The Option Parsing Library

0.2.0

Generated by Doxygen 1.7.6.1

Tue Apr 23 2013 22:32:15

Contents

Chapter 1

C Environment Library: Option Parsing Library

Version

0.2.0

Author

Jun Woong (woong.jun at gmail.com)

Date

last modified on 2013-04-23

1.1 Introduction

This document specifies the Option Parsing Library which belongs to the C Environment Library. This library is intended to implement all features of Linux's getopt() and getopt_long() in an integrated and thus more consistent fashion; the funtionality of getopt() specified by POSIX is also subsumed by the library.

Precisely, this library:

- supports three ordering modes - argument permutation mode, POSIX-compliant mode and "return-in-order" mode (see below);
- allows multiple scans of possibly multiple sets of program arguments;
- preserves the original program arguments in its original order;
- supports optional long-named options;
- supports optional short-named options; and

- supports abbreviated names for long-named options.

(Suppose that a program supports three long-named options "--html", "--html-false" and "--html-true". For various incomple options given, the library behaves as intuitively as possible, for example, "--html-f" is considered "--html-false", "--html" is recognized as it is and "--html-" results in a warning for its ambiguity. This feature is called "abbreviated names for long-named options.")

The Option Parsing Library reserves identifiers starting with `opt_` and `OPT_`, and imports no other libraries except for the standard library.

1.1.1 Concepts

There are several concepts used to specify the Option Parsing Library.

"Program arguments" or "arguments" for brevity refer to anything given to a program being executed.

"Operands" refer to arguments not starting with a hyphen character or to "-" that denotes the standard input stream. These are sometimes referred to as "non-option arguments."

"Options" refer to arguments starting with a hyphen character but excluding "-". "Short-named options" are options that start with a single hyphen and have a single character following as in "-x"; several short-named options can be grouped after a hyphen as in "-xyz" which is equivalent to "-x -y -z". "Long-named options" are options that start with two hyphens and have a non-empty character sequence following; for example, "--long-option".

If an option takes an additional argument which may immediately follow (possibly with an intervening equal sign) or appear as a separate argument, the argument is called an "option-argument." For long-named options, option-arguments must follow an equal sign unless they appear as separate ones. (See IEEE Std 1003.1, 2004 Edition, 12. Utility Conventions.)

Warning

> Note that, if an option takes an option-argument that is negative thus starts with a minus sign, the argument cannot be a separate one, since the separate argument is to be recognized as another option.

An "option description table" is an array that has a sequence of options to recognize and their properties.

1.2 How to Use The Library

Most programs parse program options in very similar ways. A typical way to handle options is given as the boilerplate code below. You can simply copy it and modify to add options your program supports to the option description table and case labels.

The storage used to parse program arguments is managed by the library.

1.2.1 Ordering Modes

By default, the library processes options and operands as if they were permutated so that operands always follow options. That is, the following two invocations of "util" (where no options take option-arguments) are equivalent (i.e., program cannot tell the difference):

```
util -a -b inputfile outputfile
util inputfile -a outputfile -b
```

This behavior canned "argument permutation," in most cases, helps users to flexibly place options among operands. Some programs, however, require options always proceed operands; for example, given the following line,

```
util -a util2 -b
```

it might be wanted to interpret this as giving "-a" to "util" but "-b" to "util2" which cannot be achieved with argument permutation. For such a case, this library provdes two modes to keep the order in which options and operands are given: the POSIX-compliant mode and the "return-in-order" mode which are denoted by `REQUIRE_ORDER` and `RETURN_IN_ORDER` in a typical implementation of getopt().

In the POSIX-compliant mode, parsing options stops immediately whenever an operand is encountered. This behavior is what POSIX requires, as its name implies.

In the "return-in-order" mode, encountering operands makes the character valued 1 returned as if the operand is an option-argument for the option whose short name has the value 1.

This ordering mode can be controlled by marking a desired ordering mode in an option description table or setting an environment variable (see `opt_t`).

1.2.2 Option Description Tables

An option description table specifies what options should be recognized with their long and short names and what should be done when encountering them, for example, whether an additional option-argument is taken and what its type is, or whether a flag is set and what should be stored into it. Including the ordering mode, all behaviors of the library can be controlled by the table. See `opt_t` for more detailed explanation.

