Mu

Miscellaneous Utilities

Version 0.0, last updated 7 February 2024

Asher Gordon

This is the manual for Mu (Miscellaneous Utilities) version 0.0, last updated 7 February 2024.

Table of Contents

1 Introduction

Mu is a general purpose convenience library. It provides functions which perform common tasks, as well as some compatibility functions.

All code examples in this manual that can be compiled on their own are available in the `doc/examples` directory in the source distribution.

1.1 Terms and Notation Used in this Manual

There are several types of arguments dealt with in this manual: arguments to command line options; non-option, positional arguments passed on the command line; and parameters passed to functions. When we refer to arguments passed to command line options, we will use the term *argument* on its own. When we refer to non-option, positional arguments passed on the command line, we will use the term *positional argument*. When we refer to the value of an environment variable, we will use the term *value*. When we refer to parameters passed to functions, we will use the term *parameter*.

When referring to a field in a C `struct` or `union`, we will use the term *field*.

In examples, messages printed to standard output will be prefixed with “⊣”, while messages printed to standard error will be prefixed with “ error ”.

1.2 Using Mu in Your Program

Mu is written in C, and currently no bindings exist for other languages. So (for now at least) you can only use Mu in C programs.

Several header files are provided by Mu. Each one provides a different category of functions. All header files are installed in the `mu` subdirectory. So to include, for example, `compat.h`, write `#include <mu/compat.h>`.

To link with the library, use `-lmu` as an argument to the linker.

Note: Since Mu is released under the terms of the GNU General Public License, you may not use it in proprietary programs. If your program links with Mu, it must be licensed under the GNU GPL or a compatible license. Please see Appendix A [GNU General Public License], page 53, for more details.

Please note: Mu is not currently stable, and the API is subject to change. Feel free to use Mu, but please keep this in mind.

1.3 Reporting Bugs

Please report bugs to the bug tracker at Savannah (`https://savannah.nongnu.org/bugs/?group=libmu&func=additem`). You may also email bug reports to the `libmu-bug@nongnu.org` mailing list. See also the list information page for libmu-bug (`https://lists.nongnu.org/mailman/listinfo/libmu-bug`). Include enough information to reproduce the bug if possible, as well as the version of Mu, your machine architecture, operating system, etc. Make sure to read the documentation for the functions you are using, and ensure that you are using the functions correctly. You should also include any error messages if applicable, and a backtrace if you can. If possible, include a source file (preferably minimal) that causes the bug to occur.

If you are reporting a test failure (run by `make check`), include the file `tests/testsuite.log` in your report. Even if you're not reporting a test failure, it can still be helpful to run `make check` and include `tests/testsuite.log`.

For more information on writing effective bug reports, I suggest reading Simon Tatham's excellent essay, *How to Report Bugs Effectively* (`https://www.chiark.greenend.org.uk/~sgtatham/bugs.html`)

You can also send reports for bugs in this manual itself to the bug tracker (`https://savannah.nongnu.org/bugs/?group=libmu&func=additem`) or mailing list.

2 Parsing Options and Environment

Mu includes option parsing capability. Mu can parse command line options; both short options (a single dash followed by a letter, e.g., `-s`) and long options (two dashes followed by a multi-letter word, e.g., `--long`). Long options may also be specified with a single '`-`' as long as the flag `MU_OPT_BUNDLE` is not set (see Section 2.9 [Option Parsing Flags], page 31).

Mu also supports parsing the environment. See Section 2.8 [Parsing the Environment], page 28, for more information.

The option structure is fairly complicated, and its organization is subject to change. For that reason, it is highly recommended that you use designated structure initializers or set the values after declaration.

You can use designated initializers like this:

```
const MU_OPT options[] = {
  {
   .short_opt = "s",
   .long_opt  = "long",
   .has_arg   = MU_OPT_NONE
  },
  { 0 }
};
```

However, since designated initializers are only available in C99 and later (see Section "Designated Inits" in `gcc`), this may not be an option for you. The following is equivalent to the above example without using designated initializers:

```
const MU_OPT options[2] = { 0 };
options[0].short_opt = "s";
options[0].long_opt  = "long";
options[0].has_arg   = MU_OPT_NONE;
```

If you would like to use option parsing features, include `mu/options.h`.

`MU_OPT_CONTEXT` [Data Type]

This is an opaque context for parsing options. It is allocated using `mu_opt_context_new` and similar functions. To free it, you must use `mu_opt_context_free`. Both functions (among others) are described below.

`MU_OPT_CONTEXT * mu_opt_context_new (int` *`argc`*`, char **`*`argv`*`, const MU_OPT *`*`options`*`, int` *`flags`*`)` [Function]

`MU_OPT_CONTEXT * mu_opt_context_new_with_env (int` *`argc`*`, char **`*`argv`*`, char **`*`environment`*`, const MU_OPT *`*`options`*`, int` *`flags`*`)` [Function]

Allocate and return a new option parsing context. *argv* is the list of arguments to be parsed. *`argv`*`[0]` should be the name the program was invoked as, and the rest of *argv* should be the arguments given on the command line. *argc* is the length of *argv*. Normally, *argc* and *argv* should be used directly from `main`.

The returned context can also be used for parsing environment variables through the `env_var` field of the option structure (see Section 2.1 [Option Structure], page 5). When `mu_opt_context_new` is used to allocate the option parsing context, environment variables are parsed in the program environment (see Section "Environment Access" in `libc`). If you want to use an alternative environment, use `mu_opt_context_new_with_env` to allocate the option parsing context, in which case environment variables will be parsed in *environment*. *environment* can also be `NULL`, in which case environment variable parsing will be disabled. *environment* (or the program environment in the case of `mu_opt_context_new`) is never modified, unless, of course, any of the callbacks modify it (see Section 2.5 [Option Callbacks], page 18).

Normally, all arguments will be parsed at once when `mu_parse_opts` is called, and the context returned by these functions should only be passed to `mu_parse_opts` once. However, you can use `mu_opt_context_set_arg_callback` or the `MU_OPT_CONTINUE` flag if you care about the order in which options and arguments appear on the command line (see Section 2.10 [Ordered Option Parsing], page 33).

options is the list of options and environment variables that can occur in *argv* and *environment* (or the program environment). See Section 2.1 [Option Structure], page 5. Note that *options* is copied into the returned context, not used directly. Because of this, you need not worry about *options* going out of scope. For example, you might write a function which returns an option parsing context from a list of options which is local to that function's scope. Of course, you still need to ensure that no pointers referenced by any of the fields in *options* go out of scope (`cb_data` for example; see Section 2.1 [Option Structure], page 5).

flags is a bitmask of flags that effect how options are parsed (see Section 2.9 [Option Parsing Flags], page 31).

If you want to write a function which parses some options, but then leaves the rest for the caller to parse, consider using the `MU_OPT_ALLOW_INVALID` flag (see Section 2.9 [Option Parsing Flags], page 31). Note, however, that it may instead be better to use `mu_opt_context_add_options` (see below).

On error, this function returns `NULL` and the external variable `errno` will be set to indicate the error. For a function which terminates the program on error, use `mu_opt_context_xnew` and `mu_opt_context_xnew_with_env` (see Chapter 4 [Safety Functions], page 48).

`int mu_opt_context_free (MU_OPT_CONTEXT *`*`context`*`)` [Function]

Free the option context, *context*. For the `cb_data` fields in each option, the `cb_data_destructor` field (if any) will be used to free that data (see Section 2.1 [Option Structure], page 5, and Section 2.5 [Option Callbacks], page 18). If any of the destructors return nonzero, `mu_opt_context_free` will return nonzero as well. Otherwise, `mu_opt_context_free` will return zero.

Note that *all* the destructors are called, even if one or more of them return nonzero.

`int mu_parse_opts (MU_OPT_CONTEXT *`*`context`*`)` [Function]

Parse the options given in *context*. Use `mu_opt_context_new` or `mu_opt_context_new_with_env` (see above) to create the context. On success, the number of arguments parsed is returned. You can pass this value to `mu_shift_args` (see below). On error, an error code is returned, which can be detected by `MU_OPT_ERR` (see Section 2.12 [Option Parsing Errors], page 44).

Note that if `mu_opt_context_set_arg_callback` was called on *context*, the number of *positional* arguments will be included in the return value as well, if neither the `MU_OPT_PERMUTE` nor `MU_OPT_STOP_AT_ARG` flags are used. However, if either of these flags are used, the return value will *not* include the number of positional arguments parsed, even if `mu_opt_context_set_arg_callback` was called. This is so that you can shift the arguments by the return value and the remaining arguments will be the non-option positional arguments. Note that this would not be useful if neither the `MU_OPT_PERMUTE` flag nor the `MU_OPT_STOP_AT_ARG` flag was passed, because it would not be guaranteed that all the arguments left after shifting actually *were* the non-option positional arguments. For this reason, positional arguments are included in the return value when neither of these flags are passed. See Section 2.10 [Ordered Option Parsing], page 33.

`enum` *MU_OPT_WHERE* [Enumerated Type]

This type specifies whether to append or prepend options (see `mu_opt_context_add_options` below). Values of this type can be either of the following:

`MU_OPT_PREPEND`

Indicate that options should be prepended (before existing options).

`MU_OPT_APPEND`

Indicate that options should be appended (after existing options).

`int mu_opt_context_add_options (MU_OPT_CONTEXT *`*`context`*`, const MU_OPT *`*`options`*`, enum MU_OPT_WHERE` *`where`*`)` [Function]

Add *options* to *context*. If *where* is `MU_OPT_APPEND`, append *options* to the current options in *context*. Otherwise, if *where* is `MU_OPT_PREPEND`, prepend *options* instead. If *where* is neither `MU_OPT_APPEND` nor `MU_OPT_PREPEND`, `mu_opt_context_add_options` will return nonzero and `errno` will be set to `EINVAL`.

On success, this function returns zero. On error, this function returns nonzero and sets `errno` to indicate the error.

If you want to add options for printing usage information, use `mu_opt_context_add_help_options` (see Section 2.11 [Formatting Help], page 37).

`void mu_shift_args (int *`*`p_argc`*`, char ***`*`p_argv`*`, int` *`amount`*`)` [Function]

This function shifts the arguments in `*`*`p_argv`* by *amount* and subtracts *amount* from `*`*`p_argc`*. The old `(*`*`p_argv`*`)[0]` will be copied to the new `(*`*`p_argv`*`)[0]` after the shift is performed. It can be useful to call this function with the return value of `mu_parse_opts` passed as *amount*[1] and *p_argc* and *p_argv* as the addresses of *argc* and *argv* respectively, as passed to `mu_opt_context_new`.

2.1 Option Structure

`MU_OPT` [Data Type]

This structure specifies a single option, the arguments the option takes, and the actions to perform when the option is found. If all fields are `0`, that will indicate that this is the end of the options list.

Unless otherwise specified, these fields may be used both in regular options and suboptions. If an option takes suboptions as arguments, there are some fields which it

[1] But first you should make sure the return value is not an error code (see Section 2.12 [Option Parsing Errors], page 44).

may not use. Likewise, if an option *does not* take suboptions as arguments, there are a few fields which *it* may not use. See below for details. See Section 2.7 [Parsing Suboptions], page 25, for more information on suboptions.

`const char *category`

This field, if used, is a category for the options following the one in which this field appears. It has no effect on option parsing, only on help and man output (see Section 2.11 [Formatting Help], page 37). This field may not contain newlines (‘`\n`’).

If this field is the empty string, the following options will be separated by a newline in help output, and it will have no effect in man page output.

If this field is used, it must be the only field used. You may not set any other fields if this field is set.

For an example of how this field is used, see Section 2.11 [Formatting Help], page 37.

`const char *short_opt`

This is the short option character if any, possibly including aliases, or `NULL` if this option does not have a short option equivalent.

You may specify multiple aliases by simply including more characters in the `short_opt` (see Section 2.2 [Option Aliases], page 10). Note that if `short_opt` is not `NULL`, it must be terminated by a null byte.

This field must not be used in suboptions (see Section 2.7 [Parsing Suboptions], page 25).

`const char *long_opt`

This is the long option string (without leading dashes), or `NULL` if this option does not have a short option equivalent. `long_opt` must not contain the ‘`=`’ character, because that is used for passing arguments.

You may specify aliases separated by ‘`|`’ (see Section 2.2 [Option Aliases], page 10).

This field must not be used in suboptions (see Section 2.7 [Parsing Suboptions], page 25).

When matching against `long_opt`, abbreviation is allowed as long as it is unambiguous.

`const char *subopt_name`

This is the name of the suboption. Like `long_opt`, it may not contain the ‘`=`’ character. Also like `long_opt`, matching allows abbreviation as long as it is not ambiguous.

You may specify aliases separated by ‘`|`’ (see Section 2.2 [Option Aliases], page 10).

This field must only be used in suboptions (see Section 2.7 [Parsing Suboptions], page 25).

`const char *env_var`

This is the name of an environment variable. Environment variables act exactly like options, except that they are passed in the environment rather than on the command line. Like `long_opt`, `env_var` must not contain the '`=`' character, because that is used for indicating the value of an environment variable.

Unlike `long_opt` and `subopt_name` (above), abbreviation is **not** allowed.

You may specify aliases separated by '`|`' (see Section 2.2 [Option Aliases], page 10).

See Section 2.8 [Parsing the Environment], page 28, for more information about parsing the environment with `mu_parse_opts` and Section "Environment Variables" in `libc` for more information on environment variables in general.

`enum MU_OPT_HAS_ARG has_arg`

This specifies whether the option takes an argument or not, and whether the argument is optional or required if the option does take an argument. `has_arg` can have the value `MU_OPT_NONE` if the option takes no argument, `MU_OPT_OPTIONAL` if the option may optionally take an argument, or `MU_OPT_REQUIRED` if the option requires an argument. See Section 2.4 [Option Arguments], page 14, for more information on how required and optional arguments are parsed and handled differently.

`enum MU_OPT_ARG_TYPE arg_type`

The type of the argument if `has_arg` is not `MU_OPT_NONE`. See Section 2.4.1 [Option Argument Types], page 15.

`int negatable`

If this is nonzero, the option may be negated. For short options, this means using '`+`' instead of '`-`', and for long options and suboptions, it means prefixing the option with '`no-`' or the specified negation prefixes (see Section 2.3.1 [Negation Prefixes], page 13). Environment variables may not be negated. See Section 2.3 [Negatable Options], page 11, for more details.

