arg, a C++ command-line parser

Chun-Chung Chen

May 7, 2010

Version: 1.0
URL: http://ccdw.org/~cjj/prog/arg/

Contents

1 A simple example

To use the **arg** parser, you need to: 1. include the header file "`arg.hh`" in your program; 2. create a parser object ("`arg::Parser`"); 3. add options to the parser object; 4. pass command-line data to the parser object. An example of simple programs using the **arg** parser as follows.

```
#include <arg.hh> // 1.
#include <iostream>

int main(int argc, char ** argv)
{
        arg::Parser p; // 2.
```

```
        int n;
        p.add_opt('n').stow(n); // 3.

        p.parse(argc, argv); // 4.

        std::cout << n << '\n';
        return 0;
}
```

This allows you to pass, on command line, an integer to the variable "`n`" in the program. This example is included in arg package as "`arg_ex0.cc`". You can compile it with the command:

```
$ c++ arg_ex0.cc arg.cc -I. -o arg_ex0
```

to produce the executable "`arg_ex0`" without installing the library.

2 Option properties

All the work required to do is adding various options to the parser. This is accomplished by the "`add_opt`" function, which returns the reference to the added option ("`arg::Option &`"). We can modify the properties of an option by calling its member functions. All the property modifying member functions of an option will return the reference of the same option which allow us to chain up these function calls. (This chaining of function call is a hack around C++'s lack of named arguments in function calls.)

Each option can have a value string immediately following a short option switch (with or without separating space, *e.g.*, "`-n10`" or "`-n 10`") or attached to a long option switch by "`=`" character (, *e.g.*, "`--num=10`"). Currently supported modifiers are list in Table 1 and described below.

`template<typename T> Option &stow(T &t);`

Stow the value to the variable "`t`" when the option is invoked. The type of "`t`" need to be streamable, supporting the "put-to" and "get-from" operators: "<<" and ">>".

`Option &store(Value *ptr = 0);`

Pass a "`Value`" object (pointed to by "`ptr`") to the option. The "`Value`" object will be deleted when the option is destructed. The "`Value`" object is used to process the value string when the option is invoked.

`Option &optional(const string &str = "");`

Make the value string of the option optional and use "`str`" instead when the value string is not supplied with the invocation of the option.

Table 1: Supported modifiers to an "`Option`" of the parser.

Modifier	Parameters	Description
`stow<T>`	`T &t`	stow the value to variable "`t`"
`store`	`Value *ptr`	pass "`Option`" a "`Value`" object to take the value string
`optional`	`const string &str`	value string is optional, defaulting to "`str`"
`set`	`int *var` `int value`	set "`*var`" to "`value`" when the option is invoked
`once`	`int init`	only allow the "`*var`" in "`set`" modifier to be set when its value is "`init`"
`call`	`CallBack *func` `void *data`	call the function "`*func`" to process the value string and pass it extra "`data`"
`help`	`const string &text` `const string &var`	supply a description "`text`" for the option and refer to the value string as "`var`"
`help_word`	`const string &var`	refer to the value string as "`var`"
`show_default`	`bool do_show`	show the default value in description

`Option &set(int *var, int value = - 1);`

Set the integer variable "`*var`" to "`value`" when the option is invoked.

`Option &once(int init = 0);`

Allow setting of "`*var`" only when its value is "`init`". This can be used to prevent multiple invocations of one option or enforce mutual exclusion between different options.

`Option &call(CallBack *func, void *data);`

Call the function "`*func`" with the value string and "`data`" when the option is invoked. The extra parameter "`data`" allows one to, for example, relay the callback to a member function of an object.

`Option &help(const string &text, const string &var = "");`

Attach a description "`text`" to the option while referring to its value string as "`var`". This can be used to produce usage information for the program.

`Option &help_word(const string &var);`

Supply a simple word "`var`" describing the value string.

`Option &show_default(bool do_show = true);`

Show the initial or default value of the variable referred in "`stow`" or "`store`" modifiers. This will be the value the program assumes when the option is not invoked.

3 Adding help and a complex example

The descriptions supplied by the "`help`" or "`help_word`" modifiers of "`Option`" can be extracted per option by the "`Option::get_help`" member function or as a whole by the "`Parser::get_help`" method of the parser in the order that options were added. Additional messages can be added to the text returned by "`Parser::get_help`" through interlacing the option addition process with the "`add_help`" method calls to the parser. One can setup customized mechanism to display the help text when it's desired. However, for simple applications, it should suffice to call "`Parser::add_opt_help`" to add a standard help option to the parser. Before the text return by "`Parser::get_help`", the standard help option also prints a header text that can be set by "`Parser::set_header`".

