pugixml 1.6 manual

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

Table of Contents

1. Overview

1.1. 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 or XML Schema validation, the library is not for you.

This is the complete manual for pugixml, which describes all features of the library in detail. If you want to start writing code as quickly as possible, you are advised to read the quick start guide first.

> **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.

1.2. Feedback

If you believe you've found a bug in pugixml (bugs include compilation problems (errors/warnings), crashes, performance degradation and incorrect behavior), 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 can be reported the same way as bugs, so if you're missing some functionality in pugixml or if the API is rough in some places and you can suggest an improvement, file an issue. However please note that there are many factors when considering API changes (compatibility with previous versions, API redundancy, etc.), so generally features that can be implemented via a small function without pugixml modification are not accepted. However, all rules have exceptions.

If you have a contribution to pugixml, such as build script for some build system/IDE, or a well-designed set of helper functions, or a binding to some language other than C++, please file an issue or open a pull request. Your contribution has to be distributed

under the terms of a license that's compatible with pugixml license; i.e. GPL/LGPL licensed code is not accepted.

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.

1.3. Acknowledgments

pugixml could not be developed without the help from many people; some of them are listed in this section. If you've played a part in pugixml development and you can not find yourself on this list, I'm truly sorry; please send me an e-mail so I can fix this.

Thanks to **Kristen Wegner** for pugxml parser, which was used as a basis for pugixml.

Thanks to **Neville Franks** for contributions to pugxml parser.

Thanks to **Artyom Palvelev** for suggesting a lazy gap contraction approach.

Thanks to **Vyacheslav Egorov** for documentation proofreading.

1.4. License

The pugixml library is distributed under the MIT license:

```
Copyright (c) 2006-2015 Arseny Kapoulkine

Permission is hereby granted, free of charge, to any
person
obtaining a copy of this software and associated
documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to
use,
copy, modify, merge, publish, distribute, sublicense,
and/or sell
copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice
shall be
included in all copies or substantial portions of the
Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
COPYRIGHT
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.
```

2. Installation

2.1. Getting pugixml

pugixml is distributed in source form. You can either download a source distribution or clone the Git repository.

2.1.1. Source distributions

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 manual you're reading now and the quick start guide) and some code examples. After downloading the distribution, install pugixml by extracting all files from the compressed archive.

If you need an older version, you can download it from the version archive.

2.1.2. Git repository

The Git repository is located at https://github.com/zeux/pugixml/. There is a Git tag "v{version}" for each version; also there is the "latest" tag, which always points to the latest stable release.

For example, to checkout the current version, you can use this command:

```
git clone https://github.com/zeux/pugixml
cd pugixml
git checkout v1.6
```

The repository contains library source, documentation, code examples and full unit test suite.

Use `latest` tag if you want to automatically get new versions. Use other tags if you want to switch to new versions only explicitly. Also please note that the master branch contains the

work-in-progress version of the code; while this means that you can get new features and bug fixes from master without waiting for a new release, this also means that occasionally the code can be broken in some configurations.

2.1.3. Subversion repository

You can access the Git repository via Subversion using https://github.com/zeux/pugixml URL. For example, to checkout the current version, you can use this command:

```
svn checkout https://github.com/zeux/pugixml/tags/v1.6
pugixml
```

2.2. Building pugixml

pugixml is distributed in source form without any pre-built binaries; you have to build them yourself.

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; `pugiconfig.hpp` is a supplementary configuration file (see Additional configuration options). 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>`).

2.2.1. Building pugixml as a part of another static library/executable

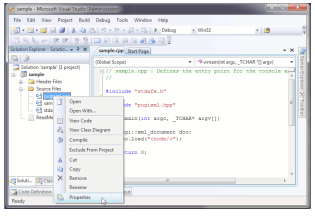
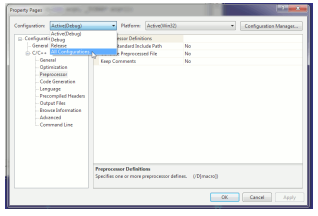
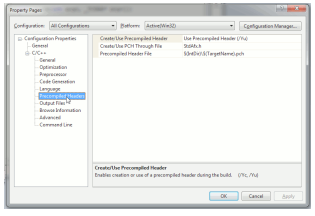
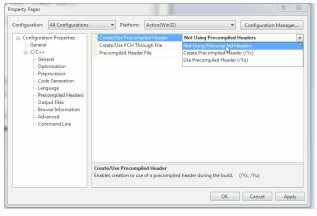
The easiest way to build pugixml is to compile the source file, `pugixml.cpp`, along with the existing library/executable. This process depends on the method of building your application; for example, if you're using Microsoft Visual Studio [1], Apple Xcode, Code::Blocks or any other IDE, just add `pugixml.cpp`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

If you're using Microsoft Visual Studio and the project has

precompiled headers turned on, you'll see the following error messages:

```
pugixml.cpp(3477) : fatal error C1010: unexpected end of
file while looking for precompiled header. Did you forget
to add '#include "stdafx.h"' to your source?
```

The correct way to resolve this is to disable precompiled headers for `pugixml.cpp`; you have to set "Create/Use Precompiled Header" option (Properties dialog → C/C++ → Precompiled Headers → Create/Use Precompiled Header) to "Not Using Precompiled Headers". You'll have to do it for all project configurations/platforms (you can select Configuration "All Configurations" and Platform "All Platforms" before editing the option):

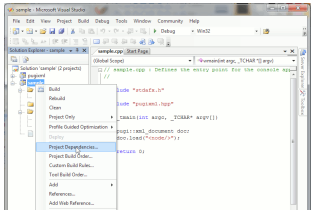
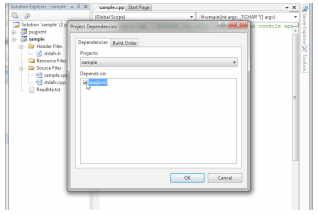
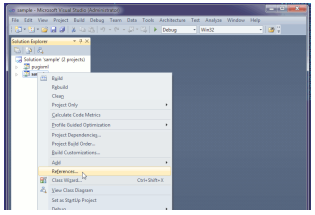
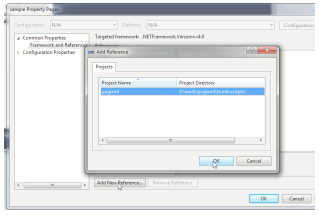





2.2.2. Building pugixml as a standalone static library

It's possible to compile pugixml as a standalone static library. This process depends on the method of building your application; pugixml distribution comes with project files for several popular IDEs/build systems. There are project files for Apple XCode, Code::Blocks, Codelite, Microsoft Visual Studio 2005, 2008, 2010+, and configuration scripts for CMake and premake4. You're welcome to submit project files/build scripts for other software; see Feedback.

There are two projects for each version of Microsoft Visual Studio: one for dynamically linked CRT, which has a name like `pugixml_vs2008.vcproj`, and another one for statically linked CRT, which has a name like `pugixml_vs2008_static.vcproj`. You should select the version that matches the CRT used in your application; the default option for new projects created by Microsoft Visual Studio is dynamically linked CRT, so unless you changed the defaults, you should use the version with dynamic CRT (i.e. `pugixml_vs2008.vcproj` for Microsoft Visual Studio 2008).

In addition to adding pugixml project to your workspace, you'll have to make sure that your application links with pugixml library. If you're using Microsoft Visual Studio 2005/2008, you can add a dependency from your application project to pugixml one. If you're using Microsoft Visual Studio 2010+, you'll have to add a reference to your application project instead. For other IDEs/systems, consult the relevant documentation.

Microsoft Visual Studio 2005/2008		Microsoft Visual Studio 2010+	
			

2.2.3. Building pugixml as a standalone shared library

It's possible to compile pugixml as a standalone shared library. The process is usually similar to the static library approach; however, no preconfigured projects/scripts are included into

pugixml distribution, so you'll have to do it yourself. Generally, if you're using GCC-based toolchain, the process does not differ from building any other library as DLL (adding -shared to compilation flags should suffice); if you're using MSVC-based toolchain, you'll have to explicitly mark exported symbols with a declspec attribute. You can do it by defining PUGIXML_API macro, i.e. via `pugiconfig.hpp`:

```
#ifdef _DLL
    #define PUGIXML_API __declspec(dllexport)
#else
    #define PUGIXML_API __declspec(dllimport)
#endif
```

CAUTION

If you're using STL-related functions, you should use the shared runtime library to ensure that a single heap is used for STL allocations in your application and in pugixml; in MSVC, this means selecting the 'Multithreaded DLL' or 'Multithreaded Debug DLL' to 'Runtime library' property (`/MD` or `/MDd` linker switch). You

> should also make sure that your runtime library choice is consistent between different projects.

2.2.4. Using pugixml in header-only mode

It's possible to use pugixml in header-only mode. This means that all source code for pugixml will be included in every translation unit that includes `pugixml.hpp`. This is how most of Boost and STL libraries work.

Note that there are advantages and drawbacks of this approach. Header mode may improve tree traversal/modification performance (because many simple functions will be inlined), if your compiler toolchain does not support link-time optimization, or if you have it turned off (with link-time optimization the performance should be similar to non-header mode). However, since compiler now has to compile pugixml source once for each translation unit that includes it, compilation times may increase noticeably. If you want to use pugixml in header mode but do not need XPath support, you can consider disabling it by using PUGIXML_NO_XPATH define to improve compilation time.

To enable header-only mode, you have to define `PUGIXML_HEADER_ONLY`. You can either do it in `pugiconfig.hpp`, or provide them via compiler command-line.

Note that it is safe to compile `pugixml.cpp` if `PUGIXML_HEADER_ONLY` is defined - so if you want to i.e. use header-only mode only in Release configuration, youcan include pugixml.cpp in your project (see Building pugixml as a part of another static library/executable), and conditionally enable header-only mode in `pugiconfig.hpp` like this:

```
#ifndef _DEBUG
    #define PUGIXML_HEADER_ONLY
#endif
```

2.2.5. Additional configuration options

pugixml uses several defines to control the compilation process. There are two ways to define them: either put the needed definitions to `pugiconfig.hpp` (it has some examples that are commented out) or provide them via compiler command-line. Consistency is important: the definitions should match in all

source files that include `pugixml.hpp` (including pugixml sources) throughout the application. Adding defines to `pugiconfig.hpp` lets you guarantee this, unless your macro definition is wrapped in preprocessor `#if`/`#ifdef` directive and this directive is not consistent. `pugiconfig.hpp` will never contain anything but comments, which means that when upgrading to a new version, you can safely leave your modified version intact.

`PUGIXML_WCHAR_MODE` define toggles between UTF-8 style interface (the in-memory text encoding is assumed to be UTF-8, most functions use `char` as character type) and UTF-16/32 style interface (the in-memory text encoding is assumed to be UTF-16/32, depending on `wchar_t` size, most functions use `wchar_t` as character type). See Unicode interface for more details.

`PUGIXML_NO_XPATH` define disables XPath. Both XPath interfaces and XPath implementation are excluded from compilation. This option is provided in case you do not need XPath functionality and need to save code space.

`PUGIXML_NO_STL` define disables use of STL in pugixml. The functions that operate on STL types are no longer present (i.e. load/save via iostream) if this macro is defined. This option is provided in case your target platform does not have a standard-compliant STL implementation.

`PUGIXML_NO_EXCEPTIONS` define disables use of exceptions in pugixml. This option is provided in case your target platform does not have exception handling capabilities.

`PUGIXML_API`, `PUGIXML_CLASS` and `PUGIXML_FUNCTION` defines let you specify custom attributes (i.e. declspec or calling conventions) for pugixml classes and non-member functions. In absence of `PUGIXML_CLASS` or `PUGIXML_FUNCTION` definitions, `PUGIXML_API` definition is used instead. For example, to specify fixed calling convention, you can define `PUGIXML_FUNCTION` to i.e. `__fastcall`. Another example is DLL import/export attributes in MSVC (see Building pugixml as a standalone shared library).

> **NOTE** In that example `PUGIXML_API` is inconsistent between several source files; this is an exception to

> the consistency rule.

`PUGIXML_MEMORY_PAGE_SIZE`, `PUGIXML_MEMORY_OUTPUT_STACK` and `PUGIXML_MEMORY_XPATH_PAGE_SIZE` can be used to customize certain important sizes to optimize memory usage for the application-specific patterns. For details see Memory consumption tuning.

`PUGIXML_HAS_LONG_LONG` define enables support for `long long` type in pugixml. This define is automatically enabled if your platform is known to have `long long` support (i.e. has C++11 support or uses a reasonably modern version of a known compiler); if pugixml does not recognize that your platform supports `long long` but in fact it does, you can enable the define manually.

2.3. Portability

pugixml is written in standard-compliant C++ with some compiler-specific workarounds where appropriate. pugixml is compatible with the C++11 standard, but does not require C++11 support. Each

version is tested with a unit test suite (with code coverage about 99%) on the following platforms:

- Microsoft Windows:
  - Borland C++ Compiler 5.82
  - Digital Mars C++ Compiler 8.51
  - Intel C++ Compiler 8.0, 9.0 x86/x64, 10.0 x86/x64, 11.0 x86/x64
  - Metrowerks CodeWarrior 8.0
  - Microsoft Visual C++ 6.0, 7.0 (2002), 7.1 (2003), 8.0 (2005) x86/x64, 9.0 (2008) x86/x64, 10.0 (2010) x86/x64, 11.0 (2011) x86/x64/ARM, 12.0 (2013) x86/x64/ARM and some CLR versions
  - MinGW (GCC) 3.4, 4.4, 4.5, 4.6 x64
- Linux (GCC 4.4.3 x86/x64, GCC 4.8.1 x64, Clang 3.2 x64)
- FreeBSD (GCC 4.2.1 x86/x64)
- Apple MacOSX (GCC 4.0.1 x86/x64/PowerPC, Clang 3.5 x64)
- Sun Solaris (sunCC x86/x64)

- Microsoft Xbox 360
- Nintendo Wii (Metrowerks CodeWarrior 4.1)
- Sony Playstation Portable (GCC 3.4.2)
- Sony Playstation 3 (GCC 4.1.1, SNC 310.1)
- Various portable platforms (Android NDK, BlackBerry NDK, Samsung bada, Windows CE)

---

3. 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 a character stream (file, string, C++ I/O stream), then traversed with the 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).

3.1. Tree structure

The XML document is represented with a tree data structure. 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 tree nodes can be of one of the following types (which together form the enumeration `xml_node_type`):

- 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. However, document node is special in several ways, which are covered below. There can be only one document node in the tree; document node does not have any XML representation.
- 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. The example XML representation of element nodes is as follows:

```
<node attr="value"><child/></node>
```

There are two element nodes here: one has name `"node"`, single attribute `"attr"` and single child `"child"`, another has name `"child"` and does not have any attributes or child nodes.

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

```
<node> text1 <child/> text2 </node>
```

Here `"node"` element has three children, two of which are PCDATA nodes with values `" text1 "` and `" text2 "`.

- Character data nodes (`node_cdata`) represent text in XML that is quoted in a special way. CDATA nodes do not differ from PCDATA nodes except in XML representation - the above text example looks like this with CDATA:

```
<node> <![CDATA[text1]]> <child/> <![CDATA[text2]]>
</node>
```

CDATA nodes make it easy to include non-escaped `<`, `&` and `>` characters in plain text. CDATA value can not contain the character sequence `]]>`, since it is used to determine the end of node contents.

- Comment nodes (`node_comment`) represent comments in XML. Comment nodes have a value, but do not have a name or children/attributes. The example XML representation of a comment node is as follows:

```
<!-- comment text -->
```

Here the comment node has value `"comment text"`. By default comment nodes are treated as non-essential part of XML markup and are not loaded during XML parsing. You can override this behavior with parse_comments flag.

