Boost.PropertyTree

Marcin Kalicinski

Table of Contents

What is Property Tree?

The Property Tree library provides a data structure that stores an arbitrarily deeply nested tree of values, indexed at each level by some key. Each node of the tree stores its own value, plus an ordered list of its subnodes and their keys. The tree allows easy access to any of its nodes by means of a path, which is a concatenation of multiple keys.

In addition, the library provides parsers and generators for a number of data formats that can be represented by such a tree, including XML, INI, and JSON.

Property trees are versatile data structures, but are particularly suited for holding configuration data. The tree provides its own, tree-specific interface, and each node is also an STL-compatible Sequence for its child nodes.

Conceptually, then, a node can be thought of as the following structure:

```
struct ptree
{
   data_type data;                          // data associated with the node
   list< pair<key_type, ptree> > children; // ordered list of named children
};
```

Both key_type and data_type are configurable, but will usually be std::string.

Many software projects develop a similar tool at some point of their lifetime, and property tree originated the same way. We hope the library can save many from reinventing the wheel.

Five Minute Tutorial

This tutorial uses XML. Note that the library is not specifically bound to XML, and any other supported format (such as INI or JSON) could be used instead. XML was chosen because the author thinks that wide range of people is familiar with it.

Suppose we are writing a logging system for some application, and need to read log configuration from a file when the program starts. The file with the log configuration looks like this:

```
<debug>
    <filename>debug.log</filename>
    <modules>
        <module>Finance</module>
        <module>Admin</module>
        <module>HR</module>
    </modules>
    <level>2</level>
</debug>
```

It contains the log filename, a list of modules where logging is enabled, and the debug level value. To store the logging configuration in the program we created a debug_settings structure:

```
struct debug_settings
{
    std::string m_file;          // log filename
    int m_level;                 // debug level
    std::set<string> m_modules;  // modules where logging is enabled
    void load(const std::string &filename);
    void save(const std::string &filename);
};
```

All that needs to be done now is to write implementations of load() and save() member functions. Let's first deal with load(). It contains just 7 lines of code, although it does all the necessary things, including error reporting:

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

// Loads debug_settings structure from the specified XML file
void debug_settings::load(const std::string &filename)
{
    // Create an empty property tree object
    using boost::property_tree::ptree;
    ptree pt;

    // Load the XML file into the property tree. If reading fails
    // (cannot open file, parse error), an exception is thrown.
    read_xml(filename, pt);

    // Get the filename and store it in the m_file variable.
    // Note that we construct the path to the value by separating
    // the individual keys with dots. If dots appear in the keys,
    // a path type with a different separator can be used.
    // If the debug.filename key is not found, an exception is thrown.
    m_file = pt.get<std::string>("debug.filename");

    // Get the debug level and store it in the m_level variable.
    // This is another version of the get method: if the value is
    // not found, the default value (specified by the second
    // parameter) is returned instead. The type of the value
    // extracted is determined by the type of the second parameter,
    // so we can simply write get(...) instead of get<int>(...).
    m_level = pt.get("debug.level", 0);

    // Iterate over the debug.modules section and store all found
    // modules in the m_modules set. The get_child() function
    // returns a reference to the child at the specified path; if
    // there is no such child, it throws. Property tree iterators
    // are models of BidirectionalIterator.
    BOOST_FOREACH(ptree::value_type &v,
            pt.get_child("debug.modules"))
        m_modules.insert(v.second.data());
}
```

Now the save() function. It is also 7 lines of code:

```
// Saves the debug_settings structure to the specified XML file
void debug_settings::save(const std::string &filename)
{
   // Create an empty property tree object
   using boost::property_tree::ptree;
   ptree pt;

   // Put log filename in property tree
   pt.put("debug.filename", m_file);

   // Put debug level in property tree
   pt.put("debug.level", m_level);

   // Iterate over the modules in the set and put them in the
   // property tree. Note that the put function places the new
   // key at the end of the list of keys. This is fine most of
   // the time. If you want to place an item at some other place
   // (i.e. at the front or somewhere in the middle), this can
   // be achieved using a combination of the insert and put_own
   // functions.
   BOOST_FOREACH(const std::string &name, m_modules)
      pt.__ptree_add__("debug.modules.module", name);

   // Write the property tree to the XML file.
   write_xml(filename, pt);
}
```

The full program debug_settings.cpp is included in the examples directory.

Property Tree as a Container

Every property tree node models the ReversibleSequence concept, providing access to its immediate children. This means that iterating over a `ptree` (which is the same as its root node - every `ptree` node is also the subtree it starts) iterates only a single level of the hierarchy. There is no way to iterate over the entire tree.

It is very important to remember that the property sequence is **not** ordered by the key. It preserves the order of insertion. It closely resembles a std::list. Fast access to children by name is provided via a separate lookup structure. Do not attempt to use algorithms that expect an ordered sequence (like binary_search) on a node's children.

There may be multiple children with the same key value in a node. However, these children are not necessarily sequential. The iterator returned by `find` may refer to any of these, and there are no guarantees about the relative position of the other equally named children.

Property Tree Synopsis

The central component of the library is the `basic_ptree` class template. Instances of this class are property trees. It is parametrized on key and data type, and key comparison policy; `ptree`, `wptree`, `iptree` and `wiptree` are typedefs of `basic_ptree` using predefined combinations of template parameters. Property tree is basically a somewhat simplified standard container (the closest being std::list), plus a bunch of extra member functions. These functions allow easy and effective access to the data stored in property tree. They are various variants of `get`, `put`, `get_value`, `put_value`, `get_child`, `put_child`. Additionally, there is `data` function to access node data directly.

See the `basic_ptree class template synopsis` for more information.

How to Populate a Property Tree

XML Parser

The XML format is an industry standard for storing information in textual form. Unfortunately, there is no XML parser in Boost as of the time of this writing. The library therefore contains the fast and tiny RapidXML parser (currently in version 1.13) to provide XML parsing support. RapidXML does not fully support the XML standard; it is not capable of parsing DTDs and therefore cannot do full entity substitution.

By default, the parser will preserve most whitespace, but remove element content that consists only of whitespace. Encoded whitespaces (e.g.) does not count as whitespace in this regard. You can pass the trim_whitespace flag if you want all leading and trailing whitespace trimmed and all continuous whitespace collapsed into a single space.

Please note that RapidXML does not understand the encoding specification. If you pass it a character buffer, it assumes the data is already correctly encoded; if you pass it a filename, it will read the file using the character conversion of the locale you give it (or the global locale if you give it none). This means that, in order to parse a UTF-8-encoded XML file into a wptree, you have to supply an alternate locale, either directly or by replacing the global one.

XML / property tree conversion schema (`read_xml` and `write_xml`):

- Each XML element corresponds to a property tree node. The child elements correspond to the children of the node.
- The attributes of an XML element are stored in the subkey `<xmlattr>`. There is one child node per attribute in the attribute node. Existence of the `<xmlattr>` node is not guaranteed or necessary when there are no attributes.
- XML comments are stored in nodes named `<xmlcomment>`, unless comment ignoring is enabled via the flags.
- Text content is stored in one of two ways, depending on the flags. The default way concatenates all text nodes and stores them in a single node called `<xmltext>`. This way, the entire content can be conveniently read, but the relative ordering of text and child elements is lost. The other way stores each text content as a separate node, all called `<xmltext>`.

The XML storage encoding does not round-trip perfectly. A read-write cycle loses trimmed whitespace, low-level formatting information, and the distinction between normal data and CDATA nodes. Comments are only preserved when enabled. A write-read cycle loses trimmed whitespace; that is, if the origin tree has string data that starts or ends with whitespace, that whitespace is lost.

JSON Parser

The JSON format is a data interchange format derived from the object literal notation of JavaScript. (JSON stands for JavaScript Object Notation.) JSON is a simple, compact format for loosely structured node trees of any depth, very similar to the property tree dataset. It is less stuctured than XML and has no schema support, but has the advantage of being simpler, smaller and typed without the need for a complex schema.

The property tree dataset is not typed, and does not support arrays as such. Thus, the following JSON / property tree mapping is used:

- JSON objects are mapped to nodes. Each property is a child node.
- JSON arrays are mapped to nodes. Each element is a child node with an empty name. If a node has both named and unnamed child nodes, it cannot be mapped to a JSON representation.
- JSON values are mapped to nodes containing the value. However, all type information is lost; numbers, as well as the literals "null", "true" and "false" are simply mapped to their string form.
- Property tree nodes containing both child nodes and data cannot be mapped.

JSON round-trips, except for the type information loss.

For example this JSON:

```
{
   "menu":
   {
      "foo": true,
      "bar": "true",
      "value": 102.3E+06,
      "popup":
      [
         {"value": "New", "onclick": "CreateNewDoc()"},
         {"value": "Open", "onclick": "OpenDoc()"},
      ]
   }
}
```

will be translated into the following property tree:

