# How to use boost::property_tree to load and write JSON

Jérémy Cochoy

2015/12/21

## Boost's Property Tree

**Property Tree** is a sublibrary of boost that allow you handling *tree of property*. It can be used to represent XML, JSON, INI files, file paths, etc. In our case, we will be interested in loading and writing JSON, to provide an interface with other applications.

Our example case will be the following json file :

```
{
    "height" : 320,
    "some" :
    {
        "complex" :
        {
            "path" : "hello"
        }
    },
    "animals" :
    {
        "rabbit" : "white",
        "dog" : "brown",
        "cat" : "grey"
    },
    "fruits" : ["apple", "raspberry", "orange"],
    "matrix" : [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
}
```

## Reading data

Let's have a look at how we can load those data into our c++ application.

### Setting up

First, we need to include the libraries and load the file.

```
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>

// Short alias for this namespace
namespace pt = boost::property_tree;

// Create a root
pt::ptree root;

// Load the json file in this ptree
pt::read_json("filename.json", root);
```

Now, we have a populated property tree thatis waiting from us to look at him. Notice that you can also read from a stream, for example `pt::read_json(std::cin, root)` is also allowed.

If your json file is illformed, you will be granted by a `pt::json_parser::json_parser_error` .

## Loading some values

We can access a value from the root by giving it's path to the `get` method.

```
// Read values
int height = root.get<int>("height", 0);
// You can also go through nested nodes
std::string msg = root.get<std::string>("some.complex.path");
```

If the field your are looking to doesn't exists, the `get()` method will throw a `pt::ptree_bad_path` exception, so that you can recorver from incomplete json files. Notice you can set a default value as second argument, or use `get_optional<T>()` wich return a `boost::optional<T>` .

Notice the getter doesn't care about the type of the input in the json file, but only rely on the ability to convert the string to the type you are asking.

## Browsing lists

So now, we would like to read a list of objects (in our cases, a list of animals).

We can handle it with a simple for loop, using an iterator. In **c++11**, it become :

```
// A vector to allow storing our animals
std::vector< std::pair<std::string, std::string> > animals;

// Iterator over all animals
for (pt::ptree::value_type &animal : root.get_child("animals"))
{
    // Animal is a std::pair of a string and a child

    // Get the label of the node
    std::string name = animal.first;
    // Get the content of the node
    std::string color = animal.second.data();
    animals.push_back(std::make_pair(name, color));
}
```

Since `animal.second` is a `ptree` , we can also call call `get()` or `get_child()` in the case our node wasn't just a string.

A bit more complexe example is given by a list of values. Each element of the list is actualy a `std::pair("", value)` (where value is a `ptree` ). It doesnt means that reading it is harder.

```
std::vector<std::string> fruits;
for (pt::ptree::value_type &fruit : root.get_child("fruits"))
{
    // fruit.first contain the string ""
    fruits.push_back(fruit.second.data());
}
```

In the case the values arent string, we can just call `fruit.second.get_value<T>()` in place of `fruit.second.data()` .

## Deeper : matrices

There is nothing now to enable reading of matrices, but it's a good way to check that you anderstood the reading of list. But enought talking, let's have a look at the code.

```
int matrix[3][3];
int x = 0;
for (pt::ptree::value_type &row : root.get_child("matrix"))
{
    int y = 0;
    for (pt::ptree::value_type &cell : row.second)
    {
        matrix[x][y] = cell.second.get_value<int>();
        y++;
    }
    x++;
}
```

You can now read any kind of JSON tree. The next step is being able to read them.

## Writing JSON

Let's say that now, we wan't to produce this tree from our application's data. To do that, all we have to do is build a `ptree` containing our data.

We start with an empty tree :

```
pt::ptree root;

//...

// Once our ptree was constructed, we can generate JSON on standard output
pt::write_json(std::cout, root);
```

## Add values

Puting values in a tree can be acomplished with the `put()` method.

```
root.put("height", height);
root.put("some.complex.path", "bonjour");
```

As you can see, very boring.

## Add a list of objects

No big deel here, although we now use `add_child()` to put our `animal` node at the root.

```
// Create a node
pt::ptree animals_node;
// Add animals as childs
for (auto &animal : animals)
    animals_node.put(animal.first, animal.second);
// Add the new node to the root
root.add_child("animals", animals_node);
```

## Add many nodes with the same name

Now start the tricky tricks. If you want to add more than one time a node named `fish` , you can't call the `put()` method. The call `node.put("name", value)` will replace the existing node named `name` . But you can do it by manually pushing your nodes, as demonstrated bellow.

```
// Add two objects with the same name
pt::ptree fish1;
fish1.put_value("blue");
pt::ptree fish2;
fish2.put_value("yellow");
oroot.push_back(std::make_pair("fish", fish1));
oroot.push_back(std::make_pair("fish", fish2));
```

## Add a list of values

If you remember, list are mades of nodes with empty name. Se we have to build node with empty names, and then use the `push_back()` once again to add all those unnamed childs.

```
// Add a list
pt::ptree fruits_node;
for (auto &fruit : fruits)
{
    // Create an unnamed node containing the value
    pt::ptree fruit_node;
    fruit_node.put("", fruit);

    // Add this node to the list.
    fruits_node.push_back(std::make_pair("", fruit_node));
}
root.add_child("fruits", fruits_node);
```

## Add a matrix

We already have all the tools needed to export our matrix. But let's demonstrate how to do it.

```
// Add a matrix
pt::ptree matrix_node;
for (int i = 0; i < 3; i++)
{
    pt::ptree row;
    for (int j = 0; j < 3; j++)
    {
        // Create an unnamed value
        pt::ptree cell;
        cell.put_value(matrix[i][j]);
        // Add the value to our row
        row.push_back(std::make_pair("", cell));
    }
    // Add the row to our matrix
    matrix_node.push_back(std::make_pair("", row));
}
// Add the node to the root
root.add_child("matrix", matrix_node);
```

# References

You can download a C++ example (./data/example.cpp) and the input JSON file (./data/example.json) for experimenting. Compile it with `clang++ -std=c++11 example.cpp -o example`.

Some links related :

- The official documentation (http://www.boost.org/doc/libs/1_60_0/doc/html/property_tree/tutorial.html)
- A post on stack overflow (http://stackoverflow.com/questions/2114466/creating-json-arrays-in-boost-using-property-trees)

**Rem** : At the moment, the boost::property_tree library doesn't output typed value. But we can expect that it will be corrected soon.