1.3 Boilerplate Code

Using the library starts with invoking opt_init(). It takes an option description table, pointers to parameters of main(), a pointer to an object to which additional information goes during parsing arguments, a default program name used when no program name is available from the environment and a directory separator. If succeeds, it returns a program name; it can be used to issue messages for example.

After the library initialized, opt_parse() insepcts each program argument and performs what specified by the option description table for it. In most cases, this process is made up of a loop containing jumps based on the return value of opt_parse().

As opt_prase() reports that all options have been inspected, a program is granted an access to remaining non-option arguments. These operands are inspected as if they were only arguments to the program.

Since opt_init() allocates storages for duplicating pointers to program arguments, opt_free() should be invoked in order to avoid memory leakage after handling operands has finished.

opt_abort() is a function that stops recognization of options being performed by conf_parse(). All remaining options are regarded as operands. It is useful when a program introduces an option stopper like "--" for its own purposes.

opt.c contains an example designed to use as many facilities of the library as possible in a disabled part and a bolierplate code that is a simplified version of the example is given here:

```
static struct {
    const char *prgname;
    int verbose;
    ...
} option;

int main(int argc, char *argv[])
{
    opt_t tab[] = {
        "verbose", 0,          &(option.verbose), 1,
        "add",     ’a’,         OPT_ARG_NO,        OPT_TYPE_NO,
        "create",  ’c’,         OPT_ARG_REQ,       OPT_TYPE_STR,
        "number",  ’n’,         OPT_ARG_OPT,       OPT_TYPE_REAL,
        "help",    UCHAR_MAX+1, OPT_ARG_NO,        OPT_TYPE_NO,
        NULL,
    }

    option.prgname = opt_init(tab, &argc, &argv, &argptr, PRGNAME, ’/’);
    if (!option.prgname) {
        fprintf(stderr, "%s: failed to parse options\n", PRGNAME);
        return EXIT_FAILURE;
    }

    while ((c = opt_parse()) != -1) {
        switch(c) {
            case ’a’:
                ... --add or -a given ...
                break;
            case ’c’:
                printf("%s: option -c given with value ’%s’\n,
 option.prgname,
                       (const char *)argptr);
                break;
            case ’n’:
                printf("%s: option -n given", option.prgname);
                if (argptr)
                    printf(" with value ’%f’\n", *(const double
 *)argptr);
                else
                    putchar(’\n’);
                break;
            case UCHAR_MAX+1:
                printf("%s: option --help given\n", option.prgname);
                opt_free();
                return 0;

            case 0:
                break;
            case '?':
            case '-':
            case '+':
            case '*':
                fprintf(stderr, "%s: ", option.prgname);
                fprintf(stderr, opt_errmsg(c), (const char *)argptr);
                opt_free();
                return EXIT_FAILURE;
            default:
                assert(!"not all options covered -- should never reach
here");
                break;
        }
    }

    if (option.verbose)
        puts("verbose flag is set");

    if (argc > 1) {
        printf("non-option ARGV-arguments:");
        for (i = 1; i < argc; i++)
            printf(" %s", argv[i]);
        putchar('\n');
    }

    opt_free();

    return 0;
}
```

The struct object `option` manages all objects set by program arguments. Note that it has the static storage duration; since its member is used as an initializer for the option description table that is an array, it has to have the static storage duration; C99 has removed this restriction.

Each row in the option description table specifies options to recognize:

- "--verbose" has no short name and has a flag variable set to 1 when encountered;
- "--add" has a short name "-a" and takes no option-arguments;
- "--create" has a short name "-c" and requires an option-argument of the string type;
- "--number" has a short name "-n" and takes an optional option-argument of the real type; and
- "--help" has no short name and takes no option-arguments.

Because the failure of opt_init() means that memory allocation failed, you do not have to call opt_free() before terminating the program.

The case labes above one handling 0 are for options given in `tab`. Those labels below them are for exceptional cases and opt_errmsg() helps to construct appropriate messages for them. In addtion, there are other ways to handle those cases; see opt_errmsg() for details. Remember that, if invoked, opt_free() should be invoked after all program arguments including non-option arguments have been processed. Since opt_init() makes copies of pointers in `argv` and opt_free() releases storages for them, any access to them gets invalidated by opt_free().

1.4 Future Directions

No future change on this library planned yet.

