Cfg66 Developer Guide 0.2.0

Chris Ahlstrom
(`ahlstromcj@gmail.com`)

August 5, 2024

Cfg66 Logo

Contents

List of Figures

List of Tables

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::inifile`
- `cfg::inimanager`
- `cfg::inisection`
- `cfg::inisections`
- `cfg::memento`
- `cfg::options`
- `cfg::palette`
- `cfg::recent`

2.1 cfg::appinfo

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

- *cfg::appkind*. 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.
- *App 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*.
- *App version*. Provides a version number, or optionally, a variant on the *Name* plus the version number.
- *Main config section name*. Holds the name of main configuration section (usually present only in the "session" or "rc" file). The default value is "[Cfg66]".
- *App home configuration directory*. Provides the name of the configuration directory for the application, such as `/home/user/.config/appname`. Modified by the `-home` option.
- *Home configuration file*. Provides the name of the main configuration file, such as `appname.rc`. Modified by the `-config` option.

- *Client name.* Either the short application name or a variant of it, useful in session managers, for example. An example under *NSM* would be `seq66v2.nUKIE`.
- *App tag.* The short application name with the version number tacked on.
- *Arg0.* Holds the complete path to the executable as determined at run-time.
- *Package name.* Provides the name of the package, which could be the same as the application or library name.
- *Session Tag.* Provides the session name for the application; it can be the application name, a modification of the application name, or something completely different, such as a session name given by a user. Useful in long error/warning/info messages.
- *App 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 `[seq66v2]`.

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

THe `cfg::basesettings` class is a base class. It provides common settings useful in any application:

- *File name.* Holds the (optional) name of the file that holds the settings data.
- *Ordinal version.* Provides a simple integer indicating the version of the file, useful in adapting to changes in settings.
- *Modify/unmodify function and "modified" flag.* Indicates if the settings have been modified.
- *Config format.* A free-form string indicating the format of the data, such as `INI`, `XML`, or `JSON`. For "66" libraries and applications, the `INI` format is sufficient.
- *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 format or values-present that isn't detectable during parsing.
- *Comments.* An object holding the main comments that describe something about the settings file.
- *Is error.* Indicated if there was a error during parsing.
- *Error message.* If an error occurred, the error message is stored for display.

2.3 cfg::comments

`cfg::comments` holds a string describing something general about the configuration, meant to be included as the text of a `[comment]` section. Provides some setter and getter functions. It is a simple class.

2.4 cfg::configfile

The `cfg::configfile` class is an `abstract base class`. It 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. The member functions `parse()` and `write()` are *pure virtual*, and must be overridden in a derived class. A good example is `cfg::inifile`.

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]). 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, etc. The `history.cpp` module provide a small test of history for the `cfg::options` class.

The `history_test` application provides a more substantive test of history. It provides an `cfg::options::container` with a few options, and then applies `undo` and `redo` operations on them, checking the results.

To do: test range validation.

2.6 cfg::inifile

`cfg::inifile` is derived from the `cfg::configfile` class. It contains a reference to a class that manages multiple INI sections: `cfg::inisections`. It overrides the `cfg::configfile::parse()`

and `cfg::configfile::write()` functions. It provides two protected functions, `parse_section()` and `write_section()`.

The `ini_test` application sets up a couple of `cfg::inisection::specification` named structures with a long description and a list of values to be stored. The `cfg::stock_cfg66_data()` and `cfg::stock_comment_data()` specification are defined as well. All section specifications are wrapped in a `cfg::inisections::specification` structure that is used to create a `cfg::inisections` object and then use it to create an INI file for output, exercising `inifile::write()`. The file generated is `tests/data/fooout.rc`.

The same setup is used to exercise `inifile::parse()`. The file read is `tests/data/fooin.rc`. The file generated from `fooin.rc` is `tests/data/fooinout.rc`. The file generated from the test's internal data is `tests/data/fooout.rc`. The results can be found in the `tests/data` directory.

