Cfg66 Developer Guide 0.1.0

Chris Ahlstrom
(ahlstromcj@gmail.com)

April 25, 2024

Cfg66 Logo

Contents

List of Figures

1 Introduction

The *Cfg66* library reworks some of the fundamental code from the *Seq66* project ([3]). This work is in preparation for the version 2 of that project, but might also be useful in other applications.

Cfg66 contains the following subdirectories of `src` and `include`, each of which holds modules in a namespace of the same name:

- `cfg`. Contains items that can be used to manage a generic configuration, including application names, settings basics, and an INI-style configuration-file system. Added are data-type indicators and help text.
- `cli`. Provides C/C++ code to handle command-line parsing without needing to use, for example, getopt. While it somewhat matches how getopt works, it also allows combining option sets and provides a parser object that contains the current status of all available options, as well as help text.
- `session`. Contains classes for managing a basic "session". Here, a session is simply a location to put configuration files; multiple locations can be supported. Session filenames are based on a "home" configuration directory, optional subdirectories, and application-specific item names.
- `util`. Contains file functions, message functions, string functions, and other functionality common to all our "66" application and libraries.

In the sections that follow, the basic are described. At some point we will make the effort to add some *Dia* diagrams to make the relationships more clear.

1.1 Naming Conventions

Cfg66 uses some conventions for naming things in this document.

- `$prefix`. The base location for installation of the application and its ancillary data files on *UNIX/Linux/BSD*:
  - `/usr/`
  - `/usr/local/`
- `$winprefix`. The base location for installation of the application and its ancillary data files on *Windows*.
  - `C:/Program Files/`
  - `C:/Program Files (x86)/`
- `$home`. The location of the user's configuration files. Not to be confused with `$HOME`, this is the standard location for configuration files. On a UNIX-style system, it would be `$HOME/.config/appname`. The files would be put into a `po` subdirectory here.
- `$winhome`. This location is different for *Windows*: `C:/Users/user/AppData/Local/PACKAGE`.

1.2 Future Work

- Hammer on this code in *Windows*.

2 Cfg Namespace

This section provides a useful walkthrough of the `cfg` namespace of the *cfg66* library.

Here are the classes (or modules) in this namespace:

- `cfg::appinfo`
- `cfg::basesettings`
- `cfg::comments`
- `cfg::configfile`
- `cfg::history`
- `cfg::inisections`
- `cfg::memento`
- `cfg::options`
- `cfg::optionsmaps`
- `cfg::palette`

2.1 cfg::appinfo

The `cfg::appinfo` class encapsulates basic information about an application:

- *Kind*. Indicates if the application is a headless application (such as a daemon), a command-line, *ncurses*, GUI, or test application. In some cases it can be useful to know how the application is running.
- *Name*. Provides the short name of the application, which can be shown in warning messages or be used to name the application as a node, for example, in *JACK*.
- *Version*. Provides a version number, or optionally, a variant on the *Name* plus the version number.
- *Home directory*. Provides the name of the configuration directory for the application, such as `/home/user/.config/appname`. Modified by the `-home` option.
- *Config file*. Provides the name of the main configuration file, such as `appname.rc`. Modified by the `-config` option.
- *Client name*. Provides a variant on the application name, useful in session managers, for example. An example under *NSM* would be `seq66v2.nUKIE`.
- *App Tag*. Seems the same as *Version*. Must clarify!
- *Arg0*. Holds the complete path to the executable as run.
- *Package*. Provides the name of the package, which could be the same as the application or library name.
- *Session Tag*. Useful in long error/warning/info messages. Must clarify!
- *Icon*. Provides the base name of the application icon, if not empty.
- *Version text*. Reconstructed version information for the application.
- *API engine*. Some applications might use various libraries. For example, for a MIDI application it might be one of these MIDI libraries: `rtmidi`, `rtl66`, or `portmidi`.
- *API version*. Indicates which version of an API is in force. In some cases this can be detected at run-time.
- *GUI version*. Indicates the GUI or "curses" version, such as *Qt 6.1* or *Gtkmm 3.0*.

- *Short client name*. Similar to *client name*, but never has anything appended to it.
- *Client name tag*. Provides the name to show on the console in error/warning/info messages, the short client name surrounded by brackets, such as `[seq66]`.

All of these items can be set at once using the `appinfo` constructor that has a large number of parameters. In addition, there are a many "free" functions in the `cfg` namespace for setting and getting these values. See `appinfo.hpp` for a summary.

2.2 cfg::basesettings

Provides common settings useful in any application.