1.5 Contact Me

Visit `http://code.woong.org` to get the lastest version of this library. Only a small portion of my homepage (`http://www.woong.org`) is maintained in English, thus one who is not good at Korean would have difficulty when navigating most of other pages served in Korean. If you think the information you are looking for is on pages written in Korean, do not hesitate to send me an email to ask for help.

Any comments about the library are welcomed. If you have a proposal or question on the library just email me, and I will reply as soon as possible.

1.6 Copyright

This package is an option parser implementation by Jun Woong. The implementation was written so as to conform with the Standard C published by ISO 9899:1990 and ISO 9899:1999.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 opt_t Struct Reference

represents an element of an option description table.

`#include <opt.h>`

Data Fields

- char ∗ lopt
- int sopt
- int ∗ flag
- int arg

4.1.1 Detailed Description

represents an element of an option description table.

`opt_t` represents an element of an option description table that is used for a user program to specify options and their properties. A option description table is an array of `opt_t`, each element of which is consisted of four members, two of which have overloaded meanings:

- `lopt:` long-named option that can be invoked by two precedeeing hypens; optional if a short-named option given; however, encouraged to always provide a long-named option

- `sopt:` short-named option that can be invoked by a precedeeing hypens; optional if both a long-named option and a flag variable provided

- `flag:` if an option does not take an additional argument, `flag` can point to an object (called "flag variable") that is set to the value of `arg` when `lopt` or `sopt` option encountered; if an option can take an additional argument, `flag` specifies whether the option-argument is mandatory (with `OPT_ARG_REQ`) or optional (with `OPT_ARG_OPT`)

- `arg`: if an option does not take an additional argument, `arg` has the value to be stored into a flag variable when `lopt` or `sopt` option encountered; if an option can take an additional argument, `arg` specifies the type of the option-argument using `OPT_TYPE_BOOL` (option-arguments starting with ’t’, ’T’, ’y’, ’Y’ and ’1’ means true and others false, int), `OPT_TYPE_INT` (signed integer, long), `OPT_TYPE_UINT` (unsigned integer, unsigned long), `OPT_TYPE_REAL` (floating-point number, double) and `OPT_TYPE_STR` (string, char $*$)

To mark an end of the table, the `lopt` member of the last element has to be set to a null pointer. If the `flag` member points to a flag variable, the pointed integer object is initalized to be 0 by opt_init().

For `OPT_TYPE_INT` and `OPT_TYPE_UINT`, the conversion of a given option-argument recognizes the C-style prefixes; numbers starting with 0 are treated as octal, and those with 0x or 0X are treated as hexadecimal.

Some examples follow:

```
opt_t options[] = {
    { "verbose", ’v’, &option_verbose, 1 },
    { "brief",   ’b’, &option_verbose, 0 },
    { NULL, }
};
```

This example says that two options ("--verbose" or "-v" and "--brief" or "-b") are recognized and `option_verbose` is set to 1 when "--verbose" or "-v" given, and set to 0 when "--brief" or "-b" given.

```
opt_t options[] = {
    "version",  ’v’,         OPT_ARG_NO, OPT_TYPE_NO,
    "help",     UCHAR_MAX+1, OPT_ARG_NO, OPT_TYPE_NO,
    "morehelp", UCHAR_MAX+2, OPT_ARG_NO, OPT_TYPE_NO,
    NULL,
};
```

This example shows options that do not take any additional arguments. Setting the `flag` member to a null pointer also says the option takes no argument in which case the value of the `arg` member ignored. Thus, the above example can be written as follows without any change on the behavior:

```
opt_t options[] = {
    "version",  ’v’,         NULL, 0,
    "help",     UCHAR_MAX+1, NULL, 0,
    "morehelp", UCHAR_MAX+2, NULL, 0,
    NULL,
};
```

where you can put any integer in the place of 0. The former is preferred, however, since it shows more explicitly the fact that no additional arguments consumed after each of the options.

