FLEXOP: a Flexible Command Option Parsing Library

VERSION 1.0

Hui Liu
2018

Contents

1 Introduction

1.1 Overview

Command options (arguments) are important parts of Unix, Linux and Mac OS operating systems. In these operating systems, most command line utilies (programs, applications) have options, such as `ls`. Its manual can be read by running command `man ls`, which should be similar to the following,

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

-a, --all
      do not ignore entries starting with .

-A, --almost-all
      do not list implied . and ..

-b, --escape
      print C-style escapes for nongraphic characters

-B, --ignore-backups
      do not list implied entries ending with ~

-C list entries by columns

-d, --directory
      list directories themselves, not their contents

-D, --dired
      generate output designed for Emacs' dired mode

-f do not sort, enable -aU, disable -ls --color

-h, --human-readable
      with -l and -s, print sizes like 1K 234M 2G etc.

-l use a long listing format
```

The arguments start with "-" are options, which control the behavior of a command line utility, such as `ls -a` will show hidden files, and `ls -l` will display output in long listing format, which shows file size, time stampes and attributes. Another advantage of options is that a utility will be friendly to script programming and automation.

1.2 Credit

The original code was from PHG (http://lsec.cc.ac.cn/phg/), a parallel framework for adaptive finite element methods.

1.3 License

The package uses GPL license. If you have any issue, please contact: hui.sc.liu@gmail.com

1.4 Citation

If you use FLEXOP library, please cite it like this,

```
@misc{flexop-library,
    author="Hui Liu",
    title="FLEXOP: a flexible command option parsing library",
    year="2018",
    note={\url{https://github.com/huiscliu/flexop/}}
}
```

1.5 Website

The official website for FLEXOP is https://github.com/huiscliu/flexop/.

2 Installation

FLEXOP uses autoconf and make to detect system parameters and user set parameters, to build and to install.

2.1 Configuration

The simplest way to configure is to run command:

```
./configure
```

2.2 Options

The script configure has many options, if user would like to check, run command:

```
./configure --help
```

Output will be like this,

```
‘configure’ configures this package to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE. See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Installation directories:
  --prefix=PREFIX install architecture-independent files in PREFIX
                          [/usr/local/flexop]
  --exec-prefix=EPREFIX install architecture-dependent files in EPREFIX
                          [PREFIX]

By default, ‘make install’ will install all the files in
‘/usr/local/flexop/bin’, ‘/usr/local/flexop/lib’ etc. You can specify
an installation prefix other than ‘/usr/local/flexop’ using ‘--prefix’,
for instance ‘--prefix=HOME’.

Optional Features:
  --disable-option-checking ignore unrecognized --enable/--with options
  --disable-FEATURE do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG] include FEATURE [ARG=yes]
```

```
  --disable-assert turn off assertions
  --enable-big-int use long int for INT
  --disable-big-int use int for INT (default),
  --with-int=type integer type(long|long long)
  --enable-long-double use long double for FLOAT
  --disable-long-double use double for FLOAT (default)

Some influential environment variables:
  CC C compiler command
  CFLAGS C compiler flags
  LDFLAGS linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  LIBS libraries to pass to the linker, e.g. -l<library>
  CPPFLAGS (Objective) C/C++ preprocessor flags, e.g. -I<include dir> if
              you have headers in a nonstandard directory <include dir>
  CPP C preprocessor
```

The default integer and floating point number are `int` and `double`. However, user can change the type of integer, such as `long long int`, by using options `--disable-big-int --with-int="long long"`, and the type of floating point number, such as `long double`, by using `--enable-long-double`. The integer number has three choices, `int`, `long int` and `long long int`, and the floating point number has two choices, `double` and `long double`.

2.3 Compilation

After configuration, `Makefile` and related scripts will be set correctly. A simple **make** command can compile the package,

```
make
```

2.4 Installation

Run command:

```
make install
```

The package will be installed to a directory. The default is /usr/local/flexop/. A different directory can be set by `--prefix=DIR`.

3 Data Types

FLEXOP_FLOAT is the floating point number type in FLEXOP, which could be double or long double, depending on the configuration. Its formal definition is as follows.

```
#if FLEXOP_USE_LONG_DOUBLE
typedef long double FLEXOP_FLOAT;
#else
typedef double FLEXOP_FLOAT;
#endif
```

FLEXOP_INT is the integer type, and as mentioned before, it could be int, long int, or long long int.