- *File name*. Hold the (optional) name of the file that holds the settings data.
- *Modified*. Indicates if the setting have been modifies.
- *Config format*. A free-form string indicating the format of the data, such as `INI`, `XML`, or `JSON`.
- *Config type*. A short string that indicates something about the format or content of the file. For example, in Seq66, common values were 'rc' and 'usr', with these values also representing the file extension.
- *Ordinal version*. A simple integer that is incremented each time a change is made in the configuration that isn't detectable during parsing.
- *Comments*. An object holding the main comments that describe something about the settings file.

Also included are an error flag and an accompanying message that the application can display.

2.3 cfg::comments

Holds a string describing something general about the configuration. Provides some setter and getter functions.

2.4 cfg::configfile

Provides some items common to configuration files, including an extensive set of functions to parse sections and configuration variables in an INI format in a line-by-line fashion.

These are the main externally-accessible values:

- *File extension*. Common values are 'rc' and 'session'.
- *File name*. Provides the file name, normally as a full-path file-specification.
- *File version*. Provides the current version of the derived configuration file format. Set in the constructor of the configfile-derived object, and incremented in that object whenever a new way of reading, writing, or formatting the configuration file is created.

Also too many to list, but it includes functions such as `get_integer()` and `write_integer()`.

These work by having the whole file read into an `std::ifstream`, and then searching the string over and over to read all the variables. Sounds inefficient, but in practise it is very fast.

Finally, there are free functions to delete a configuration file and to make a copy of a configuration file.

2.5 cfg::history

One of the things not handled so well in *Seq66* is the undo/redo functionality. The `history` template class implements undo/redo using the `memento` class described below. It follows the *Design Patterns* book ([2]) starting on page 62. Also informative is [1]. Also see the `cfg::memento` class below.

The heart of the `history` template is the history list:

```
std::deque<memento<TYPE>> m_history_list;
```

Member functions are provided to see if history entries are undoable/redoable, undo and redo them, check the maximum size of the list, The `history.cpp` module provide a small test of history for the `cfg::options` class.

2.6 cfg::inisections

The `inisections.hpp` file actually declares four classes:

- `cfg::inimap`. Defines two types:
  - `optionref`. `std::reference_wrapper<options::spec>`.
  - `optionmap`. `std::map<std::string, optionref>`.

  Also defined is a function to add an named `cfg::option` to the option map.
- `cfg::inisection`. This object contains a `specification` structure that provides the name of a section, it's description, and a container of options. It is represent in a configuration file by a section or tag name enclosed in brackets: `[output-ports]` as an example. Each section can also define a file extension (e.g. `.rc`) to indicate its locus.
- `cfg::inifile`. Defines four types:
  - `specref`. `std::reference_wrapper<inisection::specification>`.
  - `specrefs`. `std::vector<specref>`.
  - `sectionlist`. `std::vector<inisection>`.
  - `specification`. This structure holds the configuration file's extension, directory, base name, description, and a vector of `specrefs`.
- `cfg::inifiles`. Holds a list of inifile objects. Represents all of the files, and all of the options that are held by the files. The options are in a single list, and the INI items look them up by name. NOT SURE THIS CONCEPT IS USEFUL.
- `cfg::xxx`.

There is a lot to this module. For now, see the comment in the `.hpp` and `.cpp` files.

2.7 cfg::memento

The `cfg::memento` template class is a small object that holds one copy of a state object. It provides these functions:

- `memento (const TYPE & s)`.
- `bool set_state (const TYPE & s)`.
- `const TYPE & get_state () const`.

The `cfg::history` class stores a "history list": `std::deque<memento<TYPE>`.

2.8 cfg::options

The `cfg::options` holds a number of items needed for the specification, reading, and writing of options. The options can be read from configuration file or from the command-line. These items are nested in the class:

- *Static data*. Flags and numbers are provided to indicate if the option is enabled and how it is to be output in a nice format into an INI file.
- *kind*. Indicates if the option is boolean, numerical, a filename, list, string and some others. This enumeration makes it easier to process the option.
- *spec*. This nested structure contains these values. For brevity, the `option_` portion of the name and the type are not shown:
  - `code`. Optional single-character name.
  - `kind`. Is it boolean, integer, string...?
  - `cli_enabled`. Normally true; false disables.
  - `default`. Either a value or "true"/"false".
  - `value`. The actual value as parsed.
  - `read_from_cli`. Option already set from CLI.
  - `modified`. Option changed since read/save.
  - `desc`. A one-line description of option.
  - `built_in`. This option is present in all apps.
- *option*. The `cfg::options::option` is a simple pair of the name of the option and the `spec` that describes it.
- *container*. The `cfg::options::container` is a map (dictionary) of option `specs` keyed bythe name of the option.

Also specified are the name of the source file and the name of the source section in that file.

Included are quite a number of functions for looking up option values and option characteristics. Also include are free functions to make options.

The `options.cpp` module not only contains many comments explaining the module, but also a statically-initialized list of default options that any application can use. It is also a great example of how to creat a list of options.