How does it work?

```
cfg::inisections sections(exp_file_data, "fooout");
```

The `exp_file_data` specification provides the configuration type (file extension without the period) `rc`, the putative file directory `./tests/data`, the default base name of the configuration file `ini_test_session`, followed by the description of the INI sections object and a list of the sections in the INI sections object.

Once the `exp_file_data` structure is loaded into the `inisection`, we can hook it to an `inifile` and write the file:

```
cfg::inifile f_out(sections);
success = f_out.write();
```

The section data items are written to the assembled full file-specification, `./tests/data/fooout.rc`.

The next test creates an `inisections` with the base name "fooin". An `inifile` is created from this section and is used to parse `./tests/data/fooin.rc`.

```
cfg::inisections sections(exp_file_data, "fooin");
cfg::inifile f_in(sections);
success = f_in.parse();
```

Once written, we create another `inifile`, changing it's base-name to "fooinout". Then we write it:

```
cfg::inifile f_inout(sections, "fooinout");  // changes the name
success = f_inout.write();
```

The files `./tests/data/fooin.rc` and `./tests/data/fooinout.rc` should be almost identical.

2.7 cfg::inimanager

`cfg::inimanager` is meant to manage multiple `cfg::inisections` objects. Each `cfg::inisections` holds multiple `cfg::inisection` objects, and represent the setting in an `cfg::inifile`. Each INI file is associated with a unique configuration file type, such as "rc" or "session", and a section name, `cfg::inimanager` defines a `section` types, which is a a map with the key being a configuration type, such as "rc", and the value being an `cfg::inisections` object.

`cfg::inimanager` also contains a `cli::multiparser` which can map unique option names to the INI file and INI sections to which they belong. Not all options from an INI file will be useful from the command line, though.

The `ini_set_test` application first defines some *global* options peculiar to it: "–list", "–read", "–test", and "–write". These are *global* options because they are not associated with an INI file or an INI section. The `cfg::inimanager` is created with this list, and populates the options. Then two `cfg::inisection` objects are added, one for `cfg::small_data` and one for `cfg::rc_data`.

The caller can then exercise the global options, including the internal, stock options. The `-help` option will show all of the global options, plus all of the sections and options defined for `cfg::small_data` and `cfg::rc_data`. The `-read` and `write` options accept a file-name and process it.

Note: if something like the following message appears, fix the issue. Otherwise, the option cannot be parsed, even if it appears in the help text.

```
[iniset] Couldn't insert <sets-mode,(rc,[misc])>: Change option to a
unique name
```

Let us walk through some runs of `ini_set_test`.

```
$ ./build/test/ini_set_test --help
```

This command emits color-coded help text showing the options, a brief description of each, and the default value of each option. The text is extracted from the setup structures, as provided in the file `./tests/small_spec.hpp`:

```
inisection::specification small_misc_data
inisection::specification small_interaction_data
inisection::specification small_comments (stock comments)
inisection::specification small_cfg66_data (stock [Cfg66] section)
inisections::specification small_data
```

The last specification collects all of the others and adds directory and descriptions. Also included are all the specifications in the file `tests/rc_spec.hpp`, which we will not show here.

The first part of the help text shows the "global" options, which consist of internal stock options plus some additional test options defined by `cfg::options::container s_test_options` in `tests/ini_set_test.cpp`: "–list", "–read", "–write", and "–tests". The global options are not associated with either an INI file or an INI section.

Following the global options are specific options associated with an INI file and and INI section. Here's a partial example:

```
rc:[misc] A number of miscellaneous options
 --manual-ports          Show real system ports, not 'usr' port names.
                         [false]
 --port-naming=v         Port amount-of-label showing. [short]
```

The "rc" represents the INI file, which can have any name, but with an extension of `.rc`. For a given program, there can be only one INI file with that extension. The "[misc]" represents an INI section in that INI file.