```
menu
{
   foo true
   bar true
   value 102.3E+06
   popup
   {
      ""
      {
         value New
         onclick CreateNewDoc()
      }
      ""
      {
         value Open
         onclick OpenDoc()
      }
   }
}
```

INI Parser

The INI format was once widely used in the world of Windows. It is now deprecated, but is still used by a surprisingly large number of applications. The reason is probably its simplicity, plus that Microsoft recommends using the registry as a replacement, which not all developers want to do.

INI is a simple key-value format with a single level of sectioning. It is thus less rich than the property tree dataset, which means that not all property trees can be serialized as INI files.

The INI parser creates a tree node for every section, and a child node for every property in that section. All properties not in a section are directly added to the root node. Empty sections are ignored. (They don't round-trip, as described below.)

The INI serializer reverses this process. It first writes out every child of the root that contains data, but no child nodes, as properties. Then it creates a section for every child that contains child nodes, but no data. The children of the sections must only contain data. It is an error if the root node contains data, or any child of the root contains both data and content, or there's more than three levels of hierarchy. There must also not be any duplicate keys.

An empty tree node is assumed to be an empty property. There is no way to create empty sections.

Since the Windows INI parser discards trailing spaces and does not support quoting, the property tree parser follows this example. This means that property values containing trailing spaces do not round-trip.

INFO Parser

The INFO format was created specifically for the property tree library. It provides a simple, efficient format that can be used to serialize property trees that are otherwise only stored in memory. It can also be used for any other purpose, although the lack of widespread existing use may prove to be an impediment.

INFO provides several features that make it familiar to C++ programmers and efficient for medium-sized datasets, especially those used for test input. It supports C-style character escapes, nesting via curly braces, and file inclusion via #include.

INFO is also used for visualization of property trees in this documentation.

A typical INFO file might look like this:

```
key1 value1
key2
{
   key3 value3
   {
      key4 "value4 with spaces"
   }
   key5 value5
}
```

Here's a more complicated file demonstrating all of INFO's features:

```
; A comment
key1 value1   ; Another comment
key2 "value with special characters in it {};#\n\t\"\0"
{
   subkey "value split "\
          "over three"\
          "lines"
   {
      a_key_without_value ""
      "a key with special characters in it {};#\n\t\"\0" ""
      "" value    ; Empty key with a value
      "" ""       ; Empty key with empty value!
   }
}
#include "file.info"    ; included file
```

INFO round-trips except for the loss of comments and include directives.

1. These parsers will be dropped for now.

2. [include cmd_line_parser.qbk]

3. [include windows_registry_parser.qbk]

4. [include system_environment_parser.qbk]

How to Access Data in a Property Tree

Property tree resembles (almost is) a standard container with value type of `pair<string, ptree>`. It has the usual member functions, such as `insert`, `push_back`, `find`, `erase`, etc. These can of course be used to populate and access the tree. For example the following code adds key `"pi"` with data (almost) equal to mathematical *pi* value:

```
ptree pt;
pt.push_back(ptree::value_type("pi", ptree("3.14159")));
```

To find the value of `pi` we might do the following:

```
ptree::const_iterator it = pt.find("pi");
double pi = boost::lexical_cast<double>(it->second._ptree_data__());
```

This looks quite cumbersome, and would be even more so if `pi` value was not stored so near the top of the tree, and we cared just a little bit more about errors. Fortunately, there is another, correct way of doing it:

```
ptree pt;
pt.put("pi", 3.14159);                // put double
double pi = pt.get<double>("pi");     // get double
```

It doesn't get simpler than that. Basically, there are 2 families of member functions, `get` and `put`, which allow intuitive access to data stored in the tree (direct children or not).

Three Ways of Getting Data

There are three versions of get: get, get (default-value version), and get_optional, which differ by failure handling strategy. All versions take path specifier, which determines in which key to search for a value. It can be a single key, or a path to key, where path elements are separated with a special character (a '.' if not specified differently). For example debug.logging.errorlevel might be a valid path with dot as a separator.

1. The throwing version (`get`):

   ```
   ptree pt;
   /* ... */
   float v = pt.get<float>("a.path.to.float.value");
   ```

   This call locates the proper node in the tree and tries to translate its data string to a float value. If that fails, exception is thrown. If path does not exist, it will be `ptree_bad_path` exception. If value could not be translated, it will be `ptree_bad_data`. Both of them derive from `ptree_error` to make common handling possible.

2. The default-value version (`get`):

   ```
   ptree pt;
   /* ... */
   float v = pt.get("a.path.to.float.value", -1.f);
   ```

   It will do the same as above, but if it fails, it will return the default value specified by second parameter (here -1.f) instead of throwing. This is very useful in common situations where one wants to allow omitting of some keys. Note that type specification needed in throwing version is normally not necessary here, because type is determined by the default value parameter.

3. The optional version (`get_optional`):

   ```
   ptree pt;
   /* ... */
   boost::optional<float> v = pt.get_optional<float>("a.path.to.float.value");
   ```

   This version uses boost::optional class to handle extraction failure. On successful extraction, it will return boost::optional initialized with extracted value. Otherwise, it will return uninitialized boost::optional.

To retrieve value from this tree (not some subkey), use `get_value`, `get_value` (default-value version), and `get_value_optional`. They have identical semantics to `get` functions, except they don't take the `path` parameter. Don't call `get` with and empty `path` to do this as it will try to extract contents of subkey with empty name.

To use separator character other than default '.', each of the `get` versions has another form, which takes an additional parameter in front of path. This parameter of type `char/wchar_t` specifies the separating character. This is a lifesaving device for those who may have dots in their keys:

```
pt.get<float>('/', "p.a.t.h/t.o/v.a.l.u.e");
pt.get('/', "p.a.t.h/t.o/v.a.l.u.e", 0, NULL);
pt.get_optional<std::string>('/', "p.a.t.h/t.o/v.a.l.u.e");
```

One Way of Putting Data

To complement `get`, there is a `put` function. Contrary to `get`, it has only one variant. The reason is, there is no need to handle put failures so often (normally, `put` will only fail if conversion of the supplied value to `data_type` fails or the system runs out of memory. In the former case, `put` will throw `ptree_bad_data`). Sample usage of `put` might appear as:

```
ptree pt;
pt.put("a.path.to.float.value", 3.14f);
```

Calling `put` will insert a new value at specified path, so that a call to `get` specifying the same path will retrieve it. Further, `put` will insert any missing path elements during path traversal. For example, calling `put("key1.key2.key3", 3.14f)` on an empty tree will insert three new children: `key1`, `key1.key2` and `key1.key2.key3`. The last one will receive a string `"3.14"` as data, while the two former ones will have empty data strings. `put` always inserts new keys at the back of the existing sequences.

Similar to `get_value`, there is also a `put_value` function. It does the same for this property tree what `put` does for its children. Thus, it does not require `path`:

```
ptree pt;
pt.__ptree_put_own__(3.14f);
```

Appendices

Compatibility

Property tree uses partial class template specialization. There has been no attempt to work around lack of support for this. The library will therefore most probably not work with Visual C++ 7.0 or earlier, or gcc 2.x.

Property tree has been tested (regressions successfully compiled and run) with the following compilers:

- Visual C++ 8.0
- gcc 3.4.2 (MinGW)
- gcc 3.3.5 (Linux)
- gcc 3.4.4 (Linux)
- gcc 4.3.3 (Linux)
- Intel C++ 9.0 (Linux)

Rationale

1. **Why are there 3 versions of `get`? Couldn't there be just one?** The three versions reflect experience gathered during several of years of using property tree in several different applications. During that time I tried hard to come up with one, proper form of the get function, and failed. I know of these three basic patterns of usage:

   - *Just get the data and I do not care if it cannot be done.* This is used when the programmer is fairly sure that data exists. Or in homework assignments. Or when tomorrow is final deadline for your project.

   - *Get the data and revert to default value if it cannot be done.* Used when you want to allow omitting the key in question. Implemented by some similar tools (windows INI file access functions).

   - *Get the data, but I care more whether you succeeded than I do for the data itself.* Used when you want to vary control flow depending on get success/failure. Or to check for presence of a key.

Future Development

- More parsers: YAML, environment strings.
- More robust XML parser.
- Mathematical relations: ptree difference, union, intersection. Useful for finding configuration file changes etc.

Reference

Header <boost/property_tree/exceptions.hpp>

```
namespace boost {
  namespace property_tree {
    class ptree_error;
    class ptree_bad_data;
    class ptree_bad_path;
  }
}
```

Class ptree_error

boost::property_tree::ptree_error — Base class for all property tree errors. Derives from `std::runtime_error`. Call member function `what` to get human readable message associated with the error.

Synopsis

```
// In header: <boost/property_tree/exceptions.hpp>


class ptree_error {
public:
  // construct/copy/destruct
  ptree_error(const std::string &);
  ~ptree_error();
};
```

Description

`ptree_error` public construct/copy/destruct

1. ```
   ptree_error(const std::string & what);
   ```

   Instantiate a ptree_error instance with the given message.

   Parameters: `what` The message to associate with this error.

