NAME

argtable2 − an ANSI C library for parsing GNU style command line options

SYNOPSIS

```
#include <argtable2.h>
```

struct **arg_lit**
struct **arg_int**
struct **arg_dbl**
struct **arg_str**
struct **arg_file**
struct **arg_rem**
struct **arg_end**

struct **arg_xxx*** **arg_xxx0**(...)
struct **arg_xxx*** **arg_xxx1**(...)
struct **arg_xxx*** **arg_xxxn**(...)

int **arg_nullcheck**(void **argtable)
int **arg_parse**(int argc, char **argv, void **argtable)
void **arg_print_option**(FILE *fp, const char *shortopts, const char *longopts,
const char *datatype, const char *suffix)
void **arg_print_syntax**(FILE *fp, void **argtable, const char *suffix)
void **arg_print_syntaxv**(FILE *fp, void **argtable, const char *suffix)
void **arg_print_glossary**(FILE *fp, void **argtable, const char *format)
void **arg_print_errors**(FILE *fp, struct arg_end *end, const char *progname)
void **arg_free**(void **argtable)

DESCRIPTION

To parse the command line options, first construct an **arg_xxx** struct for each option expected. These structs will hold the values that are extracted from the command line. There are various forms of the **arg_xxx** structs, each one specialized for a particular type of option (ie: literal, integer, double, string, filename). Each option may be given a short option tag (eg: -x) and/or a long option tag (eg: --verbose), or no tags at all in which case the argument is identified by its position on the command line. The latter are called untagged arguments.

Each **arg_xxx** struct can store multiple occurrences of a command line option. When an **arg_xxx** struct is constructed, we specify the minimum and maximum number of occurrences that we wish the option to handle. The **arg_xxx** constructor function will then allocate sufficient storage within the struct for the maximum number of occurrences. If we wish to handle an infinite number of occurrences of an option then we really only need specify the maximum number to be equal to argc as that is the maximum number of occurrences that is possible for any given command line invocation. Later on when the parsing is performed, the number of occurrences of the option on the command line is checked for legality. Too many or too few will be flagged accordingly.

Constructing an arg_xxx data structure

Each **arg_xxx** struct has it own unique set of constructor functions and while these may differ slightly between **arg_xxx** structs, they are generally of the form:

struct **arg_xxx* arg_xxx0** (const char *shortopts, const char *longopts, const char *datatype, const char *glossary)

struct **arg_xxx* arg_xxx1** (const char *shortopts, const char *longopts, const char *datatype, const char *glossary)

struct **arg_xxx* arg_xxxn** (const char *shortopts, const char *longopts, const char *datatype, int mincount, int maxcount, const char *glossary)

The **arg_xxx0()** and **arg_xxx1()** forms are merely abbreviated forms of **arg_xxxn()** and are provided as a convenience for the most common arrangements of command line options; namely those that have zero-or-

one occurrences (mincount=0,maxcount=1) and those that have one exactly one occurrence (mincount=1,maxcount=1) respectively.

The *const char* shortopts* parameter defines the option's short form tag (eg: -x, -k3, -D"macro"). It can be left as NULL if a short option is not required, otherwise use it to specify the desired short option character in the string (without the leading "-" and without any whitespace). For example, the short option -v is defined simply as "v". In fact, a command line option may have multiple alternate short form tags defined for it by concatenating the desired characters into the shortopts string. For instance "abc" defines an option which will accept any of the three equivalent short forms -a, -b, -c interchangeably.

The *const char* longopts* parameter is similar to *shortopts*, except it defines the option's long form tags (eg: --help, --depth=3, --name=myfile.txt). It too can be left as NULL if not required, and it too can have multiple equivalent tags defined but these must be separated by commas. For example, if we wish to define two equivalent long options --quiet and --silent then we would give longopts as "quiet,silent". Remember not to include any whitespace.

If both *shortopts* and *longopts* are given as NULL then the resulting option is an untagged argument.

