REG File Parser using the Boost Spirit Parser Framework.

Written by
Ivan Romanenko,
Junior Software Developer of Apriorit Inc.

Introduction

This article describes the sample parser of reg files using Boost Spirit Parser Framework. The main aim of this article is to share experience and that's why here I'll show not so much the solution for the concrete task or the usage of the concrete technology as the way the develoment process goes. In particular we'll discuss why we use the curtain libraries and make one or another solution.

Testimanials

I would like to thank the people who developed the following projects - they made the implementation of this project easier:

- Boost:
  - Spirit Parser Framework
  - Program options
  - Pool library
  - Testing Freamwork
  - Other Utils: Bind, Function, Scope Exit, String Algorithms Library

I want to say personal thank you to Silviu Simen for his article INI file reader using the spirit library.

Background and history of this task

There was a project in which I took part and we needed to test the correct work of parser of Windows hive registry files. These files are stored in the binary representation and the structure of such file is not documented by Microsoft. But by means of research my colleagues managed to clear out this structure, and after that the question of correct parser work arised.

To perform testing I decided to use the functionality of export of registry in two formats: hive and reg. Thus I could obtain two different files for the same registry key and after that check the correct work of Windows hive registry file parser.

The structure of registry file - I'll give an example below - is very similar to the structure of the ini file, so you can use standard Windows functions for reading values in this file. But the problem is that functions work very slow for the big files, and that is why this parser was developed - parser of reg files where I used Boost Spirit Parser Framework. The reasons why standard Windows functions are slow will be considered below in this article.

What is reg file?

Let's consider the general view of reg file structure first, and some special complicated cases will be considered as neccessary.

I've taken the following material from here http://en.wikipedia.org/wiki/Windows_Registry.

.REG files (also known as Registration entries) are text-based human-readable files for storing portions of the registry. On Windows 2000 and later NT-based operating systems, they contain the string Windows Registry Editor Version 5.00 at the beginning and are Unicode-based. On Windows 9x and NT 4.0 systems, they contain the string REGEDIT4 and are ANSI-based. Windows 9x format .REG files are compatible with Windows 2000 and later NT-based systems. The Registry Editor on Windows on these systems also supports exporting .REG files in Windows 9x/NT format. Data is stored in .REG files in the following syntax:

```
[<Hive Name>\<Key Name>\<Subkey Name>]
"Value Name"=<Value type>:<Value data>
```

Example1 (different types):

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft]
"Value A"="<String value data>"
"Value B"=hex:<Binary data>
"Value C"=dword:<DWORD value integer>
"Value D"=hex(7):<Multi-string value data>
"Value E"=hex(2):<Expandable string value data>
```

Example2 (real):

```
[HKEY_CURRENT_USER\Key]
"Value string"="B"
"Value dword"=dword:00000001
"Value hard"=hex(1000800c):53,00,65,00,72,00,76,00,69,00,63,00,65,00,53,00,\
  74,00,61,00,72,00,74,00,54,00,79,00,70,00,65,00,00,00,4d,00,61,00,78,00,44
```

Making a little digression I want to stress that the number in the line " hex(1000800c) " is the type identifier and it can be any. It's often used as the data in the security branch [HKEY_LOCAL_MACHINE\SAM].

And now let's try to extend the information about possible content of reg file. Here I represent some facts obtained during our research process:

```
1) Key Name may consist of alphabetical symbols and " ,  \ ,  [ , ].
2) Number of values of one key can be from 0 to infinite
3) Value Name can be:
   - symbol '@' - it means default
   - "text"     - any symbols can be in this "text"  even these ones: \n , "
, \ , [ , ]
                  but it always ends with "\n synbols in sequence
4) Value data can be:
 - "text" - any symbols can be in this "text" but it always ends with " and
\n in sequence
 - binary:
   - dword:XX
   - hex:XX
   - hex(N):XX
   Comments:
     XX can be the pairs of number symbols separated by comas and
     it can end with '\' symbol that means that data continue in the next
line
   Example:
```

```
   dword:72,...,00,\
   00,..,20
```

Two approaches and their comparison

As it was mentioned the structures of reg file and ini file are quite the same so I started to search for the methods of working with ini files. I found the standard Windows functions.

Using Windows API functions

Windows gives a lot of funcions to work with ini files, we are interested in two of them for our task:

```
GetPrivateProfileSectionNames
Retrieves the names of all sections in an initialization file.

