pugixml 1.6 quick start guide

website – http://pugixml.org repository – http://github.com/zeux/pugixml

Table of Contents

Introduction

pugixml is a light-weight C++ XML processing library. It consists of a DOM-like interface with rich traversal/modification capabilities, an extremely fast XML parser which constructs the DOM tree from an XML file/buffer, and an XPath 1.0 implementation for complex data-driven tree queries. Full Unicode support is also available, with two Unicode interface variants and conversions between different Unicode encodings (which happen automatically during parsing/saving). The library is extremely portable and easy to integrate and use. pugixml is developed and maintained since 2006 and has many users. All code is distributed under the MIT license, making it completely free to use in both open-source and proprietary applications.

pugixml enables very fast, convenient and memory-efficient XML document processing. However, since pugixml has a DOM parser, it can’t process XML documents that do not fit in memory; also the parser is a non-validating one, so if you need DTD/Schema validation, the library is not for you.

This is the quick start guide for pugixml, which purpose is to enable you to start using the library quickly. Many important library features are either not described at all or only mentioned briefly; for more complete information you should read the complete manual.

NOTE No documentation is perfect; neither is this one. If you find errors or omissions, please don’t hesitate to submit an issue or open a pull request with a fix.

Installation

You can download the latest source distribution as an archive:

pugixml-1.6.zip (Windows line endings)/pugixml-1.6.tar.gz (Unix line endings)

The distribution contains library source, documentation (the guide you're reading now and the manual) and some code examples. After downloading the distribution, install pugixml by extracting all files from the compressed archive.

The complete pugixml source consists of three files - one source file, `pugixml.cpp`, and two header files, `pugixml.hpp` and `pugiconfig.hpp`. `pugixml.hpp` is the primary header which you need to include in order to use pugixml classes/functions. The rest of this guide assumes that `pugixml.hpp` is either in the current directory or in one of include directories of your projects, so that `#include "pugixml.hpp"` can find the header; however you can also use relative path (i.e. `#include`

`"../libs/pugixml/src/pugixml.hpp"`) or include directory-relative path (i.e. `#include <xml/thirdparty/pugixml/src/pugixml.hpp>`).

The easiest way to build pugixml is to compile the source file, `pugixml.cpp`, along with the existing library/executable. This process depends on the method of building your application; for example, if you're using Microsoft Visual Studio [1], Apple Xcode, Code::Blocks or any other IDE, just add `pugixml.cpp` to one of your projects. There are other building methods available, including building pugixml as a standalone static/shared library; read the manual for further information.

Document object model

pugixml stores XML data in DOM-like way: the entire XML document (both document structure and element data) is stored in memory as a tree. The tree can be loaded from character stream (file, string, C++ I/O stream), then traversed via special API or

XPath expressions. The whole tree is mutable: both node structure and node/attribute data can be changed at any time. Finally, the result of document transformations can be saved to a character stream (file, C++ I/O stream or custom transport).

The root of the tree is the document itself, which corresponds to C++ type `xml_document`. Document has one or more child nodes, which correspond to C++ type `xml_node`. Nodes have different types; depending on a type, a node can have a collection of child nodes, a collection of attributes, which correspond to C++ type `xml_attribute`, and some additional data (i.e. name).

The most common node types are:

- Document node (`node_document`) - this is the root of the tree, which consists of several child nodes. This node corresponds to `xml_document` class; note that `xml_document` is a sub-class of `xml_node`, so the entire node interface is also available.
- Element/tag node (`node_element`) - this is the most common type of node, which represents XML elements. Element nodes have a name, a collection of attributes and a collection of child

nodes (both of which may be empty). The attribute is a simple name/value pair.

- Plain character data nodes (`node_pcdata`) represent plain text in XML. PCDATA nodes have a value, but do not have name or children/attributes. Note that **plain character data is not a part of the element node but instead has its own node**; for example, an element node can have several child PCDATA nodes.