When only long-named options need to be provided without introducing flag variables, values from `UCHAR_MAX+1` to `INT_MAX` (inclusive) can be used for the `sopt` member; both are defined in <limits.h>. (Even if the C standard does not require `UCHAR_MAX` less than `INT_MAX`, many parts of C, especially, of the standard library cannot work correctly without such a relationship on a hosted implementation.)

```
opt_t options[] = {
    "",   ’x’, OPT_ARG_NO, OPT_TYPE_NO,
    NULL,
};
```

On the other hand, providing an empty string for the `lopt` member as in this example can specify that an option is only short-named. Note that, however, this is discouraged; long-named options are much more user-friendly especially for novices.

```
opt_t options[] = {
    "input", ’i’, OPT_ARG_REQ, OPT_TYPE_STR,
    "port",  ’p’, OPT_ARG_REQ, OPT_TYPE_UINT,
    "start", ’s’, OPT_ARG_REQ, OPT_TYPE_REAL,
    "end",   ’e’, OPT_ARG_REQ, OPT_TYPE_REAL,
    NULL,
};
```

This example shows options that take additional arguments. `OPT_ARG_REQ` for the `flag` member specifies that the option requires an option-argument and that the type of the argument is given in the `arg` member. For `OPT_TYPE_INT`, `OPT_TYPE_UINT` and `OPT_TYPE_REAL`, strtol(), strtoul() and strtod() are respectively used to convert option-arguments.

```
opt_t options[] = {
    "negative", ’n’, OPT_ARG_OPT, OPT_TYPE_REAL,
    NULL,
};
```

This table specifies the option "--negative" or "-n" takes an optionally given argument. If an option-argument with the expected form (which is determined by strtod() in this case) follows the option, it is taken. If there is no argument, or is an argument but has no expected form, the option works as if `OPT_ARG_OPT` and `OPT_TYPE_REAL` are replaced by `OPT_ARG_NO` and `OPT_TYPE_NO`.

The following examples show how to control the ordering mode.

```
opt_t options[] = {
    "+", 0, OPT_ARG_NO, OPT_TYPE_NO,
    ...
    NULL,
};
```

Setting the first long-named option to "+" or setting the environment variable named `POSIXLY_CORRECT` says option processing performed by opt_parse() immediately stops whenever an operand encountered (which POSIX requires).

```
opt_t options[] = {
    "-", 0, OPT_ARG_NO, OPT_TYPE_NO,
    ...
    NULL,
};
```

In addition, setting the first long-named option to "-" makes opt_parse() returns the character valued 1 when encounters an operand as if the operand is an option-argument for the option whose short name has the value 1.

4.1.2 Field Documentation

4.1.2.1 int **opt_t::arg**

value for flag variable or type of additional argument

4.1.2.2 int∗ **opt_t::flag**

pointer to flag varible or information about additional argument

4.1.2.3 char∗ **opt_t::lopt**

long-named option (optional for some cases)

4.1.2.4 int **opt_t::sopt**

short-named option (optional for some cases)

The documentation for this struct was generated from the following file:

- opt.h

Chapter 5

File Documentation

5.1 opt.c File Reference

Source for Option Parsing Library (CEL)

`#include <assert.h> #include <ctype.h> #include <errno.h> #include <limits.h> #include <stddef.h> #include <stdio.h> #include <string.h> #include <stdlib.h> #include "opt.h"` Include dependency graph for opt.c:

Functions

- const char ∗() opt_init (const opt_t ∗o, int ∗pc, char ∗∗pv[ ], const void ∗∗pa, const char ∗name, int sep)

  prepares to start parsing program arguments.
- int() opt_parse (void)

  parses program options.
- void() opt_abort (void)

  aborts parsing options.
- const char ∗ opt_errmsg (int c)

  returns a diagnostic format string for an error code.
- void() opt_free (void)

  cleans up any storage used and disables the internal state.

5.1.1 Detailed Description

Source for Option Parsing Library (CEL)

5.1.2 Function Documentation

5.1.2.1 void() opt_abort (void)

aborts parsing options.

opt_abort() aborts parsing options immediately handling the remaining arguments as operands. Having invoked opt_abort(), opt_parse() need not be called to access to operands; `argc` and @ argv are properly adjusted as if opt_parse() has returned -1 except that the remaining options (if any) are treated as operands. If opt_parse() invoked after aborting the parsing, opt_parse() does nothing and returns -1.

Returns

nothing

5.1.2.2 const char∗ opt_errmsg (int *c*)

returns a diagnostic format string for an error code.