2.9 cfg::optionsmaps

Presently not used in this library. More to come?

2.10 cfg::palette

The `cfg::palette` template class can be used to define a palette of color code pair with a platform-specific color class such as `QColor` from *Qt*. This is a feature copped from *Seq66*.

3 Cli Namespace

This section provides a useful walkthrough of the `cli` namespace of the *cfg66* library. In addition, a `C`-only module is provided.

Here are the classes (or modules) in this namespace:

- `cliparser_c`
- `cli::parser`

3.1 cliparser_c Module

This module is actually a `C++` module that implements a number of `extern "C"` functions. The functions themselves access an internal and hidden `cli::parser` object and call its member functions to perform the functions.

This module provides a `C` structure that mirrors `options::spec`, plus some free functions to access this structure.

3.2 cli::parser

The `cli::parser` class contains a number of options in a `cfg::options` object. Many of the member functions are pass-alongs to this object.

It also holds values indicating if some basic options (help, version, verbosity, log-file usage) are set. The `parse()` function looks for stock option such as `-help` and `-option log filename`. The `--` sequence can terminate a list of options.

It is meant to be similar to getopt, but much more flexible and perhaps easier to set up.

4 Session Namespace

This section provides a useful walkthrough of the `session` namespace of the *cfg66* library. In addition, a `C`-only module is provided.

Here are the classes (or modules) in this namespace:

- `session::climanager`
- `session::configuration`
- `session::directories`
- `session::manager`

4.1 session::climanager

The `session::climanager` class is derived from `session::manager` and overrides a number of virtual functions It also provides a function to read a configuration file. It provides a `run()` loop which does nothing but check for calls to close and save the session and wait for a small polling period.

4.2 session::configuration

The `session::configuration` class is derived from `cfg::basesettings`. It contains a `session::directories` management class and a set of `directories::entries` items. Options for help, a log-file, and auto-save are provided.

4.3 session::directories

The `session::directories` class manages a set of `entry` directory item.

Each `entry` specifies:

- Name of the section covered by a configuration file.
- It active/inactive status.
- The directory for the files(s).
- The base name of the file(s).
- The optional extension of the file(s).

Provides the full path specification of each file, constructed from the entries, and keyed by the section name. The "home" configuration path and the session path are also specified.

The `directories.cpp` module explains the directory layouts and provides default "rc" and "log" directory specifications.

4.4 session::manager

The `session::manager` class is base class for providing an application with "session" information, where a session is a group of configuration items that allow an application to run in a sequestered environment. Think of the *JACK Session Manager* or the *New/Non Session Manager*.

The base session manager class holds the following information:

- `session::configuration`. See the section about this class above.
- `cli::parser`. Provides access to the command-line parser.
- *Capabilities*. This application-dependent string publishes some information about the application. Useful with the *New/Non Session Manager*.
- *Manager name*. The name of the session manager. For example, it is returned by the *New/Non Session Manager*.
- *Manager path*. This item holds the directory where the session information is to be stored. For example, the *New/Non Session Manager* returns a string like `/home/user/NSM Sessions/JackSession/seq66`
- *Display name*. This is the name of the session to be displayed, such as *JackSession* in the string above.
- *Client ID*. This is the name of the client (e.g. for managing port connections), such as *Seq66* or *seq66.nUKIE*.

Also included are indicators for `-help`, dirty status, and error messages.

A large number of `virtual` members functions are included. Some of the important functions are for the following actions:

- Parsing an option (configuration) file.
- Parsing the command line.
- Creating, writing, and reading a configuration file.
- Creating, saving, and closing a session.
- Creating a session directory.
- Creating a "project".
- Creating a manager.
- Running a session, often in a loop or a GUI thread.

The `manager.cpp` module contains a short statically-initialized list of default options.

Note that there are currently a number of "To Dos" in this class.

5 Util Namespace

This section provides a useful walkthrough of the `util` namespace of the *cfg66* library.

Here are the classes (or modules) in this namespace:

- `util::filefunctions module`

- `util::msgfunctions module`
- `util::named_bools`
- `util::strfunctions module`

All of the modules are `C++` modules with free functions in the `util` namespace.

5.1 util::filefunctions module

The `util::filefunctions` module contains a large number of function dealing file-names and files.

The file functions are basically wrappers around the `C FILE *` API.

The file-name functions are useful for building paths and for splitting paths into parts.

Really, just skim the `filefunctions` modules to learn what is there. They include every convenient function we needed in implementing *Seq66*.

5.2 util::msgfunctions module

The `util::msgfunctions` module defines functions for writing messages to the console along with tags showing the short name of the application that wrote them, and in color.

Also included are some "async safe" functions for output and for converting unsigned numbers to string arrays.

