CLI Language

Getting Started Guide

Copyright © 2009 Code Synthesis Tools CC

Permission is granted to copy, distribute, and/or modify this document under the terms of the MIT License.

This document is available in the following formats: XHTML, PDF, and PostScript.

Table of Contents

1 Introduction 1
2 Hello World Example 1
2.1 Defining Command Line Interface 1
2.2 Translating CLI Definitions to C++ 2
2.3 Implementing Application Logic 3
2.4 Compiling and Running 4
2.5 Adding Documentation 5
3 CLI Language 6
3.1 Option Class Definition 6
3.2 Option Definition 13
3.3 Option Documentation 15
3.4 Include Directive 17
3.5 Namespace Definition 18

1 Introduction

Command Line Interface (CLI) definition language is a domain-specific language (DSL) for defining command line interfaces of C++ programs. CLI definitions are automatically translated to C++ classes using the CLI compiler. These classes implement parsing of the command line arguments and provide a convenient and type-safe interface for accessing the extracted data.

Beyond this guide, you may also find the following sources of information useful:

- CLI Compiler Command Line Manual
- The `INSTALL` file in the CLI distribution provides build instructions for various platforms.
- The `examples/` directory in the CLI distribution contains a collection of examples and a README file with an overview of each example.
- The cli-users mailing list is the place to ask technical questions about the CLI language and compiler. Furthermore, the cli-users mailing list archives may already have answers to some of your questions.

2 Hello World Example

In this chapter we will examine how to define a very simple command line interface in CLI, translate this interface to C++, and use the result in an application. The code presented in this chapter is based on the `hello` example which can be found in the `examples/hello/` directory of the CLI distribution.

2.1 Defining Command Line Interface

Our `hello` application is going to print a greeting line for each name supplied on the command line. It will also support two command line options, `--greeting` and `--exclamations`, that can be used to customize the greeting line. The `--greeting` option allows us to specify the greeting phrase instead of the default `"Hello"`. The `--exclamations` option is used to specify how many exclamation marks should be printed at the end of each greeting. We will also support the `--help` option which triggers printing of the usage information.

We can now write a description of the above command line interface in the CLI language and save it into `hello.cli`:

```
include <string>;

class options
{
  bool --help;
  std::string --greeting = "Hello";
  unsigned int --exclamations = 1;
};
```

While some details in the above code fragment might not be completely clear (the CLI language is covered in greater detail in the next chapter), it should be easy to connect declarations in `hello.cli` to the command line interface described in the preceding paragraphs. The next step is to translate this interface specification to C++.

2.2 Translating CLI Definitions to C++

Now we are ready to translate `hello.cli` to C++. To do this we invoke the CLI compiler from a terminal (UNIX) or a command prompt (Windows):

```
$ cli hello.cli
```

This invocation of the CLI compiler produces three C++ files: `hello.hxx` `hello.ixx`, and `hello.cxx`. You can change the file name extensions for these files with the compiler command line options. See the CLI Compiler Command Line Manual for more information.

The following code fragment is taken from `hello.hxx`; it should give you an idea about what gets generated:

```
#include <string>

class options
{
public:
  options (int argc, char** argv);
  options (int argc, char** argv, int& end);

  // Option accessors.
  //
public:
  bool
  help () const;

  const std::string&
  greeting () const;

  unsigned int
  exclamations () const;
```

```
private:
  ..
};
```

The `options` C++ class corresponds to the `options` CLI class. For each option in this CLI class an accessor function is generated inside the C++ class. The `options` C++ class also defines a number of overloaded constructs that we can use to parse the `argc/argv` array. Let's now see how we can use this generated class to implement option parsing in our `hello` application.