- Processing instruction node (`node_pi`) represent processing instructions (PI) in XML. PI nodes have a name and an optional value, but do not have children/attributes. The example XML representation of a PI node is as follows:

```
<?name value?>
```

Here the name (also called PI target) is `"name"`, and the value is `"value"`. By default PI nodes are treated as non-essential part of XML markup and are not loaded during XML parsing. You can override this behavior with parse_pi flag.

- Declaration node (`node_declaration`) represents document declarations in XML. Declaration nodes have a name (`"xml"`) and an optional collection of attributes, but do not have value or children. There can be only one declaration node in a document; moreover, it should be the topmost node (its parent

should be the document). The example XML representation of a declaration node is as follows:

```
<?xml version="1.0"?>
```

Here the node has name `"xml"` and a single attribute with name `"version"` and value `"1.0"`. By default declaration nodes are treated as non-essential part of XML markup and are not loaded during XML parsing. You can override this behavior with parse_declaration flag. Also, by default a dummy declaration is output when XML document is saved unless there is already a declaration in the document; you can disable this with format_no_declaration flag.

- Document type declaration node (`node_doctype`) represents document type declarations in XML. Document type declaration nodes have a value, which corresponds to the entire document type contents; no additional nodes are created for inner elements like `<!ENTITY>`. There can be only one document type declaration node in a document; moreover, it should be the topmost node (its parent should be the document). The

example XML representation of a document type declaration node is as follows:

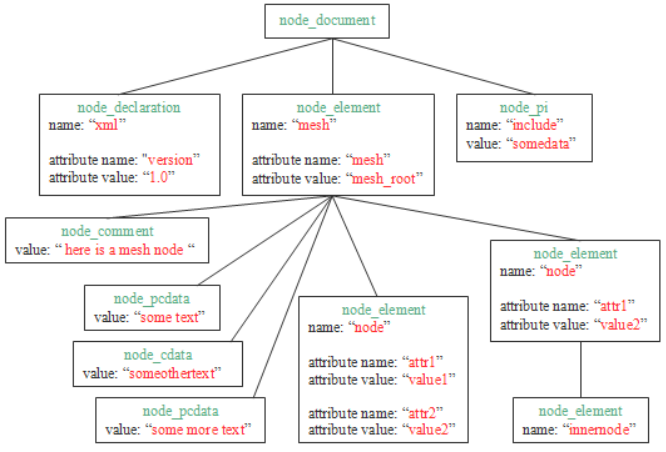
```
<!DOCTYPE greeting [ <!ELEMENT greeting (#PCDATA)> ]>
```

Here the node has value `"greeting [ <!ELEMENT greeting (#PCDATA)> ]"`. By default document type declaration nodes are treated as non-essential part of XML markup and are not loaded during XML parsing. You can override this behavior with parse_doctype flag.

Finally, here is a complete example of XML document and the corresponding tree representation (samples/tree.xml):

```
<?xml version="1.0"?>
<mesh name="mesh_root">
    <!-- here is a mesh
node -->
    some text

<![CDATA[someothertext]]>
    some more text
```



```
    <node attr1="value1"
attr2="value2" />
    <node attr1="value2">
        <innernode/>
    </node>
</mesh>
<?include somedata?>
```

3.2. C++ interface

> **NOTE**
>
> All pugixml classes and functions are located in the `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;`). The namespace will be omitted from all declarations in this documentation hereafter; all code examples will use fully qualified names.

Despite the fact that there are several node types, there are only

three C++ classes representing the tree (`xml_document`, `xml_node`, `xml_attribute`); some operations on `xml_node` are only valid for certain node types. The classes are described below.

`xml_document` is the owner of the entire document structure; it is a non-copyable class. The interface of `xml_document` consists of loading functions (see Loading document), saving functions (see Saving document) 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. Alternatively you can use the `document_element` function to get the element node that's the immediate child of the document.

Default constructor of `xml_document` initializes the document to the tree with only a root node (document node). You can then populate it with data using either tree modification functions or loading functions; all loading functions destroy the previous tree with all occupied memory, which puts existing node/attribute handles for this document to invalid state. If you want to destroy

the previous tree, you can use the `xml_document::reset` function; it destroys the tree and replaces it with either an empty one or a copy of the specified document. Destructor of `xml_document` also destroys the tree, thus the lifetime of the document object should exceed the lifetimes of any node/attribute handles that point to the tree.

> **CAUTION**
>
> While technically node/attribute handles can be alive when the tree they're referring to is destroyed, calling any member function for these handles results in undefined behavior. Thus it is recommended to make sure that the document is destroyed only after all references to its nodes/attributes are destroyed.

`xml_node` is the handle to document node; it can point to any node in the document, including the document node itself. There is a common interface for nodes of all types; the actual node type can be queried via the `xml_node::type()` method. 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. The size of `xml_node` is equal to that of a pointer, so it is nothing more than a lightweight wrapper around a pointer; you can safely pass or return `xml_node` objects by value without additional overhead.

There is a special value of `xml_node` type, known as null node or empty node (such nodes have type `node_null`). 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 (see documentation for specific functions for more detailed information). 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, which makes error handling easier.

`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.

Both `xml_node` and `xml_attribute` have the default constructor which initializes them to null objects.

`xml_node` and `xml_attribute` try to behave like pointers, that is, they can be compared with other objects of the same type, making it possible to use them as keys in associative containers. All handles to the same underlying object are equal, and any two handles to different underlying objects are not equal. Null handles only compare as equal to themselves. The result of relational comparison can not be reliably determined from the order of nodes in file or in any other way. Do not use relational comparison operators except for search optimization (i.e. associative container keys).

If you want to use `xml_node` or `xml_attribute` objects as keys in hash-based associative containers, you can use the `hash_value` member functions. They return the hash values that are guaranteed to be the same for all handles to the same

underlying object. The hash value for null handles is 0.

Finally handles can be implicitly cast to boolean-like objects, so that you can test if the node/attribute is empty with the following code: `if (node) { … }` or `if (!node) { … } else { … }`. Alternatively you can check if a given `xml_node` / `xml_attribute` handle is null by calling the following methods:

```
bool xml_attribute::empty() const;
bool xml_node::empty() const;
```

Nodes and attributes do not exist without a document tree, so you can't create them without adding them to some document. Once underlying node/attribute objects are destroyed, the handles to those objects become invalid. While this means that destruction of the entire tree invalidates all node/attribute handles, it also means that destroying a subtree (by calling xml_node::remove_child) or removing an attribute invalidates the corresponding handles. There is no way to check handle validity; you have to ensure correctness through external mechanisms.

3.3. Unicode interface

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, as discussed in Additional configuration options. If this define is set, the wchar_t interface is used; otherwise (by default) the char interface is used. The exact wide character encoding is assumed to be either UTF-16 or UTF-32 and is determined based on the size of `wchar_t` type.

> **NOTE** If the size of `wchar_t` is 2, pugixml assumes UTF-16 encoding instead of UCS-2, which means that some characters are represented as two code points.

All tree functions that work with strings work with either C-style null terminated strings or STL strings of the selected character type. For example, node name accessors look like this in char mode:

```
const char* xml_node::name() const;
bool xml_node::set_name(const char* value);
```

and like this in wchar_t mode:

```
const wchar_t* xml_node::name() const;
bool xml_node::set_name(const wchar_t* value);
```

There is a special type, `pugi::char_t`, that is defined as the character type and depends on the library configuration; it will be also used in the documentation hereafter. There is also a type `pugi::string_t`, which is defined as the STL string of the character type; it corresponds to `std::string` in char mode and to `std::wstring` in wchar_t mode.

In addition to the interface, the internal implementation changes to store XML data as `pugi::char_t`; this means that these two modes have different memory usage characteristics. The conversion to `pugi::char_t` upon document loading and from `pugi::char_t` upon document saving happen automatically,

which also carries minor performance penalty. The general advice however is to select the character mode based on usage scenario, i.e. if UTF-8 is inconvenient to process and most of your XML data is non-ASCII, wchar_t mode is probably a better choice.

There are cases when you'll have to convert string data between UTF-8 and wchar_t encodings; the following helper functions are provided for such purposes:

```
std::string as_utf8(const wchar_t* str);
std::wstring as_wide(const char* str);
```

Both functions accept a null-terminated string as an argument `str`, and return the converted string. `as_utf8` performs conversion from UTF-16/32 to UTF-8; `as_wide` performs conversion from UTF-8 to UTF-16/32. Invalid UTF sequences are silently discarded upon conversion. `str` has to be a valid string; passing null pointer results in undefined behavior. There are also two overloads with the same semantics which accept a string as an argument:

```
std::string as_utf8(const std::wstring& str);
std::wstring as_wide(const std::string& str);
```

NOTE

Most examples in this documentation assume char interface and therefore will not compile with PUGIXML_WCHAR_MODE. This is done to simplify the documentation; usually the only changes you'll have to make is to pass `wchar_t` string literals, i.e. instead of

```
xml_node node = doc.child("bookstore").find_child_by_attribute("book", "id", "12345");
```

you'll have to use

```
xml_node node = doc.child(L"bookstore").find_child_by_attribute(L"book", L"id", L"12345");
```

3.4. Thread-safety guarantees

Almost all functions in pugixml have the following thread-safety

guarantees:

- it is safe to call free (non-member) functions from multiple threads
- it is safe to perform concurrent read-only accesses to the same tree (all constant member functions do not modify the tree)
- it is safe to perform concurrent read/write accesses, if there is only one read or write access to the single tree at a time

Concurrent modification and traversing of a single tree requires synchronization, for example via reader-writer lock. Modification includes altering document structure and altering individual node/attribute data, i.e. changing names/values.

The only exception is set_memory_management_functions; it modifies global variables and as such is not thread-safe. Its usage policy has more restrictions, see Custom memory allocation/deallocation functions.

3.5. Exception guarantees

With the exception of XPath, pugixml itself does not throw any exceptions. Additionally, most pugixml functions have a no-throw exception guarantee.

This is not applicable to functions that operate on STL strings or IOstreams; such functions have either strong guarantee (functions that operate on strings) or basic guarantee (functions that operate on streams). Also functions that call user-defined callbacks (i.e. xml_node::traverse or xml_node::find_node) do not provide any exception guarantees beyond the ones provided by the callback.

If exception handling is not disabled with PUGIXML_NO_EXCEPTIONS define, XPath functions may throw xpath_exception on parsing errors; also, XPath functions may throw `std::bad_alloc` in low memory conditions. Still, XPath functions provide strong exception guarantee.

3.6. Memory management

pugixml requests the memory needed for document storage in big chunks, and allocates document data inside those chunks. This section discusses replacing functions used for chunk allocation

and internal memory management implementation.

3.6.1. Custom memory allocation/deallocation functions

All memory for tree structure, tree data and XPath objects is allocated via globally specified functions, which default to malloc/free. You can set your own allocation functions with set_memory_management function. The function interfaces are the same as that of malloc/free:

```
typedef void* (*allocation_function)(size_t size);
typedef void (*deallocation_function)(void* ptr);
```

You can use the following accessor functions to change or get current memory management functions:

```
void set_memory_management_functions(allocation_function
allocate, deallocation_function deallocate);
allocation_function get_memory_allocation_function();
deallocation_function get_memory_deallocation_function();
```

Allocation function is called with the size (in bytes) as an argument and should return a pointer to a memory block with alignment that is suitable for storage of primitive types (usually a maximum of `void*` and `double` types alignment is sufficient) and size that is greater than or equal to the requested one. If the allocation fails, the function has to return null pointer (throwing an exception from allocation function results in undefined behavior).

Deallocation function is called with the pointer that was returned by some call to allocation function; it is never called with a null pointer. If memory management functions are not thread-safe, library thread safety is not guaranteed.

This is a simple example of custom memory management (samples/custom_memory_management.cpp):

```
void* custom_allocate(size_t size)
{
    return new (std::nothrow) char[size];
}
```

```
void custom_deallocate(void* ptr)
{
    delete[] static_cast<char*>(ptr);
}
```

```
pugi::set_memory_management_functions(custom_allocate,
custom_deallocate);
```

When setting new memory management functions, care must be taken to make sure that there are no live pugixml objects. Otherwise when the objects are destroyed, the new deallocation function will be called with the memory obtained by the old allocation function, resulting in undefined behavior.

3.6.2. Memory consumption tuning

There are several important buffering optimizations in pugixml that rely on predefined constants. These constants have default values that were tuned for common usage patterns; for some applications, changing these constants might improve memory consumption or increase performance. Changing these constants is not recommended unless their default values result in visible

problems.

These constants can be tuned via configuration defines, as discussed in Additional configuration options; it is recommended to set them in `pugiconfig.hpp`.

- `PUGIXML_MEMORY_PAGE_SIZE` controls the page size for document memory allocation. Memory for node/attribute objects is allocated in pages of the specified size. The default size is 32 Kb; for some applications the size is too large (i.e. embedded systems with little heap space or applications that keep lots of XML documents in memory). A minimum size of 1 Kb is recommended.
- `PUGIXML_MEMORY_OUTPUT_STACK` controls the cumulative stack space required to output the node. Any output operation (i.e. saving a subtree to file) uses an internal buffering scheme for performance reasons. The default size is 10 Kb; if you're using node output from threads with little stack space, decreasing this value can prevent stack overflows. A minimum size of 1 Kb is recommended.

- `PUGIXML_MEMORY_XPATH_PAGE_SIZE` controls the page size for XPath memory allocation. Memory for XPath query objects as well as internal memory for XPath evaluation is allocated in pages of the specified size. The default size is 4 Kb; if you have a lot of resident XPath query objects, you might need to decrease the size to improve memory consumption. A minimum size of 256 bytes is recommended.

3.6.3. Document memory management internals

Constructing a document object using the default constructor does not result in any allocations; document node is stored inside the xml_document object.

When the document is loaded from file/buffer, unless an inplace loading function is used (see Loading document from memory), a complete copy of character stream is made; all names/values of nodes and attributes are allocated in this buffer. This buffer is allocated via a single large allocation and is only freed when document memory is reclaimed (i.e. if the xml_document object is destroyed or if another document is loaded in the same object). Also when loading from file or stream, an additional large

allocation may be performed if encoding conversion is required; a temporary buffer is allocated, and it is freed before load function returns.

All additional memory, such as memory for document structure (node/attribute objects) and memory for node/attribute names/values is allocated in pages on the order of 32 Kb; actual objects are allocated inside the pages using a memory management scheme optimized for fast allocation/deallocation of many small objects. Because of the scheme specifics, the pages are only destroyed if all objects inside them are destroyed; also, generally destroying an object does not mean that subsequent object creation will reuse the same memory. This means that it is possible to devise a usage scheme which will lead to higher memory usage than expected; one example is adding a lot of nodes, and them removing all even numbered ones; not a single page is reclaimed in the process. However this is an example specifically crafted to produce unsatisfying behavior; in all practical usage scenarios the memory consumption is less than that of a general-purpose allocator because allocation meta-data is very small in size.

4. 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 for performance reasons. Also some XML transformations (i.e. EOL handling or attribute value normalization) can impact parsing speed and thus can be disabled. However for vast majority of XML documents there is no performance difference between different parsing options. Parsing options also control whether certain XML nodes are parsed; see Parsing options for more information.

XML data is always converted to internal character format (see Unicode interface) 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. Unless explicit encoding is specified, loading functions perform automatic encoding detection based on first few characters of XML data, so in almost all cases you do not have to specify document encoding. Encoding conversion is described in more detail in Encodings.

4.1. Loading document from file

The most common source of XML data is files; pugixml provides dedicated functions for loading an XML document from file:

```
xml_parse_result xml_document::load_file(const char*
path, unsigned int options = parse_default, xml_encoding
encoding = encoding_auto);
xml_parse_result xml_document::load_file(const wchar_t*
path, unsigned int options = parse_default, xml_encoding
encoding = encoding_auto);
```

These functions accept the file path as its first argument, and also

two optional arguments, which specify parsing options (see Parsing options) and input data encoding (see Encodings). The path has the target operating system format, so it can be a relative or absolute one, it should have the delimiters of the target system, it should have the exact case if the target file system is case-sensitive, etc.