Despite the fact that there are several node types, there are only three C++ types representing the tree (`xml_document`, `xml_node`, `xml_attribute`); some operations on `xml_node` are only valid for certain node types. They are described below.

NOTE

> All pugixml classes and functions are located in `pugi` namespace; you have to either use explicit name qualification (i.e. `pugi::xml_node`), or to gain access to relevant symbols via `using` directive (i.e. `using pugi::xml_node;` or `using namespace pugi;`).

`xml_document` is the owner of the entire document structure; destroying the document destroys the whole tree. The interface of `xml_document` consists of loading functions, saving functions and the entire interface of `xml_node`, which allows for document inspection and/or modification. Note that while `xml_document` is a sub-class of `xml_node`, `xml_node` is not a polymorphic type; the inheritance is present only to simplify usage.

`xml_node` is the handle to document node; it can point to any node in the document, including document itself. There is a common interface for nodes of all types. Note that `xml_node` is only a handle to the actual node, not the node itself - you can have several `xml_node` handles pointing to the same underlying object. Destroying `xml_node` handle does not destroy the node and does not remove it from the tree.

There is a special value of `xml_node` type, known as null node or empty node. It does not correspond to any node in any document, and thus resembles null pointer. However, all operations are defined on empty nodes; generally the operations don’t do anything and return empty nodes/attributes or empty strings as their result. This is useful for chaining calls; i.e. you can get the grandparent of a node like so: `node.parent().parent()`; if a node is a null node or it does not have a parent, the first `parent()` call returns null node; the second `parent()` call then also returns null node, so you don't have to check for errors twice. You can test if a handle is null via implicit boolean cast: `if (node) { …□ }` or `if (!node) { …□ }`.

`xml_attribute` is the handle to an XML attribute; it has the same semantics as `xml_node`, i.e. there can be several `xml_attribute` handles pointing to the same underlying object and there is a special null attribute value, which propagates to function results.

There are two choices of interface and internal representation when configuring pugixml: you can either choose the UTF-8 (also called char) interface or UTF-16/32 (also called wchar_t) one. The choice is controlled via `PUGIXML_WCHAR_MODE` define; you can set it via `pugiconfig.hpp` or via preprocessor options. All tree functions that work with strings work with either C-style null terminated strings or STL strings of the selected character type.

Read the manual for additional information on Unicode interface.

Loading document

pugixml provides several functions for loading XML data from various places - files, C++ iostreams, memory buffers. All functions use an extremely fast non-validating parser. This parser is not fully W3C conformant - it can load any valid XML document, but does not perform some well-formedness checks. While considerable effort is made to reject invalid XML documents, some validation is not performed because of performance reasons. XML data is always converted to internal character format before parsing. pugixml supports all popular Unicode encodings (UTF-8, UTF-16 (big and little endian), UTF-32 (big and little endian); UCS-2 is naturally supported since it’s a strict subset of UTF-16) and handles all encoding conversions automatically.

The most common source of XML data is files; pugixml provides a separate function for loading XML document from file. This function accepts file path as its first argument, and also two optional arguments, which specify parsing options and input data encoding, which are described in the manual.

This is an example of loading XML document from file (samples/load_file.cpp):

```
pugi::xml_document doc;

pugi::xml_parse_result result =
doc.load_file("tree.xml");

std::cout << "Load result: " << result.description() <<
", mesh name: " <<
doc.child("mesh").attribute("name").value() << std::endl;
```

`load_file`, as well as other loading functions, destroys the existing document tree and then tries to load the new tree from the specified file. The result of the operation is returned in an `xml_parse_result` object; this object contains the operation status, and the related information (i.e. last successfully parsed position in the input file, if parsing fails).

Parsing result object can be implicitly converted to `bool`; if you do not want to handle parsing errors thoroughly, you can just check the return value of load functions as if it was a `bool`: `if (doc.load_file("file.xml")) { … } else { … }`. Otherwise you can use the `status` member to get parsing status, or the `description()` member function to get the status in a string form.

