GNU Gengetopt

A command line option parser generator
for GNU Gengetopt version 2.20
updated on June 2007

Lorenzo Bettini

This manual is for GNU Gengetopt (version 2.20, 30 June 2007), a tool to write command line option parsers for C programs.

Copyright © 2001 - 2007 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover Texts being "A GNU Manual," and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled "GNU Free Documentation License."

(a) The FSF's Back-Cover Text is: "You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development."

Table of Contents

Audience

Gengetopt is a tool to generate C code to parse the command line arguments `argc` and `argv` that are part of every C or C++ program. The generated code uses the C library function `getopt_long` to perform the actual command line parsing.

This manual is written for C and C++ programmers, specifically the *lazy* ones. If you've written any non-trivial C program, you've had to deal with argument parsing. It isn't particularly difficult, nor is it particularly exciting. It *is* one of the classic programming nuisances, which is why most books on programming leave it as an exercise for the reader. Gengetopt can save you from this work, leaving you free to focus on the interesting parts of your program.

Thus your program will be able to handle command line options such as:

```
myprog --input foo.c -o foo.o --no-tabs -i 100 *.class
```

And both long options (those that start with `--`) and short options (start with `-` and consist of only one character) can be handled (see Chapter 4 [Terminology], page 21 for further details). For standards about short and long options you may want to take a look at the GNU Coding Standards (`http://www.gnu.org/prep/standards_toc.html`).

Gengetopt can also generate a function to save the command line options into a file (see Chapter 2 [Basic Usage], page 6), and a function to read the command line options from a file (see Chapter 6 [Configuration files], page 24). Of course, these two kinds of files are compliant.

Gengetopt Copying Conditions

Gengetopt is free software; you are free to use, share and modify it under the terms of the GNU General Public License that accompanies this manual.

The code that Gengetopt generates is also free software; however it is licensed with a simple all-permissive license instead of the GPL or LGPL. You are free to do anything you like with the generated code, including incorporating it into or linking it with proprietary software.

Gengetopt was originally written by Roberto Arturo Tena Sanchez. It is currently maintained by Lorenzo Bettini http://www.lorenzobettini.it.

A primordial version of Chapter 4 [Terminology], page 21 was written by Adam Greenblatt.

1 Installation

See the file 'INSTALL' for detailed building and installation instructions; anyway if you're used to compiling Linux software that comes with sources you may simply follow the usual procedure, i.e. untar the file you downloaded in a directory and then:

```
cd <source code main directory>
./configure
make
make install
```

Note: unless you specify a different install directory by `--prefix` option of configure (e.g. `./configure --prefix=<your home>`), you must be root to run `make install`.

Files will be installed in the following directories:

```
executables
          /prefix/bin

docs      /prefix/share/doc/gengetopt

examples  /prefix/share/doc/gengetopt/examples

additional files
          /prefix/share/gengetopt
```

Default value for prefix is `/usr/local` but you may change it with `--prefix` option to configure.

1.1 Download

You can download it from GNU's ftp site: ftp://ftp.gnu.org/gnu/gengetopt or from one of its mirrors (see http://www.gnu.org/prep/ftp.html).

I do not distribute Windows binaries anymore; since, they can be easily built by using Cygnus C/C++ compiler, available at http://www.cygwin.com. However, if you don't feel like downloading such compiler, you can request such binaries directly to me, by e-mail (find my e-mail at my home page) and I can send them to you.

Archives are digitally signed by me (Lorenzo Bettini) with GNU gpg (http://www.gnupg.org). My GPG public key can be found at my home page (http://www.lorenzobettini.it).

You can also get the patches, if they are available for a particular release (see below for patching from a previous version).

1.2 Anonymous CVS Access

This project's CVS repository can be checked out through anonymous (pserver) CVS with the following instruction:

```
cvs -z3 -d:pserver:anonymous@cvs.savannah.gnu.org:/sources/gengetopt co gengetopt
```

Further instructions can be found at the address:

http://savannah.gnu.org/projects/gengetopt.

Please notice that this way you will get the latest development sources of Gengetopt, which may also be unstable. This solution is the best if you intend to correct/extend this program: you should send me patches against the latest cvs repository sources.

If, on the contrary, you want to get the sources of a given release, through cvs, say, e.g., version X.Y.Z, you must specify the tag `rel_X_Y_Z` when you run the cvs command or the `cvs update` command.

NOTICE: This convention holds since release 2.14.

When you compile the sources that you get through the cvs repository, before running the `configure` and `make` commands, you should, at least the first time, run the command:

```
sh autogen.sh
```

This will bootstrap the autotools in the correct order, and also copy possibly missing files. You should have installed recent versions of `automake`, `autoconf` and `libtool` in order for this to succeed. You will also need `flex` and `bison`.

1.3 What you need to build gengetopt

Gengetopt has been developed under GNU/Linux, using gcc (C++), and bison (yacc) and flex (lex), and ported under Win32 with Cygnus C/C++compiler, available at http://www.cygwin.com.

For developing gengetopt, I use the excellent GNU Autoconf[1], GNU Automake[2] and GNU Libtool[3]. Since version 2.19 I also started to use Gnulib - The GNU Portability Library[4], "a central location for common GNU code, intended to be shared among GNU packages" (for instance, I rely on Gnulib for checking for the presence and correctness of `getopt_long` function, Section 9.2.1 [Use Gnulib], page 34).

Moreover *GNU Gengen* (http://www.gnu.org/software/gengen) is used for automatically generating the code that generates the command line parser.

Actually, you don't need all these tools above to build gengetopt because I provide generated sources, unless you want to develop gengetopt.

The code generated by gengetopt relies on the `getopt_long` function that is usually in the standard C library; however, there may be some implementations of the C library that don't include it; we refer to Chapter 9 [No getopt_long], page 34, for instructions on how to check whether getopt_long is part of the library and how to deal with their lacking (using autoconf and automake).

1.4 Patching from a previous version

If you downloaded a patch, say '`gengetopt-1.3-1.3.1-patch.gz`' (i.e., the patch to go from version 1.3 to version 1.3.1), cd to the directory with sources from the previous version (gengetopt-1.3) and type:

```
gunzip -cd ../gengetopt-1.3-1.3.1.patch.gz | patch -p1
```

[1] http://www.gnu.org/software/autoconf
[2] http://www.gnu.org/software/automake
[3] http://www.gnu.org/software/libtool
[4] http://www.gnu.org/software/gnulib

and restart the compilation process (if you had already run configure a simple make should do).

2 Basic Usage

The command line options, which have to be handled by gengetopt generated function, are specified in a file (typically with '`.ggo`' extension). This file consist in lines of sentences with the formats shown below. Commands in `{}` are optional (the `option` sentences need not to be given in separate lines):

```
package "<packname>"
version "<version>"
purpose "<purpose>"
usage "<usage>"
description "<description>"

args "<command line options>"

option <long> <short> "<desc>"
    {argtype} {typestr="<type descr>"}
    {values="<value1>","<value2>",...}
    {default="<default value>"}
    {dependon="<other option>"}
    {required} {argoptional} {multiple}
    {hidden}

option <long> <short> <desc> flag <on/off>

section "section name" {sectiondesc="optional section description"}

text "a textual sentence"
```

Where:

`package`
: This has the precedence over `PACKAGE` generated by autoconf. Optional.

`version`
: This has the precedence over `VERSION` generated by autoconf. Optional.

`purpose`
: What the program does (even on more than one line), it will be printed with the help, before the usage string. Optional.

`usage`
: The "Usage" string that will be printed with the help[1]. If not specified, it will be generated automatically. Optional.

`description`
: The "Description" string that will be printed with the help[2], after the usage string. Optional.

[1] Since version 2.19.

[2] Since version 2.19.

`args`

With `args`[3] you can specify options that will be added to the command line options of gengetopt itself. For instance, if you always run gengetopt on your input file with the options `--no-handle-error --string-parser -u`, you can add these options in the input file like this:

```
args "--no-handle-error --string-parser -u"
```

and remove those recurrent options from the command line. Optional.

`long`

The long option, a double quoted string with upper and lower case characters, digits, a dash (`-`) and a dot (`.`). No spaces allowed. The name of the variables generated to store arguments (see later in this section) are long options converted to be legal C variable names. This means that `.` and `-` are both replaced by `_`.

`short`

The short option, a single upper or lower case char, or a digit. If a `-` is specified, then no short option is considered for the long option (thus long options with no associated short options are allowed).

