# 2 Command-Line Arguments (args.hhf)

The HLA args module provides access to, and support for, Windows Command Line Interpreter or Linux/*nix shell command line parameters.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

**A Note About Thread Safety**: The args module maintains a couple of static global variables that maintain the command-line values. Currently, these values apply to all threads in a process. You should take care when changing these values in threads. The command-line is a resource that must be shared amongst all threads in an application. If you write multi-threaded applications, it is your responsibility to serialize access to the command-line functions.

## 2.1 The Args Module

To use the args functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "args.hhf" )
or
#include( "stdlib.hhf" )
```

## 2.2 Retrieving the Command Line

The arg.cmdLn and arg.a_cmdLn functions retrieve the entire command-line as a string. This string typically consists of the list of command-line parameters, each parameter separated by a single space. Note that on some operating systems, the HLA standard library command-line functions might actually synthesize this string by concatenating the command-line arguments together, so you can not expect this string to be an exact representation of the command line that the user typed (that is, the user may have typed extra spaces or other delimiter characters that the OS' shell discarded before passing the command line text to the program). Also, the command-line string will not contain I/O redirection or other process-related items (e.g., pipes) found on the command line.

**procedure arg.cmdLn();**

*arg.cmdLn* returns a pointer to a static string held inside the stdlib code. This function returns the string pointer in EAX. The caller must not modify any data in this string (use arg.a_cmdLn if you need a malleable string).

```
HLA high-level calling sequence examples:

arg.cmdLn();
stdout.put( "Command line: ", (type string eax), nl );

HLA low-level calling sequence examples:

  call arg.cmdLn;
  push( eax );
  call stdout.puts;
```

**procedure arg.a_cmdLn();**

*arg.a_cmdLn* returns, in EAX, a string containing a copy of the command-line parameter text. This is a pointer to a string allocated on the heap. It is the caller's responsibility to free this storage (via a call to str.free) when it is done using the string.

```
HLA high-level calling sequence examples:

arg.a_cmdLn();
stdout.put( "Command line: ", (type string eax), nl );
str.free( eax );

HLA low-level calling sequence examples:

  call arg.cmdLn;
  push( eax );
  call stdout.puts;
  push( eax );
  call str.free;
```

## 2.3 Argument Count and Item

The functions in this category are the standard argument functions that most programs use: arg.c, arg.v, and arg.a_v. The arg.c function returns the number of command-line parameters and arg.v/arg.a_v return a pointer to a string that corresponds to an individual parameter.

**`procedure arg.c();`**

*arg.c* returns the number of command-line parameters in EAX. This count includes the program name on the command line.

```
HLA high-level calling sequence examples:

arg.c();
stdout.put( "Number of arguments: ", (type uns32 eax), nl );

HLA low-level calling sequence examples:

  call arg.c;
  mov( eax, argCount );
```

**`procedure arg.v( whichArg:dword );`**

*arg.v* returns a pointer to the string that corresponds to the specified command-line argument. Argument indexes are in the range 0..arg.c()-1. This function will raise an exception if you pass it an argument value outside this range. This function returns a pointer to a string whose storage is internal to the standard library. You must treat this data as read-only data and not modify this data.

```
HLA high-level calling sequence examples:

arg.c();
mov( eax, edx );
for( mov( 0, ecx ); ecx < edx; inc( ecx )) do

  arg.v(ecx);
  stdout.put( "arg[", (type uns32 ecx), "]=", (type string eax), nl );

endfor;

HLA low-level calling sequence examples:

  pushd( 0 );
  call arg.v;
  mov( eax, firstArg );
```

**procedure arg.a_v( whichArg:dword );**

*arg.a_v* returns a pointer to the string that corresponds to the specified command-line argument. Argument indexes are in the range 0..arg.c()-1. This function will raise an exception if you pass it an argument value outside this range. This function returns a pointer to a string it allocates on the heap. The caller must free this storage (via a call to str.free) when it is done using the string.

```
HLA high-level calling sequence examples:

arg.c();
mov( eax, edx );
for( mov( 0, ecx ); ecx < edx; inc( ecx )) do

  arg.a_v(ecx);
  stdout.put( "arg[", (type uns32 ecx), "]=", (type string eax), nl );
  str.free( eax );

endfor;

HLA low-level calling sequence examples:

  pushd( 0 );
  call arg.a_v;
  mov( eax, firstArg );
      .
      .
      .
  str.free( firstArg );
```

## 2.4 Deleting Command Line Arguments

The functions in this category delete command-line arguments from the internal array or deallocate all the command-line arguments from the internal array. When an individual command-line parameter is deleted, the indexes of the following command-line arguments are reduced by one (that is, argument n+1 becomes argument n, argument n+2 becomes argument n+1, etc., up to the number of command-line arguments). Also note that deleting a command-line argument reduces the value that arg.c returns by one.

**procedure arg.delete( index:uns32 );**

*arg.delete* removes the specified command-line parameter from the internal argv array. The index parameter must be in the range 0..*argc*, where *argc* is the current value that arg.c() returns. If the index parameter is outside this range, this function will raise an ex.BoundsError exception. This function decrements the value that arg.c returns by one. Note that this function does not affect the value that arg.cmdLn returns.

