CommandLine 2.0 Library Manual

Written by Chris Lattner

Introduction

This document describes the CommandLine argument processing library. It will show you how to use it, and what it can do. The CommandLine library uses a declarative approach to specifying the command line options that your program takes. By default, these options declarations implicitly hold the value parsed for the option declared (of course this can be changed).

Although there are a **lot** of command line argument parsing libraries out there in many different languages, none of them fit well with what I needed. By looking at the features and problems of other libraries, I designed the CommandLine library to have the following features:

1. Speed: The CommandLine library is very quick and uses little resources. The parsing time of the library is directly proportional to the number of arguments parsed, not the the number of options recognized.

Additionally, command line argument values are captured transparently into user defined global variables, which can be accessed like any other variable (and with the same performance).
2. Type Safe: As a user of CommandLine, you don't have to worry about remembering the type of arguments that you want (is it an int? a string? a bool? an enum?) and keep casting it around. Not only does this help prevent error prone constructs, it also leads to dramatically cleaner source code.

3. No subclasses required: To use CommandLine, you instantiate variables that correspond to the arguments that you would like to capture, you don't subclass a parser. This means that you don't have to write **any** boilerplate code.

4. Globally accessible: Libraries can specify command line arguments that are automatically enabled in any tool that links to the library. This is possible because the application doesn't have to keep a "list" of arguments to pass to the parser. This also makes supporting dynamically loaded options trivial.

5. Cleaner: CommandLine supports enum and other types directly, meaning that there is less error and more security built into the library. You don't have to worry about whether your integral command line argument accidentally got assigned a value that is not valid for your enum type.

6. Powerful: The CommandLine library supports many different types of arguments, from simple boolean flags to scalars arguments (strings, integers, enums, doubles), to lists of arguments. This is possible because CommandLine is...

7. Extensible: It is very simple to add a new argument type to CommandLine. Simply specify the parser that you want to use with the command line option when you declare it. Custom parsers are no problem.

8. Labor Saving: The CommandLine library cuts down on the amount of grunt work that you, the user, have to do. For example, it automatically provides a `--help` option that shows the available command line options for your tool. Additionally, it does most of the basic correctness checking for you.

9. Capable: The CommandLine library can handle lots of different forms of options often found in real programs. For example, positional arguments, `ls` style grouping options (to allow processing '`ls -lad`' naturally), `ld` style prefix options (to parse '`-lmalloc -L/usr/lib`'), and interpreter style options.

This document will hopefully let you jump in and start using CommandLine in your utility quickly and painlessly. Additionally it should be a simple reference manual to figure out how stuff works. If it is failing in some area (or you want an extension to the library), nag the author, Chris Lattner.

Quick Start Guide

This section of the manual runs through a simple CommandLine'ification of a basic compiler tool. This is intended to show you how to jump into using the CommandLine library in your own program, and show you some of the cool things it can do.

To start out, you need to include the CommandLine header file into your program:

```
  #include "Support/CommandLine.h"
```

Additionally, you need to add this as the first line of your main program:

```
int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv);
  ...
}
```

... which actually parses the arguments and fills in the variable declarations.

Now that you are ready to support command line arguments, we need to tell the system which ones we want, and what type of argument they are. The CommandLine library uses a declarative syntax to model command line arguments with the global variable declarations that capture the parsed values. This means that for every

command line option that you would like to support, there should be a global variable declaration to capture the result. For example, in a compiler, we would like to support the unix standard '`-o <filename>`' option to specify where to put the output. With the CommandLine library, this is represented like this:

```
cl::opt<string> OutputFilename("o", cl::desc("Specify output filename"), cl::value_desc("filename"));
```

This declares a global variable "`OutputFilename`" that is used to capture the result of the "`o`" argument (first parameter). We specify that this is a simple scalar option by using the "`cl::opt`" template (as opposed to the "`cl::list` template), and tell the CommandLine library that the data type that we are parsing is a string.

The second and third parameters (which are optional) are used to specify what to output for the "`--help`" option. In this case, we get a line that looks like this:

```
USAGE: compiler [options]

OPTIONS:
  -help             - display available options (--help-hidden for more)
  -o <filename>     - Specify output filename
```

Because we specified that the command line option should parse using the `string` data type, the variable declared is automatically usable as a real string in all contexts that a normal C++ string object may be used. For example:

```
  ...
  ofstream Output(OutputFilename.c_str());
  if (Out.good()) ...
  ...
```

There are many different options that you can use to customize the command line option handling library, but the above example shows the general interface to these options. The options can be specified in any order, and are specified with helper functions like `cl::desc(...)`, so there are no positional dependencies to remember. The available options are discussed in detail in the Reference Guide.