2.3 Implementing Application Logic

At this point we have everything we need to implement our application:

```
#include <iostream>
#include "hello.hxx"

using namespace std;

void
usage ()
{
  cerr << "usage: driver [options] <names>" << endl
       << "options:" << endl;
  options::print_usage (cerr);
}

int
main (int argc, char* argv[])
{
  try
  {
    int end; // End of options.
    options o (argc, argv, end);

    if (o.help ())
    {
      usage ();
      return 0;
    }

    if (end == argc)
    {
      cerr << "no names provided" << endl;
      usage ();
      return 1;
    }
```

```
    // Print the greetings.
    //
    for (int i = end; i < argc; i++)
    {
      cout << o.greeting () << ", " << argv[i];

      for (unsigned int j = 0; j < o.exclamations (); j++)
        cout << '!';

      cout << endl;
    }
  }
  catch (const cli::exception& e)
  {
    cerr << e << endl;
    usage ();
    return 1;
  }
}
```

At the beginning of our application we create the `options` object which parses the command line. The `end` variable contains the index of the first non-option argument. We then access the option values as needed during the application execution. We also catch and print `cli::exception` in case something goes wrong, for example, an unknown option is specified or an option value is invalid.

2.4 Compiling and Running

After saving our application from the previous section in `driver.cxx`, we are ready to build and run our program. On UNIX this can be done with the following commands:

```
$ c++ -o driver driver.cxx hello.cxx

$ ./driver world
Hello, world!

$ ./driver --greeting Hi --exclamations 3 John Jane
Hi, John!!!
Hi, Jane!!!
```

We can also test the error handling:

```
$ ./driver -n 3 Jane
unknown option '-n'
usage: driver [options] <names>
options:
--help
--greeting <arg>
--exclamations <arg>
```

```
$ ./driver --exclamations abc Jane
invalid value 'abc' for option '--exclamations'
usage: driver [options] <names>
options:
--help
--greeting <arg>
--exclamations <arg>
```

2.5 Adding Documentation

As we have seen in the previous sections, the `options` C++ class provides the `print_usage()` function which we can use to display the application usage information. Right now this information is very basic and does not include any description of the purpose of each option:

```
$ ./driver --help
usage: driver [options] <names>
options:
--help
--greeting <arg>
--exclamations <arg>
```

To make the usage information more descriptive we can document each option in the command line interface definition. This information can also be used to automatically generate program documentation in various formats, such as HTML and man page. For example:

```
include <string>;

class options
{
  bool --help {"Print usage information and exit."};

  std::string --greeting = "Hello"
  {
    "<text>",
    "Use <text> as a greeting phrase instead of the default \"Hello\"."
  };

  unsigned int --exclamations = 1
  {
    "<num>",
    "Print <num> exclamation marks instead of 1 by default."
  };
};
```

If we now save this updated command line interface to `hello.cli` and recompile our application, the usage information printed by the program will look like this:

```
usage: driver [options] <names>
options:
--help               Print usage information and exit.
--greeting <text>    Use <text> as a greeting phrase instead of the
                     default "Hello".
--exclamations <num> Print <num> exclamation marks instead of 1 by
                     default.
```

We can also generate the program documentation in the HTML (`--generate-html` CLI option) and man page (`--generate-man` CLI option) formats. For example:

```
$ cli --generate-html hello.cli
```

The resulting `hello.html` file contains the following documentation:

`--help`
: Print usage information and exit.

`--greeting` *text*
: Use *text* as a greeting phrase instead of the default "Hello".

`--exclamations` *num*
: Print *num* exclamation marks instead of 1 by default.

This HTML fragment can be combined with custom prologue and epilogue to create a complete program documentation (`--html-prologue`/`--html-epilogue` options for the HTML output, `--man-prologue`/`--man-epilogue` options for the man page output). For an example of such complete documentation see the CLI Compiler Command Line Manual and the `cli(1)` man page. For more information on the option documentation syntax, see Section 3.3, Option Documentation.

3 CLI Language

This chapter describes the CLI language and its mapping to C++. A CLI file consists of zero or more Include Directives followed by one or more Namespace Definitions or Option Class Definitions. C and C++-style comments can be used anywhere in the CLI file except in character and string literals.

