ProgOptions: Programmers Guide

DocumentId:GradSoft-PR-e-26.09.2000-v1.0.3

May 8, 2003

Contents

1 Introduction

ProgOptions is a component for handling of program command string options.

Using ProgOptions it is possible to simply implement following facilities:

1. ascertaining the presence in the command line of some (previously determined) options
2. getting option arguments (- note the ProgOptions allows to use a few syntax style, for example, the

   ```
   --a Oh
   ```

   syntax for "a" option with "Oh" argument is valid as well as the

   ```
   --a="Say Oh"
   ```

syntax for the same option with another "Say Oh" argument;

3. as soon as the presence of sertain option is ascertained, invoking the function connected.

This document is a non-formal description of package, the full specification is present in API reference.
(`www.gradsoft.kiev.ua/common/ToolBox/ProgOptions/API/index.html`).

2 The basic way of using

The "basic" way of ProgOption usage is suitable for the case of command line accessble through the pair (argc,argv) with personnels determined as (int) and (char**) obtained through the "main" function arguments. If this case is present, obtaining of ProgOptions functionality needs following steps:

- Construct a new instance of ProgOptions keeping in mind the ProgOptions::ProgOptions is defined as follows:

  ProgOptions::ProgOptions(const char* optPrefix="–", const char* pkgPrefix="", bool allowUnknownOptions=false);

  where

  1. optPrefix is the prefix by means of which end user must marking out options recognizable (i.e. previously described)
  2. pkgPrefix is a substring for marking out special options "help" and "config" (see following for them)
  3. allowUnknownOptions is boolean adjusting precedence rule for the case of don't recognizable option(s) appear. The "true" is mind that the presence of don't recognizable options will be ignored while the "false" is mind that the parsing of command line will be interrupted and error message will be dispatched to stdandart error stream.

- Describe some options will be recognized and it's connected actions using method ProgOptions::put and set additional help messages using method ProgOptions::setAdditionalHelp

- Call ProgOptions::parse for parsing argv. After this step the command line divided to argv will be parsed, and options recognized will be placed to internal structures of ProgOptions, and all callback functions will be called.

- After this with usage of `ProgOptions::is_set("option")` you can check option setting. For options with argument you can read value of argument by using ProgOptions::argument("option")

Example:

```
#include <GradSoft/ProgOptions.h>  // first step

GradSoft::ProgOptions options;   // construct a new instance of ProgOptions:

void init()
{
 options.put("qqq", "qqq option", false );           // discribe the option "qqq"
                                                      // which not need arguments
 options.put("zzz", "option with argument", true );  // discribe the option "zzz"
                                                      // which need an argument

 options.setAdditionalHelp(true,                     //
   "This program illustate usage of GradSoft library"// set additional help messages
 );                                                   //
 options.setAdditionalHelp(false,                    //
   "and this is shown at the end of usage screen"    //
 );                                                   //
};

int main(int argc, char** argv)
{
 init();

 if (!options.parse(argc,argv)) return 1;      // parse options

 if (options.is_set("qqq")) {                  // find out, is the option seted
   cout << "qqq is set" << endl;
 }
                                                // find out, is the option seted
                                                // and get his argument if yes
 if (options.is_set("zzz")) {
   cout << "zzz is set with argument:" << options.argument("zzz") << endl;
 }
 return 0;
}
```

Now run this program:

```
./a.out --qqq --zzz zz-arg
qqq is set
zzz is set with argument:zz-arg
```

And now run with option not described:

```
./a.out --Go
a:Go:unknown option
```

2.1 Special options help and config

In addition to set of options, described by programmer, ProgOptions define two specially acting options: `--<pkgPrefix>help` and `--<pkg>config`. The result of calling program with option `--<pkgPrefix>help` is passing to stdandart error stream (cerr) a standard help message (which is the list of options with descriptions inputted on the step of using ProgOptions::put) added by additional help messages specified via ProgOptions::setAdditionalHelp. The result of calling program with `--<pkg>config fname` is parsing options described in the file `<fname>` (see next section 4.1 for more details).