Continuing the example, we would like to have our compiler take an input filename as well as an output filename, but we do not want the input filename to be specified with a hyphen (ie, not `-filename.c`). To support this style of argument, the CommandLine library allows for positional arguments to be specified for the program. These positional arguments are filled with command line parameters that are not in option form. We use this feature like this:

```
cl::opt<string> InputFilename(cl::Positional, cl::desc("<input file>"), cl::init("-"));
```

This declaration indicates that the first positional argument should be treated as the input filename. Here we use the `cl::init` option to specify an initial value for the command line option, which is used if the option is not specified (if you do not specify a `cl::init` modifier for an option, then the default constructor for the data type is used to initialize the value). Command line options default to being optional, so if we would like to require that the user always specify an input filename, we would add the `cl::Required` flag, and we could eliminate the `cl::init` modifier, like this:

```
cl::opt<string> InputFilename(cl::Positional, cl::desc("<input file>"), cl::Required);
```

Again, the CommandLine library does not require the options to be specified in any particular order, so the above declaration is equivalent to:

```
cl::opt<string> InputFilename(cl::Positional, cl::Required, cl::desc("<input file>"));
```

By simply adding the `cl::Required` flag, the CommandLine library will automatically issue an error if the argument is not specified, which shifts all of the command line option verification code out of your application into the library. This is just one example of how using flags can alter the default behaviour of the library, on a per-option basis. By adding one of the declarations above, the `--help` option synopsis is now extended to:

```
USAGE: compiler [options] <input file>

OPTIONS:
  -help             - display available options (--help-hidden for more)
  -o <filename>     - Specify output filename
```

... indicating that an input filename is expected.

Boolean Arguments

In addition to input and output filenames, we would like the compiler example to support three boolean flags: "`-f`" to force overwriting of the output file, "`--quiet`" to enable quiet mode, and "`-q`" for backwards compatibility with some of our users. We can support these by declaring options of boolean type like this:

```
cl::opt<bool> Force ("f", cl::desc("Overwrite output files"));
cl::opt<bool> Quiet ("quiet", cl::desc("Don't print informational messages"));
cl::opt<bool> Quiet2("q", cl::desc("Don't print informational messages"), cl::Hidden);
```

This does what you would expect: it declares three boolean variables ("`Force`", "`Quiet`", and "`Quiet2`") to recognize these options. Note that the "`-q`" option is specified with the "`cl::Hidden`" flag. This modifier prevents it from being shown by the standard "`--help`" output (note that it is still shown in the "`--help-hidden`" output).

The CommandLine library uses a different parser for different data types. For example, in the string case, the argument passed to the option is copied literally into the content of the string variable... we obviously cannot do that in the boolean case, however, so we must use a smarter parser. In the case of the boolean parser, it allows no options (in which case it assigns the value of true to the variable), or it allows the values "`true`" or "`false`" to be specified, allowing any of the following inputs:

```
 compiler -f          # No value, 'Force' == true
 compiler -f=true     # Value specified, 'Force' == true
 compiler -f=TRUE     # Value specified, 'Force' == true
 compiler -f=FALSE    # Value specified, 'Force' == false
```

... you get the idea. The bool parser just turns the string values into boolean values, and rejects things like '`compiler -f=foo`'. Similarly, the float, double, and int parsers work like you would expect, using the '`strtol`' and '`strtod`' C library calls to parse the string value into the specified data type.

With the declarations above, "`compiler --help`" emits this:

```
USAGE: compiler [options] <input file>

OPTIONS:
  -f     - Overwrite output files
  -o     - Override output filename
  -quiet - Don't print informational messages
  -help  - display available options (--help-hidden for more)
```

and "`opt --help-hidden`" prints this:

```
USAGE: compiler [options] <input file>

OPTIONS:
  -f     - Overwrite output files
  -o     - Override output filename
  -q     - Don't print informational messages
  -quiet - Don't print informational messages
  -help  - display available options (--help-hidden for more)
```

This brief example has shown you how to use the '`cl::opt`' class to parse simple scalar command line arguments. In addition to simple scalar arguments, the CommandLine library also provides primitives to support CommandLine option aliases, and lists of options.

Argument Aliases

So far, the example works well, except for the fact that we need to check the quiet condition like this now:

```
...
  if (!Quiet && !Quiet2) printInformationalMessage(...);
...
```

... which is a real pain! Instead of defining two values for the same condition, we can use the "cl::alias" class to make the "-q" option an **alias** for the "-quiet" option, instead of providing a value itself:

```
cl::opt<bool> Force ("f", cl::desc("Overwrite output files"));
cl::opt<bool> Quiet ("quiet", cl::desc("Don't print informational messages"));
cl::alias     QuietA("q", cl::desc("Alias for -quiet"), cl::aliasopt(Quiet));
```

The third line (which is the only one we modified from above) defines a "-q alias that updates the "Quiet" variable (as specified by the cl::aliasopt modifier) whenever it is specified. Because aliases do not hold state, the only thing the program has to query is the Quiet variable now. Another nice feature of aliases is that they automatically hide themselves from the -help output (although, again, they are still visible in the --help-hidden output).

Now the application code can simply use:

```
...
  if (!Quiet) printInformationalMessage(...);
...
```

... which is much nicer! The "cl::alias" can be used to specify an alternative name for any variable type, and has many uses.

Selecting an alternative from a set of possibilities

So far, we have seen how the CommandLine library handles builtin types like std::string, bool and int, but how does it handle things it doesn't know about, like enums or 'int*'s?

The answer is that it uses a table driven generic parser (unless you specify your own parser, as described in the Extension Guide). This parser maps literal strings to whatever type is required, are requires you to tell it what this mapping should be.