`desc`

This description will be printed during the output of `--help`. Wrapping will be automatically performed.

`argtype`

`string`, `int`, `short`, `long`, `float`, `double`, `longdouble` or `longlong`. If no type is specified the option does not accept an argument.

`typestr`

a description for the type of the values for the option. This will be used during the output of `--help` (e.g., `"filename"` instead of simply `STRING`, or `"portnumber"` instead of simply `INT`).

`values`

a list of strings containing all the possible values that can be passed to the option. The type is considered string and must not be specified.

`default`

an optional default value for the option. The value must always be specified as a double quoted string.

`dependon`

this option depends on another option (whose long name description is specified). If this option is given at command line but not the option it depends on, an error will be generated.

`required`

[3] Since version 2.18

`required` or `optional`. This specifies whether such option must be given at each program invocation. These keywords were introduced in release 2.17. Before, you had to use the keywords `yes` or `no`. You can still use these keywords but their use is not advised since they are not much explicative.

If not specified, an option is considered mandatory; if you do not want this behavior, you can require that by default options are considered optional, by using the command line option `--default-optional`[4].

`argoptional`

If this flag is specified then this option has an argument that is optional. In this case, when you specify the argument at command line, please use `=` in case you use a long option, and avoid spaces if you use short option. For instance, if the option with optional argument is `-B|--bar`, use the following command line syntax: `-B15` or `--bar=15`, and NOT the following one `-B 15` nor `--bar 15`.

By using this specification together with `default` you can obtain an option that even if not explicitly specified will have the default value, and if it is specified without an argument it will have, again, the default value.

`multiple`

If this flag is specified then this option can be specified more than once at command line; all the values for this option are stored in an array. You can also specify the number of occurrences that a multiple option must be specified. Notice that this is independent from the `required` flag. See Chapter 7 [Multiple Options], page 28.

`hidden`

If an option is "hidden" it will not appear in the output of `--help` but it can still be specified at command line[5]. In case hidden options are used, the command line option `--full-help` will also be generated. This will print also the hidden options[6].

`on/off`

`on` or `off`. This is the state of the flag when the program starts. If user specifies the option, the flag toggles.

For strings (delimited by `"`) the following convention is adopted[7]: a string spanning more than one line will be interpreted with the corresponding line breaks; if the line break is not desired one can use the backslash `\` to break the line without inserting a line break. A line break in a string can also be inserted with the string `\n`. Here are some examples:

```
"This string will be interpreted
into two lines exactly as it is"
```

[4] Since version 2.20.

[5] Since version 2.15.

[6] Since version 2.16.

[7] This is true since version 2.19. Before this version, strings were not allowed to spawn more than one line.

```
"This string is specified with two lines \
but interpreted as specified in only one line \
i.e., without explicit line break"

"This string\nwill have a line break"
```

Moreover, if the character " must be specified in the string, it will have to be escaped with the backslash[8]:

```
"This string contains \"a quoted string\" inside"
```

The part that must be provided in the correct order is

```
option <long> <short> "<desc>"
```

while the other specifications can be given in any order[9]. Thus, for instance

```
option <long> <short> "<desc>" {argtype} {typestr="<type descr>"}
```

is the same as

```
option <long> <short> "<desc>" {typestr="<type descr>"} {argtype}
```

Comments begin with # in any place (but in strings) of the line and ends in the end of line.

Notice that the options `-h,--help` and `-V,--version` are added automatically; however, if you specify an option yourself that has `h` as short form or `help` as long form, then `-h,--help` is not added (and you have to handle the help option manually). The same holds for `-V,--version`.

In case hidden options are used, See [Hidden options], page 8, the command line option `--full-help` will also be generated. This will print also the hidden options[10].

Options can be part of sections, that provide a more meaningful descriptions of the options. A *section* can be defined with the following syntax (the `sectiondesc` is optional) and all the options following a section declaration are considered part of that sections:

```
section "section name" {sectiondesc="optional section description"}
```

Notice that the separation in sections is stronger than separation in groups of mutual exclusive options (see Chapter 5 [Group options], page 23). Furthermore, sections should not be inserted among group options (but only externally). A section makes sense only if it is followed by some options. If you don't specify any option after a section, that section will not be printed at all. If you need to simply insert some text in the output of `--help`, then you must use `text`, explained in the next paragraph.

You can insert, among options, a textual string that will be printed in the output of `--help`[11]:

```
text "\nA text description with possible line\nbreaks"
```

Of course, you can use this mechanism even to manually insert blank lines among options with an empty text string:

---

[8] Since version 2.19.

[9] This holds since version 2.15: in previous versions the option specifications had to be given in a fixed order.

[10] Since version 2.16.

[11] Since version 2.18.

```
text ""
```

You can also specify the list of `values` that can be passed to an option (in that case the option has type `string`). If a value that is not in the list is passed, an error is raised. You can think of such options as *enumerated* options. It is not necessary to pass the complete value at the command line option: a non ambiguous prefix will do. For instance, if the accepted values are `"foo","bar","foobar"`, then you can pass at the command line the value `"b"` and the value `"bar"` will be selected, or the value `"foob"` and the value `"foobar"` will be selected; instead, passing the value `"fo"` will raise an ambiguity error.

Here's an example of such a file (the file is called '`sample1.ggo`')

```
# Name of your program
package "sample1" # don't use package if you're using automake
# Version of your program
version "2.0"   # don't use version if you're using automake

# Options
option  "str-opt"         s "A string option, for a filename"
        string typestr="filename"     optional
text "\nA brief text description"
text " before the other options.\n"
option  "my-opt"          m "Another integer option, \
this time the description of the option should be \"quite\" long to \
require wrapping... possibly more than one wrapping :-) \
especially if I\nrequire a line break"      int     optional
option  "int-opt"         i "A int option"         int        yes
section "more involved options"
        sectiondesc="the following options\nare more complex"
text ""
option  "flag-opt"        - "A flag option"        flag       off
option  "funct-opt"       F "A function option"    optional
section "last option section"
option  "long-opt"        - "A long option"        long       optional
option  "def-opt"         - "A string option with default"
        string default="Hello" optional
option  "enum-opt"         - "A string option with list of values"
        values="foo","bar","hello","bye" default="hello" optional
option  "secret"         S "hidden option will not appear in --help"
        int optional hidden
option  "dependant"         D
        "option that depends on str-opt" int optional dependon="str-opt"
text "\nAn ending text."
```

The simplest way to use gengetopt is to pass this file as the standard input, i.e.:

```
gengetopt < sample1.ggo
```

By default gengetopt generates '`cmdline.h`' and '`cmdline.c`'. Otherwise we can specify these names with a command line option:

```
gengetopt < sample1.ggo --file-name=cmdline1 --unamed-opts
```

The option `--unamed-opts` allows the generated command line parser to accept also names, without an option (for instance you can pass a file name without an option in front of it, and also use wildcards, such as `*.c`, `foo*.?` and so on). These are also called *parameters* (see Chapter 4 [Terminology], page 21). You can specify an optional description for these additional names (default is `FILES`).