GetPrivateProfileSection
Retrieves all the keys and values for the specified section of an
initialization file.
```

So we should call GetPrivateProfileSectionNames one time to obtain the list of the keys and then call GetPrivateProfileSection to obtain values inside the key.

The problem is that if the file is quite big, i.e. there are a lot of keys in it, then we should call GetPrivateProfileSection for many-many times to read from the file. Here is some row test data: file size: 30 MB, file includes about 30 000 keys, the parsing of this file takes about 20 minutes. And I should say that reg files are often bigger than 100 MB.

So, the problem:

Unjustified number of readings from the file.

Solution:

It's neccessary to load the file to the memory at one time or in some parts and then parse its content by means of the own tools.

Using custom parser

As far as we should parse reg file content by our own tools it's good idea to use already developed work. Thus I came accross the article INI file reader using the spirit library. Using the example of ini file parser I developed my reg file parser.

Why boost::spirit ?

The Spirit Parser Framework is an object oriented recursive descent parser generator framework implemented using template metaprogramming techniques. Expression templates allow users to approximate the syntax of Extended Backus Naur Form (EBNF) completely in C++.

I was also attracted by:

1. There are no intermediate conversions to some code needed and also no external applications except for compiler.
2. You should include only two header files and no libraries to use Spirit Parser Framework.

Useful Links :

- Boost Spirit Parser Framework
- Wikipedia about Spirit Parser Framework

Fleeting glance on boost::spirit

Main idea of using boost::spirit is in using rules. Usually several basic rules are defined, and then other rules are defined by means of overridden operators as the combinations of basic rules. Next example shows the creation of rules using "AND" and "NOT" operators:

```
RuleType simpleRule = ~ch_t('A') & ~ch_t('B');
```

This rule works for any symbol except for 'A' and 'B'. Below in this article each rule will be described in details but now I want to give short information about possible operators - to let you imagine what are possible operations with the rules.

```
Set operators:
 a | b    Union             Match a or b. Also referred to as alternative
 a & b    Intersection      Match a and b
 a - b    Difference        Match a but not b. If both match and b's matched
text
                            is shorter than a's matched text, a successful
match is made
 a ^ b    XOR               Match a or b, but not both

Sequencing Operators:
 a >> b    Sequence          Match a and b in sequence
 a && b    Sequential-and    Sequential-and. Same as above, match a and b in
sequence
 a || b    Sequential-or     Match a or b in sequence

Optional and Loops:
 *a        -    Match a zero (0) or more times
 +a        -    Match a one (1) or more times
 !a        -    Match a zero (0) or one (1) time
 a % b     -    Match a list of one or more repetitions of a separated by
occurrences of b.
                This is the same as a >> *(b >> a). Note that a must not also
match b

Single character parsers:
 anychar_p  Matches any single character (including the null terminator:
'\0')
 alnum_p    Matches alpha-numeric characters
 alpha_p    Matches alphabetic characters
 blank_p    Matches spaces or tabs
 cntrl_p    Matches control characters
 digit_p    Matches numeric digits
 graph_p    Matches non-space printing characters
 lower_p    Matches lower case letters
 print_p    Matches printable characters
 punct_p    Matches punctuation symbols
 space_p    Matches spaces, tabs, returns, and newlines
 upper_p    Matches upper case letters
 xdigit_p   Matches hexadecimal digits
```

```
Other comments:
 negation ~
 Example: ~ch_t('x') - matches any character except 'x'
```

Introduction to the parser development

Further in this article I'll try to keep the high level of clearness so I'm sorry for advance if you think that my descriptions are too detailed. Description starts from the heart of the parser - its algorithm. Algorithm is described in parts, then wrapper for this algorithm is described and then others auxiliary classes. At the end we'll consider concrete usage example and test.

Algorithm description

Algorithm is represented by the function and interface:

```
template<class charT>
struct IResultProcessor
{
    virtual ~IResultProcessor(){}

    virtual void OnKeyFound(const charT* begin, const charT* end)=0;
    virtual void OnValueNameFound(const charT* begin, const charT* end)=0;
    virtual void OnValueDataFound(const charT* begin, const charT* end)=0;
};

template<class charT>
inline bool ParseRegFileImpl(const charT* buffer,
                             IResultProcessor<charT>* resultProc);
```

It follows from the function names that the algorithm will call OnKeyFound function for each key name found, OnValueNameFound for each value name found and OnValueDataFound for value content.

Observant reader can ask a question: "Why the processing of Value is separated into two functions OnValueNameFound and OnValueDataFound, after all it it one single entity?". The answer is simple: "The implementation of this processing inside the algorithm would be hard and so algorithm entrust calling side with the processing of these two parts".

Preparatory work

At this stage we will prepare the environment for the convenient work. In particular we should use name space, shorten the line for rule creation and define frequently used rules.