Lets say that we would like to add four optimizations levels to our optimizer, using the standard flags "-g", "-O0", "-O1", and "-O2". We could easily implement this with boolean options like above, but there are several problems with this strategy:

1. A user could specify more than one of the options at a time, for example, "opt -O3 -O2". The CommandLine library would not be able to catch this erroneous input for us.
2. We would have to test 4 different variables to see which ones are set.
3. This doesn't map to the numeric levels that we want... so we cannot easily see if some level >= "-O1" is enabled.

To cope with these problems, we can use an enum value, and have the CommandLine library fill it in with the appropriate level directly, which is used like this:

```
enum OptLevel {
  g, O1, O2, O3
};

cl::opt<OptLevel> OptimizationLevel(cl::desc("Choose optimization level:"),
  cl::values(
    clEnumVal(g , "No optimizations, enable debugging"),
```

```
    clEnumVal(O1, "Enable trivial optimizations"),
    clEnumVal(O2, "Enable default optimizations"),
    clEnumVal(O3, "Enable expensive optimizations"),
   0));

...
  if (OptimizationLevel >= O2) doPartialRedundancyElimination(...);
...
```

This declaration defines a variable "`OptimizationLevel`" of the "`OptLevel`" enum type. This variable can be assigned any of the values that are listed in the declaration (Note that the declaration list must be terminated with the "`0`" argument!). The CommandLine library enforces that the user can only specify one of the options, and it ensure that only valid enum values can be specified. The "`clEnumVal`" macros ensure that the command line arguments matched the enum values. With this option added, our help output now is:

```
USAGE: compiler [options] <input file>

OPTIONS:
  Choose optimization level:
    -g          - No optimizations, enable debugging
    -O1         - Enable trivial optimizations
    -O2         - Enable default optimizations
    -O3         - Enable expensive optimizations
  -f            - Overwrite output files
  -help         - display available options (--help-hidden for more)
  -o <filename> - Specify output filename
  -quiet        - Don't print informational messages
```

In this case, it is sort of awkward that flag names correspond directly to enum names, because we probably don't want a enum definition named "`g`" in our program. Because of this, we can alternatively write this example like this:

```
enum OptLevel {
  Debug, O1, O2, O3
};

cl::opt<OptLevel> OptimizationLevel(cl::desc("Choose optimization level:"),
  cl::values(
   clEnumValN(Debug, "g", "No optimizations, enable debugging"),
    clEnumVal(O1        , "Enable trivial optimizations"),
    clEnumVal(O2        , "Enable default optimizations"),
    clEnumVal(O3        , "Enable expensive optimizations"),
   0));

...
  if (OptimizationLevel == Debug) outputDebugInfo(...);
...
```

By using the "`clEnumValN`" macro instead of "`clEnumVal`", we can directly specify the name that the flag should get. In general a direct mapping is nice, but sometimes you can't or don't want to preserve the mapping, which is when you would use it.

Named Alternatives

Another useful argument form is a named alternative style. We shall use this style in our compiler to specify different debug levels that can be used. Instead of each debug level being its own switch, we want to support the following options, of which only one can be specified at a time: "`--debug-level=none`", "`--debug-level=quick`", "`--debug-level=detailed`". To do this, we use the exact same format as our optimization level flags, but we also specify an option name. For this case, the code looks like this:

```
enum DebugLev {
  nodebuginfo, quick, detailed
};
```

```
// Enable Debug Options to be specified on the command line
cl::opt<DebugLev> DebugLevel("debug_level", cl::desc("Set the debugging level:"),
  cl::values(
    clEnumValN(nodebuginfo, "none", "disable debug information"),
     clEnumVal(quick,               "enable quick debug information"),
     clEnumVal(detailed,            "enable detailed debug information"),
    0));
```

This definition defines an enumerated command line variable of type "`enum DebugLev`", which works exactly the same way as before. The difference here is just the interface exposed to the user of your program and the help output by the "`--help`" option:

```
USAGE: compiler [options] <input file>

OPTIONS:
  Choose optimization level:
    -g          - No optimizations, enable debugging
    -O1         - Enable trivial optimizations
    -O2         - Enable default optimizations
    -O3         - Enable expensive optimizations
  -debug_level  - Set the debugging level:
    =none       - disable debug information
    =quick      - enable quick debug information
    =detailed   - enable detailed debug information
  -f            - Overwrite output files
  -help         - display available options (--help-hidden for more)
  -o <filename> - Specify output filename
  -quiet        - Don't print informational messages
```

Again, the only structural difference between the debug level declaration and the optimiation level declaration is that the debug level declaration includes an option name ("`debug_level`"), which automatically changes how the library processes the argument. The CommandLine library supports both forms so that you can choose the form most appropriate for your application.