In '`cmdline1.h`' you'll find the generated C struct:

```
/* cmdline1.h */

/* File autogenerated by gengetopt version 2.20  */

#ifndef CMDLINE1_H
#define CMDLINE1_H

/* If we use autoconf.  */
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#ifndef CMDLINE_PARSER_PACKAGE
#define CMDLINE_PARSER_PACKAGE "sample1"
#endif

#ifndef CMDLINE_PARSER_VERSION
#define CMDLINE_PARSER_VERSION "2.0"
#endif

struct gengetopt_args_info
{
  const char *help_help; /* Print help and exit help description.  */
  const char *full_help_help; /* Print help, including hidden options, and exit help description.  */
  const char *version_help; /* Print version and exit help description.  */
  char * str_opt_arg;        /* A string option, for a filename.  */
  char * str_opt_orig;        /* A string option, for a filename original value given at command line.  */
  const char *str_opt_help; /* A string option, for a filename help description.  */
  int my_opt_arg;        /* Another integer option, this time the description of the option should be \"quite\" long to require wrapping... possibly more than one wrapping :-) especially if \nrequire a line break.  */
  char * my_opt_orig;        /* Another integer option, this time the description of the option should be \"quite\" long to quire wrapping... possibly more than one wrapping :-) especially if \nrequire a line break original value given at command line.  */
  const char *my_opt_help; /* Another integer option, this time the description of the option should be \"quite\" long to require wrapping... possibly more than one wrapping :-) especially if \nrequire a line break help description.  */
  int int_opt_arg;        /* A int option.  */
  char * int_opt_orig;        /* A int option original value given at command line.  */
  const char *int_opt_help; /* A int option help description.  */
  int flag_opt_flag;        /* A flag option (default=off).  */
  const char *flag_opt_help; /* A flag option help description.  */
  const char *funct_opt_help; /* A function option help description.  */
  long long_opt_arg;        /* A long option.  */
  char * long_opt_orig;        /* A long option original value given at command line.  */
  const char *long_opt_help; /* A long option help description.  */
  char * def_opt_arg;        /* A string option with default (default='Hello').  */
  char * def_opt_orig;        /* A string option with default original value given at command line.  */
  const char *def_opt_help; /* A string option with default help description.  */
  char * enum_opt_arg;        /* A string option with list of values (default='hello').  */
  char * enum_opt_orig;        /* A string option with list of values original value given at command line.  */
  const char *enum_opt_help; /* A string option with list of values help description.  */
  int secret_arg;        /* hidden option will not appear in --help.  */
```

```
  char * secret_orig;        /* hidden option will not appear in --help original value given at command line.  */
  const char *secret_help; /* hidden option will not appear in --help help description.  */
  int dependant_arg;        /* option that depends on str-opt.  */
  char * dependant_orig;        /* option that depends on str-opt original value given at command line.  */
  const char *dependant_help; /* option that depends on str-opt help description.  */

  int help_given ;        /* Whether help was given.  */
  int full_help_given ;        /* Whether full-help was given.  */
  int version_given ;        /* Whether version was given.  */
  int str_opt_given ;        /* Whether str-opt was given.  */
  int my_opt_given ;        /* Whether my-opt was given.  */
  int int_opt_given ;        /* Whether int-opt was given.  */
  int flag_opt_given ;        /* Whether flag-opt was given.  */
  int funct_opt_given ;        /* Whether funct-opt was given.  */
  int long_opt_given ;        /* Whether long-opt was given.  */
  int def_opt_given ;        /* Whether def-opt was given.  */
  int enum_opt_given ;        /* Whether enum-opt was given.  */
  int secret_given ;        /* Whether secret was given.  */
  int dependant_given ;        /* Whether dependant was given.  */

  char **inputs ; /* unamed options */
  unsigned inputs_num ; /* unamed options number */
} ;

extern const char *gengetopt_args_info_purpose;
extern const char *gengetopt_args_info_usage;
extern const char *gengetopt_args_info_help[];
extern const char *gengetopt_args_info_full_help[];

int cmdline_parser (int argc, char * const *argv,
  struct gengetopt_args_info *args_info);
int cmdline_parser2 (int argc, char * const *argv,
  struct gengetopt_args_info *args_info,
  int override, int initialize, int check_required);
int cmdline_parser_file_save(const char *filename,
  struct gengetopt_args_info *args_info);

void cmdline_parser_print_help(void);
void cmdline_parser_print_full_help(void);
void cmdline_parser_print_version(void);

void cmdline_parser_init (struct gengetopt_args_info *args_info);
void cmdline_parser_free (struct gengetopt_args_info *args_info);

int cmdline_parser_required (struct gengetopt_args_info *args_info,
  const char *prog_name);

extern char *cmdline_parser_enum_opt_values[] ;        /* Possible values for enum-opt.  */


#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif /* CMDLINE1_H */
```

The `<option>_given` field is set to a value different from 0 when an argument for `<option>` has been specified. If the option accepts an argument and it is not of `flag` type The `<option>_arg` field is set to the value passed at the command line. The `<option>_arg` field has the corresponding C type specified in the file passed to gengetopt.

Notice that if an option has a default value, then the corresponding `<option>_arg` will be initialized with that value but the corresponding `<option>_given` will NOT be initialized to 1. Thus, `<option>_given` will effectively inform you if the user has specified that command line option.

The additional field `<option>_orig` is always a string containing the original value passed at the command line. This may be different, for instance, in case of numerical arguments: gengetopt converts the passed value (a string) into the corresponding numerical type; due to conversions, float representations, etc., this may not correspond exactly to the original value passed at command line. It can also be different when enumerated options are used (see above): in particular the `<option>_arg` field will contain a value taken from the specified list, while `<option>_orig` contains the (non-ambiguous) prefix specified at the command line.

The user can always access this original value by using `<option>_orig` instead of `<option>_arg`, as he sees fit[12]. For instance, gengetopt itself uses the original value when it saves the command line options into a file (see the `_file_save` function in the following). However, apart from very specific command line processing, the user might hardly need the `<option>_orig` field, and can be always safely use `<option>_arg`.

The `<option>_help` contains the string (concerning this very option) that is printed when `--help` command line is given.

If it is of `flag` type, only the field `<option>_flag` is generated.

The strings `cmdline_parser_purpose` and `cmdline_parser_usage` contain the `purpose` as specified in the input file and the generated "usage" string as printed when `--help` command line is given. Finally, the string array `cmdline_parser_help` contains the strings (one for each option) printed when `--help` command line is given (this array is terminated by a null string element). If hidden options are used also the `cmdline_parser_full_help` array is available (containing also help strings concerning hidden options). All these strings can be used by the programmer to build a customized help output[13].

Even if `<option>_given` is 0, the corresponding `<option>_arg` is set to default value (if one has been specified for `<option>`). However, in this case, the `<option>_orig` is set to `NULL`.

Notice that by default the generated function is called `cmdline_parser` (see the command line options below, to override this name), and it takes the arguments that main receives and a pointer to such a struct, that it will be filled. Another version, `cmdline_parser2`, can be specified more arguments. Since you typically need this second version only in conjunction with other "kinds" of parsers such as configuration files and multiple parsers, you can find more details about it in Chapter 6 [Configuration files], page 24.

IMPORTANT: The array passed to the parser function (that in turn is passed to `getopt_long` is expected to have in the first element (of index 0) the name of the program that was invoked. This will be used, for instance, for printing possible errors.

[12] The `<option>_orig` was introduced in the release 2.14.

[13] These strings and the `<option>_help` were introduced in the release 2.17.

`cmdline_parser_free` can be called to deallocate memory allocated by the parser for string and multiple options.

`cmdline_parser_init` can be called to initialize the struct (it is not mandatory, since it is done automatically by the command line parser).

`cmdline_parser_file_save`[14] can be used to save the command line options into a file. The contents of this file are consistent with the configuration files (Chapter 6 [Configuration files], page 24). Notice that if an option has a default value, this option will be saved into the file only if it was passed explicitly at command line (or read from a configuration file), i.e., default values will not be saved into the file.

And here's how these functions can be used inside the main program:

```
/* main1.cc */
/* we try to use gengetopt generated file in a C++ program */
/* we don't use autoconf and automake vars */

#include <iostream>
#include "stdlib.h"

#include "cmdline1.h"

using std::cout;
using std::endl;

int
main (int argc, char **argv)
{
  gengetopt_args_info args_info;

  cout << "This one is from a C++ program" << endl ;
  cout << "Try to launch me with some options" << endl ;
  cout << "(type sample1 --help for the complete list)" << endl ;
  cout << "For example: ./sample1 *.* --funct-opt" << endl ;

  /* let's call our cmdline parser */
  if (cmdline_parser (argc, argv, &args_info) != 0)
    exit(1) ;

  cout << "Here are the options you passed..." << endl;

  for ( unsigned i = 0 ; i < args_info.inputs_num ; ++i )
    cout << "file: " << args_info.inputs[i] << endl ;

  if ( args_info.funct_opt_given )
    cout << "You chose --funct-opt or -F." << endl ;

  if ( args_info.str_opt_given )
    cout << "You inserted " << args_info.str_opt_arg << " for " <<
      "--str-opt option." << endl ;

  if ( args_info.int_opt_given )
    cout << "This is the integer you input: " <<
      args_info.int_opt_arg << "." << endl;

  if (args_info.flag_opt_given)
```

[14] This function was introduced in the release 2.14.

```
    cout << "The flag option was given!" << endl;

  cout << "The flag is " << ( args_info.flag_opt_flag ? "on" : "off" ) <<
    "." << endl ;

  if (args_info.enum_opt_given) {
    cout << "enum-opt value: " << args_info.enum_opt_arg << endl;
    cout << "enum-opt (original specified) value: "
        << args_info.enum_opt_orig << endl;
  }

  if (args_info.secret_given)
    cout << "Secret option was specified: " << args_info.secret_arg
        << endl;

  cout << args_info.def_opt_arg << "! ";

  cout << "Have a nice day! :-)" << endl ;

  cmdline_parser_free (&args_info); /* release allocated memory */

  return 0;
}
```

Now you can compile '`main1.cc`' and the '`cmdline1.c`' generated by gengetopt and link all together to obtain `sample1` executable:

```
gcc -c cmdline1.c
g++ -c main1.cc
g++ -o sample1 cmdline1.o main1.o
```

(Here we assume that `getopt_long` is included in the standard C library; see Chapter 1 [Installation], page 3 and Chapter 9 [No getopt_long], page 34).