This field may only be used if `has_arg` is `MU_OPT_NONE`.

`int *found_opt`

If `found_opt` is not `NULL`, `*found_opt` will be set to `1` if the option was found, or `0` if the option was not found.

`int *found_arg`

If `found_arg` is not `NULL`, `*found_arg` will be set to `1` if an argument to the option was found, or `0` if no argument was found.

`void *arg` If `arg` is not `NULL`, `*arg` will be set to the argument if an argument was found. To test if an argument was found, use `found_arg`. The type of the argument is determined by `arg_type` (see Section 2.4.1 [Option Argument Types], page 15).

This field must only be used if `has_arg` is not `MU_OPT_NONE` and `arg_type` is not `MU_OPT_SUBOPT` (see Section 2.4 [Option Arguments], page 14, and Section 2.7 [Parsing Suboptions], page 25).

`int bool_default`
`long int_default`
`double float_default`
`const char *string_default`
`FILE *file_default`
`DIR *dir_default`
`int enum_default`

The default values for `*arg` (see above) if the option is not found. `bool_default` should be used for arguments of type `MU_OPT_BOOL`, `int_default` should be used for arguments of type `MU_OPT_INT` and so on. See Section 2.4.1 [Option Argument Types], page 15, for more information.

These fields may only be used if `has_arg` is not `MU_OPT_NONE`.

`const char **argstr`

If `argstr` in not `NULL`, `*argstr` will be set to the raw, unprocessed argument unless otherwise specified in Section 2.4.1 [Option Argument Types], page 15. `*argstr` is equal to `*arg` if and only if `arg_type` is `MU_OPT_STRING`.

This field must only be used if `has_arg` is not `MU_OPT_NONE` and `arg_type` is not `MU_OPT_SUBOPT` (see Section 2.4 [Option Arguments], page 14, and Section 2.7 [Parsing Suboptions], page 25).

`int (*callback_none) (void *, char *)`
`int (*callback_negatable) (int, void *, char *)`
`int (*callback_bool) (int, int, void *, char *)`
`int (*callback_int) (int, long, void *, char *)`
`int (*callback_float) (int, double, void *, char *)`
`int (*callback_string) (int, const char *, void *, char *)`
`int (*callback_file) (int, const char *, FILE *, void *, char *)`
`int (*callback_directory) (int, const char *, DIR *, void *, char *)`
`int (*callback_enum) (int, int, void *, char *)`
`int (*callback_subopt) (int, void *, char *)`

If the corresponding callback for `arg_type` is set (see Section 2.4.1 [Option Argument Types], page 15), it will be called when the option is found. See Section 2.5 [Option Callbacks], page 18, for a description of the arguments these callbacks take and their return values.

Note: only one of the callbacks may be set at a time.

`callback_none` may only be used if `has_arg` is `MU_OPT_NONE` and `negatable` is zero. `callback_negatable` may only be used if `has_arg` is `MU_OPT_NONE` and `negatable` is nonzero.

`void *cb_data`

The *data* argument to pass to the above callbacks. See Section 2.5 [Option Callbacks], page 18.

This field must not be used if `arg_type` is `MU_OPT_SUBOPT` (see Section 2.7 [Parsing Suboptions], page 25, and Section 2.4.1 [Option Argument Types], page 15).

`int (*cb_data_destructor) (void *data)`

If `cb_data` contains dynamically allocated data or anything else that needs to be released back to the system (e.g., file descriptors), set `cb_data_destructor` to a function which will release all of that data, including `cb_data` itself if it was dynamically allocated as well. If an error occurred, e.g., when closing a file descriptor, `cb_data_destructor` should return nonzero.

As a simple example, if `cb_data` was dynamically allocated but does not contain dynamically allocated data, you can set this to a function which will call `free(data)` and return `0`.

This field must not be used if `arg_type` is `MU_OPT_SUBOPT` (see Section 2.7 [Parsing Suboptions], page 25, and Section 2.4.1 [Option Argument Types], page 15).

`long ibound.lower`
`long ibound.upper`

Lower and upper bounds (inclusive) for integer arguments. If you don't want any bounds, set `ibound.lower` to `LONG_MIN` and `ibound.upper` to `LONG_MAX`.

These fields must only be used if `arg_type` is `MU_OPT_INT` (see Section 2.4.1 [Option Argument Types], page 15).

`double fbound.lower`
`double fbound.upper`

Lower and upper bounds (inclusive) for floating point arguments. If you don't want any bounds, set `fbound.lower` to `-HUGE_VAL` and `fbound.upper` to `HUGE_VAL`.

These fields must only be used if `arg_type` is `MU_OPT_FLOAT` (see Section 2.4.1 [Option Argument Types], page 15).

`const char *file_mode`

The file mode to pass to `fopen` when opening a file argument. See Section "Opening Streams" in `libc`.

This field must only be used if `arg_type` is `MU_OPT_FILE` (see Section 2.4.1 [Option Argument Types], page 15).

`const MU_ENUM_VALUE *enum_values`

The enumeration specification. See Section 2.6 [Parsing Enums], page 22, for more information.

This field must only be used if `arg_type` is `MU_OPT_ENUM` (see Section 2.4.1 [Option Argument Types], page 15).

`int enum_case_match`

If this is nonzero, enumerated arguments will be matched against the values in `enum_values` case sensitively. Otherwise, matching will be case insensitive. See Section 2.6 [Parsing Enums], page 22, for more information.

This field must only be used if `arg_type` is `MU_OPT_ENUM` (see Section 2.4.1 [Option Argument Types], page 15).

`const MU_OPT *subopts`

This is a list of valid suboptions for this option. See Section 2.7 [Parsing Suboptions], page 25.

This field must only be used if `arg_type` is `MU_OPT_SUBOPT` (see Section 2.4.1 [Option Argument Types], page 15).

`const char *arg_help`

This is a string which will be displayed in the help message as the argument for your option (see Section 2.11 [Formatting Help], page 37). For example, '`FILE`', '`NAME`', or, if you're not feeling very imaginative, '`ARG`'. This field may not contain newlines ('`\n`').

If you leave this as `NULL`, a default will be chosen based on `arg_type` (see Section 2.4.1 [Option Argument Types], page 15).

`const char *help`

The full help text for your option, used when formatting the help message (see Section 2.11 [Formatting Help], page 37). This text may make references to the string passed in `arg_help`. There should be no newlines in this string (even if it is quite long[2]) unless you really want a line break in a certain place. Normally, you should just let line wrapping happen automatically.

If this field is left as `NULL`, the option will not be documented in either help or `man` output (see Section 2.11 [Formatting Help], page 37). Of course, the option will still be parsed as usual.

`const char *negated_help`

The help text for the negated option. If this is left as `NULL` and `help` (see above) is not `NULL`, it will default to '`negate` *`option`*', where *option* is the non-negated option or suboption (including aliases). However, if `negated_help` is `NULL` and `help` is also `NULL`, neither the option nor the negated option will be documented. If `negated_help` is non-`NULL` but `help` is `NULL`, only the negated option will be documented.

This field may only be used if `has_arg` is `MU_OPT_NONE` and `negatable` is nonzero.

2.2 Aliases for Options and Environment Variables

The fields `long_opt`, `subopt_name`, and `env_var` of the `MU_OPT` structure (see Section 2.1 [Option Structure], page 5) allow aliases separated by '`|`'. The `short_opt` field allows aliases to be specified as multiple characters in the string (which must be terminated by a null byte).

For example, the string '`abc`', when specified as the `short_opt` field, indicates three equivalent short options: `-a`, `-b`, and `-c`. In the `long_opt` field, '`foo|bar|baz`' would specify three equivalent long options: `--foo`, `--bar`, and `--baz`. The same goes for `subopt_name` and `env_var` (but see Section 2.8 [Parsing the Environment], page 28, for more information on environment variable aliases).

[2] You should try to keep the help text fairly short, though.

Duplicate aliases are not allowed, and will be diagnosed as an error. For example, in the short_opt field, 'abcb' will be diagnosed because the 'b' is repeated. Likewise, 'foo|bar|foo' would be diagnosed in any of the long_option, subopt_name, or env_var fields. Empty aliases are diagnosed as well (including the entire string being empty). So 'foo||bar' would be diagnosed in any of the long_option, subopt_name, or env_var fields, because there is an empty alias between 'foo' and 'bar'. In any of the same fields, in addition to short_opt, the empty string ('') will be diagnosed as well. Leave a field as NULL if you don't want any options (or environment variables) of that type.

2.3 Negatable Options

A *negatable* option is an option which can be specified later on the command line in a different form, to negate the effect of a previous specification. Short options are negated using '+' rather than '-' to specify the option. Long options and suboptions must be prefixed with 'no-' or the specified negation prefixes (see Section 2.3.1 [Negation Prefixes], page 13). For example,

```
$ prog --foo --no-foo
```

should act as though --foo were never specified. Only negatable options can be negated. For an option to be negatable, its negatable field must be set to a nonzero value (see Section 2.1 [Option Structure], page 5). Options that take arguments (i.e., options for which the has_arg field is not MU_OPT_NONE) may not be negated. Indeed, setting the negatable field to *any* value for an option which takes an argument results in undefined behavior.

Environment variables may not be negated. The reason for this is because it is not easy to control the order in which environment variables appear. Thus, if environment variable negation were allowed and FOO were a negatable environment variable,

```
$ FOO= NO_FOO= prog
```

may or may not act as though FOO were specified. So if an option for which the negatable field is nonzero also has a non-NULL env_var field, NO_FOO will be ignored. Note, however, that the callback_negatable callback should still be used (but it may be better not to use callbacks at all; see below). Rather than having an env_var field for a negatable option, it is instead better to make a separate environment variable that has a boolean value (see Section 2.4.1 [Option Argument Types], page 15).

Since environment variables may not be negated, specifying the negatable field for an environment variable which has no equivalent options is useless. Because of this, it is not allowed and will be diagnosed.

When option parsing is finished, the value that the found_opt field points to (if any) will be nonzero if the last instance of the option found on the command line was not negated, or zero if it was negated.

Negatable options should use the callback_negatable field if they are using a callback (see Section 2.5 [Option Callbacks], page 18), although it is usually preferable not to use a callback. Suppose you have a certain negatable option, and you want to, say, open a file when it is found. If you used a callback, you would need to open the file whenever *value* was nonzero, and then close it again when it is zero. Although in this case this would be fairly easy to implement (although far from ideal), it is still much better to simply wait until option parsing is finished, and then check the value that the `found_opt` field points to.

Here is an example illustrating how to parse negatable options:

```
#include <stdio.h>
#include <mu/options.h>
#include <mu/safe.h>            /* For mu_opt_context_x{new,free} */

/* Print a message when we find the negatable option. Usually, we
   shouldn't use a callback for negatable options, but we are just
   using it to print a message. */
static int print_negatable(int value, void *data, char *err) {
  printf("    Found the negatable option, and it was%s negated.\n",
         value ? " not" : "");
  return 0;
}

int main(int argc, char **argv) {
  int found_negatable;
  int ret;
  const MU_OPT options[] = {
    {
     .short_opt          = "n",
     .long_opt           = "negatable",
     .has_arg            = MU_OPT_NONE,
     .negatable          = 1,
     .found_opt          = &found_negatable,
     .callback_negatable = print_negatable
    },
    { 0 }
  };
  MU_OPT_CONTEXT *context;

  /* Parse the options. */
  context = mu_opt_context_xnew(argc, argv, options, MU_OPT_PERMUTE);
  ret = mu_parse_opts(context);
  mu_opt_context_xfree(context);
  if (MU_OPT_ERR(ret))
    return 1;

  printf("It appears that the negatable option was%s given.\n",
         found_negatable ? "" : " not");

  return 0;
}
```

Here is the output of the above program:

```
$ ./option-negatable
⊣ It appears that the negatable option was not given.
$ ./option-negatable --negatable -n --no-negatable +n
⊣     Found the negatable option, and it was not negated.
⊣     Found the negatable option, and it was not negated.
⊣     Found the negatable option, and it was negated.
⊣     Found the negatable option, and it was negated.
⊣ It appears that the negatable option was not given.
$ ./option-negatable -n +n --negatable
⊣     Found the negatable option, and it was not negated.
⊣     Found the negatable option, and it was negated.
⊣     Found the negatable option, and it was not negated.
⊣ It appears that the negatable option was given.
```

2.3.1 Negation Prefixes

By default, long options and suboptions are negated by prefixing them with '`no-`' (see Section 2.3 [Negatable Options], page 11). However, alternative negation prefixes may be specified as well. For example, you might want to parse options in a similar style to XBoard, with options negated by a single '`x`' (see Section "Options" in `xboard`).

Negation prefixes, like regular options, are case sensitive. Thus, if you have a negation prefix of '`no-`', '`No-`' will not be recognized (or will be treated as a separate option).

`int mu_opt_context_set_no_prefixes (MU_OPT_CONTEXT *`*`context`*`, ...)` [Function]
`int mu_opt_context_set_no_prefix_array (MU_OPT_CONTEXT *`*`context`*`, char **`*`strings`*`)` [Function]
`int mu_subopt_context_set_no_prefixes (MU_SUBOPT_CONTEXT *`*`context`*`, ...)` [Function]
`int mu_subopt_context_set_no_prefix_array (MU_SUBOPT_CONTEXT *`*`context`*`, char **`*`strings`*`)` [Function]

> Set a list of negation prefixes in *context*. In the case of `mu_opt_context_set_no_prefixes` and `mu_subopt_context_set_no_prefixes`, the negation prefixes are specified in the variable arguments. In the case of `mu_opt_context_set_no_prefix_array` and `mu_subopt_context_set_no_prefix_array`, the negation prefixes are specified in *strings*. In both cases, the list of negation prefixes must be terminated by `NULL`.
>
> Duplicate negation prefixes are not allowed. If duplicates are present in *strings* or the variable arguments, `errno` will be set to `EINVAL` and these functions will return nonzero.
>
> Subsequent calls to these functions are allowed, but will overwrite negation prefixes set by previous calls. However, it is **not** allowed to call these functions after *context* has been passed to `mu_parse_opts` or `mu_parse_subopts`.