File path is passed to the system file opening function as is in case of the first function (which accepts `const char* path`); the second function either uses a special file opening function if it is provided by the runtime library or converts the path to UTF-8 and uses the system file opening function.

`load_file` 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). See Handling parsing errors for error handling details.

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;
```

4.2. Loading document from memory

Sometimes XML data should be loaded from some other source than a 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. All these scenarios require loading document from memory. First you should prepare a contiguous memory block with all XML data; then you have to invoke one of buffer loading functions. These functions will handle the necessary encoding conversions, if any, and then will parse the data into the corresponding XML tree. There are several buffer loading functions, which differ in the

behavior and thus in performance/memory usage:

```
xml_parse_result xml_document::load_buffer(const void*
contents, size_t size, unsigned int options =
parse_default, xml_encoding encoding = encoding_auto);
xml_parse_result xml_document::load_buffer_inplace(void*
contents, size_t size, unsigned int options =
parse_default, xml_encoding encoding = encoding_auto);
xml_parse_result
xml_document::load_buffer_inplace_own(void* contents,
size_t size, unsigned int options = parse_default,
xml_encoding encoding = encoding_auto);
```

All functions accept the buffer which is represented by a pointer to XML data, `contents`, and data size in bytes. Also there are two optional arguments, which specify parsing options (see Parsing options) and input data encoding (see Encodings). The buffer does not have to be zero-terminated.

`load_buffer` function works with immutable buffer - it does not ever modify the buffer. Because of this restriction it has to create a private buffer and copy XML data to it before parsing (applying encoding conversions if necessary). This copy operation carries a

performance penalty, so inplace functions are provided - `load_buffer_inplace` and `load_buffer_inplace_own` store the document data in the buffer, modifying it in the process. In order for the document to stay valid, you have to make sure that the buffer's lifetime exceeds that of the tree if you're using inplace functions. In addition to that, `load_buffer_inplace` does not assume ownership of the buffer, so you'll have to destroy it yourself; `load_buffer_inplace_own` assumes ownership of the buffer and destroys it once it is not needed. This means that if you're using `load_buffer_inplace_own`, you have to allocate memory with pugixml allocation function (you can get it via get_memory_allocation_function).

The best way from the performance/memory point of view is to load document using `load_buffer_inplace_own`; this function has maximum control of the buffer with XML data so it is able to avoid redundant copies and reduce peak memory usage while parsing. This is the recommended function if you have to load the document from memory and performance is critical.

There is also a simple helper function for cases when you want to

load the XML document from null-terminated character string:

```
xml_parse_result xml_document::load_string(const char_t*
contents, unsigned int options = parse_default);
```

It is equivalent to calling `load_buffer` with `size` being either `strlen(contents)` or `wcslen(contents) * sizeof(wchar_t)`, depending on the character type. This function assumes native encoding for input data, so it does not do any encoding conversion. In general, this function is fine for loading small documents from string literals, but has more overhead and less functionality than the buffer loading functions.

This is an example of loading XML document from memory using different functions (samples/load_memory.cpp):

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

```
// You can use load_buffer to load document from
```

```
immutable memory block:
pugi::xml_parse_result result = doc.load_buffer(source,
size);
```

```
// 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;
```

```
// You can use load_buffer_inplace_own to load document
from mutable memory block and to pass the ownership of
this block
// The block has to be allocated via pugixml allocation
function - using i.e. operator new here is incorrect
```

```
char* buffer =
static_cast<char*>(pugi::get_memory_allocation_function()
(size));
memcpy(buffer, source, size);

// The block will be deleted by the document
pugi::xml_parse_result result =
doc.load_buffer_inplace_own(buffer, size);
```

```
// You can use load to load document from null-terminated
strings, for example literals:
pugi::xml_parse_result result = doc.load_string("<mesh
name='sphere'><bounds>0 0 1 1</bounds></mesh>");
```

4.3. Loading document from C++ IOstreams

To enhance interoperability, pugixml provides functions for loading document from any object which implements C++ `std::istream` interface. This allows you to load documents from any standard C++ stream (i.e. file stream) or any third-party compliant implementation (i.e. Boost Iostreams). There are two functions, one works with narrow character streams, another

handles wide character ones:

```
xml_parse_result xml_document::load(std::istream& stream,
unsigned int options = parse_default, xml_encoding
encoding = encoding_auto);
xml_parse_result xml_document::load(std::wistream&
stream, unsigned int options = parse_default);
```

`load` with `std::istream` argument loads the document from stream from the current read position to the end, treating the stream contents as a byte stream of the specified encoding (with encoding autodetection as necessary). Thus calling `xml_document::load` on an opened `std::ifstream` object is equivalent to calling `xml_document::load_file`.

`load` with `std::wstream` argument treats the stream contents as a wide character stream (encoding is always encoding_wchar). Because of this, using `load` with wide character streams requires careful (usually platform-specific) stream setup (i.e. using the `imbue` function). Generally use of wide streams is discouraged, however it provides you the ability to load documents from non-Unicode encodings, i.e. you can load Shift-JIS encoded data if you

set the correct locale.

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);
```

4.4. Handling parsing errors

All document loading functions return the parsing result via `xml_parse_result` object. It contains parsing status, the offset of last successfully parsed character from the beginning of the source stream, and the encoding of the source stream:

```
struct xml_parse_result
{
    xml_parse_status status;
    ptrdiff_t offset;
    xml_encoding encoding;
```

```
    operator bool() const;
    const char* description() const;
};
```

Parsing status is represented as the `xml_parse_status` enumeration and can be one of the following:

- `status_ok` means that no error was encountered during parsing; the source stream represents the valid XML document which was fully parsed and converted to a tree.
- `status_file_not_found` is only returned by `load_file` function and means that file could not be opened.
- `status_io_error` is returned by `load_file` function and by `load` functions with `std::istream`/`std::wstream` arguments; it means that some I/O error has occurred during reading the file/stream.
- `status_out_of_memory` means that there was not enough memory during some allocation; any allocation failure during parsing results in this error.
- `status_internal_error` means that something went

horribly wrong; currently this error does not occur

- `status_unrecognized_tag` means that parsing stopped due to a tag with either an empty name or a name which starts with incorrect character, such as `#`.
- `status_bad_pi` means that parsing stopped due to incorrect document declaration/processing instruction
- `status_bad_comment`, `status_bad_cdata`, `status_bad_doctype` and `status_bad_pcdata` mean that parsing stopped due to the invalid construct of the respective type
- `status_bad_start_element` means that parsing stopped because starting tag either had no closing `>` symbol or contained some incorrect symbol
- `status_bad_attribute` means that parsing stopped because there was an incorrect attribute, such as an attribute without value or with value that is not quoted (note that `<node attr=1>` is incorrect in XML)
- `status_bad_end_element` means that parsing stopped because ending tag had incorrect syntax (i.e. extra non-

whitespace symbols between tag name and `>`)

- `status_end_element_mismatch` means that parsing stopped because the closing tag did not match the opening one (i.e. `<node></nedo>`) or because some tag was not closed at all
- `status_no_document_element` means that no element nodes were discovered during parsing; this usually indicates an empty or invalid document

`description()` member function can be used to convert parsing status to a string; the returned message is always in English, so you'll have to write your own function if you need a localized string. However please note that the exact messages returned by `description()` function may change from version to version, so any complex status handling should be based on `status` value. Note that `description()` returns a `char` string even in `PUGIXML_WCHAR_MODE`; you'll have to call as_wide to get the `wchar_t` string.

If parsing failed because the source data was not a valid XML, the resulting tree is not destroyed - despite the fact that load function

returns error, you can use the part of the tree that was successfully parsed. Obviously, the last element may have an unexpected name/value; for example, if the attribute value does not end with the necessary quotation mark, like in `<node attr="value>some data</node>` example, the value of attribute `attr` will contain the string `value>some data</node>`.

In addition to the status code, parsing result has an `offset` member, which contains the offset of last successfully parsed character if parsing failed because of an error in source data; otherwise `offset` is 0. For parsing efficiency reasons, pugixml does not track the current line during parsing; this offset is in units of pugi::char_t (bytes for character mode, wide characters for wide character mode). Many text editors support 'Go To Position' feature - you can use it to locate the exact error position. Alternatively, if you're loading the document from memory, you can display the error chunk along with the error description (see the example code below).

> CAUTION
>
> Offset is calculated in the XML buffer in native encoding; if encoding conversion is performed

CAUTION

during parsing, offset can not be used to reliably track the error position.

Parsing result also has an `encoding` member, which can be used to check that the source data encoding was correctly guessed. It is equal to the exact encoding used during parsing (i.e. with the exact endianness); see Encodings for more information.

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 { … }`.

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";
}
```

4.5. Parsing options

All document loading functions accept the optional parameter `options`. This is a bitmask that customizes the parsing process: you can select the node types that are parsed and various transformations that are performed with the XML text. Disabling certain transformations can improve parsing performance for some documents; however, the code for all transformations is very well optimized, and thus the majority of documents won't get

any performance benefit. As a rule of thumb, only modify parsing flags if you want to get some nodes in the document that are excluded by default (i.e. declaration or comment nodes).

> **NOTE** You should use the usual bitwise arithmetics to manipulate the bitmask: to enable a flag, use `mask | flag`; to disable a flag, use `mask & ~flag`.

These flags control the resulting tree contents:

- `parse_declaration` determines if XML document declaration (node with type node_declaration) is to be put in DOM tree. If this flag is off, it is not put in the tree, but is still parsed and checked for correctness. This flag is **off** by default.
- `parse_doctype` determines if XML document type declaration (node with type node_doctype) is to be put in DOM tree. If this flag is off, it is not put in the tree, but is still parsed and checked for correctness. This flag is **off** by default.
- `parse_pi` determines if processing instructions (nodes with type node_pi) are to be put in DOM tree. If this flag is off, they

are not put in the tree, but are still parsed and checked for correctness. Note that `<?xml …?>` (document declaration) is not considered to be a PI. This flag is **off** by default.

- `parse_comments` determines if comments (nodes with type node_comment) are to be put in DOM tree. If this flag is off, they are not put in the tree, but are still parsed and checked for correctness. This flag is **off** by default.
- `parse_cdata` determines if CDATA sections (nodes with type node_cdata) are to be put in DOM tree. If this flag is off, they are not put in the tree, but are still parsed and checked for correctness. This flag is **on** by default.
- `parse_trim_pcdata` determines if leading and trailing whitespace characters are to be removed from PCDATA nodes. While for some applications leading/trailing whitespace is significant, often the application only cares about the non-whitespace contents so it's easier to trim whitespace from text during parsing. This flag is **off** by default.
- `parse_ws_pcdata` determines if PCDATA nodes (nodes with type node_pcdata) that consist only of whitespace characters

are to be put in DOM tree. Often whitespace-only data is not significant for the application, and the cost of allocating and storing such nodes (both memory and speed-wise) can be significant. For example, after parsing XML string `<node> <a/> </node>`, `<node>` element will have three children when `parse_ws_pcdata` is set (child with type node_pcdata and value `" "`, child with type node_element and name `"a"`, and another child with type node_pcdata and value `" "`), and only one child when `parse_ws_pcdata` is not set. This flag is **off** by default.

- `parse_ws_pcdata_single` determines if whitespace-only PCDATA nodes that have no sibling nodes are to be put in DOM tree. In some cases application needs to parse the whitespace-only contents of nodes, i.e. `<node> </node>`, but is not interested in whitespace markup elsewhere. It is possible to use parse_ws_pcdata flag in this case, but it results in excessive allocations and complicates document processing; this flag can be used to avoid that. As an example, after parsing XML string `<node> <a> </a> </node>` with `parse_ws_pcdata_single` flag set, `<node>` element will have one child `<a>`, and `<a>`

element will have one child with type node_pcdata and value `" "`. This flag has no effect if parse_ws_pcdata is enabled. This flag is **off** by default.

- `parse_fragment` determines if document should be treated as a fragment of a valid XML. Parsing document as a fragment leads to top-level PCDATA content (i.e. text that is not located inside a node) to be added to a tree, and additionally treats documents without element nodes as valid. This flag is **off** by default.

CAUTION

Using in-place parsing (load_buffer_inplace) with `parse_fragment` flag may result in the loss of the last character of the buffer if it is a part of PCDATA. Since PCDATA values are null-terminated strings, the only way to resolve this is to provide a null-terminated buffer as an input to `load_buffer_inplace` - i.e. `doc.load_buffer_inplace("test\0", 5, pugi::parse_default | pugi::parse_fragment)`.

These flags control the transformation of tree element contents:

- `parse_escapes` determines if character and entity references are to be expanded during the parsing process. Character references have the form `&#...;` or `&#x...;` (`...` is Unicode numeric representation of character in either decimal (`&#...;`) or hexadecimal (`&#x...;`) form), entity references are `<`, `>`, `&`, `&apos;` and `&quot;` (note that as pugixml does not handle DTD, the only allowed entities are predefined ones). If character/entity reference can not be expanded, it is left as is, so you can do additional processing later. Reference expansion is performed on attribute values and PCDATA content. This flag is **on** by default.
- `parse_eol` determines if EOL handling (that is, replacing sequences `\r\n` by a single `\n` character, and replacing all standalone `\r` characters by `\n`) is to be performed on input data (that is, comment contents, PCDATA/CDATA contents and attribute values). This flag is **on** by default.
- `parse_wconv_attribute` determines if attribute value normalization should be performed for all attributes. This

means, that whitespace characters (new line, tab and space) are replaced with space (`' '`). New line characters are always treated as if parse_eol is set, i.e. `\r\n` is converted to a single space. This flag is **on** by default.

- `parse_wnorm_attribute` determines if extended attribute value normalization should be performed for all attributes. This means, that after attribute values are normalized as if parse_wconv_attribute was set, leading and trailing space characters are removed, and all sequences of space characters are replaced by a single space character. parse_wconv_attribute has no effect if this flag is on. This flag is **off** by default.

> NOTE
>
> `parse_wconv_attribute` option performs transformations that are required by W3C specification for attributes that are declared as CDATA; parse_wnorm_attribute performs transformations required for NMTOKENS attributes. In the absence of document type declaration all attributes should behave as if they are declared as CDATA, thus parse_wconv_attribute

> is the default option.

Additionally there are three predefined option masks:

- `parse_minimal` has all options turned off. This option mask means that pugixml does not add declaration nodes, document type declaration nodes, PI nodes, CDATA sections and comments to the resulting tree and does not perform any conversion for input data, so theoretically it is the fastest mode. However, as mentioned above, in practice parse_default is usually equally fast.
- `parse_default` is the default set of flags, i.e. it has all options set to their default values. It includes parsing CDATA sections (comments/PIs are not parsed), performing character and entity reference expansion, replacing whitespace characters with spaces in attribute values and performing EOL handling. Note, that PCDATA sections consisting only of whitespace characters are not parsed (by default) for performance reasons.
- `parse_full` is the set of flags which adds nodes of all types to the resulting tree and performs default conversions for input

data. It includes parsing CDATA sections, comments, PI nodes, document declaration node and document type declaration node, performing character and entity reference expansion, replacing whitespace characters with spaces in attribute values and performing EOL handling. Note, that PCDATA sections consisting only of whitespace characters are not parsed in this mode.

This is an example of using different parsing options (samples/load_options.cpp):

```
const char* source = "<!--comment--><node>&lt;</node>";

// Parsing with default options; note that comment node
is not added to the tree, and entity reference &lt; is
expanded
doc.load_string(source);
std::cout << "First node value: [" <<
doc.first_child().value() << "], node child value: [" <<
doc.child_value("node") << "]\n";