The setup process is fairly simple once all of the INI specification structures have been coded. First, we add the global test options to the stock options. Then we add the `inisections` defined in the test "hpp" files.

```
cfg::inimanager cfg_set(s_test_options);
bool success = cfg_set.add_inisections(cfg::small_data);
if (success)
    success = cfg_set.add_inisections(cfg::rc_data);
```

Next, we need to create a `cli::multiparser` and parse the command-line:

```
cli::multiparser & clip = cfg_set.multi_parser();
success = clip.parse(argc, argv);
```

Now, not all of the global options have been implemented, so we describe only the ones that work.

```
$ ./build/test/ini_set_test --list --verbose
```

The "verbose" option does nothing, but the "list" option shows the current values of all options, and it stars the ones that have been modified from the command line: "verbose" and "list". We can add a "log" option to dump the list to a file:

```
$ ./build/test/ini_set_test --log=t.log --list --verbose
```

The next test option:

```
$ ./build/test/ini_set_test --test
```

That option is currently a minimal implementation. More to come.

```
$ ./build/test/ini_set_test --read=tests/data/fooin.rc
```

Note that the "=" can be replaced by a space.

This option depends on being able to find the the desired INI sections object in the INI manager and finding that it is active (it has entries). This INI sections object can then be used to determine what needs to be found in the INI file.

```
$ ./build/test/ini_set_test --write=t.rc
```

This option writes the current settings to `t.rcv` and `t.small` in the `tests/data` directory.

We can combine a number of options to read an INI file, list the data read to a log file, and write it to another file:

```
$ ./build/test/ini_set_test --list --log test.log
        --read tests/data/ini_set_test.rc --write ini-out.rc
```

The output in `test.log` and `ini-out.rc` should match up well with `tests/data/ini_set_test.rc`.

One more use case. Copy `tests/data/ini_set_test.rc` to a temporary file, `i.rc`. Then edit all of the comments (indicated by the hash mark, '#') so that they are obviously different. Then read that file and write it to another temporary file.

```
$ ./build/test/ini_set_test --read=i.rc --write=j.rc
```

Comparing `i.rc` and `j.rc`, one sees that `j.rc` retains the original comments, which ultimately came from the specification defined in the application. That is, the comments are not modifiable in the 'rc' file. (They aren't modifiable in *Seq66* either.

In the future it would be good to read in the comments as well and store them in the descriptions.

We can combine a number of options to read an INI file, list the data read to a log file, and write it to another file:

```
$ ./build/test/ini_set_test --list --log test.log
   --read tests/data/ini_set_test.rc --write ini-out.rc
```

The output in `test.log` and `ini-out.rc` should match up well with `tests/data/ini_set_test.rc`.

2.8 cfg::inisection

`cfg::inisection` contains a `specification` structure that provides the name of a section, it's description, and a container of `cfg::options`. It is represented 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.

It provides a number of functions to return option setting, help text, and more. Also provided are free functions to return a stock main configuration and a stock comments section.

2.9 cfg::inisections

The `inisections.hpp` file has been split, and no long declares four classes.

`cfg::inisections` 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::inisections` holds a list of `inisection` objects. It 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.

Functions are provided to pass along to the `cfg::inisection` objects and to look up INI sections and options. The various parts of a file-name (path, base-name, and file extension) are held, and can be used to get the full file-specification for reading and writing an INI file.

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

2.10 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»`.

2.11 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 `spec`s 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 good example of how to create a list of options.

2.12 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*.

2.13 cfg::recent

`recent` provides for the management of a list of recently-used items. It is implemented as a deque of strings.

Included are functions to get the most-recent file-name, a file-name by index (with optional removal of the path, for use in menu displays), and functions to manage the list. This is a feature adapted and extended 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`
- `cli::multiparser`

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. This parser is meant for simple usage; it has no support for accessing configuration files, and supports just one set of options.

3.3 cli::parser / Global Options

The `cli::parser` also provides support for options that can be command to all applications. These stock options are called "global" options, since they are not associated with any configuration file.

- `-version`. Provides the version of the application.
- `-help`. Emits help text. This help text is assembled from the specification structures set up according to the `cfg::options::container`.
- `-description`. This option is meant to show the description fields of the options.
- `-verbose`. Indicates to emit additional information about the run of the application.
- `-inspect`. This option is meant to activate debugging code. The applications determines what this means.
- `-investigate`. This option is meant to activate temporary debugging code for the current purpose desired by the programmer. The applications determines what this means.
- `-log`. This option enables a log file for recording error and information output.