```
#if FLEXOP_USE_LONG_LONG
typedef long long int FLEXOP_INT;
#elif FLEXOP_USE_LONG
typedef long int FLEXOP_INT;
#else
typedef int FLEXOP_INT;
#endif
```

FLEXOP_UINT is the unsigned integer type, and it could be unsigned int, unsigned long int, or unsigned long long int, depending on the configuration.

```
#if FLEXOP_USE_LONG_LONG
typedef unsigned long long int FLEXOP_UINT;
#elif FLEXOP_USE_LONG
typedef unsigned long int FLEXOP_UINT;
#else
typedef unsigned int FLEXOP_UINT;
#endif
```

The FLEXOP supports many option types, such as integer, unsigned integer, floating point number and vector, which is represented by FLEXOP_VTYPE. Its formal definition is shown by the follows.

```
typedef enum {
    VT_TITLE,

    VT_BOOL,
    VT_KEYWORD,
    VT_HANDLER,

    VT_INT,
    VT_UINT,
```

```
    VT_FLOAT,
    VT_STRING,

    VT_VEC_INT,
    VT_VEC_UINT,
    VT_VEC_FLOAT,
    VT_VEC_STRING,

} FLEXOP_VTYPE;
```

Here are detailed explanations:

- VT_TITLE defines a section. For example, in many applications, options may be divided into many sections, such as model, numerical, gridding, visualization. In FLEXOP, when a title (or section) is registered, all following options registered after the title belong to this section, unless a new section (title) is registered.
- VT_BOOL defines a boolean option, which has true (1) or false (0) status.
- VT_KEYWORD defines a keyword, whose value is from a pre-defined set.
- VT_HANDLER defines a user-defined handler, which handles option parsing using the provided function. It provides a method to define new parsers and even to replace all build-in parsers, such as integer, floating point number, string and vector.
- VT_INT defines an integery.
- VT_UINT defines an unsigned integery.
- VT_FLOAT defines a floating point number.
- VT_STRING defines a string.
- VT_VEC_INT defines a vector of integers.
- VT_VEC_UINT defines a vector of unsigned integers.
- VT_VEC_FLOAT defines a vector of floating point number.
- VT_VEC_STRING defines a vector of strings.

The VT_HANDLER type requires a user-provided function, which has the following type. It returns 0 if successful, otherwise returns non-zero value.

```
typedef int (*FLEXOP_HANDLER)(FLEXOP_KEY *o, const char *arg);
```

The vector has a uniform structure for integar, unsigned integer, floating point number and string.

```
typedef struct FLEXOP_VEC_
{
    void *d; /* data */
    char *key; /* key */

    FLEXOP_VTYPE type; /* data type of the members */
    FLEXOP_INT size; /* size of the vector */
    ...

} FLEXOP_VEC;
```

4 Utilities

4.1 Print

flexop_printf outputs to stdout.

```
int flexop_printf(const char *fmt, ...);
```

flexop_error prints output error message and quits with error code.

```
void flexop_error(int code, const char *fmt, ...);
```

flexop_warning print warning info.

```
void flexop_warning(const char *fmt, ...);
```

flexop_set_print_mark sets mark for flexop_printf to control its behavior, if m is non-zero (true) value, then flexop_printf acts as a normal print function. However, if m is zero (false), flexop_printf will not print anything. This function is important to parallel computing, since only one process prints info to stdout usually.

```
void flexop_set_print_mark(int m);
```

4.2 Memory

The following functions provide memory allocation, calloc, reallocation, freeing and copying.

```
void * flexop_malloc(size_t n);
void * flexop_calloc(size_t n);
void * flexop_realloc(void *ptr, size_t n);
void flexop_free(void *p);
```

4.3 Conversion

flexop_atoi converts string to integer, which checks if input is legal integer.

```
FLEXOP_INT flexop_atoi(const char *ptr);
```

flexop_atou converts string to unsigned integer, which checks if input is legal integer.

```
FLEXOP_UINT flexop_atou(const char *ptr);
```

flexop_atof converts string to floating point number, which checks if input is legal.

```
FLEXOP_FLOAT flexop_atof(const char *ptr);
```

4.4 Vector

flexop_vec_initialized checks if a vector is initialized or not.