// Parsing with additional parse_comments option; comment
node is now added to the tree
doc.load_string(source, pugi::parse_default |
```

```
pugi::parse_comments);
std::cout << "First node value: [" <<
doc.first_child().value() << "], node child value: [" <<
doc.child_value("node") << "]\n";

// Parsing with additional parse_comments option and
without the (default) parse_escapes option; &lt; is not
expanded
doc.load_string(source, (pugi::parse_default |
pugi::parse_comments) & ~pugi::parse_escapes);
std::cout << "First node value: [" <<
doc.first_child().value() << "], node child value: [" <<
doc.child_value("node") << "]\n";

// Parsing with minimal option mask; comment node is not
added to the tree, and &lt; is not expanded
doc.load_string(source, pugi::parse_minimal);
std::cout << "First node value: [" <<
doc.first_child().value() << "], node child value: [" <<
doc.child_value("node") << "]\n";
```

4.6. Encodings

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. Most loading functions accept the optional parameter `encoding`. This is a value of enumeration type `xml_encoding`, that can have the following values:

- `encoding_auto` means that pugixml will try to guess the encoding based on source XML data. The algorithm is a modified version of the one presented in Appendix F.1 of XML recommendation; it tries to match the first few bytes of input data with the following patterns in strict order:
  - If first four bytes match UTF-32 BOM (Byte Order Mark), encoding is assumed to be UTF-32 with the endianness equal to that of BOM;
  - If first two bytes match UTF-16 BOM, encoding is assumed to be UTF-16 with the endianness equal to that of BOM;
  - If first three bytes match UTF-8 BOM, encoding is assumed to be UTF-8;
  - If first four bytes match UTF-32 representation of `<`, encoding is assumed to be UTF-32 with the corresponding endianness;

    - If first four bytes match UTF-16 representation of `<?`, encoding is assumed to be UTF-16 with the corresponding endianness;
    - If first two bytes match UTF-16 representation of `<`, encoding is assumed to be UTF-16 with the corresponding endianness (this guess may yield incorrect result, but it's better than UTF-8);
    - Otherwise encoding is assumed to be UTF-8.
- `encoding_utf8` corresponds to UTF-8 encoding as defined in the Unicode standard; UTF-8 sequences with length equal to 5 or 6 are not standard and are rejected.
- `encoding_utf16_le` corresponds to little-endian UTF-16 encoding as defined in the Unicode standard; surrogate pairs are supported.
- `encoding_utf16_be` corresponds to big-endian UTF-16 encoding as defined in the Unicode standard; surrogate pairs are supported.
- `encoding_utf16` corresponds to UTF-16 encoding as defined

in the Unicode standard; the endianness is assumed to be that of the target platform.

- `encoding_utf32_le` corresponds to little-endian UTF-32 encoding as defined in the Unicode standard.
- `encoding_utf32_be` corresponds to big-endian UTF-32 encoding as defined in the Unicode standard.
- `encoding_utf32` corresponds to UTF-32 encoding as defined in the Unicode standard; the endianness is assumed to be that of the target platform.
- `encoding_wchar` corresponds to the encoding of `wchar_t` type; it has the same meaning as either `encoding_utf16` or `encoding_utf32`, depending on `wchar_t` size.
- `encoding_latin1` corresponds to ISO-8859-1 encoding (also known as Latin-1).

The algorithm used for `encoding_auto` correctly detects any supported Unicode encoding for all well-formed XML documents (since they start with document declaration) and for all other XML documents that start with `<`; if your XML document does not start

with `<` and has encoding that is different from UTF-8, use the specific encoding.

> **NOTE** The current behavior for Unicode conversion is to skip all invalid UTF sequences during conversion. This behavior should not be relied upon; moreover, in case no encoding conversion is performed, the invalid sequences are not removed, so you'll get them as is in node/attribute contents.

4.7. Conformance to W3C specification

pugixml 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.

There is only one non-conformant behavior when dealing with valid XML documents: pugixml does not use information supplied in document type declaration for parsing. This means that entities

declared in DOCTYPE are not expanded, and all attribute/PCDATA values are always processed in a uniform way that depends only on parsing options.

As for rejecting invalid XML documents, there are a number of incompatibilities with W3C specification, including:

- Multiple attributes of the same node can have equal names.
- All non-ASCII characters are treated in the same way as symbols of English alphabet, so some invalid tag names are not rejected.
- Attribute values which contain `<` are not rejected.
- Invalid entity/character references are not rejected and are instead left as is.
- Comment values can contain `--`.
- XML data is not required to begin with document declaration; additionally, document declaration can appear after comments and other nodes.
- Invalid document type declarations are silently ignored in

some cases.

5. Accessing document data

pugixml features an extensive interface for getting various types of data from the document and for traversing the document. This section provides documentation for all such functions that do not modify the tree except for XPath-related functions; see XPath for XPath reference. As discussed in C++ interface, there are two types of handles to tree data - xml_node and xml_attribute. The handles have special null (empty) values which propagate through various functions and thus are useful for writing more concise code; see this description for details. The documentation in this section will explicitly state the results of all function in case of null inputs.

5.1. Basic traversal functions

The internal representation of the document is a tree, where each node has a list of child nodes (the order of children corresponds to

their order in the XML representation), and additionally element nodes have a list of attributes, which is also ordered. Several functions are provided in order to let you get from one node in the tree to the other. These functions roughly correspond to the internal representation, and thus are usually building blocks for other methods of traversing (i.e. XPath traversals are based on these functions).

```
xml_node xml_node::parent() const;
xml_node xml_node::first_child() const;
xml_node xml_node::last_child() const;
xml_node xml_node::next_sibling() const;
xml_node xml_node::previous_sibling() const;

xml_attribute xml_node::first_attribute() const;
xml_attribute xml_node::last_attribute() const;
xml_attribute xml_attribute::next_attribute() const;
xml_attribute xml_attribute::previous_attribute() const;
```

`parent` function returns the node's parent; all non-null nodes except the document have non-null parent. `first_child` and `last_child` return the first and last child of the node,

respectively; note that only document nodes and element nodes can have non-empty child node list. If node has no children, both functions return null nodes. `next_sibling` and `previous_sibling` return the node that's immediately to the right/left of this node in the children list, respectively - for example, in `<a/><b/><c/>`, calling `next_sibling` for a handle that points to `<b/>` results in a handle pointing to `<c/>`, and calling `previous_sibling` results in handle pointing to `<a/>`. If node does not have next/previous sibling (this happens if it is the last/first node in the list, respectively), the functions return null nodes. `first_attribute`, `last_attribute`, `next_attribute` and `previous_attribute` functions behave similarly to the corresponding child node functions and allow to iterate through attribute list in the same way.

> **NOTE**
>
> Because of memory consumption reasons, attributes do not have a link to their parent nodes. Thus there is no `xml_attribute::parent()` function.

Calling any of the functions above on the null handle results in a

null handle - i.e. `node.first_child().next_sibling()` returns the second child of `node`, and null handle if `node` is null, has no children at all or if it has only one child node.

With these functions, you can iterate through all child nodes and display all attributes like this (samples/traverse_base.cpp):

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

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

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

5.2. Getting node data

Apart from structural information (parent, child nodes, attributes), nodes can have name and value, both of which are strings. Depending on node type, name or value may be absent. node_document nodes do not have a name or value, node_element and node_declaration nodes always have a name but never have a value, node_pcdata, node_cdata, node_comment and node_doctype nodes never have a name but always have a value (it may be empty though), node_pi nodes always have a name and a value (again, value may be empty). In order to get node's name or value, you can use the following functions:

```
const char_t* xml_node::name() const;
const char_t* xml_node::value() const;
```

In case node does not have a name or value or if the node handle is null, both functions return empty strings - they never return null pointers.

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 several helper functions to parse such data:

```
const char_t* xml_node::child_value() const;
const char_t* xml_node::child_value(const char_t* name)
const;
xml_text xml_node::text() const;
```

`child_value()` returns the value of the first child with type node_pcdata or node_cdata; `child_value(name)` is a simple wrapper for `child(name).child_value()`. For the above example, calling `node.child_value("description")` and `description.child_value()` will both produce string `"This is a node"`. If there is no child with relevant type, or if the handle is null, `child_value` functions return empty string.

`text()` returns a special object that can be used for working with PCDATA contents in more complex cases than just retrieving the

value; it is described in Working with text contents sections.

There is an example of using some of these functionsat the end of the next section.

5.3. Getting attribute data

All attributes have name and value, both of which are strings (value may be empty). There are two corresponding accessors, like for `xml_node`:

```
const char_t* xml_attribute::name() const;
const char_t* xml_attribute::value() const;
```

In case the attribute handle is null, both functions return empty strings - they never return null pointers.

If you need a non-empty string if the attribute handle is null (for example, you need to get the option value from XML attribute, but if it is not specified, you need it to default to `"sorted"` instead of `""`), you can use `as_string` accessor:

```
const char_t* xml_attribute::as_string(const char_t* def
= "") const;
```

It returns `def` argument if the attribute handle is null. If you do not specify the argument, the function is equivalent to `value()`.

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:

```
int xml_attribute::as_int(int def = 0) const;
unsigned int xml_attribute::as_uint(unsigned int def = 0)
const;
double xml_attribute::as_double(double def = 0) const;
float xml_attribute::as_float(float def = 0) const;
bool xml_attribute::as_bool(bool def = false) const;
long long xml_attribute::as_llong(long long def = 0)
const;
unsigned long long xml_attribute::as_ullong(unsigned long
long def = 0) const;
```

`as_int`, `as_uint`, `as_llong`, `as_ullong`, `as_double` and `as_float` convert attribute values to numbers. If attribute handle is null or attribute value is empty, `def` argument is returned (which is 0 by default). Otherwise, all leading whitespace characters are truncated, and the remaining string is parsed as an integer number in either decimal or hexadecimal form (applicable to `as_int`, `as_uint`, `as_llong` and `as_ullong`; hexadecimal format is used if the number has `0x` or `0X` prefix) or as a floating point number in either decimal or scientific form (`as_double` or `as_float`). Any extra characters are silently discarded, i.e. `as_int` will return `1` for string `"1abc"`.

In case the input string contains a number that is out of the target numeric range, the result is undefined.

> **CAUTION**
>
> Number conversion functions depend on current C locale as set with `setlocale`, so may return unexpected results if the locale is different from `"C"`.

`as_bool` converts attribute value to boolean as follows: if

attribute handle is null, `def` argument is returned (which is `false` by default). If attribute value is empty, `false` is returned. Otherwise, `true` is returned if the first character is one of `'1', 't', 'T', 'y', 'Y'`. This means that strings like `"true"` and `"yes"` are recognized as `true`, while strings like `"false"` and `"no"` are recognized as `false`. For more complex matching you'll have to write your own function.

> **NOTE** `as_llong` and `as_ullong` are only available if your platform has reliable support for the `long long` type, including string conversions.

This is an example of using these functions, along with node data retrieval ones (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";
}
```

5.4. Contents-based traversal functions

Since a lot of document traversal consists of finding the node/attribute with the correct name, there are special functions for that purpose:

```
xml_node xml_node::child(const char_t* name) const;
xml_attribute xml_node::attribute(const char_t* name)
const;
xml_node xml_node::next_sibling(const char_t* name)
const;
xml_node xml_node::previous_sibling(const char_t* name)
const;
```

`child` and `attribute` return the first child/attribute with the specified name; `next_sibling` and `previous_sibling` return the first sibling in the corresponding direction with the specified

name. All string comparisons are case-sensitive. In case the node handle is null or there is no node/attribute with the specified name, null handle is returned.

`child` and `next_sibling` functions can be used together to loop through all child nodes with the desired name like this:

```
for (pugi::xml_node tool = tools.child("Tool"); tool;
tool = tool.next_sibling("Tool"))
```

Occasionally the needed node is specified not by the unique name but instead by the value of some attribute; for example, it is common to have node collections with each node having a unique id: `<group><item id="1"/> <item id="2"/></group>`. There are two functions for finding child nodes based on the attribute values:

```
xml_node xml_node::find_child_by_attribute(const char_t*
name, const char_t* attr_name, const char_t* attr_value)
const;
xml_node xml_node::find_child_by_attribute(const char_t*
attr_name, const char_t* attr_value) const;
```

The three-argument function returns the first child node with the specified name which has an attribute with the specified name/value; the two-argument function skips the name test for the node, which can be useful for searching in heterogeneous collections. If the node handle is null or if no node is found, null handle is returned. All string comparisons are case-sensitive.

In all of the above functions, all arguments have to be valid strings; passing null pointers results in undefined behavior.

This is an example of using these 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";
```

```
}
```

5.5. Range-based for-loop support

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 2012+, 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.

```
implementation-defined-type xml_node::children() const;
implementation-defined-type xml_node::children(const
char_t* name) const;
implementation-defined-type xml_node::attributes() const;
```

`children` function allows you to enumerate all child nodes; `children` function with `name` argument allows you to enumerate all child nodes with a specific name; `attributes` function allows you to enumerate all attributes of the node. Note that you can also use node object itself in a range-based for

construct, which is equivalent to using `children()`.

This is an example of using these functions (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;
}
```

5.6. Traversing node/attribute lists via

iterators

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:

```
class xml_node_iterator;
class xml_attribute_iterator;

typedef xml_node_iterator xml_node::iterator;
iterator xml_node::begin() const;
iterator xml_node::end() const;

typedef xml_attribute_iterator
xml_node::attribute_iterator;
attribute_iterator xml_node::attributes_begin() const;
attribute_iterator xml_node::attributes_end() const;
```

`begin` and `attributes_begin` return iterators that point to the first node/attribute, respectively; `end` and `attributes_end` return past-the-end iterator for node/attribute list, respectively -

this iterator can't be dereferenced, but decrementing it results in an iterator pointing to the last element in the list (except for empty lists, where decrementing past-the-end iterator results in undefined behavior). Past-the-end iterator is commonly used as a termination value for iteration loops (see sample below). If you want to get an iterator that points to an existing handle, you can construct the iterator with the handle as a single constructor argument, like so: `xml_node_iterator(node)`. For `xml_attribute_iterator`, you'll have to provide both an attribute and its parent node.

`begin` and `end` return equal iterators if called on null node; such iterators can't be dereferenced. `attributes_begin` and `attributes_end` behave the same way. For correct iterator usage this means that child node/attribute collections of null nodes appear to be empty.

Both types of iterators have bidirectional iterator semantics (i.e. they can be incremented and decremented, but efficient random access is not supported) and support all usual iterator operations - comparison, dereference, etc. 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;
}
```

> Node and attribute iterators are somewhere in the middle between const and non-const iterators.

CAUTION

While dereference operation yields a non-constant reference to the object, so that you can use it for tree modification operations, modifying this reference using assignment - i.e. passing iterators to a function like `std::sort` - will not give expected results, as assignment modifies local handle that's stored in the iterator.

5.7. Recursive traversal with xml_tree_walker

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:

```
class xml_tree_walker
{
public:
    virtual bool begin(xml_node& node);
    virtual bool for_each(xml_node& node) = 0;
```

```
    virtual bool end(xml_node& node);

    int depth() const;
};

bool xml_node::traverse(xml_tree_walker& walker);
```

The traversal is launched by calling `traverse` function on traversal root and proceeds as follows:

- First, `begin` function is called with traversal root as its argument.
- Then, `for_each` function is called for all nodes in the traversal subtree in depth first order, excluding the traversal root. Node is passed as an argument.
- Finally, `end` function is called with traversal root as its argument.

If `begin`, `end` or any of the `for_each` calls return `false`, the traversal is terminated and `false` is returned as the traversal result; otherwise, the traversal results in `true`. Note that you

don't have to override `begin` or `end` functions; their default implementations return `true`.

You can get the node's depth relative to the traversal root at any point by calling `depth` function. It returns `-1` if called from `begin`/`end`, and returns 0-based depth if called from `for_each` - depth is 0 for all children of the traversal root, 1 for all grandchildren and so on.

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);
```

5.8. Searching for nodes/attributes with predicates

While there are existing functions for getting a node/attribute with known contents, they are often not sufficient for simple queries. As an alternative for manual iteration through nodes/attributes until the needed one is found, you can make a predicate and call one of `find_` functions:

```
template <typename Predicate> xml_attribute
xml_node::find_attribute(Predicate pred) const;
template <typename Predicate> xml_node
xml_node::find_child(Predicate pred) const;
template <typename Predicate> xml_node
xml_node::find_node(Predicate pred) const;
```

The predicate should be either a plain function or a function object which accepts one argument of type `xml_attribute` (for `find_attribute`) or `xml_node` (for `find_child` and `find_node`), and returns `bool`. The predicate is never called with null handle as an argument.