```
    using namespace boost::spirit;
    typedef rule<scanner<const charT*> > RuleType;
    typedef chlit<charT> ch_t;  // Single character
    typedef chset<charT> chs_t; // Character set
    typedef IResultProcessor<charT> ResProcT;

    chs_t anychar_t(anychar_p); // Pattern anychar to character set
    chs_t eol_CR('\r'); // End of line CR
    chs_t eol_LF('\n'); // End of line LF
    chs_t eol_t(eol_CR);// CR or LF end of line
    eol_t |= eol_LF;
```

Auxiliary rules

```
    // Matches spaces or tabs
    RuleType blanks   = * blank_p;

    // Symbols ']'and '[' - separate Key Name
    RuleType not_name_separator = ~ch_t(']') & ~ch_t('[');

    // empty data
    RuleType empty_data = blanks >> (eol_t | ch_t('\0'));

    // Data that we don't interested
    // @ - it's default value name , " - requires additional processing
    RuleType other_data = *(anychar_t & not_name_separator & ~ch_t('@')&
~ch_t('"'));
```

The first rule is to omit blanks and tabs. Asterisk means that the rule can work several times in a row or don't work at all.

The second rule is to make sure that the current symbol is not the name separator.

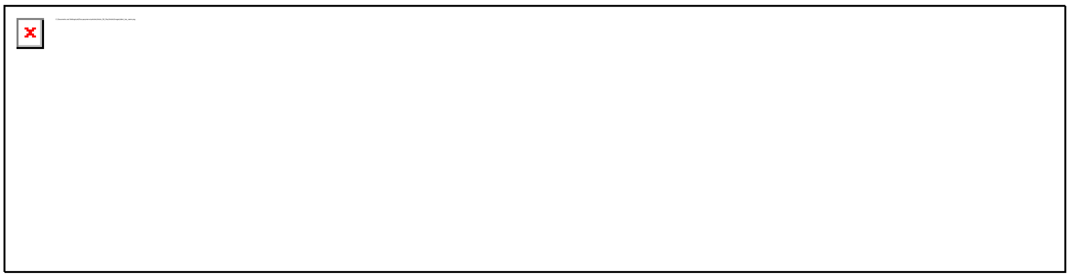
The third rule is to omit empty data. Operator >> means that the rule at the left side of the operator and the rule at the right side should be performed in sequence, one after another.

The forth rule is assigned to the free data that are not the parts of separators. Any symbol matches this rule except for name separators, @ - identifier of value by default and symbol " that signals about the end of the line. This rule can work either several times or no one time.

Rule for the key name

```
    RuleType ident_kname_continue = ch_t(']') >> ~eol_t;
    RuleType ident_key_name = *(anychar_t & ~ch_t(']')) ||
ident_kname_continue >> ident_key_name;
```

This rule can be illustrated on the scheme:



Rule ident_kname_continue is an auxiliary for the ident_key_name. First of all ident_key_name is the recursive rule and after the process meets symbol] it tries to define if it is the end of the name or the part of the name. If symbol] is the part of the name i.e there is no symbol of line end after it, then this rule starts itself again and continue parsing.

Let's consider the following case for example:

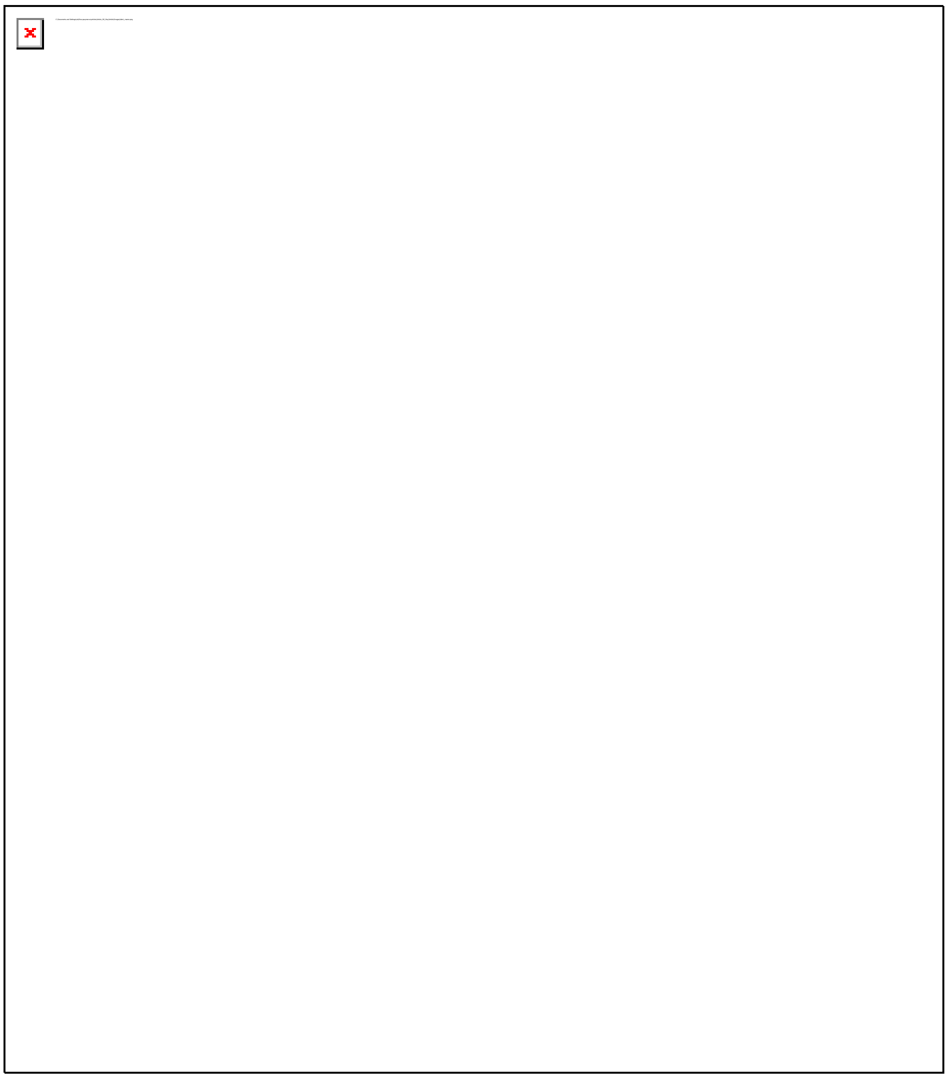
```
    [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\_HARD_NAME[123]_ABCD]
```

In this examle symbol] is the part if the name and that's why it's neccessary to check if there is the symbol of line end after it. You can meet such names in the registry.

Rule for the Value name