The *const char* datatype* parameter is a descriptive string you can use to customize the appearance of the argument data type in error messages and so forth. It does not affect the actual data type definition as that is a fixed property of the **arg_xxx** struct. So for example, defining a *datatype* of "<bar>" will result in the option being display something like "-x <bar>" or "--foo=<bar>" depending upon your option tags. If given as NULL, the *datatype* string will revert to the default value for the particular **arg_xxx** struct. You can effectively disable the default by specifying *datatype* as an empty string.

The *int mincount* parameter specifies the minimum number of occurrences that the option must appear on the command line. If the option does not appear at least that many times then the parser reports it as a syntax error. The *mincount* defaults to 0 for the **arg_xxx0()** functions and 1 for **arg_xxx1()** functions.

The *int maxcount* parameter specifies the maximum number of occurrences that the option may appear on the command line. Any occurrences beyond the maximum are discarded by the parser reported as syntax errors. The *maxcount* defaults to 1 for both the **arg_xxx0()** and **arg_xxx1()** functions.

The *const char* glossary* parameter is another descriptive string but this one appears in the glossary table summarizing the program's command line options. The glossary table is generated automatically by the **arg_print_glossary** function (see later). For example, a glossary string of "the foobar factor" would appear in the glossary table along side the option something like:

```
--foo=<bar>    the foobar factor
```

Specifying a NULL glossary string causes that option to be omitted from the glossary table.

See below for the exact definitions of the individual **arg_xxx** structs and their constructor functions.

Defining the argument table

Once the **arg_xxx** structs are in hand they must be collated into an argument table. The argument table is merely an array of void pointers that point to the various **arg_xxx** structs constructed previously. The last entry of the argument table must always point to an **arg_end** struct. The **arg_end** struct is a special case in that it does not represent a command line option but instead serves to terminate the argument table. It also stores the errors encountered by the parser for that argument table. Thus when we construct the **arg_end** struct we specify the maximum number of errors we wish it to store, errors beyond that limit are discarded. An arbitrary number like 20 usually suffices.

The following example shows an argument table that defines the command line options: [-a] [-b] [-c] [--scalar=<n>] [-v|--verbose] [-o myfile] <file> [<file>]

```
struct arg_lit *a     = arg_lit0("a", NULL, "the -a option");
struct arg_lit *b     = arg_lit0("b", NULL, "the -b option");
struct arg_lit *c     = arg_lit0("c", NULL, "the -c option");
struct arg_int *scal  = arg_int0(NULL, "scalar","<n>", "foo value");
struct arg_lit *verb  = arg_lit0("v", "verbose, "verbose output");
struct arg_file *o    = arg_file0("o", NULL,"myfile", "output file");
```

```
struct arg_file *file = arg_filen(NULL,NULL,"<file>",1,2,"input files");
struct arg_end *end   = arg_end(20);
void *argtable[] = {a,b,c,scal,verb,o,file,end};
```

Having constructed our **arg_xxx** structs we should ensure that all of them were successfully allocated by checking that none of them returned NULL. We could do that manually, but the **arg_nullcheck** function has been provided for just that purpose. It checks the argument table for NULL entries and returns non-zero if any were encountered.

```
if (arg_nullcheck(argtable) != 0)
    printf("error: insufficient memory");
```

Parsing the command line

With the argument table defined, it is simply a matter of passing it onto the **arg_parse** function along with **argc** and **argv**. The parser will extract the command line arguments according to the specifications within the argument table and store the results back in the **arg_xxx** data structures. Any errors that are encountered are recorded in the argument table's **arg_end** struct for reporting later and a count of those errors is returned by the function.

```
int nerrors = arg_parse(argc,argv,argtable);
```

If the number of errors returned is zero then we know that the command line arguments were all correctly parsed and their results are stored in the **arg_xxx** structs ready for use. Otherwise we can choose to retry parsing the command line with an alternative argument table if we have one, or just display the errors and exit.

Error Reporting

If the **arg_parse** function reported errors then we need to display them as **arg_parse** does not do so itself. As mentioned earlier, the arg_parse function stores the errors it encounters in the argument table's **arg_end** struct. We don't need to know the internal details of the **arg_end** struct, we simply call the **arg_print_errors** function to print those errors in the order they were encountered.

```
if (nerrors>0)
    arg_print_errors(stdout,end,"myprog");
```

Notice that we also pass it the program name, in this case "myprog", as the function prepends the name to each error message it prints as in **myprog: invalid option "-x"**.