This is an example of handling loading errors (samples/load_error_handling.cpp):

```
pugi::xml_document doc;
pugi::xml_parse_result result = doc.load_string(source);

if (result)
{
    std::cout << "XML [" << source << "] parsed without
errors, attr value: [" <<
doc.child("node").attribute("attr").value() << "]\n\n";
}
else
{
    std::cout << "XML [" << source << "] parsed with
```

```
errors, attr value: [" <<
doc.child("node").attribute("attr").value() << "]\n";
    std::cout << "Error description: " <<
result.description() << "\n";
    std::cout << "Error offset: " << result.offset << "
(error at [..." << (source + result.offset) << "]\n\n";
}
```

Sometimes XML data should be loaded from some other source than file, i.e. HTTP URL; also you may want to load XML data from file using non-standard functions, i.e. to use your virtual file system facilities or to load XML from gzip-compressed files. These scenarios either require loading document from memory, in which case you should prepare a contiguous memory block with all XML data and to pass it to one of buffer loading functions, or loading document from C++ IOstream, in which case you should provide an object which implements `std::istream` or `std::wistream` interface.

There are different functions for loading document from memory; they treat the passed buffer as either an immutable one (`load_buffer`), a mutable buffer which is owned by the caller (`load_buffer_inplace`), or a mutable buffer which ownership belongs to pugixml (`load_buffer_inplace_own`). There is also a simple helper function, `xml_document::load`, for cases when you want to load the XML document from null-terminated character string.

This is an example of loading XML document from memory using one of these functions (samples/load_memory.cpp); read the sample code for more examples:

```
const char source[] = "<mesh name='sphere'><bounds>0 0 1
1</bounds></mesh>";
size_t size = sizeof(source);
```

```
// You can use load_buffer_inplace to load document from
mutable memory block; the block's lifetime must exceed
that of document
char* buffer = new char[size];
memcpy(buffer, source, size);

// The block can be allocated by any method; the block is
modified during parsing
pugi::xml_parse_result result =
doc.load_buffer_inplace(buffer, size);

// You have to destroy the block yourself after the
document is no longer used
delete[] buffer;
```

This is a simple example of loading XML document from file using streams (samples/load_stream.cpp); read the sample code for more complex examples involving wide streams and locales:

```
std::ifstream stream("weekly-utf-8.xml");
pugi::xml_parse_result result = doc.load(stream);
```

Accessing document data

pugixml features an extensive interface for getting various types of data from the document and for traversing the document. You can use various accessors to get node/attribute data, you can traverse the child node/attribute lists via accessors or iterators, you can do depth-first traversals with `xml_tree_walker` objects, and you can use XPath for complex data-driven queries.

You can get node or attribute name via `name()` accessor, and value via `value()` accessor. Note that both functions never return null pointers - they either return a string with the relevant content, or an empty string if name/value is absent or if the handle is null. Also there are two notable things for reading values:

- It is common to store data as text contents of some node - i.e. `<node><description>This is a node</description></node>`. In this case, `<description>` node does not have a value, but instead has a child of type `node_pcdata` with value `"This is a node"`. pugixml provides `child_value()` and `text()` helper functions to parse such data.
- In many cases attribute values have types that are not strings - i.e. an attribute may always contain values that should be treated as integers, despite the fact that they are represented as strings in XML. pugixml provides several accessors that convert

attribute value to some other type.

