INICPP

INICPP wiki

inicpp is a C++ parser of INI files with schema validation.

Main features

With **inicpp** library you can do following:

- Specify format of configuration file (*schema*)
  - For every section should be defined:
    - identifier
    - mandatory/optional
    - comment with short description of this section
  - For every option in a section should be defined:
    - identifier
    - mandatory/optional
    - one element or list of elements
    - type ofthe elementu and restriction of valid values
    - default value when the option is optional
    - comment (describing valid values)
- Load configuration file in two modes:
  - **strict** – content must be valid according to given *schema*, exeption is thrown otherwise
  - **relaxed** – unknown section and options are read as strings, known section have full validity check
- Access to loaded configuration with possibily to modify values
- Write configuration to file (try as similar as loaded configuration, eq. not write default values which wasn't present)
- Write default configuration (*schema*) to file including comments
- Read/write from/to stream

Getting started

If you want to use this library, you should first read [[INI format specification]] and make sure it fits your needs. It not, feel free to fork this project, improve it and alternatively send us a pull request.

Building

Building of the library itself is described in README in root of project source tree. You can choose if you need dynamic linked library, static linked library or both.

For extra assurance you can also run our unit tests, but there shouldn't be any problem.

Getting know the API

For basic high-level knowledge of library's features, please read [[Design overview]] page. Then you should look at our examples, compile, run and understand them. After that, you will have pretty good overview of the API and you should be able to use the library in your own projects. Specific information then can be found in programmer's documentation online.

Including into other projects

Inicpp library is easy to include into other projects. All code is in `inicpp` namespace, so you won't have any naming issues. All needed header files are located in `include/inicpp/` directory in source tree. There is also `inicpp.h` header, which includes all dependencies, so this is only include you need to use in your sources.

Cmake project

Building `inicpp` inside other `cmake` project is super easy. In your `CMakeLists.txt` file you need to add these files (assume this library sources are located in `vendor/inicpp/` directory and your project's name is *foo*):

```
include_directories(vendor/inicpp/include)
target_link_libraries(foo inicpp)
add_subdirectory(vendor/inicpp)
```

Then in your sources you need just to include headers like this and start coding:

```
#include "inicpp/inicpp.h"
```

Other

For other projects, build the library as in README and link it with your executable as any other library.

Specification of .ini format

- Comment
  - text from `;` symbol to the end of line
- Identifier
  - string of characters from `a-z, A-Z, 0-9, _, ~, -, ., :, $, space` begining with one of `a-z, A-Z, . , $, :`
- Section
  - part of configuration file marked with identifier
  - starts with a header, which is identifier of the section in squared brackets at the begining of line, for example `[Defaults]`
  - ends at begining of following section or end of file
  - cannot be splitted to more parts, every identifier can be present only once in the config file
  - contains 0 or more configuration options
- Option
  - pair of (identifier, value) in any section
  - in file written as `identifier=value`
  - white spaces at the begining and end of identifier or value are stripped unless preceded with `\`
  - spaces between words (characters without spaces) are part of identifier or value
  - value is represented by one or more elements of the same type with separator `,` or `:`. When both separators are present, `,` has higher priority and will be used.
- Element
  - text representing value paired with identifier of option
  - text of the element can be link to other element (also in different section)
    - format of the link is `${section#option}`

        * value is only text replacement, so special characters and commands will be interpreted after insertion
        * link is not interpreted when `\` is preceding of `$` character
    - element can be one of the types: `boolean`, `signed`, `unsigned`, `float`, `enum` or `string`
        * **boolean** element can be `0`, `f`, `n`, `off`, `no`, `disabled` for *false* value and `1`, `t`, `y`, `on`, `yes`, `enabled` for *true* value
        * **signed** type is 64-bit value with limitations of the type (in 2 complement)
        * **unsigned** is 64-bit integer value
        * **signed** and **unsigned** specifies decimal numbers by default. Hexadecimal numbex must be prefixed with `0x`, octal with `0` and binary with `0b`
        * **float** is 64-bit floating point type (IEEE 754 standard)
        * **enum** type is one of former defined set of strings
        * **string** can contain arbitrary characters except `,`, `:` and `;`, which must be prefixed with `\`