Example:

Start program mentioned with option `--help`::

```
./a.out --help
This program illustate usage of ProgOptions library
  --qqq  qqq option
  --zzz <argument>  option with argument
and this is shown at the end of usage screen
```

Note: Using `--config <filename>` leads to some troubles for us when some part of options must be passed into "external" software, let us, into function ORB_init(). The point is that the ProgOptions::parse(argc,argv) method merge all options, given in argv vector and extracted from the file in especial argument vector being used then, and do not change starting argv. Therefore, if you call ORB_init() with argc and argv being parameters of main(), options from the file will be lost. Therefore, if you want to pass command-line options from the file into ORB_init() really, you must:

1. parse initial argument vector using "parse(argc,argv)";
2. pass to ORB_init() an argument vector and its length having been created during parsing.

At the same time, since an external software can remove options recognized by itself, it's the most correctly to give it a copy of argument vector created, and its length, in which purpose wrapper-class ProgOptions::ArgsHolder can be used:

```
ProgOptions options("--","",true);
if (!options.parse(argc,argv)) return 1;

ProgOptions::ArgsHolder argsHolder;
argsHolder.takeArgv(options);

CORBA::ORB_var orb = CORBA::ORB_init(argsHolder.argc,argsHolder.argv);
```

It is handly, because new argv created via argsHolder::takeArgv(const ProgOptions&) will be deleted automatically when destructor of argsHolder will be called.

Note: to date, syntax `--config=<filename>` do not possible for special option `--config`. Use `--config=<filename>` instead.

You can overload special option `--help` and `--config` using method ProgOption::put. Overloaded option has no feature and default action.

3 callback functions

Yet one way of handling options handling: callback functions. You can specify such function in parameter |callback| of method `ProgOptions::put` and it will be called during parsing appropriate option.

To illustrate this, let's add to our program next piece of code:

```
bool zz1Callback(const GradSoft::ProgOptions* options, const char* argument,
                 void* )
{
 cout << "zz1:callback called with argument " <<
            ((!argument) ? "NULL" : argument) << endl;
 return true;
}

void init()
{
 .....
 options.put("zz1", "option with argument and callback", true, zz1Callback );
};
```

Now run program with this option:

```
/a.out --zz1 xx
zz1:callback called with argument xx
```

Callback function is called during each occurence of option in command string, so you can organize handling of few identical options.

```
/a.out --zz1 xx --zz1 yy
zz1:callback called with argument xx
zz1:callback called with argument yy
```

4 Additional resources

4.1 Settiong of ProgOptions properties

- optPrefix, pkgPrefix (const char*)– this properties control syntax, for options, handled by ProgOptions: it looks for options in form: `<optPrefix><pkgPrefix><optionName>`.

  - Default value: `--` and empty string accordinally
  - Way of setting: parameters of constructor.

- allowUnknownOptions (bool) – if this property is set to true, than during parsing ProgOptions ignore elements of command string, which do not follow ProgOptions syntaxm or options, which was not set via ProgOptions::put. If this property is set to false, than ProgOptions::parse break parsing and return false on first occurence of such element.
  - Default value: false
  - Way of setting:
    - via parameter of constructor
    - `ProgOptions::setAllowUnknownOptions(bool);`
  - reading: `bool ProgOptions::getAllowUnknownOptions()`

4.2 Using of the config file

As it will be mentioned above, if you use ProgOptions::parse in you program, it will lead to options described in the file `fname` will be parsed if the end user of program apply `--config fname` in the command line.

The additional possibility is that: you may read options from file and parse them independently of end user choice using special method ProgOptions::parseFile(const char* configFname, const char* executable="unspecified"). For example, calling

```
options.parseFile("D:\Demo\config.ini");
```

(where options is instance of ProgOptions) means that the text in the `D:\Demo\config.ini` will be readed, parts of text recognized as "options" will be extracted and then interpreted as well as these options be in command line. (Second parameter of this method constitutes the name of the program asking for options and is optional: you may set it in the case of call of ProgOptions::parse not precede the call of ProgOptions::parseFile.)