2. ```
   ~ptree_error();
   ```

Class ptree_bad_data

boost::property_tree::ptree_bad_data — Error indicating that translation from given value to the property tree data_type (or vice versa) failed. Derives from ptree_error.

Synopsis

```
// In header: <boost/property_tree/exceptions.hpp>


class ptree_bad_data : public boost::property_tree::ptree_error {
public:
  // construct/copy/destruct
  template<typename T> ptree_bad_data(const std::string &, const T &);
  ~ptree_bad_data();

  // public member functions
  template<typename T> T data();
};
```

Description

`ptree_bad_data` public construct/copy/destruct

1. ```
   template<typename T> ptree_bad_data(const std::string & what, const T & data);
   ```

   Instantiate a ptree_bad_data instance with the given message and data.

   Parameters:

`data`	The value associated with this error that was the source of the translation failure.
`what`	The message to associate with this error.

2. ```
   ~ptree_bad_data();
   ```

`ptree_bad_data` public member functions

1. ```
   template<typename T> T data();
   ```

   Retrieve the data associated with this error. This is the source value that failed to be translated.
   ```

Class ptree_bad_path

boost::property_tree::ptree_bad_path — Error indicating that specified path does not exist. Derives from ptree_error.

Synopsis

```
// In header: <boost/property_tree/exceptions.hpp>


class ptree_bad_path : public boost::property_tree::ptree_error {
public:
  // construct/copy/destruct
  template<typename T> ptree_bad_path(const std::string &, const T &);
  ~ptree_bad_path();

  // public member functions
  template<typename T> T path();
};
```

Description

`ptree_bad_path` public construct/copy/destruct

1. ```
   template<typename T> ptree_bad_path(const std::string & what, const T & path);
   ```

   Instantiate a ptree_bad_path with the given message and path data.

   Parameters:
   - `path` The path that could not be found in the property_tree.
   - `what` The message to associate with this error.

2. ```
   ~ptree_bad_path();
   ```

`ptree_bad_path` public member functions

1. ```
   template<typename T> T path();
   ```

   Retrieve the invalid path.

Header <boost/property_tree/id_translator.hpp>

```
namespace boost {
  namespace property_tree {
    template<typename T> struct id_translator;

    template<typename T> struct translator_between<T, T>;
    template<typename Ch, typename Traits, typename Alloc>
      struct translator_between<std::basic_string< Ch, Traits, Alloc >, std::ba↵
sic_string< Ch, Traits, Alloc >>;
  }
}
```
```

Struct template id_translator

boost::property_tree::id_translator — Simple implementation of the Translator concept. It does no translation.

Synopsis

```
// In header: <boost/property_tree/id_translator.hpp>

template<typename T>
struct id_translator {
  // types
  typedef T internal_type;
  typedef T external_type;

  // public member functions
  boost::optional< T > get_value(const T &);
  boost::optional< T > put_value(const T &);
};
```

Description

`id_translator` public member functions

1. ```
   boost::optional< T > get_value(const T & v);
   ```

2. ```
   boost::optional< T > put_value(const T & v);
   ```

Struct template translator_between<T, T>

boost::property_tree::translator_between<T, T>

Synopsis

```
// In header: <boost/property_tree/id_translator.hpp>

template<typename T>
struct translator_between<T, T> {
  // types
  typedef id_translator< T > type;
};
```

Struct template translator_between<std::basic_string< Ch, Traits, Alloc >, std::basic_string< Ch, Traits, Alloc >>

boost::property_tree::translator_between<std::basic_string< Ch, Traits, Alloc >, std::basic_string< Ch, Traits, Alloc >>

Synopsis

```
// In header: <boost/property_tree/id_translator.hpp>

template<typename Ch, typename Traits, typename Alloc>
struct translator_between<std::basic_string< Ch, Traits, Alloc >, std::basic_string< Ch, Traits, Al↵
loc >> {
  // types
  typedef id_translator< std::basic_string< Ch, Traits, Alloc > > type;
};
```

Header <boost/property_tree/info_parser.hpp>

```
namespace boost {
  namespace property_tree {
    namespace info_parser {
      template<typename Ptree, typename Ch>
        void read_info(std::basic_istream< Ch > &, Ptree &);
      template<typename Ptree, typename Ch>
        void read_info(std::basic_istream< Ch > &, Ptree &, const Ptree &);
      template<typename Ptree>
        void read_info(const std::string &, Ptree &,
                       const std::locale & = std::locale());
      template<typename Ptree>
        void read_info(const std::string &, Ptree &, const Ptree &,
                       const std::locale & = std::locale());
      template<typename Ptree, typename Ch>
        void write_info(std::basic_ostream< Ch > &, const Ptree &,
                        const info_writer_settings< Ch > & = info_writer_settings< Ch >());
      template<typename Ptree>
        void write_info(const std::string &, const Ptree &,
                        const std::locale & = std::locale(),
                        const info_writer_settings< type↵
name Ptree::key_type::value_type > & = info_writer_make_settings< type↵
name Ptree::key_type::value_type >());
    }
  }
}
```

Function template read_info

boost::property_tree::info_parser::read_info

Synopsis

```
// In header: <boost/property_tree/info_parser.hpp>


template<typename Ptree, typename Ch>
  void read_info(std::basic_istream< Ch > & stream, Ptree & pt);
```

Description

Read INFO from a the given stream and translate it to a property tree.

Throws: info_parser_error
Notes: Replaces the existing contents. Strong exception guarantee.

Function template read_info

boost::property_tree::info_parser::read_info

Synopsis

```
// In header: <boost/property_tree/info_parser.hpp>


template<typename Ptree, typename Ch>
  void read_info(std::basic_istream< Ch > & stream, Ptree & pt,
                 const Ptree & default_ptree);
```

Description

Read INFO from a the given stream and translate it to a property tree.

Parameters:	`default_ptree`	If parsing fails, pt is set to a copy of this tree.
Notes:	Replaces the existing contents. Strong exception guarantee.	

Function template read_info

boost::property_tree::info_parser::read_info

Synopsis

```
// In header: <boost/property_tree/info_parser.hpp>


template<typename Ptree>
  void read_info(const std::string & filename, Ptree & pt,
                 const std::locale & loc = std::locale());
```

Description

Read INFO from a the given file and translate it to a property tree. The tree's key type must be a string type, i.e. it must have a nested value_type typedef that is a valid parameter for basic_ifstream.

Throws: info_parser_error
Notes: Replaces the existing contents. Strong exception guarantee.

Function template read_info

boost::property_tree::info_parser::read_info

Synopsis

```
// In header: <boost/property_tree/info_parser.hpp>


template<typename Ptree>
  void read_info(const std::string & filename, Ptree & pt,
                 const Ptree & default_ptree,
                 const std::locale & loc = std::locale());
```

Description

Read INFO from a the given file and translate it to a property tree. The tree's key type must be a string type, i.e. it must have a nested value_type typedef that is a valid parameter for basic_ifstream.

Parameters:	`default_ptree`	If parsing fails, pt is set to a copy of this tree.
Notes:	Replaces the existing contents. Strong exception guarantee.	

Function template write_info

boost::property_tree::info_parser::write_info

Synopsis

```
// In header: <boost/property_tree/info_parser.hpp>


template<typename Ptree, typename Ch>
  void write_info(std::basic_ostream< Ch > & stream, const Ptree & pt,
                  const info_writer_settings< Ch > & settings = info_writer_settings< Ch >());
```

Description

Writes a tree to the stream in INFO format.

Parameters: `settings` The settings to use when writing the INFO data.
Throws: info_parser_error

Function template write_info

boost::property_tree::info_parser::write_info

Synopsis

```
// In header: <boost/property_tree/info_parser.hpp>


template<typename Ptree>
  void write_info(const std::string & filename, const Ptree & pt,
                  const std::locale & loc = std::locale(),
                  const info_writer_settings< typename Ptree::key_type::value_type > & set↵
tings = info_writer_make_settings< typename Ptree::key_type::value_type >());
```

Description

Writes a tree to the file in INFO format. The tree's key type must be a string type, i.e. it must have a nested value_type typedef that is a valid parameter for basic_ofstream.

Parameters: `settings` The settings to use when writing the INFO data.
Throws: info_parser_error

Header <boost/property_tree/ini_parser.hpp>

```
namespace boost {
  namespace property_tree {
    namespace ini_parser {
      class ini_parser_error;
      bool validate_flags(int);
      template<typename Ptree>
        void read_ini(std::basic_istream< typename Ptree::key_type::value_type > &,
                      Ptree &);
      template<typename Ptree>
        void read_ini(const std::string &, Ptree &,
                      const std::locale & = std::locale());
      template<typename Ptree>
        void write_ini(std::basic_ostream< typename Ptree::key_type::value_type > &,
                       const Ptree &, int = 0);
      template<typename Ptree>
        void write_ini(const std::string &, const Ptree &, int = 0,
                       const std::locale & = std::locale());
    }
  }
}
```

Class ini_parser_error

boost::property_tree::ini_parser::ini_parser_error

Synopsis