Displaying the command line syntax

The **arg_print_syntax** and **arg_print_syntaxv** functions display the command line syntax defined by an argument table. The latter is the verbose form (distinguished by the "v" suffix on the function name) and displays all alternative forms of each option verbatim, for example:

```
[-a] [-b] [-c] [--scalar=<n>] [-o myfile] [-v|--verbose] <file> [<file>]
```

Whereas the former displays the syntax in abbreviated GNU style wherein only the first form of each option is displayed. Furthermore all short options are concatenated into a single option string at the head of the display, so for example **-a -b -c -v** is displayed as **-abcv**.

```
[-abcv] [--scalar=<n>] [-o myfile] <file> [<file>]
```

Notice that optional command line arguments are automatically enclosed in square brackets whereas mandatory arguments are not.

Displaying the option glossary

The **arg_print_glossary** function displays a glossary table of all the command line options defined by an argument table in which the option's syntax is shown beside the option's glossary string.

```
arg_print_glossary(stdout, argtable, "  %-25s %s\n");
```

The format string passed to the **arg_print_glossary** function is actually a printf format string. It must contain exactly two "%s" format parameters, the first is for the option's syntax string and the second is for the argument's glossary string. Here is some example output:

```
-a                the -a option
-b                the -b option
-c                the -c option
--scalar=<n>      foo value
-v,--verbose      verbose option
-o myfile         output file
<file>            input files
```

FUNCTION REFERENCE

int arg_nullcheck (void **argtable)

Returns non-zero if the *argtable[]* array contains any NULL entries up until the terminating **arg_end*** entry. Returns zero otherwise.

int arg_parse (int argc, char **argv, void **argtable)

Parse the command line arguments in *argv[]* using the command line syntax specified in *argtable[]*, returning the number of errors encountered. Error details are recorded in the argument table's **arg_end** structure from where they can be displayed later with the **arg_print_errors** function. Upon a successful parse, the **arg_xxx** structures referenced in *argtable[]* will contain the argument values extracted from the command line.

void arg_print_option (FILE *fp, const char *shortopts, const char *longopts, const char *datatype, const char *suffix)

This function prints an option's syntax, as in **-K|--scalar=<int>**, where the short options, long options, and datatype are all given as parameters of this function. It is primarily used within the **arg_xxx** structures' *errorfn* functions as a way of displaying an option's syntax inside of error messages. However, it can also be used in user code if desired. The *suffix* string is provided as a convenience for appending newlines and so forth to the end of the display and can be given as NULL if not required.

void arg_print_syntax (FILE *fp, void **argtable, const char *suffix)

Prints the GNU style command line syntax for the given argument table, as in: [-abcv] [--scalar=<n>] [-o myfile] <file> [<file>]

The *suffix* string is provided as a convenience for appending newlines and so forth to the end of the display and can be given as NULL if not required.

void arg_print_syntaxv (FILE *fp, void **argtable, const char *suffix)

Prints the verbose form of the command line syntax for the given argument table, as in: [-a] [-b] [-c] [--scalar=<n>] [-o myfile] [-v|--verbose] <file> [<file>]

The *suffix* string is provided as a convenience for appending newlines and so forth to the end of the display and can be given as NULL if not required.

void arg_print_glossary (FILE *fp, void **argtable, const char *format)

Prints a glossary table describing each option in the given argument table. The *format* string is passed to printf to control the formatting of each entry in the the glossary. It must have exactly two "%s" format parameters as in "%-25s %s\n", the first is for the option's syntax and the second for its glossary string. If an option's glossary string is NULL then that option in omitted from the glossary display.

void arg_print_errors (FILE *fp, struct arg_end *end, const char *progname)

Prints the details of all errors stored in the *end* data structure. The *progname* string is prepended to each error message.

void arg_free (void ** argtable)

Deallocates the memory used by each **arg_xxx** struct referenced by *argtable[]*. It does this by calling **free** for each non-NULL entry up to and including the terminating **arg_end** entry.