```
    // if "= symbols sequence is found - value name ends
    RuleType ident_vname_continue = +ch_t('"') >> ~ch_t('=');
    // If '"' symbol is found - check if value name ends, if it doesn't
parsing continues.
    RuleType ident_vname_sz_skip = ch_t('\\') >> ch_t('"');
    RuleType ident_vname_sz_body = *(anychar_t & ~ch_t('"') ||
ident_vname_sz_skip );
    RuleType ident_vname_sz = ident_vname_sz_body >> *(ident_vname_continue
>> ident_vname_sz_body);
    // Rule for default value name
    RuleType ident_vname_sz_impl = ch_t('"') >> ident_vname_sz >> +ch_t('"');
    RuleType ident_vname_def = ch_t('@');
    // "text" or DEFAULT
    RuleType ident_value_name = ident_vname_def | ident_vname_sz_impl;
```

This rule can be represented on the scheme:



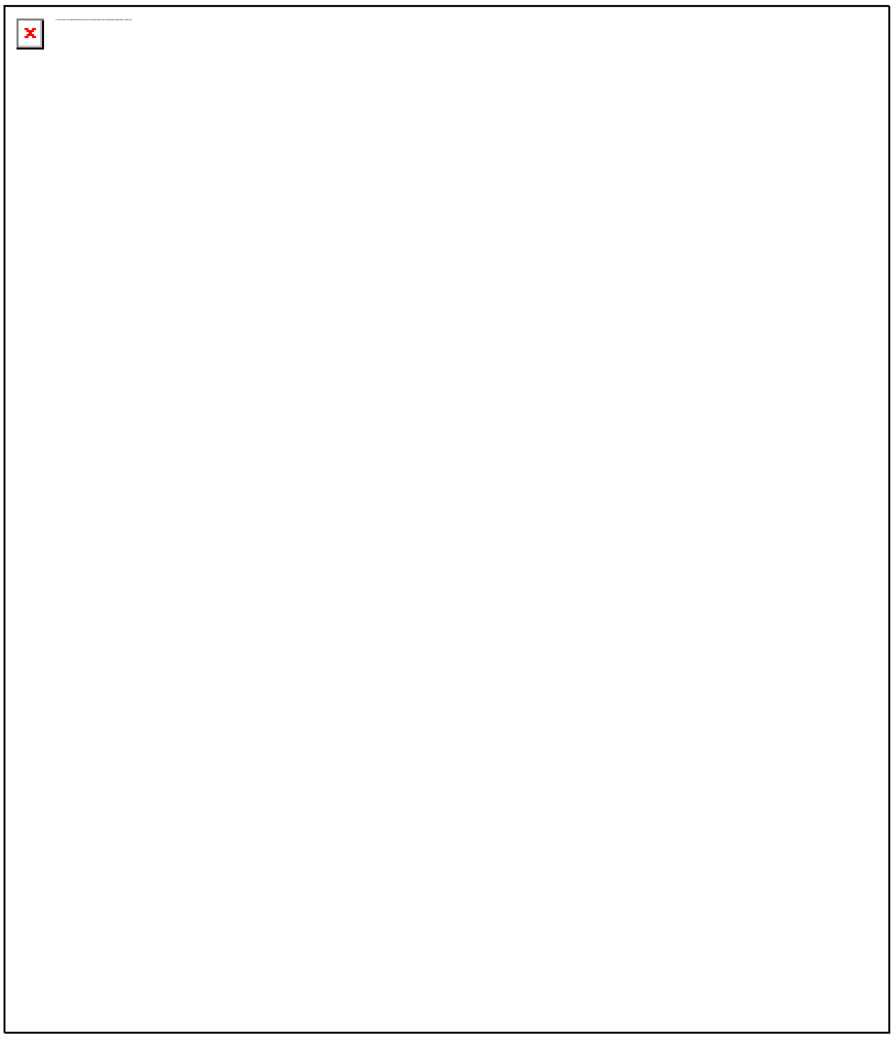
Rule for the value content

