

# Flags-Template

*name: csl*

*E-Mail: [3079625093@qq.com](mailto:3079625093@qq.com)*

```
1  _|| _|
2  _| _| _||| _||| _|||
3  _||| _| _| _| _| _||
4  _| _| _| _| _| _||
5  _| _| _||| _||| _||| _|
6      _|
7      _||
8
9  _| _| _|
10 _||| _|| _||| _|| _||| _| _||| _||| _|||
11 _| _||| _| _| _| _| _| _| _| _|||
12 _| _| _| _| _| _| _| _| _| _|
13 _|| _||| _| _| _| _||| _| _||| _| _|||
14      _|
15      _|
```

[1. Overview](#)

[2. Structure](#)

[3. Usage](#)

[Example for Source Code](#)

[Output](#)

[4. Apis](#)

[Argument Types](#)

[Option Property](#)

[Option](#)

[OptionParser](#)

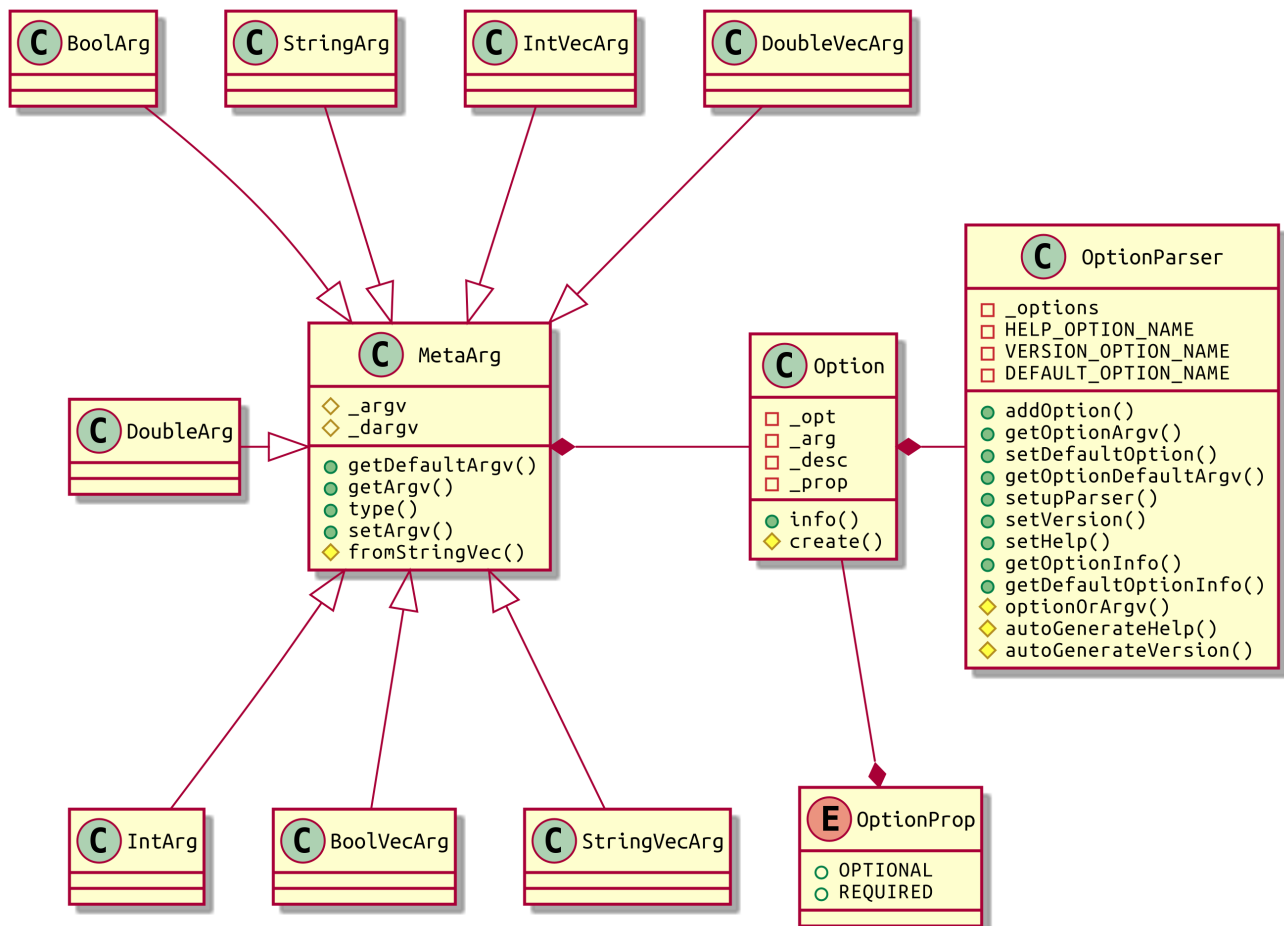
## 1. Overview

this is a simple 'program-command-line-parameter-parsing' library using cpp-template.

the main functions:

- Add command line parameters to the specified program and set the relevant properties of the command line parameters;
- Parse the passed in parameters based on the set command line parameters;
- During parsing, identify and check the command line parameters (such as wrong type, wrong option name, inconsistent selectability);

## 2. Structure



### 3. Usage

#### Example for Source Code

```

1  #include "flags.hpp"
2
3  using namespace ns_flags;
4
5  int main(int argc, char const *argv[]) {
6      /**
7       * @brief try-catch is not necessary but it is strongly recommended,
8       * because you can get a lot of advice when there are errors in your code
9       */
10     try {
11         OptionParser parser;
12         /**
13          * @brief define some kinds of arguments
14          * [int, std::string, bool, double]
15          * std::vector<int, std::string, bool, double>
16          */
17         parser.addOption<IntArg>("id", 0, "the id of current thread");
18         parser.addOption<StringArg>("usr", "null", "the name of usr");
19         parser.addOption<BoolArg>("sex", true,
20             "the sex of usr [male: true, female: false]");