Parsing a list of options

Now that we have the standard run of the mill argument types out of the way, lets get a little wild and crazy. Lets say that we want our optimizer to accept a **list** of optimizations to perform, allowing duplicates. For example, we might want to run: "`compiler -dce -constprop -inline -dce -strip`". In this case, the order of the arguments and the number of appearances is very important. This is what the "`cl::list`" template is for. First, start by defining an enum of the optimizations that you would like to perform:

```
enum Opts {
  // 'inline' is a C++ keyword, so name it 'inlining'
  dce, constprop, inlining, strip
};
```

Then define your "`cl::list`" variable:

```
cl::list<Opts> OptimizationList(cl::desc("Available Optimizations:"),
  cl::values(
    clEnumVal(dce               , "Dead Code Elimination"),
    clEnumVal(constprop         , "Constant Propagation"),
   clEnumValN(inlining, "inline", "Procedure Integration"),
    clEnumVal(strip             , "Strip Symbols"),
  0));
```

This defines a variable that is conceptually of the type "`std::vector<enum Opts>`". Thus, you can access it with standard vector methods:

```
  for (unsigned i = 0; i != OptimizationList.size(); ++i)
    switch (OptimizationList[i])
```

```
  ...
```

... to iterate through the list of options specified.

Note that the "cl::list" template is completely general and may be used with any data types or other arguments that you can use with the "cl::opt" template. One especially useful way to use a list is to capture all of the positional arguments together if there may be more than one specified. In the case of a linker, for example, the linker takes several '.o' files, and needs to capture them into a list. This is naturally specified as:

```
...
cl::list<std::string> InputFilenames(cl::Positional, cl::desc("<Input files>"), cl::OneOrMore);
...
```

This variable works just like a "vector<string>" object. As such, accessing the list is simple, just like above. In this example, we used the cl::OneOrMore modifier to inform the CommandLine library that it is an error if the user does not specify any .o files on our command line. Again, this just reduces the amount of checking we have to do.

Adding freeform text to help output

As our program grows and becomes more mature, we may decide to put summary information about what it does into the help output. The help output is styled to look similar to a Unix man page, providing concise information about a program. Unix man pages, however often have a description about what the program does. To add this to your CommandLine program, simply pass a third argument to the cl::ParseCommandLineOptions call in main. This additional argument is then printed as the overview information for your program, allowing you to include any additional information that you want. For example:

```
int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv, " CommandLine compiler example\n\n"
                              "  This program blah blah blah...\n");
  ...
}
```

Would yield the help output:

```
OVERVIEW: CommandLine compiler example

  This program blah blah blah...

USAGE: compiler [options] <input file>

OPTIONS:
  ...
  -help             - display available options (--help-hidden for more)
  -o <filename>     - Specify output filename
```

Reference Guide

Now that you know the basics of how to use the CommandLine library, this section will give you the detailed information you need to tune how command line options work, as well as information on more "advanced" command line option processing capabilities.

Positional Arguments

Positional arguments are those arguments that are not named, and are not specified with a hyphen. Positional arguments should be used when an option is specified by its position alone. For example, the standard Unix grep tool takes a regular expression argument, and an optional filename to search through (which defaults to standard input if a filename is not specified). Using the CommandLine library, this would be specified as:

```
cl::opt<string> Regex   (cl::Positional, cl::desc("<regular expression>"), cl::Required);
cl::opt<string> Filename(cl::Positional, cl::desc("<input file>"), cl::init("-"));
```

Given these two option declarations, the `--help` output for our grep replacement would look like this:

```
USAGE: spiffygrep [options] <regular expression> <input file>

OPTIONS:
  -help - display available options (--help-hidden for more)
```

... and the resultant program could be used just like the standard `grep` tool.

Positional arguments are sorted by their order of construction. This means that command line options will be ordered according to how they are listed in a .cpp file, but will not have an ordering defined if they positional arguments are defined in multiple .cpp files. The fix for this problem is simply to define all of your positional arguments in one .cpp file.

Specifying positional options with hyphens

Sometimes you may want to specify a value to your positional argument that starts with a hyphen (for example, searching for '`-foo`' in a file). At first, you will have trouble doing this, because it will try to find an argument named '`-foo`', and will fail (and single quotes will not save you). Note that the system `grep` has the same problem:

```
$ spiffygrep '-foo' test.txt
Unknown command line argument '-foo'.  Try: spiffygrep --help'

$ grep '-foo' test.txt
grep: illegal option -- f
grep: illegal option -- o
grep: illegal option -- o
Usage: grep -hblcnsviw pattern file . . .
```

The solution for this problem is the same for both your tool and the system version: use the '`--`' marker. When the user specifies '`--`' on the command line, it is telling the program that all options after the '`--`' should be treated as positional arguments, not options. Thus, we can use it like this:

```
$ spiffygrep -- -foo test.txt
  ...output...
```

The `cl::ConsumeAfter` modifier

The `cl::ConsumeAfter` formatting option is used to construct programs that use "interpreter style" option processing. With this style of option processing, all arguments specified after the last positional argument are treated as special interpreter arguments that are not interpreted by the command line argument.