3.1 Option Class Definition

The central part of the CLI language is *option class*. An option class contains one or more *option* definitions, for example:

```
class options
{
  bool --help;
  int --compression;
};
```

If we translate the above CLI fragment to C++, we will get a C++ class with the following interface:

```
class options
{
public:
  options (int& argc,
           char** argv,
           bool erase = false,
           cli::unknown_mode opt_mode = cli::unknown_mode::fail,
           cli::unknown_mode arg_mode = cli::unknown_mode::stop);

  options (int start,
           int& argc,
           char** argv,
           bool erase = false,
           cli::unknown_mode opt_mode = cli::unknown_mode::fail,
           cli::unknown_mode arg_mode = cli::unknown_mode::stop);

  options (int& argc,
           char** argv,
           int& end,
           bool erase = false,
           cli::unknown_mode opt_mode = cli::unknown_mode::fail,
           cli::unknown_mode arg_mode = cli::unknown_mode::stop);

  options (int start,
           int& argc,
           char** argv,
           int& end,
           bool erase = false,
           cli::unknown_mode opt_mode = cli::unknown_mode::fail,
           cli::unknown_mode arg_mode = cli::unknown_mode::stop);

  options (cli::scanner&,
           cli::unknown_mode opt_mode = cli::unknown_mode::fail,
           cli::unknown_mode arg_mode = cli::unknown_mode::stop);

  options (const options&);

  options&
  operator= (const options&);

public:
  static void
```

```
  print_usage (std::ostream&);

public:
  bool
  help () const;

  int
  compression () const;
};
```

An option class is mapped to a C++ class with the same name. The C++ class defines a set of public overloaded constructors, a public copy constructor and an assignment operator, as well as a set of public accessor functions and, if the `--generate-modifier` CLI compiler option is specified, modifier functions corresponding to option definitions. It also defines a public static `print_usage()` function that can be used to print the usage information for the options defined by the class.

The `argc/argv` arguments in the overloaded constructors are used to pass the command line arguments array, normally as passed to `main()`. The `start` argument is used to specify the position in the arguments array from which the parsing should start. The constructors that don't have this argument, start from position 1, skipping the executable name in `argv[0]`. The `end` argument is used to return the position in the arguments array where the parsing of options stopped. This is the position of the first program argument, if any. If the `erase` argument is `true`, then the recognized options and their values are removed from the `argv` array and the `argc` count is updated accordingly.

The `opt_mode` and `arg_mode` arguments specify the parser behavior when it encounters an unknown option and argument, respectively. The `unknown_mode` type is part of the generated CLI runtime support code. It has the following interface:

```
namespace cli
{
  class unknown_mode
  {
  public:
    enum value
    {
      skip,
      stop,
      fail
    };

    unknown_mode (value v);
    operator value () const;
  };
}
```

If the mode is `skip`, the parser skips an unknown option or argument and continue parsing. If the mode is `stop`, the parser stops the parsing process. The position of the unknown entity is stored in the `end` argument. If the mode is `fail`, the parser throws the `cli::unknown_option` or `cli::unknown_argument` exception (described blow) on encountering an unknown option or argument, respectively.

Instead of the `argc/argv` arguments, the last overloaded constructor accepts the `cli::scanner` object. It is part of the generated CLI runtime support code and has the following abstract interface:

```
namespace cli
{
  class scanner
  {
  public:
    virtual bool
    more () = 0;

    virtual const char*
    peek () = 0;

    virtual const char*
    next () = 0;

    virtual void
    skip () = 0;
  };
}
```

The CLI runtime also provides two implementations of this interface: `cli::argv_scanner` and `cli::argv_file_scanner`. The first implementation is a simple scanner for the `argv` array (it is used internally by all the other constructors) and has the following interface:

```
namespace cli
{
  class argv_scanner
  {
  public:
    argv_scanner (int& argc, char** argv, bool erase = false);
    argv_scanner (int start, int& argc, char** argv, bool erase = false);

    int
    end () const;

    ...
  };
}
```