Now let's try some tests with this program:

```
$ ./sample1 -s "hello" --int-opt 1234
This one is from a C++ program
Try to launch me with some options
(type sample1 --help for the complete list)
For example: ./sample1 *.* --funct-opt
Here are the options you passed...
You inserted hello for --str-opt option.
This is the integer you input: 1234.
The flag is off.
Have a nice day! :-)
```

You can also pass many file names to the command line (this also shows how flags work):

```
$ ./sample1 *.h -i -100 -x
This one is from a C++ program
Try to launch me with some options
(type sample1 --help for the complete list)
For example: ./sample1 *.* --funct-opt
Here are the options you passed...
file: cmdline1.h
```

```
file: cmdline2.h
file: cmdline.h
file: getopt.h
This is the integer you input: -100.
The flag is on.
Have a nice day! :-)
```

And if we try to omit the `--int-opt` (or `-i`), which is required, we get an error:

```
$ ./sample1
This one is from a C++ program
Try to launch me with some options
(type sample1 --help for the complete list)
For example: ./sample1 *.* --funct-opt
sample1: ‘--int-opt’ (‘-i’) option required!
```

Now, let's test the enumerated options, notice the use of a prefix for specifying an acceptable value, and the difference between the actual passed value and the one recorded in `<option>_arg`:

```
$ ./sample1 -i 10 --enum-opt h
...
enum-opt value: hello
enum-opt (original specified) value: h
...
```

While the next one raises an ambiguity error (between `"bar"` and `"bye"`):

```
$ ./sample1 -i 10 --enum-opt b
...
./sample1: ambiguous argument, "b", for option ‘--enum-opt’
```

Here is the output of `--help` of the parser generated from ‘`sample1.ggo`’ by specifying the following options to gengetopt: `--long-help -u --show-required` (see Chapter 3 [Invoking gengetopt], page 18 for further explanation for these command line options).

```
This one is from a C++ program
Try to launch me with some options
(type sample1 --help for the complete list)
For example: ./sample1 *.* --funct-opt
sample1 2.0

Usage: sample1 -iINT|--int-opt=INT [-h|--help] [--full-help] [-V|--version]
         [-sfilename|--str-opt=filename] [-mINT|--my-opt=INT] [--flag-opt]
         [-F|--funct-opt] [--long-opt=LONG] [--def-opt=STRING]
         [--enum-opt=STRING] [-DINT|--dependant=INT]  [FILES]...

  -h, --help              Print help and exit
      --full-help         Print help, including hidden options, and exit
  -V, --version           Print version and exit
  -s, --str-opt=filename  A string option, for a filename

A brief text description before the other options.
```

```
  -m, --my-opt=INT        Another integer option, this time the description of
                            the option should be "quite" long to require
                            wrapping... possibly more than one wrapping :-)
                            especially if I
                            require a line break
  -i, --int-opt=INT       A int option (mandatory)

more involved options:
  the following options
  are more complex

      --flag-opt          A flag option  (default=off)
  -F, --funct-opt         A function option

last option section:
      --long-opt=LONG     A long option
      --def-opt=STRING    A string option with default  (default=‘Hello’)
      --enum-opt=STRING   A string option with list of values  (possible
                            values="foo", "bar", "hello", "bye"
                            default=‘hello’)
  -D, --dependant=INT     option that depends on str-opt

An ending text.
```

Notice how `filename` is printed instead of `STRING` for the option `--str-opt` (since `typestr` was used in the ‘`sample1.ggo`’ file) and how the description of `--my-opt` is wrapped to 80 columns, and how the `\n` is actually interpreted as a newline request. Also the usage string is wrapped. Moreover, since `-S,--secret` is an hidden option (See [Hidden options], page 8.) it is not printed; if you wanted that to be printed, you should use `--full-help`.

Finally, notice how the `text` strings are printed in the help output (and the empty line after the “more involved options” section achieved with an empty `text` string).

If you’re curious you may want to take a look at the generated C file ‘`cmdline1.c`’.

You may find other examples in ‘`/prefix/share/doc/gengetopt/examples`’ or in the ‘`tests`’ of the source tarbal.

3 Invoking gengetopt

This is the output of `gengetopt --help`:

```
gengetopt

This program generates a C function that uses getopt_long function
to parse the command line options, validate them and fill a struct.

Usage: gengetopt [OPTIONS]...

  -h, --help                     Print help and exit
  -V, --version                  Print version and exit

Main options:

  -i, --input=filename           input file (default std input)
  -f, --func-name=name           name of generated function
                                   (default=‘cmdline_parser’)
  -a, --arg-struct-name=name     name of generated args info struct
                                   (default=‘gengetopt_args_info’)
  -F, --file-name=name           name of generated file  (default=‘cmdline’)
      --output-dir=path          output directory
  -c, --c-extension=ext          extension of c file  (default=‘c’)
  -H, --header-extension=ext     extension of header file  (default=‘h’)
  -l, --long-help                long usage line in help
      --default-optional         by default, an option is considered optional if
                                   not specified otherwise
  -u, --unamed-opts[=STRING]     accept options without names (e.g., file names)
                                    (default=‘FILES’)

The parser generated is thought to be used to parse the command line arguments.
However, you can also generate parsers for configuration files, or strings that
contain the arguments to parse, by using the following two options.

  -C, --conf-parser              generate a config file parser
  -S, --string-parser            generate a string parser (the string contains
                                   the command line)

Additional options:

  -G, --include-getopt           adds the code for getopt_long in the generated
                                   C file
  -n, --no-handle-help           do not handle --help|-h automatically
  -N, --no-handle-version        do not handle --version|-V automatically
  -e, --no-handle-error          do not exit on errors
      --show-required[=STRING]   in the output of help will specify which
                                   options are mandatory, by using the optional
                                   passed string  (default=‘(mandatory)’)
  -g, --gen-version              put gengetopt version in the generated file
                                   (default=on)
      --set-package=STRING       set the package name (override package defined
                                   in the .ggo file)
      --set-version=STRING       set the version number (override version
                                   defined in the .ggo file)
      --show-help                show the output of --help instead of generating
                                   code
      --show-full-help           show the output of --help (including hidden
                                   options) instead of generating code
```

```
      --show-version            show the output of --version instead of
                                  generating code

  Please refer to the info manual for further explanations.
```

The options should be clear; in particular:

`--func-name`
: if no `--func-name` is given, `cmdline_parser` is taken by default;

`--output-dir`
: if no `--output-dir`[1] is given, the files are generated in the current directory;

`--arg-struct-name`
: allows to specify the name of the generated struct for command line arguments (default is `gengetopt_args_info`)

`--long-help`
: the "Usage" line reports all the options; this may be unpleasant if options are many;

`--default-optional`
: If this command line option is given, by default, options are considered optional (if not explicitly specified otherwise). Otherwise, options are considered mandatory (if not explicitly specified otherwise).

`--unamed-opts`
: the program will accept also options without a name, which, in most case, means that we can pass many file names to the program (see the example in Chapter 2 [Basic Usage], page 6, where we call `sample1 *.h`). You can specify an optional description for these additional names (default is `FILES`).

`--no-handle-help`
`--no-handle-version`
: if `--no-handle-help` (`--no-handle-version`) is given the command line option `--help|-h` (`--version|-V`) is not handled automatically, so the programmer will be able to print some other information; then the function for printing the standard help (version) response can be used; this function is called `<parser-name>_print_help` (`<parser-name>_print_version`), where `<parser-name>` is the name specified with `--func-name` or the default, `cmdline_parser`.