`find_attribute` function iterates through all attributes of the specified node, and returns the first attribute for which the predicate returned `true`. If the predicate returned `false` for all attributes or if there were no attributes (including the case where the node is null), null attribute is returned.

`find_child` function iterates through all child nodes of the specified node, and returns the first node for which the predicate returned `true`. If the predicate returned `false` for all nodes or if there were no child nodes (including the case where the node is null), null node is returned.

`find_node` function performs a depth-first traversal through the subtree of the specified node (excluding the node itself), and returns the first node for which the predicate returned `true`. If

the predicate returned `false` for all nodes or if subtree was empty, null node is returned.

This is an example of using predicate-based functions (samples/traverse_predicate.cpp):

```
bool small_timeout(pugi::xml_node node)
{
    return node.attribute("Timeout").as_int() < 20;
}

struct allow_remote_predicate
{
    bool operator()(pugi::xml_attribute attr) const
    {
        return strcmp(attr.name(), "AllowRemote") == 0;
    }

    bool operator()(pugi::xml_node node) const
    {
        return node.attribute("AllowRemote").as_bool();
    }
};
```

```
// Find child via predicate (looks for direct children
only)
std::cout <<
tools.find_child(allow_remote_predicate()).attribute("Fil
ename").value() << std::endl;

// Find node via predicate (looks for all descendants in
depth-first order)
std::cout <<
doc.find_node(allow_remote_predicate()).attribute("Filena
me").value() << std::endl;

// Find attribute via predicate
std::cout <<
tools.last_child().find_attribute(allow_remote_predicate(
)).value() << std::endl;

// We can use simple functions instead of function
objects
std::cout <<
tools.find_child(small_timeout).attribute("Filename").val
ue() << std::endl;
```

5.9. Working with text contents

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 a special class, `xml_text`, to work with such data. Working with text objects to modify data is described in the documentation for modifying document data; this section describes the access interface of `xml_text`.

You can get the text object from a node by using `text()` method:

```
xml_text xml_node::text() const;
```

If the node has a type `node_pcdata` or `node_cdata`, then the node itself is used to return data; otherwise, a first child node of type `node_pcdata` or `node_cdata` is used.

You can check if the text object is bound to a valid PCDATA/CDATA node by using it as a boolean value, i.e. `if (text) { … }` or `if (!text) { … }`. Alternatively you can check it by using the

`empty()` method:

```
bool xml_text::empty() const;
```

Given a text object, you can get the contents (i.e. the value of PCDATA/CDATA node) by using the following function:

```
const char_t* xml_text::get() const;
```

In case text object is empty, the function returns an empty string - it never returns a null pointer.

If you need a non-empty string if the text object is empty, or if the text contents is actually a number or a boolean that is stored as a string, you can use the following accessors:

```
const char_t* xml_text::as_string(const char_t* def = "")
const;
int xml_text::as_int(int def = 0) const;
unsigned int xml_text::as_uint(unsigned int def = 0)
const;
```

```
double xml_text::as_double(double def = 0) const;
float xml_text::as_float(float def = 0) const;
bool xml_text::as_bool(bool def = false) const;
long long xml_text::as_llong(long long def = 0) const;
unsigned long long xml_text::as_ullong(unsigned long long
def = 0) const;
```

All of the above functions have the same semantics as similar `xml_attribute` members: they return the default argument if the text object is empty, they convert the text contents to a target type using the same rules and restrictions. You can refer to documentation for the attribute functions for details.

`xml_text` is essentially a helper class that operates on `xml_node` values. It is bound to a node of type node_pcdata or node_cdata. You can use the following function to retrieve this node:

```
xml_node xml_text::data() const;
```

Essentially, assuming `text` is an `xml_text` object, calling `text.get()` is equivalent to calling `text.data().value()`.

This is an example of using `xml_text` object (samples/text.cpp):

```
std::cout << "Project name: " <<
project.child("name").text().get() << std::endl;
std::cout << "Project version: " <<
project.child("version").text().as_double() << std::endl;
std::cout << "Project visibility: " <<
(project.child("public").text().as_bool(/* def= */ true)
? "public" : "private") << std::endl;
std::cout << "Project description: " <<
project.child("description").text().get() << std::endl;
```

5.10. Miscellaneous functions

If you need to get the document root of some node, you can use the following function:

```
xml_node xml_node::root() const;
```

This function returns the node with type node_document, which is the root node of the document the node belongs to (unless the node is null, in which case null node is returned).

While pugixml supports complex XPath expressions, sometimes a simple path handling facility is needed. There are two functions, for getting node path and for converting path to a node:

```
string_t xml_node::path(char_t delimiter = '/') const;
xml_node xml_node::first_element_by_path(const char_t*
path, char_t delimiter = '/') const;
```

Node paths consist of node names, separated with a delimiter (which is `/` by default); also paths can contain self (`.`) and parent (`..`) pseudo-names, so that this is a valid path: `"../../foo/./bar"`. `path` returns the path to the node from the document root, `first_element_by_path` looks for a node represented by a given path; a path can be an absolute one (absolute paths start with the delimiter), in which case the rest of the path is treated as document root relative, and relative to the given node. For example, in the following document: `<a><b><c/></b></a>`, node `<c/>` has path `"a/b/c"`; calling `first_element_by_path` for document with path `"a/b"` results in node `<b/>`; calling `first_element_by_path` for node `<a/>`

with path `"../a/./b/../."` results in node `<a/>`; calling `first_element_by_path` with path `"/a"` results in node `<a/>` for any node.

In case path component is ambiguous (if there are two nodes with given name), the first one is selected; paths are not guaranteed to uniquely identify nodes in a document. If any component of a path is not found, the result of `first_element_by_path` is null node; also `first_element_by_path` returns null node for null nodes, in which case the path does not matter. `path` returns an empty string for null nodes.

> **NOTE** `path` function returns the result as STL string, and thus is not available if PUGIXML_NO_STL is defined.

pugixml does not record row/column information for nodes upon parsing for efficiency reasons. However, if the node has not changed in a significant way since parsing (the name/value are not changed, and the node itself is the original one, i.e. it was not deleted from the tree and re-added later), it is possible to get the offset from the beginning of XML buffer:

```
ptrdiff_t xml_node::offset_debug() const;
```

If the offset is not available (this happens if the node is null, was not originally parsed from a stream, or has changed in a significant way), the function returns -1. Otherwise it returns the offset to node's data from the beginning of XML buffer in pugi::char_t units. For more information on parsing offsets, see parsing error handling documentation.

---

6. Modifying document data

The document in pugixml is fully mutable: you can completely change the document structure and modify the data of nodes/attributes. This section provides documentation for the relevant functions. 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.

6.1. Setting node data

As discussed before, nodes can have name and value, both of which are strings. Depending on node type, name or value may be absent. node_document nodes do not have a name or value, node_element and node_declaration nodes always have a name

but never have a value, node_pcdata, node_cdata, node_comment and node_doctype nodes never have a name but always have a value (it may be empty though), node_pi nodes always have a name and a value (again, value may be empty). In order to set node's name or value, you can use the following functions:

```
bool xml_node::set_name(const char_t* rhs);
bool xml_node::set_value(const char_t* rhs);
```

Both functions try to set the name/value to the specified string, and return the operation result. The operation fails if the node can not have name or value (for instance, when trying to call `set_name` on a node_pcdata node), if the node handle is null, or if there is insufficient memory to handle the request. The provided string is copied into document managed memory and can be destroyed after the function returns (for example, you can safely pass stack-allocated buffers to these functions). The name/value content is not verified, so take care to use only valid XML names, or the document may become malformed.

This is an example of setting node 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;
```

6.2. Setting attribute data

All attributes have name and value, both of which are strings (value may be empty). You can set them with the following

functions:

```
bool xml_attribute::set_name(const char_t* rhs);
bool xml_attribute::set_value(const char_t* rhs);
```

Both functions try to set the name/value to the specified string, and return the operation result. The operation fails if the attribute handle is null, or if there is insufficient memory to handle the request. The provided string is copied into document managed memory and can be destroyed after the function returns (for example, you can safely pass stack-allocated buffers to these functions). The name/value content is not verified, so take care to use only valid XML names, or the document may become malformed.

In addition to string functions, several functions are provided for handling attributes with numbers and booleans as values:

```
bool xml_attribute::set_value(int rhs);
bool xml_attribute::set_value(unsigned int rhs);
bool xml_attribute::set_value(double rhs);
```

```
bool xml_attribute::set_value(float rhs);
bool xml_attribute::set_value(bool rhs);
bool xml_attribute::set_value(long long rhs);
bool xml_attribute::set_value(unsigned long long rhs);
```

The above functions convert the argument to string and then call the base `set_value` function. Integers are converted to a decimal form, floating-point numbers are converted to either decimal or scientific form, depending on the number magnitude, boolean values are converted to either `"true"` or `"false"`.

CAUTION

Number conversion functions depend on current C locale as set with `setlocale`, so may generate unexpected results if the locale is different from `"C"`.

NOTE

`set_value` overloads with `long long` type are only available if your platform has reliable support for the type, including string conversions.

For convenience, all `set_value` functions have the

corresponding assignment operators:

```
xml_attribute& xml_attribute::operator=(const char_t*
rhs);
xml_attribute& xml_attribute::operator=(int rhs);
xml_attribute& xml_attribute::operator=(unsigned int
rhs);
xml_attribute& xml_attribute::operator=(double rhs);
xml_attribute& xml_attribute::operator=(float rhs);
xml_attribute& xml_attribute::operator=(bool rhs);
xml_attribute& xml_attribute::operator=(long long rhs);
xml_attribute& xml_attribute::operator=(unsigned long
long rhs);
```

These operators simply call the right `set_value` function and return the attribute they're called on; the return value of `set_value` is ignored, so errors are ignored.

This is an example of setting attribute name and value (samples/modify_base.cpp):

```
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;
```

6.3. Adding nodes/attributes

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:

```
xml_attribute xml_node::append_attribute(const char_t*
name);
xml_attribute xml_node::prepend_attribute(const char_t*
name);
xml_attribute xml_node::insert_attribute_after(const
char_t* name, const xml_attribute& attr);
xml_attribute xml_node::insert_attribute_before(const
char_t* name, const xml_attribute& attr);

xml_node xml_node::append_child(xml_node_type type =
node_element);
xml_node xml_node::prepend_child(xml_node_type type =
node_element);
xml_node xml_node::insert_child_after(xml_node_type type,
const xml_node& node);
xml_node xml_node::insert_child_before(xml_node_type
type, const xml_node& node);

xml_node xml_node::append_child(const char_t* name);
xml_node xml_node::prepend_child(const char_t* name);
xml_node xml_node::insert_child_after(const char_t* name,
const xml_node& node);
xml_node xml_node::insert_child_before(const char_t*
name, const xml_node& node);
```

`append_attribute` and `append_child` create a new

node/attribute at the end of the corresponding list of the node the method is called on; `prepend_attribute` and `prepend_child` create a new node/attribute at the beginning of the list; `insert_attribute_after`, `insert_attribute_before`, `insert_child_after` and `insert_attribute_before` add the node/attribute before or after the specified node/attribute.

Attribute functions create an attribute with the specified name; you can specify the empty name and change the name later if you want to. Node functions with the `type` argument create the node with the specified type; since node type can't be changed, you have to know the desired type beforehand. Also note that not all types can be added as children; see below for clarification. Node functions with the `name` argument create the element node (node_element) with the specified name.

All functions return the handle to the created object on success, and null handle on failure. There are several reasons for failure:

- Adding fails if the target node is null;
- Only node_element nodes can contain attributes, so attribute

adding fails if node is not an element;

- Only node_document and node_element nodes can contain children, so child node adding fails if the target node is not an element or a document;
- node_document and node_null nodes can not be inserted as children, so passing node_document or node_null value as `type` results in operation failure;
- node_declaration nodes can only be added as children of the document node; attempt to insert declaration node as a child of an element node fails;
- Adding node/attribute results in memory allocation, which may fail;
- Insertion functions fail if the specified node or attribute is null or is not in the target node's children/attribute list.

Even if the operation fails, the document remains in consistent state, but the requested node/attribute is not added.

> `attribute()` and `child()` functions do not add

CAUTION

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";
```

6.4. Removing nodes/attributes

If you do not want your document to contain some node or attribute, you can remove it with one of the following functions:

```
bool xml_node::remove_attribute(const xml_attribute& a);
bool xml_node::remove_child(const xml_node& n);
```

`remove_attribute` removes the attribute from the attribute list of the node, and returns the operation result. `remove_child` removes the child node with the entire subtree (including all descendant nodes and attributes) from the document, and returns the operation result. Removing fails if one of the following is true:

- The node the function is called on is null;
- The attribute/node to be removed is null;

- The attribute/node to be removed is not in the node's attribute/child list.

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.

If you want to remove the attribute or child node by its name, two additional helper functions are available:

```
bool xml_node::remove_attribute(const char_t* name);
bool xml_node::remove_child(const char_t* name);
```

These functions look for the first attribute or child with the specified name, and then remove it, returning the result. If there is no attribute or child with such name, the function returns `false`; if there are two nodes with the given name, only the first node is deleted. If you want to delete all nodes with the specified

name, you can use code like this: `while (node.remove_child("tool")) ;`.

