The C^{++} `JParser` class

M. de Jong

July 15, 2022

Abstract

An auxiliary class has been developed to facilitate the parsing of command line arguments. Command line options can be given by the user when a program is launched. This class has similar functionality as the `JProperties` class.

1 Introduction

Command line options are often used to define the running conditions of a program. Combined with a shell script, this allows the user to run the same program with different conditions consecutively. To set a parameter to its supposed value, it is common practice to associate a single character (the 'option') to each parameter. An auxiliary class - `JParser` - has been developed to facilitate the mapping of options to different data types. It is derived from the STL `map` class. The first template type of this `map` is by default type `char`, but could be any data type. This corresponds then to the option specifier of the input data. The second template type is a custom made class which can hold a pointer to any data type. Various methods are defined for this class via a general purpose interface. The list of methods includes the read and write methods for the corresponding parameter object. The only requirement of the data types is that the standard C^{++} I/O redirect operators are defined, namely:

```
std::istream& operator>>(std::istream&, T&)
std::ostream& operator<<(std::ostream&, const T&)
```

where T refers to the type of the parameter object.

The internal `map` can be built using the member method `insert()` or the associative array feature. The macro `make_field()` can be used to add an entry to the `map` with additional information. Default values can be assigned when using the `make_field()` macro as well as a list of possible values can be specified. The parsing of `bool` types is specialised; their default values are always `false` and when the corresponding option is parsed, the value is set to `true` without actually reading any data. Several boolean options can thus consecutively be parsed as no data are read. In case multiple values are provided for the same option, e.g. `-i 1 -i 2`, the last value normally prevails. For all STL container classes, multiple values are differently handled. For example, if the data type is the STL `vector<`T`>`, then each time the corresponding option is encountered an object of type T is read and added to the container using the `push_back()` member method. Multiple values can also be parsed in one go when quoted as in `-i "1 2 3"`.

The following static parameters are used to define the behavior of the `JParser` class. The default values are given in the last column.

type	name	value
char	START_OF_OPTION	'-'
char	HELP_OPTION	'h'
char	REVISION_OPTION	'v'
char	END_OF_OPTIONS	'-'
char	PRINT_OPTION	'!'
char	PID_OPTION	'P'

The parameter START_OF_OPTION is used for the recognition of the start of an option; the parameter HELP_OPTION for printing a help message; and the parameter REVISION_OPTION for printing the revision. The parameter END_OF_OPTIONS marks the end of the option list. The parameter PRINT_OPTION is reserved for printing additional/optional information and the parameter PID_OPTION for printing the process identifier (PID) to a file. The options "-h" and "-h!" will thus print short and long help information, respectively and the option "-v" will print the revision data. All options following "--" are ignored. The option "--!" will print the actual values of all options. This is useful in e.g. batch processing to report the running conditions of an application in the log file. The return value of the function operator (see below) will actually contain the remaining options. The option "--P <file name>" will write the PID to a file with the specified name. This is useful to externally control the running of the application. The constructor's argument debug can be used to set the level of printing of debug, warning and error messages.

The general syntax of a command line should look as follows.

```
<application> -<option specifier> "<all data to be read for corresponding object>"
```

In this, the default start of an option is used. The option specifier corresponds to one of the keys in the map. The data to be read should only be quoted when it contains white spaces and/or newlines. The complete character array is used to construct an std::istringstream object. The parameter object pointed to by the custom class is then read from this std::istream. In case no data are read (e.g. boolean data type), multiple option specifiers can consecutively be provided.

An exception is thrown when an error occurs. The list of possible errors includes an unknown option specifier, a read failure and an invalid value (i.e. the value is not equal to one of the specified possible values). All options should be set, either by default or by an explicit command line option. Otherwise, an error will be thrown.

2 Examples

2.1 A very simple example.

```
#include "Jeep/JParser.hh"

int main(int argc, char**argv)
{
  using namespace std;

  int    aap;
  string noot;

  JParser<> z;
```

```
  try {

    z['a'] = aap;
    z['n'] = noot;

    z(argc, argv);
  }
  catch(const exception &error) {
    cerr << error.what() << endl;
    return 2;
  }

  cout << "aap " << aap << endl;
  cout << "noot " << noot << endl;
}
```

Executing the program with the following command line arguments will yield:

```
example1 -h
usage: example1
 -h  "help"
 -h! "help with print of default and possible values"
 -v  "print revision"
 --  "end of options; remainder will be discarded"
 --! "end of options with print of actual values"
 -a <arg>
 -n <arg>

example1
option: -a <arg> has no value

example1 -a 123 -n "hello world"
aap 123
noot hello world
```

Where `example1` refers to the file name of the main program. The option `-h` causes the main program to end (exit code 1). A missing value will also cause the main program to end (exit code 2).