The `cli::argv_file_scanner` implementation provides support for reading command line arguments from the `argv` array as well as files specified with command line options. It is generated only if explicitly requested with the `--generate-file-scanner` CLI compiler option and has the following interface:

```
namespace cli
{
  class argv_file_scanner
  {
  public:
    argv_file_scanner (int& argc,
                       char** argv,
                       const std::string& file_option,
                       bool erase = false);

    argv_file_scanner (int start,
                       int& argc,
                       char** argv,
                       const std::string& file_option,
                       bool erase = false);
    ...
  };
}
```

The `file_option` argument is used to pass the option name that should be recognized as specifying the file containing additional options. Such a file contains a set of options, each appearing on a separate line optionally followed by space and an option value. Empty lines and lines starting with # are ignored. The semantics of providing options in a file is equivalent to providing the same set of options in the same order on the command line at the point where the options file is specified, except that shell escaping and quoting is not required. Multiple files can be specified by including several file options on the command line or inside other files.

The parsing constructor (those with the `argc/argv` or `cli::scanner` arguments) can throw the following exceptions: `cli::unknown_option`, `cli::unknown_argument`, `cli::missing_value`, and `cli::invalid_value`. The first two exceptions are thrown on encountering unknown options and arguments, respectively, as described above. The `missing_value` exception is thrown when an option value is missing. The `invalid_value` exception is thrown when an option value is invalid, for example, a non-integer value is specified for an option of type `int`.

Furthermore, all scanners (and thus the parsing constructors that call them) can throw the `cli::eos_reached` exception which indicates that one of the `peek()`, `next()`, or `skip()` functions were called while `more()` returns `false`. Catching this exception normally indicates an error in an option parser implementation. The `argv_file_scanner` class can also throw the `cli::file_io_failure` exception which indicates that a file could not be opened or there was a reading error.

All CLI exceptions are derived from the common `cli::exception` class which implements the polymorphic `std::ostream` insertion. For example, if you catch the `cli::unknown_option` exception as `cli::exception` and print it to `std::cerr`, you will get the error message corresponding to the `unknown_option` exception.

The exceptions described above are part of the generated CLI runtime support code and have the following interfaces:

```
#include <exception>

namespace cli
{
  class exception: public std::exception
  {
  public:
    virtual void
    print (std::ostream&) const = 0;
  };

  inline std::ostream&
  operator<< (std::ostream& os, const exception& e)
  {
    e.print (os);
    return os;
  }

  class unknown_option: public exception
  {
  public:
    unknown_option (const std::string& option);

    const std::string&
    option () const;

    virtual void
    print (std::ostream&) const;

    virtual const char*
    what () const throw ();
  };

  class unknown_argument: public exception
  {
  public:
    unknown_argument (const std::string& argument);

    const std::string&
    argument () const;

    virtual void
```

```
    print (std::ostream&) const;

    virtual const char*
    what () const throw ();
  };

  class missing_value: public exception
  {
  public:
    missing_value (const std::string& option);

    const std::string&
    option () const;

    virtual void
    print (std::ostream&) const;

    virtual const char*
    what () const throw ();
  };

  class invalid_value: public exception
  {
  public:
    invalid_value (const std::string& option,
                   const std::string& value);

    const std::string&
    option () const;

    const std::string&
    value () const;

    virtual void
    print (std::ostream&) const;

    virtual const char*
    what () const throw ();
  };

  class eos_reached: public exception
  {
  public:
    virtual void
    print (std::ostream&) const;

    virtual const char*
    what () const throw ();
  };

  class file_io_failure: public exception
```

```
  {
  public:
    file_io_failure (const std::string& file);

    const std::string&
    file () const;

    virtual void
    print (std::ostream&) const;

    virtual const char*
    what () const throw ();
  };
}
```

3.2 Option Definition

An option definition consists of four components: *type*, *name*, *default value*, and *documentation*. An option type can be any C++ type as long as its string representation can be parsed using the `std::istream` interface. If the option type is user-defined then you will need to include its declaration using the Include Directive.

