Chapter 25. Boost.PropertyTree

With the class `boost::property_tree::ptree`, Boost.PropertyTree provides a tree structure to store key/value pairs. Tree structure means that a trunk exists with numerous branches that have numerous twigs. A file system is a good example of a tree structure. File systems have a root directory with subdirectories that themselves can have subdirectories and so on.

To use `boost::property_tree::ptree`, include the header file `boost/property_tree/ptree.hpp`. This is a master header file, so no other header files need to be included for Boost.PropertyTree.

Example 25.1. Accessing data in `boost::property_tree::ptree`

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

using boost::property_tree::ptree;

int main()
{
  ptree pt;
  pt.put("C:.Windows.System", "20 files");

  ptree &c = pt.get_child("C:");
  ptree &windows = c.get_child("Windows");
  ptree &system = windows.get_child("System");
  std::cout << system.get_value<std::string>() << '\n';
}
```

Example 25.1 uses `boost::property_tree::ptree` to store a path to a directory. This is done with a call to `put()`. This member function expects two parameters because `boost::property_tree::ptree` is a tree structure that saves key/value pairs. The tree doesn't just consist of branches and twigs, a value must be assigned to each branch and twig. In Example 25.1 the value is "20 files".

The first parameter passed to `put()` is more interesting. It is a path to a directory. However, it doesn't use the backlash, which is the common path separator on Windows. It uses the dot.

You need to use the dot because it's the separator Boost.PropertyTree expects for keys. The parameter "C:.Windows.System" tells **`pt`** to create a branch called C: with a branch called Windows that has another branch called System. The dot creates the nested structure of branches. If "C:\Windows\System" had been passed as the parameter, **`pt`** would only have one branch called C:\Windows\System.

After the call to `put()`, **`pt`** is accessed to read the stored value "20 files" and write it to standard output. This is done by jumping from branch to branch – or directory to directory.

To access a subbranch, you call `get_child()`, which returns a reference to an object of the same type `get_child()` was called on. In Example 25.1, this is a reference to `boost::property_tree::ptree`. Because every branch can have subbranches, and because there is no structural difference between higher and lower branches, the same type is used.

The third call to `get_child()` retrieves the `boost::property_tree::ptree`, which represents the directory System. `get_value()` is called to read the value that was stored at the beginning of the example with `put()`.

Please note that `get_value()` is a function template. You pass the type of the return value as a template parameter. That way `get_value()` can do an automatic type conversion.

Example 25.2. Accessing data in `basic_ptree<std::string, int>`

```
#include <boost/property_tree/ptree.hpp>
#include <utility>
#include <iostream>

int main()
{
  typedef boost::property_tree::basic_ptree<std::string, int> ptree;
  ptree pt;
  pt.put(ptree::path_type{"C:\\Windows\\System", '\\'}, 20);
  pt.put(ptree::path_type{"C:\\Windows\\Cursors", '\\'}, 50);

  ptree &windows = pt.get_child(ptree::path_type{"C:\\Windows", '\\'});
  int files = 0;
  for (const std::pair<std::string, ptree> &p : windows)
    files += p.second.get_value<int>();
  std::cout << files << '\n';
}
```

There are two changes in Example 25.2 compared with Example 25.1. These changes are to save paths to directories and the number of files in directories more easily. First, paths use a backslash as the separator when passed to `put()`. Secondly, the number of files is stored as an `int`.

By default, Boost.PropertyTree uses a dot as the separator for keys. If you need to use another character, such as the backslash, as the separator, you don't pass the key as a string to `put()`. Instead you wrap it in an object of type `boost::property_tree::ptree::path_type`. The constructor of this class, which depends on `boost::property_tree::ptree`, takes the key as its first parameter and the separator character as its second parameter. That way, you can use a path such as C:\Windows\System, as shown in Example 25.2, without having to replace backslashes with dots.

`boost::property_tree::ptree` is based on the class template `boost::property_tree::basic_ptree`. Because keys and values are often strings, `boost::property_tree::ptree` is predefined. However, you can use `boost::property_tree::basic_ptree` with different types for keys and values. The tree in Example 25.2 uses an `int` to store the number of files in a directory rather than a string.

`boost::property_tree::ptree` provides the member functions `begin()` and `end()`. However, `boost::property_tree::ptree` only lets you iterate over the branches in one level. Example 25.2 iterates over the subdirectories of C:\Windows. You can't get an iterator to iterate over all branches in all levels.

The `for` loop in Example 25.2 reads the number of files in all subdirectories of C:\Windows to calculate a total. As a result, the example displays `70`. The example doesn't access objects of

type `ptree` directly. Instead it iterates over elements of type `std::pair<std::string, ptree>`. **`first`** contains the key of the current branch. That is System and Cursors in Example 25.2. **`second`** provides access to an object of type `ptree`, which represents the possible subdirectories. In the example, only the values assigned to System and Cursors are read. As in Example 25.1, the member function `get_value()` is called.

`boost::property_tree::ptree` only stores the value of the current branch, not its key. You can get the value with `get_value()`, but there is no member function to get the key. The key is stored in `boost::property_tree::ptree` one level up. This also explains why the `for` loop iterates over elements of type `std::pair<std::string, ptree>`.

Example 25.3. Accessing data with a translator

