoleAlias(alias, value, program)) {
    std::wcout << alias << L"=" << value << std::endl;
  } else {
    std::wcout << L"Failed to add alias" << std::endl;
  }
}
```

To add an alias, we call `AddConsoleAlias` with the alias, the value, and the program it should be applied to. An example alias might be

```
scratch add cmd.exe proj
  "cd /D \"%USERPROFILE%\Documents\Visual Studio 2015\Projects\$*\""
```

(All one line; split into two for expository purposes.)

This lets you type `proj` to go to your Visual Studio projects directory, and `proj scratch` to go to the `scratch` project. Note that we had to quote the value twice, once to get it past the scratch program's command line parser, and a second time to get it past `cmd.exe`'s command line parser.

Next is deletion: To delete an alias, you set it to a null pointer.

```
void do_delete(int argc, wchar_t **argv)
{
  auto program = argv[2];
  auto alias = argv[3];
  if (AddConsoleAlias(alias, nullptr, program)) {
    std::wcout << alias << L" deleted" << std::endl;
  } else {
    std::wcout << L"Failed to delete alias" << std::endl;
  }
}
```

Continuing our example, if you get bored of the `proj` alias, you can delete it by saying `scratch delete cmd.exe proj`.

The next command is for showing the value of an alias,

```
void do_show(int argc, wchar_t **argv)
{
  auto program = argv[2];
  auto alias = argv[3];
  wchar_t value[8192];
  if (GetConsoleAlias(alias, value, sizeof(value), program)) {
    std::wcout << alias << L"=" << value << std::endl;
  } else {
    std::wcout << L"Cannot show (maybe it isn't defined)" << std::endl;
  }
}
```

There is no way to query the length of an alias's value, but since the maximum command line length supported by `cmd.exe` is 8192, a buffer size of 8192 is a safe bet for now. (This is a Little Program and doesn't need to worry itself with pesky things like forward compatibility.)

The last two commands are for showing all the aliases defined for a specific program, and for showing the programs that have aliases defined. The two functions are very similar, so we present them together. First, a simple version that is subtly defective:

```
// code in italics is wrong
void do_showall(int argc, wchar_t **argv)
{
  auto program = argv[2];
  auto bytes = GetConsoleAliasesLength(program);
  std::vector<wchar_t> buffer(
    (bytes + sizeof(wchar_t) + 1) / sizeof(wchar_t));
  if (GetConsoleAliases(buffer.data(), bytes, program)) {
    for (auto current = buffer.data();
         current < buffer.data() + buffer.size();
         current += wcslen(current) + 1) {
      std::wcout << current << std::endl;
    }
  }
}

void do_showexes(int argc, wchar_t **argv)
{
  auto bytes = GetConsoleAliasExesLength();
  std::vector<wchar_t> buffer(
    (bytes + sizeof(wchar_t) + 1) / sizeof(wchar_t));
  if (GetConsoleAliasExes(buffer.data(), bytes)) {
    for (auto current = buffer.data();
         current < buffer.data() + buffer.size();
         current += wcslen(current) + 1) {
      std::wcout << current << std::endl;
    }
  }
}
```

One annoyance here is that the `GetConsoleAliasesLength` function returns a byte count rather than a `TCHAR` count, so we have to do conversion between bytes and `TCHAR`s. In case we get an odd number back (which shouldn't ever happen, but better safe than sorry), we round up to get the number of `wchar_t`s.

The next annoyance is that the `GetConsoleAliases` function returns a series of null-terminated strings, but the last string is not double-null-terminated (or more accurately, terminated with an empty string). This means that *you don't know when you're finished*! After you process one string, the next byte could be the start of the next string, or it could just be uninitialized garbage. If another thread deletes an alias between the calls to `GetConsoleAliasesLength` and `GetConsoleAliases`, then we pass a too-large buffer to `GetConsoleAliases`, and the unused bytes contain uninitialized garbage, and we have no way to know when the valid data ends and the uninitialized garbage begins.

Not knowing when you have reached the end of the valid data is a really bad situation for a program to be in.

We can work around this problem by zeroing out the memory before we call `GetConsoleAliases`; that way, if the buffer we pass turns out to be too large (because another thread deleted an alias in the meantime), the extra zeros we wrote created a double-null-terminated string buffer.

On the other hand, if that didn't happen, then we want to stop when we reached the end of the buffer.

The final problem is that another thread could *add* an alias in between our calls to `GetConsoleAliasesLength` and `GetConsoleAliases`, and the call to `GetConsoleAliases` will fail because the buffer is too small. In that case, we want to loop back and try again with a bigger buffer.

All of the preceding issues with `GetConsoleAliases` also apply to `GetConsoleAliasExes`.

Here's the resulting code that tries to solve all of the problems:

```
template<typename GetLengthBytes, typename GetContents>
void PrintAliasValue(
  const GetLengthBytes& getLengthBytes,
  const GetContents& getContents)
{
  std::vector<wchar_t> buffer;
  do {
    auto bytes = getLengthBytes();
    auto length = (bytes + sizeof(wchar_t) - 1) / sizeof(wchar_t);
    buffer.resize(length);
    ZeroMemory(buffer.data(), bytes);
    SetLastError(ERROR_SUCCESS);
    if (getContents(buffer.data(), bytes)) {
      for (auto current = buffer.data();
           current < buffer.data() + buffer.size() && *current;
           current += wcslen(current) + 1) {
        std::wcout << current << std::endl;
      }
    }
  } while (GetLastError() == ERROR_MORE_DATA);
}

void do_showall(int argc, wchar_t **argv)
{
  auto program = argv[2];
  PrintAliasValue(
    [program]() { return GetConsoleAliasesLength(program); },
    [program](LPTSTR buffer, DWORD length) {
        return GetConsoleAliases(buffer, length, program); });
}

void do_showexes(int argc, wchar_t **argv)
{
  PrintAliasValue(
    []() { return GetConsoleAliasExesLength(); },
    [](LPTSTR buffer, DWORD length) {
       return GetConsoleAliasExes(buffer, length); });
}
```

The underlying algorithm is the same: Get the byte length, allocate a vector of characters, zero-initialize the data in the vector so that we can detect that a short buffer was returned, then ask for the data. If it succeeds, then read out the data, but stop when we hit one of our preallocated zeroes, or when we reach the end of the buffer, whichever comes first. If it fails because the buffer is too small, then loop back and try again.

So there you have it. A quick tour of the console alias functions. Now you can write your own `DOSKEY` replacement.

Raymond Chen

Follow