Given an error code that is one of ’?’, ’-’, ’+’ and ’∗’, opt_errmsg() returns a string that can be used as a format string for the printf() family. A typical way to handle exceptional cases opt_parse() may return is as follows:

```
switch(c) {
    ... cases for valid options ...
    case 0:
        break;
    case ’?’:
        fprintf(stderr, "%s: unknown option ’%s’\n", option.prgname, (
const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
    case ’-’:
        fprintf(stderr, "%s: no or invalid argument given for ’%s’\n",
option.prgname,
                (const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
    case ’+’:
        fprintf(stderr, "%s: option ’%s’ takes no argument\n", option.
prgname,
                (const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
    case ’*’:
        fprintf(stderr, "%s: ambiguous option ’%s’\n", option.prgname,
                (const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
    default:
        assert(!"not all options covered -- should never reach here");
        break;
}
```

where "case 0" is for options that sets a flag variable so in most cases leaves nothing for a user code to do. The following four case labels handle erroneous cases and the default case is there to handle what is never supposed to happen.

As repeating this construct for every program using this library is cumbersome, for convenience opt_errmsg() is provided to handle those four erroneous cases as follows:

```
switch(c) {
    ... cases for valid options ...
    case 0:
        break;
    case '?':
    case '-':
    case '+':
    case '*':
        fprintf(stderr, "%s: ", option.prgname);
        fprintf(stderr, opt_errmsg(c), (const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
    default:
        assert(!"not all options covered -- should never reach here");
        break;
}
```

or more compatly:

```
switch(c) {
    ... cases for valid options ...
    case 0:
        break;
    default:
        fprintf(stderr, "%s: ", option.prgname);
        fprintf(stderr, opt_errmsg(c), (const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
}
```

The difference of the last two is that the latter turns the assertion in the former (that possibly gets dropped from the delivery code) into a defensive check (that does not). Note that the returned format string contains a newline.

If a user needs flexibility on the format of diagnostics or actions done in those cases, resort to the cumbersome method shown first.

Possible exceptions: none

Unchecked errors: none

Parameters

`in`	*c*	error code opt_parse() returned

Returns

format string for diagnostic message

5.1.2.3 void() opt_free (void)

cleans up any storage used and disables the internal state.

opt_free() cleans up any storage allocated by opt_init() and used by opt_parse(). It also initializes the internal state, which allows for multiple scans; see opt_init() for some caveat when scanning options multiple times.

Warning

> opt_free(), if invoked, should be invoked after all arguments including operands have been processed. Since opt_init() makes copies of pointers in `argv` of main(), and opt_free() releases storages for them, any access to them gets invalidated by opt_free().

Possible exceptions: none

Unchecked errors: none

Returns

> nothing

5.1.2.4 const char∗() **opt_init** (const **opt_t** ∗ *o,* int ∗ *pc,* char ∗∗ *pv[ ],* const void ∗∗ *pa,* const char ∗ *name,* int *sep*)

prepares to start parsing program arguments.

opt_init() prepares to start parsing program arguments. It takes everything necessary to parse arguments and sets the internal state properly that is referred to by opt_parse() later. It also constructs a more readable program name by omitting any path preceeding the pure name. To do this job, it takes a directory separator character through `sep` and a default program name through `name` that is used when no name is available through `argv`. A typical use of opt_init() is given at the commented-out example code in the source file.

On success, opt_init() returns a program name (non-null pointer). On failure, it returns the null pointer; opt_init() may fail only when allocating small-sized storage fails, in which case further execution of the program is very likely to fail due to the same problem.

opt_init() can be called again for multiple scans of options, but only after opt_free() has been invoked. Note that, in such a case, only the internal state and flag variables given with an option description table are initialized. Other objects probably used for processing options in a user code retain their values, thus should be initialized explicitly by a user code. A convenient way to handle that initialization is to introduce a structure grouping all such objects. For example:

```
struct option {
    int html;
    const char *input;
    double val;
    ...
} option;
```

where, say, `html` is a flag variable for --html, `input` is an argument for -i or --input, `val` is an argument for -n or --number, and so on. By assigning a properly initialized value to the structure, the initialization can be readily done:

```
For C90:
    struct option empty = { 0, };
    option = empty;

For C99:
    option = (struct option){ 0, };
```

Note that, in this example, the object `option` should have the static storage duration in order for the `html` member to be given as an initalizer for an option description table; C99 has no such restriction.