```
#include <boost/property_tree/ptree.hpp>
#include <boost/optional.hpp>
#include <iostream>
#include <cstdlib>

struct string_to_int_translator
{
  typedef std::string internal_type;
  typedef int external_type;

  boost::optional<int> get_value(const std::string &s)
  {
    char *c;
    long l = std::strtol(s.c_str(), &c, 10);
    return boost::make_optional(c != s.c_str(), static_cast<int>(l));
  }
};

int main()
{
  typedef boost::property_tree::iptree ptree;
  ptree pt;
  pt.put(ptree::path_type{"C:\\Windows\\System", '\\'}, "20 files");
  pt.put(ptree::path_type{"C:\\Windows\\Cursors", '\\'}, "50 files");

  string_to_int_translator tr;
  int files =
    pt.get<int>(ptree::path_type{"c:\\windows\\system", '\\'}, tr) +
    pt.get<int>(ptree::path_type{"c:\\windows\\cursors", '\\'}, tr);
  std::cout << files << '\n';
}
```

Example 25.3 uses with `boost::property_tree::iptree` another predefined tree from Boost.PropertyTree. In general, this type behaves like `boost::property_tree::ptree`. The only difference is that `boost::property_tree::iptree` doesn't distinguish between lower and upper case. For example, a value stored with the key C:\Windows\System can be read with c:\windows\system.

Unlike Example 25.1, `get_child()` isn't called multiple times to access subbranches. Just as `put()` can be used to store a value in a subbranch directly, a value from a subbranch can be read with `get()`. The key is defined the same way – for example using `boost::property_tree::iptree::path_type`.

Like `get_value()`, `get()` is a function template. You have to pass the type of the return value as a template parameter. Boost.PropertyTree does an automatic type conversion.

To convert types, Boost.PropertyTree uses *translators*. The library provides a few translators out of the box that are based on streams and can convert types automatically.

Example 25.3 defines the translator `string_to_int_translator`, which converts a value of type `std::string` to `int`. The translator is passed as an additional parameter to `get()`. Because the translator is just used to read, it only defines one member function, `get_value()`. If you want to use the translator for writing, too, then you would need to define a member function `put_value()` and then pass the translator as an additional parameter to `put()`.

`get_value()` returns a value of the type that is used in **`pt`**. However, because a type conversion doesn't always succeed, `boost::optional` is used. If a value is stored in Example 25.3 that can't be converted to an `int` with `std::strtol()`, an empty object of type `boost::optional` will be returned.

Please note that a translator must also define the two types `internal_type` and `external_type`. If you need to convert types when storing data, define `put_value()` similar to `get_value()`.

If you modify Example 25.3 to store the value "20" instead of value "20 files", `get_value()` can be called without passing a translator. The translators provided by Boost.PropertyTree can convert from `std::string` to `int`. However, the type conversion only succeeds when the entire string can be converted. The string must not contain any letters. Because `std::strtol()` can do a type conversion as long as the string starts with digits, the more liberal translator `string_to_int_translator` is used in Example 25.3.

Example 25.4. Various member functions of `boost::property_tree::ptree`