This is an example of using these functions (samples/traverse_base.cpp):

```
for (pugi::xml_node tool = tools.child("Tool"); tool;
tool = tool.next_sibling("Tool"))
{
    std::cout << "Tool " <<
tool.attribute("Filename").value();
    std::cout << ": AllowRemote " <<
tool.attribute("AllowRemote").as_bool();
    std::cout << ", Timeout " <<
tool.attribute("Timeout").as_int();
    std::cout << ", Description '" <<
tool.child_value("Description") << "'\n";
}
```

Since a lot of document traversal consists of finding the node/attribute with the correct name, there are special functions for that purpose. For example, `child("Tool")` returns the first node which has the name `"Tool"`, or null handle if there is no such node. This is an example of using such functions (samples/traverse_base.cpp):

```
std::cout << "Tool for *.dae generation: " <<
tools.find_child_by_attribute("Tool", "OutputFileMasks",
"*.dae").attribute("Filename").value() << "\n";

for (pugi::xml_node tool = tools.child("Tool"); tool;
tool = tool.next_sibling("Tool"))
{
    std::cout << "Tool " <<
tool.attribute("Filename").value() << "\n";
}
```

Child node lists and attribute lists are simply double-linked lists; while you can use `previous_sibling`/`next_sibling` and other such functions for iteration, pugixml additionally provides node and attribute iterators, so that you can treat nodes as containers of other nodes or attributes. All iterators are bidirectional and support all usual iterator operations. The iterators are invalidated if the node/attribute objects they're pointing to are removed from the tree; adding nodes/attributes does not invalidate any iterators.

Here is an example of using iterators for document traversal (samples/traverse_iter.cpp):

```
for (pugi::xml_node_iterator it = tools.begin(); it !=
tools.end(); ++it)
{
    std::cout << "Tool:";

    for (pugi::xml_attribute_iterator ait = it-
>attributes_begin(); ait != it->attributes_end(); ++ait)
    {
        std::cout << " " << ait->name() << "=" << ait-
>value();
    }

    std::cout << std::endl;
}
```

If your C++ compiler supports range-based for-loop (this is a C++11 feature, at the time of writing it’s supported by Microsoft Visual Studio 11 Beta, GCC 4.6 and Clang 3.0), you can use it to enumerate nodes/attributes. Additional helpers are provided to support this; note that they are also compatible with Boost Foreach, and possibly other pre-C++11 foreach facilities.

Here is an example of using C++11 range-based for loop for document traversal (samples/traverse_rangefor.cpp):

```
for (pugi::xml_node tool: tools.children("Tool"))
{
    std::cout << "Tool:";

    for (pugi::xml_attribute attr: tool.attributes())
    {
        std::cout << " " << attr.name() << "=" <<
attr.value();
    }

    for (pugi::xml_node child: tool.children())
    {
        std::cout << ", child " << child.name();
    }

    std::cout << std::endl;
}
```

The methods described above allow traversal of immediate children of some node; if you want to do a deep tree traversal, you'll have to do it via a recursive function or some equivalent method. However, pugixml provides a helper for depth-first traversal of a subtree. In order to use it, you have to implement `xml_tree_walker` interface and to call `traverse` function.

This is an example of traversing tree hierarchy with xml_tree_walker (samples/traverse_walker.cpp):

```
struct simple_walker: pugi::xml_tree_walker
{
    virtual bool for_each(pugi::xml_node& node)
    {
        for (int i = 0; i < depth(); ++i) std::cout << "
"; // indentation

        std::cout << node_types[node.type()] << ":
name='" << node.name() << "', value='" << node.value() <<
"'\n";

        return true; // continue traversal
    }
};
```

```
simple_walker walker;
```

```
doc.traverse(walker);
```

Finally, for complex queries often a higher-level DSL is needed. pugixml provides an implementation of XPath 1.0 language for such queries. The complete description of XPath usage can be found in the manual, but here are some examples:

```
pugi::xpath_node_set tools =
doc.select_nodes("/Profile/Tools/Tool[@AllowRemote='true'
and @DeriveCaptionFrom='lastparam']");

std::cout << "Tools:\n";

for (pugi::xpath_node_set::const_iterator it =
tools.begin(); it != tools.end(); ++it)
{
    pugi::xpath_node node = *it;
    std::cout <<
node.node().attribute("Filename").value() << "\n";
}

pugi::xpath_node build_tool =
doc.select_node("//Tool[contains(Description, 'build
system')]");
```

```
if (build_tool)
    std::cout << "Build tool: " <<
build_tool.node().attribute("Filename").value() << "\n";
```

CAUTION XPath functions throw `xpath_exception` objects on error; the sample above does not catch these exceptions.