Possible exceptions: none

Unchecked errors: non-null but invalid arguments given

Parameters

`in`	*o*	option description table
`in`	*pc*	pointer to `argc`
`in`	*pv*	pointer to `argv`
`in`	*pa*	pointer to object to contain argument or erroneous option
`in`	*name*	default program name
`in`	*sep*	directory separator (e.g., '/' on UNIX-like systems)

Returns

program name or null pointer

Return values

non-null	program name
null	failure

5.1.2.5 int() opt_parse (void)

parses program options.

opt_parse() parses program options. In typical cases, the caller of opt_parse() has to behave based on the result of opt_parse() that is one of:

- '?': unrecognized option; the pointer given to opt_init() through `pa` points to a string that represents the option
- '-': valid option, but no option-argument given even if the option requires it, or invalid option-argument given; the pointer given to opt_init() through `pa` points to a string that represents the option
- '+': valid option, but an option-argument given even if the option takes none; the pointer given to opt_init() through `pa` points to a string that represents the option
- '∗': ambiguious option; it is impossible to identify a unique option with the given prefix of a long-named option

- 0: valid option; a given flag variable is set properly, thus nothing for a user code to do
- -1: all options have been processed
- 1: (only when the first long-named option is "-") an operand is given; the pointer given to opt_init() through `pa` points to the operand

This means that a valid short-named option cannot have the value of ’?’, ’-’, ’+’, ’*’, -1 or 1; 0 is allowed to say no short-named option given when a flag variable is provided; see `opt_t` for details. In addition, ’=’ cannot also be used.

If an option takes an option-argument, the pointer whose address passed to opt_init() through `pa` is set to point to the argument. A subsequent call to opt_parse() may overwrite it unless the type is `OPT_TYPE_STR`.

After opt_parse() returns -1, `argc` and `argv` (precisely, objects whose addresses are passed to opt_init() through `pc` and `pv`) are adjusted for a user code to process remaining operands as if there were no options or option-arguments in program arguments; see the commented-out example code given in the source file. Once opt_parse() starts the parsing, `argc` and the elements of `argv` are indeterminate, thus an access to them is not allowed.

opt_parse() changes neither the original contents of `argv` nor strings pointed to by the elements of `argv`, thus by granting copies of `argc` and `argv` to opt_init() as in the following example, a user code can access to program arguments unchanged if necessary even after options have been parsed by opt_parse().

```
int main(int argc, char *argv[])
{
    const void *arg;
    int argc2 = argc;
    char **argv2 = argv;
    ...
    pname = opt_init(options, &argc2, &argv2, &arg, "program", ’/’);
    while (opt_parse() != -1) {
        ...
    }

    for (i = 1; i < argc2; i++)
        printf("operands: %s\n", argv2[i]);

    opt_free();

    for (i = 1; i < argc; i++)
        printf("untouched program arguments: %s\n", argv[i]);
    ...
}
```

Warning

opt_init() has to be invoked successfully before calling opt_parse().

Possible exceptions: none

Unchecked errors: `argc` or `argv` modified by a user between calls to opt_parse(), modifying an object containing an option-argument or a problematic option

5.2 opt.h File Reference

Documentation for Option Parsing Library (CEL)

This graph shows which files directly or indirectly include this file:

Data Structures

- struct opt_t

  represents an element of an option description table.

Defines

macros for describing option-arguments; see @c opt_t

- #define OPT_ARG_REQ (&opt_arg_req)
- #define OPT_ARG_NO (&opt_arg_no)
- #define OPT_ARG_OPT (&opt_arg_opt)

Enumerations

- enum { OPT_TYPE_NO, OPT_TYPE_BOOL, OPT_TYPE_INT, OPT_TYPE_UINT, OPT_TYPE_REAL, OPT_TYPE_STR }

  defines enum contants for types of argument conversion.

Functions

option processing functions:

- const char ∗ opt_init (const opt_t ∗, int ∗, char ∗∗[], const void ∗∗, const char ∗, int)

  prepares to start parsing program arguments.
- int opt_parse (void)

  parses program options.
- void opt_abort (void)

  aborts parsing options.
- const char ∗ opt_errmsg (int)

  returns a diagnostic format string for an error code.
- void opt_free (void)

  cleans up any storage used and disables the internal state.

5.2.1 Detailed Description

Documentation for Option Parsing Library (CEL) Header for Option Parsing Library (CEL)

5.2.2 Define Documentation

5.2.2.1 #define **OPT_ARG_NO** (&opt_arg_no)

no argument taken

5.2.2.2 #define **OPT_ARG_OPT** (&opt_arg_opt)

optional argument

5.2.2.3 #define **OPT_ARG_REQ** (&opt_arg_req)

mandatory argument

5.2.3 Enumeration Type Documentation

5.2.3.1 anonymous enum

defines enum contants for types of argument conversion.