21         parser.addOption<DoubleArg>("height", 1.7, "the height of usr", OptionProp::REQUIRED);
22         parser.addOption<IntVecArg>("ids", {1, 2, 3}, "the ids of threads");
    
```

```

23     parser.addOption<StringVecArg>("langs", {"cpp", "python"},
24                                     "the used languages of usr");
25     parser.addOption<BoolVecArg>("choice", {true, false}, "the choice of usr");
26     parser.addOption<DoubleVecArg>("scores", {2.3, 4.5}, "the score of usr");
27     /**
28      * @brief set version and help docs
29      * @attention if you do not set the help docs, then the help docs
30      * will generate automatically
31      */
32     parser.setVersion("2.0.0");
33     // parser.set_help("");
34
35     parser.setDefaultOption<StringVecArg>({""}, "the default option", OptionProp::REQUIRED);
36     /**
37      * @brief finally, you can set up the parser and then use these arguments
38      */
39     parser.setupParser(argc, argv);
40
41     /**
42      * @brief print the info of arguments
43      */
44     std::cout << parser << std::endl;
45     std::cout << parser.getDefaultOptionInfo<StringVecArg>() << std::endl;
46
47     /**
48      * @brief use the arguments
49      */
50     auto id = parser.getOptionArgv<IntArg>("id");
51     std::cout << "the 'id' I get is: " << id << std::endl;
52 } catch (const std::exception &e) {
53     std::cerr << e.what() << '\n';
54 }
55 return 0;
56 }

```

## Output

if you want to over view the example log file for command lines and outputs, please click [the log file](#).

if run command line:

```
1 | ./flags hello "I'm" flags!
```

will output:

```
1 | [ error from 'libflags'-'setupParser' ] the option named '--height' is 'OptionProp::REQUIRED', but you
   | didn't use it
```

if run command line:

```
1 | /flags hello "I'm" flags! --height 98.8 --sex true --usr csl --id 12 --choice true false true --ids 12
   | 34 123 --scores 12.3 45.6 78.9 --langs cpp java python html
```