```
int flexop_vec_initialized(FLEXOP_VEC *vec);
```

flexop_vec_init initializes a vector. type is the type of members, such as integer and floating point number. tsize is the size of a member, which is unknown for internal types, such as integer, floating point number, but it is possible that the size is uncertain for special data structures. key is a string.

```
void flexop_vec_init(FLEXOP_VEC *vec, FLEXOP_VTYPE type, FLEXOP_INT tsize,
    const char *key);
```

flexop_vec_destroy destroys a vector and releases memory.

```
void flexop_vec_destroy(FLEXOP_VEC *vec);
```

flexop_vec_add_entry adds an entry to a vector, which is a pointer to the entry, such as pointer to integer, pointer to floating poing number, or pointer to a string (char *).

```
void flexop_vec_add_entry(FLEXOP_VEC *v, void *e);
```

flexop_vec_get_size gets the size of a vector.

```
FLEXOP_INT flexop_vec_get_size(FLEXOP_VEC *v);
```

flexop_vec_int_get_value gets the n-th member.

```
FLEXOP_INT flexop_vec_int_get_value(FLEXOP_VEC *v, FLEXOP_INT n);
```

flexop_vec_uint_get_value gets the n-th member.

```
FLEXOP_UINT flexop_vec_uint_get_value(FLEXOP_VEC *v, FLEXOP_INT n);
```

flexop_vec_float_get_value gets the n-th member.

```
FLEXOP_FLOAT flexop_vec_float_get_value(FLEXOP_VEC *v, FLEXOP_INT n);
```

flexop_vec_string_get_value gets the n-th member.

```
char * flexop_vec_string_get_value(FLEXOP_VEC *v, FLEXOP_INT n);
```

`flexop_vec_print` prints a vector to `stdout`.

```
void flexop_vec_print(FLEXOP_VEC *v);
```

5 Option Management

5.1 Convention

Assuming an application `app` employs the FLEXOP library as option parser, and it defines several options:

- `date` as integer.
- `fpe` as floating point number.
- `name` as string.
- `check` as boolean.
- `solver` as keyword.
- `vi` as vector of integer.
- `vf` as vector of floating point number.
- `vs` as vector of string.

The code sample below shows how to use these options in command line.

```
# integer
app -date 8

# floating point number
app -fpe 1.3

# string
app -name jack

# boolean true
app -check

# boolean false
app +check

# keyword
app -solver cg

# vector of integer
app -vi "1 3 5 7 26"

# vector of floating point number
```

```
app -vf "1.1 1e-3 2.24 0.001"

#vector of strings
app -vs "excuse me do you know where jack is"
```

We can see that all options start with "-" except for boolean type. For boolean type, "-" means true (1), and "+" mean false (0). For vector types, values must be put within double quotes. All these options can be combined as follows,

```
app -date 8 -fpe 1.3 -name jack -check

app -solver cg -vi "1 3 5 7 26"

app -vf "1.1 1e-3 2.24 0.001" -vs "hi do you know where jack is"
```

If an option is repeated in command line, the last one is applied. For example,

```
app -date 8 -date 22
```

The value of 22 is applied to `date`.

5.2 Built-In Options

The FLEXOP library has a few built-in options:

- `help`: this option will print help info and quit. User can call like this

  ```
  app -help
  ```

  User will see section (title, category) information, which is accepted by `help`, such as

  ```
  # print generic (built-in) info
  app -help generic

  # print all user registered info
  app -help user

  # print options belong to section (category) xyz
  app -help xyz
  ```

- `option_file`: a file to store options. User can define options in a file. The file will be parsed after parsing command line options, which means it has higher priority.

  ```
  app -option_file some_file
  ```

5.3 Usage

The following code sample shows basic calling sequences, which have one optional step and three mandatory steps.

