
Parsing command line arguments using Glib

Ikke

Version 0.1

Copyright © 2005 Ikke

Published under the "Creative Commons Attribution-ShareAlike License Belgium"

Table of Contents

Introduction	1
GOption Features overview	2
Some basic parsing samples	2
Other types of options	4
Option grouping	6
Conclusion	7

Introduction

Everyone who ever saw a console (and which FOSS developer didn't ever see one?) knows command line options. They're a very powerful way to give information to a program, whether it's console or GUI based. Command line options are very useful when scripting: you can put the flags in a script, avoiding the need to answer to questions the program may ask you.

On GNU systems, there's the convention to use 2 types of flags: short ones and long ones. Short flags take one hyphen and a character, like `ls -l`. Long flags got 2 hyphens, and consist of a string, like `ls --all`. Both types can take arguments: when using the short form, the argument must be given after the flag, separated with a space (`ls -w 10`), the long form can take an argument in 2 forms: the same way the short format is used (`ls --width 10`), or separated with an equal sign (`ls -width=10`).

Parsing these command line options can be a very tedious task, especially in C. The options are given as an array of string to your `main()` entypoint, and as we all know, string manipulations in C can be a hell of a job.

To make life easier for programmers, people writing libc implementations started to add functions to their libraries that made command line parsing easier. On *nix, we got `getopt()`, and on GNU systems `getopt_long()`. These functions can be very difficult to use in some situations though, and got their limitations, like non-portability to some systems.

After a while the Popt library was created to make life easier once more. This library allows the developers to parse the command line for long and short options in one command, but iterations were still needed. Also the portability issue was still there. Despite its limitations the Popt library was the preferred way to parse command line options for the Gnome project for a long time though.

Lately a new player entered the arena, trying to solve all previously mentioned problems. Since the latest stable release of the Glib library, a new functionality called "GOptions" was introduced. This group of functions and types solves the option parsing problems in several ways:

- Because it's part of Glib, your code is easily portable to every platform Glib supports, including Linux, *BSD, MacOSX, Windows, and others.

- One step parsing: no iterations necessary.
- Integrated in Glib, no external libraries needed, as Glib is installed on most *nix desktop systems nowadays.

GOption Features overview

Some of the features GOptions give you:

- Long and short option parsing, without the need to use multiple options (i.e.: when defining an option, one can assign it both a long and a short option format, no need to define 2 options, once with the long, once with the short format).
- Automagic assigning of variables: similar to Popt, one can assign a gchar * to an option (or some other type, depending on the option type), which will be given a value during parsing.
- Automatic usage generation (when parsing the command line, GOption will show usage info when --help, -? or --help-all is found).
- Options grouping.
- Full internationalization (I18N) support: given option descriptions or possible values can be gettext'ized.

This feature overview is not complete, and more features may be added in future releases.

Currently (Glib 2.6.2) some features are broken: "-?" parsing does not work, options aren't aligned correctly in some cases,... GOptions are a very new feature in Glib, if you find any bugs you should report them in the Glib section [http://bugzilla.gnome.org/enter_bug.cgi?product=glib] of GNOME Bugzilla [<http://bugzilla.gnome.org>].

Some basic parsing samples

We'll start with some basic samples of options parsing. First create a new directory inside `src/`. I called it `goptions`.

As mentioned before, GOptions are only available in Glib ≥ 2.6 , so it's a good thing to add a check to the `configure` script. This is easy to do: we just need to edit `configure.in`, so **pkg-config** will also check the version of the installed package. Find the section where we query `glib-2.0`, and add ≤ 2.6 in there, like this:

```
PKG_CHECK_MODULES(GLIB, glib-2.0 >= 2.6, have_glib=true, have_glib=false)
```

Now generate a new `configure` script by running `autogen.sh`.

We'll start writing some simple code that checks whether the `-b` or `--boolean` is given.

```
#include <glib.h>

int main(int argc, char *argv[])
{
    gboolean b = FALSE;
    GOptionEntry options[] = {
        { "boolean", 'b', 0, G_OPTION_ARG_NONE, &b, "Boolean test", NULL },
        { NULL }
    };
    GOptionContext *ctx;

    ctx = g_option_context_new("- Our first GOption sample");
    g_option_context_add_main_entries(ctx, options, "example1");
```

```

g_option_context_parse(ctx, &argc, &argv, NULL);
g_option_context_free(ctx);

g_print("'b' == '%s'\n", (b == TRUE? "TRUE" : "FALSE"));

return 0;
}

```

Let's get through this code step by step:

- First we include the Glib header file as usual.
- Our `main()` entry point takes an integer and an array of strings as arguments, like every conventional C program.
- We define a Glib `gboolean` with default value `FALSE`
- We define an array of `GOptionEntries`. This is the definition of a `GOptionEntry` (from the `GOption` API):

```

typedef struct {
    const gchar *long_name;
    gchar       short_name;
    gint        flags;

    GOptionArg  arg;
    gpointer    arg_data;

    const gchar *description;
    const gchar *arg_description;
} GOptionEntry;

