

{cvu}

Volume 26 • Issue 5 • November 2014 • £3

Features

Playing by the Rules

Pete Goodliffe

In the Toolbox: Taming the inbox

Chris Oldwood

Debuggers Are Still for Wimps

Frances Buontempo

Parsing Configuration Files in C++ with Boost

Giuseppe Vacanti

Const and Concurrency

Ralph McArdell

Regulars

C++ Standards Report

Code Critique

Regional Meetings

Book Reviews

Parsing Configuration Files in C++ with Boost

Giuseppe Vacanti describes how to deal with program options, C++ style.

In the (Unix) world of command-line tools I inhabit, parsing configuration files is a common first step most of my tools perform before starting a long and complex computation. Although command-line tools may make you think of command-line switches, I often write simulation or data analysis tools that require up to a few tens of parameters to describe either an experiment or a complex geometrical model: not something you want to type out often on the command line. Storing the parameters in a file, possibly allowing the user to perform a command line override of some of them, is one possible solution.

After thinking about the format of these configuration files, I chose to use the INI format. The latter is loosely defined, and a number of variations on the theme exist, some of them rather complex, involving nested sections and entries extending over multiple lines. I have settled for the classical format consisting of a series of **key=value** lines, possibly grouped into sections, like the following example:

```
energy = 10
mass = 22
[detector]
position_x = 1
position_y = 2
position_z = 3
[detector.material]
atomic_number = 43
```

I am aware that the format has some limitations, especially when it comes to string values that either extend over multiple lines or contain escaped characters, but I do not care for these features.

There are a number of libraries able to read INI files, but most of them require the client to perform all the work of checking whether parameters are present (or misspelled), and to what type they should be converted. For instance:

```
ini_file i_f(file_name);
if(i_f.has_key(section_name, key_name)){
    cout << i_f.key<double>(section_name,
        key_name) << endl;
}
```

Why this is so can be understood by building a minimal INI file parser in C++. In this parser the INI file could be represented by a `map<string, map<string, string>>`, and with a bit of encapsulation one would arrive to the interface in Listing 1.

With such a parser the number of checks and conversions the client must perform in order to extract all the required parameters is large, and prone to error.

Enter the Boost library Program Options [1]. The main purpose of the library is to parse command line options, but it offers the possibility to read the options from an INI file, which is exactly what I want. Additionally, the library will check whether a key has a value that can be converted to the appropriate type, it can deal with duplicate/multiple entries (allowed or not allowed), can be used to provide default values for missing keys, and more. In the following I explain how to use the library for this purpose.

GIUSEPPE VACANTI

Giuseppe Vacanti is a physicist who works at a small high-tech company in the Netherlands, and who likes to solve problems with C++ and Python. He can be reached at giuseppe@vacanti.org



```
class ini_file {
public:
    ini_file(istream & is) { ... }
    bool has_key(const string & section,
        const string & name) const {
        Iter p = m_ini.find(section);
        bool ret_val = false;
        if(p != m_ini.end()){
            section_t & sec = m_ini[section];
            ret_val = sec.find(name) != sec.end();
        }
        return ret_val;
    }

    template<typename T> T key(const string &
        section, const string & name) const {
        assert(has_key(section, name));
        return boost::lexical_cast<T>
            (m_ini[section][name]);
    }

private:
    typedef map<string, string> section_t;
    typedef map<string, section_t> ini_t;
    typedef ini_t::const_iterator Iter;
    mutable ini_t m_ini;
};
```

Listing 1

Let's start by going back to my sample INI file, and let us map it to a configuration data structure like the one in Listing 2.

Listing 3 is the piece of code that reads the parameters from the configuration file and assigns them to the appropriate variable in the data structure.

The library lives inside the namespace `boost::program_options`, here shortened to `po`.

We must first define a variable of type `po::options_description`, and add options to it. We do this by associating a name to a type and the corresponding variable in the configuration structure. Note that keys in a section have their name prefixed with the section name.

Having filled in the options description we can parse the configuration file (in this case passed in through the standard input, but any istream will do). As you can see, we need three lines of code to do this. The reason is that

```
struct configuration {
    unsigned int version;
    double energy;
    double mass;
    struct detector {
        double position_x;
        double position_y;
        double position_z;
        struct material {
            double atomic_number;
        } material;
    } detector;
};
```

Listing 2

```
configuration cfg;

namespace po=boost::program_options;
po::options_description desc;
desc.add_options()
    ("version", po::value<unsigned int>(&cfg.version))
    ("energy", po::value<double>(&cfg.energy))
    ("mass", po::value<double>(&cfg.mass))
    ("detector.position_x", po::value<double>(&cfg.detector.position_x))
    ("detector.position_y", po::value<double>(&cfg.detector.position_y))
    ("detector.position_z", po::value<double>(&cfg.detector.position_z))
    ("detector.material.atomic_number",
     po::value<double>(&cfg.detector.material.atomic_number))
    ;
const bool allow_unregistered = false;
po::variables_map vm;
po::store(po::parse_config_file(std::cin, desc, allow_unregistered), vm);
po::notify(vm);
```

the library allows one to merge options from multiple sources (for instance, default values from a configuration file that can be overridden on the command line), so that multiple calls to `po::store` may have to be executed before calling `po::notify`, that actually makes the parsing happen.

I mentioned earlier that the library has a number of features that simplify the life of the programmer.

Type checking

A first feature worth mentioning is that if the key value cannot be converted to the type specified, an exception will be thrown. For instance, if the configuration file contained the key

```
version = one
```

the program would be terminated with something like Listing 4.

