The C^{++} JProperties class

Maarten de Jong

2023-01-21 16:35:51

Abstract

An auxiliary class has been developed to facilitate the mapping of key words to C^{++} objects. It allows the user to parse ASCII data in a way that for every key word, the corresponding object is read. It has similar functionality as the $Java$ Properties class.

1 Introduction

Many programs require some parameters to be set in order to operate correctly. When many parameters are involved, the input is usually read from a file (so called data cards). To set a parameter to its supposed value, it is common practice to associate a key word to each parameter. The input data then consist of pairs of key words and parameter values. It is not possible to store references to different data types in a single container using one of the Standard Template Library (STL) container classes (e.g. map). Hence, the parsing of input data is often cumbersome. It is usually done using multiple if statements. An auxiliary class - JProperties - has been developed to facilitate the mapping of key words to different data types. It is derived from the STL map class. The first template type of this map is the standard C^{++} string class. This corresponds then to the key word of the input data. The second template type is a custom made class which can hold a pointer to any data type as well as pointers to read and write methods. The only requirement of these 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 gmake_property() can also be used to add an entry to the map. It takes the parameter name as the default key word. When the same key word appears on the input, the last value is taken. 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 key word is encountered an object of type T is read and added to the container using the push_back() member method. For the STL map<Key,$Elem$> class, both the Key and $Elem$ objects are read and added to the map using the member method insert(). A warning is printed each time an unknown key is encountered.
The constructor's arguments parameters and debug can be used to alter the behaviour of the JProperties class and to control the printing of debug, warning and error messages, respectively. The class JEquationParameters is used to store the parameters. The list of public member methods includes:

method	meaning
`getSeparator()`	get list of characters between key and value
`setSeparator(..)`	set list of characters between key and value
`getEndOfLine()`	get list of end of line characters
`setEndOfLine(..)`	set list of end of line characters
`getDivision()`	get list of characters between consecutive keys
`setDivision(..)`	set list of characters between consecutive keys
`getSkipLine()`	get list of skip line characters
`setSkipLine(..)`	set list of skip line characters
`getLeftBracket()`	get left bracket
`getRightBracket()`	get right bracket
`setBrackets(..)`	set left and right bracket

2 Examples

A very simple example.

```
#include <iostream>
#include <string>
#include "Jeep/JProperties.hh"

int main()
{
  using namespace std;

  string buffer =
    "noot % 3.1415927 \n"
    "mies % abcdefg   \n"
    "aap  % -12345    \n";

  JProperties zap;

  int    aap;
  double noot;
  string mies;

  zap["aap"]  = aap;
  zap["noot"] = noot;
  zap["mies"] = mies;

  if (zap.read(buffer))
    cout << "read success" << endl;
  else
    cerr << "read failure" << endl;

  for (string::iterator i = buffer.begin(); i != buffer.end(); ++i) {
    if (*i == '%') {
      *i = '=';
    }
  }
```

```
  if (zap.read(buffer))
    cout << "read success" << endl;
  else
    cerr << "read failure" << endl;

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

The output of this program looks like this:

```
examplea
read failure
read success
aap:  -12345
noot: 3.14159
mies: abcdefg
```

Where `examplea` refers to the file name of the main program. Note in this example the default equation parameters. The input can equally be read from an input file. For this, the member method `read(std::istream&)` can directly be used.

The following acts in the same way as the example above.

```
  zap.insert(gmake_property(aap));
  zap.insert(gmake_property(noot));
  zap.insert(gmake_property(mies));
```

Here, the key words correspond implicitly to the parameter names.

The following example reads the event numbers from a standard `km3` output file and prints them.

```
#include <iostream>
#include <iomanip>

#include "Jeep/JProperties.hh"


/**
 * Data structure for start_event tag.
 */
struct JEvent {
  int number;
  int type;

  static int count;      // global event counter
};


int JEvent::count = 0;   // initialisation of global event counter


inline std::istream& operator>>(std::istream& in, JEvent& event)
```

```
{
  JEvent::count += 1;

  return in >> event.number >> event.type;
}


inline std::ostream& operator<<(std::ostream& out, const JEvent& event)
{
  return out << event.number << ' ' << event.type;
}


/**
 * Program to test JProperties class,
 */
int main(int argc, char **argv)
{
  using namespace std;
  using namespace JEEP;

  JEvent event;

  JProperties zap(JEquationParameters(":", "\n", "", ""), 1);

  zap["start_event"] = event;

  zap.read(cin);

  cout << "Number of events " << JEvent::count << endl;
}
```

The typical output of this program looks like this:

```
Number of events 54228
```

In this example, the equation parameters are set to correctly parse a ".evt" file and the debug level is set to disable the printing of warnings as there are many "unknown" keys in the input file.