

Command Line Parser – documentation

(paramParser)

1 Introduction

The *paramParser* class or library offers you a simple way of argument parsing for command line executables, including even simple file parsing. This document explains how to use *paramParser* by ways of class documentation and code examples. *paramParser* allows for a maximum of two identifiers for each argument; the idea behind this is to offer longer options (e.g. “-name test”) as well as short options (e.g. “-n test”). *paramParser* is used as both the name of the class and the name of the complete implementation. Arguments will be called parameters within the description synonymously! You might want to consider Section 3 for starters, as via an example it explains probably all that is necessary for a quick start.

2 Class overview

Alltogether there are three classes: *paramObj*, *parseObj* and *paramParser*. The first one, *paramObj*, is a small class used internally within the other two classes and represents, as the name suggests, structured objects for argument description. *paramParser* is a class handling all possible arguments the command line accepts and *parseObj* is the connection between the abstract declaration of the parser by *paramParser* and the command line arguments given.

Class *paramObj*

public methods:

- **paramObj(string)**
Create parameter object with just one parameter name. min, max will be set to 0.
- **paramObj(string, string)**
Create parameter object with two possible parameter names. min, max will be set to 0.
- **paramObj(string, string, int)**
Create parameter object with two possible parameter names. min, max will both be set to the given integer value.
- **paramObj(string, string, int, int)**
Create parameter object with two possible parameter names. min, max will be set to the given integer values.
- **paramObj(string, int)**
Create parameter object with just one parameter name. min, max will both be set to the given integer value.
- **paramObj(string, int, int)**
Create paramObj with just one parameter name. min, max will be set to the given integer values.

- **void init(string, string p2="", int mi=0, int ma=0)**
Supermethod substituting all the above mentioned constructors. This is the one to be called.
- **paramObj& operator=(const paramObj&)**
Equalizes parameter name(s) and min, max.
- **int getMinNumberOfValues()**
Returns min, the minimal number of values for the current parameter.
- **int getMaxNumberOfValues()**
Returns max, the maximal number of values for the current parameter.
- **string getParamName1()**
Returns par1, the first parameter name.
- **string getParamName2()**
Returns par2, the second parameter name.
- **bool used()**
Returns used, an indicator whether the current parameter was used, i.a.w. whether or not it was given by command line or option file(s).
- **int instances()**
Returns inst, the number of instances of the same parameter type given within the command line and option file arguments.

protected methods and variables:

- **void setUsed()**
Marks a parameter as used, i.o.w. as given by command line or option file arguments. Should only be called by the friend objects of class paramParser.
- **string par1**
1st parameter name (suggestion: long style).
- **string par2**
2nd parameter name (suggestion: short style).
- **int min**
minimal number of values for current parameter.
- **int max**
maximal number of values for current parameter
- **bool isUsed**
indicator of the current parameter being used, i.o.w. being given by command line or option file arguments
- **int inst**
number of instances of this parameter given by command line or option file arguments.

Class *paramParser*

public methods:

- **paramParser()**
Constructor to be called.
- **void init()**
Initialises a paramParser object and is called by the constructor. This method will probably be protected in future releases.
- **void addParam(arguments)**
'arguments' is any of the combinations of string, int of paramObj constructors, see above for instructions.
- **paramObj* findParameter(string s1, string s2="")**
Searches for a paramObj by string value. Returns a pointer to the object, if s1 (OR s2) matches any of the two parameter names par1, par2 of possible parameters, NULL there is none such.
s1 will be searched for first! If no match can be found, s2 will be searched for. Thus there is no integrity check done here, whether s1 and s2 are referring to the same parameter!
- **parseObj* findOption(string s1, string s2="")**
Searches for a command line argument. Same rules as for findParamater apply here. In case of success, a pointer to the NEXT occurrence is returned. The search starts from the current position, which is either the start of the list or the result from a previous search. If the end of the argument list has been reached, the search pointer will be reset to the start of the list.
- **parseObj* getOption()**
Returns a pointer to the current (command line) argument object. parseObj* getNextOption(). Returns a pointer to the next argument object.
- **void restartParsing()**
This method resets the pointer of the parseObj list (this is the pointer representing the command line arguments) to its start. Be aware that this method does NOT reset any processable-state (see documentation below for further details on this).
- **void resetParser()**
This method resets all processable-states to true and sets the pointer of the parseObj list to its start.
- **void parseCommandLine(int argc, char** argv)** Parses the command line and adds valid options (that is those with a valid name and a valid amount of values) as well as invalid options to the parser, marking them accordingly.
argc is the number of arguments, argv the char* array;
- **void parseFile(string fn, parseObj* po)**
Parses a file (for further details on this, see below) and adds options to the parser in the same manner as parseCommandLine. fn is the filename, po a pointer to an already declared and allocated parseObj object.