`--no-handle-error`
: if `--no-handle-error` is given, an error in the parsing does not provoke the exit of the program; instead, since the parser function, in case of an error, returns a value different 0, the program can print a help message, as gengetopt itself does in case of an error (try it!).

`--show-required`
: if `--show-required` is given, possibly with a string, in the output of `--help` will be made explicit which options are actually required, See [Basic Usage], page 16.

[1] Since version 2.17.

`--gen-version`
: is a flag (default on) that when disabled does not put in the output file the gengetopt version (it is useful for testing purposes).

`--conf-parser`
: Detailed in Chapter 6 [Configuration files], page 24.

`--string-parser`
: Detailed in Chapter 8 [String Parsers and Multiple Parsers], page 30.

`--include-getopt`
: Adds the code for `getopt_long` into the generated parser C file. This will make your generated parser much bigger, but it will be compiled in any system, even if `getopt_long` is not part of the C library where your program is being compiled. See also Chapter 9 [No getopt_long], page 34.

`--show-help`
`--show-full-help`
`--show-version`
: only make gengetopt show the output of `--help`, `--full-help` and `--version` command lines without generating any code, See [Automatically added options], page 9. For instance, I use the `--show-help` option to generate a texinfo file with the output of help (this also shows an example of use of `--set-package` and `--set-version`):

```
../src/gengetopt --show-help -i ../src/cmdline.ggo \
   --set-package="gengetopt" \
   --set-version="" > help_output.texinfo
```

You may have already guessed it: gengetopt uses gengetopt itself for command line options, and its specification file is `cmdline.ggo` in the source directory. In particular the command line for gengetopt itself is generated with the following command:

```
gengetopt --input=cmdline.ggo --no-handle-version \
          --no-handle-help --no-handle-error
```

Indeed when `--help|-h` is passed on the command line, gengetopt will call `cmdline_parser_print_help()` and then the lines for reporting bugs. When `--version|-V` is passed, it will call cmdline_parser_print_version() and then prints a copyright. If an error occurs it prints a message on the screen:

```
$ ./gengetopt --zzzz
./gengetopt: unrecognized option ‘--zzzz’
Run gengetopt --help to see the list of options.
```

4 Terminology

An *argument* is an element of the `argv` array passed into your C or C++ program by your operating system.

An *option* is an argument that begins with '`-`', or '`--`'.

A *value* is an argument, or part of an argument, that is associated with a particular option (an option may also not accept any value). For example, in

```
> ls --width=80
```

`ls` is called with one argument, '`--width=80`', which is an option that has a value, `80`, while in

```
> ls --width 80
```

`ls` is called with two arguments, '`--width`', which is an option, and `80` which might or might not be a value. In this case, whether the `80` is treated as a value associated with the preceding '`--width`' option, or as the name of a file to list depends on how `ls` parses the '`--width`' option.

The order in which options are specified is *usually* unimportant:

```
> ls -a -l
> ls -l -a
```

both do exactly the same thing.

An *parameter* is an argument that is not an option. For example, in

```
> cp --archive source dest
```

`cp` is called with three arguments, the option '`--archive`', the parameter `source`, and the parameter `dest`. Unlike options, the order in which parameters are specified usually *is* important:

```
> cp --archive --verbose source dest
> cp --verbose --archive source dest
> cp --archive source --verbose dest
> cp --archive --verbose dest source
```

The first three `cp` commands do the same thing, but the fourth one is completely different.

If you're new to Gengetopt, you may wish to skip the rest of this section. It goes into more detail about different sorts of options, and how they are parsed.

Note that some parameters may begin with '`-`' or '`--`'. Equivalently, not *all* arguments that begin with '`-`' or '`--`' are options. Consider

```
> ls -- -file
> tar -c -f - . > ../foo.tar
```

The `ls` command has two arguments; the first argument, '`--`' is ignored by `ls`, but causes the '`-file`' argument to be interpreted as a parameter. The `tar` command has four arguments. The '`-c`' argument tells tar to create an archive; the '`-f`' argument, which takes a value, `-`, tells tar that the archive should be written onto the standard output, and the fourth argument, `.`, tells tar what directories to include in the archive. (The remaining two items, `>` and `../foo.tar`, tell the shell to redirect the `tar` command's output to the file ../foo.tar. The `tar` command doesn't even see them.)

The GNU convention is that '`-`' by itself is always interpreted as a value or parameter, while the first '`--`' by itself is always ignored, but causes all subsequent arguments to be interpreted as parameters. Gengetopt always behaves this way.

A *short option* is an option that begins with '`-`'. Not including the leading dash, short options must be one character long:

```
> ls -a -l -t --width=80
```

The '`-a`', '`-l`', and '`-t`' options are all short options. Multiple short options may be combined into a single argument:

```
> ls -alt --width=80
```

is equivalent to the above example.

A *long option* is an option that begins with '`-`' or '`--`'. Ignoring the leading punctuation, long options may be one or more characters long:

```
> ls --all -fs
```

The `ls` command has two arguments; the long option '`--all`', and the pair of short options '`-fs`'.

Long options need not have synonymous short options; after all, complex programs like `cc` have more long options than there are valid short option characters; it wouldn't be possible to assign a short option to each of them. Short options are encouraged, but not required, to have a synonymous long option.

Long options may be abbreviated, as long as the abbreviation is not ambiguous. Gengetopt automatically treats unambiguous abbreviations as synonyms.

Short options may have values just like long options, but if several short options are grouped together into one argument, only the last one may have a value. Values in the same argument as a long option are delimited by an equals sign, values in the same argument as a short option are not:

```
> ls --width 60  # ok, value is "60"
> ls --width=60  # ok, value is "60"
> ls -w60        # ok, value is "60"
> ls -w 60       # ok, value is "60"
> ls -w=60       # unexpected, value is "=60"
> ls -T7 -w60    # ok, value for -T is 7, value for -w is 60
> ls -T7w60      # unexpected, value for -T is "7w60", no -w at all
```

A *required option* must be present, otherwise an error will be raised.

A *multiple option* is an option that may appear more than once on the command line. Gengetopt would create a tidy array for multiple options (see Chapter 7 [Multiple Options], page 28, for further details about dealing with multiple options).

5 Group options

It is also possible to group options; options belonging to a *group* are considered *in mutual exclusion*. In order to use this feature, first the `group` has to be defined, and then a `groupoption` can be defined. A groupoption has basically the same syntax of a standard option, apart that the required flag must not be specified (it would not make sense, since the options of the same group are mutual exclusive) and the group to which the option belongs has to be specified.

```
defgroup "<group name>" {groupdesc="<group description>"} {yes}
groupoption <long> <short> "<desc>" <argtype> group="<group name>" \
     {argoptional} {multiple}
```

If a group is defined as required, then one (but only one) option belonging to the group has to be specified.

Here's an example (taken from the test '`test_group_cmd.ggo`'):

```
defgroup "my grp2"
defgroup "grp1" groupdesc="an option of this group is required" required
groupoption "opta" a "string a" group="grp1" multiple
groupoption "optA" A "string A" string group="grp1" argoptional
groupoption "optAmul" M "string M" string group="grp1" argoptional multiple
groupoption "optb" b "string b" group="grp1"
groupoption "optc" - "string c" group="my grp2"
groupoption "optd" d "string d" group="my grp2"
```

The group `grp1` is required, so either `--opta` or `--optb` has to be specified (but only one of them). Here's the output of some executions:

```
$ ./test_groups
test_groups: 0 options of group grp1 were given. One is required
$ ./test_groups -a          OK
$ ./test_groups -a -a       OK (the same option given twice)
$ ./test_groups -a -b
test_groups: 2 options of group grp1 were given. One is required
$ ./test_groups -a -c       OK
$ ./test_groups -a --optc -d
test_groups: 2 options of group my grp2 were given. At most one is required
```

6 Configuration files

It is often useful to specify command line options directly in a configuration file, so that the value of some options are read from this file if they are not given as command line options. When the command line option `-C|--conf-parser` is given to gengetopt, apart from the standard command line option parser, also this additional parser is generated (its name is `<commandline_parser>_configfile`):

```
int
<cmd_parser_name>_configfile (char * const filename,
                              struct gengetopt_args_info *args_info,
                              int override,
                              int initialize,
                              int check_required);
```

The parameter `override` tells whether the values read in the configuration file have to override those specified at the command line. The `initialize` tells whether the args_info struct has to be initialize.