```
    // if value content is binary
    // Data can be any character except for '\\' and '\n' in sequence
    RuleType vdata_bin_body =*(anychar_t & ~ch_t('\0') & ~eol_t &
~ch_t('\\'));
    RuleType vdata_bin_continue = ch_t('\\') >> eol_t;
    // if '\\' and '\n' in sequence data continue on the next line
    RuleType vdata_bin = vdata_bin_body >> *(vdata_bin_continue >>
vdata_bin_body);

    // if value content is string
    // String data always ends with '"' and '\n' symbols in sequence
    RuleType vdata_sz_continue = +ch_t('"') >> ~eol_t;
    RuleType vdata_sz_body_impl = *(anychar_t & ~ch_t('"'));
    RuleType vdata_sz_body = vdata_sz_body_impl >> *(vdata_sz_continue >>
vdata_sz_body_impl);
    // String data always starts and ends with '"' symbol
    RuleType ident_vdata_sz = ch_t('"') >> vdata_sz_body >> +ch_t('"');

    // Check value data format
    RuleType ident_vdata = ident_vdata_sz | vdata_bin;
```

This rule can be represented on the scheme in such way:



Rule that describes the line with the key name

```
    // line with key name
    RuleType l_key =
        other_data>> // Can be comments or start title
        ch_t('[') >> // start of the key name
        // Call OnKeyFound function if rule succeeds
        ident_key_name   [bind(&CRegFileParserImpl::OnKeyFound, this, _1,_2)
] >>
        blanks    >> // can be blanks
        ch_t(']') >> // end of the key name
        blanks    >> // can be blanks
        *eol_t;      // one or more end of line symbols
```

This rule requires the execution of several rules in sequence. The new idea here is to use operator []. In this case it means that when the rule is activated functor transferred as the parametr should be called. And the range where this rule worked will be transferred to this functor via two parameters.

Rule that describes the line with the Value

```
    // lines with value name and data
    RuleType l_values =
        other_data>> // Can be comments or start title
        // Call OnValueNameFound function if rule succeed
        ident_value_name   [bind(&CRegFileParserImpl::OnValueNameFound, this,
_1,_2) ] >>
        blanks    >> // can be blanks
        ch_t('=') >> // always separates value name and value data
        blanks    >> // can be blanks
        // Call OnValueDataFound function if rule succeeded
        value_data  [bind(&CRegFileParserImpl::OnValueDataFound, this, _1,_2)
]  >>
        blanks    >> // can be blanks
        *eol_t;      // one or more end of line symbols
```

This rule is much like the previous one. At this moment we can return to the question stated during the interface discussion: "Why the processing of Value is separated into two functions OnValueNameFound and OnValueDataFound, after all it it one single entity?". Now we can give more detailed answer. As it's seen from the code above it's hard to support interface corresponding to the reg file structure by means of boost::spirit - the call of funcion with 4 parameters for the whole value.

Gathering everything together

```
    // Any line can satisfy one of the three rules
    RuleType lines = l_key | l_values | empty_data;

    // Start lexeme_d pars that compares also additional symbols
    // if does just *lines, symbols ' ','\t','\n' are not compared
    RuleType reg_file =  lexeme_d [*lines] ;

    // Execute parse
    return (parse(buffer, reg_file).full);
```

So we finished the algorithm of reg file parsing. The rules for the key name, value is agregated and file parsing starts.

The whole function looks like the following :

```
template<class charT>
inline bool ParseRegFileImpl(const charT* buffer,
                             IResultProcessor<charT>* resultProc)
{
    using namespace boost::spirit;
    typedef rule<scanner<const charT*> > RuleType;
    typedef chlit<charT> ch_t;  // Single character
    typedef chset<charT> chs_t; // Character set
    typedef IResultProcessor<charT> ResProcT;

    chs_t anychar_t(anychar_p); // Pattern to char set
    chs_t eol_CR('\r'); // End of line CR
    chs_t eol_LF('\n'); // End of line LF
    chs_t eol_t(eol_CR);// CR or LF end of line
    eol_t |= eol_LF;

/* ---------------------------------------------------------------------------
----------- */
/* Help rules*/

    // Matches spaces or tabs
    RuleType blanks   = * blank_p;
    // Symbols ']'and '[' - separate Key Name
    RuleType not_name_separator = ~ch_t(']') & ~ch_t('[');
    // empty data
    RuleType empty_data = blanks >> (eol_t | ch_t('\0'));
    // Data in what we aren't interested
    // @ - it's default value name, " - requires additional processing
    RuleType other_data = *(anychar_t & not_name_separator & ~ch_t('@')&
~ch_t('"'));