will output:

```
1 | OptionParser Info: {
2 |   {'opt': def-opt, 'type': StringVecArg, 'desc': the default option, 'prop': Required};
3 |   {'opt': choice, 'type': BoolVecArg, 'desc': the choice of usr, 'prop': Optional};
4 |   {'opt': ids, 'type': IntVecArg, 'desc': the ids of threads, 'prop': Optional};
5 |   {'opt': scores, 'type': DoubleVecArg, 'desc': the score of usr, 'prop': Optional};
6 |   {'opt': lans, 'type': StringVecArg, 'desc': the used langusges of usr, 'prop': Optional};
7 |   {'opt': height, 'type': DoubleArg, 'desc': the height of usr, 'prop': Required};
8 |   {'opt': sex, 'type': BoolArg, 'desc': the sex of usr [male: true, female: false], 'prop':
Optional};
9 |   {'opt': usr, 'type': StringArg, 'desc': the name of usr, 'prop': Optional};
10 |   {'opt': version, 'type': StringArg, 'desc': display the version of this program, 'prop': Optional};
11 |   {'opt': id, 'type': IntArg, 'desc': the id of current thread, 'prop': Optional};
12 |   {'opt': help, 'type': StringArg, 'desc': display the help docs, 'prop': Optional};
13 | }
14 | {'opt': def-opt, 'type': StringVecArg, 'desc': the default option, 'prop': Required, 'default': [],
'value': [hello, I'm, flags!]}
15 | the 'id' I get is: 12
```

if run command line:

```
1 | ./flags --help
```

will output:

```
1 | Usage: ./flags [def-opt target(s)] [--option target(s)] ...
2 |
3 |   Options      Property      Type      Describes
4 |   -----
5 |   --def-opt    Required      StringVecArg  the default option
6 |
7 |   --choice     Optional      BoolVecArg   the choice of usr
8 |   --ids        Optional      IntVecArg    the ids of threads
9 |   --scores     Optional      DoubleVecArg the score of usr
10 |  --lans        Optional      StringVecArg the used langusges of usr
11 |  --height     Required      DoubleArg    the height of usr
12 |  --sex        Optional      BoolArg      the sex of usr [male: true, female: false]
13 |  --usr        Optional      StringArg    the name of usr
14 |  --id         Optional      IntArg       the id of current thread
15 |
16 |  --help       Optional      StringArg    display the help docs
17 |  --version    Optional      StringArg    display the version of this program
18 |
19 | help docs for program "./flags"
```

if run command line:

```
1 | ./flags --version
```

will output:

```
1 | ./flags version: 2.0.0
```

if run command line:

```
1 | ./flags --nema 12
```

will output:

```
1 | [ error from 'libflags'-'setupParser' ] the option named '--nema' is invalid
```

## 4. Apis

### *Arguement Types*

Here are the types you can use in the 'arguement-parser':

```
1 | class IntArg : public MetaArg {
2 |     ...
3 | };
4 |
5 | class DoubleArg : public MetaArg {
6 |     ...
7 | };
8 |
9 | class BoolArg : public MetaArg {
10 |     ...
11 | };
12 |
13 | class StringArg : public MetaArg {
14 |     ...
15 | };
16 |
17 | class IntVecArg : public MetaArg {
18 |     ...
19 | };
20 |
21 | class DoubleVecArg : public MetaArg {
22 |     ...
23 | };
24 |
25 | class BoolVecArg : public MetaArg {
26 |     ...
27 | };
28 |
29 | class StringVecArg : public MetaArg {
30 |     ...
31 | };
```

### *Option Property*

```

1 enum class OptionProp {
2     /**
3      * @brief options
4      */
5     OPTIONAL,
6     REQUIRED
7 };

```

## Option

```

1
2 class Option {
3     template <typename ArgType>
4     std::string info() {
5         ...
6     }
7
8     /**
9      * @brief create an option
10     *
11     * @tparam ArgType the argument class type. eg: IntArg, DoubleArg
12     * @param optName the option's name
13     * @param defaultArgv the default argument value
14     * @param desc the describe of the argument
15     * @param prop the prop of option
16     * @return Option
17     */
18     template <typename ArgType>
19     static Option create(const std::string &optName, const typename ArgType::value_type &defaultArgv,
20                        const std::string &desc, OptionProp prop = OptionProp::OPTIONAL) {
21         ...
22     }
23
24 private:
25     std::string _opt;
26     std::shared_ptr<MetaArg> _arg;
27     std::string _desc;
28     OptionProp _prop;
29 };