```
// In header: <boost/property_tree/ini_parser.hpp>


class ini_parser_error {
public:
  // construct/copy/destruct
  ini_parser_error(const std::string &, const std::string &, unsigned long);
};
```

Description

Indicates an error parsing INI formatted data.

`ini_parser_error` public construct/copy/destruct

1. ```
   ini_parser_error(const std::string & message, const std::string & filename,
                    unsigned long line);
   ```

   Construct an `ini_parser_error`

Parameters:		
	`filename`	The name of the file being parsed containing the error.
	`line`	The line in the given file where an error was encountered.
	`message`	Message describing the parser error.
   ```

Function validate_flags

boost::property_tree::ini_parser::validate_flags

Synopsis

```
// In header: <boost/property_tree/ini_parser.hpp>


bool validate_flags(int flags);
```

Description

Determines whether the `flags` are valid for use with the ini_parser.

Parameters: `flags` value to check for validity as flags to ini_parser.
Returns: true if the flags are valid, false otherwise.

Function template read_ini

boost::property_tree::ini_parser::read_ini

Synopsis

```
// In header: <boost/property_tree/ini_parser.hpp>


template<typename Ptree>
  void read_ini(std::basic_istream< typename Ptree::key_type::value_type > & stream,
                Ptree & pt);
```

Description

Read INI from a the given stream and translate it to a property tree.

Parameters:	`pt`	The property tree to populate.
	`stream`	Stream from which to read in the property tree.
Throws:	ini_parser_error	
Notes:	Clears existing contents of property tree. In case of error the property tree is not modified.	

Function template read_ini

boost::property_tree::ini_parser::read_ini

Synopsis

```
// In header: <boost/property_tree/ini_parser.hpp>


template<typename Ptree>
  void read_ini(const std::string & filename, Ptree & pt,
                const std::locale & loc = std::locale());
```

Description

Read INI from a the given file and translate it to a property tree.

Parameters:	`filename`	Name of file from which to read in the property tree.
	`loc`	The locale to use when reading in the file contents.
	`pt`	The property tree to populate.
Throws:	ini_parser_error	
Notes:	Clears existing contents of property tree. In case of error the property tree unmodified.	

Function template write_ini

boost::property_tree::ini_parser::write_ini

Synopsis

```
// In header: <boost/property_tree/ini_parser.hpp>


template<typename Ptree>
  void write_ini(std::basic_ostream< typename Ptree::key_type::value_type > & stream,
                 const Ptree & pt, int flags = 0);
```

Description

Translates the property tree to INI and writes it the given output stream.

Parameters:	`flags`	The flags to use when writing the INI file. No flags are currently supported.
	`pt`	The property tree to tranlsate to INI and output.
	`stream`	The stream to which to write the INI representation of the property tree.
Requires:	*pt* cannot have data in its root.	
	pt cannot have keys both data and children.	
	pt cannot be deeper than two levels.	
	There cannot be duplicate keys on any given level of *pt*.	
Throws:	ini_parser_error	

Function template write_ini

boost::property_tree::ini_parser::write_ini

Synopsis

```
// In header: <boost/property_tree/ini_parser.hpp>


template<typename Ptree>
  void write_ini(const std::string & filename, const Ptree & pt,
                 int flags = 0, const std::locale & loc = std::locale());
```

Description

Translates the property tree to INI and writes it the given file.

Parameters:

`filename`	The name of the file to which to write the INI representation of the property tree.
`flags`	The flags to use when writing the INI file. The following flags are supported: • `skip_ini_validity_check` -- Skip check if ptree is a valid ini. The validity check covers the preconditions but takes `O(n log n)` time.
`loc`	The locale to use when writing the file.
`pt`	The property tree to tranlsate to INI and output.

Requires: *pt* cannot have data in its root.

pt cannot have keys both data and children.

pt cannot be deeper than two levels.

There cannot be duplicate keys on any given level of *pt*.

Throws: info_parser_error

Header <boost/property_tree/json_parser.hpp>

```
namespace boost {
  namespace property_tree {
    namespace json_parser {
      template<typename Ptree>
        void read_json(std::basic_istream< typename Ptree::key_type::value_type > &,
                       Ptree &);
      template<typename Ptree>
        void read_json(const std::string &, Ptree &,
                       const std::locale & = std::locale());
      template<typename Ptree>
        void write_json(std::basic_ostream< typename Ptree::key_type::value_type > &,
                        const Ptree &, bool = true);
      template<typename Ptree>
        void write_json(const std::string &, const Ptree &,
                        const std::locale & = std::locale(), bool = true);
    }
  }
}
```

Function template read_json

boost::property_tree::json_parser::read_json

Synopsis

```
// In header: <boost/property_tree/json_parser.hpp>


template<typename Ptree>
  void read_json(std::basic_istream< typename Ptree::key_type::value_type > & stream,
                 Ptree & pt);
```

Description

Read JSON from a the given stream and translate it to a property tree.

Parameters: `pt` The property tree to populate.
`stream` Stream from which to read in the property tree.
Throws: json_parser_error
Notes: Clears existing contents of property tree. In case of error the property tree unmodified.

Items of JSON arrays are translated into ptree keys with empty names. Members of objects are translated into named keys.

JSON data can be a string, a numeric value, or one of literals "null", "true" and "false". During parse, any of the above is copied verbatim into ptree data string.

Function template read_json

boost::property_tree::json_parser::read_json

Synopsis

```
// In header: <boost/property_tree/json_parser.hpp>


template<typename Ptree>
  void read_json(const std::string & filename, Ptree & pt,
                 const std::locale & loc = std::locale());
```

Description

Read JSON from a the given file and translate it to a property tree.

Parameters:	`filename`	Name of file from which to read in the property tree.
	`loc`	The locale to use when reading in the file contents.
	`pt`	The property tree to populate.
Throws:	json_parser_error	
Notes:	Clears existing contents of property tree. In case of error the property tree unmodified.	
	Items of JSON arrays are translated into ptree keys with empty names. Members of objects are translated into named keys.	
	JSON data can be a string, a numeric value, or one of literals "null", "true" and "false". During parse, any of the above is copied verbatim into ptree data string.	

Function template write_json

boost::property_tree::json_parser::write_json

Synopsis

```
// In header: <boost/property_tree/json_parser.hpp>


template<typename Ptree>
  void write_json(std::basic_ostream< typename Ptree::key_type::value_type > & stream,
                  const Ptree & pt, bool pretty = true);
```

Description

Translates the property tree to JSON and writes it the given output stream.

Parameters:	`pretty`	Whether to pretty-print. Defaults to true for backward compatibility.
	`pt`	The property tree to tranlsate to JSON and output.
	`stream`	The stream to which to write the JSON representation of the property tree.
Requires:	*pt* cannot contain keys that have both subkeys and non-empty data.	
Throws:	json_parser_error	
Notes:	Any property tree key containing only unnamed subkeys will be rendered as JSON arrays.	

Function template write_json

boost::property_tree::json_parser::write_json

Synopsis

```
// In header: <boost/property_tree/json_parser.hpp>


template<typename Ptree>
  void write_json(const std::string & filename, const Ptree & pt,
                  const std::locale & loc = std::locale(),
                  bool pretty = true);
```

Description

Translates the property tree to JSON and writes it the given file.

Parameters:	`filename`	The name of the file to which to write the JSON representation of the property tree.
	`loc`	The locale to use when writing out to the output file.
	`pretty`	Whether to pretty-print. Defaults to true and last place for backward compatibility.
	`pt`	The property tree to translate to JSON and output.
Requires:	*pt* cannot contain keys that have both subkeys and non-empty data.	
Throws:	json_parser_error	
Notes:	Any property tree key containing only unnamed subkeys will be rendered as JSON arrays.	

Header <boost/property_tree/ptree.hpp>

```
namespace boost {
  namespace property_tree {
    template<typename Key, typename Data, typename KeyCompare>
      class basic_ptree;
  }
}
```

Class template basic_ptree

boost::property_tree::basic_ptree

Synopsis

```
// In header: <boost/property_tree/ptree.hpp>

template<typename Key, typename Data, typename KeyCompare>
class basic_ptree {
public:
  // types
  typedef basic_ptree< Key, Data, KeyCompare > self_type;
  typedef Key                                  key_type;
  typedef Data                                 data_type;
  typedef KeyCompare                           key_compare;
  typedef std::pair< const Key, self_type >    value_type;
  typedef std::size_t                          size_type;
  typedef path_of< Key >::type                 path_type;

  // construct/copy/destruct
  basic_ptree();
  basic_ptree(const data_type &);
  basic_ptree(const self_type &);
  basic_ptree& operator=(const self_type &);
  ~basic_ptree();