IMPORTANT: you have to explicitly set `initialize` to true (i.e., different from 0) if you call the config file parser before the standard command line option parser, otherwise unpredictable results may show.

The `check_required` tells whether the check for required options must be performed or not.

If you call the config file parser before the standard command line option parser and then you want to call the standard command line parser you MUST use this second version, passing 0 for initialized, so that collected values from the config file are not lost:

```
int
<cmd_parser_name>2 (int argc,
                    char * const *argv,
                    struct gengetopt_args_info *args_info,
                    int override,
                    int initialize,
                    int check_required);
```

Notice, that with this version you can also specify whether the options passed at the command line must override the ones read from the config file. Moreover, you have to specify whether the check for missing required options must be performed or not. This concerns also options of a required group (Chapter 5 [Group options], page 23).

If you decide not to request the check for required option, you can test it manually, after the command line parsing returns by using the following generated function:

```
int
<cmd_parser_name>_required (struct gengetopt_args_info *args_info,
                            const char *program_name);
```

where `program_name` is the name of your executable (usually you should pass `argv[0]` as argument). If the function returns a value different from 0, then some required options are missing. An error has already been printed by this function. This concerns also options of a required group (Chapter 5 [Group options], page 23).

The config file has the following simple syntax: lines starting with # are considered comments and:

```
<option_name> = {<option_val>}
```

or simply (if the option does not take an argument):

```
<option_name>
```

which means that `option_name` is given, and if it accepts an argument, then its value is `option_val`. The `=` is not mandatory.

Since version 2.19, it is possible to include other files (i.e., other configuration files) in a configuration file, by using the `include` syntax:

```
include "filename"
```

For instance here's a program that uses this feature (this is the test '`test_conf_parser`'):

```
/* test_conf_parser.c test */

/* test all kinds of options and the conf file parser */

#include <stdlib.h>
#include <stdio.h>

#include "test_conf_parser_cmd.h"

static struct my_args_info args_info;

int
main (int argc, char **argv)
{
  int i;
  int result = 0;

  if (test_conf_parser_cmd_parser (argc, argv, &args_info) != 0) {
    result = 1;
    goto stop;
  }

  /* override cmd options, but do not initialize args_info, check for required options */
  if (test_conf_parser_cmd_parser_configfile
      (args_info.conf_file_arg, &args_info, 1, 0, 1) != 0)
    {
      result = 1;
      goto stop;
    }

  printf ("value of required: %s\n", args_info.required_arg);
  printf ("value of string: %s\n", args_info.string_arg);
  printf ("value of no-short_given: %d\n", args_info.no_short_given);
  printf ("value of int: %d\n", args_info.int_arg);
  printf ("value of float: %f\n", args_info.float_arg);

  printf ("value of multi-string_given: %d\n", args_info.multi_string_given);
  for (i = 0; i < args_info.multi_string_given; ++i)
    printf ("  value of multi-string: %s\n", args_info.multi_string_arg [i]);

  printf ("value of multi-string-def_given: %d\n",
          args_info.multi_string_def_given);
  for (i = 0; i < args_info.multi_string_def_given; ++i)
```

```
    printf ("  value of multi-string-def: %s\n",
            args_info.multi_string_def_arg [i]);
  if (!args_info.multi_string_def_given && args_info.multi_string_def_arg [0])
    printf ("default value of multi-string-def: %s\n",
            args_info.multi_string_def_arg [0]);

  printf ("value of opta: %s\n", args_info.opta_arg);

  printf ("noarg given %d times\n", args_info.noarg_given);
  printf ("noarg_noshort given %d times\n", args_info.noarg_noshort_given);

  printf ("opt-arg given: %d\n", args_info.opt_arg_given);
  printf ("opt-arg value: %s\n", (args_info.opt_arg_arg ? args_info.opt_arg_arg : "not given"));

  if (args_info.file_save_given) {
    if (test_conf_parser_cmd_parser_file_save (args_info.file_save_arg, &args_info) == EXIT_FAILURE)
      result = 1;
    else
      printf ("saved configuration file %s\n", args_info.file_save_arg);
  }

 stop:
  test_conf_parser_cmd_parser_free (&args_info);

  return result;
}
```

So if we use the following config file

```
# required option
required "this is a test"
float 3.14
no-short
string another
```

and we run `test_conf_parser` like that, we will have

```
./test_conf_parser -r bar -i 100 --float 2.14 --conf-file test_conf.conf
value of required: this is a test
value of string: another
value of no-short: 1
value of int: 100
value of float: 3.140000
```

If, instead we call the `test_conf_parser_cmd_parser_configfile` with 0 for override argument, we get the following result

```
value of required: bar
value of string: another
value of no-short: 1
value of int: 100
value of float: 2.140000
```

This second example use the second version of the command line parser: first call the configuration file parser and then the command line parser (the command line options will override the configuration file options):

```
/* test_conf_parser_ov2.c test */

/* test all kinds of options and the conf file parser */
/* differently from test_conf_parser_ov.c, first scan the conf file and
   then the command line */

#include <stdlib.h>
#include <stdio.h>

#include "test_conf_parser_cmd.h"

static struct my_args_info args_info;

int
main (int argc, char **argv)
{
  /* do not override command line options, initialize args_info */
  if (test_conf_parser_cmd_parser_configfile
      ("./test_conf2.conf", &args_info, 0, 1, 0) != 0)
    exit(1);

  if (test_conf_parser_cmd_parser2 (argc, argv, &args_info, 1, 0, 1) != 0)
    exit(1) ;

  printf ("value of required: %s\n", args_info.required_arg);
  printf ("value of string: %s\n", args_info.string_arg);
  printf ("value of no-short_given: %d\n", args_info.no_short_given);
  printf ("value of int: %d\n", args_info.int_arg);
  printf ("value of float: %f\n", args_info.float_arg);

  test_conf_parser_cmd_parser_free (&args_info);

  return 0;
}
```

This is an invocation and its results:

```
./test_conf_parser_ov2 -r "bar" --float 2.14 -i 100
value of required: bar
value of string: another
value of no-short: 1
value of int: 100
value of float: 2.140000
```

6.1 Further details on the configuration file parser

The generated config file parser function uses the constant `CONFIG_FILE_LINE_SIZE` to read each line of the configuration file. By default this constant is set to `2048` that should be enough for most applications. If your application uses configuration files with lines that are longer, you can compile the generated C file by specifying an explicit value for this constant with the `-D` command line option of `gcc`.

7 Multiple Options

If an option is specified as `multiple`, then it can be specified multiple times at command line. In this case, say the option is called `foo`, the generated `foo_given` field in the args structure contains the number of times it was specified and the generated field `foo_arg` is an array containing all the values that were specified for this option.

Notice that if a default value is specified for a multiple option, that value is assigned to the option only if no other value is specified on the command line (and the corresponding `_given` field will be set to 1), i.e., a default value IS NOT always part of the values of a multiple option.

For instance, if the gengetopt file is as follows

```
# test options that can be given more than once
option "string"      s "string option" string optional multiple
option "int"         i "int option" int optional multiple
```

Then the command line options can be collected like that

Then if this program is called with the following command line options

```
/* test options that can be given more than once */

#include <stdlib.h>
#include <stdio.h>

#include "test_multiple_cmd.h"

static struct gengetopt_args_info args_info;

int
main (int argc, char **argv)
{
  int i = 0;

  if (test_multiple_cmd_parser (argc, argv, &args_info) != 0)
    exit(1) ;

  for (i = 0; i < args_info.string_given; ++i)
    printf ("passed string: %s\n", args_info.string_arg[i]);

  for (i = 0; i < args_info.int_given; ++i)
    printf ("passed int: %d\n", args_info.int_arg[i]);

  return 0;
}
```

The output of the program will be

```
passed string: world
passed string: hello
passed string: bar
passed string: foo
passed int: 200
passed int: 100
```

You can also pass arguments to a multiple option separated by commas (if you need to actually specify the comma operator as part of the argument you can escape it with \), as in the following:

```
./test_multiple -s"foo","bar","hello" -i100,200 -s "world"
```

You can specify the number of occurrences of multiple options by using the following syntax (that must be given after the `multiple` keyword):

`(number)` requires that the multiple option, if given, must be given exactly `number` times

`(number1-number2)`
: requires that the multiple option, if given, must be given not less than `number1` times and no more than `number2` times

`(number-)`
: requires that the multiple option, if given, must be given not less than `number` times

`(-number)`
: requires that the multiple option, if given, must be given not more than `number` times

Here are some examples:

```
option "string"      s "string option" string optional multiple(4)
option "string"      s "string option" string optional multiple(1-4)
option "string"      s "string option" string optional multiple(-5)
```

Notice that this is independent from the `required` flag.