Here is an example of how alternative negation prefixes may be used:

```
#include <stdio.h>
#include <mu/options.h>
#include <mu/safe.h>            /* For mu_opt_context_x* */

/* Print a message when we find the negatable option. */
static int print_negatable(int value, void *data, char *err) {
  printf("    Found the negatable option, and it was%s negated.\n",
         value ? " not" : "");
  return 0;
}

int main(int argc, char **argv) {
  int ret;
  const MU_OPT options[] = {
    {
     .short_opt          = "n",
     .long_opt           = "negatable",
     .has_arg            = MU_OPT_NONE,
      .negatable          = 1,
      .callback_negatable = print_negatable
    },
    { 0 }
  };
  MU_OPT_CONTEXT *context;

  context = mu_opt_context_xnew(argc, argv, options, MU_OPT_PERMUTE);

  /* Set the negation prefixes. This must be done *before*
     mu_parse_opts() is called. */
  mu_opt_context_xset_no_prefixes(context, "negate-", "no-", "x", NULL);

  /* Now parse the options. */
  ret = mu_parse_opts(context);
  mu_opt_context_xfree(context);
  if (MU_OPT_ERR(ret))
    return 1;

  return 0;
}
```

And the output:

```
$ ./negation-prefixes --negatable
⊣     Found the negatable option, and it was not negated.
$ ./negation-prefixes --negate-negatable
⊣     Found the negatable option, and it was negated.
$ ./negation-prefixes --no-negatable
⊣     Found the negatable option, and it was negated.
$ ./negation-prefixes --xnegatable
⊣     Found the negatable option, and it was negated.
$ ./negation-prefixes --foo-negatable
error   ./negation-prefixes: '--foo-negatable': invalid option
```

2.4 Option Arguments

Options can take arguments and environment variables can have values (see Section 2.8 [Parsing the Environment], page 28). Some options and environment variables require arguments or values, while others may optionally take arguments or have values, while still others might take no arguments at all or not allow any values. Mu supports all of these types of options and environment variables through the `has_arg` field of the `MU_OPT` structure (see Section 2.1 [Option Structure], page 5).

`enum MU_OPT_HAS_ARG` [Enumerated Type]

`MU_OPT_NONE`

This indicates that an option takes no arguments. If the `MU_OPT_BUNDLE` flag (see Section 2.9 [Option Parsing Flags], page 31) is *not* specified, then any text following a short option which doesn't take an argument will be an error.[3] An '`=`' is an error in a long option that does not take an argument, because '`=`' is used to specify arguments to long options.

`*has_arg` will always be set to `0` if `has_arg` is not `NULL`.

[3] But only if the text following the short option is in the same argument as the option itself, e.g., `-o`*`foo`*, not `-o` *`foo`*. If the following text is in a new argument, as in the latter case, it will be treated as a positional argument, not an argument to `-o`.

`MU_OPT_OPTIONAL`

This indicates that an option may take an argument, but that the option doesn't require an argument. Even if the `MU_OPT_BUNDLE` flag is passed (see Section 2.9 [Option Parsing Flags], page 31), short options with optional arguments may not be bundled except as the last option in a bundle. The reason for this is as follows: Suppose short option `b` takes an optional argument. And suppose short options `a` and `c` take no argument. Now what should `-abc` mean (assuming `MU_OPT_BUNDLE` was passed)? Is it three options without arguments, `a`, `b`, and `c`? Or is it two options, `a` and `b`, the latter of which taking an argument, '`c`'? It is in fact the latter, two options `a` and `b`, with `b` taking an argument, '`c`'. Note, however, that if `b` is specified as the last option like so: `-acb`, there is no ambiguity, because there is nothing following the `b` option (and if there is text following it in another argument, it will be treated as a positional argument; see below).

Short options taking an optional argument *must* have their arguments specified with the option itself. For example, if a short option, `b`, takes an optional argument, it must be specified as `-b`*`arg`*, not `-b` *`arg`*. The reason for this is because `-b` *`arg`* could be a short option `b` with an argument *arg*, or a short option `b` with no argument, and a positional argument *arg*. Likewise, long options taking optional arguments must be specified as `--long=`*`arg`*, not `--long` *`arg`*.

If `has_arg` is not `NULL`, then `*has_arg` will be set to `1` if the option has an argument, or `0` if the option doesn't have an argument.

`MU_OPT_REQUIRED`

This indicates that an option requires an argument. If no argument is specified, it is an error. Like short options with optional arguments, short options with required arguments may not be bundled except as the last option in a bundle. See above for an explanation.

For short options with required arguments, the argument may be passed with the option itself like so: `-r`*`arg`*, or immediately after the argument like so: `-r` *`arg`*. Arguments to long options with required arguments may also be specified with the option itself like so: `--required=`*`arg`*, or immediately after the argument like so: `--required` *`arg`*.

`*has_arg` will always be set to `1` if `has_arg` is not `NULL`.

2.4.1 Option Argument Types

Mu supports several option types, and more may be added in the future. These types will automatically be processed from the string argument, and the processed argument will be returned in `*arg` if `arg` is not `NULL` (see Section 2.1 [Option Structure], page 5). `arg` should be a pointer to a value of the type indicated by the `arg_type` field, which must be one of the values in the table below. However, if `arg_type` is `MU_OPT_SUBOPT`, `arg` must not be used (see Section 2.7 [Parsing Suboptions], page 25). If an error occurs while processing an argument, an error message will be printed to standard error, and `mu_parse_opts` will return an error code (see Section 2.12 [Option Parsing Errors], page 44).

Unless otherwise specified, `*argstr` will be set to the unprocessed string argument if `argstr` is not `NULL` (see Section 2.1 [Option Structure], page 5). However, if `arg_type` is `MU_OPT_SUBOPT`, `argstr` must not be used.

You can also parse your own types using callbacks (see Section 2.5 [Option Callbacks], page 18).

`enum MU_OPT_ARG_TYPE` [Enumerated Type]

`MU_OPT_BOOL`

This is a boolean value. The type of `*arg` should be `int` and `bool_default` should be used for the default value (see Section 2.1 [Option Structure], page 5). If `MU_OPT_BOOL` is given in the `arg_type` field, the argument can either be '`yes`' or '`true`' for a true value, or '`no`' or '`false`' for a false value. Matching is case insensitive and allows abbreviation.

If the argument is none of '`yes`', '`no`', '`true`', or '`false`', it will be parsed as an integer (see below). Zero is false and any other integer is true.

If the argument is not an integer either, that will be an error.

`MU_OPT_INT`

This is an integer value. The type of `*arg` should be `long` and `int_default` should be used for the default value (see Section 2.1 [Option Structure], page 5). The radix (or base) that the argument is parsed as depends on the first non-whitespace characters after an optional '`+`' or '`-`' sign. If these characters are '`0x`' or '`0X`', the integer is parsed as hexadecimal. Otherwise, if the first character is '`0`', and the following character is *not* '`x`' or '`X`', the integer will be parsed as octal. Otherwise, the integer will be parsed as decimal. See Section "Parsing of Integers" in `libc` for more information on how integers are parsed.

If the parsed integer is outside the bounds specified by the `ibound` field (see Section 2.1 [Option Structure], page 5), then that will be treated as an error.

`MU_OPT_FLOAT`

This is a floating-point value. The type of `*arg` should be `double` and `float_default` should be used for the default value (see Section 2.1 [Option Structure], page 5). The radix (or base) that the argument is parsed as depends on the first non-whitespace characters after an optional '`+`' or '`-`' sign. If these characters are '`0x`' or '`0X`', the number is parsed as hexadecimal. Otherwise, if the first character is '`0`', and the following character is *not* '`x`' or '`X`', the number will be parsed as octal. Otherwise, the number will be parsed as decimal. See Section "Parsing of Floats" in `libc` for more information on how floating-point numbers are parsed. See Section "Parsing of Floats" in `libc` for more information on how floating-point numbers are parsed.

If the parsed floating-point numebr is outside the bounds specified by the `fbound` field (see Section 2.1 [Option Structure], page 5), then that will be treated as an error.

MU_OPT_STRING

This is a string value. The type of *arg should be const char * and string_default should be used for the default value (see Section 2.1 [Option Structure], page 5) (i.e., the type of arg should be const char **). *argstr (if argstr is not NULL) will be set to the same value as *arg.

MU_OPT_FILE

This is a file argument. The type of *arg should be FILE * (i.e., the type of arg should be FILE **) and file_default should be used for the default value (see Section 2.1 [Option Structure], page 5). file_mode should be used for this type and only for this type (see Section 2.1 [Option Structure], page 5).

file_mode describes the mode to use when opening the file, and how '-' should be handled. If file_mode indicates that the file should be opened in read-only mode, '-' will be handled as standard input. If file_mode indicates that the file should be opened in write-only mode, '-' will be handled as standard output. If file_mode indicates that the file should be opened for both reading and writing, '-' will cause an error.

If argstr is not NULL, *argstr will be set to '<stdin>' if '-' was handled as standard input, '<stdout>' if '-' was handled as standard output, or the file name given as the argument to the option if the argument was not '-'.

If an error occurs while opening a file, an error message will be printed to standard error and mu_parse_opts will return an error code (see Section 2.12 [Option Parsing Errors], page 44).

For more information on how files are opened and how file_mode is parsed, see Section "Opening Streams" in libc.

MU_OPT_DIRECTORY

This is a directory argument. The type of *arg should be DIR * (i.e., the type of arg should be DIR **) and dir_default should be used for the default value (see Section 2.1 [Option Structure], page 5). See Section "Opening a Directory Stream" in libc for more information on how directories are opened.

If you'd like to know the name of the directory as given as the argument to the option, you can use the argstr field (see Section 2.1 [Option Structure], page 5).

If an error occurs while opening a directory, an error message will be printed to standard error and mu_parse_opts will return an error code (see Section 2.12 [Option Parsing Errors], page 44).

MU_OPT_ENUM

This is an enumerated argument. The enumeration specification is the enum_values field (see Section 2.1 [Option Structure], page 5).

For more information, See Section 2.6 [Parsing Enums], page 22.

`MU_OPT_SUBOPT`
: This indicates that the option takes suboptions as arguments. Suboptions may not take suboptions as arguments. See Section 2.7 [Parsing Suboptions], page 25, for more information.

2.5 Option Callbacks

Option callbacks are useful when you have more advanced option parsing needs. Each option argument type has a different callback. There are also callbacks for options which don't take arguments: `callback_none` for non-negatable options, and `callback_negatable` for negatable options (see Section 2.3 [Negatable Options], page 11). All callback names mentioned are members of the `MU_OPT` structure.

For a callback that is called when a *positional* argument is seen, use `mu_opt_context_set_arg_callback` (see Section 2.10 [Ordered Option Parsing], page 33).

The callback names and prototypes for each argument type are listed below (although `MU_OPT_NONE` is not a type, and should be passed in the `has_arg` field of the `MU_OPT` structure, not the `arg_type` field):

`MU_OPT_NONE`
: If the `negatable` field is zero (see Section 2.1 [Option Structure], page 5):

    int (*callback_none) (void **data*, char **err*)

    Otherwise, if `negatable` is nonzero:

    int (*callback_negatable) (int *value*, void **data*, char **err*)

`MU_OPT_BOOL`
: int (*callback_bool) (int *has_arg*, int *arg*, void **data*, char **err*)

`MU_OPT_INT`
: int (*callback_int) (int *has_arg*, long *arg*, void **data*, char **err*)

`MU_OPT_FLOAT`
: int (*callback_float) (int *has_arg*, double *arg*, void **data*, char **err*)

`MU_OPT_STRING`
: int (*callback_string) (int *has_arg*, const char **arg*, void **data*, char **err*)

`MU_OPT_FILE`
: int (*callback_file) (int *has_arg*, const char **filename*, FILE **file*, void **data*, char **err*)

`MU_OPT_DIRECTORY`
: int (*callback_directory) (int *has_arg*, const char **dirname*, DIR **directory*, void **data*, char **err*)

`MU_OPT_ENUM`
: int (*callback_enum) (int *has_arg*, int *arg*, void **data*, char **err*)

`MU_OPT_SUBOPT`
: int (*callback_subopt) (int *has_arg*, void **data*, char **err*)

A callback will be called as soon as an option is found, so callbacks are guaranteed to be called in the same order as the options appear on the command line. This means that if an option takes suboptions as arguments, the callback for the main option will be called before the callbacks for the suboptions (see Section 2.7 [Parsing Suboptions], page 25). The *has_arg* parameter will be passed as `1` if the option has an argument, or `0` if the option doesn't have an argument (except for `callback_none` which doesn't have a *has_arg* parameter).

If the option has an argument, *arg* will be set to that argument, except in the case of suboptions (see Section 2.7 [Parsing Suboptions], page 25). In the case of `callaback_file` and `callback_directory`, *filename* or *dirname* will be set to the name of the file or directory respectively.[4] For `callback_negatable`, *value* will be zero if the option was negated, or nonzero if it wasn't (see Section 2.3 [Negatable Options], page 11).

If you need to provide extra information to a callback, provide it in the `cb_data` field of the `MU_OPT` structure. This will then be passed as the *data* parameter to a callback. Note: a callback should *not* free this parameter even if it is dynamically allocated. In the case that `cb_data` is dynamically allocated and/or contains dynamically allocated data, you should also set the `cb_data_destructor` field to a function which will free all dynamically allocated data in `cb_data`.

When you call `mu_opt_context_free` (see Chapter 2 [Parsing Options and Environment], page 3) or `mu_subopt_context_free` (see Section 2.7 [Parsing Suboptions], page 25), each `cb_data_destructor` field is called with the corresponding `cb_data` in order to free that data. If an error occurs while freeing callback data (for example, an error closing a file), `cb_data_destructor` should return nonzero. Otherwise, `cb_data_destructor` should return zero.

For `callback_file` and `callback_directory`, the *file* or *directory* argument **will be closed** after the callback returns if you leave the `arg` field of the `MU_OPT` structure as `NULL` (see Section 2.1 [Option Structure], page 5). So you must not close the *file* or *directory* argument in the callback.