LITERAL OPTIONS

Examples

-x, -y, -z, --help, --verbose

Data Structure

struct **arg_lit**
{
struct **arg_hdr** hdr;
int count;
};

Constructor Functions

struct **arg_lit* arg_lit0** (const char *shortopts, const char *longopts, const char *glossary)
struct **arg_lit* arg_lit1** (const char *shortopts, const char *longopts, const char *glossary)
struct **arg_lit* arg_litn** (const char *shortopts, const char *longopts, int mincount, int maxcount, const char *glossary)

Description

Literal options take no argument values so all that is to be seen in the **arg_lit** struct is the *count* of the number of times the option was present on the command line. Upon a successful parse, *count* is guaranteed to be within the *mincount* and *maxcount* limits set for the option at construction.

INTEGER OPTIONS

Examples

-x2, -y 7, -z-3, --size=734, --count 124

Data Structure

struct **arg_int**
{
struct **arg_hdr** hdr;
int count;
int *ival;
};

Constructor Functions

struct **arg_int* arg_int0** (const char *shortopts, const char *longopts, const char *datatype, const char *glossary)
struct **arg_int* arg_int1** (const char *shortopts, const char *longopts, const char *datatype, const char *glossary)
struct **arg_int* arg_intn** (const char *shortopts, const char *longopts, const char *datatype, int mincount, int maxcount, const char *glossary

Description

The **arg_int** struct contains the *count* of the number of times the option was present on the command line and a pointer (*ival*) to an array containing the integer values given with those particular options. The array is fixed at construction time to hold *maxcount* integers at most.

Upon a successful parse, *count* is guaranteed to be within the *mincount* and *maxcount* limits set for the option at construction with the appropriate values store in the *ival* array. The parser will not accept any values beyond that limit.

It is quite acceptable to set default values in the *ival* array prior to calling arg_parse if desired as the parser does alter *ival* entries for which no command line argument is received.

DOUBLE OPTIONS

Examples

-x2.234, -y 7e-03, -z-3.3E+6, --pi=3.1415, --tolerance 1.0E-6

Data Structure

struct **arg_dbl**
{
struct **arg_hdr** hdr;
int count;
double *dval;
};

Constructor Functions

struct **arg_dbl* arg_dbl0** (const char *shortopts, const char *longopts, const char *datatype, const char *glossary)

struct **arg_dbl* arg_dbl1** (const char *shortopts, const char *longopts, const char *datatype, const char *glossary)

struct **arg_dbl* arg_dbln** (const char *shortopts, const char *longopts, const char *datatype, int mincount, int maxcount, const char *glossary

Description

Like **arg_int** but the arguments values are stored as doubles in *dval*.

STRING OPTIONS

Examples

-Dmacro, -t mytitle, -m "my message string", --title="hello world"

Data Structure

struct **arg_str**
{
struct **arg_hdr** hdr;
int count;
const char **sval;
};

Constructor Functions

struct **arg_str* arg_str0** (const char *shortopts, const char *longopts, const char *datatype, const char *glossary)

struct **arg_str* arg_str1** (const char *shortopts, const char *longopts, const char *datatype, const char *glossary)

struct **arg_str* arg_strn** (const char *shortopts, const char *longopts, const char *datatype, int mincount, int maxcount, const char *glossary)

Description

Like **arg_int** but the arguments values are pointers to strings. Note that the string pointers in *sval[]* actually refer to the original *argv[]* command line string buffers so you should not attempt to alter them.

FILENAME OPTIONS

Examples

-o myfi le, -Ihome/foo/bar, --input=~/doc/letter.txt, --name a.out

Data Structure

struct **arg_fi le**
{
struct **arg_hdr** hdr;
int count;
const char **fi lename;
const char **basename;
const char **extension;
};

Constructor Functions

struct **arg_file* arg_file0** (const char *shortopts, const char *longopts, const char *datatype, const char *glossary)

struct **arg_file* arg_file1** (const char *shortopts, const char *longopts, const char *datatype, const char *glossary)

struct **arg_file* arg_filen** (const char *shortopts, const char *longopts, const char *datatype, int mincount, int maxcount, const char *glossary)

Description

Like **arg_str** but the argument strings are presumed to have filename qualities so some additional pasring is done to separate out the filename's basename and extension (if they exist). The three arrays filename[], basename[], extension[] each store up to maxcount entries, and the i'th entry of each of these arrays refer to different components of the same string buffer.