- **void setParamChar(string)**
Default is the minus char, '-'. This methods redefines pre, the string used to indicate a new argument/ option within a command line or file. It MUST be de- fined: there needs to be at least one character to identifie arguments as such.
- **void setFileHeaderLine(string)**
Defines header, the headerline for option files. If this is defined, the first line of an option must must coincide exactly with the given string. Otherwise, the parser will not parse the file. Usage: As option files are textfiles, the first line might be some identifier the be checked. This can be left empty or unset (default), resulting in no headerline.
- **void setCommentTag(string)**
Used for option files, sets ctag. the given string will indicate comments. Everything following this string in a line will be ignored by the parser. Default value is '#'; if left empty, comments are not allowed.
- **void setVerbose(bool)**
If turned on, error messages indicating invalid options or invalid number of values for a given option will NOT be displayed. Otherwise (default) they will be written to the standard output.
- **bool getVerbose()**
Returns the verbose status.
- **int size()**
Returns the number of options given altogether by command line and file.
- **void clearParseList()**
Clears the parse list, resulting in an empty option list.

protected methods and variables:

- **void addCreatedParseObject(parseObj*)**
Used internally by parseCommandLine and parseFile.Adds a parseObj (an object representing an option) to the parse list.
- **tParamObjList *paramList**
List of all possible and allowed options/ parameters.
- **tParseObjList *parseList**
List of valid options after parsing command line and file(s).
- **string pre**
String indicating options in command line and option files.
- **string header**
Headerline for option files.
- **string ctag**
Tag for options.
- **string fileError**
Leading string when displaying errors.

- **bool verbose**
Boolean to indicate whether to display errors and warnings or not.
- **tParseObjListIt searchIt**
Iterator for search operations within the parse list.
- **tParseObjListIt travIt**
Traversal iterator; can be used and changed freely.

Class *parseObj*

public methods:

- **parseObj(paramObj*)**
The constructor takes a paramObj object as the only parameter. In most cases, the programmer itself does not need to invoke the constructor.
- **void init(paramObj*)**
Called by the constructor.
- **optList* getValues()**
Returns values, the list of the values for the parameter given by the command line and option file(s).
- **bool valid()**
Returns isValid, an indicator for being a valid option with a valid amount of values.
- **bool processable()**
Returns whether an option has already been process AND is valid. Whatever the results, setUsed will set set true afterwards, so further calls will return false. The idea of this is to flag used options and thus not apply them again by accident. See below for further details and an example.
- **int size()**
Returns the number of values for an option.
- **string getValue(int i)**
Returns Value number i, "" if n is not a valid value number.
- **void addValue(string)**
Adds a value to the list of values of an option.
- **int isOption(string s1, string s2="")**
Tests whether the object/ option is of given type. Returns 1 if both s1 and s2 match the option or if s2="" and s1 matches, 0 if none of them matches and s1 xor s2 matches.
- **void show()**
Prints the current option/ argument. An internal counter is increased to show its number, too.
- **void setValid()**
Sets isValid to true and thus makes an option valid. Should i.g. not be called by the programmer, but by a paramParser obj.
- **void setInvalid()**
Sets isValid to false. See setValid().
- **int getMinNumberOfValues()**
Returns data->min and thus the minimal number of values (specified when having added it) this option needs.
- **int getMaxNumberOfValues()**
Returns data->min and thus the minimal number of values (specified when having added it) this option needs.

- **string getParamName1()**
Returns data->par1, the first name of the option specified when having added it.
- **string getParamName2()**
Returns data->par1, the second name of the option specified when having added it.
- **int instances()**
Returns data->inst, the number of instances of this option given by command line and option file(s).
- [**void wasProcessable()**
Returns wasPrable; This function is of use for debugging only...]

protected methods and variables:

- **paramObj *data**
Pointer to main paramObj data.
- **void setValidValues(bool)**
Sets isValid and indicates if the amount of values for an option is valid.
- **void setProcessable(bool)**
(Re)sets wasUsed to the given value. Can be used to reset the parser. See below for more details and an example.
- **optList values**
Values of the option.
- **bool isValid**
See above.
- **bool wasUsed**
Boolean variable that indicates whether an option already was used. Is set to true by the processable-method.
- [**bool wasPrable**
For debugging purposes. Keeps track of the processable state one step back compared to the result of the function processable.]

3 A small example

A small example will clarify some things mentioned in the class overview. The source code for this should be part of the package you downloaded; it is a single file called *adder.cpp*.

Our aim is to develop a small command line tool with three options:

- **option 1:** `-help`
Prints out help instructions.
- **option 2:** `-f filename` *or* `-file filename`
Reads arguments from a file.
- **option 3:** `-a value1 [value2]`
Value2 is optional. Adds value1 to Value2 (=0 if not given) and prints the result.

The help should be printed only if there are no other options given. With this simple example, all necessary ideas are covered.