/* ---------------------------------------------------------------------------
----------- */
/* Rules that describe identifier of key name */

    RuleType ident_kname_continue = ch_t(']') >> ~eol_t;
    RuleType ident_key_name = *(anychar_t & ~ch_t(']')) ||
ident_kname_continue >> ident_key_name;

/* ---------------------------------------------------------------------------
----------- */
/* Rules that describe identifier of value name */

    // Skip \" sequence
    RuleType ident_vname_sz_skip = ch_t('\\') >> ch_t('"');
    RuleType ident_vname_sz_impl = *(anychar_t & ~ch_t('"') ||
ident_vname_sz_skip );
    // Add trailing symbols to match pattern
    RuleType ident_vname_sz = ch_t('"') >> ident_vname_sz_impl >> +ch_t('"');
    // Rule for default value name
    RuleType ident_vname_def = ch_t('@');
    // "text" or DEFAULT
    RuleType ident_value_name = ident_vname_def | ident_vname_sz;

/* ---------------------------------------------------------------------------
----------- */
/* Rules that describe value content */

    // if value content is binary
    // Data can be any character except for '\\' and '\n' in sequence
    RuleType vdata_bin_body =*(anychar_t & ~ch_t('\0') & ~eol_t &
~ch_t('\\'));
    RuleType vdata_bin_continue = ch_t('\\') >> eol_t;
    // if '\\' and '\n' in sequence then data continue on the next line
    RuleType vdata_bin = vdata_bin_body >> *(vdata_bin_continue >> 
vdata_bin_body);

    // if value content is string
    // String data always end with '"' and '\n' symbols in sequence
    RuleType vdata_sz_continue = +ch_t('"') >> ~eol_t;
    RuleType vdata_sz_body_impl = *(anychar_t & ~ch_t('"'));
    RuleType vdata_sz_body = vdata_sz_body_impl >> *(vdata_sz_continue >> 
vdata_sz_body_impl);
    // String data always start and end with '"' symbol
    RuleType ident_vdata_sz = ch_t('"') >> vdata_sz_body >> +ch_t('"');

    // Check value content format
    RuleType ident_vdata = ident_vdata_sz | vdata_bin;

/* ------------------------------------------------------------------------------
----------- */
/* Put all rules together */

    // line with key name
    RuleType l_key = 
        other_data>> // Can be comments or start title
        ch_t('[') >> // starts key name
        // Call OnKeyFound function if rule succeed
        ident_key_name   [bind(&ResProcT::OnKeyFound, resultProc, _1,_2) ] >>
        blanks    >> // can be blanks
        ch_t(']') >> // ends key name
        blanks    >> // can be blanks
        *eol_t;      // one or more end of line symbols

    // lines with value name and data
    RuleType l_values =
        other_data>> // Can be comments or start title
        // Call OnValueNameFound function if rule succeed
        ident_value_name   [bind(&ResProcT::OnValueNameFound, resultProc, 
_1,_2) ] >>
        blanks    >> // can be blanks
        ch_t('=') >> // always separates value name and value data
        blanks    >> // can be blanks
        // Call OnValueDataFound function if rule succeed
        ident_vdata  [bind(&ResProcT::OnValueDataFound, resultProc, _1,_2) ]
>>
        blanks    >> // can be blanks
        *eol_t;      // one or more end of line symbols

    // Any line can satisfy one of three rules
    RuleType lines = l_key | l_values | empty_data;

    // Do lexeme_d pars that compare also additional symbols
    // if do just *lines, symbols ' ','\t','\n' are not compared
    RuleType reg_file =  lexeme_d [*lines] ;

    // Execute parse
    return (parse(buffer, reg_file).full);
}
```

CRegFileParser

It's time to consider wrapper class for this algorithm. This class assumes processing of value name and value content. In other words this class supports the following intrface:

```
template<class charT>
struct IRegFileObserver
{
    virtual ~IRegFileObserver(){}

    virtual void OnKeyFound(const charT* begin, const charT* end)=0;
    virtual void OnValueFound(const charT* nameBegin, const charT* nameEnd,
                              const charT* dataBegin, const charT* 
dataEnd)=0;
};
```

And the class itself :

```
template<class charT>
class CRegFileParser:public IResultProcessor<charT>
{
    typedef std::pair<const charT*,const charT*> BuferRange;
    BuferRange lastVName_;