Enumerator:

OPT_TYPE_NO	cannot have type
OPT_TYPE_BOOL	has boolean (int) type
OPT_TYPE_INT	has integer (long) type
OPT_TYPE_UINT	has unsigned integer (unsigned long) type
OPT_TYPE_REAL	has floating-point (double) type
OPT_TYPE_STR	has string (char $*$) type

5.2.4 Function Documentation

5.2.4.1 void **opt_abort** (void)

aborts parsing options.

opt_abort() aborts parsing options immediately handling the remaining arguments as operands. Having invoked opt_abort(), opt_parse() need not be called to access to operands; `argc` and @ argv are properly adjusted as if opt_parse() has returned -1 except that the remaining options (if any) are treated as operands. If opt_parse() invoked after aborting the parsing, opt_parse() does nothing and returns -1.

Returns

nothing

5.2.4.2 const char∗ opt_errmsg (int *c*)

returns a diagnostic format string for an error code.

Given an error code that is one of '?', '-', '+' and '∗', opt_errmsg() returns a string that can be used as a format string for the printf() family. A typical way to handle exceptional cases opt_parse() may return is as follows:

```
switch(c) {
    ... cases for valid options ...
    case 0:
        break;
    case '?':
        fprintf(stderr, "%s: unknown option '%s'\n", option.prgname, (
const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
    case '-':
        fprintf(stderr, "%s: no or invalid argument given for '%s'\n",
option.prgname,
                (const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
    case '+':
        fprintf(stderr, "%s: option '%s' takes no argument\n", option.
prgname,
                (const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
    case '*':
        fprintf(stderr, "%s: ambiguous option '%s'\n", option.prgname,
                (const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
    default:
        assert(!"not all options covered -- should never reach here");
        break;
}
```

where "case 0" is for options that sets a flag variable so in most cases leaves nothing for a user code to do. The following four case labels handle erroneous cases and the default case is there to handle what is never supposed to happen.

As repeating this construct for every program using this library is cumbersome, for convenience opt_errmsg() is provided to handle those four erroneous cases as follows:

```
switch(c) {
    ... cases for valid options ...
    case 0:
        break;
    case '?':
    case '-':
    case '+':
    case '*':
        fprintf(stderr, "%s: ", option.prgname);
        fprintf(stderr, opt_errmsg(c), (const char *)argptr);
        opt_free();
        return EXIT_FAILURE;
    default:
            assert(!"not all options covered -- should never reach here");
            break;
    }
```

or more compatly:

```
    switch(c) {
        ... cases for valid options ...
        case 0:
            break;
        default:
            fprintf(stderr, "%s: ", option.prgname);
            fprintf(stderr, opt_errmsg(c), (const char *)argptr);
            opt_free();
            return EXIT_FAILURE;
    }
```

The difference of the last two is that the latter turns the assertion in the former (that possibly gets dropped from the delivery code) into a defensive check (that does not). Note that the returned format string contains a newline.

If a user needs flexibility on the format of diagnostics or actions done in those cases, resort to the cumbersome method shown first.

Possible exceptions: none

Unchecked errors: none

Parameters

`in`	*c*	error code opt_parse() returned

Returns

format string for diagnostic message

5.2.4.3 void opt_free (void)

cleans up any storage used and disables the internal state.

opt_free() cleans up any storage allocated by opt_init() and used by opt_parse(). It also initializes the internal state, which allows for multiple scans; see opt_init() for some caveat when scanning options multiple times.

Warning

opt_free(), if invoked, should be invoked after all arguments including operands have been processed. Since opt_init() makes copies of pointers in `argv` of main(), and opt_free() releases storages for them, any access to them gets invalidated by opt_free().

Possible exceptions: none

Unchecked errors: none

Returns

nothing

5.2.4.4 const char∗ opt_init (const opt_t ∗ *o,* int ∗ *pc,* char ∗∗ *pv[ ],* const void ∗∗ *pa,* const char ∗ *name,* int *sep*)

prepares to start parsing program arguments.

opt_init() prepares to start parsing program arguments. It takes everything necessary to parse arguments and sets the internal state properly that is referred to by opt_parse() later. It also constructs a more readable program name by omitting any path preceeding the pure name. To do this job, it takes a directory separator character through `sep` and a default program name through `name` that is used when no name is available through `argv`. A typical use of opt_init() is given at the commented-out example code in the source file.