8 String Parsers and Multiple Parsers

The parsers generated by gengetopt (indeed the C and header files) are self-contained and different parsers can be linked in the same program, without interferences. This is useful, e.g., in cases where a specific command line option argument has a complex syntax that accepts options itself according to terminology already defined, the one handled by `getopt_long`, see Chapter 4 [Terminology], page 21. Another case when multiple parsers can be useful is when your command behaves differently according to a specific command line option.

Obviously there exists only one instance of command line arguments passed to the `main` function (namely the variables `argc` and `argv`) so passing the same arguments to different command line parsers is likely to generate errors: the different command line parsers are likely to have different syntaxes for accepted options.

For this reason gengetopt can generate parser functions that take a string containing the further options to parse, instead of taking an array. This additional parser will have the parser name and the suffix `_string`. If you want these additional parsers to be generated you have to pass the command line option `-S|--string-parser` to gengetopt (see Chapter 3 [Invoking gengetopt], page 18). The two functions will be:

```
int <parser_name>_string (const char *cmdline,
    struct test_first_cmdline_cmd_struct *args_info,
    const char *prog_name);
int <parser_name>_string2 (const char *cmdline,
    struct test_first_cmdline_cmd_struct *args_info,
    const char *prog_name,
    int override, int initialize, int check_required);
```

The second one allows you to specify more details about the parsing (this is the same as for configuration files, thus we refer to that section for the details of the two functions and default values, see Chapter 6 [Configuration files], page 24).

Of course, these functions can be used in general to simulate the invocation of a program with specific command line options (stored in the first string argument), or in general to parse options that are all stored in a string (instead of a vector).

The first argument of these parsers is a string containing the options to parse (remember that this must respect the option format handled by `getopt_long`, see Chapter 4 [Terminology], page 21). The second one is the pointer to the struct that will be filled with passed options and arguments, as usual. The third option is the program name: this will be used when errors have to be printed. This last argument can be null: in this case, the first element of the first string argument is considered the program name.

Let's show these functionalities with an example. Consider a program that accepts two command line options (required in this case):

```
# test for multiple parsers, this is the main file
# test_main_cmdline_cmd.ggo

option "first-cmd" F "the first command line to parse" required \
       typestr="first command" string multiple
option "second-cmd" S "the second command line to parse" required \
```

```
        typestr="second command" string multiple
```

These two options accept strings as argument that in turn are considered command line arguments, according to specific syntaxes. The first one is:

```
# test for multiple parsers, this is the first command line file
# test_first_cmdline_cmd.ggostr

option "option-a" a "option a of the first command line to parse"
optional int option "multi" M \
         "multiple option of the first command line to parse" \
         optional string multiple
```

and the second one is:

```
# test for multiple parsers, this is the second command line file
# test_second_cmdline_cmd.ggostr

option "option-a" a "option a of the second command line to parse" \
       optional string
option "option-b" b "option a of the second command line to parse" \
       optional string
option "my-multi" M \
       "multiple option of the second command line to parse" \
       optional string multiple
```

These last two files are processed with gengetopt using the `--string-parser`. Let's put everything together in this main file:

```
#include <stdio.h>
#include <stdlib.h>

#include "test_main_cmdline_cmd.h"
#include "test_first_cmdline_cmd.h"
#include "test_second_cmdline_cmd.h"

int
main(int argc, char **argv)
{
  struct gengetopt_args_info main_args_info;
  struct test_first_cmdline_cmd_struct first_args_info;
  struct test_second_cmdline_cmd_struct second_args_info;
  int exit_code = 0, i, j;

  if (test_main_cmdline_cmd_parser (argc, argv, &main_args_info) != 0) {
    exit_code = 1;
    return exit_code;
  }

  for (j = 0; j < main_args_info.second_cmd_given; ++j) {
    printf("second cmdline: %s\n", main_args_info.second_cmd_arg[j]);
    if (test_second_cmdline_cmd_parser_string
       (main_args_info.second_cmd_arg[j], &second_args_info, argv[0]) == 0) {
      if (second_args_info.option_a_given)
       printf("  --option-a: %s\n", second_args_info.option_a_arg);
      if (second_args_info.option_b_given)
       printf("  --option-b: %s\n", second_args_info.option_b_arg);
```

```
      for (i = 0; i < second_args_info.my_multi_given; ++i)
       printf("  --my-multi: %s\n", second_args_info.my_multi_arg[i]);

      test_second_cmdline_cmd_parser_free (&second_args_info);
    }
  }

  for (j = 0; j < main_args_info.first_cmd_given; ++j) {
    printf("first cmdline: %s\n", main_args_info.first_cmd_arg[j]);
    if (test_first_cmdline_cmd_parser_string
       (main_args_info.first_cmd_arg[j], &first_args_info, argv[0]) == 0) {
      if (first_args_info.option_a_given)
       printf("  --option-a: %d\n", first_args_info.option_a_arg);
      for (i = 0; i < first_args_info.multi_given; ++i)
       printf("  --multi: %s\n", first_args_info.multi_arg[i]);

      test_first_cmdline_cmd_parser_free (&first_args_info);
    }
  }

  test_main_cmdline_cmd_parser_free (&main_args_info);

  return exit_code;
}
```

Notice that in the `for` loops we always free the elements of the argument structures in order to avoid memory leaks.

Now if you can run this program as follows (notice that we use the comma separated arguments for multiple option arguments but we escape it with `\` because otherwise, e.g., 200 and 300 would be intended as further arguments of `--first-cmd` instead of `--multi`, see Chapter 7 [Multiple Options], page 28):

```
./test_multiple_parsers \
        --first-cmd="-M400 -a10 --multi 100\,200\,300" \
        --second-cmd="-a20 -b10 --my-multi=a\,b\,c\,d\,e\,f" \
        -F"-M500 -M600" -S"--my-multi g"
second cmdline: -a20 -b10 --my-multi=a,b,c,d,e,f
  --option-a: 20
  --option-b: 10
  --my-multi: a
  --my-multi: b
  --my-multi: c
  --my-multi: d
  --my-multi: e
  --my-multi: f
second cmdline: --my-multi g
  --my-multi: g
first cmdline: -M400 -a10 --multi 100,200,300
  --option-a: 10
  --multi: 400
  --multi: 100
  --multi: 200
```

```
  --multi: 300
first cmdline: -M500 -M600
  --multi: 500
  --multi: 600
```

9 What if getopt_long is not available?

If you use gengetopt to generate C functions for parsing command line arguments you have to know that these generated functions use `getopt_long` to actually read the command line and parsing it. This function is typically part of the standard C library, but some implementations may not include it. If you want your program to be portable on several systems, and be compilable with many C compilers, you can rely on one of the following solutions.

9.1 Include the `getopt_long` code into the generated parser

Since version 2.17, gengetopt can include into the generated C parser file the code of `getopt_long`, so that the include code will be used to actually parse the command line arguments, instead of that taken from the C library.

This solution is actually quite easy, but it has two main drawbacks:

- The generate C file of the parser will be much bigger;
- You won't be able to use the latest version of `getopt_long` of the C library

It is up to you to choose between this and the automake/autoconf based solution.

Actually, this solution has the advantage that your program won't behave strangely when used with another implementation of `getopt_long`.

I prefer the automake/autoconf based solution, as described in Section 9.2 [Use automake/autoconf], page 34, in particular the one described in Section 9.2.1 [Use Gnulib], page 34, which is also the one I adopt for gengetopt itself.

9.2 Use automake/autoconf to check for the existence of `getopt_long`

Autoconf and Automake are great tools to generate a configure script that automatically checks for the configuration of your system and for possible missing functions required to compile your program. However, in case of detected missing functions, your program must be able to provide a replacement for such functions. In the next sections we describe two mechanisms for including the (possible) missing code for `getopt_long` and for checking its presence with automake/autoconf. Since version 2.19, gengetopt itself uses the first mechanism.