    IRegFileObserver<charT>* pObserver_;

    bool bLastVNameProcessed_;
public:
    CRegFileParser(IRegFileObserver<charT>* pObserver)
        : pObserver_(pObserver)
        , bLastVNameProcessed_(true)
    {}

    bool Parse(const charT* buffer)
    {
        return ParseRegFileImpl(buffer,this);
    }

private:
    void OnKeyFound(const charT* begin, const charT* end)
    {
        CheckVNameProcessed(true);

        pObserver_->OnKeyFound(begin,end);
    }
    void OnValueNameFound(const charT* begin, const charT* end)
    {
        CheckVNameProcessed(true);

        bLastVNameProcessed_ = false;
        lastVName_.first = begin;
        lastVName_.second = end;
    }
    void OnValueDataFound(const charT* begin, const charT* end)
    {
        CheckVNameProcessed(false);

        bLastVNameProcessed_ = true;
        pObserver_-
>OnValueFound(lastVName_.first,lastVName_.second,begin,end);
    }
    void CheckVNameProcessed(bool needToBeProcessed)
    {
        if(bLastVNameProcessed_ != needToBeProcessed)
            throw std::exception("Value data not found for founded value 
name");
```

```
    }
};
```

Pool using

Before considering of observers created for testing we should describe the usage of boost pool library.

What is Pool?

Pool allocation is a memory allocation scheme that is very fast, but limited in its usage.

Installation

The Boost Pool library is a header file library. That means there is no .lib, .dll, or .so to build; just add the Boost directory to your compiler's include file path, and you should be good to go!

How do I use Pool?

To make it convenient I created the following classes that have the functionality of the standard STL containers but use boost pool library:

```
#include <boost/pool/pool_alloc.hpp>
#include <map>
#include <vector>
#include <list>
#include <string>

template<class _Kty, class _Ty>
class QMap : public std::map< _Kty, _Ty, std::less<_Kty>,
                              boost::fast_pool_allocator< std::pair<_Kty,_Ty>
> >
{};

template<class _Ty>
class QVect : public std::vector< _Ty, boost::pool_allocator<_Ty> >
{};

template<class _Ty>
class QList : public std::list< _Ty, boost::pool_allocator<_Ty> >
{};

template<class _Ty>
class QStringStream : public std::basic_stringstream<
_Ty,std::char_traits<_Ty>,

boost::pool_allocator<_Ty> >
{};

template<class _Ty>
class QString: public std::basic_string<_Ty,std::char_traits<_Ty>,
                                        boost::pool_allocator<_Ty> >
{
    typedef std::basic_string<_Ty,std::char_traits<_Ty>,
                              boost::pool_allocator<_Ty> >
    BaseClass;
public:
    QString(){}
    // construct from [_First, _Last), const pointers
    QString(const _Ty* first,const _Ty* last)
        : BaseClass(first,last)
    {}
};
```

The only disadvantage - as you could notice - is not full support of original constructors. I actually think that it's not the big problem, you can just insert the necessary constructor if it's needed.

Observers

Some observers were created for testing purposes:

1. CRegStatusObserver - assigned to print status of parsing process on the screen in percentage.
2. CRegCountObserver - assigned to print the number of keys and values on the screen.
3. CRegPrintObserver - assigned to print all keys and values on the screen.
4. CRegFullObserver - assigned to store all keys in std::map, where the name of the registry key is key and the vector of all values of this registry key is std::vector values.
5. CRegObserversPool - assigned to make it possible to use several observers simultaneously.

It's important to mention that these observers print information on the screen but they can be used to print to the file or some other stream. For example CRegCountObserver:

```
template<class charT,class streamT>
class CRegCountObserver:public IRegFileObserver<charT>
{
    size_t keysCount_;
    size_t valuesCount_;

    streamT& out_;
public:
    CRegCountObserver(streamT& out)
        : keysCount_(0)
        , valuesCount_(0)
        , out_(out)
    {}
    ~CRegCountObserver()
    {
        out_ << "Keys count: "
             << keysCount_
             << "\t Values count: "
             << valuesCount_
             << "\n";
    }
...
};
```

And some examples of its usage:

```
// 1
CRegCountObserver<char> screenObsr(std::cout);

// 2
```

```
std::stringstream strStream;
CRegPrintObserver<char,std::stringstream> stringObsr(strStream);

// 3
std::fstream fileStream;
CRegPrintObserver<char,std::fstream> fileObsr(fileStream);
```

The usage of other observers is the same.

Auto-tests description

All auto-test are represented in the one project RegFileParserAutoTest.

The test is the console application developed with Boost Testing Framework. 5 complicated cases were chosen to be the test data: one for key name, two for value name and two for value content.

There is one more case with the typical content of reg file. So we have 6 cases and as far as there are two versions of reg file format - ANSI and UNICODE (Regedit4 and Regedit5 correspondingly) each of our cases dublicates for two formats. As the result we have 12 test files.

To control the results of parsing we use comparison of original content and content parsed and saved in the memory using CRegPrintObserver:

```
template<class charT>
class TestRunner
{
protected:
    typedef std::basic_string<charT> stringT;
    typedef std::basic_stringstream<charT> stringstreamT;

    typedef std::basic_fstream<charT> fstreamT;
public:

    ...

    static void
    RunTest(const std::wstring& fileName)
    {
        std::vector<char> buffer;
        ReadFile(fileName,&buffer);

        stringstreamT stream;
        reg_parser::CRegPrintObserver<charT,stringstreamT>
coutObserver(stream);
        reg_parser::CRegFileParser<charT> regParser(&coutObserver);