Example of valid .ini file

```
[Section 1]
; comment
Option 1 = value 1                     ; option 'Option 1' has value 'value 1'
oPtion 1    =  \ value 2\ \ \          ; option 'oPtion 1' has value ' value 2   ', 'oPtion 1' and 'Option

[$Section::subsection]                 ; no subsection, only valid identifier of section
Option 2=value 1:value 2:value 3       ; option 'Option 2' is list of 'value 1', 'value 2' and 'value 3'
Option 3 =value 1, ${Section 1#Option 1} ; option 'Option 3' is list of 'value 1' and 'value 1'
Option 4= v1,${$Section::subsection#Option 3},v2 ; option 'Option 4' is list of 'v1', 'value 1', 'value 1'
Option 5= v1, v2:v3                    ; option 'Option 5' is list of 'v1' a 'v2:v3'

[Numbers]
num = -1285
num_bin = 0b01101001
num_hex = 0x12ae,0xAc2B
num_oct = 01754

float1 = -124.45667356
float2 = +4.1234565E+45
float3 = 412.34565e45
float4 = -1.1245864E-6

[Other]
bool1 = 1
bool2 = on
bool3=f
```

Used C++ language features

Inicpp project is using a lot of new C++ features, mostly from C++11 and C++14 standards. Let's introduce some of them:

- **using directive**

Modern way of `typedef` is use of `using` keyword. So `using vec = std::vector<int>;` and `typedef std::vector<int> vec` are equivalent.

- **smart pointers**

Using smart pointers for memory management is a great practise to avoid memory leaks. We are using both `std::unique_ptr` and `std::shared_ptr`. Creating is performed by `std::make_unique` and `std::make_shared` functions, which should be better than assigning raw pointer to newly allocated memory into one of the smart pointers. `std::make_unique` is part of C++14, `std::make_shared` was also in C++11.

- **literals**

From C++11 you can define your own literals (suffixes to types). Literal is defined as `operator "" suffix()` function. Suffix `s` to `const char *` value is used in inicpp, which returns `std::string` object. This suffix is in `std::literals` namespace from C++14. For example, if you have `auto str = "hello"s;`, `str` variable will be of type `std::string` (because `"hello"s` is of the `std::string` type).

- **type traits**

For retrieving some information about type argument in templates could be used type traits. Comparing type argument againts other type is done by `std::is_same<T, U>` function (since C++11).

- **enum class**

Enum classes are quite known features of C++11. It allows you write type safe enumerations. Small example: `enum class option_item: bool { single, list };` says, that `option_item` is enumeration with 2 values with underlying type `bool`. Values are used as `option_item::single`.

- **brace initializer list**

Brace initializer list can be used to easily initialize some object, mostly `struct`. For example, we have `struct A {int a; int b; };`. Initialization of such object could be done as `A a = { 1, 5 };` of better `A a { 1, 5 };`. In inicpp context, section schema konstructor takes a structure with some data. In GCC (tested 5.3 and 6), following is allowed:

```
section_schema sect_schema({ "section_name", item_requirement::mandatory, "comment" });
```

But for some reason, this can't be done in MSVC++ compiler yet. More fun is with option schema, where required data structure is derived from some base structure. Simple brace initialization is not possible this way, but this will change with C++17 standard, where following will be possible:

```
option_schema_params<string_ini_t> opt { {"opt", item_requirement::mandatory,
    option_item::single, "default value", "opt comment"}, nullptr};
```

- **move semantics**

One of the greatest features of C++11 standard is move semantics. We try to use it when possible to allow compiler gain some more speedup to the library.