Additional global options can be added, as illustrated in the `ini_set_test` program.

The `parse()` function looks for stock option such as `-help` and `-option log filename`. The - sequence terminates a list of options.

It is meant to be similar to `getopt(3)`, but much more flexible and perhaps easier to set up.

The caller can call the following member functions:

- `show_information_only()`. This function returns a boolean value that is true if help, description, or version were requested. Generally, if this information is shown on a command-line, the application does nothing but show the desired information, and exits.
- `version_request()`. Indicates if version information was requested.
- `help_request()`. Indicates if application help was requested. The command-line help text is shown. The caller can also provide further help if desired.
- `description_request()`. Indicates if description information was requested.
- `verbose_request()` and `verbose()`. Indicates if extra output (either command line or via dialog boxes) was requested.
- `inspect_request()`. Indicates if inspection was specified was requested. Generally can be used to output data values as appropriate.
- `investigate_request()`. Indicates if investigation was specified was requested. Defined by the application.
- `use_log_file()` and `log_file()`. Indicates that standard I/O should be redirected to a log file.

There are functions to add, verify, and set option values:

- `add()`.
- `verify()`.
- `set_value()`.
- `change_value()`.
- `clear()`.
- `unmodify()`.

- `unmodify_all()`.

There are also functions to test and retrieve option values:

- *string*. All options are encoded as strings. Strings can be retrieved using `cli::parser::value()`. The default string can be retrieved by `default_value()`.
- *boolean*. `is_boolean()` tests for a value being boolean.
- *other, to do*. There are many more value-related functions in the `cfg::options` class, but they use a reference to a `cfg::options::spec` structure, and can be accessed only by getting the option and spec reference. See section 2.11 "cfg::options" on page 11.

The format of the default value is either a simple string showing the default value, or a *range* string such as "0<=10<=99", where the middle value is the default value.

3.4 cli::parser / Usage

This section provides a walk through the test application `cliparser_test`. It starts with an `appinfo` function to see the name that will appear in console messages:

```
cfg::set_client_name("cli");
cfg::set_app_version("0.2.0");
```

Next, a parser is created via:

```
cli::parser clip(s_test_options);
```

The `s_test_options` list is defined in the `test_spec` header file. Here is an excerpt. Note that we could use an anonymous namespace instead of the keyword `static`.

```
static cfg::options::container s_test_options
{
    {
        {
            "alertable",
            {
                'a', cfg::options::kind::boolean, cfg::options::enabled,
                "false", "", false, false,
                "If specified, the application is alertable.", false
            }
        },
        {
            "canned-code",
            {
                'c', cfg::options::kind::boolean, cfg::options::enabled,
                "true", "", false, false,
                "If specified, the application employs canned code.", false
            }
        },
        . . .
    }
};
```

As an example, this list provides the `-a` and `-alertable` option, which is a boolean option and is enabled. It has a default value of "false", which is set once the list is initialized. The values for "set-from-cli" and "dirty" are false by default at first. After the one-line description, the boolean indicates if the options is a built-in (global, stock) option. That value is set only in the `global_options()` function in the `options` module. The options in this list are added to the global options.

After construction, the parser can be applied to the command-line arguments:

```
 bool success = clip.parse(argc, argv);
```

This can modify (set the value, raise the "set-from-cli" and "dirty" flags) options based on the command-line.