The next mean allow you to combine all option parsed by ProgOptions::parse, ProgOptions::parseFile and ProgOptions::parseString (see following for the latter) and save these into the file specified using method `ProgOptions::saveToFile`. For example, using

```
ProgOptions options;
...
options.parse(argc,argv);
...
```

```
options.parseFile("D:\Demo\config.ini");
...
options.saveToFile("D:\Democonfig1.ini");
```

in continuous block leads to all options gotten by argv and readed from the file `D:\Demo\config.ini` (either recognized and not recognized) will be saved in formatted file `D:\Demo\config1.ini`, and will be obtained again if programmer use ProgOptions::parseFile.

It is important:

Bouth ProgOptions::parseFile and ProgOptions::saveToFile methods return boolean value, which indicate success or unsuccess of operatuion. In the case of ProgOptions::saveToFile, if the method return false, than the system error has been occured and error information is aviable via operation system errno interface. Thus, the standard code fragment for storing options in file looks like the next example:

```
if (!optios.saveToFile(configFname)) {
   perror(options.argv(0));
   ....
}
....
```

In the case of ProgOptions::parseFile the precidence rule analogous is not valid because ProgOptions::parseFile return false in few different cases:

1. system error occur,
2. config file formated incorrectly,
3. parsing of options is impossible.

If the system error or bad format error occur, ProgOptions::parseFile signals about situation by means of message which dispatchs to standard error stream.

4.3 Format of ProgOptions config file

The file to parse must been formatted:

1. The file must being a text formed by meaningful words, symbol-separators and comments. The options to parse are meaningful words only.
2. The (single) meaningful word is <sequence>[<sequence>...] where <sequence> is unbroken sequence of "visible" characters, sequence of visible characters, spaces and tabulators in double inverted commas, or sequence of sequences described.

   Example:

```
   aaa_bbb "aaa bbb" aaa" bbb ccc"
```

will be interpreted as three separate words `'aaa_bbb'`, `'aaa bbb'` and `'aaa bbb ccc'`

3. Comments are:

   (a) markers `'/*'` and `'*/'` and every characters standing between them;

   (b) marker `'//'` and part of string after it;

   (c) the string begining from the `'#'`.

   Examples:

   ```
   # This string will be ignored

     aaa /* This part of string will be ignored */ bbb

     ccc // This part of string will be ignored too
   ```

4. The file must contain a marker being @"ProgOptions config file" or "@ProgOptions config file" (or analogous) as the first meaningful word in the first 10000 bytes of the file (hence, the first string of file could not be empty)

5. Effects:

   (a) if the commas is not closed, the bounds of word is end of line;

   (b) if the double comma must be member of meanigful word, it may be used as '\"'.

4.4 Trivial example of config file

The trivial example which illustrate the structure of ProgOptions config file is follow:

```
@"ProgOptions config file"

# The list of options:

    --a          /* single option */
    --b c        // option with argument
    --d="a b c"  // another option with argument
```

4.5 Usage of ProgOptions::parseString

The next additional possibility is that you may parse the string in C-style with result the same as string has been readed from file. The methodv ProgOptions::parseFile(const char* configFname, const char* executable="unspecified") do it. The small difference in acting of ProgOptions::parseFile and ProgOptions::parseString methods is that the file marker `'@ProgOptions Config File'` and symbol `'#'` at the begining of string must not have and have any special sence in latter case.

5 Programming environment conventions

Using Progoptions on Windows NT, you must:

1. to define WIN32 macro before inclusion of ProgOptions.h header file;
2. to use iostream, fstream, etc. standard headers instead iostream.h, fstream.h, etc. ones.

6 Changes

31.01.2002 - support of using `--config <filename>` option described

24.01.2002 - warning concerning impossibility of `--config=<filename>` syntax for non-overloaded option `--config` added.

03.01.2002 - next points updated in accordance with version 1.4.0 of ToolBox:

1. example in section 2 (include file `GradSoft/ProgOptions` don't used from date);
2. descripton of configuration file parsing procedure in section 4.3 (the way to use '//','/*', and '*/' constructs in option body arised);
3. environment conventions in section 5 ("old" standard headers such as iostream.h, fstream.h and so on must not be used from date).

17.02.2001 - corrections, formal attributes of documentation set is added, susection abbout properties is added.

29.06.2000 - initial version.