```

## OptionParser

```

1 class OptionParser {
2 public:
3     OptionParser() {
4         ...
5     }
6
7 public:
8     /**

```

```

9      * @brief add an option to the option parser
10     *
11     * @tparam ArgType the argument class type. eg: IntArg, DoubleArg
12     * @param optName the option's name
13     * @param defaultArgv the default argument value
14     * @param desc the describe of the argument
15     * @param prop the prop of option
16     * @return OptionParser&
17     */
18     template <typename ArgType>
19     OptionParser &addOption(const std::string &optName,
20                           const typename ArgType::value_type &defaultArgv,
21                           const std::string &desc,
22                           OptionProp prop = OptionProp::OPTIONAL) {
23         ...
24     }
25
26     /**
27     * @brief add an option to the option parser
28     *
29     * @tparam ArgType the argument class type. eg: IntArg, DoubleArg
30     * @param defaultArgv the default argument value
31     * @param desc the describe of the argument
32     * @param prop the prop of option
33     * @return OptionParser&
34     */
35     template <typename ArgType>
36     OptionParser &setDefaultOption(const typename ArgType::value_type &defaultArgv,
37                                   const std::string &desc = "the default option",
38                                   OptionProp prop = OptionProp::OPTIONAL) {
39         ...
40     }
41
42     /**
43     * @brief get the default argument value of the option
44     *
45     * @tparam ArgType the argument class type. eg: IntArg, DoubleArg
46     * @param optName the option's name
47     * @return ArgType::value_type
48     */
49     template <typename ArgType>
50     typename ArgType::value_type getOptionDefaultArgv(const std::string &optionName) {
51         ...
52     }
53
54     /**
55     * @brief get the default argument value of the option
56     *
57     * @tparam ArgType the argument class type. eg: IntArg, DoubleArg
58     * @param optName the option's name
59     * @return ArgType::value_type
60     */
61     template <typename ArgType>
62     typename ArgType::value_type getOptionArgv(const std::string &optionName) {
63         ...
64     }
65
66     /**
67     * @brief get the default argument value of the option
68     *
69     * @tparam ArgType the argument class type. eg: IntArg, DoubleArg

```

```

70     * @param optName the option's name
71     * @return ArgType::value_type
72     */
73     template <typename ArgType>
74     typename ArgType::value_type getDefaultOptionArgv() {
75         ...
76     }
77
78     /**
79     * @brief set up the parser
80     *
81     * @param argc the count of the argument
82     * @param argv the value of the argument
83     * @return OptionParser&
84     */
85     OptionParser &setupParser(int argc, char const *argv[]) {
86         ...
87     }
88
89     /**
90     * @brief set the version string
91     *
92     * @param version the string
93     * @return OptionParser&
94     */
95     OptionParser &setVersion(const std::string &version) {
96         ...
97     }
98
99     /**
100    * @brief set the help string
101    *
102    * @param help the string
103    * @return OptionParser&
104    */
105    OptionParser &setHelp(const std::string &help) {
106        ...
107    }
108
109    /**
110    * @brief get the option's info
111    *
112    * @tparam ArgType the type of argument
113    * @param optionName the name of option
114    * @return std::string
115    */
116    template <typename ArgType>
117    std::string getOptionInfo(const std::string &optionName) {
118        ...
119    }
120
121    /**
122    * @brief get the default option's Info
123    *
124    * @tparam ArgType the type of argument
125    * @return std::string
126    */
127    template <typename ArgType>
128    std::string getDefaultOptionInfo() {
129        ...
130    }

```



```
131
132 private:
133     std::unordered_map<std::string, Option> _options;
134
135     const std::string HELP_OPTION_NAME = "help";
136     const std::string VERSION_OPTION_NAME = "version";
137     const std::string DEFAULT_OPTION_NAME = "def-opt";
138 };
```