An option of a type other than `bool` is expected to have a value. An option of type `bool` is treated as a flag and does not have a value. That is, a mere presence of such an option on the command line sets this option's value to `true`.

The name component specifies the option name as it will be entered in the command line. A name can contain any number of aliases separated by `|`. The C++ accessor and modifier function names are derived from the first name by removing any leading special characters, such as `-`, `/`, etc., and replacing special characters in other places with underscores. For example, the following option definition:

```
class options
{
  int --compression-level | --comp | -c;
};
```

Will result in the following accessor function:

```
class options
{
  int
  compression_level () const;
};
```

While any option alias can be used on the command line to specify this option's value.

If the option name conflicts with one of the CLI language keywords, it can be specified as a string literal:

```
class options
{
  bool "int";
};
```

The following component of the option definition is the optional default value. If the default value is not specified, then the option is initialized with the default constructor. In particular, this means that a `bool` option will be initialized to `false`, an `int` option will be initialized to `0`, etc.

Similar to C++ variable initialization, the default option value can be specified using two syntactic forms: an assignment initialization and constructor initialization. The two forms are equivalent except that the constructor initialization can be used with multiple arguments, for example:

```
include <string>;

class options
{
  int -i1 = 5;
  int -i2 (5);

  std::string -s1 = "John";
  std::string -s2 ("Mr John Doe", 8, 3);
};
```

The assignment initialization supports character, string, boolean, and simple integer literals (including negative integers) as well as identifiers. For more complex expressions use the constructor initialization or wrap the expressions in parenthesis, for example:

```
include "constants.hxx"; // Defines default_value.

class options
{
  int -a = default_value;
  int -b (25 * 4);
  int -c = (25 / default_value + 3);
};
```

By default, when an option is specified two or more times on the command line, the last value overrides all the previous ones. However, a number of standard C++ containers are handled differently to allow collecting multiple option values or building key-value maps. These containers are `std::vector`, `std::set`, and `std::map`.

When `std::vector` or `std::set` is specified as an option type, all the values for this option are inserted into the container in the order they are encountered. As a result, `std::vector` will contain all the values, including duplicates while `std::set` will contain all the unique values. For example:

```
include <set>;
include <vector>;

class options
{
  std::vector<int> --vector | -v;
  std::set<int> --set | -s;
};
```

If we have a command line like this: `-v 1 -v 2 -v 1 -s 1 -s 2 -s 1`, then the vector returned by the `vector()` accessor function will contain three elements: `1`, `2`, and `1` while the set returned by the `set()` accessor will contain two elements: `1` and `2`.

When `std::map` is specified as an option type, the option value is expected to have two parts: the key and the value, separated by =. All the option values are then parsed into key/value pairs and inserted into the map. For example:

```
include <map>;
include <string>;

class options
{
  std::map<std::string, std::string> --map | -m;
};
```

The possible option values for this interface are: `-m a=A`, `-m =B` (key is an empty string), `-m c=` (value is an empty string), or `-m d` (same as `-m d=`).

The last component in the option definition is optional documentation. It is discussed in the next section.

3.3 Option Documentation

Option documentation mimics C++ string array initialization: it is enclosed in `{}` and consists of one or more documentation strings separated by a comma, for example:

```
class options
{
  int --compression = 5
  {
    "<level>",
    "Set compression to <level> instead of 5 by default.
```

```
    With the higher compression levels the program may produce a
    smaller output but may also take longer and use more memory."
  };
};
```

The option documentation consists of a maximum of three documentation strings. The first string is the value documentation string. It describes the option value and is only applicable to options with types other than `bool` (options of type `bool` are flags and don't have an explicit value). The second string (or the first string for options of type `bool`) is the short documentation string. It provides a brief description of the option. The last entry in the option documentation is the long documentation string. It provides a detailed description of the option. The short documentation string is optional. If only two strings are present in the option documentation (one string for options of type `bool`), then the second (first) string is assumed to be the long documentation string.