```
#include <boost/property_tree/ptree.hpp>
#include <utility>
#include <iostream>

using boost::property_tree::ptree;

int main()
{
  ptree pt;
  pt.put("C:.Windows.System", "20 files");

  boost::optional<std::string> c = pt.get_optional<std::string>("C:");
  std::cout << std::boolalpha << c.is_initialized() << '\n';

  pt.put_child("D:.Program Files", ptree{"50 files"});
  pt.add_child("D:.Program Files", ptree{"60 files"});

  ptree d = pt.get_child("D:");
  for (const std::pair<std::string, ptree> &p : d)
    std::cout << p.second.get_value<std::string>() << '\n';

  boost::optional<ptree&> e = pt.get_child_optional("E:");
  std::cout << e.is_initialized() << '\n';
}
```

You can call the member function `get_optional()` if you want to read the value of a key, but you aren't sure if the key exists. `get_optional()` returns the value in an object of type `boost::optional`. The object is empty if the key wasn't found. Otherwise, `get_optional()` works the same as `get()`.

It might seem like `put_child()` and `add_child()` are the same as `put()`. The difference is that `put()` creates only a key/value pair while `put_child()` and `add_child()` insert an entire subtree. Note that an object of type `boost::property_tree::ptree` is passed as the second parameter to `put_child()` and `add_child()`.

The difference between `put_child()` and `add_child()` is that `put_child()` accesses a key if that key already exists, while `add_child()` always inserts a new key into the tree. That's why the tree in Example 25.4 has two keys called "D:.Program Files". Depending on the use case, this can be confusing. If a tree represents a file system, there shouldn't be two identical paths. You have to avoid inserting identical keys if you don't want duplicates in a tree.

Example 25.4 displays the value of the keys below "D:" in the `for` loop. The example writes `50 files` and `60 files` to standard output, which proves there are two identical keys called "D:.Program Files".

The last member function introduced in Example 25.4 is `get_child_optional()`. This function is used like `get_child()`. `get_child_optional()` returns an object of type `boost::optional`. You call `boost::optional` if you aren't sure whether a key exists.

Example 25.5. Serializing a `boost::property_tree::ptree` in the JSON format

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

using namespace boost::property_tree;

int main()
{
  ptree pt;
  pt.put("C:.Windows.System", "20 files");
  pt.put("C:.Windows.Cursors", "50 files");

  json_parser::write_json("file.json", pt);

  ptree pt2;
  json_parser::read_json("file.json", pt2);

  std::cout << std::boolalpha << (pt == pt2) << '\n';
}
```

Boost.PropertyTree does more than just provide structures to manage data in memory. As can be seen in Example 25.5, the library also provides functions to save a `boost::property_tree::ptree` in a file and load it from a file.

The header file `boost/property_tree/json_parser.hpp` provides access to the functions `boost::property_tree::json_parser::write_json()` and `boost::property_tree::json_parser::read_json()`. These functions make it possible to

save and load a `boost::property_tree::ptree` serialized in the JSON format. That way you can support configuration files in the JSON format.

If you want to call functions that store a `boost::property_tree::ptree` in a file or load it from a file, you must include header files such as `boost/property_tree/json_parser.hpp`. It isn't sufficient to only include `boost/property_tree/ptree.hpp`.

In addition to the functions `boost::property_tree::json_parser::write_json()` and `boost::property_tree::json_parser::read_json()`, Boost.PropertyTree provides functions for additional data formats. You use `boost::property_tree::ini_parser::write_ini()` and `boost::property_tree::ini_parser::read_ini()` from `boost/property_tree/ini_parser.hpp` to support INI-files. With `boost::property_tree::xml_parser::write_xml()` and `boost::property_tree::xml_parser::read_xml()` from `boost/property_tree/xml_parser.hpp`, data can be loaded and stored in XML format. With `boost::property_tree::info_parser::write_info()` and `boost::property_tree::info_parser::read_info()` from `boost/property_tree/info_parser.hpp`, you can access another format that was developed and optimized to serialize trees from Boost.PropertyTree.

None of the supported formats guarantees that a `boost::property_tree::ptree` will look the same after it has been saved and reloaded. For example, the JSON format can lose type information because `boost::property_tree::ptree` can't distinguish between `true` and "true". The type is always the same. Even if the various functions make it easy to save and load a `boost::property_tree::ptree`, don't forget that Boost.PropertyTree doesn't support the formats completely. The main focus of the library is on the structure `boost::property_tree::ptree` and not on supporting various data formats.

Exercise

Create a program that loads this JSON-file and writes the names of all animals to standard output. If "all" is set to `true` the program should not only write the names but all properties of all animals to standard output:

```
{
  "animals": [
    {
      "name": "cat",
      "legs": 4,
      "has_tail": true
    },
    {
      "name": "spider",
      "legs": 8,
      "has_tail": false
    }
  ],
  "log": {
    "all": true
  }
}
```