For instance, **-o /home/heitmann/mydir/foo.txt** would be parsed as:

```
filename[i]  = "/home/heitmann/mydir/foo.txt"
basename[i]  =          "foo.txt"
extension[i] =            "txt"
```

If the filename has no leading path then the basename is the same as the filename, and if no extension could be identified then it is given as NULL. Note that filename extensions are defined as all text following the last "." in the filename. Thus **-o foo** would be parsed as:

```
filename[i]  = "foo"
basename[i]  = "foo"
extension[i] = NULL
```

As with arg_str, the string pointers in *filename[]*, *basename[]*, and *extension[]* actually refer to the original *argv[]* command line string buffers so you should not attempt to alter them.

Note also that the parser only ever treats the filenames as strings and never attempts to open them as files or perform any directory lookups on them.

REMARK OPTION

Data Structure

```
struct arg_rem
  {
  struct arg_hdr hdr;
  };
```

Constructor Function

struct **arg_rem* arg_rem** (const char* datatype, const char* glossary)

Description

The **arg_rem** struct is a dummy struct in the sense it does not represent a command line option to be parsed. Instead it provides a means to include additional *datatype* and *glossary* strings in the output of the **arg_print_syntax**, **arg_print_syntaxv**, and **arg_print_glossary functions**. As such, **arg_rem** structs may be used in the argument table to insert additional lines of text into the glossary descriptions or to insert additional text fields into the syntax description. It has no data members apart from the mandatory *arg_hdr* struct.

END-OF-TABLE OPTION

Data Structure

```
struct arg_end
  {
  struct arg_hdr hdr;
  int count;
  int *error;
  void **parent;
```

```
const char **argval;
};
```

Constructor Function

struct **arg_end* arg_end** (int maxerrors)

Description

The arg_end struct is primarily used to mark the end of an argument table and doesn't represent any command line option. Every argument table must have an **arg_end** structure as its last entry.

Apart from terminating the argument table, the **arg_end** structure also stores the error codes generated by the **arg_parse** function as it attempts to parse the command line with the given argument table. The *maxerrors* parameter passed to the **arg_end** constructor specifies the maximum number of errors that the structure can store. Any further errors are discarded and replaced with the single error code ARG_ELIMIT which is later reported to the user by the message "too many errors". A *maxerrors* limit of 20 is quite reasonable.

The **arg_print_errors** function will print the errors stored in the **arg_end** struct in the same order as they occurred, so there is no need to understand the internals of the **arg_end** struct.

For those that are curious, the three arrays *error[]*, *parent[]*, and *argval[]* are each allocated *maxerrors* entries at construction. As usual, the *count* variable gives the number of entries actually stored in these arrays. The same value applies to all three arrays as the i'th entry of each all refer to different aspects of the same error condition.

The *error[i]* entry holds the error code returned by the *hdr.scanfn* function of the particular **arg_xxx** that is reporting the error. The meaning if the code is usually known only to the issuing **arg_xxx** struct. The predefined error codes that **arg_end** handles from the parser itself are the exceptions.

The *parent[i]* entry points to the parent **arg_xxx** structure that reported the error. That same **arg_xxx** structure is also responsible for displaying a pertinent error message when called on to do so by the **arg_print_errors** function. It calls the *hdr.errorfn* function of each parent **arg_xxx** struct listed in the **arg_end** structure.

Lastly, the *argval[i]* entry points to the command line argument at which the error occurred, although this may be NULL when there is no relevant command line value. For instance, if an error reports a missing option then there will be no matching command line argument value.

FILES

<installdir>/include/argtable2.h
<installdir>/lib/libargtable2.a
<installdir>/man/man3/argtable2.3
<installdir>/share/doc/argtable2-x/
<installdir>/share/doc/argtable2-x/examples/

where <installdir> is typically /usr/local.

AUTHOR

Stewart Heitmann <sheitmann@users.sourceforge.net>