As a concrete example, lets say we are developing a replacement for the standard Unix Bourne shell (`/bin/sh`). To run `/bin/sh`, first you specify options to the shell itself (like `-x` which turns on trace output), then you specify the name of the script to run, then you specify arguments to the script. These arguments to the script are parsed by the bourne shell command line option processor, but are not interpreted as options to the shell itself. Using the CommandLine library, we would specify this as:

```
cl::opt<string> Script(cl::Positional, cl::desc("<input script>"), cl::init("-"));
cl::list<string>  Argv(cl::ConsumeAfter, cl::desc("<program arguments>..."));
cl::opt<bool>    Trace("x", cl::desc("Enable trace output"));
```

which automatically provides the help output:

```
USAGE: spiffysh [options] <input script> <program arguments>...

OPTIONS:
  -help - display available options (--help-hidden for more)
  -x    - Enable trace output
```

At runtime, if we run our new shell replacement as '`spiffysh -x test.sh -a -x -y bar`', the `Trace` variable will be set to true, the `Script` variable will be set to "`test.sh`", and the `Argv` list will contain `["-a", "-x", "-y", "bar"]`, because they were specified after the last positional argument (which is the script name).

There are several limitations to when `cl::ConsumeAfter` options can be specified. For example, only one `cl::ConsumeAfter` can be specified per program, there must be at least one positional argument specified, and the `cl::ConsumeAfter` option should be a cl::list option.

Internal vs External Storage

By default, all command line options automatically hold the value that they parse from the command line. This is very convenient in the common case, especially when combined with the ability to define command line options in the files that use them. This is called the internal storage model.

Sometimes, however, it is nice to separate the command line option processing code from the storage of the value parsed. For example, lets say that we have a '`-debug`' option that we would like to use to enable debug information across the entire body of our program. In this case, the boolean value controlling the debug code should be globally accessable (in a header file, for example) yet the command line option processing code should not be exposed to all of these clients (requiring lots of .cpp files to #include `CommandLine.h`).

To do this, set up your .h file with your option, like this for example:

```
// DebugFlag.h - Get access to the '-debug' command line option
//

// DebugFlag - This boolean is set to true if the '-debug' command line option
// is specified.  This should probably not be referenced directly, instead, use
// the DEBUG macro below.
//
extern bool DebugFlag;

// DEBUG macro - This macro should be used by code to emit debug information.
// In the '-debug' option is specified on the command line, and if this is a
// debug build, then the code specified as the option to the macro will be
// executed.  Otherwise it will not be.  Example:
//
// DEBUG(cerr << "Bitset contains: " << Bitset << "\n");
//
#ifdef NDEBUG
#define DEBUG(X)
#else
#define DEBUG(X) \
  do { if (DebugFlag) { X; } } while (0)
#endif
```

This allows clients to blissfully use the `DEBUG()` macro, or the `DebugFlag` explicitly if they want to. Now we just need to be able to set the `DebugFlag` boolean when the option is set. To do this, we pass an additial argument to our command line argument processor, and we specify where to fill in with the cl::location attribute:

```
bool DebugFlag;      // the actual value
static cl::opt<bool, true>       // The parser
Debug("debug", cl::desc("Enable debug output"), cl::Hidden,
      cl::location(DebugFlag));
```

In the above example, we specify "`true`" as the second argument to the cl::opt template, indicating that the template should not maintain a copy of the value itself. In addition to this, we specify the cl::location attribute,

so that `DebugFlag` is automatically set.

Option Attributes

This section describes the basic attributes that you can specify on options.

- The option name attribute (which is required for all options, except positional options) specifies what the option name is. This option is specified in simple double quotes:

  ```
  cl::opt<bool> Quiet("quiet");
  ```

- The **`cl::desc`** attribute specifies a description for the option to be shown in the `--help` output for the program.

- The **`cl::value_desc`** attribute specifies a string that can be used to fine tune the `--help` output for a command line option. Look here for an example.

- The **`cl::init`** attribute specifies an inital value for a scalar option. If this attribute is not specified then the command line option value defaults to the value created by the default constructor for the type. **Warning**: If you specify both **`cl::init`** and **`cl::location`** for an option, you must specify **`cl::location`** first, so that when the command-line parser sees **`cl::init`**, it knows where to put the initial value. (You will get an error at runtime if you don't put them in the right order.)

- The **`cl::location`** attribute where to store the value for a parsed command line option if using external storage. See the section on Internal vs External Storage for more information.

- The **`cl::aliasopt`** attribute specifies which option a cl::alias option is an alias for.

- The **`cl::values`** attribute specifies the string-to-value mapping to be used by the generic parser. It takes a **null terminated** list of (option, value, description) triplets that specify the option name, the value mapped to, and the description shown in the `--help` for the tool. Because the generic parser is used most frequently with enum values, two macros are often useful:

  1. The **`clEnumVal`** macro is used as a nice simple way to specify a triplet for an enum. This macro automatically makes the option name be the same as the enum name. The first option to the macro is the enum, the second is the description for the command line option.

  2. The **`clEnumValN`** macro is used to specify macro options where the option name doesn't equal the enum name. For this macro, the first argument is the enum value, the second is the flag name, and the second is the description.

  You will get a compile time error if you try to use cl::values with a parser that does not support it.

Option Modifiers

Option modifiers are the flags and expressions that you pass into the constructors for `cl::opt` and `cl::list`. These modifiers give you the ability to tweak how options are parsed and how `--help` output is generated to fit your application well.