Option documentation is used to print the usage information as well as to generate program documentation in the HTML and man page formats. For usage information the short documentation string is used if provided. If only the long string is available, then, by default, only the first sentence from the long string is used. You can override this behavior and include the complete long string in the usage information by specifying the `--long-usage` CLI compiler option. When generating the program documentation, the long documentation strings are always used.

The value documentation string can contain text enclosed in <> which is automatically recognized by the CLI compiler and typeset according to the selected output in all three documentation strings. For example, in usage the `level` value for the `--compression` option presented above will be displayed as `<level>` while in the HTML and man page output it will be typeset in italic as *`level`*. Here is another example using the `std::map` type:

```
include <map>;
include <string>;

class options
{
  std::map<std::string, std::string> --map
  {
    "<key>=<value>",
    "Add the <key>, <value> pair to the map."
  };
};
```

The resulting HTML output for this option would look like this:

`--map` *key*=*value*
    Add the *key*, *value* pair to the map.

As you might have noticed from the examples presented so far, the documentation strings can span multiple lines which is not possible in C++. Also, all three documentation strings support the following basic formatting mechanisms. The start of a new paragraph is indicated by a blank line. A fragment of text can be typeset in monospace font (normally used for code fragments) by enclosing it in the `\c{}` block. Similarly, text can be typeset in bold or italic fonts using the `\b{}` and `\i{}` blocks, respectively. You can also combine several font properties in a single block, for example, `\cb{bold code}`. If you need to include literal `}` in a formatting block, you can use the `\}` escape sequence, for example, `\c{int a[] = {1, 2\}}`. The following example shows how we can use these mechanisms:

```
class options
{
  int --compression = 5
  {
    "<level>",
    "Set compression to <level> instead of 5 by default.

     With the higher compression levels the program \i{may}
     produce a smaller output but may also \b{take longer}
     and \b{use more memory}."
  };
};
```

The resulting HTML output for this option would look like this:

`--compression` *level*

Set compression to *level* instead of 5 by default.

With the higher compression levels the program *may* produce a smaller output but may also **take longer** and **use more memory**.

3.4 Include Directive

If you are using user-defined types in your option definitions, you will need to include their declarations with the include directive. Include directives can use `< >` or `" "`-enclosed paths. The CLI compiler does not actually open or read these files. Instead, the include directives are translated to C++ preprocessor `#include` directives in the generated C++ header file. For example, the following CLI definition:

```
include <string>;
include "types.hxx"; // Defines the name_type class.

class options
{
  std::string --string;
  name_type --name;
};
```

Will result in the following C++ header file:

```
#include <string>
#include "types.hxx"

class options
{
  ...

  const std::string&
  string () const;

  const name_type&
  name () const;

  ...
};
```

Without the `#include` directives the `std::string` and `name_type` types in the `options` class would be undeclared and result in compilation errors.

3.5 Namespace Definition

Option classes can be placed into namespaces which are translated directly to C++ namespaces. For example:

```
namespace compiler
{
  namespace lexer
  {
    class options
    {
      int --warning-level = 0;
    };
  }

  namespace parser
  {
    class options
    {
      int --warning-level = 0;
    };
  }

  namespace generator
  {
    class options
    {
```

```
      int --target-width = 32;
    };
  }
}
```

The above CLI namespace structure would result in the equivalent C++ namespaces structure:

```
namespace compiler
{
  namespace lexer
  {
    class options
    {
      int
      warning_level () const;
    };
  }

  namespace parser
  {
    class options
    {
      int
      warning_level () const;
    };
  }

  namespace generator
  {
    class options
    {
      int
      target_width () const;
    };
  }
}
```