  // public member functions
  void swap(self_type &);
  size_type size() const;
  size_type max_size() const;
  bool empty() const;
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  value_type & front();
  const value_type & front() const;
  value_type & back();
  const value_type & back() const;
  iterator insert(iterator, const value_type &);
  template<typename It> void insert(iterator, It, It);
  iterator erase(iterator);
  iterator erase(iterator, iterator);
  iterator push_front(const value_type &);
  iterator push_back(const value_type &);
  void pop_front();
  void pop_back();
  void reverse();
  template<typename Compare> void sort(Compare);
  void sort();
  bool operator==(const self_type &) const;
  bool operator!=(const self_type &) const;
  assoc_iterator ordered_begin();
  const_assoc_iterator ordered_begin() const;
  assoc_iterator not_found();
  const_assoc_iterator not_found() const;
  assoc_iterator find(const key_type &);
```

```
  const_assoc_iterator find(const key_type &) const;
  std::pair< assoc_iterator, assoc_iterator > equal_range(const key_type &);
  std::pair< const_assoc_iterator, const_assoc_iterator >
  equal_range(const key_type &) const;
  size_type count(const key_type &) const;
  size_type erase(const key_type &);
  iterator to_iterator(assoc_iterator);
  const_iterator to_iterator(const_assoc_iterator) const;
  data_type & data();
  const data_type & data() const;
  void clear();
  self_type & get_child(const path_type &);
  const self_type & get_child(const path_type &) const;
  self_type & get_child(const path_type &, self_type &);
  const self_type & get_child(const path_type &, const self_type &) const;
  optional< self_type & > get_child_optional(const path_type &);
  optional< const self_type & > get_child_optional(const path_type &) const;
  self_type & put_child(const path_type &, const self_type &);
  self_type & add_child(const path_type &, const self_type &);
  template<typename Type, typename Translator>
    unspecified get_value(Translator) const;
  template<typename Type> Type get_value() const;
  template<typename Type, typename Translator>
    Type get_value(const Type &, Translator) const;
  template<typename Ch, typename Translator>
    unspecified get_value(const Ch *, Translator) const;
  template<typename Type> unspecified get_value(const Type &) const;
  template<typename Ch> unspecified get_value(const Ch *) const;
  template<typename Type, typename Translator>
    optional< Type > get_value_optional(Translator) const;
  template<typename Type> optional< Type > get_value_optional() const;
  template<typename Type, typename Translator>
    void put_value(const Type &, Translator);
  template<typename Type> void put_value(const Type &);
  template<typename Type, typename Translator>
    unspecified get(const path_type &, Translator) const;
  template<typename Type> Type get(const path_type &) const;
  template<typename Type, typename Translator>
    Type get(const path_type &, const Type &, Translator) const;
  template<typename Ch, typename Translator>
    unspecified get(const path_type &, const Ch *, Translator) const;
  template<typename Type>
    unspecified get(const path_type &, const Type &) const;
  template<typename Ch> unspecified get(const path_type &, const Ch *) const;
  template<typename Type, typename Translator>
    optional< Type > get_optional(const path_type &, Translator) const;
  template<typename Type>
    optional< Type > get_optional(const path_type &) const;
  template<typename Type, typename Translator>
    self_type & put(const path_type &, const Type &, Translator);
  template<typename Type> self_type & put(const path_type &, const Type &);
  template<typename Type, typename Translator>
    self_type & add(const path_type &, const Type &, Translator);
  template<typename Type> self_type & add(const path_type &, const Type &);

  // private member functions
  self_type * walk_path(path_type &) const;
  self_type & force_path(path_type &);
};
```

Description

Property tree main structure. A property tree is a hierarchical data structure which has one element of type `Data` in each node, as well as an ordered sequence of sub-nodes, which are additionally identified by a non-unique key of type `Key`.

Key equivalency is defined by `KeyCompare`, a predicate defining a strict weak ordering.

Property tree defines a Container-like interface to the (key-node) pairs of its direct sub-nodes. The iterators are bidirectional. The sequence of nodes is held in insertion order, not key order.

`basic_ptree` public types

1. typedef basic_ptree< Key, Data, KeyCompare > self_type;

   Simpler way to refer to this basic_ptree<C,K,P,A> type. Note that this is private, and made public only for doxygen.

`basic_ptree` public construct/copy/destruct

1. ```
   basic_ptree();
   ```

   Creates a node with no children and default-constructed data.

2. ```
   basic_ptree(const data_type & data);
   ```

   Creates a node with no children and a copy of the given data.

3. ```
   basic_ptree(const self_type & rhs);
   ```

4. ```
   basic_ptree& operator=(const self_type & rhs);
   ```

   Basic guarantee only.

5. ```
   ~basic_ptree();
   ```

`basic_ptree` public member functions

1. ```
   void swap(self_type & rhs);
   ```

   Swap with other tree. Only constant-time and nothrow if the data type's swap is.

2. ```
   size_type size() const;
   ```

   The number of direct children of this node.

3. ```
   size_type max_size() const;
   ```

4. ```
   bool empty() const;
   ```

   Whether there are any direct children.
   ```

5. ```
   iterator begin();
   ```

6. ```
   const_iterator begin() const;
   ```

7. ```
   iterator end();
   ```

8. ```
   const_iterator end() const;
   ```

9. ```
   reverse_iterator rbegin();
   ```

10. ```
    const_reverse_iterator rbegin() const;
    ```

11. ```
    reverse_iterator rend();
    ```

12. ```
    const_reverse_iterator rend() const;
    ```

13. ```
    value_type & front();
    ```

14. ```
    const value_type & front() const;
    ```

15. ```
    value_type & back();
    ```

16. ```
    const value_type & back() const;
    ```

17. ```
    iterator insert(iterator where, const value_type & value);
    ```

    Insert a copy of the given tree with its key just before the given position in this node. This operation invalidates no iterators.
    Returns: An iterator to the newly created child.

18. ```
    template<typename It> void insert(iterator where, It first, It last);
    ```

    Range insert. Equivalent to:

    ```
    for(; first != last; ++first) insert(where, *first);
    ```

19. ```
    iterator erase(iterator where);
    ```
    ```

Erase the child pointed at by the iterator. This operation invalidates the given iterator, as well as its equivalent assoc_iterator.
Returns: A valid iterator pointing to the element after the erased.

20.
```
iterator erase(iterator first, iterator last);
```

Range erase. Equivalent to:

```
 while(first != last;) first = erase(first);
```

21.
```
iterator push_front(const value_type & value);
```

Equivalent to insert(begin(), value).

22.
```
iterator push_back(const value_type & value);
```

Equivalent to insert(end(), value).

23.
```
void pop_front();
```

Equivalent to erase(begin()).

24.
```
void pop_back();
```

Equivalent to erase(boost::prior(end())).

25.
```
void reverse();
```

Reverses the order of direct children in the property tree.

26.
```
template<typename Compare> void sort(Compare comp);
```

Sorts the direct children of this node according to the predicate. The predicate is passed the whole pair of key and child.

27.
```
void sort();
```

Sorts the direct children of this node according to key order.

28.
```
bool operator==(const self_type & rhs) const;
```

Two property trees are the same if they have the same data, the keys and order of their children are the same, and the children compare equal, recursively.

29.
```
bool operator!=(const self_type & rhs) const;
```

30.
```
assoc_iterator ordered_begin();
```

Returns an iterator to the first child, in order.

31.
```
const_assoc_iterator ordered_begin() const;
```

Returns an iterator to the first child, in order.

32.
```
assoc_iterator not_found();
```

Returns the not-found iterator. Equivalent to end() in a real associative container.

33.
```
const_assoc_iterator not_found() const;
```

Returns the not-found iterator. Equivalent to end() in a real associative container.

34.
```
assoc_iterator find(const key_type & key);
```

Find a child with the given key, or not_found() if there is none. There is no guarantee about which child is returned if multiple have the same key.

35.
```
const_assoc_iterator find(const key_type & key) const;
```

Find a child with the given key, or not_found() if there is none. There is no guarantee about which child is returned if multiple have the same key.

36.
```
std::pair< assoc_iterator, assoc_iterator > equal_range(const key_type & key);
```

Find the range of children that have the given key.

37.
```
std::pair< const_assoc_iterator, const_assoc_iterator >
equal_range(const key_type & key) const;
```

Find the range of children that have the given key.

38.
```
size_type count(const key_type & key) const;
```

Count the number of direct children with the given key.

39.
```
size_type erase(const key_type & key);
```

Erase all direct children with the given key and return the count.

40.
```
iterator to_iterator(assoc_iterator it);
```

Get the iterator that points to the same element as the argument.
Notes: A valid assoc_iterator range (a, b) does not imply that (to_iterator(a), to_iterator(b)) is a valid range.

41.
```
const_iterator to_iterator(const_assoc_iterator it) const;
```

Get the iterator that points to the same element as the argument.
Notes: A valid const_assoc_iterator range (a, b) does not imply that (to_iterator(a), to_iterator(b)) is a valid range.

42.
```
data_type & data();
```

Reference to the actual data in this node.

43.
```
const data_type & data() const;
```

Reference to the actual data in this node.

44.
```
void clear();
```

Clear this tree completely, of both data and children.

45.
```
self_type & get_child(const path_type & path);
```

Get the child at the given path, or throw `ptree_bad_path`.