These options fall into five main catagories:

1. Hiding an option from `--help` output
2. Controlling the number of occurances required and allowed
3. Controlling whether or not a value must be specified
4. Controlling other formatting options
5. Miscellaneous option modifiers

It is not possible to specify two options from the same catagory (you'll get a runtime error) to a single option, except for options in the miscellaneous catagory. The CommandLine library specifies defaults for all of these settings that are the most useful in practice and the most common, which mean that you usually shouldn't have to worry about these.

Hiding an option from `--help` output

The `cl::NotHidden`, `cl::Hidden`, and `cl::ReallyHidden` modifiers are used to control whether or not an option appears in the `--help` and `--help-hidden` output for the compiled program:

The **`cl::NotHidden`** modifier (which is the default for `cl::opt` and `cl::list` options), indicates the option is to appear in both help listings.

The **`cl::Hidden`** modifier (which is the default for `cl::alias` options), indicates that the option should not appear in the `--help` output, but should appear in the `--help-hidden` output.

The **`cl::ReallyHidden`** modifier, indicates that the option should not appear in any help output.

Controlling the number of occurances required and allowed

This group of options is used to control how many time an option is allowed (or required) to be specified on the command line of your program. Specifying a value for this setting allows the CommandLine library to do error checking for you.

The allowed values for this option group are:

The **`cl::Optional`** modifier (which is the default for the `cl::opt` and `cl::alias` classes) indicates that your program will allow either zero or one occurance of the option to be specified.

The **`cl::ZeroOrMore`** modifier (which is the default for the `cl::list` class) indicates that your program will allow the option to be specified zero or more times.

The **`cl::Required`** modifier indicates that the specified option must be specified exactly one time.

The **`cl::OneOrMore`** modifier indicates that the option must be specified at least one time.

The **`cl::ConsumeAfter`** modifier is described in the Positional arguments section

If an option is not specified, then the value of the option is equal to the value specified by the `cl::init` attribute. If the `cl::init` attribute is not specified, the option value is initialized with the default constructor for the data type.

If an option is specified multiple times for an option of the `cl::opt` class, only the last value will be retained.

Controlling whether or not a value must be specified

This group of options is used to control whether or not the option allows a value to be present. In the case of the CommandLine library, a value is either specified with an equal sign (e.g. '`-index-depth=17`') or as a trailing string (e.g. '`-o a.out`').

The allowed values for this option group are:

The **`cl::ValueOptional`** modifier (which is the default for `bool` typed options) specifies that it is acceptable to have a value, or not. A boolean argument can be enabled just by appearing on the command line, or it can have an explicit '`-foo=true`'. If an option is specified with this mode, it is illegal

for the value to be provided without the equal sign. Therefore '`-foo true`' is illegal. To get this behavior, you must use the cl::ValueRequired modifier.

The **`cl::ValueRequired`** modifier (which is the default for all other types except for unnamed alternatives using the generic parser) specifies that a value must be provided. This mode informs the command line library that if an option is not provides with an equal sign, that the next argument provided must be the value. This allows things like '`-o a.out`' to work.

The **`cl::ValueDisallowed`** modifier (which is the default for unnamed alternatives using the generic parser) indicates that it is a runtime error for the user to specify a value. This can be provided to disallow users from providing options to boolean options (like '`-foo=true`').

In general, the default values for this option group work just like you would want them to. As mentioned above, you can specify the cl::ValueDisallowed modifier to a boolean argument to restrict your command line parser. These options are mostly useful when extending the library.

Controlling other formatting options

The formatting option group is used to specify that the command line option has special abilities and is otherwise different from other command line arguments. As usual, you can only specify at most one of these arguments.

The **`cl::NormalFormatting`** modifier (which is the default all options) specifies that this option is "normal".

The **`cl::Positional`** modifier specifies that this is a positional argument, that does not have a command line option associated with it. See the Positional Arguments section for more information.

The **`cl::ConsumeAfter`** modifier specifies that this option is used to capture "interpreter style" arguments. See this section for more information.

The **`cl::Prefix`** modifier specifies that this option prefixes its value. With 'Prefix' options, there is no equal sign that separates the value from the option name specified. This is useful for processing odd arguments like '`-lmalloc -L/usr/lib`' in a linker tool. Here, the '`l`' and '`L`' options are normal string (list) options, that have the cl::Prefix modifier added to allow the CommandLine library to recognize them. Note that cl::Prefix options must not have the cl::ValueDisallowed modifier specified.

The **`cl::Grouping`** modifier is used to implement unix style tools (like `ls`) that have lots of single letter arguments, but only require a single dash. For example, the '`ls -labF`' command actually enables four different options, all of which are single letters. Note that cl::Grouping options cannot have values.

The CommandLine library does not restrict how you use the cl::Prefix or cl::Grouping modifiers, but it is possible to specify ambiguous argument settings. Thus, it is possible to have multiple letter options that are prefix or grouping options, and they will still work as designed.