5.3 util::named_bools

The `util::name_bools` class makes it easy to look up and set a "small" number of boolean values by name.

This class could be useful if one does not want the full capability of the classes in the `cfg` namespace.

5.4 util::strfunctions module

The `util::strfunctions` module defines functions for manipulating strings: tokenization, left/right space trimming, conversion between strings and values with the added feature of defaulting, word-wrapping, and formatting of `std::string` values without using `std::stringstream`.

6 Cfg66 Tests

This section provides a useful walkthrough of the testing of the *cfg66* library. They illustrate the various ways in which the *Cfg66* library can be used by a developer.

The tests so far are these executables:

- cliparser_test_c
- cliparser_test
- history_test
- ini_test
- manager_test
- options_test

These tests are supported by data structures define in the following header files:

- textttctrl_spec.hpp
- textttdrums_spec.hpp
- texttttmutes_spec.hpp
- textttpalette_spec.hpp
- textttplaylist_spec.hpp
- texttttrc_spec.hpp
- textttsession_spec.hpp
- texttttest_spec.hpp
- textttusr_spec.hpp

These header files will be discussed as needed in the following sections.

6.1 Cfg66 cliparser_test_c Test

This test of the C command-line interface uses the free functions in cliparser_c.h. The test module itself sets up a small set of test options:

```
static options_spec s_test_options [] =
{
    //   Name          Code  Kind     Enabled   Default     Value      Dirty
    {
        "alertable",    'a', "boolean", true,   "false",    "false",   false,
        "If specified, the application is alertable."
    }, . . .
};
```

The test makes changes to the options and verifies that they took hold. The test command is simple:

```
$ ./build/test/cliparser_test_c
```

It shows the changes and a result statement.

6.2 Cfg66 cliparser_test Test

This test uses the following tests options (only one is shown) static initialization:

```
static cfg::options::container s_test_options
{
    //   Name, Code,  Kind, Enabled,
    //   Default, Value, FromCli, Dirty,
    //   Description, Built-in
    {
        {
            "alertable",
            {
                'a', "boolean", cfg::options::enabled,
                "false", "false", false, false,
                "If specified, the application is alertable.",
                false
            }
        }, . . .
    }
};
```

It serves as a good example of how to create a list of options. More flexible than GNU's getopt setup and simplifies generating help text.

The test makes changes to the options and verifies that they took hold. The test command is not as simple as the `C` version, as verbosity is needed to see the changes:

```
$ ./build/test/cliparser_test --verbose
```

It shows the changes and a result statement. This test needs a little bit of cleanup.

6.3 Cfg66 history_test Test

The history test also sets up a test options "array". Then it makes changes to the options, such as changing variables, undoing the change, and redoing the change. It shows the changes and a result statement.

See this test file for some "To do" items.

6.4 Cfg66 ini_test Test

This test program uses all of these "data" headers:

- textttctrl_spec.hpp
- textttdrums_spec.hpp
- textttmutes_spec.hpp
- textttpalette_spec.hpp
- textttplaylist_spec.hpp
- texttrc_spec.hpp
- textttsession_spec.hpp

- textttusr_spec.hpp

It defines these static test items:

- `cfg::options::container s_test_options`. This sets up a single option called "test", used as a command-line option.
- `cfg::inisection::specification s_simple_ini_spec`. This sets up an `[experiments]` section with a number of option-variable definitions.
- `cfg::inisection::specification s_section_spec`. This sets up an `[section-test]` section.
- `cfg::inifile::specification exp_file_data` This items sets up the "experiment" configuration directory using `cfg::inisection::specification s_simple_ini_spec`. `cfg::inisection::specifi s_section_spec`.

Additional sections are defined and add to a `cfg::inifile::specification` declaration.

Hmmm. some are unsued.

6.5 Cfg66 manager_test Test

This test defines `cfg::appinfo s_application_info`. This is used here: `cfg::initialize_appinfo(s_applicati argv[0])`.

The "To do" here is to actually implement `simple_smoke_test`.

6.6 Cfg66 options_test Test

This test program uses only the "data" header `test_spec.hpp` which defines `cfg::options::container s_test_options`. This container is used to initialize a `cli::parser`. That object then gets the command-line arguments.

Obviously, we still have a lot of work to do with these tests.

7 Summary

Contact: If you have ideas about *Cfg66* or a bug report, please email us (at mailto:ahlstromcj@gmail.com).

8 References

The *Cfg66* reference list.

References

[1] Twinkle Sharma. *Deque in C++* https://www.scaler.com/topics/cpp/deque-in-cpp/. 2024.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Use your search engine. 1994.

[3] Chris Ahlstrom. *A reboot of the Seq24 project as "Seq66"*. https://github.com/ahlstromcj/seq66/. 2015-2024.