Notes: Depending on the path, the result at each level may not be completely determinate, i.e. if the same key appears multiple times, which child is chosen is not specified. This can lead to the path not being resolved even though there is a descendant with this path. Example:

```
a -> b -> c
  -> b
```

The path "a.b.c" will succeed if the resolution of "b" chooses the first such node, but fail if it chooses the second.

46.
```
const self_type & get_child(const path_type & path) const;
```

Get the child at the given path, or throw `ptree_bad_path`.

47.
```
self_type & get_child(const path_type & path, self_type & default_value);
```

Get the child at the given path, or return `default_value`.

48.
```
const self_type &
get_child(const path_type & path, const self_type & default_value) const;
```

Get the child at the given path, or return `default_value`.

49.
```
optional< self_type & > get_child_optional(const path_type & path);
```

Get the child at the given path, or return boost::null.

50.
```
optional< const self_type & > get_child_optional(const path_type & path) const;
```

Get the child at the given path, or return boost::null.

51.
```
self_type & put_child(const path_type & path, const self_type & value);
```

Set the node at the given path to the given value. Create any missing parents. If the node at the path already exists, replace it.

Returns: A reference to the inserted subtree.

Notes: Because of the way paths work, it is not generally guaranteed that a node newly created can be accessed using the same path.

If the path could refer to multiple nodes, it is unspecified which one gets replaced.

52.
```
self_type & add_child(const path_type & path, const self_type & value);
```

Add the node at the given path. Create any missing parents. If there already is a node at the path, add another one with the same key.
Parameters: `path` Path to the child. The last fragment must not have an index.
Returns: A reference to the inserted subtree.
Notes: Because of the way paths work, it is not generally guaranteed that a node newly created can be accessed using the same path.

53.
```
template<typename Type, typename Translator>
  unspecified get_value(Translator tr) const;
```

Take the value of this node and attempt to translate it to a `Type` object using the supplied translator.
Throws: ptree_bad_data

54.
```
template<typename Type> Type get_value() const;
```

Take the value of this node and attempt to translate it to a `Type` object using the default translator.
Throws: ptree_bad_data

55.
```
template<typename Type, typename Translator>
  Type get_value(const Type & default_value, Translator tr) const;
```

Take the value of this node and attempt to translate it to a `Type` object using the supplied translator. Return `default_value` if this fails.

56.
```
template<typename Ch, typename Translator>
  unspecified get_value(const Ch * default_value, Translator tr) const;
```

Make get_value do the right thing for string literals.

57.
```
template<typename Type>
  unspecified get_value(const Type & default_value) const;
```

Take the value of this node and attempt to translate it to a `Type` object using the default translator. Return `default_value` if this fails.

58.
```
template<typename Ch> unspecified get_value(const Ch * default_value) const;
```

Make get_value do the right thing for string literals.

59.
```
template<typename Type, typename Translator>
  optional< Type > get_value_optional(Translator tr) const;
```

Take the value of this node and attempt to translate it to a `Type` object using the supplied translator. Return boost::null if this fails.

60.
```
template<typename Type> optional< Type > get_value_optional() const;
```

Take the value of this node and attempt to translate it to a `Type` object using the default translator. Return boost::null if this fails.

61.
```
template<typename Type, typename Translator>
  void put_value(const Type & value, Translator tr);
```

Replace the value at this node with the given value, translated to the tree's data type using the supplied translator.
Throws: ptree_bad_data

62.
```
template<typename Type> void put_value(const Type & value);
```

Replace the value at this node with the given value, translated to the tree's data type using the default translator.
Throws: ptree_bad_data

63.
```
template<typename Type, typename Translator>
  unspecified get(const path_type & path, Translator tr) const;
```

Shorthand for get_child(path).get_value(tr).

64.
```
template<typename Type> Type get(const path_type & path) const;
```

Shorthand for get_child(path).get_value<Type>().

65.
```
template<typename Type, typename Translator>
  Type get(const path_type & path, const Type & default_value, Translator tr) const;
```

Shorthand for get_child(path, empty_ptree()) .get_value(default_value, tr). That is, return the translated value if possible, and the default value if the node doesn't exist or conversion fails.

66.
```
template<typename Ch, typename Translator>
  unspecified get(const path_type & path, const Ch * default_value,
                  Translator tr) const;
```

Make get do the right thing for string literals.

67.
```
template<typename Type>
  unspecified get(const path_type & path, const Type & default_value) const;
```

Shorthand for get_child(path, empty_ptree()) .get_value(default_value). That is, return the translated value if possible, and the default value if the node doesn't exist or conversion fails.

68.
```
template<typename Ch>
  unspecified get(const path_type & path, const Ch * default_value) const;
```

Make get do the right thing for string literals.

69.
```
template<typename Type, typename Translator>
  optional< Type > get_optional(const path_type & path, Translator tr) const;
```

Shorthand for:

```
if(optional\<self_type&\> node = get_child_optional(path))
  return node->get_value_optional(tr);
return boost::null;
```

That is, return the value if it exists and can be converted, or nil.

70.
```
template<typename Type>
  optional< Type > get_optional(const path_type & path) const;
```

Shorthand for:

```
 if(optional\<const self_type&\> node = get_child_optional(path))
   return node->get_value_optional();
 return boost::null;
```

71.
```
template<typename Type, typename Translator>
  self_type & put(const path_type & path, const Type & value, Translator tr);
```

Set the value of the node at the given path to the supplied value, translated to the tree's data type. If the node doesn't exist, it is created, including all its missing parents.
Returns: The node that had its value changed.
Throws: ptree_bad_data

72.
```
template<typename Type>
  self_type & put(const path_type & path, const Type & value);
```

Set the value of the node at the given path to the supplied value, translated to the tree's data type. If the node doesn't exist, it is created, including all its missing parents.
Returns: The node that had its value changed.
Throws: ptree_bad_data

73.
```
template<typename Type, typename Translator>
  self_type & add(const path_type & path, const Type & value, Translator tr);
```

If the node identified by the path does not exist, create it, including all its missing parents. If the node already exists, add a sibling with the same key. Set the newly created node's value to the given paremeter, translated with the supplied translator.
Parameters: `path` Path to the child. The last fragment must not have an index.
`tr` The translator to use.
`value` The value to add.
Returns: The node that was added.
Throws: ptree_bad_data

74.
```
template<typename Type>
  self_type & add(const path_type & path, const Type & value);
```

If the node identified by the path does not exist, create it, including all its missing parents. If the node already exists, add a sibling with the same key. Set the newly created node's value to the given paremeter, translated with the supplied translator.
Parameters: `path` Path to the child. The last fragment must not have an index.
`value` The value to add.
Returns: The node that was added.
Throws: ptree_bad_data

`basic_ptree` private member functions

1.
```
self_type * walk_path(path_type & p) const;
```

2.
```
self_type & force_path(path_type & p);
```

Header <boost/property_tree/ptree_fwd.hpp>

```
namespace boost {
  namespace property_tree {
    typedef string_path< std::string, id_translator< std::string > > path;
    typedef basic_ptree< std::string, std::string > ptree;
    typedef unspecified iptree;
    typedef string_path< std::wstring, id_translator< std::wstring > > wpath;
    typedef basic_ptree< std::wstring, std::wstring > wptree;
    typedef unspecified wiptree;
    template<typename K, typename D, typename C>
      void swap(basic_ptree< K, D, C > &, basic_ptree< K, D, C > &);
  }
}
```

Type definition path

path

Synopsis

```
// In header: <boost/property_tree/ptree_fwd.hpp>

typedef string_path< std::string, id_translator< std::string > > path;
```

Description

Implements a path using a std::string as the key.

Type definition ptree

ptree

Synopsis

```
// In header: <boost/property_tree/ptree_fwd.hpp>

typedef basic_ptree< std::string, std::string > ptree;
```

Description

A property tree with std::string for key and data, and default comparison.

Type definition iptree

iptree

Synopsis

```
// In header: <boost/property_tree/ptree_fwd.hpp>

typedef unspecified iptree;
```

Description

A property tree with std::string for key and data, and case-insensitive comparison.

Type definition wpath

wpath

Synopsis

```
// In header: <boost/property_tree/ptree_fwd.hpp>

typedef string_path< std::wstring, id_translator< std::wstring > > wpath;
```

Description

Implements a path using a std::wstring as the key.

Type definition wptree

wptree

Synopsis

```
// In header: <boost/property_tree/ptree_fwd.hpp>

typedef basic_ptree< std::wstring, std::wstring > wptree;
```

Description

A property tree with std::wstring for key and data, and default comparison.

Type definition wiptree

wiptree

Synopsis

```
// In header: <boost/property_tree/ptree_fwd.hpp>

typedef unspecified wiptree;
```

Description

A property tree with std::wstring for key and data, and case-insensitive comparison.

Function template swap

boost::property_tree::swap

Synopsis

```
// In header: <boost/property_tree/ptree_fwd.hpp>


template<typename K, typename D, typename C>
  void swap(basic_ptree< K, D, C > & pt1, basic_ptree< K, D, C > & pt2);
```

Description

Swap two property tree instances.