A example program demonstrates the usage of the standard help option is as follows.

```
#include <arg.hh>
#include <iostream>
using namespace std;
const string version = "1.0";
int main(int argc, char ** argv)
{
        arg::Parser parser;
        parser.set_header("arg Testing Program v" + version);
        parser.add_help("");
        parser.add_help("available options are:");

        int n = 10;
        parser.add_opt('n', "number").stow(n)
                .help("set number of nodes to INT", "INT")
                .show_default();

        string f;
        int f_given;
        f_given = 0;
        parser.add_opt('i', "input").stow(f)
                .help("read data from FILE", "FILE")
                .set(& f_given).once();

        parser.add_opt_help();
        parser.add_opt_version(version);
        // parse command line
        try {
                parser.parse(argc, argv);
        }
        catch (arg::Error e) {
                cout << "Error parsing command line: "
                         << e.get_msg() << '\n';
                return 1;
        }
        // check for parameter consistency
        if (! f.size()) {
                cout << "Need to specify the input file!\n";
                return 1;
        }
        // output
        cout << "The parameters are:\n"
                << "number = " << n << '\n'
                << "input = " << f << '\n';
        return 0;
}
```

Users of the program can invoke the "-h" switch to get a list of available options for the program as follows.

```
$ ./arg_ex1 -h
arg Testing Program v1.0
```

```
available options are:
  -n, --number=INT            set number of nodes to INT (
      default: 10)
  -i, --input=FILE            read data from FILE
  -h, --help                  display this help list and exit
  -V, --version               print program version and exit
```

Another standard option used in the above example is the "`--version`" option that can be added to the parser with "`Parser::add_opt_version`" method.

Localization A theme in the design of **arg** is to localize all information related to a command line option. That is demonstrated in the last example where we put codes related to each parameter into a single block. This save one from hunting all over the place when, say, just making change to a single parameter.

Consistency There is no simple way to provide a general mechanism that can specify and check the consistency for the supplied command-line options that's not itself Turing-complete. The only assistance provide by the **arg** parser are the "`set`" modifier that can be used to track, *e.g.*, if an option was invoked and the "`once`" modifier that can be used to, say, prevent multiple invocations of an option or enforce mutual exclusion between options.

Exception All errors occurs in parsing the command-line parameters result in exceptions of the base type "`arg::Error`". The corresponding message can be obtained by the "`Error::get_msg`" method of the exception.

4 Callback function

One can specify a callback function to be called upon the invocation of an option and to process the value string if there is any. A callback function "`func`" has the signature:

```
bool func(int key, const std::string &vstr, void *data);
```

The arguments passed to the callback are the short option "`key`", the value string "`vstr`", and the extra "`data`". The return value of the callback function is used to indicate if the processing of the option is successful. A "`false`" return value of the callback function will result in an "`OptError`" exception.

The extra "`data`" parameter allows one to parametrize the callback function. For example, if we pass the pointer to an object as the extra "`data`",

```
Kitchen kitc;
parser.add_opt("vegi").call(& func, & kitc);
```

we can setup a callback function that will relays the call to a member function of the object:

```
bool func(int key, const std::string &vstr, void *data)
{
        Kitchen *k = static_cast<Type *>(data);
        return k->cook(vstr);
}
```

On a side note, if we were not concerned with additional dependency, this callback mechanism should probably have been replaced by making use of signal-and-handler libraries, such as, libsigc++.

5 Subparser

Sometimes, the value string of an option represents some suboptions and should be processed by a subparser. The "`arg::SubParser`" is both a "`Parser`" and a "`Value`". It should be attached to an option of the parser using the "`Parser::store`" modifier. As a consequence, we need to allocate the subparser dynamically so that it can be safely deleted by the destructor of the option. The default separator of the suboptions is the comma "`,`" but can be changed with "`SubParser::set_sep`" method. For example, with the following code:

```
Parser p;
SubParser * sp = new SubParser;
int va1 = 0;
sp->add_opt("param1").stow(va1);
double va2 = 1.0;
sp->add_opt("param2").stow(va2);
sp->add_opt_help();
parser.add_opt('o', "options").store(sp);
```

We can pass values to va1 and va2 on the command line like:

```
$ ./arg_ex -o param1=2,param2=0.5
```

Or, obtain a list of supported suboptions with:

```
$ ./arg_ex -o help
```

6 Value class

A "`Value`" object is a mechanism to process a value string and store it somewhere, *e.g.*, a variable, as well as to produce a string representation of the stored value. They should be dynamically allocated before passing to the option and will be deleted by the option upon destruction. Beside the "`StreamableValue`" subclass used by the "`Option::stow`" modifier to handle storage to types that support put-to and get-from stream operations (through "<<" and ">>" operators), additional value classes are available in "`val.hh`". This currently includes "`SetValue`" that represents storage to a choice from a set of names;

"`ListValue`" that represents storage to separator (defaulting to comma) separated list of values of a given type; and "`RelValue`" that can be used to either set or perform relative change to a "`double`" type variable.

7 Non-option arguments

Currently, command line arguments that do not start with "-" or "`--`" and are not value strings of any options are swept into a "`std::vector<std::string>`" array that can be access through the member function "`Parser::args`". In future releases of **arg**, new mechanism might be built to process them as an ordered list of different variable values.

8 Information

This parser was developed from scratch for some projects in scientific computation in 2004. The aim is to minimize the coding efforts of adding command-line parameters to C++ programs. Earlier version of the code was released as part of ccGo since 2005 under the GPL license. The current standalone version is released under LGPL and can be downloaded from the information page at http://ccdw.org/~cjj/prog/arg/. Please send any comments and bug reports to <cjj@u.washington.edu>.