9.2.1 Use Gnulib

Since version 2.19 I also started to use Gnulib - The GNU Portability Library[1], "a central location for common GNU code, intended to be shared among GNU packages". Gnulib provides an easy and smooth way to add to your package sources the sources of functions that you want to check during configure. It will also handle the checks for these functions in the configure script, and in case they're not in your system (or they're present but with some missing features) it compiles their sources into a library (that you will need to link your program to, as illustrated in the following).

[1] http://www.gnu.org/software/gnulib

Once you retrieved gnulib (for the moment it is available only through CVS, see the home page), you can invoke '`gnulib-tool --import`' that will copy source files, create a '`Makefile.am`' to build them, generate a file '`gnulib-comp.m4`' with Autoconf M4 macro declarations used by '`configure.ac`', and generate a file '`gnulib-cache.m4`' containing the cached specification of how Gnulib is used. In particular, you must specify the modules you want to import, and in our case, it is getopt:

```
gnulib-tool --import getopt
```

By default, the source code is copied into '`lib/`' and the M4 macros in '`m4/`'. You can override these paths by using `--source-base=DIRECTORY` and `--m4-base=DIRECTORY`. For instance, gengetopt uses `gl` and `gl/m4`, respectively. We will use these directories in the rest of this section.

You must ensure Autoconf can find the macro definitions in '`gnulib-comp.m4`'. Use the `ACLOCAL_AMFLAGS` specifier in your top-level '`Makefile.am`' file (and the first time you run `aclocal` you have to use the `-I` as well); for instance, in the case of gengetopt we have:

```
ACLOCAL_AMFLAGS = -I gl/m4
```

You are now ready to call the M4 macros in `gnulib-comp.m4` from '`configure.ac`'. The macro `gl_EARLY` must be called as soon as possible after verifying that the C compiler is working. Typically, this is immediately after `AC_PROG_CC`, as in:

```
...
AC_PROG_CC
gl_EARLY
...
```

The core part of the gnulib checks are done by the macro `gl_INIT`. Place it further down in the file, typically where you normally check for header files or functions. For example:

```
...
# For gnulib.
gl_INIT
...
```

`gl_INIT` will in turn call the macros related with the gnulib functions, be it specific gnulib macros. So there is no need to call those macros yourself when you use the corresponding gnulib modules.

You must also make sure that the gnulib library is built. Add the `Makefile` in the gnulib source base directory to `AC_CONFIG_FILES`, as in:

```
AC_CONFIG_FILES(... gl/Makefile ...)
```

You must also make sure that `make` will recurse into the gnulib directory. To achieve this, add the gnulib source base directory to a `SUBDIRS` Makefile.am statement, as in:

```
SUBDIRS = gl
```

Finally, you have to add compiler and linker flags in the appropriate source directories, so that you can make use of the gnulib library. Since the '`getopt`' module copies files into the build directory, '`top_builddir/gl`' is needed as well as '`top_srcdir/gl`'. For example:

```
...
AM_CPPFLAGS = -I$(top_srcdir)/gl -I$(top_builddir)/gl
...
```

```
LDADD = gl/libgnu.a
...
```

Don't forget to `#include` the various header files. In this example, you would need to make sure that '`#include "getopt.h"`' is evaluated when compiling all source code files, that want to make use of `getopt` or `getopt_long`. If you simply use the files generated by gengetopt, you won't need include this header though, since it is already handled by the generated files.

Every now and then, check whether there are updates in the Gnulib modules, and if the modules you use (e.g., `getopt`) are upgraded, please remember to also update your files, simply by running:

```
gnulib-tool --update
```

We refer to Gnulib documentation for further explanations and features.

9.2.2 Use getopt_long sources

NOTICE: this was the procedure used by gengetopt itself up to version 2.18. We suggest now to use the procedure described in Section 9.2.1 [Use Gnulib], page 34, since the files described in the following might not be kept up-to-date.

We provide C files that actually implement `getopt_long` function: '`getopt.c`' '`getopt1.c`' and '`gnugetopt.h`'. You'll find these files in the '`<install prefix>/share/gengetopt`' directory where '`<install prefix>`' is the one you specified during compilation. If no prefix had been specified, '`/usr/local`' is the default. If you downloaded gengetopt in binary form prefix will probably be '`/usr/local`' or '`/usr`'.

You can rename '`gnugetopt.h`' to '`getopt.h`' and then simply compile these files and link them to the executable of you program. However, if you use automake and autoconf here's a more elegant solution: you should download the file '`adl_func_getopt_long.m4`' you find at this site:

http://autoconf-archive.cryp.to

and add its contents to your '`acinclude.m4`'. You can find this macro also in the '`acinclude.m4`' in the sources of gengetopt.

This macro checks if getopt_long function is in C library; if it is not then it adds '`getopt.o`' and '`getopt1.o`' to the objects files that will be linked to your executable (`LIBOBJS`).

Then in '`Makefile.am`' of your source directory you have to add the contents of `LIBOBJS` to the `LDADD` of the program that has to use getopt_long; e.g., if the program '`foo`' has to use getopt_long, you have to add the following line

```
foo_LDADD = @LIBOBJS@
```

Now these files will be compiled and linked to your program only if necessary.

Moreover you have to add '`getopt.c`' '`getopt1.c`' and '`gnugetopt.h`' to your distribution. Note that it is not necessary to put these file names among the `foo_SOURCES` contents), but you have to add '`gnugetopt.h`' to `EXTRA_DIST`:

```
EXTRA_DIST = gnugetopt.h
```

You may want to take a look at gengetopt's '`configure.in`' and '`src/Makefile.am`': they both use the techniques described here.

10 Known Bugs and Limitations

If you find a bug in `gengetopt`, please send electronic mail to

`bug-gengetopt at gnu dot org`

Include the version number, which you can find by running '`gengetopt --version`'. Also include in your message the output that the program produced and the output you expected.

If you have other questions, comments or suggestions about `gengetopt`, contact the author via electronic mail (find the address at http://www.lorenzobettini.it). The author will try to help you out, although he may not have time to fix your problems.

The list of to-dos in the '`TODO`'.

10.1 Getopt and subsequent calls

It seems that `getopt_long`, at least the version in the GNU library, if invoked with different `argv` arrays, might access memory in a bad way leading to crashes or unexpected behaviors. This happens because it keeps pointers to locations of the previous arrays if not initialized each time by setting `optind = 0`[1]. Unfortunately this initialization behavior seems to be part only of the implementation of GNU library and actually it is not documented (you can see it by taking a look into the source of '`getopt.c`'); other implementations of `getopt_long` might not be affected by this problem; alternatively, as reported by a user, `optind = 0` leads some `getopt_long` implementations to consider the program name as a command line option (since it is in position 0), which is bad anyway!

Probably this is usually not a problem since you usually parse only the command line, thus you only invoke the command line parser only once, and only with one instance of array (i.e., the `argv` passed to `main`). However, it can lead to problems when you use advanced features, as in the case of configuration file parsing (see Chapter 6 [Configuration files], page 24) and multiple parsers (see Chapter 8 [String Parsers and Multiple Parsers], page 30).

To deal with this problem, in case one uses configuration parsers (`-C,--conf-parser`) or string parsers (`-S,--string-parser`) a customized version of `getopt_long`, called indeed `custom_getopt_long` is inserted in the generated C file. This version does not suffer from the problem above (and of course does not interfere with the standard `getopt_long`, which will be used to parse the actual command line arguments).

Of course this is transparent to the programmer, and we report this detail here only because the inserted `custom_getopt_long` will add about 20k of C code in the generated file, so don't panic if you start to use configuration or string parsers and your program gets a little bigger than before :-).

[1] `optind` is the global variable in getopt implementation that is the index in ARGV of the next element to be scanned. This is used for communication to and from the caller and for communication between successive calls to `getopt_long`.

11 Mailing Lists

The following mailing lists are available:

`help-gengetopt at gnu dot org`

for generic discussions about the program and for asking for help about it (open mailing list), http://mail.gnu.org/mailman/listinfo/help-gengetopt

`info-gengetopt at gnu dot org`

for receiving information about new releases and features (read-only mailing list), http://mail.gnu.org/mailman/listinfo/info-gengetopt.

If you want to subscribe to a mailing list just go to the URL and follow the instructions, or send me an e-mail and I'll subscribe you.

I'll describe new features in new releases also in my blog, at this URL:

http://tronprog.blogspot.com/search/label/gengetopt

Index

R

S

T

U

V

W