```
HLA high-level calling sequence examples:

arg.c();
stdout.put( "Number of arguments: ", (type uns32 eax), nl );
arg.delete( 0 );
arg.c();
stdout.put
(
  "Number of arguments after delete: ",
  (type uns32 eax),
  nl
);

HLA low-level calling sequence examples:
```

```
    pushd( 2 );
    call arg.delete;
```

**procedure arg.destroy();**

*arg.destroy* deletes all of the command line parameters from internal storage and resets the command-line processor. The next function that requests a command line parameter value will force the run-time system to regenerate the command-line argument list.

```
HLA high-level calling sequence examples:

arg.destroy();
arg.c(); // Regenerates original command-line
mov( eax, originalArgc );

HLA low-level calling sequence examples:

  call arg.destroy;
  call arg.c;
  mov( eax, originalArgc );
```

## 2.5 Argument Iterators

The iterators in this category process command-line arguments within a foreach loop.

**iterator arg.args();**

*arg.args* is an iterator (used in an HLA foreach loop) that returns successive command line parameters on each iteration of the foreach loop. This iterator returns a pointer to a newly allocated string in the EAX register. It is the caller's responsibility (usually in the body of the foreach loop) to free the allocated storage with a call to str.free.

```
HLA high-level calling sequence examples:

foreach arg.args() do

  stdout.put( "current Arg: ", (type string eax), nl );
  str.free( eax );

endfor;

HLA low-level calling sequence examples:

  (You really should only use the high-level calling sequence
  for a foreach loop that calls an iterator.)
```

**iterator arg.globalOptions( options:cset );**

*arg.globalOptions* is an iterator (i.e., you use it in a FOREACH loop) that yields a sequence of command line parameter options. A command line parameter option is a command line parameter that begins with a '-' or '/' character. arg.globalOptions only returns those command line parameters whose first character is a member of the "options" character set.

A typical command line might be something like the following:

```
c:> pgmName  -o2 -warn filename1 -c filename2 -d filename3 -x
```

The command line options in this example are "-o2", "-warn", "-c", "-d", and "-x". The arg.globalOptions iterator only considers those command line parameters that begin with "-" and whose first character is a member of the options parameter. Assuming options contains at least {'c', 'd', 'o', 'w', 'x'} then the arg.globalOptions iterator will return the strings "o2", "warn", "c", "d", and "x", in that order (that is, in the order they appear on the command line). If the first character of a command line option is not in the options character set, then arg.globalOptions does not return that particular command line parameter.

Note that *arg.globalOptions* does not remove the command line parameters from the command line string or from the arg.v array. If you want to remove them, you must explicitly do so using the *arg.delete* function. Note, however, that you must not delete command line arguments while scanning through the arguments in a FOREACH loop using the *arg.globalOptions* iterator.

Note that this iterator allocates storage for each string it returns on the heap. It is the caller's responsibility to free the storage when the caller is done using the string.

```
HLA high-level calling sequence examples:

foreach arg.globalOptions( {'a', 'b', 'c' } ) do

  stdout.put( "current option: ", (type string eax), nl );
  str.free( eax );

endfor;

HLA low-level calling sequence examples:

  (You really should only use the high-level calling
  sequence for a foreach loop that calls an iterator.)
```

**iterator arg.localOptions( options:cset );**

*arg.localOption* is an iterator that yields the sequence of command line options begining at parameter "index". This iterator only yields strings as long as successive parameters begin with a "-". It fails upon encountering a command line parameter that is not an option (that is, begins with "-"). Note that this iterator only yields those command line options whose first character beyond the "-" character is a member of the options character set.

Typically, you would use the *arg.localOption* iterator inside a FOREACH loop to obtain the command line parameters for a specific filename on the command line. That is, some programs process multiple files and let you associate command line parameters with a single filename. Consider the following simple example:

```
c:> pgm  -o2 -c file1  -o5 file2  -c file3
```

In this example, the "pgm" program (presumably) associates "-o2" and "-c" with file1, "-o5" with file2, and "-c" with file3.

Were you to call *arg.localOption* as follows:

```
foreach arg.localOption( 1, {'o', 'c'} ) do ... endfor;
```

then the *arg.localOption* iterator would return two strings: the first would be "o2" and the second would be "c". Within that iterator your code should save these options and count the number of command line parameters processed so it will know the index of the associated filename command line parameter once it is done processing the options. Typically, you would bury this FOREACH loop (with some minor modifications) inside some other loop that processes each filename (or other command line parameter preceded by command line options).

Note that this iterator allocates storage for each string it returns on the heap. It is the caller's responsibility to free the storage when the caller is done using the string.

See the CmdLnDemo.hla file in the Examples directory of the HLA distribution for an example of each of these routines.

```
HLA high-level calling sequence examples:

foreach arg.localOptions( 4, {'a', 'b', 'c' } ) do

  stdout.put( "current option: ", (type string eax), nl );
  str.free( eax );

endfor;

HLA low-level calling sequence examples:

  (You really should only use the high-level calling
  sequence for a foreach loop that calls an iterator.)
```