Header <boost/property_tree/ptree_serialization.hpp>

```
namespace boost {
  namespace property_tree {
    template<typename Archive, typename K, typename D, typename C>
      void save(Archive &, const basic_ptree< K, D, C > &, const unsigned int);
    template<typename Archive, typename K, typename D, typename C>
      void load(Archive &, basic_ptree< K, D, C > &, const unsigned int);
    template<typename Archive, typename K, typename D, typename C>
      void serialize(Archive &, basic_ptree< K, D, C > &, const unsigned int);
  }
}
```

Function template save

boost::property_tree::save

Synopsis

```
// In header: <boost/property_tree/ptree_serialization.hpp>


template<typename Archive, typename K, typename D, typename C>
  void save(Archive & ar, const basic_ptree< K, D, C > & t,
            const unsigned int file_version);
```

Description

Serialize the property tree to the given archive.

Parameters:	`ar`	The archive to which to save the serialized property tree. This archive should conform to the concept laid out by the Boost.Serialization library.
	`file_version`	file_version for the archive.
	`t`	The property tree to serialize.
Postconditions:	`ar` will contain the serialized form of `t`.	
Notes:	In addition to serializing to regular archives, this supports serializing to archives requiring name-value pairs, e.g. XML archives. However, the output format in the XML archive is not guaranteed to be the same as that when using the Boost.PropertyTree library's `boost::property_tree::xml_parser::write_xml`.	

Function template load

boost::property_tree::load

Synopsis

```
// In header: <boost/property_tree/ptree_serialization.hpp>


template<typename Archive, typename K, typename D, typename C>
  void load(Archive & ar, basic_ptree< K, D, C > & t,
            const unsigned int file_version);
```

Description

De-serialize the property tree to the given archive.

Parameters:	`ar`	The archive from which to load the serialized property tree. This archive should conform to the concept laid out by the Boost.Serialization library.
	`file_version`	file_version for the archive.
	`t`	The property tree to de-serialize.
Postconditions:	`t` will contain the de-serialized data from `ar`.	
Notes:	In addition to de-serializing from regular archives, this supports loading from archives requiring name-value pairs, e.g. XML archives. The format should be that used by boost::property_tree::save.	

Function template serialize

boost::property_tree::serialize

Synopsis

```
// In header: <boost/property_tree/ptree_serialization.hpp>


template<typename Archive, typename K, typename D, typename C>
  void serialize(Archive & ar, basic_ptree< K, D, C > & t,
                 const unsigned int file_version);
```

Description

Load or store the property tree using the given archive.

Parameters:	`ar`	The archive from which to load or save the serialized property tree. The type of this archive will determine whether saving or loading is performed.
	`file_version`	file_version for the archive.
	`t`	The property tree to load or save.

Header <boost/property_tree/stream_translator.hpp>

```
namespace boost {
  namespace property_tree {
    template<typename Ch, typename Traits, typename E,
             typename Enabler = void>
      struct customize_stream;

    template<typename Ch, typename Traits>
      struct customize_stream<Ch, Traits, Ch, void>;
    template<typename Ch, typename Traits, typename F>
      struct customize_stream<Ch, Traits, F, typename boost::enable_if< detail::is_inex↵
act< F > >::type>;
    template<typename Ch, typename Traits>
      struct customize_stream<Ch, Traits, bool, void>;
    template<typename Ch, typename Traits>
      struct customize_stream<Ch, Traits, signed char, void>;
    template<typename Ch, typename Traits>
      struct customize_stream<Ch, Traits, unsigned char, void>;

    template<typename Ch, typename Traits, typename Alloc, typename E>
      class stream_translator;

    template<typename Ch, typename Traits, typename Alloc, typename E>
      struct translator_between<std::basic_string< Ch, Traits, Alloc >, E>;
  }
}
```

Struct template customize_stream

boost::property_tree::customize_stream

Synopsis

```
// In header: <boost/property_tree/stream_translator.hpp>

template<typename Ch, typename Traits, typename E, typename Enabler = void>
struct customize_stream {

  // public static functions
  static void insert(std::basic_ostream< Ch, Traits > &, const E &);
  static void extract(std::basic_istream< Ch, Traits > &, E &);
};
```

Description

`customize_stream` public static functions

1. ```
   static void insert(std::basic_ostream< Ch, Traits > & s, const E & e);
   ```

2. ```
   static void extract(std::basic_istream< Ch, Traits > & s, E & e);
   ```

Struct template customize_stream<Ch, Traits, Ch, void>

boost::property_tree::customize_stream<Ch, Traits, Ch, void>

Synopsis

```
// In header: <boost/property_tree/stream_translator.hpp>

template<typename Ch, typename Traits>
struct customize_stream<Ch, Traits, Ch, void> {

  // public static functions
  static void insert(std::basic_ostream< Ch, Traits > &, Ch);
  static void extract(std::basic_istream< Ch, Traits > &, Ch &);
};
```

Description

`customize_stream` public static functions

1. ```
   static void insert(std::basic_ostream< Ch, Traits > & s, Ch e);
   ```

2. ```
   static void extract(std::basic_istream< Ch, Traits > & s, Ch & e);
   ```

Struct template customize_stream<Ch, Traits, F, typename boost::enable_if< detail::is_inexact< F > >::type>

boost::property_tree::customize_stream<Ch, Traits, F, typename boost::enable_if< detail::is_inexact< F > >::type>

Synopsis

```
// In header: <boost/property_tree/stream_translator.hpp>

template<typename Ch, typename Traits, typename F>
struct customize_stream<Ch, Traits, F, typename boost::enable_if< detail::is_inex↵
act< F > >::type> {

  // public static functions
  static void insert(std::basic_ostream< Ch, Traits > &, const F &);
  static void extract(std::basic_istream< Ch, Traits > &, F &);
};
```

Description

`customize_stream` public static functions

1. ```
   static void insert(std::basic_ostream< Ch, Traits > & s, const F & e);
   ```

2. ```
   static void extract(std::basic_istream< Ch, Traits > & s, F & e);
   ```

Struct template customize_stream<Ch, Traits, bool, void>

boost::property_tree::customize_stream<Ch, Traits, bool, void>

Synopsis

```
// In header: <boost/property_tree/stream_translator.hpp>

template<typename Ch, typename Traits>
struct customize_stream<Ch, Traits, bool, void> {

  // public static functions
  static void insert(std::basic_ostream< Ch, Traits > &, bool);
  static void extract(std::basic_istream< Ch, Traits > &, bool &);
};
```

Description

`customize_stream` public static functions

1. ```
   static void insert(std::basic_ostream< Ch, Traits > & s, bool e);
   ```

2. ```
   static void extract(std::basic_istream< Ch, Traits > & s, bool & e);
   ```

Struct template customize_stream<Ch, Traits, signed char, void>

boost::property_tree::customize_stream<Ch, Traits, signed char, void>

Synopsis

```
// In header: <boost/property_tree/stream_translator.hpp>

template<typename Ch, typename Traits>
struct customize_stream<Ch, Traits, signed char, void> {

  // public static functions
  static void insert(std::basic_ostream< Ch, Traits > &, signed char);
  static void extract(std::basic_istream< Ch, Traits > &, signed char &);
};
```

Description

`customize_stream` public static functions

1. ```
   static void insert(std::basic_ostream< Ch, Traits > & s, signed char e);
   ```

2. ```
   static void extract(std::basic_istream< Ch, Traits > & s, signed char & e);
   ```

Struct template customize_stream<Ch, Traits, unsigned char, void>

boost::property_tree::customize_stream<Ch, Traits, unsigned char, void>

Synopsis

```
// In header: <boost/property_tree/stream_translator.hpp>

template<typename Ch, typename Traits>
struct customize_stream<Ch, Traits, unsigned char, void> {

  // public static functions
  static void insert(std::basic_ostream< Ch, Traits > &, unsigned char);
  static void extract(std::basic_istream< Ch, Traits > &, unsigned char &);
};
```

Description

`customize_stream` public static functions

1. ```
   static void insert(std::basic_ostream< Ch, Traits > & s, unsigned char e);
   ```

2. ```
   static void extract(std::basic_istream< Ch, Traits > & s, unsigned char & e);
   ```

Class template stream_translator

boost::property_tree::stream_translator — Implementation of Translator that uses the stream overloads.