        // Run parsing
        if( !regParser.Parse( (charT*)&buffer[0] ) )
            throw std::exception("Parsing fail.");

        // Prepare data for comparison
        stringT parsedStr;
        AddRegHeader(&parsedStr);
        parsedStr += stream.str();

        stringT originalStr = stringT((charT*)&buffer[0]);
        // Trim unneeded symbols
        using namespace boost;
        trim_if(originalStr,is_any_of("\n\t\0 "));
        trim_if(parsedStr,is_any_of("\n\t\0 "));

        if( parsedStr != originalStr)
        {
            // Save to file result of parsing
            std::wstring parsedFileName = fileName + L"_parsed";
            fstreamT file(parsedFileName.c_str(),fstreamT::out |
fstreamT::trunc);
            if( file.is_open() == false )
                throw std::exception("Can't create file for result of
parsing.");

            BOOST_SCOPE_EXIT( (&file) )
            {
                file.close();
            }
            BOOST_SCOPE_EXIT_END

            file << parsedStr;

            throw std::exception("Parsed data not equal to original data.");
        }
    }

    ...
};
```

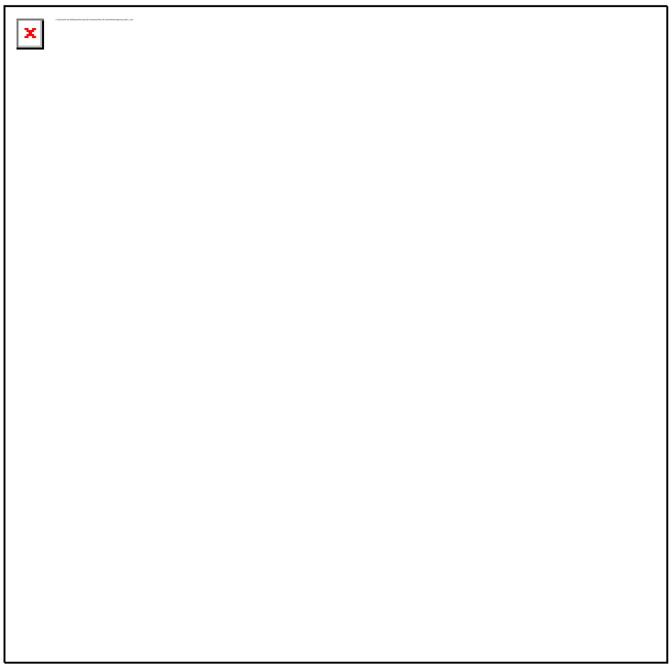
Description for manual testing

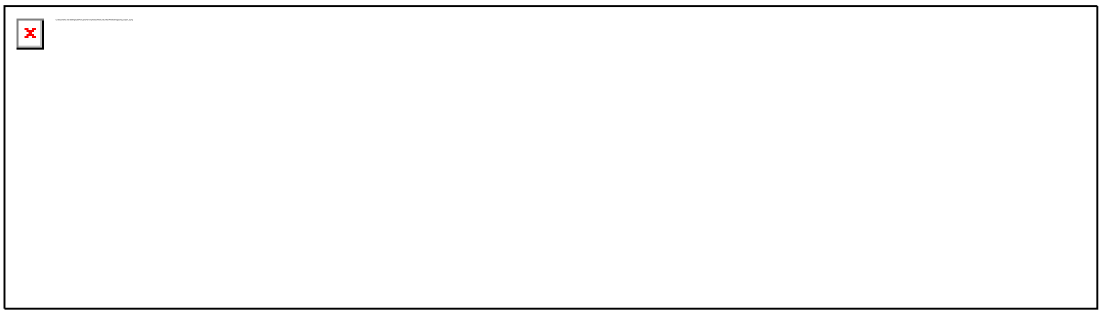
Manual tests are represented in one project RegFileParserTestCmd. Test is console application implemented using Boost Program Options that has such parameters:

```
C:\RegFileParser\bin\Debug>RegFileParserTestCmd.exe --help
Allowed options:
  --help                produce help message
  --reg_file arg        source reg file
  --print               Enable printing to the screen
  --count               Enable counting parsed keys and values
  --status              Enable printing status of parsing
```

It's a pitty that I haven't enough time to add option to save parsed content to the file, and using save option like it is in autotest doesn't seem to be aesthetic for me.

Registry export using Regedit





Conclusion

This article is a special piece of knowledge so may be it's not as systematic as it should be.

It was really interesting to learn boost spirit and I managed to get pleasure of my work with it and some aesthetic satisfacion - not only get my task done. After all this approach discovered to be very effective, for example the calculation of number of keys and values in the registry file of 250 MB now takes a couple of seconds only.

So learn something new and good luck to you in your developement!

Download sources from Apriorit site.