```
{
    /* 1: preset values (optional) */
    flexop_preset("-date 23");

    /* 2: register (mandatory) */
    flexop_register_int("date", "int", &i);
    flexop_register_float("fpe", "float", &f);

    flexop_register_vec_int("vi", "vector of int", &vi);
    flexop_register_vec_float("vf", "vector of float", &vf);

    /* 3: parse (mandatory) */
    flexop_init(&argc, &argv);

    /* 4: clean memory (mandatory) */
    flexop_finalize();
}
```

The first step is optional. It provided some pre-defined command line options, which serve as default values. If user provides command line options when running an application, the options provides by command line will ovrride these pre-defined options. User can provide any amounts of pre-defined legal options.

The second step is to register options. User can register any amounts of options, such as 3 and 1 million. The registration must be lie between pre-defined options and `flexop_init`.

The third step is to parse user-provided options in command line by `flexop_init`. After this, the internal variables have proper values.

The forth step is to clean up, which releases all internal memory allocated by FLEXOP, such as vector and string. User is not required to manage internal memory for key, name, string and vector, which is hard if user is not familiar with the source code.

5.4 Registration

`flexop_register_title` registers a title (section), which could be used to divided options into different groups.

```
void flexop_register_title(const char *str, const char *help, const char *category);
```

`flexop_register_bool` registers a boolean variable.

```
void flexop_register_bool(const char *name, const char *help, int *var);

# code
{
    // true, initial value
    int check_email = 1;

    flexop_register_bool("check", "if check emails", &check_email);
}

# usage
    # true, check_email = 1
    app -check

    # false, check_email = 0
    app +check

    # error
    app -check 12

    # error
    app -check hello
```

flexop_register_int registers a integer variable.

```
void flexop_register_int(const char *name, const char *help, FLEXOP_INT *var);

# code
{
    FLEXOP_INT m = 3; /* March */

    flexop_register_int("month", "month of the year", &m);
}

# usage
    # Jan (1)
    app -month 1

    # error
    app -month 3.2

    # error
    app -month hi
```

flexop_register_uint registers an unsigned integer.

```
void flexop_register_uint(const char *name, const char *help, FLEXOP_UINT *var);

# code
{
    FLEXOP_UINT m = 1;

    flexop_register_uint("score", "score of mid-term", &m);
}

# usage
    # 70
    app -score 70

    # error
    app -score 7.1

    # error
    app -score 7.1e-3
```

`flexop_register_float` registers a floating point number.

```
void flexop_register_float(const char *name, const char *help, FLEXOP_FLOAT *var);

# code
{
    FLEXOP_FLOAT f = 1.3;

    flexop_register_float("rating", "rating of Wendy's", &f);
}

# usage
    # 4.5
    app -rating 4.5

    # 4.3
    app -rating 0.43e+1

    # 1e-3
    app -rating 1e-3

    # error
    app -rating hello
```

`flexop_register_string` registers a string (`char *`). In this function, space is assume to be separator, unless a string is put within double quotes, such as "Jack Smith".

```
void flexop_register_string(const char *name, const char *help, char **var);
```

```
# code
{
    char *s = "Jack";

    flexop_register_string("name", "name of VIP", &s);
}

# usage
    # Jenny
    app -name Jenny
    app -name "Jenny"

    # Jack Smith
    app -name "Jack Smith"

    # JS II
    app -name "JS II"
```

`flexop_register_keyword` registers a keyword and an integer variable. When being used in command line, only registered keywords are allowed, and the keyword array must end with `NULL`. The value of the integer is the position of the option in the keyword array, which follows C language style that starts from 0.

```
void flexop_register_keyword(const char *name, const char *help,
    const char **keys, int *var);

# code
{
    const char *keys[] = {"one", "two", "three", "four", NULL};
    int order = 0;

    flexop_register_keyword("o", "order of digital number", keys, &order);
}

# usage
    # order = 0
    app -o one

    # order = 2
    app -o three

    # error
    app -o five

    # error
    app -o hi
```

flexop_register_handler registers a handle function, func, and a variable, hvar, when the option name is parsed, the FLEXOP library will call the function, which will save parsed values to variable provided by user, hvar. When the variable is NULL, the function will print help information, such as how-to, and user is responsible for releasing all memories allocated by the handler. The handler will return true (non-zero value) if succeed, otherwise 0 is returned.

```
void flexop_register_handler(const char *name, const char *help,
    FLEXOP_HANDLER func, void *hvar);

# code
int string_to_float(FLEXOP_KEY *o, const char *arg)
{
    FLEXOP_FLOAT *t = o->hvar;

    assert(o != NULL);

    if (arg == NULL) {
        flexop_printf("usage: -s floating-point-number, such as \"-s 2.3\"\n");
        return 0;
    }

    *t = flexop_atof(arg);

    return 1;
}

{
    FLEXOP_FLOAT stof = 2.1;

    flexop_register_handler("s", "string to floating point number",
        string_to_float, &stof);
}

# usage
    # stof = 3.0
    app -s 3

    # stof = 1e-3
    app -s 1e-3

    # error, not floating point number
    app -s sam
```

flexop_register_vec_int registers a integer vector. The option must be lie within double

quotes if more than one member and separator by space.