You also must **not** use the `cb_data` field to get the opened file/directory. For example, the following code is *wrong*:

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <mu/options.h>
#include <mu/safe.h>            /* For mu_opt_context_x{new,free} */

int file_callback(int has_arg, const char *filename,
                  FILE *file, void *data, char *err) {
  /* Make sure the file is named "foo". */
  if (strcmp(filename, "foo")) {
    snprintf(err, MU_OPT_ERR_MAX, "file is not named \"foo\"");
    return 1;
  }

  /* It is named "foo"; return `file' in `*data'.
     This is WRONG! Do not do this! */
  *(FILE **)data = file;

  return 0;
```

[4] However, for `callback_file`, *filename* might be '`<stdin>`' or '`<stdout>`' when *file* is standard input or standard output respectively. See Section 2.4.1 [Option Argument Types], page 15.

```
}

int main(int argc, char **argv) {
  FILE *file = NULL;
  char buf[256];
  size_t size;
  int ret;
  const MU_OPT options[] = {
    {
     .short_opt     = "f",
     .long_opt      = "file",
     .has_arg       = MU_OPT_REQUIRED,
     .arg_type      = MU_OPT_FILE,
     .file_mode     = "r",
     .callback_file = file_callback,
     .cb_data       = &file
    },
    { 0 }
  };
  MU_OPT_CONTEXT *context;

  /* Parse the options. */
  context = mu_opt_context_xnew(argc, argv, options, MU_OPT_PERMUTE);
  ret = mu_parse_opts(context);
  mu_opt_context_xfree(context);
  if (MU_OPT_ERR(ret))
    return 1;

  if (!file) {
    /* We weren't passed the `-f' option. */
    return 0;
  }

  /* Read the file. This invokes UNDEFINED BEHAVIOR because
     `file' was already closed by `mu_parse_opts'! */
  size = fread(buf, sizeof(*buf), sizeof(buf), file);
  if (ferror(file)) {
    fprintf(stderr, "%s: cannot read foo: %s\n",
            argv[0], strerror(errno));
    return 1;
  }
  fclose(file);

  /* Print the contents of the file to standard output. */
  fwrite(buf, sizeof(*buf), size, stdout);

  return 0;
}
```

When this program is run, it invokes undefined behavior. The correct way to do this is to *not* use the `cb_data` field, and instead set the `arg` field to `&file`. This way, `mu_parse_opts` will not close the file or directory after the callback returns.

If a callback needs to indicate an error (if its argument is in the wrong format, for example), it should return nonzero. Otherwise, on success, it should return `0`. If a callback returns nonzero, you must write an error string to *err* which will then be used by `mu_parse_opts` to print an error message. You must not write more than `MU_OPT_ERR_MAX` characters to *err* (including the terminating null byte). However, if you write exactly `MU_OPT_ERR_MAX` bytes to *err*, you need not terminate *err* with a null byte.

Below is an example of how to use callbacks. Of course, this trivial example would be better expressed using enumerated argument parsing (see Section 2.6 [Parsing Enums], page 22).

```
#include <stdio.h>
#include <string.h>
#include <mu/options.h>
#include <mu/safe.h>            /* For mu_opt_context_x{new,free} */

enum selection { FOO, BAR, BAZ };

/* Parse a selection. `has_arg' will always be true because the option
   takes a required argument. */
int parse_selection(int has_arg, const char *arg,
                    void *data, char *err) {
  enum selection sel;

  if (!strcmp(arg, "foo"))
    sel = FOO;
  else if (!strcmp(arg, "bar"))
    sel = BAR;
  else if (!strcmp(arg, "baz"))
    sel = BAZ;
  else {
    /* `err' will be used by `mu_parse_opts' to print an error
       message. */
    snprintf(err, MU_OPT_ERR_MAX, "invalid selection: %s", arg);
    /* Indicate to `mu_parse_opts' that an error occured by returning
       a nonzero value. */
    return 1;
  }

  /* Store the selection in `*data'. */
  *(enum selection *)data = sel;

  /* Success! */
  return 0;
}

int main(int argc, char **argv) {
  enum selection sel;
  int found_sel;
  int ret;
  const MU_OPT options[] = {
    {
     .short_opt       = "s",
     .long_opt        = "selection",
     .has_arg         = MU_OPT_REQUIRED,
     .arg_type        = MU_OPT_STRING,
     .found_arg       = &found_sel,
     .callback_string = parse_selection,
     .cb_data         = &sel
    },
    { 0 }
  };
  MU_OPT_CONTEXT *context;
  /* Parse the options. */
  context = mu_opt_context_xnew(argc, argv, options, MU_OPT_PERMUTE);
  ret = mu_parse_opts(context);
  mu_opt_context_xfree(context);
  if (MU_OPT_ERR(ret))
    return 1; /* `mu_parse_opts' will print an error message for us */

  if (found_sel) {
    /* Print the selection. */
    fputs("You selected: ", stdout);
    switch (sel) {
    case FOO:
      puts("FOO");
      break;
    case BAR:
      puts("BAR");
      break;
    case BAZ:
      puts("BAZ");
      break;
    default:
      puts("an unknown value!"); /* This should not happen */
    }
  }
  else
    puts("You didn't select anything.");

  return 0;
}
```

Here is what the output of the example program looks like:

```
$ ./option-callback
⊣ You didn't select anything.
$ ./option-callback -s foo
⊣ You selected: FOO
$ ./option-callback --selection=bar
⊣ You selected: BAR
$ ./option-callback -s qux
error   ./option-callback: invalid selection: qux
$ ./option-callback -s
error   ./option-callback: '-s': option requires argument
```

2.6 Parsing Enumerated Arguments to Options

It is often useful to have an option which takes an argument which is a string that is restricted to a set of values. For example, GNU `ls` (and many other programs) take a `--color` option, which has an argument that can be '`always`', '`auto`', or '`never`' (in addition to various synonyms). Mu supports similar argument parsing, called *enumerated arguments*.

Names and values for enumerated types are specified in the `enum_values` field of the `MU_OPT` structure (see Section 2.1 [Option Structure], page 5). `enum_values` is an array of `MU_ENUM_VALUE` structures as defined below. `enum_values` is terminated by an element with a `name` field of `NULL`.

If the `enum_case_match` field of the `MU_OPT` structure is nonzero (see Section 2.1 [Option Structure], page 5), matching is case sensitive. Otherwise, matching is case insensitive.

Like long options and suboptions, abbreviation is allowed when passing enumerated arguments, as long as it is not ambiguous.

`MU_ENUM_VALUE` [Data Type]

Unlike `MU_OPT` (see Section 2.1 [Option Structure], page 5), this structure is simple and it's organization is guaranteed. Therefore, you may use positional initializers to initialize this structure. Of course, you can still use designated initializers if you prefer.

`const char *name`

This field specifies the name to match against when parsing the argument. Like long options, suboptions, and environment variables, `name` may have aliases separated by '`|`' (see Section 2.2 [Option Aliases], page 10). Alternatively, aliases can be specified by using separate entries with the same `value` (see below).

Duplicates in this field, either duplicate aliases or duplicates between entries, are not allowed. Note that if `enum_case_match` is zero, case is not considered. So, if `enum_case_match` is zero, you cannot have two entries, '`foo`' and '`FOO`', nor can you have two aliases specified as '`foo|FOO`'.

`int value`

This is the value of the enumeration. It is the value passed as the *arg* parameter of `callback_enum` (see Section 2.5 [Option Callbacks], page 18), and, if the `arg` field of the `MU_OPT` structure is not `NULL`, it is the value stored in `*arg`.

The following example illustrates how to use both case insensitive enumerated argument parsing, and case sensitive enumerated argument parsing:

```
#include <stdio.h>
#include <mu/options.h>
#include <mu/safe.h>            /* For mu_opt_context_x{new,free} */

enum selection { FOO, BAR, BAZ };

int main(int argc, char **argv) {
  enum selection sel;
  int found_sel;
  int ret;
  const MU_ENUM_VALUE enum_table[] = {
    /* Aliases can be specified for enumerated arguments. */
    { "foo|alias-foo", FOO },
    { "bar", BAR },
    /* Aliases can alternatively be specified like this. */
    { "alias-bar", BAR },
    { "baz", BAZ },
    /* This terminates the enumeration specification. */
    { 0 }
  };
  const MU_OPT options[] = {
    {
     .short_opt       = "s",
     .long_opt        = "selection",
     .has_arg         = MU_OPT_REQUIRED,
     .arg_type        = MU_OPT_ENUM,
     /* This indicates that matching should be case insensitive. */
     .enum_case_match = 0,
     .enum_values     = enum_table,
     .found_arg       = &found_sel,
     .arg             = &sel
    },
    {
     .short_opt       = "c",
     .long_opt        = "case-selection",
     .has_arg         = MU_OPT_REQUIRED,
     .arg_type        = MU_OPT_ENUM,
     /* This indicates that matching should be case sensitive. */
     .enum_case_match = 1,
     .enum_values     = enum_table,
     .found_arg       = &found_sel,
     .arg             = &sel
    },
    { 0 }
  };
  MU_OPT_CONTEXT *context;

  /* Parse the options. */
  context = mu_opt_context_xnew(argc, argv, options, MU_OPT_PERMUTE);
  ret = mu_parse_opts(context);
  mu_opt_context_xfree(context);
  if (MU_OPT_ERR(ret))
    return 1;

  if (found_sel) {
    /* Print the selection. */
    fputs("You selected: ", stdout);
    switch (sel) {
    case FOO:
      puts("FOO");
      break;
    case BAR:
      puts("BAR");
      break;
    case BAZ:
      puts("BAZ");
      break;
    default:
      /* This is guaranteed not to happen. */
      puts("an unknown value!");
    }
  }
  else
    puts("You didn't select anything.");

  return 0;
}
```

Here is the output of the above example, to show exactly how enumerated arguments are parsed:

```
$ ./option-enum --selection=foo
⊣ You selected: FOO
$ ./option-enum --selection=alias-foo
⊣ You selected: FOO
$ ./option-enum --selection=alias-bar
```

```
 ⊣ You selected: BAR
$ ./option-enum --selection=bAr
 ⊣ You selected: BAR
$ ./option-enum --selection=Ba
 error   ./option-enum: 'Ba': argument for '--selection' is ambiguous; possibilities:
 error     bar
 error     baz
$ ./option-enum --selection=qux
 error   ./option-enum: 'qux': invalid argument for '--selection'; must be one of 'foo', 'bar', 'alias-bar
$ ./option-enum --case-selection=FoO
 error   ./option-enum: 'FoO': invalid argument for '--case-selection'; must be one of 'foo', 'bar', 'alia
$ ./option-enum --case-selection=foo
 ⊣ You selected: FOO
```

2.7 Parsing Suboptions

Sometimes you may want to have an option which takes suboptions as arguments. You can do this through the `subopts` field of the `MU_OPT` structure. The `subopts` field is a list of `MU_OPT`s, terminated by a suboption with all fields equal to `0`. Suboptions are in every way like regular options, except that they may not have suboptions of their own and they must use the `subopt_name` field instead of `short_opt` or `long_opt`. See Section 2.1 [Option Structure], page 5.

Suboptions may also specify the `env_var` field for an equivalent environment variables as well. Environment variables may also take suboptions as a value. See Section 2.8 [Parsing the Environment], page 28.

Note: even though environment variables may be specified for suboptions, you may not have a suboption which *only* specifies an environment variable. I.e., you may not have a suboption which has no `subopt_name` field (such an option will be considered as a terminator for the suboption list). If you want to do this, use a regular option instead.

Options which take suboptions as arguments may use the `callback_subopt` field as a callback (see Section 2.5 [Option Callbacks], page 18). If a callback is used and the option is found on the command line (or in the environment for an environment variable), the callback for that option is guaranteed to be called *before* any callbacks for the suboptions themselves. Note, however, that if a suboption has an equivalent environment variable (using the `env_var` field), the callback for the option which takes that suboption as an argument will **not be called at all** (though the callback for the suboption will be called). Indeed, it is impossible to call the callback for the option which takes the suboption as an argument, because two different options with different callbacks may take the same suboptions as arguments. Nor would it make any sense, because the option never actually appeared on the command line (or environment).

Suboptions are specified as a comma-separated list, with '`=`' used to specify arguments. The commas must not contain spaces around them, and arguments cannot be specified any other way than with '`=`'.

Suboptions, like long options, may be abbreviated as long as they are not ambiguous. Note that this is in contrast to `getsubopt`, which does not allow abbreviation (see Section "Suboptions" in `libc`).

Like regular options, suboptions should use the `help` field, which will be used in the help message (see Section 2.11 [Formatting Help], page 37, and Section 2.1 [Option Structure], page 5). Suboptions may also use the arg_help field if they take arguments (see Section 2.4 [Option Arguments], page 14).

Normally, suboptions are parsed by mu_parse_opts from an argument to a regular option, using the subopts field. However, you may also parse suboptions in a user-specified string as well. Doing so is not too dissimilar from parsing regular options.

MU_SUBOPT_CONTEXT [Data Type]

This is an opaque context for parsing suboptions. It is allocated using mu_subopt_context_new and freed using mu_opt_context_free.

MU_SUBOPT_CONTEXT * mu_subopt_context_new (const char *prog_name, const char *suboptstr, const MU_OPT *subopts) [Function]

Allocate and return a new suboption parsing context. The name the program was invoked as should be passed in *prog_name* (normally argv[0]), and is used for error reporting.

The suboptions will be parsed in *suboptstr*. A copy of *suboptstr* will be made, so you need not worry about it going out of scope or being modified (this copy will be freed by mu_subopt_context_free). The suboptions are specified in *subopts*.

int mu_subopt_context_free (MU_SUBOPT_CONTEXT *context) [Function]

Free the suboption context, *context*. Callback data is freed as for mu_opt_context_free (see Chapter 2 [Parsing Options and Environment], page 3). Like mu_opt_context_free, mu_subopt_context_free will return nonzero if any of the destructors returned nonzero, or zero if all destructors returned zero. Also like mu_opt_context_free, all destructors are called even if one or more of them return nonzero.

int mu_parse_subopts (MU_SUBOPT_CONTEXT *context) [Function]

Parse the suboptions given in *context*. Use mu_subopt_context_new (see above) to create the context. Zero is returned on success, or an error code on error (see Section 2.12 [Option Parsing Errors], page 44).

Note that you may not call this function more than once. To do so is an error and will be diagnosed.

The following example illustrates the use of suboptions:

```
#include <stdio.h>
#include <mu/options.h>
#include <mu/safe.h>           /* For mu_opt_context_x* */

int subopt_none(void *data, char *err) {
  puts("suboption found: none");
  return 0;
}

int subopt_opt(int has_arg, const char *arg,
               void *data, char *err) {
  puts("suboption found: opt");
  if (has_arg)
    printf("argument: %s\n", arg);
  return 0;
}
```

```
int subopt_req(int has_arg, const char *arg,
               void *data, char *err) {
  printf("suboption found: req\nargument: %s\n", arg);
  return 0;
}

int main(int argc, char **argv) {
  int ret;
  /* These are the suboptions that can be passed to the `-o'
     option. They are specified just like regular options, except
     that `subopt_name' is used instead of `long_opt' or
     `short_opt', and they may not have suboptions of their
     own. */
  const MU_OPT suboptions[] = {
    {
     .subopt_name     = "none",
     .has_arg         = MU_OPT_NONE,
     .callback_none   = subopt_none,
     .help            = "a suboption taking no arguments"
    },
    {
     .subopt_name     = "opt",
     .has_arg         = MU_OPT_OPTIONAL,
     .arg_type        = MU_OPT_STRING,
     .callback_string = subopt_opt,
     .help            = "a suboption taking an optional argument"
    },
    {
     .subopt_name     = "req",
     .has_arg         = MU_OPT_REQUIRED,
     .arg_type        = MU_OPT_STRING,
     .callback_string = subopt_req,
     .help            = "a suboption taking a required argument"
    },
    { 0 }
  };
  const MU_OPT options[] = {
    {
     .short_opt = "o",
     .long_opt  = "options",
     .has_arg   = MU_OPT_REQUIRED,
     .arg_type  = MU_OPT_SUBOPT,
     .subopts   = suboptions,
     .help      = "a regular option which takes suboptions"
    },
    { 0 }
  };
  MU_OPT_CONTEXT *context;

  context = mu_opt_context_xnew(argc, argv, options, MU_OPT_PERMUTE);

  /* Add the help option. */
  mu_opt_context_add_help(context, NULL, NULL, "Parse suboptions.",
                          NULL, "1", NULL, NULL, NULL);
  mu_opt_context_xadd_help_options(context, MU_HELP_BOTH);

  /* Parse the options. */
  ret = mu_parse_opts(context);
  mu_opt_context_xfree(context);
  if (MU_OPT_ERR(ret))
    return 1;

  return 0;
}
```

And here is the output of the example program (note, the COLUMNS environment variable is set to 65 so that the help message will look good in this manual):

```
$ COLUMNS=65
$ export COLUMNS
$ ./subopts -o none,opt=foo
⊣ suboption found: none
⊣ suboption found: opt
⊣ argument: foo
$ ./subopts -o req
error  ./subopts: 'req': suboption requires argument
$ ./subopts --help
⊣ Usage: ./subopts [OPTION]...
⊣ Parse suboptions.
⊣
⊣ Mandatory arguments to long options are mandatory for short options too.
⊣   -o, --options=SUBOPTS   a regular option which takes suboptions
⊣   -h, --help[=plain|man]  print this help in plain text format if 'plain', or as a man(1) page if
⊣                             'man'; if the argument is omitted, it will default to 'plain'.
⊣
⊣ Suboptions for -o, --options:
⊣   none                    a suboption taking no arguments
⊣   opt[=STRING]            a suboption taking an optional argument
⊣   req=STRING              a suboption taking a required argument
```

2.8 Parsing the Environment

In addition to parsing options, mu_parse_opts supports parsing environment variables as well. Environment variables are specified using the env_var field (see Section 2.1 [Option Structure], page 5). Values of environment variables are specified in the same way as arguments are specified to options (see Section 2.4 [Option Arguments], page 14).

Unlike options, environment variables are parsed in the program environment (or the *environment* parameter to mu_opt_context_new_with_env), rather than in *argv* (see Chapter 2 [Parsing Options and Environment], page 3). And unlike long options and suboptions, environment variables may **not** be abbreviated. And whereas an invalid *option* will cause an error, an invalid *environment variable* will be ignored.

Environment variables may be specified for suboptions, and an environment variable may take suboptions as a value as well. For example, you might have an environment variable, ENV, which takes a suboption foo, which itself takes an optional string argument, say. And suppose foo has an equivalent environment variable, ENV_FOO. Then you might specify a value ‘bar’ to the foo suboption either by specifying a value to ENV like this: ENV=foo=bar, or by specifying a value directly to ENV_FOO like this: ENV_FOO=bar. The example shows how to do this as well. See Section 2.7 [Parsing Suboptions], page 25, for more information on suboptions.

Environment variables are always parsed before command line options. Environment variables and long/short options may be specified in the same option, but if this is the case, the command line option(s) will take precedence over the environment variable, since the environment variables will always be parsed first.

If an environment variable has aliases (see Section 2.2 [Option Aliases], page 10), aliases specified first will take precedence. For example, if an environment variable is specified as 'FOO|BAR', and both FOO and BAR are in the environment, then the value of FOO will take precedence because it was specified as an alias before BAR. Note also that if both FOO and BAR are specified in the environment, the value of BAR will be completely ignored. The callback (if any) will only be called once, for FOO (see Section 2.5 [Option Callbacks], page 18).

Another thing to note is that if you have an environment variable with a has_arg value of MU_OPT_NONE, then if that environment variable is encountered, and it has a value other than the empty string, that will cause an error. This is not very user-friendly behavior, and you might consider using a has_arg of MU_OPT_OPTIONAL and an arg_type of MU_OPT_BOOL. Then, if the environment variable has no value, you can default to true. This is more user-friendly, because things like ENV_VAR=yes or ENV_VAR=no will do what is expected (assuming your environment variable is called ENV_VAR).

Traditionally, environment variable names are in ALL CAPS.

Here is an example of how environment variables can be parsed:

```
#include <stdio.h>
#include <mu/options.h>
#include <mu/safe.h>            /* For mu_opt_context_x* */

/* Print a message when an option is found. */
int print_opt(int has_arg, const char *arg,
              void *data, char *err) {
  const char *name = data;
  printf("Found an option/environment variable '%s'", name);
  if (has_arg)
    printf(" with an argument '%s'", arg);
  putchar('\n');
  return 0;
}

int main(int argc, char **argv) {
  int ret;
  const MU_OPT suboptions[] = {
    {
     .subopt_name     = "subopt",
     /* Suboptions can have environment variables as well. */
     .env_var         = "ENV_SUBOPT",
     .has_arg         = MU_OPT_OPTIONAL,
     .arg_type        = MU_OPT_STRING,
     .callback_string = print_opt,
     .cb_data         = "a suboption",
     .help            =
     "a suboption with an equivalent environment variable"
    },
    { 0 }
  };
  const MU_OPT options[] = {
    {
     .short_opt       = "a",
     .long_opt        = "an-option",
     /* AN_ENV_VAR will always take precedence over ALIAS since it is
        specified first below. */
     .env_var         = "AN_ENV_VAR|ALIAS",
     .has_arg         = MU_OPT_OPTIONAL,
     .arg_type        = MU_OPT_STRING,
     .callback_string = print_opt,
     .cb_data         = "an option",
     .help            =
     "an option with an equivalent environment variable"
    },
    {
     .short_opt       = "b",
     .long_opt        = "another-option",
     .has_arg         = MU_OPT_OPTIONAL,
     .arg_type        = MU_OPT_STRING,
     .callback_string = print_opt,
     .cb_data         = "another option",
     .help            =
     "an option without an equivalent environment variable"
    },
    {
     .env_var         = "ANOTHER_ENV_VAR",
     .has_arg         = MU_OPT_REQUIRED,
     /* Environment variables can have suboptions as well. */
     .arg_type        = MU_OPT_SUBOPT,
     .subopts         = suboptions,
     .help            =
     "an environment variable (which takes "
     "suboptions) without an equivalent option"
    },
    { 0 }
  };
  MU_OPT_CONTEXT *context;

  context = mu_opt_context_xnew(argc, argv, options, MU_OPT_PERMUTE);

  /* Add the help option. */
  mu_opt_context_add_help(context, NULL, NULL,
                          "Parse options and environment variables.",
                          NULL, "1", NULL, NULL, NULL);
  mu_opt_context_xadd_help_options(context, MU_HELP_BOTH);

  /* Parse the options. */
  ret = mu_parse_opts(context);
  mu_opt_context_xfree(context);
  if (MU_OPT_ERR(ret))
    return 1;

  return 0;
}
```

And here is the output of the above program (note, the `COLUMNS` environment variable is set to 65 so that the help message will look good in this manual):

```
$ COLUMNS=65
$ export COLUMNS
$ AN_ENV_VAR=foo ./environ --an-option=bar --another-option=baz
⊣ Found an option/environment variable 'an option' with an argument 'foo'
⊣ Found an option/environment variable 'an option' with an argument 'bar'
⊣ Found an option/environment variable 'another option' with an argument 'baz'
$ ANOTHER_ENV_VAR=subopt=foo ./environ
⊣ Found an option/environment variable 'a suboption' with an argument 'foo'
$ ENV_SUBOPT=foo ./environ
⊣ Found an option/environment variable 'a suboption' with an argument 'foo'
# AN_ENV_VAR will always take precedence over ALIAS. Also note that
# the callback is only called once, even though both aliases are
# specified.
$ AN_ENV_VAR=foo ALIAS=bar ./environ
⊣ Found an option/environment variable 'an option' with an argument 'foo'
$ ALIAS=bar AN_ENV_VAR=foo ./environ
⊣ Found an option/environment variable 'an option' with an argument 'foo'
$ ./environ --help
⊣ Usage: ./environ [OPTION]...
⊣ Parse options and environment variables.
⊣
⊣   -a, --an-option[=STRING]       an option with an equivalent environment variable
⊣   -b, --another-option[=STRING]  an option without an equivalent environment variable
⊣   -h, --help[=plain|man]         print this help in plain text format if 'plain', or as a man(1)
⊣                                    page if 'man'; if the argument is omitted, it will default to
⊣                                    'plain'.
⊣
⊣ Suboptions for ANOTHER_ENV_VAR:
⊣   subopt[=STRING]                a suboption with an equivalent environment variable
⊣
⊣ ENVIRONMENT
⊣
⊣   AN_ENV_VAR, ALIAS[=STRING]     an option with an equivalent environment variable
⊣   ANOTHER_ENV_VAR=SUBOPTS        an environment variable (which takes suboptions) without an
⊣                                    equivalent option
⊣   ENV_SUBOPT[=STRING]            a suboption with an equivalent environment variable
```

2.9 Option Parsing Flags

There are several flags which affect option parsing in different ways. These flags are passed in the *flags* parameter to `mu_opt_context_new` (see Chapter 2 [Parsing Options and Environment], page 3).

`MU_OPT_PERMUTE` [Constant]

This flag indicates that `mu_parse_opts` should rearrange *argv* so that the options are at the beginning, and positional arguments are at the end.[5] If this flag is *not* given, option parsing will stop as soon as the first non-option argument is encountered. Option parsing will also stop when the string '`--`' is encountered, whether or not this flag was given. The '`--`' string will be treated as an option, i.e., it will be counted in the return value of `mu_parse_opts` and, if this flag is set, it will be moved before all the other positional arguments in *argv*.

If the environment variable `POSIXLY_CORRECT` is set, or the `MU_OPT_STOP_AT_ARG` flag is used, `mu_parse_opts` will act as though this flag were not given even if it was. Note that `POSIXLY_CORRECT` is searched for in the *env* parameter given to `mu_opt_context_new_with_env`, or in the program environment if no *env* parameter is given

[5] Positional arguments are not rearranged internally, however. I.e., the positional arguments are guaranteed to be in the same order as they originally appeared in, even if *argv* was rearranged.

or if `mu_opt_context_new` was used to create the option parsing context. If you want to ignore `POSIXLY_CORRECT` entirely, use the `MU_OPT_IGNORE_POSIX` flag.

`MU_OPT_BUNDLE` [Constant]

This flag enables bundling of short options. Without this flag, long options may be specified with a single '`-`' or '`--`'. When this flag is set, long options may only be specified with '`--`'.

So when this flag is set, `-abc` will be treated as three short options, `a`, `b`, and `c` (assuming that `a` and `b` don't take arguments[6]), whereas without this flag, `-abc` will be treated as a single long option, `abc`.

Note that when this flag is *not* set, short options with optional arguments take precedence over long options. So, if there is a short option, `o`, which takes an optional argument, and another long option, `option` (it doesn't matter whether it takes an argument or not), then the string `-option` is a short option, `o` with an argument '`ption`'.

While this may seem counter-intuitive at first, the reason for this seemingly strange behavior becomes apparent when you consider a short option, `o`, which takes an optional argument and a long option, `option`. (Again, it doesn't matter whether the long option takes an argument or not.) Suppose that instead, long options took precedence over short ones.[7] Now lets look at the `-option` example again. It would be parsed as a single long option, `option`. But what if you wanted to pass the `o` short option an argument '`ption`'? Or indeed, even just '`p`'? It would be parsed as a long option, `option`. So there is **no possible way** to pass an argument to the `o` short option such that '`ption`' begins with that argument (or *is* the argument). But, you ask, couldn't you write `-o ption`? You could, if the `o` short option takes a required argument, but not if it takes an optional argument, because optional arguments to short options are required to be specified as part of the option itself (see Section 2.4 [Option Arguments], page 14). Note that you can still write the `option` long option as `--option`, which is unambiguous, so there is no issue.

Since this behavior can be confusing and counter-intuitive, long options take precedence when the short option that would match takes a required argument, and the long option matches exactly. Going back to the above example, if the `o` short option instead took a required argument, '`-option`' would be the long option `option`, rather than the short option `o`, with an argument '`ption`'. Note, however, that '`-opt`' would be the `o` short option with an argument '`pt`'. Also note that if the `option` long option did not exist, '`-option`' *would* be the short option `o` with an argument '`ption`'. If you want to avoid ambiguity, you should always pass required arguments to short options in the next argument, and precede long options with two dashes like so: '`-o` *`arg`* `--option`'.

[6] See Section 2.4 [Option Arguments], page 14, for an explanation of why options `a` and `b` cannot take arguments.

[7] And indeed, this is the way GNU's `getopt_long_only` function works. See Section "Getopt Long Options" in `libc`, near the bottom.

`MU_OPT_CONTINUE` [Constant]

This flag should be used if you are going to call `mu_parse_opts` more than once. See Section 2.10 [Ordered Option Parsing], page 33, for more information on how to use this flag correctly.

If the `POSIXLY_CORRECT` environment variable is set, or the `MU_OPT_STOP_AT_ARG` flag is passed, all arguments after the first non-option arguments will be treated as non-option arguments as well. Note that `POSIXLY_CORRECT` is searched for in the *env* parameter given to `mu_opt_context_new_with_env`, or in the program environment if no *env* parameter is given or if `mu_opt_context_new` was used to create the option parsing context. If you want to ignore `POSIXLY_CORRECT` entirely, use the `MU_OPT_IGNORE_POSIX` flag.

`MU_OPT_ALLOW_INVALID` [Constant]

This flag makes `mu_parse_opts` treat invalid options as positional arguments. It can be useful if you are writing a function which parses some options, but then leaves the rest for the caller to parse. An example of a function which does this (although it does not use Mu) is `gtk_init` (see section *Main loop and Events* in *GTK+ 3 Reference Manual*).

Note: if you use `mu_opt_context_add_help_options`, the help option will only print the help for the context you called `mu_opt_context_add_help_options` with. `mu_opt_context_add_help_options` has no way of knowing what options will be parsed in the future. So if you are writing a function like that described above, you may wish instead to make your function take an option context as a parameter, and then add some standard ones using `mu_opt_context_add_options` (see Chapter 2 [Parsing Options and Environment], page 3).

`MU_OPT_IGNORE_POSIX` [Constant]

Ignore the `POSIXLY_CORRECT` environment variable even if it is set. This flag can be used for programs for which it would not make sense to parse options in a POSIXly correct way. For example, you might have an option which acts on the last positional argument given before it.

`MU_OPT_STOP_AT_ARG` [Constant]

Stop parsing options after the first positional argument. I.e., act as though the `POSIXLY_CORRECT` environment variable were set. If this flag is used, `MU_OPT_IGNORE_POSIX` has no effect.

Note that this behavior is the default, unless the `MU_OPT_PERMUTE` flag is used, the `MU_OPT_CONTINUE` flag is used, and/or an argument callback is used (see Section 2.10 [Ordered Option Parsing], page 33).

2.10 Ordered Option Parsing

Sometimes it is useful to know where options appear on the command line. You can tell in which order *options* (and suboptions) appear by taking advantage of the fact that callbacks (see Section 2.5 [Option Callbacks], page 18) are called in the same order that the corresponding options appear on the command line. However, if you want to determine the ordering of non-option *positional arguments* as well as options, you must instead use an *argument callback*, or use the MU_OPT_CONTINUE flag (see Section 2.9 [Option Parsing Flags], page 31).

To use an argument callback, you must use the mu_opt_context_set_arg_callback function.

void mu_opt_context_set_arg_callback (MU_OPT_CONTEXT **context*, int *callback* (const char **arg*, void **data*, char **err*), void **data*, int *destructor* (void **data*)) [Function]

This function sets an argument callback in *context*. *context* must not have been created with the MU_OPT_CONTINUE flag (see Section 2.9 [Option Parsing Flags], page 31), and it must never have been passed to mu_parse_opts (see Chapter 2 [Parsing Options and Environment], page 3).

callback will be called for each positional argument found when mu_parse_opts is called. *callback* may not be NULL. *data* will be passed to *callback* as the *data* argument. When *context* is destroyed using mu_opt_context_free, *destructor* will be called with *data* passed as its *data* argument.

callback should indicate success by returning zero. If *callback* fails, it should return nonzero and copy an error string to *err* (not exceeding MU_OPT_ERR_MAX). See [callback error indication], page 20, for more information.

Note that if either of the flags MU_OPT_PERMUTE or MU_OPT_STOP_AT_ARG are used when the option parsing context is created (see Section 2.9 [Option Parsing Flags], page 31), then the successful return value of mu_parse_opts will *not* include the positional arguments parsed (see Chapter 2 [Parsing Options and Environment], page 3). This is so that, after shifting the arguments by the return value of mu_parse_opts with mu_shift_args, the remaining arguments will be the positional arguments.

Normally, however, when using an argument callback, you shouldn't need the return value of mu_parse_opts except to check for errors.

If neither of the flags MU_OPT_PERMUTE nor MU_OPT_STOP_AT_ARG are given, then the return value of mu_parse_opts *will* include the positional arguments (i.e., a successful return from mu_parse_opts will always return the total number of arguments, options or otherwise). This is because, if neither MU_OPT_PERMUTE nor MU_OPT_STOP_AT_ARG are given, it cannot be guaranteed that all positional arguments will appear after all options. Thus, the return value of mu_parse_opts should not be used to shift the arguments, and should only be used to check for errors.

Here is an example of how to use argument callbacks:

```
#include <stdio.h>
#include <mu/options.h>
#include <mu/safe.h>            /* For mu_opt_context_x{new,free} */

/* Callbacks to print a message when we find an option or argument. */

static int print_example(void *data, char *err) {
  puts("Option found: example");
  return 0;
}

static int print_another(void *data, char *err) {
  puts("Option found: another");
  return 0;
}

static int print_argument(const char *arg, void *data, char *err) {
  printf("Argument found: %s\n", arg);
  return 0;
}

int main(int argc, char **argv) {
  const MU_OPT options[] = {
    {
     .short_opt     = "e",
     .long_opt      = "example",
     .has_arg       = MU_OPT_NONE,
     .callback_none = print_example
    },
    {
     .short_opt     = "a",
     .long_opt      = "another",
     .has_arg       = MU_OPT_NONE,
     .callback_none = print_another
    },
    { 0 }
  };
  MU_OPT_CONTEXT *context;
  int ret;

  context = mu_opt_context_xnew(argc, argv, options, 0);
  mu_opt_context_set_arg_callback(context, print_argument,
                                  NULL, NULL);
  ret = mu_parse_opts(context);
  mu_opt_context_xfree(context);
  return MU_OPT_ERR(ret);
}
```

Here is the output of the example program:

```
$ ./option-ordered-callback foo -e bar --another baz
⊣ Argument found: foo
⊣ Option found: example
⊣ Argument found: bar
⊣ Option found: another
⊣ Argument found: baz
$ ./option-ordered-callback -a foo bar --example
⊣ Option found: another
⊣ Argument found: foo
⊣ Argument found: bar
⊣ Option found: example
```

Alternatively, you can also determine the order in which options and positional arguments appear using the MU_OPT_CONTINUE flag. If you use this flag, you should not use the MU_OPT_PERMUTE flag (otherwise, all options will be parsed at once and the MU_OPT_CONTINUE flag is rendered useless). The MU_OPT_STOP_AT_ARG is also useless if you use MU_OPT_CONTINUE, because if you use MU_OPT_STOP_AT_ARG, you might as well just parse the options once and then parse the rest of the arguments, which will only be positional arguments.

Using the MU_OPT_CONTINUE flag, you should parse options (maybe using callbacks if you care about the order of the options themselves), then parse positional arguments, and then options again until all arguments are used up. Note, however, that normally you must not call `mu_parse_opts` more than once, unless you pass the `MU_OPT_CONTINUE` flag

All the environment variables will be parsed on the first call of `mu_parse_opts`. They will not be parsed again in subsequent calls. See Section 2.8 [Parsing the Environment], page 28, for more information.

After you parse each non-option argument, you must call `mu_opt_context_shift` on the option context in order to ensure that `mu_parse_opts` will not stop at the argument you just parsed.

`int mu_opt_context_shift (MU_OPT_CONTEXT *`*`context`*`, int` *`amount`*`)` [Function]

> Update the internal index of *context* by *amount*. *amount* may be negative.
>
> Normally, this function returns zero. However, in the case that the new index would be less that 1, the new index will instead be set to 1 and the amount that could not be shifted will be returned. And in the case that the new index would be greater than or equal to the number of arguments in *context*, the new index will instead be set to the number of arguments minus one, and again, the amount that could not be shifted is returned.

Here is an example illustrating how to parse options and positional arguments while preserving the order, without using argument callbacks:

```
#include <stdio.h>
#include <mu/options.h>
#include <mu/safe.h>            /* For mu_opt_context_x{new,free} */

/* Callbacks to print a message when we find an option. */

static int print_example(void *data, char *err) {
  puts("Option found: example");
  return 0;
}

static int print_another(void *data, char *err) {
  puts("Option found: another");
  return 0;
}

int main(int argc, char **argv) {
  const MU_OPT options[] = {
    {
     .short_opt     = "e",
     .long_opt      = "example",
     .has_arg       = MU_OPT_NONE,
     .callback_none = print_example
    },
    {
     .short_opt     = "a",
     .long_opt      = "another",
     .has_arg       = MU_OPT_NONE,
     .callback_none = print_another
    },
    { 0 }
  };
  MU_OPT_CONTEXT *context;

  context = mu_opt_context_xnew(argc, argv, options, MU_OPT_CONTINUE);
  while (argc > 1) {
    int ret;

    /* Parse options. */
    ret = mu_parse_opts(context);
    if (MU_OPT_ERR(ret))
      return 1;

    /* Shift the arguments (to get rid of the options we just
       parsed). */
    mu_shift_args(&argc, &argv, ret);

    if (argc > 1) {
      /* Print an argument (we don't have to print them all at once
         because if `mu_parse_opts' doesn't find any options, it will
         just return 0). */
      printf("Argument found: %s\n", argv[1]);
      /* Shift away this argument. */
      mu_shift_args(&argc, &argv, 1);
      mu_opt_context_shift(context, 1);
    }
  }
  mu_opt_context_xfree(context);

  return 0;
}
```

The behavior of the above program is identical to the one using argument callbacks (see [argument callback example], page 34).

2.11 Formatting Help

Many programs have a `-h` or `--help` option which prints out a short message describing how to use the program. Mu supports automatically generating a usage message through the use of the `help` and `arg_help` fields of the `MU_OPT` structure (see Section 2.1 [Option Structure], page 5).

`void mu_opt_context_add_help (MU_OPT_CONTEXT *context, const char *usage, const char *short_description, const char *description, const char *notes, const char *section, const char *section_name, const char *source, const char *date)` [Function]

Add usage information to *context*. The arguments are as follows:

usage This is a short, human-readable description of the arguments your program takes. For example, '`[OPTION]... [FILE]...`'. If *usage* is left `NULL`, it will default to '`[OPTION]...`' if there is a least one option in *options* or, if *options* is empty, it will default to '' (the empty string). If you really want nothing to be printed for *usage*, pass '' (the empty string) explicitly.

Alternative usages (including no arguments) can be specified, separated by newlines. For example, if your program is called `prog` and you pass a *usage* of ‘FOO\nBAR\n\nBAZ’ (note the two ‘\n’s after ‘BAR’), the help output will be as follows:

```
Usage: prog FOO
  or:  prog BAR
  or:  prog
  or:  prog BAZ
[...]
```

You can think of it as piping *usage* to