To do this, the CommandLine library uses a greedy algorithm to parse the input option into (potentially multiple) prefix and grouping options. The strategy basically looks like this:

```
parse(string OrigInput) {

1. string input = OrigInput;
2. if (isOption(input)) return getOption(input).parse();  // Normal option
3. while (!isOption(input) && !input.empty()) input.pop_back();  // Remove the last letter
4. if (input.empty()) return error();  // No matching option
5. if (getOption(input).isPrefix())
     return getOption(input).parse(input);
6. while (!input.empty()) {    // Must be grouping options
     getOption(input).parse();
```

```
        OrigInput.erase(OrigInput.begin(), OrigInput.begin()+input.length());
        input = OrigInput;
        while (!isOption(input) && !input.empty()) input.pop_back();
      }
    7. if (!OrigInput.empty()) error();
}
```

Miscellaneous option modifiers

The miscellaneous option modifiers are the only flags where you can specify more than one flag from the set: they are not mutually exclusive. These flags specify boolean properties that modify the option.

The **`cl::CommaSeparated`** modifier indicates that any commas specified for an option's value should be used to split the value up into multiple values for the option. For example, these two options are equivalent when `cl::CommaSeparated` is specified: "`-foo=a -foo=b -foo=c`" and "`-foo=a,b,c`". This option only makes sense to be used in a case where the option is allowed to accept one or more values (i.e. it is a cl::list option).

So far, the only miscellaneous option modifier is the `cl::CommaSeparated` modifier.

Top-Level Classes and Functions

Despite all of the builtin flexibility, the CommandLine option library really only consists of one function (`cl::ParseCommandLineOptions`) and three main classes: `cl::opt`, `cl::list`, and `cl::alias`. This section describes these three classes in detail.

The `cl::ParseCommandLineOptions` function

The `cl::ParseCommandLineOptions` function is designed to be called directly from `main`, and is used to fill in the values of all of the command line option variables once `argc` and `argv` are available.

The `cl::ParseCommandLineOptions` function requires two parameters (`argc` and `argv`), but may also take an optional third parameter which holds additional extra text to emit when the `--help` option is invoked.

The `cl::ParseEnvironmentOptions` function

The `cl::ParseEnvironmentOptions` function has mostly the same effects as `cl::ParseCommandLineOptions`, except that it is designed to take values for options from an environment variable, for those cases in which reading the command line is not convenient or not desired. It fills in the values of all the command line option variables just like `cl::ParseCommandLineOptions` does.

It takes three parameters: first, the name of the program (since `argv` may not be available, it can't just look in `argv[0]`), second, the name of the environment variable to examine, and third, the optional additional extra text to emit when the `--help` option is invoked.

`cl::ParseEnvironmentOptions` will break the environment variable's value up into words and then process them using `cl::ParseCommandLineOptions`. **Note:** Currently `cl::ParseEnvironmentOptions` does not support quoting, so an environment variable containing `-option "foo bar"` will be parsed as three words, `-option`, `"foo`, and `bar"`, which is different from what you would get from the shell with the same input.

The `cl::opt` class

The `cl::opt` class is the class used to represent scalar command line options, and is the one used most of the time. It is a templated class which can take up to three arguments (all except for the first have default values

though):

```
namespace cl {
  template <class DataType, bool ExternalStorage = false,
            class ParserClass = parser<DataType> >
  class opt;
}
```

The first template argument specifies what underlying data type the command line argument is, and is used to select a default parser implementation. The second template argument is used to specify whether the option should contain the storage for the option (the default) or whether external storage should be used to contain the value parsed for the option (see Internal vs External Storage for more information).

The third template argument specifies which parser to use. The default value selects an instantiation of the `parser` class based on the underlying data type of the option. In general, this default works well for most applications, so this option is only used when using a custom parser.

The `cl::list` class

The `cl::list` class is the class used to represent a list of command line options. It too is a templated class which can take up to three arguments:

```
namespace cl {
  template <class DataType, class Storage = bool,
            class ParserClass = parser<DataType> >
  class list;
}
```

This class works the exact same as the `cl::opt` class, except that the second argument is the **type** of the external storage, not a boolean value. For this class, the marker type '`bool`' is used to indicate that internal storage should be used.

The `cl::alias` class

The `cl::alias` class is a nontemplated class that is used to form aliases for other arguments.

```
namespace cl {
  class alias;
}
```

The `cl::aliasopt` attribute should be used to specify which option this is an alias for. Alias arguments default to being Hidden, and use the aliased options parser to do the conversion from string to data.

Builtin parsers

Parsers control how the string value taken from the command line is translated into a typed value, suitable for use in a C++ program. By default, the CommandLine library uses an instance of `parser<type>` if the command line option specifies that it uses values of type '`type`'. Because of this, custom option processing is specified with specializations of the '`parser`' class.

The CommandLine library provides the following builtin parser specializations, which are sufficient for most applications. It can, however, also be extended to work with new data types and new ways of interpreting the same data. See the Writing a Custom Parser for more details on this type of library extension.

- The **generic `parser<t>` parser** can be used to map strings values to any data type, through the use of the cl::values property, which specifies the mapping information. The most common use of this parser is for parsing enum values, which allows you to use the CommandLine library for all of the error checking to make

sure that only valid enum values are specified (as opposed to accepting arbitrary strings). Despite this, however, the generic parser class can be used for any data type.