Key name checks

What happens if the configuration file contains an extra key, or a key whose name is misspelled? This behaviour can be configured, but in most cases you'll want the program to tell you by disallowing keys whose name has not been registered, as shown in the example. In this case, if we had

```
versionx = 1
```

another exception would be thrown, with the message

```
what(): unrecognized option 'versionx'
```

Default values and required keys

What if one of the keys is missing from the configuration file? For instance, comment the version key out:

```
#version = 1
```

Without any measure from our part, the variable `version` will be uninitialized, and it will be assigned some random bit pattern:

```
version=1710991640
energy=10
mass=22
```

etc

We have now two options. The first one is to give certain keys a default value, by writing for instance:

```
("version", po::value<unsigned int>
 (&cfg.version)->default_value(99))
```

The second is to make version mandatory

```
("version", po::value<unsigned int>
 (&cfg.version)->required())
```

which leads to an exception when version is absent:

```
what(): the option 'version' is required but
missing
```

Multiple key instances

By default each key can appear only once in a configuration file, or an error is generated:

```
what(): option 'version' cannot be specified
more than once
```

But for some keys it may make sense to have multiple values, for instance

```
energy = 10
energy = 20
```

We make this possible by changing the declaration of energy to

```
std::vector<double> energy;
```

and the option description to

```
("energy",
 po::value<std::vector<double>>(&cfg.energy))
```

Multiple configuration sources

There can be multiple sources of configuration data, and these can be processed one after the other to obtain the final configuration. In my case the user may want to have a standard configuration file, and then under some circumstances modify one or more parameters without modifying the standard configuration file. This can be achieved by adding a second call to `po::store`, before the first one (the value of a key is set by the first parser that encounters the key name):

```
po::store(po::parse_command_line(argc, argv,
                                desc), vm);
```

So now the usage is as shown in Listing 5.

```
> ./main < example1.cfg
terminate called after throwing an instance of
'boost::exception_detail::clone_impl<boost::exception_detail::error_info_injector<boost::program_option
s::invalid_option_value>'
what(): the argument ('one') for option 'version' is invalid
Aborted (core dumped)
```

Perl is a Better Sed, and Python 2 is Good

Silas S. Brown sweats the differences between tools on common platforms.

If you've done any Unix shell scripting, you've probably come across the Stream Editor (sed). It's most often used for simple substitution, for example:

```
for N in *.wav ; do lame "$N" -o "$(echo "$N"|sed
-e 's/wav$/mp3/')"; done
```

which goes through all *.wav files and calls the MP3 encoder 'lame' on each one, passing a -o parameter as the filename with the wav at the end changed to mp3 – it's the sed -e s/x/y/ that does this substitution. [The -e argument allows you to provide multiple commands for a single invocation. Ed]

In this example, the \$ at the end of wav is there so that the substitution is made only at the very end of the filename; I don't want to confuse things

if a filename happens to contain 'wav' part-way through. In other situations you might want to add a g after the closing / to globally replace a regular expression many times in a line.

As this example shows, however, you do have to think carefully about your regular expressions (regexps), especially if you don't know what input you're going to get. In the above example, if I knew in advance exactly

SILAS S. BROWN

Silas S. Brown is a partially-sighted Computer Science post-doc in Cambridge who currently works in part-time assistant tuition. He has been an ACCU member since 1994 and can be contacted at ssb22@cam.ac.uk

Parsing Configuration Files in C++ with Boost (continued)

Listing 5

```
>./main --mass=99 --energy=123 < example1.cfg

version=1
description=this is the description
energy=123
mass=99
detector.position_x=1
detector.position_y=2
detector.position_z=3
detector.material.atomic_number = 46
```

Properties file format

You will have certainly noticed that in the options description code, keys in a section are specified as **section.key**, à la properties file. And in fact, the library can also ingest a properties file.

Custom types

The library support custom types, as long as they can be handled by Boost Lexical Cast. For a type to be handled by Lexical Cast it must be OutputStreamable, InputStreamable, CopyConstructible, and DefaultConstructible. If your type can be constructed from a string of tokens without any white space character in it, then there is nothing special you have to know.

On the other had, if you have a type like

```
struct special_type {
    double x, y;
    special_type(double x_, double y_) : x(x_),
        y(y_) {}
    special_type() : x(0), y(0) {}
};
```

things are not obvious (I had to dig into the Boost archives [2] to figure this one out), and the input stream operator for your type must be written as illustrated below because Lexical Cast does not ignore white space. See Listing 6.

Now the following works

```
special_type st =
    boost::lexical_cast<special_type>("12 13");
```

Listing 6

```
std::istream & operator>>(std::istream & is,
    special_type & val)
{
    is >> val.x;
    if((is.flags() & std::ios_base::skipws) == 0)
    {
        char whitespace;
        is >> whitespace;
    };
    is >> val.y;
    return is;
}
```

and as a consequence Program Options can handle something like

```
special = 99 101.1
```

and on the command line you can say --special="100 200".

Options style

Being primarily intended for the command line the Program Options library can be configured to handle command line options in various manners (one or two dashes, case sensitive or not etc.). Most of the default style options are not going to cause any surprise, but you want to be aware of the fact that by default Program Options will accept a shorter spelling of an option (or key in a configuration file) if it unambiguously identify the complete option. This is possibly not what you want, in which case you will have to alter this behaviour. Note that this only applies to the command line parser, because the configuration file parser is very strict, case sensitive, and does not allow shortening of the keys.

By combining INI/properties file parsing and command line options parsing in one interface, the Boost Program Options library allows one to easily layer multiple input sources and feed data of moderate complexity into a program. While the file format it supports is not as rich as others, the library has additional functionality that makes it worth considering. ■

References

- [1] Available at <http://www.boost.org/>
- [2] Start here for the complete story: <http://stackoverflow.com/questions/10382884/c-using-classes-with-boostlexical-cast>