```

- `flags` can be 0 in almost all circumstances.
- `arg` must be one of
 - `G_OPTION_ARG_NONE`, which takes no parameters. `arg_data` must be a pointer to a `gboolean`.
 - `G_OPTION_ARG_STRING`, which takes a string as parameter. `arg_data` should be a pointer to a `gchar` string.
 - `G_OPTION_ARG_INT`, which takes an integer as argument. `arg_data` should be a pointer to a `gint`.
 - `G_OPTION_ARG_CALLBACK`. `arg_data` should be a function pointer to a function used to parse the argument. Not further used here.
 - `G_OPTION_ARG_FILENAME`, which takes a filename as argument. `arg_data` should be a pointer to a `gchar` string. The filename is automatically converted to the correct encoding by Glib.
 - `G_OPTION_ARG_STRING_ARRAY`. If the option is given more than once, the parameters given to all occurrences will be stored in the `gchar` string array pointed to by `arg_data`
 - `G_OPTION_FILENAME_ARRAY`, similar to `G_OPTION_ARG_STRING_ARRAY`, but better suited for filenames.
- `description` is a description of the option.
- `arg_description` is a string displayed after the option name in the usage overview. Here you can give different possible options, the name of a variable,...

- We create a variable of type `GOptionContext`, which will store all necessary information the `GOption` functions use internally while parsing. One shouldn't access the member fields of this type directly.
- We create a new `goption` context. This allocates the necessary memory, and initializes the structure. The option we provide is a small help message, explaining what our program does.
- Now we add the main entries table to the context, providing a name. Read on to know what main entries are.
- We parse the arguments given to our program using our context. We provide `NULL` as last parameter here. This should be a `GError **`, but we don't do any error handling yet. When parsing is done, `b` will have a new value.
- We free the context. All group entries, including the ones in the main group, will be freed too.
- We end up by giving some debug output, i.e. the value of `b`.

We're not able to execute this code yet: we need to

- Create a `Makefile.am` in `src/goptions/`.
- Update `src/Makefile.am` so `src/goptions` will be built too.
- Update `configure.in` so `src/goptions/Makefile.in` gets processed.

You should be able to do this yourself now.

Once this is done, execute **autogen.sh** once more, run **make**, and play around with our `example1` executable. This is what I got:

```
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example1
'b' == 'FALSE'
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example1 --help
Usage:
  example1 [OPTION...] - Our first GOption sample

Help Options:
  --help                Show help options

Application Options:
  -b, --boolean         Boolean test

ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example1 -b
'b' == 'TRUE'
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example1 --boolean
'b' == 'TRUE'
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example1 -c
'b' == 'FALSE'
```

Other types of options

The next code is an example using several types of options:

```
#include <glib.h>

int main(int argc, char *argv[])
{
    gboolean b = FALSE;
    gchar *s = NULL;
    gchar **sa = NULL;
    gint i = 0;
```

```

GOptionEntry options[] = {
    { "boolean", 'b', 0, G_OPTION_ARG_NONE, &b, "Boolean test", NULL },
    { "string", 's', 0, G_OPTION_ARG_STRING, &s, "String test", "s" },
    { "int", 'i', 0, G_OPTION_ARG_INT, &i, "Int test", "i" },
    { "sarray", 'a', 0, G_OPTION_ARG_STRING_ARRAY, &sa, "Array test" },
    { NULL }
};
GOptionContext *ctx;

ctx = g_option_context_new("- Our second GOption sample");
g_option_context_add_main_entries(ctx, options, "example2");
g_option_context_parse(ctx, &argc, &argv, NULL);
g_option_context_free(ctx);

g_print("'b' == '%s'\n", (b == TRUE? "TRUE" : "FALSE"));
g_print("'s' == '%s'\n", s);
g_print("'i' == '%d'\n", i);

g_print("'sa':\n");
int c = 0;
while(sa != NULL && sa[c] != NULL)
{
    g_print("\t'sa[%d]' == '%s'\n", c, sa[c]);
    c++;
}

return 0;
}

```

This code should be fairly easy to understand.

Edit Makefile.am so example2 will be built too, and remake the project. Now play around with example2.

```

ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example2
'b' == 'FALSE'
's' == '(null)'
'i' == '0'
'sa':
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example2 -b -i 10
'b' == 'TRUE'
's' == '(null)'
'i' == '10'
'sa':
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example2 -b -i 10 -s blah
'b' == 'TRUE'
's' == 'blah'
'i' == '10'
'sa':
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example2 -b -i 10 -s blah
'b' == 'TRUE'
's' == 'blah'
'i' == '10'
'sa':
      'sa[0]' == 'test1'
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example2 -a test2 -b -i 10
'b' == 'TRUE'
's' == 'blah'
'i' == '20'
'sa':
      'sa[0]' == 'test2'
      'sa[1]' == 'test1'
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example2 --help
Usage:
  example2 [OPTION...] - Our first GOption sample

Help Options:
  --help          Show help options

```

```
Application Options:
-b, --boolean      Boolean test
-s, --string=s     String test
-i, --int=i        Int test
-a, --sarray=array Array test
```

Option grouping

I mentioned the "grouping" functionalities of GOptions several times before. Here's where I'll try to explain what this is, and how to use it.