```
void flexop_register_vec_int(const char *name, const char *help, FLEXOP_VEC *var);

# code
{
    FLEXOP_VEC vi;

    flexop_register_vec_int("v", "vector of int", &vi);
}

# usage
    # v = {1, 3, 5, 8, 9}
    app -v "1 3 5 8 9"

    # v = 1
    app -v 1

    # error, contain floating point number
    app -v "1 1.3"
```

flexop_register_vec_uint is similar to flexop_register_vec_int, and it registers unsigned integer.

```
void flexop_register_vec_uint(const char *name, const char *help, FLEXOP_VEC *var);
```

flexop_register_vec_float is similar to flexop_register_vec_int, and it registers floating point number.

```
void flexop_register_vec_float(const char *name, const char *help, FLEXOP_VEC *var);
```

flexop_register_vec_string registers string vector, which is separated by space.

```
void flexop_register_vec_string(const char *name, const char *help, FLEXOP_VEC *var);

# code
{
    FLEXOP_VEC vi;

    flexop_register_vec_string("v", "vector of string", &vi);
}

# usage
    # v = {"one", "two", "hello"}
    app -v "one two hello"

    # v = {"1", "2", "hi"}
    app -v "1 2 hi"
```

5.5 Setting Values

After the options have been parsed, the following setting functions can be used to change their values. However, they are optional functions and can only be used after `flexop_init`. Also, the option should match their registered type.

```
void flexop_set_options(const char *str);
```

```
int flexop_set_bool(const char *op_name, int value);
```

```
int flexop_set_int(const char *op_name, FLEXOP_INT value);
```

```
int flexop_set_uint(const char *op_name, FLEXOP_UINT value);
```

```
int flexop_set_float(const char *op_name, FLEXOP_FLOAT value);
```

```
int flexop_set_keyword(const char *op_name, const char *value);
```

```
int flexop_set_string(const char *op_name, const char *value);
```

```
int flexop_set_handler(const char *op_name, const char *value);
```

```
int flexop_set_vec_int(const char *op_name, const char *value);
```

```
int flexop_set_vec_uint(const char *op_name, const char *value);
```

```
int flexop_set_vec_float(const char *op_name, const char *value);
```

```
int flexop_set_vec_string(const char *op_name, const char *value);
```

5.6 Getting Values

The following functions can be used to get parsed values, and the option name should match their registered type.

```
int flexop_get_bool(const char *op_name);
```

```
FLEXOP_INT flexop_get_int(const char *op_name);
```

```
FLEXOP_UINT flexop_get_uint(const char *op_name);
```

```
FLEXOP_FLOAT flexop_get_float(const char *op_name);
```

```
const char * flexop_get_keyword(const char *op_name);
```

```
const char * flexop_get_string(const char *op_name);
```

```
FLEXOP_VEC * flexop_get_vec_int(const char *op_name);
```

```
FLEXOP_VEC * flexop_get_vec_uint(const char *op_name);
```

```
FLEXOP_VEC * flexop_get_vec_float(const char *op_name);
```

```
FLEXOP_VEC * flexop_get_vec_string(const char *op_name);
```

5.7 Auxiliary Functions

flexop_preset_cmdline sets pre-defined options, which can be used to set default command line options. The pre-defined options will be overridden if the same options are provided by command line or option file.

```
void flexop_preset_cmdline(const char *str);

# code
{
    flexop_preset_cmdline("-i 23");
    flexop_preset_cmdline("-month 2");
    flexop_preset_cmdline("-s 2.1");
}
```

flexop_init parses options.

```
void flexop_init(int *argc, char ***argv);
```

flexop_finalize cleans up internal memory and related work.

```
void flexop_finalize(void);
```

`flexop_show_cmdline` shows user-provided command line options and preset options, which should be used after `flexop_init`.

```
void flexop_show_cmdline(void);
```

`flexop_show_used` shows all parsed values, which should be used after `flexop_init`.

```
void flexop_show_used(void);
```