Modifying document data

The document in pugixml is fully mutable: you can completely change the document structure and modify the data of nodes/attributes. All functions take care of memory management and structural integrity themselves, so they always result in structurally valid tree - however, it is possible to create an invalid XML tree (for example, by adding two attributes with the same name or by setting attribute/node name to empty/invalid string).

Tree modification is optimized for performance and for memory consumption, so if you have enough memory you can create documents from scratch with pugixml and later save them to file/stream instead of relying on error-prone manual text writing and without too much overhead.

All member functions that change node/attribute data or structure are non-constant and thus can not be called on constant handles. However, you can easily convert constant handle to non-constant one by simple assignment: `void foo(const pugi::xml_node& n) { pugi::xml_node nc = n; }`, so const-correctness here mainly provides additional documentation.

As discussed before, nodes can have name and value, both of which are strings. Depending on node type, name or value may be absent. You can use `set_name` and `set_value` member functions to set them. Similar functions are available for attributes; however, the `set_value` function is overloaded for some other types except strings, like floating-point numbers. Also, attribute value can be set using an assignment operator. This is an example of setting node/attribute name and value (samples/modify_base.cpp):

```
pugi::xml_node node = doc.child("node");

// change node name
std::cout << node.set_name("notnode");
std::cout << ", new node name: " << node.name() <<
std::endl;

// change comment text
std::cout << doc.last_child().set_value("useless
comment");
std::cout << ", new comment text: " <<
doc.last_child().value() << std::endl;

// we can't change value of the element or name of the
comment
std::cout << node.set_value("1") << ", " <<
doc.last_child().set_name("2") << std::endl;
```

```
pugi::xml_attribute attr = node.attribute("id");

// change attribute name/value
std::cout << attr.set_name("key") << ", " <<
attr.set_value("345");
```

```
std::cout << ", new attribute: " << attr.name() << "=" <<
attr.value() << std::endl;

// we can use numbers or booleans
attr.set_value(1.234);
std::cout << "new attribute value: " << attr.value() <<
std::endl;

// we can also use assignment operators for more concise
code
attr = true;
std::cout << "final attribute value: " << attr.value() <<
std::endl;
```

Nodes and attributes do not exist without a document tree, so you can’t create them without adding them to some document. A node or attribute can be created at the end of node/attribute list or before/after some other node. All insertion functions return the handle to newly created object on success, and null handle on failure. Even if the operation fails (for example, if you’re trying to add a child node to PCDATA node), the document remains in consistent state, but the requested node/attribute is not added.

CAUTION

`attribute()` and `child()` functions do not add attributes or nodes to the tree, so code like `node.attribute("id") = 123;` will not do anything if `node` does not have an attribute with name `"id"`. Make sure you're operating with existing attributes/nodes by adding them if necessary.

This is an example of adding new attributes/nodes to the document (samples/modify_add.cpp):

```
// add node with some name
pugi::xml_node node = doc.append_child("node");

// add description node with text child
pugi::xml_node descr = node.append_child("description");
descr.append_child(pugi::node_pcdata).set_value("Simple
node");

// add param node before the description
pugi::xml_node param = node.insert_child_before("param",
descr);
```

```
// add attributes to param node
param.append_attribute("name") = "version";
param.append_attribute("value") = 1.1;
param.insert_attribute_after("type",
param.attribute("name")) = "float";
```

If you do not want your document to contain some node or attribute, you can remove it with `remove_attribute` and `remove_child` functions. Removing the attribute or node invalidates all handles to the same underlying object, and also invalidates all iterators pointing to the same object. Removing node also invalidates all past-the-end iterators to its attribute or child node list. Be careful to ensure that all such handles and iterators either do not exist or are not used after the attribute/node is removed.