Some programs got a lot of command line options, just execute **nautilus --help** in some console to get a sample of this. It can be very useful to group several related options together then, like it's done for **nautilus**. It is very easy to achieve this using GOptions, even with several extras.

Until now we added all options to the "main" group. To add options in several groups, all we have to do is create other option entries, create groups, add entries to groups and add groups to our GOptionContext.

This is a very simple sample:

```
#include <glib.h>

int main(int argc, char *argv[])
{
    gboolean b1 = FALSE, b2 = TRUE;
    gint i1 = 0, i2 = 100;
    gchar *s = NULL;
    GOptionEntry boolOptions[] = {
        { "boola", 'a', 0, G_OPTION_ARG_NONE, &b1, "Bool1 test", NULL },
        { "boolb", 'b', 0, G_OPTION_ARG_NONE, &b2, "Bool2 test", NULL },
        { NULL }
    };
    GOptionEntry intOptions[] = {
        { "intc", 'c', 0, G_OPTION_ARG_INT, &i1, "Int1 test", "i" },
        { "intd", 'd', 0, G_OPTION_ARG_INT, &i2, "Int2 test", "i" },
        { NULL }
    };
    GOptionEntry mainOptions[] = {
        { "string", 's', 0, G_OPTION_ARG_STRING, &s, "String test", "s" },
        { NULL }
    };
    GOptionContext *ctx;
    GOptionGroup *intgroup, *boolgroup;

    ctx = g_option_context_new("- Our first GOption group sample");

    intgroup = g_option_group_new("intgroup", "Test int group", "Test int g
    boolgroup = g_option_group_new("boolgroup", "Test bool group", "Test bo

    g_option_group_add_entries(boolgroup, boolOptions);
    g_option_group_add_entries(intgroup, intOptions);

    g_option_context_add_main_entries(ctx, mainOptions, "example3 Main opti
    g_option_context_add_group(ctx, boolgroup);
    g_option_context_add_group(ctx, intgroup);

    g_option_context_parse(ctx, &argc, &argv, NULL);
    g_option_context_free(ctx);

    g_print("'b1' == '%s'\n", (b1 == TRUE? "TRUE" : "FALSE"));
    g_print("'b2' == '%s'\n", (b2 == TRUE? "TRUE" : "FALSE"));
    g_print("'s' == '%s'\n", s);
    g_print("'i1' == '%d'\n", i1);
    g_print("'i2' == '%d'\n", i2);
```

```

        return 0;
    }

```

This code is fairly easy to understand too (actually, GOptions are always easy to use, unless you start playing with more experienced things like self-defined parsing functions). Notice we don't `g_option_group_free()` our groups. `g_option_context_free()` frees all groups belonging to the freed context.

Once you get this program to run, play around with it once more. Some samples:

```
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example3
```

```
'b1' == 'FALSE'
'b2' == 'TRUE'
's' == '(null)'
'i1' == '0'
'i2' == '100'
```

```
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example3 --help
```

Usage:

```
example3 [OPTION...] - Our first GOption group sample
```

Help Options:

```
--help           Show help options
--help-all      Show all help options
--help-boolgroup Test bool group help description
--help-intgroup  Test int group help description
```

Application Options:

```
-s, --string=s    String test
```

```
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example3 --help-all
```

Usage:

```
example3 [OPTION...] - Our first GOption group sample
```

Help Options:

```
--help           Show help options
--help-all      Show all help options
--help-boolgroup Test bool group help description
--help-intgroup  Test int group help description
```

Test bool group

```
-a, --boola      Bool1 test
-b, --boolb      Bool2 test
```

Test int group

```
-c, --intc=i     Int1 test
-d, --intd=i     Int2 test
```

Application Options:

```
-s, --string=s    String test
```

```
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example3 --help-boolgroup
```

Usage:

```
example3 [OPTION...] - Our first GOption group sample
```

Test bool group

```
-a, --boola      Bool1 test
-b, --boolb      Bool2 test
```

```
ikke@moonwalker ~/code/gnome-tutorial/src/goptions $ ./example3 -a -b -c 10 -d
```

```
'b1' == 'TRUE'
'b2' == 'TRUE'
's' == 'test'
'i1' == '10'
'i2' == '20'
```

Conclusion

As you can see, it's braindead easy to add command line options functionality to your programs using Glib's GOptions. More complicated things than the ones described here are possible, I did not touch I18N,... Just read through the Glib GObject API [<http://developer.gnome.org/doc/API/2.0/glib/glib-Commandline-option-parser.html#g-option-group-set-translate-func>] to learn more.

GOptions are getting adopted in the Gnome world. In GTK+2, there's a new init function `gtk_init_with_args()` which works the same way as `gtk_init_check()`, also adding the possibility for application developers to add command line options to their GTK+ application.