- The **`parser<bool>` specialization** is used to convert boolean strings to a boolean value. Currently accepted strings are "`true`", "`TRUE`", "`True`", "`1`", "`false`", "`FALSE`", "`False`", and "`0`".

- The **`parser<string>` specialization** simply stores the parsed string into the string value specified. No conversion or modification of the data is performed.

- The **`parser<int>` specialization** uses the C `strtol` function to parse the string input. As such, it will accept a decimal number (with an optional '+' or '-' prefix) which must start with a non-zero digit. It accepts octal numbers, which are identified with a '`0`' prefix digit, and hexadecimal numbers with a prefix of '`0x`' or '`0X`'.

- The **`parser<double>` and `parser<float>` specializations** use the standard C `strtod` function to convert floating point strings into floating point values. As such, a broad range of string formats is supported, including exponential notation (ex: `1.7e15`) and properly supports locales.

Extension Guide

Although the CommandLine library has a lot of functionality built into it already (as discussed previously), one of its true strengths lie in its extensibility. This section discusses how the CommandLine library works under the covers and illustrates how to do some simple, common, extensions.

Writing a custom parser

One of the simplest and most common extensions is the use of a custom parser. As discussed previously, parsers are the portion of the CommandLine library that turns string input from the user into a particular parsed data type, validating the input in the process.

There are two ways to use a new parser:

1. Specialize the `cl::parser` template for your custom data type.

   This approach has the advantage that users of your custom data type will automatically use your custom parser whenever they define an option with a value type of your data type. The disadvantage of this approach is that it doesn't work if your fundemental data type is something that is already supported.

2. Write an independent class, using it explicitly from options that need it.

   This approach works well in situations where you would line to parse an option using special syntax for a not-very-special data-type. The drawback of this approach is that users of your parser have to be aware that they are using your parser, instead of the builtin ones.

To guide the discussion, we will discuss a custom parser that accepts file sizes, specified with an optional unit after the numeric size. For example, we would like to parse "102kb", "41M", "1G" into the appropriate integer value. In this case, the underlying data type we want to parse into is '`unsigned`'. We choose approach #2 above because we don't want to make this the default for all `unsigned` options.

To start out, we declare our new `FileSizeParser` class:

```
struct FileSizeParser : public cl::basic_parser<unsigned> {
  // parse - Return true on error.
  bool parse(cl::Option &O, const char *ArgName, const std::string &ArgValue,
             unsigned &Val);
};
```

Our new class inherits from the `cl::basic_parser` template class to fill in the default, boiler plate, code for us. We give it the data type that we parse into (the last argument to the `parse` method so that clients of our custom

parser know what object type to pass in to the parse method (here we declare that we parse into 'unsigned' variables.

For most purposes, the only method that must be implemented in a custom parser is the parse method. The parse method is called whenever the option is invoked, passing in the option itself, the option name, the string to parse, and a reference to a return value. If the string to parse is not well formed, the parser should output an error message and return true. Otherwise it should return false and set 'Val' to the parsed value. In our example, we implement parse as:

```
bool FileSizeParser::parse(cl::Option &O, const char *ArgName,
                           const std::string &Arg, unsigned &Val) {
  const char *ArgStart = Arg.c_str();
  char *End;

  // Parse integer part, leaving 'End' pointing to the first non-integer char
  Val = (unsigned)strtol(ArgStart, &End, 0);

  while (1) {
    switch (*End++) {
    case 0: return false;   // No error
    case 'i':               // Ignore the 'i' in KiB if people use that
    case 'b': case 'B':     // Ignore B suffix
      break;

    case 'g': case 'G': Val *= 1024*1024*1024; break;
    case 'm': case 'M': Val *= 1024*1024;      break;
    case 'k': case 'K': Val *= 1024;           break;

    default:
      // Print an error message if unrecognized character!
      return O.error(": '" + Arg + "' value invalid for file size argument!");
    }
  }
}
```

This function implements a very simple parser for the kinds of strings we are interested in. Although it has some holes (it allows "123KKK" for example), it is good enough for this example. Note that we use the option itself to print out the error message (the error method always returns true) in order to get a nice error message (shown below). Now that we have our parser class, we can use it like this:

```
static cl::opt<unsigned, false, FileSizeParser>
MFS("max-file-size", cl::desc("Maximum file size to accept"),
    cl::value_desc("size"));
```

Which adds this to the output of our program:

```
OPTIONS:
  -help                - display available options (--help-hidden for more)
  ...
  -max-file-size=<size> - Maximum file size to accept
```

And we can test that our parse works correctly now (the test program just prints out the max-file-size argument value):

```
$ ./test
MFS: 0
$ ./test -max-file-size=123MB
MFS: 128974848
$ ./test -max-file-size=3G
MFS: 3221225472
$ ./test -max-file-size=dog
-max-file-size option: 'dog' value invalid for file size argument!
```

It looks like it works. The error message that we get is nice and helpful, and we seem to accept reasonable file sizes. This wraps up the "custom parser" tutorial.

Exploiting external storage

Dynamically adding command line options

Chris Lattner
The LLVM Compiler Infrastructure
Last modified: Fri Aug 1 16:30:11 CDT 2003