INICPP Design Overview

Design overview

Rehearsal

For basic inspiration we were looking for libraries which can parse configuration files. We targeted mainly C++ ones, but some other (C, C#, Python) could be useful as well.

- **YAML-CPP** (link)

This is probably the most friendly to use C++ configuration parser we've seen packed with all needed advanced features. We use it a lot and had no major issue with it so far. It's written in modern C++ with good integration with STL. Also, reasonably good documentation and support on StackOverflow make this library very popular. This project is our best inspiration.

- **jsoncpp** (link)

Popular and lightweight C++ library for interacting with JSON formatted files. Advantage of this project is possibility to use it as single `.h` file, which is pretty nice. Newer versions have also C++11 support. Support of comments is present despite lack of standard, but on the other hand JSON schema validation is missing. Documentation is quite great.

- **inih** (link)

Originally C library with simple C++ wrapper. It's simple and fits easily into embedded devices. But no schema validation is supported.

- **configparser** (link)

Standard Python module for parsing INI-like configuration files. It's easy to use and configure. It has some nice features like link support (`${Section:Option}` or `%(option)s` syntax). But schema validation is not supported - probably it has no sense in Python. Documentation has also high standards.

Decisions and specification

Following text specify more details about our decisions about API proposal.

- **classes**

Splitting this problem into classes is quite straightforward. Main class with loaded configuration is `config`, which holds collection of `section` classes, each containing collection of `option` classes. `parser` has only static methods to read and write configuration. Schema for validation has similar separation. There are some other classes, but not a part of public API.

- **load/store formats**

In addition to assignment, *inicpp* can read configuration from file, input stream and string. Write can be performed to output stream of file. Schema (default configuration) can be written to a file or output stream the same way as normal configuration. Great feature is possibility to save schema and loaded configuration together, which will store loaded values together with comments and default values from schema.

Format for storing values is following: - boolean - `yes` for truth, `no` otherwise - signed, unsigned - just the value in decimal base - float - value in IEEE 754 standard - enum - string representation of the value - string - properly escaped string value - list of values - comma (`,`) separated values following above mentioned guidance

We don't bother much about saving configuration in similar format as it was loaded. In our opinion, configuration is mostly written by hand and then only read by computers (which don't override it). On the other hand, if you generate the configuration by computer, it's probably large and no one will read or edit it by hand and for computers exact format doesn't matter. But if you combine printing configuration with corresponding schema, you'll get pretty neat, human readable config file.

- **links**

Links are just text replacement. They are processed during loading of configuration. To avoid problems with cyclic dependencies, link can point only to location already known (eq. up in configuration file). *Inicpp* will not preserve links when you load and store file with them. Also, links cannot be created at runtime (technicaly they can be created as basic string value, but no evaluation is performed). If someone has tuned config with links, he won't let any library to update it. Otherwise machine created configuration needs no links, computers don't care and human will barely read computer generated files.

- **comments**

Inicpp will not read comments from input file and will not store them when configuration is saved. Reasons are similar to previous bullet. In our opinion, comment are important only for human created configurations.

- **types**

Types are mapped to native C++ types as close as possible. For pleasant usage there are typedefs for each of the types as well as enumeration value for specifying the type to schema.

INI type	C++ native type	inicpp typedef	inicpp option_type
boolean	bool	boolean_ini_t	option_type::boolean_e
signed	int64_t	signed_ini_t	option_type::signed_e
unsigned	uint64_t	unsigned_ini_t	option_type::unsigned_e
float	double	float_ini_t	option_type::float_e
enum	internal_enum_type	enum_ini_t	option_type::enum_e
string	std::string	string_ini_t	option_type::string_e

Enum values are represented by custom type. Basicaly, `enum_ini_t` type could be `std::string`, but it's not possible because of already existing `string_ini_t` type. `enum_ini_t` has constructor from `std::string` and also conversion operator to the same type, so it could be easily used. Also, overloaded equality operator is provided.

- **lists**

All items in list must be of the same type. One-element option can be treated as a list with one item. There are methods to add or remove item from that list. List is allways `std::vector` with all it's advantages and disadvantages (like complexity of removing item from the middle).

When both `,` and `:` symbols are present in input configuration, `,` has higher priority and will be used as a delimitier.

- **value range validation**

When creating an option schema, functor for validation values can be specified. It's a type (class with overloaded `operator()`, function, lambda, ...) taking one argument (the value to be validated) and returning boolean value if it's valid or not. This will provide great extensibility and configurability.

- **config validation**

Configuration can be validated directly in loading process or at any time after. When loading configuratino without schema, all values are stored as strings, but runtime conversion can be performed. After validation, all values are stored as their proper type with optional runtime conversion to string.

If validation fails, it's guaratied that configuration is in consistent state, option before the error are already validated

and options after that are left as is. After fixing the problem, you can continue with validation (already validated options can be revalidated without issues).

- **exceptions**

In modern programming is good practise to use exceptions instead of error return values. Code with exceptions can be cleaner and thrown exception cannot be ignored, so all error states must be explicitely solved. We will use own set of exceptions, which will be derived from standard base class `std::exception`.

- **documentation**

This project is located on GitHub, so documentation will keep standards of that platform. Short overview with build instructions and examples is in README.md file at root directory of source code. User documantation will be stored as a wiki within the project repository. Generated documentation will be using Doxygen, which is standard for C++ language. Html format will be located here.

- **building**

Inicpp is built using `cmake`. Nowadays it's very popular and multiplatform build tool. Cmake can generate project for native tools like GNU Makefile on Linux of Visual Studio solution for example. Using this tool greatly improves reusability and interoperability with other projects. Detailed instructions can be found in README.

The code is mainly tested with the newest stable GCC compiler (6.1.1) and current version of Microsoft Visual C++ Compiler (2015), but any other C++ compiler with C++14 support should be ok.

- **unit testing**

For unit testing we have chosen Google Test framework, which is one of the best ones for C++ language. For simple usage, we provide a copy as a git submodule in our own repository.