2.2 Example using default and possible values.

```
#include <vector>

#include "Jeep/JParser.hh"

int main(int argc, char**argv)
{
  using namespace std;

  vector<int> values;

  for (int i = 0; i != 5; ++i) {
    values.push_back(i);
  }
```

```
  int    aap;
  int    noot;
  string mies;

  JParser<> z;

  try {

    z['a'] = make_field(aap)  = 1, 2, 3, 4;
    z['n'] = make_field(noot) = values;
    z['m'] = make_field(mies, "this is extra information") = "hello world";

    z(argc, argv);
  }
  catch(const exception &error) {
    cerr << error.what() << endl;
    return 2;
  }

  cout << z;
}
```

Here, the macro `make_field()` is used to pass the variable to the parser. The parser then takes the first value following the = operator as default. A comma separated list of values is taking as possible values, all other values are then considered invalid. The macro `make_field()` allows for a second, optional argument, to provide additional information for each parameter. This information will be printed following option "`-h!`". Executing the program with the following command line arguments will yield:

```
example2 -h!
usage: example2
 -h  "help"
 -h! "help with print of default and possible values"
 -v  "print revision"
 --  "end of options; remainder will be discarded"
 --! "end of options with print of actual values"
 -a <aap> = 1 [1, 2, 3, 4]
 -m <mies> = hello world "this is extra information"
 -n <noot> = 0 [0, 1, 2, 3, 4]

example2
aap=1
mies=hello world
noot=0

example2 -a 6
option: -a <aap> has illegal value
```

An illegal value will cause the main program to end (exit code 2).

2.3 Example using a different data type for the options.

The following example shows the usage of a different data type for the options. In this case, the `string` class is used instead of the default type `char`.

```
#include "Jeep/JParser.hh"

int main(int argc, char**argv)
{
  using namespace std;

  int    aap;
  string noot;

  JParser<string> z;

  try {

    z["aap"]  = make_field(aap)  = 1, 2, 3, 4;
    z["noot"] = make_field(noot) = "hello world";

    z(argc, argv);
  }
  catch(const exception &error) {
    cerr << error.what() << endl;
    return 2;
  }

  cout << z;
}
```

Executing the program with the following command line arguments will yield:

```
example3 -aap 4 -noot "it's me"
aap=4
noot=it's me
```

2.4 Example using boolean and container data types.

The following example shows the usage of `bool` and `vector` data types.

```
#include "Jeep/JParser.hh"

int main(int argc, char**argv)
{
  using namespace std;

  bool aap, noot, mies;
  vector<string> zus;

  JParser<> z;

  try {

    z['a'] = make_field(aap);
    z['b'] = make_field(noot);
    z['c'] = make_field(mies);
    z['d'] = make_field(zus)   = JPARSER::initialised();
```

```
    z(argc, argv);
  }
  catch(const exception &error) {
    cerr << error.what() << endl;
    return 2;
  }

  cout << "aap: " << aap << endl;
  cout << "noot: " << noot << endl;
  cout << "mies: " << mies << endl;
  cout << "zus: ";
  copy(zus.begin(), zus.end(), ostream_iterator<string>(cout," "));
  cout << endl;
}
```

Executing the program with the following command line arguments will yield:

```
example4
aap: 0
noot: 0
mies: 0
zus:

example4 -abc -d hello -d "it's" -d me -d again
aap: 1
noot: 1
mies: 1
zus: hello it's me again
```

2.5 Example using auxiliary data types and return data.

The following example shows the usage of `JPARSER::JCounter` data type.

```
#include "Jeep/JParser.hh"

int main(int argc, char**argv)
{
  using namespace std;

  JPARSER::JCounter counter;

  JArgs     args;
  JParser<> z;

  try {

    z['c'] = make_field(counter);

    args = z(argc, argv);
  }
  catch(const exception &error) {
    cerr << error.what() << endl;
    return 2;
  }
```

```
  cout << "counter: " << counter << endl;
  cout << args << endl;
}
```

Executing the program with the following command line arguments will yield:

```
example5
counter: 0


example5 -ccc
counter: 3


example5 -c -c -c
counter: 3

example5 -- a b c
counter: 0
example5 a b c
```