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);
```

6.5. Working with text contents

pugixml provides a special class, `xml_text`, to work with text contents stored as a value of some node, i.e. `<node><description>This is a`

`node</description></node>`. Working with text objects to retrieve data is described in the documentation for accessing document data; this section describes the modification interface of `xml_text`.

Once you have an `xml_text` object, you can set the text contents using the following function:

```
bool xml_text::set(const char_t* rhs);
```

This function tries to set the contents to the specified string, and returns the operation result. The operation fails if the text object was retrieved from a node that can not have a value and is not an element node (i.e. it is a node_declaration node), if the text object is empty, or if there is insufficient memory to handle the request. The provided string is copied into document managed memory and can be destroyed after the function returns (for example, you can safely pass stack-allocated buffers to this function). Note that if the text object was retrieved from an element node, this function creates the PCDATA child node if necessary (i.e. if the element node does not have a PCDATA/CDATA child already).

In addition to a string function, several functions are provided for handling text with numbers and booleans as contents:

```
bool xml_text::set(int rhs);
bool xml_text::set(unsigned int rhs);
bool xml_text::set(double rhs);
bool xml_text::set(float rhs);
bool xml_text::set(bool rhs);
bool xml_text::set(long long rhs);
bool xml_text::set(unsigned long long rhs);
```

The above functions convert the argument to string and then call the base `set` function. These functions have the same semantics as similar `xml_attribute` functions. You can refer to documentation for the attribute functions for details.

For convenience, all `set` functions have the corresponding assignment operators:

```
xml_text& xml_text::operator=(const char_t* rhs);
xml_text& xml_text::operator=(int rhs);
xml_text& xml_text::operator=(unsigned int rhs);
```

```
xml_text& xml_text::operator=(double rhs);
xml_text& xml_text::operator=(float rhs);
xml_text& xml_text::operator=(bool rhs);
xml_text& xml_text::operator=(long long rhs);
xml_text& xml_text::operator=(unsigned long long rhs);
```

These operators simply call the right `set` function and return the attribute they're called on; the return value of `set` is ignored, so errors are ignored.

This is an example of using `xml_text` object to modify text contents (samples/text.cpp):

```
// change project version
project.child("version").text() = 1.2;

// add description element and set the contents
// note that we do not have to explicitly add the
node_pcdata child
project.append_child("description").text().set("a test
project");
```

6.6. Cloning nodes/attributes

With the help of previously described functions, it is possible to create trees with any contents and structure, including cloning the existing data. However since this is an often needed operation, pugixml provides built-in node/attribute cloning facilities. Since nodes and attributes do not exist without a document tree, you can't create a standalone copy - you have to immediately insert it somewhere in the tree. For this, you can use one of the following functions:

```
xml_attribute xml_node::append_copy(const xml_attribute&
proto);
xml_attribute xml_node::prepend_copy(const xml_attribute&
proto);
xml_attribute xml_node::insert_copy_after(const
xml_attribute& proto, const xml_attribute& attr);
xml_attribute xml_node::insert_copy_before(const
xml_attribute& proto, const xml_attribute& attr);

xml_node xml_node::append_copy(const xml_node& proto);
xml_node xml_node::prepend_copy(const xml_node& proto);
xml_node xml_node::insert_copy_after(const xml_node&
proto, const xml_node& node);
```

```
xml_node xml_node::insert_copy_before(const xml_node&
proto, const xml_node& node);
```

These functions mirror the structure of `append_child`, `prepend_child`, `insert_child_before` and related functions - they take the handle to the prototype object, which is to be cloned, insert a new attribute/node at the appropriate place, and then copy the attribute data or the whole node subtree to the new object. The functions return the handle to the resulting duplicate object, or null handle on failure.

The attribute is copied along with the name and value; the node is copied along with its type, name and value; additionally attribute list and all children are recursively cloned, resulting in the deep subtree clone. The prototype object can be a part of the same document, or a part of any other document.

The failure conditions resemble those of `append_child`, `insert_child_before` and related functions, consult their documentation for more information. There are additional caveats specific to cloning functions:

- Cloning null handles results in operation failure;
- Node cloning starts with insertion of the node of the same type as that of the prototype; for this reason, cloning functions can not be directly used to clone entire documents, since node_document is not a valid insertion type. The example below provides a workaround.
- It is possible to copy a subtree as a child of some node inside this subtree, i.e. `node.append_copy(node.parent().parent());`. This is a valid operation, and it results in a clone of the subtree in the state before cloning started, i.e. no infinite recursion takes place.

This is an example with one possible implementation of include tags in XML (samples/include.cpp). It illustrates node cloning and usage of other document modification functions:

```
bool load_preprocess(pugi::xml_document& doc, const char*
path);

bool preprocess(pugi::xml_node node)
```

```
{
    for (pugi::xml_node child = node.first_child();
child; )
    {
        if (child.type() == pugi::node_pi &&
strcmp(child.name(), "include") == 0)
        {
            pugi::xml_node include = child;

            // load new preprocessed document (note:
ideally this should handle relative paths)
            const char* path = include.value();

            pugi::xml_document doc;
            if (!load_preprocess(doc, path)) return
false;

            // insert the comment marker above include
directive
            node.insert_child_before(pugi::node_comment,
include).set_value(path);

            // copy the document above the include
directive (this retains the original order!)
            for (pugi::xml_node ic = doc.first_child();
ic; ic = ic.next_sibling())
            {
```

```
                node.insert_copy_before(ic, include);
            }

            // remove the include node and move to the
next child
            child = child.next_sibling();

            node.remove_child(include);
        }
        else
        {
            if (!preprocess(child)) return false;

            child = child.next_sibling();
        }
    }

    return true;
}

bool load_preprocess(pugi::xml_document& doc, const char*
path)
{
    pugi::xml_parse_result result = doc.load_file(path,
pugi::parse_default | pugi::parse_pi); // for <?include?>

    return result ? preprocess(doc) : false;
```

```
}
```

6.7. Moving nodes

Sometimes instead of cloning a node you need to move an existing node to a different position in a tree. This can be accomplished by copying the node and removing the original; however, this is expensive since it results in a lot of extra operations. For moving nodes within the same document tree, you can use of the following functions instead:

```
xml_node xml_node::append_move(const xml_node& moved);
xml_node xml_node::prepend_move(const xml_node& moved);
xml_node xml_node::insert_move_after(const xml_node&
moved, const xml_node& node);
xml_node xml_node::insert_move_before(const xml_node&
moved, const xml_node& node);
```

These functions mirror the structure of `append_copy`, `prepend_copy`, `insert_copy_before` and `insert_copy_after` - they take the handle to the moved object and move it to the appropriate place with all attributes and/or child nodes. The

functions return the handle to the resulting object (which is the same as the moved object), or null handle on failure.

The failure conditions resemble those of `append_child`, `insert_child_before` and related functions, consult their documentation for more information. There are additional caveats specific to moving functions:

- Moving null handles results in operation failure;
- Moving is only possible for nodes that belong to the same document; attempting to move nodes between documents will fail.
- `insert_move_after` and `insert_move_before` functions fail if the moved node is the same as the `node` argument (this operation would be a no-op otherwise).
- It is impossible to move a subtree to a child of some node inside this subtree, i.e. `node.append_move(node.parent().parent());` will fail.

6.8. Assembling document from fragments

pugixml provides several ways to assemble an XML document from other XML documents. Assuming there is a set of document fragments, represented as in-memory buffers, the implementation choices are as follows:

- Use a temporary document to parse the data from a string, then clone the nodes to a destination node. For example:

```
bool append_fragment(pugi::xml_node target, const
char* buffer, size_t size)
{
    pugi::xml_document doc;
    if (!doc.load_buffer(buffer, size)) return false;

    for (pugi::xml_node child = doc.first_child();
child; child = child.next_sibling())
        target.append_copy(child);
}
```

- Cache the parsing step - instead of keeping in-memory buffers, keep document objects that already contain the parsed fragment:

```
bool append_fragment(pugi::xml_node target, const
pugi::xml_document& cached_fragment)
{
    for (pugi::xml_node child =
cached_fragment.first_child(); child; child =
child.next_sibling())
        target.append_copy(child);
}
```

- Use `xml_node::append_buffer` directly:

```
xml_parse_result xml_node::append_buffer(const void*
contents, size_t size, unsigned int options =
parse_default, xml_encoding encoding = encoding_auto);
```

The first method is more convenient, but slower than the other two. The relative performance of `append_copy` and `append_buffer` depends on the buffer format - usually `append_buffer` is faster if the buffer is in native encoding (UTF-8 or wchar_t, depending on `PUGIXML_WCHAR_MODE`). At the same time it might be less efficient in terms of memory usage - the implementation makes a copy of the provided buffer, and the copy has the same lifetime as the document - the memory used by

that copy will be reclaimed after the document is destroyed, but no sooner. Even deleting all nodes in the document, including the appended ones, won't reclaim the memory.

`append_buffer` behaves in the same way as xml_document::load_buffer - the input buffer is a byte buffer, with size in bytes; the buffer is not modified and can be freed after the function returns.

Since `append_buffer` needs to append child nodes to the current node, it only works if the current node is either document or element node. Calling `append_buffer` on a node with any other type results in an error with `status_append_invalid_root` status.

---

7. 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 (see Output options), and also perform necessary encoding conversions (see Encodings). This section documents the relevant functionality.

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 (unless format_no_escapes flag is set). 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.

CDATA sections with values that contain `"]]>"` are split into several sections as follows: section with value `"pre]]>post"` is written as `<![CDATA[pre]]]><![CDATA[>post]]>`. While this alters the structure of the document (if you load the document

after saving it, there will be two CDATA sections instead of one), this is the only way to escape CDATA contents.

7.1. Saving document to a file

If you want to save the whole document to a file, you can use one of the following functions:

```
bool xml_document::save_file(const char* path, const
char_t* indent = "\t", unsigned int flags =
format_default, xml_encoding encoding = encoding_auto)
const;
bool xml_document::save_file(const wchar_t* path, const
char_t* indent = "\t", unsigned int flags =
format_default, xml_encoding encoding = encoding_auto)
const;
```

These functions accept file path as its first argument, and also three optional arguments, which specify indentation and other output options (see Output options) and output data encoding (see Encodings). The path has the target operating system format, so it can be a relative or absolute one, it should have the delimiters of

the target system, it should have the exact case if the target file system is case-sensitive, etc.

File path is passed to the system file opening function as is in case of the first function (which accepts `const char* path`); the second function either uses a special file opening function if it is provided by the runtime library or converts the path to UTF-8 and uses the system file opening function.

`save_file` opens the target file for writing, outputs the requested header (by default a document declaration is output, unless the document already has one), and then saves the document contents. If the file could not be opened, the function returns `false`. Calling `save_file` is equivalent to creating an `xml_writer_file` object with `FILE*` handle as the only constructor argument and then calling `save`; see Saving document via writer interface for writer interface details.

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;
```

7.2. Saving document to C++ IOstreams

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:

```
void xml_document::save(std::ostream& stream, const
char_t* indent = "\t", unsigned int flags =
format_default, xml_encoding encoding = encoding_auto)
const;
void xml_document::save(std::wostream& stream, const
char_t* indent = "\t", unsigned int flags =
format_default) const;
```

`save` with `std::ostream` argument saves the document to the stream in the same way as `save_file` (i.e. with requested header and with encoding conversions). On the other hand, `save` with `std::wstream` argument saves the document to the wide stream with encoding_wchar encoding. Because of this, using `save` with wide character streams requires careful (usually platform-specific) stream setup (i.e. using the `imbue` function). Generally use of wide streams is discouraged, however it provides you with the ability to save documents to non-Unicode encodings, i.e. you can save Shift-JIS encoded data if you set the correct locale.

Calling `save` with stream target is equivalent to creating an `xml_writer_stream` object with stream as the only constructor argument and then calling `save`; see Saving document via writer interface for writer interface details.

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);
```

7.3. Saving document via writer interface

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:

```
class xml_writer
{
public:
    virtual void write(const void* data, size_t size) =
0;
};

void xml_document::save(xml_writer& writer, const char_t*
indent = "\t", unsigned int flags = format_default,
xml_encoding encoding = encoding_auto) const;
```

In order to output the document via some custom transport, for example sockets, you should create an object which implements

`xml_writer` interface and pass it to `save` function. `xml_writer::write` function is called with a buffer as an input, where `data` points to buffer start, and `size` is equal to the buffer size in bytes. `write` implementation must write the buffer to the transport; it can not save the passed buffer pointer, as the buffer contents will change after `write` returns. The buffer contains the chunk of document data in the desired encoding.

`write` function is called with relatively large blocks (size is usually several kilobytes, except for the last block that may be small), so there is often no need for additional buffering in the implementation.

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);
    }
};
```

7.4. Saving a single subtree

While the previously described functions save the whole document to the destination, it is easy to save a single subtree. The following functions are provided:

```
void xml_node::print(std::ostream& os, const char_t*
indent = "\t", unsigned int flags = format_default,
xml_encoding encoding = encoding_auto, unsigned int depth
= 0) const;
void xml_node::print(std::wostream& os, const char_t*
indent = "\t", unsigned int flags = format_default,
unsigned int depth = 0) const;
void xml_node::print(xml_writer& writer, const char_t*
indent = "\t", unsigned int flags = format_default,
xml_encoding encoding = encoding_auto, unsigned int depth
= 0) const;
```

These functions have the same arguments with the same meaning as the corresponding `xml_document::save` functions, and allow you to save the subtree to either a C++ IOstream or to any object that implements `xml_writer` interface.

Saving a subtree differs from saving the whole document: the process behaves as if format_write_bom is off, and format_no_declaration is on, even if actual values of the flags are different. This means that BOM is not written to the destination, and document declaration is only written if it is the node itself or is one of node's children. Note that this also holds if you're saving a document; this example (samples/save_subtree.cpp) illustrates the difference:

```
// get a test document
pugi::xml_document doc;
doc.load_string("<foo bar='baz'><call>hey</call></foo>");

// print document to standard output (prints <?xml
version="1.0"?><foo bar="baz"><call>hey</call></foo>)
doc.save(std::cout, "", pugi::format_raw);
std::cout << std::endl;
```

```
// print document to standard output as a regular node
(prints <foo bar="baz"><call>hey</call></foo>)
doc.print(std::cout, "", pugi::format_raw);
std::cout << std::endl;

// print a subtree to standard output (prints
<call>hey</call>)
doc.child("foo").child("call").print(std::cout, "",
pugi::format_raw);
std::cout << std::endl;
```

7.5. Output options

All saving functions accept the optional parameter `flags`. This is a bitmask that customizes the output format; you can select the way the document nodes are printed and select the needed additional information that is output before the document contents.

> **NOTE** You should use the usual bitwise arithmetics to manipulate the bitmask: to enable a flag, use `mask | flag`; to disable a flag, use `mask & ~flag`.

These flags control the resulting tree contents:

- `format_indent` determines if all nodes should be indented with the indentation string (this is an additional parameter for all saving functions, and is `"\t"` by default). If this flag is on, before every node the indentation string is output several times, where the amount of indentation depends on the node's depth relative to the output subtree. This flag has no effect if format_raw is enabled. This flag is**on** by default.
- `format_raw` switches between formatted and raw output. If this flag is on, the nodes are not indented in any way, and also no newlines that are not part of document text are printed. Raw mode can be used for serialization where the result is not intended to be read by humans; also it can be useful if the document was parsed with parse_ws_pcdata flag, to preserve the original document formatting as much as possible. This flag is **off** by default.
- `format_no_escapes` disables output escaping for attribute values and PCDATA contents. If this flag is off, special symbols (`"`, `&`, `<`, `>`) and all non-printable characters (those with

codepoint values less than 32) are converted to XML escape sequences (i.e. `&`) during output. If this flag is on, no text processing is performed; therefore, output XML can be malformed if output contents contains invalid symbols (i.e. having a stray `<` in the PCDATA will make the output malformed). This flag is **off** by default.

These flags control the additional output information:

- `format_no_declaration` disables default node declaration output. By default, if the document is saved via `save` or `save_file` function, and it does not have any document declaration, a default declaration is output before the document contents. Enabling this flag disables this declaration. This flag has no effect in `xml_node::print` functions: they never output the default declaration. This flag is **off** by default.
- `format_write_bom` enables Byte Order Mark (BOM) output. By default, no BOM is output, so in case of non UTF-8 encodings the resulting document's encoding may not be recognized by some parsers and text editors, if they do not implement sophisticated encoding detection. Enabling this flag adds an

encoding-specific BOM to the output. This flag has no effect in `xml_node::print` functions: they never output the BOM. This flag is **off** by default.

- `format_save_file_text` changes the file mode when using `save_file` function. By default, file is opened in binary mode, which means that the output file willcontain platform-independent newline `\n` (ASCII 10). If this flag is on, file is opened in text mode, which on some systems changes the newline format (i.e. on Windows you can use this flag to output XML documents with `\r\n` (ASCII 13 10) newlines. This flag is **off** by default.

Additionally, there is one predefined option mask:

- `format_default` is the default set of flags, i.e. it has all options set to their default values. It sets formatted output with indentation, without BOM and with default node declaration, if necessary.