Not all options will have a code, in general, especially if there are a large number of options. Duplicates are checked for. The existing codes can be listed:

```
std::string msg = "Option codes: " + clip.code_list();
```

The command line can also be scanned to see if an option is present. The last parameter indicates if the option is required to exist:

```
bool findme_active = clip.check_option(argc, argv, "find-me", false);
```

Built-in (global) options that make the application show something and then quit can be tested:

```
if (clip.show_information_only())
{
    if (clip.help_request()) . . .
    if (clip.description_request()) . . .
    if (clip.version_request()) . . .
}
```

Some other built-in options:

```
if (clip.verbose_request()) . . .
if (clip.inspect_request()) . . .
if (clip.investigate_request()) . . .
if (clip.use_log_file()) . . .
```

Debug text can be shown; it lists all the options, their values, their default values, and if the value has been modified:

```
std::string dbgtxt = clip.debug_text(cfg::options::stock);
```

Values can also be set or changed by the application itself, rather than from the command-line. It can be retrieved by the long option name or the option code.

```
success = clip.change_value("username", "C. Ahlstrom");
std::string name = clip.value("u");    // or "username"
```

```
success = name == "C. Ahlstrom";
```

If an error occurs, which can be checked by a function's return code or by a call to `cli::parser::error_msg()`, then the message can be retrieved for display:

```
   std::string errmsg = clip.error_msg();
   util::error_message(errmsg);
```

3.5 cli::multiparser

The `cli::multiparser` class extends `cli::parser` in a number of ways, in order to support a suite of command-line options covering mutltiple configuration files.

Support is provided to look up the long option name from an option code character. (The `cfg::options` class also provides this, but it is not directly exposed to `cli::parser`.

```
using codes = std::map<char, std::string>;
codes & code_mappings();
```

This function is used in the `multiparser` override of the the virtual `cli::parser::parse()` function.

In order to support multiple configuration files and multiple configuration sections, we need to way to find out in which file and section an option resides.

```
using duo = struct
{
   std::string config_type;
   std::string config_section;
};
using names = std::map<std::string, duo>;
names & cli_mappings();
```

This function is used to find the desired option set, a pointer to the `cfg::options` for a particular file and section.

For a walk-through, see the section concerning the "INI set test", section 2.7 "cfg::inimanager" on page 7.

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::bytevector class`
- `util::filefunctions module`
- `util::msgfunctions module`
- `util::named_bools class`
- `util::strfunctions module`

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

5.1 util::bytevector

The `util::bytevector` class provides an `std::vector` of "bytes" (`unsigned char`) with functions to put bytes into the vector and read them out. There are also functions to read a file and write the vector to the file.

The bytes are treated as a stream of big-endian data. Integers are extracted from the bytes a byte at a time, starting with the most significant byte. Since this data is big-endian, it is suitable for use with MIDI files and network data streams.

This module defines some types in the `util` namespace:

- `byte` = `uint8_t`
- `ushort` = `uint16_t`

- `ulong` = `uint32_t`
- `ulonglong` = `uint64_t`
- `bytes` = `std::vector<byte>`.

The `util::bytes` type holds the big-endian data. The `util::bytevector` class keeps track of the section of data being processed, and the position in the data. In addition, errors are detected, and an error message is stored for later display.

Various overloads of the `assign` member function are used for loading data into the byte-vector. Functions are provided to "get" and "peek" at items in the byte-vector. The former change the position value, while the latter do not. Functions are provided to "put" and "poke" items into the byte-vector. The former change the position value, while the latter do not.

Lastly, "read" and "write" functions are provided for interactions with a data file.

5.2 util::filefunctions module

The `util::filefunctions` module contains a large number of function dealing file-names and files. The file-name functions are useful for building paths and for splitting paths into parts. The file functions are basically wrappers around the `C FILE *` API. Also provided are functions to check file attributes, to read/write strings, and to open, close, copy, and append data.

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

5.3 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 represent whether the message is an error, warning, debug, status, file-related, or session message.

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

Setters and getters are provided for the `-verbose` and `-investigate` options of the command-line parser.

Again, skim the `util::msgfunctions` module for details.

5.4 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 options-related classes in the `cfg` namespace.

5.5 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`.

Skim the `util::strfunctions` module for details.

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:

- texttctrl_spec.hpp
- textttdrums_spec.hpp
- textttmutes_spec.hpp
- textttpalette_spec.hpp
- textttplaylist_spec.hpp
- textttrc_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:

- texttctrl_spec.hpp
- texttdrums_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`. Start with page 62. 1994.

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

Index