3.1 Initialization

Let us look at the first part of code to realize this:

```
#include "paramObj.h"
#include "parseObj.h"
#include "paramParser.h"
#include <string>
#include <cstdlib>          // for atof

void printHelp(const char* caller="") {
    string usg="\nUsage: " + (string)caller + " [options]\n";
    printf(usg.c_str());
    printf("Options:\n");
    printf("    --help          print this help and quit\n");
    printf("    -f --file filename read options from file 'filename'\n");
    printf("    -a --add v1 [v2]   calculate v1+v2 and print the result\n");
    printf("                      if only v1 is used, v2 shall be zero\n");
}

void initParser(paramParser* pp) {
    // remark: pp must be 'new'ed already!
    if (pp == NULL) return;
    pp->setParamChar("-"); // set character for options
    pp->setFileHeaderLine("** MiniAdder **"); // headerline for files
    pp->addParam("--help"); // add 'help'-option: zero values
    pp->addParam("-f", "--file", 1); // add 'file'-option: 1 value
    pp->addParam("-a", 1, 2); // add 'add'-option: 1 or 2 values
}
```

There is not much to say here: The method *printHelp* will be called when the help option is given. *initParser*, as the name already states, initialies the parser by adding, one by one, all allowed arguments with their respective name(s) and number of parameters/ values. If option files are used (section 3.2), the first line of those needs to be exactly like the one set by *pp->setFileHandle*. Finally, all options have to start with the '-' char, wich is set the defaultone, but included here for the sake of clarity by using *pp->setParamChar(...)*.

3.2 Parsing

The second code snippet will explain how to implement the parsing step:

```
int action(int argc, char** argv, paramParser* pp) {
    parseObj* pa;

    pp->parseCommandLine(argc, argv);

    pa = pp->getOption();

    // no options? print help!
    if (pp->size() < 1) {
        printHelp(argv[0]);
        return 0;
    }
    if (pp->size() == 1) {
        if ( (pa->isOption("--help") > 0) && pa->processable() ) {
            printHelp(argv[0]);
            return 0;
        }
    }

    // traverse options
    while (pa != NULL) {
        if ( (pa->isOption("-f", "--file") > 0) && pa->processable() ) {
            try{
                pp->parseFile(pa->getValue(0), pa);
            }
            catch(char* p) {
                printf("Error in option file %s!\n", p);
            }
        }
        if ( (pa->isOption("-a") > 0) && pa->processable() ) {
            float v1, v2;
            int nop = pa->size();
            switch (nop) {
                case 1: v2 = 0; v1 = atof(pa->getValue(0).c_str()); break;
                case 2: v2 = atof(pa->getValue(1).c_str());
                        v1 = atof(pa->ngetValue(0).c_str()); break;
                default: v2 = 0; v1 = 0;
            }
            printf("%.3f + %.3f = %.3f\n", v1, v2, v1+v2);
        }
        pa = pp->getNextOption();
    }
    return 0;
}

int main(int argc, char** argv) {
    paramParser* p = new paramParser();
    initParser(p);
    action(argc, argv, p);
    delete p;

    return 0;
}
```

After parsing the command line via *pp->parseCommandline(argc, argv)*, the first thing to do is to create a *parseObj* pointer that interacts with the parser itself. This will hold the main

information of the argument line and file options like option name(s), values and validity. `pa->getOption()` will return the very first argument available; the arguments themselves will be in the same order as given by the caller, including argument line **and** file options.

The first two *if*-cases cover the calls of printing the help. As you can see, `argv[0]` is used to display the filename itself. The traversal itself handles the remaining two options. Principally, arbitrary many files (as many as the argument line length allows) can be included and files can include further files, too (however, be sure not to cast yourself into infinite cycles). We will focus on the add option here and come back to an example file afterwards.

`pa->isOption("-a")` checks, whether the current option/ argument is of type "-a", which represents our summation operation. In case of success, a value greater than zero is returned. The objective of the command `pa->processable()` is to check, whether this option has already been processed. Only the current option is affected by this, thus if there are several options of the same kind, the remaining ones are not influenced at all.

If this is the first time of such a check for the current option, the result will be *true* and internally the *used*-flag will be set to false, rendering any further calls of `pa->processable()` of **this** option to result in *false*. The idea behind this is, to allow arbitrary many runs; the processable-state of all options can be reset via `pp->resetParser()`. In future versions, options can be made processable individually, too. The rest of the code within this *add*-operation is just simple conversion, addition and printing of the result.

3.3 An argument file example

Here comes a small file serving as an example for the adder, called "adderExamplefile.txt":

```
** MiniAdder **
-a
-a -3 4 -a 5 6
-a -1 -a -6
```

Now, execute *adder* like this

```
>./adder -a 1 1 -f adderExamplefile.txt
```

to get the following output:

```
1.000000 + 1.000000 = 2.000000
Invalid number of values for parameter -a. Must be between 1 and 2, not 0
0.000000 + 0.000000 = 0.000000
-3.000000 + 4.000000 = 1.000000
5.000000 + 6.000000 = 11.000000
-1.000000 + 0.000000 = -1.000000
-6.000000 + 0.000000 = -6.000000
```

Version 1.0, November 2008

This implementation can be copied, modified and used for free within non-commercial projects. Usage in commercial or shareware projects must be permitted by the author!

Contact information available at **www.gilgamash.com**.