This is an example that shows the outputs of different output options (samples/save_options.cpp):

```
// get a test document
pugi::xml_document doc;
doc.load_string("<foo bar='baz'><call>hey</call></foo>");

// default options; prints
// <?xml version="1.0"?>
// <foo bar="baz">
//         <call>hey</call>
// </foo>
doc.save(std::cout);
std::cout << std::endl;

// default options with custom indentation string; prints
// <?xml version="1.0"?>
// <foo bar="baz">
// --<call>hey</call>
// </foo>
doc.save(std::cout, "--");
std::cout << std::endl;

// default options without indentation; prints
// <?xml version="1.0"?>
// <foo bar="baz">
// <call>hey</call>
// </foo>
doc.save(std::cout, "\t", pugi::format_default &
```

```
~pugi::format_indent); // can also pass "" instead of
indentation string for the same effect
std::cout << std::endl;

// raw output; prints
// <?xml version="1.0"?><foo
bar="baz"><call>hey</call></foo>
doc.save(std::cout, "\t", pugi::format_raw);
std::cout << std::endl << std::endl;

// raw output without declaration; prints
// <foo bar="baz"><call>hey</call></foo>
doc.save(std::cout, "\t", pugi::format_raw |
pugi::format_no_declaration);
std::cout << std::endl;
```

7.6. Encodings

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 during output. The output encoding is set via the `encoding` parameter of saving functions, which is of type `xml_encoding`. The possible values for the

encoding are documented in Encodings; the only flag that has a different meaning is `encoding_auto`.

While all other flags set the exact encoding, `encoding_auto` is meant for automatic encoding detection. The automatic detection does not make sense for output encoding, since there is usually nothing to infer the actual encoding from, so here `encoding_auto` means UTF-8 encoding, which is the most popular encoding for XML data storage. This is also the default value of output encoding; specify another value if you do not want UTF-8 encoded output.

Also note that wide stream saving functions do not have `encoding` argument and always assume encoding_wchar encoding.

> **NOTE**
>
> The current behavior for Unicode conversion is to skip all invalid UTF sequences during conversion. This behavior should not be relied upon; if your node/attribute names do not contain any valid UTF sequences, they may be output as if they are empty,

which will result in malformed XML document.

7.7. Customizing document declaration

When you are saving the document using `xml_document::save()` or `xml_document::save_file()`, a default XML document declaration is output, if `format_no_declaration` is not specified and if the document does not have a declaration node. However, the default declaration is not customizable. If you want to customize the declaration output, you need to create the declaration node yourself.

NOTE

By default the declaration node is not added to the document during parsing. If you just need to preserve the original declaration node, you have to add the flag parse_declaration to the parsing flags; the resulting document will contain the original declaration node, which will be output during saving.

Declaration node is a node with typenode_declaration; it behaves like an element node in that it has attributes with values (but it does not have child nodes). Therefore setting custom version, encoding or standalone declaration involves adding attributes and setting attribute values.

This is an example that shows how to create a custom declaration node (samples/save_declaration.cpp):

```
// get a test document
pugi::xml_document doc;
doc.load_string("<foo bar='baz'><call>hey</call></foo>");

// add a custom declaration node
pugi::xml_node decl =
doc.prepend_child(pugi::node_declaration);
decl.append_attribute("version") = "1.0";
decl.append_attribute("encoding") = "UTF-8";
decl.append_attribute("standalone") = "no";

// <?xml version="1.0" encoding="UTF-8" standalone="no"?>
// <foo bar="baz">
//         <call>hey</call>
// </foo>
```

```
doc.save(std::cout);
std::cout << std::endl;
```

8. XPath

If the task at hand is to select a subset of document nodes that match some criteria, it is possible to code a function using the existing traversal functionality for any practical criteria. However, often either a data-driven approach is desirable, in case the criteria are not predefined and come from a file, or it is inconvenient to use traversal interfaces and a higher-level DSL is required. There is a standard language for XML processing, XPath, that can be useful for these cases. pugixml implements an almost complete subset of XPath 1.0. Because of differences in document object model and some performance implications, there are minor violations of the official specifications, which can be found in Conformance to W3C specification. The rest of this section describes the interface for XPath functionality. Please note that if

you wish to learn to use XPath language, you have to look for other tutorials or manuals; for example, you can read W3Schools XPath tutorial, XPath tutorial at tizag.com, and the XPath 1.0 specification.

8.1. XPath types

Each XPath expression can have one of the following types: boolean, number, string or node set. Boolean type corresponds to `bool` type, number type corresponds to `double` type, string type corresponds to either `std::string` or `std::wstring`, depending on whether wide character interface is enabled, and node set corresponds to xpath_node_set type. There is an enumeration, `xpath_value_type`, which can take the values `xpath_type_boolean`, `xpath_type_number`, `xpath_type_string` or `xpath_type_node_set`, accordingly.

Because an XPath node can be either a node or an attribute, there is a special type, `xpath_node`, which is a discriminated union of these types. A value of this type contains two node handles, one of `xml_node` type, and another one of `xml_attribute` type; at most

one of them can be non-null. The accessors to get these handles are available:

```
xml_node xpath_node::node() const;
xml_attribute xpath_node::attribute() const;
```

XPath nodes can be null, in which case both accessors return null handles.

Note that as per XPath specification, each XPath node has a parent, which can be retrieved via this function:

```
xml_node xpath_node::parent() const;
```

`parent` function returns the node's parent if the XPath node corresponds to `xml_node` handle (equivalent to `node().parent()`), or the node to which the attribute belongs to, if the XPath node corresponds to `xml_attribute` handle. For null nodes, `parent` returns null handle.

Like node and attribute handles, XPath node handles can be

implicitly cast to boolean-like object to check if it is a null node, and also can be compared for equality with each other.

You can also create XPath nodes with one of the three constructors: the default constructor, the constructor that takes node argument, and the constructor that takes attribute and node arguments (in which case the attribute must belong to the attribute list of the node). The constructor from `xml_node` is implicit, so you can usually pass `xml_node` to functions that expect `xpath_node`. Apart from that you usually don't need to create your own XPath node objects, since they are returned to you via selection functions.

XPath expressions operate not on single nodes, but instead on node sets. A node set is a collection of nodes, which can be optionally ordered in either a forward document order or a reverse one. Document order is defined in XPath specification; an XPath node is before another node in document order if it appears before it in XML representation of the corresponding document.

Node sets are represented by `xpath_node_set` object, which has an interface that resembles one of sequential random-access

containers. It has an iterator type along with usual begin/past-the-end iterator accessors:

```
typedef const xpath_node* xpath_node_set::const_iterator;
const_iterator xpath_node_set::begin() const;
const_iterator xpath_node_set::end() const;
```

And it also can be iterated via indices, just like `std::vector`:

```
const xpath_node& xpath_node_set::operator[](size_t
index) const;
size_t xpath_node_set::size() const;
bool xpath_node_set::empty() const;
```

All of the above operations have the same semantics as that of `std::vector`: the iterators are random-access, all of the above operations are constant time, and accessing the element at index that is greater or equal than the set size results in undefined behavior. You can use both iterator-based and index-based access for iteration, however the iterator-based one can be faster.

The order of iteration depends on the order of nodes inside the set; the order can be queried via the following function:

```
enum xpath_node_set::type_t {type_unsorted, type_sorted,
type_sorted_reverse};
type_t xpath_node_set::type() const;
```

`type` function returns the current order of nodes; `type_sorted` means that the nodes are in forward document order, `type_sorted_reverse` means that the nodes are in reverse document order, and `type_unsorted` means that neither order is guaranteed (nodes can accidentally be in a sorted order even if `type()` returns `type_unsorted`). If you require a specific order of iteration, you can change it via `sort` function:

```
void xpath_node_set::sort(bool reverse = false);
```

Calling `sort` sorts the nodes in either forward or reverse document order, depending on the argument; after this call `type()` will return `type_sorted` or `type_sorted_reverse`.

Often the actual iteration is not needed; instead, only the first element in document order is required. For this, a special accessor is provided:

```
xpath_node xpath_node_set::first() const;
```

This function returns the first node in forward document order from the set, or null node if the set is empty. Note that while the result of the node does not depend on the order of nodes in the set (i.e. on the result of `type()`), the complexity does - if the set is sorted, the complexity is constant, otherwise it is linear in the number of elements or worse.

While in the majority of cases the node set is returned by XPath functions, sometimes there is a need to manually construct a node set. For such cases, a constructor is provided which takes an iterator range (`const_iterator` is a typedef for `const xpath_node*`), and an optional type:

```
xpath_node_set::xpath_node_set(const_iterator begin,
const_iterator end, type_t type = type_unsorted);
```

The constructor copies the specified range and sets the specified type. The objects in the range are not checked in any way; you'll have to ensure that the range contains no duplicates, and that the objects are sorted according to the `type` parameter. Otherwise XPath operations with this set may produce unexpected results.

8.2. Selecting nodes via XPath expression

If you want to select nodes that match some XPath expression, you can do it with the following functions:

```
xpath_node xml_node::select_node(const char_t* query,
xpath_variable_set* variables = 0) const;
xpath_node_set xml_node::select_nodes(const char_t*
query, xpath_variable_set* variables = 0) const;
```

`select_nodes` function compiles the expression and then executes it with the node as a context node, and returns the resulting node set. `select_node` returns only the first node in document order from the result, and is equivalent to calling

`select_nodes(query).first()`. If the XPath expression does not match anything, or the node handle is null, `select_nodes` returns an empty set, and `select_node` returns null XPath node.

If exception handling is not disabled, both functions throw xpath_exception if the query can not be compiled or if it returns a value with type other than node set; see Error handling for details.

While compiling expressions is fast, the compilation time can introduce a significant overhead if the same expression is used many times on small subtrees. If you're doing many similar queries, consider compiling them into query objects (see Using query objects for further reference). Once you get a compiled query object, you can pass it to select functions instead of an expression string:

```
xpath_node xml_node::select_node(const xpath_query&
query) const;
xpath_node_set xml_node::select_nodes(const xpath_query&
query) const;
```

If exception handling is not disabled, both functions throw

xpath_exception if the query returns a value with type other than node set.

This is an example of selecting nodes using XPath expressions (samples/xpath_select.cpp):

```
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";
```

8.3. Using query objects

When you call `select_nodes` with an expression string as an argument, a query object is created behind the scenes. A query object represents a compiled XPath expression. Query objects can be needed in the following circumstances:

- You can precompile expressions to query objects to save compilation time if it becomes an issue;
- You can use query objects to evaluate XPath expressions which result in booleans, numbers or strings;
- You can get the type of expression value via query object.

Query objects correspond to `xpath_query` type. They are immutable and non-copyable: they are bound to the expression at creation time and can not be cloned. If you want to put query objects in a container, allocate them on heap via `new` operator and store pointers to `xpath_query` in the container.

You can create a query object with the constructor that takes XPath expression as an argument:

```
explicit xpath_query::xpath_query(const char_t* query,
xpath_variable_set* variables = 0);
```

The expression is compiled and the compiled representation is stored in the new query object. If compilation fails, xpath_exception is thrown if exception handling is not disabled (see Error handling for details). After the query is created, you can query the type of the evaluation result using the following function:

```
xpath_value_type xpath_query::return_type() const;
```

You can evaluate the query using one of the following functions:

```
bool xpath_query::evaluate_boolean(const xpath_node& n)
const;
double xpath_query::evaluate_number(const xpath_node& n)
```

```
const;
string_t xpath_query::evaluate_string(const xpath_node&
n) const;
xpath_node_set xpath_query::evaluate_node_set(const
xpath_node& n) const;
xpath_node xpath_query::evaluate_node(const xpath_node&
n) const;
```

All functions take the context node as an argument, compute the expression and return the result, converted to the requested type. According to XPath specification, value of any type can be converted to boolean, number or string value, but no type other than node set can be converted to node set. Because of this, `evaluate_boolean`, `evaluate_number` and `evaluate_string` always return a result, but `evaluate_node_set` and `evaluate_node` result in an error if the return type is not node set (see Error handling).

> **NOTE**
>
> Calling `node.select_nodes("query")` is equivalent to calling `xpath_query("query").evaluate_node_set(node)`. Calling `node.select_node("query")` is equivalent to

> calling `xpath_query("query").evaluate_node(node)`.

Note that `evaluate_string` function returns the STL string; as such, it's not available in PUGIXML_NO_STL mode and also usually allocates memory. There is another string evaluation function:

```
size_t xpath_query::evaluate_string(char_t* buffer,
size_t capacity, const xpath_node& n) const;
```

This function evaluates the string, and then writes the result to `buffer` (but at most `capacity` characters); then it returns the full size of the result in characters, including the terminating zero. If `capacity` is not 0, the resulting buffer is always zero-terminated. You can use this function as follows:

- First call the function with `buffer = 0` and `capacity = 0`; then allocate the returned amount of characters, and call the function again, passing the allocated storage and the amount of characters;

- First call the function with small buffer and buffer capacity; then, if the result is larger than the capacity, the output has been trimmed, so allocate a larger buffer and call the function again.

This is an example of using query objects (samples/xpath_query.cpp):

```
// Select nodes via compiled query
pugi::xpath_query
query_remote_tools("/Profile/Tools/Tool[@AllowRemote='tru
e']");

pugi::xpath_node_set tools =
query_remote_tools.evaluate_node_set(doc);
std::cout << "Remote tool: ";
tools[2].node().print(std::cout);

// Evaluate numbers via compiled query
pugi::xpath_query query_timeouts("sum(//Tool/@Timeout)");
std::cout << query_timeouts.evaluate_number(doc) <<
std::endl;

// Evaluate strings via compiled query for different
```

```
context nodes
pugi::xpath_query query_name_valid("string-
length(substring-before(@Filename, '_')) > 0 and
@OutputFileMasks");
pugi::xpath_query query_name("concat(substring-
before(@Filename, '_'), ' produces ',
@OutputFileMasks)");

for (pugi::xml_node tool =
doc.first_element_by_path("Profile/Tools/Tool"); tool;
tool = tool.next_sibling())
{
    std::string s = query_name.evaluate_string(tool);

    if (query_name_valid.evaluate_boolean(tool))
std::cout << s << std::endl;
}
```

8.4. Using variables

XPath queries may contain references to variables; this is useful if you want to use queries that depend on some dynamic parameter without manually preparing the complete query string, or if you want to reuse the same query object for similar queries.

Variable references have the form `$name`; in order to use them, you have to provide a variable set, which includes all variables present in the query with correct types. This set is passed to `xpath_query` constructor or to `select_nodes`/`select_node` functions:

```
explicit xpath_query::xpath_query(const char_t* query,
xpath_variable_set* variables = 0);
xpath_node xml_node::select_node(const char_t* query,
xpath_variable_set* variables = 0) const;
xpath_node_set xml_node::select_nodes(const char_t*
query, xpath_variable_set* variables = 0) const;
```

If you're using query objects, you can change the variable values before `evaluate`/`select` calls to change the query behavior.

> **NOTE** The variable set pointer is stored in the query object; you have to ensure that the lifetime of the set exceeds that of query object.

Variable sets correspond to `xpath_variable_set` type, which is

essentially a variable container.

You can add new variables with the following function:

```
xpath_variable* xpath_variable_set::add(const char_t*
name, xpath_value_type type);
```

The function tries to add a new variable with the specified name and type; if the variable with such name does not exist in the set, the function adds a new variable and returns the variable handle; if there is already a variable with the specified name, the function returns the variable handle if variable has the specified type. Otherwise the function returns null pointer; it also returns null pointer on allocation failure.

New variables are assigned the default value which depends on the type: `0` for numbers, `false` for booleans, empty string for strings and empty set for node sets.

You can get the existing variables with the following functions:

```
xpath_variable* xpath_variable_set::get(const char_t*
name);
const xpath_variable* xpath_variable_set::get(const
char_t* name) const;
```

The functions return the variable handle, or null pointer if the variable with the specified name is not found.