Synopsis

```
// In header: <boost/property_tree/stream_translator.hpp>

template<typename Ch, typename Traits, typename Alloc, typename E>
class stream_translator {
public:
  // types
  typedef std::basic_string< Ch, Traits, Alloc > internal_type;
  typedef E                                      external_type;

  // construct/copy/destruct
  stream_translator(std::locale = std::locale());

  // public member functions
  boost::optional< E > get_value(const internal_type &);
  boost::optional< internal_type > put_value(const E &);
};
```

Description

`stream_translator` public construct/copy/destruct

1. ```
   stream_translator(std::locale loc = std::locale());
   ```

`stream_translator` public member functions

1. ```
   boost::optional< E > get_value(const internal_type & v);
   ```

2. ```
   boost::optional< internal_type > put_value(const E & v);
   ```
   ```

Struct template translator_between<std::basic_string< Ch, Traits, Alloc >, E>

boost::property_tree::translator_between<std::basic_string< Ch, Traits, Alloc >, E>

Synopsis

```
// In header: <boost/property_tree/stream_translator.hpp>

template<typename Ch, typename Traits, typename Alloc, typename E>
struct translator_between<std::basic_string< Ch, Traits, Alloc >, E> {
  // types
  typedef stream_translator< Ch, Traits, Alloc, E > type;
};
```

Header <boost/property_tree/string_path.hpp>

```
namespace boost {
  namespace property_tree {
    template<typename String, typename Translator> class string_path;

    template<typename Ch, typename Traits, typename Alloc>
      struct path_of<std::basic_string< Ch, Traits, Alloc >>;
    template<typename String, typename Translator>
      string_path< String, Translator >
      operator/(string_path< String, Translator > p1,
                const string_path< String, Translator > & p2);
    template<typename String, typename Translator>
      string_path< String, Translator >
      operator/(string_path< String, Translator > p1,
                const typename String::value_type * p2);
    template<typename String, typename Translator>
      string_path< String, Translator >
      operator/(const typename String::value_type * p1,
                const string_path< String, Translator > & p2);
  }
}
```

Class template string_path

boost::property_tree::string_path — Default path class. A path is a sequence of values. Groups of values are separated by the separator value, which defaults to '.' cast to the sequence's value type. The group of values is then passed to the translator to get a key.

Synopsis

```
// In header: <boost/property_tree/string_path.hpp>

template<typename String, typename Translator>
class string_path {
public:
  // types
  typedef Translator::external_type key_type;
  typedef String::value_type        char_type;

  // construct/copy/destruct
  string_path(char_type = char_type('.'));
  string_path(const String &, char_type = char_type('.'),
              Translator = Translator());
  string_path(const char_type *, char_type = char_type('.'),
              Translator = Translator());
  string_path(const string_path &);
  string_path& operator=(const string_path &);

  // private member functions
   BOOST_STATIC_ASSERT((is_same< String, typename Translator::internal_type >::value));
  s_c_iter cstart() const;

  // public member functions
  key_type reduce();
  bool empty() const;
  bool single() const;
  std::string dump() const;
  string_path & operator/=(const string_path &);
};
```

Description

If instantiated with std::string and id_translator<std::string>, it accepts paths of the form "one.two.three.four".

`string_path` public construct/copy/destruct

1. ```
   string_path(char_type separator = char_type('.'));
   ```

   Create an empty path.

2. ```
   string_path(const String & value, char_type separator = char_type('.'),
               Translator tr = Translator());
   ```

   Create a path by parsing the given string.

   Parameters:

`separator`	The separator used in parsing. Defaults to '.'.
`tr`	The translator used by this path to convert the individual parts to keys.
`value`	A sequence, possibly with separators, that describes the path, e.g. "one.two.three".

3. ```
   string_path(const char_type * value, char_type separator = char_type('.'),
               Translator tr = Translator());
   ```
   ```

Create a path by parsing the given string.

Parameters:

`separator`	The separator used in parsing. Defaults to '.'.
`tr`	The translator used by this path to convert the individual parts to keys.
`value`	A zero-terminated array of values. Only use if zero- termination makes sense for your type, and your sequence supports construction from it. Intended for string literals.

4.
```
string_path(const string_path & o);
```

5.
```
string_path& operator=(const string_path & o);
```

`string_path` private member functions

1.
```
 BOOST_STATIC_ASSERT((is_same< String, typename Translator::internal_type >::value));
```

2.
```
s_c_iter cstart() const;
```

`string_path` public member functions

1.
```
key_type reduce();
```

Take a single element off the path at the front and return it.

2.
```
bool empty() const;
```

Test if the path is empty.

3.
```
bool single() const;
```

Test if the path contains a single element, i.e. no separators.

4.
```
std::string dump() const;
```

5.
```
string_path & operator/=(const string_path & o);
```

Append a second path to this one.

Requires: o's separator is the same as this one's, or o has no separators

Struct template path_of<std::basic_string< Ch, Traits, Alloc >>

boost::property_tree::path_of<std::basic_string< Ch, Traits, Alloc >>

Synopsis

```
// In header: <boost/property_tree/string_path.hpp>

template<typename Ch, typename Traits, typename Alloc>
struct path_of<std::basic_string< Ch, Traits, Alloc >> {
  // types
  typedef std::basic_string< Ch, Traits, Alloc >           _string;
  typedef string_path< _string, id_translator< _string > > type;
};
```

Header <boost/property_tree/xml_parser.hpp>

```
namespace boost {
  namespace property_tree {
    namespace xml_parser {
      template<typename Ptree>
        void read_xml(std::basic_istream< typename Ptree::key_type::value_type > &,
                      Ptree &, int = 0);
      template<typename Ptree>
        void read_xml(const std::string &, Ptree &, int = 0,
                      const std::locale & = std::locale());
      template<typename Ptree>
        void write_xml(std::basic_ostream< typename Ptree::key_type::value_type > &,
                       const Ptree &,
                       const xml_writer_settings< type↵
name Ptree::key_type::value_type > & = xml_writer_settings< type↵
name Ptree::key_type::value_type >());
      template<typename Ptree>
        void write_xml(const std::string &, const Ptree &,
                       const std::locale & = std::locale(),
                       const xml_writer_settings< type↵
name Ptree::key_type::value_type > & = xml_writer_settings< type↵
name Ptree::key_type::value_type >());
    }
  }
}
```

Function template read_xml

boost::property_tree::xml_parser::read_xml

Synopsis

```
// In header: <boost/property_tree/xml_parser.hpp>


template<typename Ptree>
  void read_xml(std::basic_istream< typename Ptree::key_type::value_type > & stream,
                Ptree & pt, int flags = 0);
```

Description

Reads XML from an input stream and translates it to property tree.

Parameters: `flags` Flags controlling the behaviour of the parser. The following flags are supported:

- `no_concat_text` -- Prevents concatenation of text nodes into datastring of property tree. Puts them in separate `<xmltext>` strings instead.
- `no_comments` -- Skip XML comments.
- `trim_whitespace` -- Trim leading and trailing whitespace from text, and collapse sequences of whitespace.

`pt` The property tree to populate.

`stream` Stream from which to read in the property tree.

Throws: xml_parser_error

Notes: Clears existing contents of property tree. In case of error the property tree unmodified.

XML attributes are placed under keys named `<xmlattr>`.

Function template read_xml

boost::property_tree::xml_parser::read_xml

Synopsis

```
// In header: <boost/property_tree/xml_parser.hpp>


template<typename Ptree>
  void read_xml(const std::string & filename, Ptree & pt, int flags = 0,
                const std::locale & loc = std::locale());
```

Description

Reads XML from a file using the given locale and translates it to property tree.

Parameters:

`filename`	The file from which to read in the property tree.
`flags`	Flags controlling the bahviour of the parser. The following flags are supported: • `no_concat_text` -- Prevents concatenation of text nodes into datastring of property tree. Puts them in separate `<xmltext>` strings instead. • `no_comments` -- Skip XML comments.
`loc`	The locale to use when reading in the file contents.
`pt`	The property tree to populate.

Throws: xml_parser_error

Notes: Clears existing contents of property tree. In case of error the property tree unmodified.

XML attributes are placed under keys named `<xmlattr>`.

Function template write_xml

boost::property_tree::xml_parser::write_xml

Synopsis

```
// In header: <boost/property_tree/xml_parser.hpp>


template<typename Ptree>
  void write_xml(std::basic_ostream< typename Ptree::key_type::value_type > & stream,
                 const Ptree & pt,
                 const xml_writer_settings< typename Ptree::key_type::value_type > & set↵
tings = xml_writer_settings< typename Ptree::key_type::value_type >());
```

Description

Translates the property tree to XML and writes it the given output stream.

Parameters:	`pt`	The property tree to tranlsate to XML and output.
	`settings`	The settings to use when writing out the property tree as XML.
	`stream`	The stream to which to write the XML representation of the property tree.
Throws:	xml_parser_error	

Function template write_xml

boost::property_tree::xml_parser::write_xml

Synopsis

```
// In header: <boost/property_tree/xml_parser.hpp>


template<typename Ptree>
  void write_xml(const std::string & filename, const Ptree & pt,
                 const std::locale & loc = std::locale(),
                 const xml_writer_settings< typename Ptree::key_type::value_type > & set↵
tings = xml_writer_settings< typename Ptree::key_type::value_type >());
```

Description

Translates the property tree to XML and writes it the given file.

Parameters:	`filename`	The file to which to write the XML representation of the property tree.
	`loc`	The locale to use when writing the output to file.
	`pt`	The property tree to tranlsate to XML and output.
	`settings`	The settings to use when writing out the property tree as XML.
Throws:	xml_parser_error	