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_rex**
struct **arg_file**
struct **arg_date**
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_glossary_gnu**(FILE *fp, void **argtable)
void **arg_print_errors**(FILE *fp, struct arg_end *end, const char *progname)
void **arg_freetable**(void **argtable, size_t n)

DESCRIPTION

Argtable is an ANSI C library for parsing GNU style command line arguments with a minimum of fuss. It enables the programmer to define their program's argument syntax directly in the source code as an array of structs. The command line is then parsed according to that specification and the resulting values stored directly into user-defined program variables where they are accessible to the main program.

This man page is only for reference. Introductory and advanced texts on argtable should be available on this system in pdf, html, and postscript from under **/usr/local/share/doc/argtable2/** along with example source code. Alternatively refer to the argtable homepage at http://argtable.sourceforge.net.

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.

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_glossary_gnu (FILE *fp, void **argtable)
An alternate form of **arg_print_glossary()** that prints the glossary using strict GNU formatting conventions wherein long options are vertically aligned in a second column, and lines are wrapped at 80 characters.

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_freetable (void ** argtable, size_t n)
Deallocates the memory used by each **arg_xxx** struct referenced by *argtable[]*. It does this by calling **free** for each of the *n* pointers in the argtable array and then nulling them for safety.

LITERAL OPTIONS (struct arg_lit)

Command line 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 (struct arg_int)

Command line 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 (struct arg_dbl)

Command line 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 (struct arg_str)

Command line 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

The **arg_str** struct contains the *count* of the number of times the option was present on the command line and a pointer (*sval*) to an array containing pointers to the parsed string values. The array is fixed at construction time to hold *maxcount* string pointers at most. These pointers in this array reference the actual command line string buffers stored in argv[], so the string contents should not be should not be altered. Although it is quite acceptable to set default string pointers in the *sval* array prior to calling arg_parse as the parser does alter them if no matching command line argument is received.

REGULAR EXPRESSION OPTIONS (struct arg_rex)

Command line examples

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

Data Structure

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

Constructor Functions

struct **arg_rex* arg_rex0** (const char *shortopts, const char *longopts, const char *pattern, const char *datatype, int flags, const char *glossary)

struct **arg_rex* arg_rex1** (const char *shortopts, const char *longopts, const char *pattern, const char *datatype, int flags, const char *glossary)

struct **arg_rex* arg_rexn** (const char *shortopts, const char *longopts, const char *pattern, const char *datatype, int mincount, int maxcount, int flags, const char *glossary)

Description

Like **arg_str** but but the string argument values are only accepted if they match a predefined regular expression. The regular expression is defined by the *pattern* parameter passed to the *arg_rex* constructor. The regular expression parsing is done using regex, and its behaviour can be controlled via standard regex bit flags which are passed to argtable via the *flags* parameter in the *arg_rex* conbstructors. However the only two regex flags that are relevant to argtable are REG_EXTENDED (use extended regular expressions rather than basic ones) and REG_ICASE (ignore case). These may be logically ORed if desired. This argument type is useful for matching command line keywords, particularly if case insensitive strings or pattern matching is required. See **regex(3)** for more details of regular expression matching.

Restrictions

Argtable does not support **arg_date** functionality under Microsoft Windows platforms as the Microsoft compilers do include the necessary **regex** support as standard.

FILENAME OPTIONS (struct arg_file)

Command line xamples

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

Data Structure

```
struct arg_file
  {
  struct arg_hdr hdr;
  int count;
  const char **filename;
  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.

DATE/TIME OPTIONS (struct arg_date)

Command line examples

12/31/04, -d 1982-11-28, --time 23:59

Data Structure

```
struct arg_date
  {
  struct arg_hdr hdr;
```

```
const char *format;
int count;
struct tm *tmval;
};
```

Constructor Functions

struct **arg_date* arg_date0** (const char *shortopts, const char *longopts, const char *format, const char *datatype, const char *glossary)

struct **arg_date* arg_date1** (const char *shortopts, const char *longopts, const char *format, const char *datatype, const char *glossary)

struct **arg_date* arg_daten** (const char *shortopts, const char *longopts, const char *format, const char *datatype, int mincount, int maxcount, const char *glossary)

Description

Accepts a timestamp string from the command line and converts it to *struct tm* format using the system **strptime** function. The time format is defined by the *format* string passed to the *arg_date* constructor, and is passed directly to **strptime**. See **strptime(3)** for more details on the format string.

Restrictions

Argtable does not support **arg_date** functionality under Microsoft Windows as the Microsoft compilers do include the necessary **strptime** support as standard.

REMARK OPTIONS (struct arg_rem)

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 OPTIONS (struct arg_end)

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

/usr/local/include/argtable2.h
/usr/local/lib/libargtable2.a
/usr/local/lib/libargtable2.so
/usr/local/man3/argtable2.3
/usr/local/share/doc/argtable2/
/usr/local/share/doc/argtable2/example/

AUTHOR

Stewart Heitmann <sheitmann@users.sourceforge.net>