```
sed '1s/^/Usage: prog /; 2,/^/  or:  prog /'
```

(assuming your program is called prog).

short_description

This is a very short description (shorter than *description*) of the program. It is used as the description in the NAME section.

If this parameter is passed as NULL, a default will be substituted. This parameter is only used for man page output.

description

This is a short, human-readable description of whatever your program does. It is printed right below the *usage* line. It should be one or two sentences long. For example:

> Frobnicate frobs. Nonexistent frobs will be treated as empty.

You may reference metasyntactic variables specified in *usage* here (e.g., ‘FILE’) if you like. If you set this to NULL, no description will be printed.

Note: you should *not* write

> Mandatory arguments to long options are mandatory for short options too.

in *description*. This will automatically be added to the help text if it makes sense (i.e., if there exist long options with required arguments that have short option equivalents).

notes This will be printed at the end of the help message, after the options. This is where you can put examples, bug report addresses, etc.

section The section number of the manual page. This can be any string (but see *section_name* below), although it should be a number followed by an optional suffix. The optional suffix can be something like ncurses uses for its man pages, ‘NCURSES’ (so the full section would be ‘3NCURSES’). Most likely you should just set this to a number.

This parameter must not be NULL, unless man page output is not being used. This parameter is only used for man page output.

section_name

This is the name of the manual section. For example, ‘User Commands’.

If this parameter is passed as NULL, a default will be chosen based on *section*. In this case, *section* must start with a number between 1 and 9 inclusive except for 7. Section 7 does not have a default name because it is more of a “miscellaneous” section, and thus you must provide the name yourself.

This parameter is only used for man page output.

source This is the "source" of your program. If your program is part of a suite, put the name and version of the suite here. Otherwise, put the name and version of your program here.

If this parameter is passed as `NULL` (not recommended), the name your program was invoked as will be used. This parameter is only used for man page output.

date This is the date that the help text was last updated. Update *date* every time you change the help text for any option, or you change the name of an option or add a new one. You need not update this for trivial changes.

The format for *date* is conventionally '`YYYY-MM-DD`'.

If this parameter is passed as `NULL` (not recommended), the date at which your program was run to generate the man page will be used instead. Note that the *date* parameter is passed as `NULL` in the examples for simplicity, but this is still not recommended.

This parameter is only used for man page output.

`int mu_opt_context_add_help_options (MU_OPT_CONTEXT *context, int flags)` [Function]

Add help options to *context* based on *flags*. If `MU_HELP_PREPEND` is present in *flags*, the help options will be prepended to the current options, i.e., inserted before them. Othewise, if `MU_HELP_PREPEND` is not present in *flags*, the help options will be appended to the current options, i.e., inserted after them.

flags tells `mu_opt_context_add_help_options` what kind of help options/environment variables should be created, in addition to whether it should append or prepend the help options. The following values may be passed in *flags*, and can be combined with `|` (bitwise OR).

`MU_HELP_PREPEND`
This indicates that `mu_opt_context_add_help_options` should add the help options before the current options, rather than after. Note that order of the help options themselves is unchanged.

`MU_HELP_SHORT`
`MU_HELP_LONG`
`MU_HELP_QUESTION_MARK`
These flags tell `mu_opt_context_add_help_options` to create an option which takes a single optional argument, *format*. `MU_HELP_SHORT` will create a short option, `-h`, while `MU_HELP_LONG` will create a long option, `--help`, and `MU_HELP_QUESTION_MARK` will create a short option, `-?`. *format* specifies the output format to use when outputting help. It can either be '`man`' to output in a format which can be parsed by the `man` program, or it can be '`plain`' or '`text`' to output in a human-readable, plain-text format.

If *format* is omitted, it will default to the value of the `MU_HELP_FORMAT` environment variable if `MU_HELP_ENV` is passed in *flags*. Otherwise, if the

MU_HELP_FORMAT environment variable is not set or does not have a value or MU_HELP_ENV was not passed in *flags*, *format* will default to '`plain`'.

Note: MU_HELP_QUESTION_MARK is not included in MU_HELP_ALL (see below). If you want to pass all flags including MU_HELP_QUESTION_MARK, you must pass it explicitly, like so: MU_HELP_ALL | MU_HELP_QUESTION_MARK.

MU_HELP_MAN_SHORT
MU_HELP_MAN_LONG

These flags tell `mu_opt_context_add_help_options` to create an option which takes no argument, and always outputs help in `man` format. MU_HELP_MAN_SHORT will create a short option, `-m`, while MU_HELP_MAN_LONG will create a long option, `--man`.

MU_HELP_ENV

This flag tells `mu_opt_context_add_help_options` to create an environment variable, MU_HELP_FORMAT, which will specify an output format to use if none was specified to `-h` or `--help`. You must only use this flag if MU_HELP_SHORT or MU_HELP_LONG was also passed in *flags*.

MU_HELP_BOTH

Equivalent to MU_HELP_SHORT | MU_HELP_LONG.

MU_HELP_MAN_BOTH

Equivalent to MU_HELP_MAN_SHORT | MU_HELP_MAN_LONG.

MU_HELP_ALL

Equivalent to passing all flags except for MU_HELP_QUESTION_MARK, i.e., MU_HELP_BOTH | MU_HELP_MAN_BOTH | MU_HELP_ENV.

Output formatted for the `man` program will be piped to `man` if standard output is a terminal (as determined by `isatty`), otherwise the raw `roff` code will be output to standard output. If standard output is a terminal but an error occurred while executing the `man` program, a warning message will be printed and the raw `roff` code will be output to standard output as if standard output was not a terminal.

Note: some systems may not provide the necessary functionality to run the `man` command. In that case, `roff` code will always be output, regardless of whether standard output is a terminal.

`int mu_format_help (FILE *`*`stream`*`, const MU_OPT_CONTEXT *`*`context`*`)` [Function]

`int mu_format_help_man (FILE *`*`stream`*`, const MU_OPT_CONTEXT *`*`context`*`)` [Function]

These functions format a help message, printing it to *stream*. If you'd like to automatically create a help option that does this, see `mu_opt_context_add_help_options` above. You might also want to call these functions manually, for example, if your program receives no arguments or if `mu_parse_opts` returns an error code (see Chapter 2 [Parsing Options and Environment], page 3, and Section 2.12 [Option Parsing Errors], page 44).

`mu_format_help` will output a human-readable, plain text message, while `mu_format_help_man` will output `roff` code.

The strings passed to mu_opt_context_add_help are used in the help message, as well as the options in *context*.

char * mu_format_help_string (const MU_OPT_CONTEXT **context*, unsigned short *goal*, unsigned short *width*) [Function]

char * mu_format_help_man_string (const MU_OPT_CONTEXT **context*) [Function]

These functions are like mu_format_help and mu_format_help_man respectively (see above), except that they return the output as a string rather than printing it. If an error occurs, NULL will be returned and errno will be set to indicate the error. If the returned string is not NULL, it will by dynamically allocated and must be freed when you are done with it (see Section "Freeing after Malloc" in libc).

Unlike mu_format_help, mu_format_help_string is unable to determine the *goal* and *width* to use, so you must specify these parameters yourself. See Chapter 3 [Formatting Text], page 45, for the meanings of *goal* and *width*.

You should provide help text for individual options in the help and arg_help fields of the MU_OPT structure (see Section 2.1 [Option Structure], page 5).

arg_help is similar to the *usage* parameter to mu_format_help. It should be a simple string describing the kind of arguments the option takes. For example, you might write 'FILE' if your option takes a file argument, or 'WxH' if it takes a width and height argument, separated by an 'x'. If this is left as NULL, a default will be chosen based on the type of argument your option takes, specified in the arg_type field of the MU_OPT structure.

Note: you should *not* use '[' and ']' in the arg_help string. The arg_help string will automatically be enclosed in '[' and ']' if the option takes an optional argument.

help is a short description of what the option does. Most GNU utilities use a single sentence, begun with a lowercase letter[8] and ended without a period. However, you can format help however you like, but keep in mind that it should be fairly short. One sentence or, if you really must, two.

If the help field is left as NULL, the corresponding option will remain undocumented as if it did not exist. See Section 2.1 [Option Structure], page 5, for more information.

Below is an example illustrating the usage of both mu_opt_context_add_help_options and mu_format_help. Note that the category field is used to denote option categories (see Section 2.1 [Option Structure], page 5).

```
#include <stdio.h>
#include <stdlib.h>
#include <mu/options.h>
#include <mu/compat.h>          /* For __attribute__() */
#include <mu/safe.h>            /* For mu_opt_context_x* */

__attribute__((noreturn))
int print_version(void *data, char *err) {
  puts("Version 1.0");
  exit(0);
}

int main(int argc, char **argv) {
```

[8] Unless it should be uppercase for another reason, for example a proper noun or acronym.

```
int ret;
const MU_OPT opts_start[] = {
  {
   .short_opt = "n",
   .long_opt  = "none",
   .has_arg   = MU_OPT_NONE,
   .help      = "an option which takes no argument"
  },
  { .category = "Options taking arguments" },
  {
   .short_opt = "o",
   .long_opt  = "optional",
   .has_arg   = MU_OPT_OPTIONAL,
   .arg_type  = MU_OPT_STRING,
   .arg_help  = "OPTARG",
   .help      = "an option which optionally takes an argument"
  },
  {
   .short_opt = "r",
   .long_opt  = "required",
   .has_arg   = MU_OPT_REQUIRED,
   .arg_type  = MU_OPT_STRING,
   .arg_help  = "REQARG",
   .help      = "an option which requires an argument"
  },
  { .category = "Help options and environment variables" },
  { 0 }
};
/* Options to add after the help options. */
const MU_OPT opts_end[] = {
  { .category = "Version information" },
  {
   .short_opt = "v",
   .long_opt  = "version",
   .has_arg   = MU_OPT_NONE,
   .callback_none = print_version,
   .help      = "print version information and exit"
  },
  { 0 }
};
MU_OPT_CONTEXT *context;

context = mu_opt_context_xnew(argc, argv, opts_start,
                              MU_OPT_BUNDLE | MU_OPT_PERMUTE);

/* Add the help data. */
mu_opt_context_add_help(context, "[OPTION]...", "do stuff",
                        "Do stuff. If this text is really long, it "
                        "will be wrapped. Some more text to make "
                        "this text long enough to be wrapped.",
                        "Report bugs to <libmu-bug@nongnu.org>.",
                        "1", NULL, "Mu Examples", NULL);
/* Create the help option. MU_HELP_ALL is equivalent to
   MU_HELP_SHORT | MU_HELP_LONG | MU_HELP_MAN_SHORT |
   MU_HELP_MAN_LONG | MU_HELP_ENV, so it will create the options
   '-h', '--help', '-m', and '--man', and it will create the
   environment variable 'MU_HELP_FORMAT'. */
mu_opt_context_xadd_help_options(context, MU_HELP_ALL);
```

```
  /* Add the other options. */
  mu_opt_context_xadd_options(context, opts_end, MU_OPT_APPEND);

  /* Parse the options. */
  ret = mu_parse_opts(context);

  /* If there was an option parsing error, print a usage message so
     the user knows how to use us properly. */
  if (ret == MU_OPT_ERR_PARSE)
    mu_format_help(stderr, context);

  mu_opt_context_xfree(context);

  return !!MU_OPT_ERR(ret);
}
```

This is what the output looks like (note, the `COLUMNS` environment variable is set to 65 so that the output will look good in this manual):

```
$ COLUMNS=65
$ export COLUMNS
$ ./option-help --help
⊣ Usage: ./option-help [OPTION]...
⊣ Do stuff. If this text is really long, it will be wrapped. Some more text to make this text long
⊣ enough to be wrapped.
⊣
⊣ Mandatory arguments to long options are mandatory for short options too.
⊣   -n, --none                  an option which takes no argument
⊣
⊣ Options taking arguments:
⊣   -o, --optional[=OPTARG]     an option which optionally takes an argument
⊣   -r, --required=REQARG       an option which requires an argument
⊣
⊣ Help options and environment variables:
⊣   -h, --help[=plain|man]      print this help in plain text format if 'plain', or as a man(1) page
⊣                                 if 'man'; if the argument is omitted, it will default to the value
⊣                                 of the MU_HELP_FORMAT environment variable if set, otherwise
⊣                                 'plain'.
⊣   -m, --man                   print this help as a man(1) page
⊣
⊣ Version information:
⊣   -v, --version               print version information and exit
⊣
⊣ ENVIRONMENT
⊣
⊣ Help options and environment variables:
⊣   MU_HELP_FORMAT[=plain|man]  the default format for -h, --help
⊣
⊣ Report bugs to <libmu-bug@nongnu.org>.
$ ./option-help --foo
 error   ./option-help: '--foo': invalid option
 error   Usage: ./option-help [OPTION]...
 error   Do stuff. If this text is really long, it will be wrapped. Some more text to make this text long
 error   enough to be wrapped.
 error
 error   Mandatory arguments to long options are mandatory for short options too.
 error     -n, --none                  an option which takes no argument
 error
 error   Options taking arguments:
```

```
error   -o, --optional[=OPTARG]     an option which optionally takes an argument
error   -r, --required=REQARG       an option which requires an argument
error
error Help options and environment variables:
error   -h, --help[=plain|man]      print this help in plain text format if 'plain', or as a man(1) pag
error                                 if 'man'; if the argument is omitted, it will default to the valu
error                                 of the MU_HELP_FORMAT environment variable if set, otherwise
error                                 'plain'.
error   -m, --man                   print this help as a man(1) page
error
error Version information:
error   -v, --version               print version information and exit
error
error ENVIRONMENT
error
error Help options and environment variables:
error   MU_HELP_FORMAT[=plain|man]  the default format for -h, --help
error
error Report bugs to <libmu-bug@nongnu.org>.
```

2.12 Option Parsing Errors

`mu_parse_opts` and `mu_parse_subopts` can fail for several reasons. On failure, these functions will return an error code depending on the reason for failure. The error code can be one of the following:

`MU_OPT_ERR_PARSE` [Constant]

An option parsing error. This indicates that the user made an error when specifying options on the command line. You may wish to print a help message when `mu_parse_opts` returns this value (see Section 2.11 [Formatting Help], page 37, for an example).

`MU_OPT_ERR_IO` [Constant]

This indicates that an input/output error occurred while parsing the arguments. This could indicate, for example, failure to open a file specified as an argument to an option which has a `arg_type` field of `MU_OPT_FILE` (see Section 2.1 [Option Structure], page 5).

`MU_OPT_ERR_CALLBACK` [Constant]

This value is returned from `mu_parse_opts` when a callback returns a nonzero value (see Section 2.5 [Option Callbacks], page 18). You can have your callback set an error flag if you want more details.

`int MU_OPT_ERR (int` *`retval`*`)` [Macro]

This macro returns true if *retval* is one of the above error codes.

3 Formatting Text

Mu provides several functions for formatting text. The symbols described below are declared in `mu/format.h`.

`unsigned short mu_format_tab_stop` [Variable]

The formatting functions always convert `TAB` characters ('`\t`') to spaces. This global variable specifies the tab stop to be used by the formatting functions. You may set it directly. The default value is `MU_FORMAT_TAB_STOP` (see below).

`MU_FORMAT_TAB_STOP` [Constant]

The default value for `mu_format_tab_stop` (see above). Equal to `8`.

`int mu_format (FILE *stream, unsigned short *cursor, unsigned short goal, unsigned short width, unsigned short indent, unsigned short subindent, const char *format, ...)` [Function]

First, this function creates an internal string based on the `printf`-style format string, *format*, and a variable number of extra arguments which are processed according to '`%`'-directives in *format*. See Section "Formatted Output" in `libc` for more information on how *format* and the variable arguments are processed.

After this internal string is created, it is then printed to *stream*, with formatting being done according to the various parameters. For a description of what these parameters do, see Section 3.1 [Controlling Formatted Output], page 46.

This function returns `0` on success, or nonzero on error, in which case `errno` will be set to indicate the error (see Section "Error Reporting" in `libc`).

`char * mu_format_string (unsigned short *cursor, unsigned short goal, unsigned short width, unsigned short indent, unsigned short subindent, const char *format, ...)` [Function]

This function is just like `mu_format` (see above), except that it returns the result in a dynamically allocated string rather than printing it to a stream.

The return value is the allocated string on success, or `NULL` on error, in which case `errno` will be set to indicate the error (see Section "Error Reporting" in `libc`). If this function succeeds, the returned string must be freed when you are done with it (see Section "Freeing after Malloc" in `libc`).

`int mu_vformat (FILE *stream, unsigned short *cursor, unsigned short goal, unsigned short width, unsigned short indent, unsigned short subindent, const char *format, va_list ap)` [Function]

This function is nearly identical to `mu_format`, except that it takes a `va_list` argument, *ap*, rather than a variable list of arguments. This is useful if you want to write a variadic function which calls `mu_vformat` on its arguments (see Section "Variadic Functions" in `libc`).

`char * mu_vformat_string (unsigned short *cursor, unsigned short goal, unsigned short width, unsigned short indent, unsigned short subindent, const char *format, va_list ap)` [Function]

This function is nearly identical to `mu_format_string`, except that it takes a `va_list` argument, *ap*, rather than a variable list of arguments. See `mu_vformat` above for why this may be useful.

3.1 Controlling Formatted Output

Although the various formatting functions (see Chapter 3 [Formatting Text], page 45) differ slightly in usage, they each take a common set of arguments to control the formatted output. The meaning of each of these arguments is described in the table below:

goal This parameter is the *goal* width. Lines will be wrapped at this width as long as that does not cause words to be split into more than one line, but will be continued beyond this width if wrapping would split words into more than one line.

A *goal* of `0` is treated as infinite.

width This is the absolute maximum length lines are allowed to be. If a line is any longer than this, it will be wrapped even if that means splitting in the middle of a word. If the line *is* split in the middle of a word, a '`-`' will be appended to the end of the line (if there is room[1]) to indicate that the word is continued on the next line.

A *width* of `0` is treated as infinite.

cursor This is the address of an `unsigned short` which holds the current column of output text. You should initialize the value whose address is *cursor* to `0` before calling any of the formatting functions for the first time with that *cursor* argument. See Section 3.2 [Formatting Example], page 47, for an example of how this is used.

Note: *cursor* may **not** be `NULL`.

indent This specifies the column that the first line of text should be indented to. If `*cursor` is already greater than *indent*, then no indentation will be performed (i.e., it will be as though `indent` were `0`).

Lines following the first one are indented according to *subindent*.

subindent This is the indentation to use for lines after the first one (see above). Note that this only applies to a single call of a formatting function. For example, if you do this:

```
unsigned short cursor = 0;
mu_format(stdout, 0, 0, &cursor, 10, 5, "foo\n");
mu_format(stdout, 0, 0, &cursor, 10, 5, "bar\n");
```

both '`foo`' and '`bar`' will be indented 10 characters. If you want '`bar`' to be indented 5 characters, say that explicitly by passing *indent* as 5 (i.e., `mu_format(stdout, 0, 0, &cursor, 5, 5, "bar\n")`).

[1] There might not be room for a '`-`' if `width - indent < 2`.

3.2 Formatting Example

Here is an example illustrating the use of `mu_format` and `mu_format_string`:

```
#include <stdio.h>
#include <stdlib.h>
#include <mu/format.h>

int main(void) {
  char *str;
  unsigned short cursor = 0;

  /* Format a message to standard output. */
  puts("===== mu_format =====");
  mu_format(stdout, &cursor, 40, 50, 4, 2, "\
This is some text. The first line will be indented 4 \
characters, while following lines will be indented 2. Lines \
will be wrapped at 40 characters, except \
reaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaally \
long words, which will be wrapped at 50 characters. A line \
break will appear here:\nno matter what.\n");

  /* Write a similarly formatted message to a string. */
  str = mu_format_string(&cursor, 40, 50, 4, 2,
                         "This text is similarly formatted "
                         "to the text above.\n");

  /* Print the string to standard output. */
  puts("===== mu_format_string =====");
  fputs(str, stdout);

  /* We must free the string since `mu_format_string' dynamically
     allocates it. */
  free(str);

  return 0;
}
```

And here is the output:

```
$ ./format
⊣ ===== mu_format =====
⊣     This is some text. The first line
⊣   will be indented 4 characters, while
⊣   following lines will be indented 2.
⊣   Lines will be wrapped at 40
⊣   characters, except
⊣   reaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-
⊣   aaaaaaaaaally long words, which will
⊣   be wrapped at 50 characters. A line
⊣   break will appear here:
⊣   no matter what.
⊣ ===== mu_format_string =====
⊣     This text is similarly formatted to
⊣   the text above.
```

4 Safety Functions

Some functions, like `malloc`, rarely fail, and it is often impossible to recover when these functions do fail. When using these functions, it may be tempting to simply ignore errors. However, if you do that, it can cause errors to occur elsewhere, making the source of the error hard to find.

For example, if you call `malloc` and it returns `NULL`, then if you later try to deference that pointer, it will cause a segmentation fault. A common solution is to write a function, traditionally called `xmalloc`, which calls `malloc` and terminates the program on failure.

However, it can be a pain to write these functions over and over again, which is why Mu provides them. In addition, the functions provided in Mu report the exact line in your source where the error occurred, which might be able to help you find where your program is using a lot of memory (or where it causes a different type of error to occur).

In addition, Mu provides several convenience functions for error and warning reporting.

The functions described in this chapter are declared in `mu/safe.h`.

`void mu_die (int` ***status***`, const char *`***format***`, ...)` [Function]

> This function prints a formatted error message, specified by *format*, to standard error and exits the program. If *status* is negative, `mu_die` will exit the program by calling `abort` (see Section "Aborting a Program" in `libc`). Otherwise, `mu_die` will pass *status* to `exit` (see Section "Normal Termination" in `libc`). *status* must not be `0`.
>
> If the string to be printed, i.e., the expansion of *format*, ends in a newline ('`\n`'), the expanded string will be printed verbatim. Otherwise, if the expanded string does not end in a newline, context information will be prepended to the string when it is printed.
>
> If an error occurred while printing the error message, `mu_die` will terminate by calling `abort` regardless of the value of *status*.

`int mu_warn (const char *`***format***`, ...)` [Function]

> This function prints a formatted message to standard error in the exact same way as `mu_die` specified above, but it returns instead of calling `exit` or `abort`. If `mu_warn` could successfully print a message to standard error, `mu_warn` will return `0`. Otherwise, `mu_warn` will return a nonzero value.

`void mu_vdie (int` ***status***`, const char *`***format***`, va_list` ***ap***`)` [Function]

`int mu_vwarn (const char *`***format***`, va_list` ***ap***`)` [Function]

> Like `mu_die` and `mu_warn` respectively (see above), except that these functions take a `va_list` argument, *ap*, instead of variable arguments. See Section "Variable Arguments Output" in `libc`.

`void * mu_xmalloc (size_t` ***size***`)` [Function]

> Returns a pointer to dynamically allocated memory of size *size*. The returned pointer must be passed to `free`. See Section "Basic Allocation" in `libc`.

`void * mu_xcalloc (size_t` ***count***`, size_t` ***eltsize***`)` [Function]

> Returns a pointer to dynamically allocated memory of size ***`count * eltsize`***. The returned memory is guaranteed to be initialized to zero, and this function is also

guaranteed to fail safely and reliably in the event that *count* * *eltsize* overflows. The returned pointer must be passed to `free`. See Section "Allocating Cleared Space" in `libc`.

`void * mu_xrealloc (void *ptr, size_t newsize)` [Function]

Changes the size of the block whose address is *ptr* to be *newsize*. If the return value is not equal to *ptr*, *ptr* will be freed. See Section "Changing Block Size" in `libc`.

`void * mu_xreallocarray (void *ptr, size_t count, size_t eltsize)` [Function]

Equivalent to `mu_xrealloc(ptr, count * eltsize)` (see above), except that this function will fail safely and reliably in the event that *count* * *eltsize* overflows. This function is guaranteed to be available even if your system does not define `reallocarray`. See Section "Changing Block Size" in `libc` and Chapter 5 [Compatibility Functions], page 52.

`char * mu_xstrdup (const char *string)` [Function]

Allocates memory large enough to hold a copy of *string*, and copies *string* to the newly allocated memory. *string* must be null-terminated. The returned pointer must be passed to `free`. See Section "Copying Strings and Arrays" in `libc`.

`char * mu_xstrndup (const char *string, size_t max)` [Function]

Like `mu_xstrdup`, but only copies *max* bytes if there was no null byte in the first *max* bytes of *string*. The returned string will always be terminated with a null byte. See Section "Truncating Strings" in `libc`.

`unsigned int mu_xasprintf (char **ptr, const char *format, ...)` [Function]

Allocates memory large enough to hold the output string, and returns the allocated string in *ptr* (which must be passed to `free`). Returns the number of characters in **ptr*, not including the terminating null byte. This function is guaranteed to be available even if your system does not define `asprintf`. See Section "Dynamic Output" in `libc` and Chapter 5 [Compatibility Functions], page 52.

`unsigned int mu_xvasprintf (char **ptr, const char *format, va_list ap)` [Function]

Like `mu_xasprintf` (see above), but takes a `va_list` argument, *ap*, instead of variable arguments. This function is guaranteed to be available even if your system does not define `vasprintf`. See Section "Variable Arguments Output" in `libc` and Chapter 5 [Compatibility Functions], page 52.

`void mu_xformat (FILE *stream, unsigned short *cursor, unsigned short goal, unsigned short width, unsigned short indent, unsigned short subindent, const char *format, ...)` [Function]

`char * mu_xformat_string (unsigned short *cursor, unsigned short goal, unsigned short width, unsigned short indent, unsigned short subindent, const char *format, ...)` [Function] void mu_xvformat (FILE **stream*, unsigned short **cursor*, unsigned short *goal*, unsigned short *width*, unsigned short *indent*, unsigned short *subindent*, const char **format*, va_list *ap*) [Function]

char * mu_xvformat_string (unsigned short **cursor*, unsigned short *goal*, unsigned short *width*, unsigned short *indent*, unsigned short *subindent*, const char **format*, va_list *ap*) [Function]

These functions are like their non-x counterparts, except that they terminate the program on error. See Chapter 3 [Formatting Text], page 45.

MU_OPT_CONTEXT * mu_opt_context_xnew (int *argc*, char ***argv*, const MU_OPT **options*, int *flags*) [Function]

MU_OPT_CONTEXT * mu_opt_context_xnew_with_env (int *argc*, char ***argv*, char ***environment* const MU_OPT **options*, int *flags*) [Function]

Create a new option parsing context. See Chapter 2 [Parsing Options and Environment], page 3.

MU_SUBOPT_CONTEXT * mu_subopt_context_xnew (const char **prog_name*, const char **suboptstr*, const MU_OPT **subopts*) [Function]

Create a new suboption parsing context. See Section 2.7 [Parsing Suboptions], page 25.

void mu_opt_context_xfree (MU_OPT_CONTEXT **context*) [Function]

Free an option parsing context. See Chapter 2 [Parsing Options and Environment], page 3.

void mu_subopt_context_xfree (const MU_SUBOPT_CONTEXT **context*) [Function]

Free a suboption parsing context. See Section 2.7 [Parsing Suboptions], page 25.

void mu_opt_context_xset_no_prefixes (MU_OPT_CONTEXT **context*, ...) [Function]

void mu_opt_context_xset_no_prefix_array (MU_OPT_CONTEXT **context*, char ***no_prefixes*) [Function]

void mu_subopt_context_xset_no_prefixes (MU_SUBOPT_CONTEXT **context*, ...) [Function]

void mu_subopt_context_xset_no_prefix_array (MU_SUBOPT_CONTEXT **context*, char ***no_prefixes*) [Function]

Set alternative negation prefixes for option and suboption parsing contexts. See Section 2.3.1 [Negation Prefixes], page 13.

void mu_opt_context_xadd_options (MU_OPT_CONTEXT **context*, const MU_OPT **options*, enum MU_OPT_WHERE *where*) [Function]

Add *options* to *context*, either at the beginning or end based on *where*. See Chapter 2 [Parsing Options and Environment], page 3.

`void mu_opt_context_xadd_help_options (MU_OPT_CONTEXT *context, int flags)` [Function]

Add help options to *context* based on *flags*. See Section 2.11 [Formatting Help], page 37.

`void mu_xformat_help (FILE *stream, const MU_OPT_CONTEXT *context)` [Function]

`void mu_xformat_help_man (FILE *stream, const MU_OPT_CONTEXT *context)` [Function]

Format a help message from *context*, printing it to *stream*. See Section 2.11 [Formatting Help], page 37.

`char * mu_xformat_help_string (const MU_OPT_CONTEXT *context, unsigned short goal, unsigned short width)` [Function]

`char * mu_xformat_help_man_string (const MU_OPT_CONTEXT *context)` [Function]

Format a help message from *context*, returning it as a dynamically allocated string. See Section 2.11 [Formatting Help], page 37.

5 Compatibility Functions

Some systems provide useful functions, but you cannot use these without worrying about your program not being portable to other systems. That is why Mu provides the functions described below. On systems where these functions are provided, Mu will use the provided functions. This is because, in many cases, the functions are hard optimized, and the alternatives provided by Mu will not be as efficient.

The functions described in this chapter are declared in `mu/compat.h`.

The functions described in this chapter are the only symbols not beginning with '`mu_`' (or '`MU_`' for macros). The reason for this is so that you can simply include `mu/compat.h` in source files where you use these functions, and it automatically becomes portable (as long as it doesn't have any other portability issues).

You don't have to worry about defining any feature test macros, such as `_GNU_SOURCE`, although it doesn't hurt to do so. Just make sure you include `mu/compat.h` **after** any system headers.

All of the functions below set the global variable `errno` on failure. See Section "Error Reporting" in `libc`.

`int asprintf (char **`*`ptr`*`, const char *`*`format`*`, ...)` [Function]

> This function returns a formatted string in `*`*`ptr`* based on the `printf`-style format string *format*. `*`*`ptr`* is dynamically allocated and must be passed to `free`. The return value is the number of characters in `*`*`ptr`* on success, or `-1` on error. See Section "Dynamic Output" in `libc`.

`int vasprintf (char **`*`ptr`*`, const char *`*`format`*`, va_list `*`ap`*`)` [Function]

> Like `asprintf` (see above), but takes a `va_list` argument, *ap*, instead of variable arguments. See Section "Variable Arguments Output" in `libc`.

`char * strchrnul (const char *`*`string`*`, int `*`c`*`)` [Function]

> Returns a pointer to the first occurrence of *c* (converted to a `char`) in *string*, or a pointer to the terminating null byte if *c* does not occur in *string*. See Section "Search Functions" in `libc`

`void * reallocarray (void *`*`ptr`*`, size_t `*`count`*`, `*`eltsize`*`)` [Function]

> Equivalent to `realloc(`*`ptr`*`, `*`count`*` * `*`eltsize`*`)`, except that this function will fail safely and reliably in the event that *`count`*` * `*`eltsize`* overflows. See Section "Changing Block Size" in `libc`.

Appendix A GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users’ Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see https://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `https://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `https://www.gnu.org/licenses/why-not-lgpl.html`.

Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
  Copyright (C)  year  your name.
  Permission is granted to copy, distribute and/or modify this document
  under the terms of the GNU Free Documentation License, Version 1.3
  or any later version published by the Free Software Foundation;
  with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
  Texts.  A copy of the license is included in the section entitled ``GNU
  Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
    with the Invariant Sections being list their titles, with
    the Front-Cover Texts being list, and with the Back-Cover Texts
    being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix C Concept Index

Appendix D Function and Macro Index

Appendix E Type Index

Appendix F Variable and Constant Index