On success, opt_init() returns a program name (non-null pointer). On failure, it returns the null pointer; opt_init() may fail only when allocating small-sized storage fails, in which case further execution of the program is very likely to fail due to the same problem.

opt_init() can be called again for multiple scans of options, but only after opt_free() has been invoked. Note that, in such a case, only the internal state and flag variables given with an option description table are initialized. Other objects probably used for processing options in a user code retain their values, thus should be initialized explicitly by a user code. A convenient way to handle that initialization is to introduce a structure grouping all such objects. For example:

```
struct option {
    int html;
    const char *input;
    double val;
    ...
} option;
```

where, say, `html` is a flag variable for --html, `input` is an argument for -i or --input, `val` is an argument for -n or --number, and so on. By assigning a properly initialized value to the structure, the initialization can be readily done:

```
For C90:
    struct option empty = { 0, };
    option = empty;

For C99:
    option = (struct option){ 0, };
```

Note that, in this example, the object `option` should have the static storage duration in order for the `html` member to be given as an initalizer for an option description table; C99 has no such restriction.

Possible exceptions: none

Unchecked errors: non-null but invalid arguments given

Parameters

`in`	*o*	option description table
`in`	*pc*	pointer to `argc`
`in`	*pv*	pointer to `argv`
`in`	*pa*	pointer to object to contain argument or erroneous option
`in`	*name*	default program name
`in`	*sep*	directory separator (e.g., '/' on UNIX-like systems)

Returns

program name or null pointer

Return values

non-null	program name
null	failure

5.2.4.5 int **opt_parse** (void)

parses program options.

opt_parse() parses program options. In typical cases, the caller of opt_parse() has to behave based on the result of opt_parse() that is one of:

- '?': unrecognized option; the pointer given to opt_init() through `pa` points to a string that represents the option
- '-': valid option, but no option-argument given even if the option requires it, or invalid option-argument given; the pointer given to opt_init() through `pa` points to a string that represents the option
- '+': valid option, but an option-argument given even if the option takes none; the pointer given to opt_init() through `pa` points to a string that represents the option
- '∗': ambiguious option; it is impossible to identify a unique option with the given prefix of a long-named option
- 0: valid option; a given flag variable is set properly, thus nothing for a user code to do
- -1: all options have been processed
- 1: (only when the first long-named option is "-") an operand is given; the pointer given to opt_init() through `pa` points to the operand

This means that a valid short-named option cannot have the value of '?', '-', '+', '∗', -1 or 1; 0 is allowed to say no short-named option given when a flag variable is provided; see `opt_t` for details. In addition, '=' cannot also be used.

If an option takes an option-argument, the pointer whose address passed to opt_init() through `pa` is set to point to the argument. A subsequent call to opt_parse() may overwrite it unless the type is `OPT_TYPE_STR`.

After opt_parse() returns -1, `argc` and `argv` (precisely, objects whose addresses are passed to opt_init() through `pc` and `pv`) are adjusted for a user code to process remaining operands as if there were no options or option-arguments in program arguments; see the commented-out example code given in the source file. Once opt_parse() starts the parsing, `argc` and the elements of `argv` are indeterminate, thus an access to them is not allowed.

opt_parse() changes neither the original contents of `argv` nor strings pointed to by the elements of `argv`, thus by granting copies of `argc` and `argv` to opt_init() as in the following example, a user code can access to program arguments unchanged if necessary even after options have been parsed by opt_parse().

```
int main(int argc, char *argv[])
{
    const void *arg;
    int argc2 = argc;
    char **argv2 = argv;
    ...
    pname = opt_init(options, &argc2, &argv2, &arg, "program", '/');
    while (opt_parse() != -1) {
        ...
    }

    for (i = 1; i < argc2; i++)
        printf("operands: %s\n", argv2[i]);

    opt_free();

    for (i = 1; i < argc; i++)
        printf("untouched program arguments: %s\n", argv[i]);
    ...
}
```

Warning

opt_init() has to be invoked successfully before calling opt_parse().

Possible exceptions: none

Unchecked errors: `argc` or `argv` modified by a user between calls to opt_parse(), modifying an object containing an option-argument or a problematic option