This is an example of removing attributes/nodes from the document (samples/modify_remove.cpp):

```
// remove description node with the whole subtree
pugi::xml_node node = doc.child("node");
node.remove_child("description");
```

```
// remove id attribute
pugi::xml_node param = node.child("param");
param.remove_attribute("value");

// we can also remove nodes/attributes by handles
pugi::xml_attribute id = param.attribute("name");
param.remove_attribute(id);
```

Saving document

Often after creating a new document or loading the existing one and processing it, it is necessary to save the result back to file. Also it is occasionally useful to output the whole document or a subtree to some stream; use cases include debug printing, serialization via network or other text-oriented medium, etc. pugixml provides several functions to output any subtree of the document to a file, stream or another generic transport interface; these functions allow to customize the output format, and also perform necessary encoding conversions.

Before writing to the destination the node/attribute data is properly formatted according to the node type; all special XML symbols, such as < and &, are properly escaped. In order to guard against forgotten node/attribute names, empty node/attribute names are printed as `":anonymous"`. For well-formed output, make sure all node and attribute names are set to meaningful values.

If you want to save the whole document to a file, you can use the `save_file` function, which returns `true` on success. This is a simple example of saving XML document to file (samples/save_file.cpp):

```
// save document to file
std::cout << "Saving result: " <<
doc.save_file("save_file_output.xml") << std::endl;
```

To enhance interoperability pugixml provides functions for saving document to any object which implements C++ `std::ostream` interface. This allows you to save documents to any standard C++ stream (i.e. file stream) or any third-party compliant implementation (i.e. Boost Iostreams). Most notably, this allows for easy debug output, since you can use `std::cout` stream as saving target. There are two functions, one works with narrow character streams, another handles wide character ones.

This is a simple example of saving XML document to standard output (samples/save_stream.cpp):

```
// save document to standard output
std::cout << "Document:\n";
doc.save(std::cout);
```

All of the above saving functions are implemented in terms of writer interface. This is a simple interface with a single function, which is called several times during output process with chunks of document data as input. In order to output the document via some custom transport, for example sockets, you should create an object which implements `xml_writer_file` interface and pass it to `xml_document::save` function.

This is a simple example of custom writer for saving document data to STL string (samples/save_custom_writer.cpp); read the sample code for more complex examples:

```
struct xml_string_writer: pugi::xml_writer
{
    std::string result;

    virtual void write(const void* data, size_t size)
    {
        result.append(static_cast<const char*>(data),
size);
    }
};
```

While the previously described functions save the whole document to the destination, it is easy to save a single subtree. Instead of calling `xml_document::save`, just call `xml_node::print` function on the target node. You can save node contents to C++ IOstream object or custom writer in this way. Saving a subtree slightly differs from saving the whole document; read the manual for more information.

Feedback

If you believe you’ve found a bug in pugixml, please file an issue via issue submission form. Be sure to include the relevant information so that the bug can be reproduced: the version of pugixml, compiler version and target architecture, the code that uses pugixml and exhibits the bug, etc. Feature requests and contributions can be filed as issues, too.

If filing an issue is not possible due to privacy or other concerns, you can contact pugixml author by e-mail directly: arseny.kapoulkine@gmail.com.

License

The pugixml library is distributed under the MIT license:

```
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
USE OR
OTHER DEALINGS IN THE SOFTWARE.
```

This means that you can freely use pugixml in your applications, both open-source and proprietary. If you use pugixml in a product, it is sufficient to add an acknowledgment like this to the product distribution:

```
This software is based on pugixml library
(http://pugixml.org).
pugixml is Copyright (C) 2006-2015 Arseny Kapoulkine.
```

1. All trademarks used are properties of their respective owners.