Additionally, there are the helper functions for setting the variable value by name; they try to add the variable with the corresponding type, if it does not exist, and to set the value. If the variable with the same name but with different type is already present, they return `false`; they also return `false` on allocation failure. Note that these functions do not perform any type conversions.

```
bool xpath_variable_set::set(const char_t* name, bool
value);
bool xpath_variable_set::set(const char_t* name, double
value);
bool xpath_variable_set::set(const char_t* name, const
char_t* value);
bool xpath_variable_set::set(const char_t* name, const
```

```
xpath_node_set& value);
```

The variable values are copied to the internal variable storage, so you can modify or destroy them after the functions return.

If setting variables by name is not efficient enough, or if you have to inspect variable information or get variable values, you can use variable handles. A variable corresponds to the `xpath_variable` type, and a variable handle is simply a pointer to `xpath_variable`.

In order to get variable information, you can use one of the following functions:

```
const char_t* xpath_variable::name() const;
xpath_value_type xpath_variable::type() const;
```

Note that each variable has a distinct type which is specified upon variable creation and can not be changed later.

In order to get variable value, you should use one of the following

functions, depending on the variable type:

```
bool xpath_variable::get_boolean() const;
double xpath_variable::get_number() const;
const char_t* xpath_variable::get_string() const;
const xpath_node_set& xpath_variable::get_node_set()
const;
```

These functions return the value of the variable. Note that no type conversions are performed; if the type mismatch occurs, a dummy value is returned (`false` for booleans, `NaN` for numbers, empty string for strings and empty set for node sets).

In order to set variable value, you should use one of the following functions, depending on the variable type:

```
bool xpath_variable::set(bool value);
bool xpath_variable::set(double value);
bool xpath_variable::set(const char_t* value);
bool xpath_variable::set(const xpath_node_set& value);
```

These functions modify the variable value. Note that no type

conversions are performed; if the type mismatch occurs, the functions return `false`; they also return `false` on allocation failure. The variable values are copied to the internal variable storage, so you can modify or destroy them after the functions return.

This is an example of using variables in XPath queries (samples/xpath_variables.cpp):

```
// Select nodes via compiled query
pugi::xpath_variable_set vars;
vars.add("remote", pugi::xpath_type_boolean);

pugi::xpath_query
query_remote_tools("/Profile/Tools/Tool[@AllowRemote =
string($remote)]", &vars);

vars.set("remote", true);
pugi::xpath_node_set tools_remote =
query_remote_tools.evaluate_node_set(doc);

vars.set("remote", false);
pugi::xpath_node_set tools_local =
query_remote_tools.evaluate_node_set(doc);
```

```
std::cout << "Remote tool: ";
tools_remote[2].node().print(std::cout);

std::cout << "Local tool: ";
tools_local[0].node().print(std::cout);

// You can pass the context directly to
select_nodes/select_node
pugi::xpath_node_set tools_local_imm =
doc.select_nodes("/Profile/Tools/Tool[@AllowRemote =
string($remote)]", &vars);

std::cout << "Local tool imm: ";
tools_local_imm[0].node().print(std::cout);
```

8.5. Error handling

There are two different mechanisms for error handling in XPath implementation; the mechanism used depends on whether exception support is disabled (this is controlled with PUGIXML_NO_EXCEPTIONS define).

By default, XPath functions throw `xpath_exception` object in

case of errors; additionally, in the event any memory allocation fails, an `std::bad_alloc` exception is thrown. Also `xpath_exception` is thrown if the query is evaluated to a node set, but the return type is not node set. If the query constructor succeeds (i.e. no exception is thrown), the query object is valid. Otherwise you can get the error details via one of the following functions:

```
virtual const char* xpath_exception::what() const
throw();
const xpath_parse_result& xpath_exception::result()
const;
```

If exceptions are disabled, then in the event of parsing failure the query is initialized to invalid state; you can test if the query object is valid by using it in a boolean expression: `if (query) { … }`. Additionally, you can get parsing result via the result() accessor:

```
const xpath_parse_result& xpath_query::result() const;
```

Without exceptions, evaluating invalid query results in `false`,

empty string, `NaN` or an empty node set, depending on the type; evaluating a query as a node set results in an empty node set if the return type is not node set.

The information about parsing result is returned via `xpath_parse_result` object. It contains parsing status and the offset of last successfully parsed character from the beginning of the source stream:

```
struct xpath_parse_result
{
    const char* error;
    ptrdiff_t offset;

    operator bool() const;
    const char* description() const;
};
```

Parsing result is represented as the error message; it is either a null pointer, in case there is no error, or the error message in the form of ASCII zero-terminated string.

`description()` member function can be used to get the error message; it never returns the null pointer, so you can safely use `description()` even if query parsing succeeded. Note that `description()` returns a `char` string even in `PUGIXML_WCHAR_MODE`; you'll have to call as_wide to get the `wchar_t` string.

In addition to the error message, parsing result has an `offset` member, which contains the offset of last successfully parsed character. This offset is in units of pugi::char_t (bytes for character mode, wide characters for wide character mode).

Parsing result object can be implicitly converted to `bool` like this: `if (result) { … } else { … }`.

This is an example of XPath error handling (samples/xpath_error.cpp):

```
// Exception is thrown for incorrect query syntax
try
{
    doc.select_nodes("//nodes[#true()]");
```

```
}
catch (const pugi::xpath_exception& e)
{
    std::cout << "Select failed: " << e.what() <<
std::endl;
}

// Exception is thrown for incorrect query semantics
try
{
    doc.select_nodes("(123)/next");
}
catch (const pugi::xpath_exception& e)
{
    std::cout << "Select failed: " << e.what() <<
std::endl;
}

// Exception is thrown for query with incorrect return
type
try
{
    doc.select_nodes("123");
}
catch (const pugi::xpath_exception& e)
{
    std::cout << "Select failed: " << e.what() <<
```

```
std::endl;
}
```

8.6. Conformance to W3C specification

Because of the differences in document object models, performance considerations and implementation complexity, pugixml does not provide a fully conformant XPath 1.0 implementation. This is the current list of incompatibilities:

- Consecutive text nodes sharing the same parent are not merged, i.e. in `<node>text1 <![CDATA[data]]> text2</node>` node should have one text node child, but instead has three.
- Since the document type declaration is not used for parsing, `id()` function always returns an empty node set.
- Namespace nodes are not supported (affects `namespace::` axis).
- Name tests are performed on QNames in XML document instead of expanded names; for `<foo xmlns:ns1='uri'`

`xmlns:ns2='uri'><ns1:child/><ns2:child/></foo>`, query `foo/ns1:*` will return only the first child, not both of them. Compliant XPath implementations can return both nodes if the user provides appropriate namespace declarations.

- String functions consider a character to be either a single `char` value or a single `wchar_t` value, depending on the library configuration; this means that some string functions are not fully Unicode-aware. This affects `substring()`, `string-length()` and `translate()` functions.

---

9. Changelog

v1.6 10.04.2015

Maintenance release. Changes:

- Specification changes:
  1. Attribute/text values now use more digits when printing

floating point numbers to guarantee round-tripping.

2. Text nodes no longer get extra surrounding whitespace when pretty-printing nodes with mixed contents

- Bug fixes:

1. Fixed translate and normalize-space XPath functions to no longer return internal NUL characters
2. Fixed buffer overrun on malformed comments inside DOCTYPE sections
3. DOCTYPE parsing can no longer run out of stack space on malformed inputs (XML parsing is now using bounded stack space)
4. Adjusted processing instruction output to avoid malformed documents if the PI value contains `?>`

v1.5 27.11.2014

Major release, featuring a lot of performance improvements and some new features.

- Specification changes:
  1. `xml_document::load(const char_t*)` was renamed to `load_string`; the old method is still available and will be deprecated in a future release
  2. `xml_node::select_single_node` was renamed to `select_node`; the old method is still available and will be deprecated in a future release.
- New features:
  1. Added `xml_node::append_move` and other functions for moving nodes within a document
  2. Added `xpath_query::evaluate_node` for evaluating queries with a single node as a result
- Performance improvements:
  1. Optimized XML parsing (10-40% faster with clang/gcc, up to 10% faster with MSVC)
  2. Optimized memory consumption when copying nodes in the same document (string contents is now shared)

3. Optimized node copying (10% faster for cross-document copies, 3x faster for inter-document copies; also it now consumes a constant amount of stack space)
4. Optimized node output (60% faster; also it now consumes a constant amount of stack space)
5. Optimized XPath allocation (query evaluation now results in fewer temporary allocations)
6. Optimized XPath sorting (node set sorting is 2-3x faster in some cases)
7. Optimized XPath evaluation (XPathMark suite is 100x faster; some commonly used queries are 3-4x faster)

- Compatibility improvements:
  1. Fixed `xml_node::offset_debug` for corner cases
  2. Fixed undefined behavior while calling memcpy in some cases
  3. Fixed MSVC 2015 compilation warnings
  4. Fixed `contrib/foreach.hpp` for Boost 1.56.0

- Bug fixes

    1. Adjusted comment output to avoid malformed documents if the comment value contains `--`
    2. Fix XPath sorting for documents that were constructed using append_buffer
    3. Fix `load_file` for wide-character paths with non-ASCII characters in MinGW with C++11 mode enabled

v1.4 27.02.2014

Major release, featuring various new features, bug fixes and compatibility improvements.

- Specification changes:

    1. Documents without element nodes are now rejected with `status_no_document_element` error, unless `parse_fragment` option is used

- New features:

    1. Added XML fragment parsing (`parse_fragment` flag)

2. Added PCDATA whitespace trimming (`parse_trim_pcdata` flag)
3. Added long long support for `xml_attribute` and `xml_text` (`as_llong`, `as_ullong` and `set_value`/`set` overloads)
4. Added hexadecimal integer parsing support for `as_int`/`as_uint`/`as_llong`/`as_ullong`
5. Added `xml_node::append_buffer` to improve performance of assembling documents from fragments
6. `xml_named_node_iterator` is now bidirectional
7. Reduced XPath stack consumption during compilation and evaluation (useful for embedded systems)

- Compatibility improvements:
  1. Improved support for platforms without wchar_t support
  2. Fixed several false positives in clang static analysis
  3. Fixed several compilation warnings for various GCC versions

- Bug fixes:
  1. Fixed undefined pointer arithmetic in XPath implementation
  2. Fixed non-seekable iostream support for certain stream types, i.e. Boost `file_source` with pipe input
  3. Fixed `xpath_query::return_type` for some expressions
  4. Fixed dllexport issues with `xml_named_node_iterator`
  5. Fixed `find_child_by_attribute` assertion for attributes with null name/value

v1.2 1.05.2012

Major release, featuring header-only mode, various interface enhancements (i.e. PCDATA manipulation and C++11 iteration), many other features and compatibility improvements.

- New features:
  1. Added `xml_text` helper class for working with PCDATA/CDATA contents of an element node

2. Added optional header-only mode (controlled by `PUGIXML_HEADER_ONLY` define)
3. Added `xml_node::children()` and `xml_node::attributes()` for C++11 ranged for loop or `BOOST_FOREACH`
4. Added support for Latin-1 (ISO-8859-1) encoding conversion during loading and saving
5. Added custom default values for `xml_attribute::as_*` (they are returned if the attribute does not exist)
6. Added `parse_ws_pcdata_single` flag for preserving whitespace-only PCDATA in case it's the only child
7. Added `format_save_file_text` for `xml_document::save_file` to open files as text instead of binary (changes newlines on Windows)
8. Added `format_no_escapes` flag to disable special symbol escaping (complements `~parse_escapes`)
9. Added support for loading document from streams that do not support seeking

10. Added `PUGIXML_MEMORY_*` constants for tweaking allocation behavior (useful for embedded systems)
11. Added `PUGIXML_VERSION` preprocessor define

- Compatibility improvements:
  1. Parser does not require setjmp support (improves compatibility with some embedded platforms, enables `/clr:pure` compilation)
  2. STL forward declarations are no longer used (fixes SunCC/RWSTL compilation, fixes clang compilation in C++11 mode)
  3. Fixed AirPlay SDK, Android, Windows Mobile (WinCE) and C++/CLI compilation
  4. Fixed several compilation warnings for various GCC versions, Intel C++ compiler and Clang
- Bug fixes:
  1. Fixed unsafe bool conversion to avoid problems on C++/CLI
  2. Iterator dereference operator is const now (fixes Boost

`filter_iterator` support)

3. `xml_document::save_file` now checks for file I/O errors during saving

v1.0 1.11.2010

Major release, featuring many XPath enhancements, wide character filename support, miscellaneous performance improvements, bug fixes and more.

- XPath:
  1. XPath implementation is moved to `pugixml.cpp` (which is the only source file now); use `PUGIXML_NO_XPATH` if you want to disable XPath to reduce code size
  2. XPath is now supported without exceptions (`PUGIXML_NO_EXCEPTIONS`); the error handling mechanism depends on the presence of exception support
  3. XPath is now supported without STL (`PUGIXML_NO_STL`)
  4. Introduced variable support

5. Introduced new `xpath_query::evaluate_string`, which works without STL
6. Introduced new `xpath_node_set` constructor (from an iterator range)
7. Evaluation function now accept attribute context nodes
8. All internal allocations use custom allocation functions
9. Improved error reporting; now a last parsed offset is returned together with the parsing error

- Bug fixes:
  1. Fixed memory leak for loading from streams with stream exceptions turned on
  2. Fixed custom deallocation function calling with null pointer in one case
  3. Fixed missing attributes for iterator category functions; all functions/classes can now be DLL-exported
  4. Worked around Digital Mars compiler bug, which lead to minor read overfetches in several functions

5. `load_file` now works with 2+ Gb files in MSVC/MinGW
6. XPath: fixed memory leaks for incorrect queries
7. XPath: fixed `xpath_node()` attribute constructor with empty attribute argument
8. XPath: fixed `lang()` function for non-ASCII arguments

- Specification changes:
  1. CDATA nodes containing `]]>` are printed as several nodes; while this changes the internal structure, this is the only way to escape CDATA contents
  2. Memory allocation errors during parsing now preserve last parsed offset (to give an idea about parsing progress)
  3. If an element node has the only child, and it is of CDATA type, then the extra indentation is omitted (previously this behavior only held for PCDATA children)
- Additional functionality:
  1. Added `xml_parse_result` default constructor
  2. Added `xml_document::load_file` and

`xml_document::save_file` with wide character paths

3. Added `as_utf8` and `as_wide` overloads for `std::wstring` / `std::string` arguments
4. Added DOCTYPE node type (`node_doctype`) and a special parse flag, `parse_doctype`, to add such nodes to the document during parsing
5. Added `parse_full` parse flag mask, which extends `parse_default` with all node type parsing flags except `parse_ws_pcdata`
6. Added `xml_node::hash_value()` and `xml_attribute::hash_value()` functions for use in hash-based containers
7. Added `internal_object()` and additional constructor for both `xml_node` and `xml_attribute` for easier marshalling (useful for language bindings)
8. Added `xml_document::document_element()` function
9. Added `xml_node::prepend_attribute`, `xml_node::prepend_child` and

`xml_node::prepend_copy` functions

10. Added `xml_node::append_child`, `xml_node::prepend_child`, `xml_node::insert_child_before` and `xml_node::insert_child_after` overloads for element nodes (with name instead of type)
11. Added `xml_document::reset()` function

- Performance improvements:
  1. `xml_node::root()` and `xml_node::offset_debug()` are now O(1) instead of O(logN)
  2. Minor parsing optimizations
  3. Minor memory optimization for strings in DOM tree (`set_name`/`set_value`)
  4. Memory optimization for string memory reclaiming in DOM tree (`set_name`/`set_value` now reallocate the buffer if memory waste is too big)
  5. XPath: optimized document order sorting

6. XPath: optimized child/attribute axis step
7. XPath: optimized number-to-string conversions in MSVC
8. XPath: optimized concat for many arguments
9. XPath: optimized evaluation allocation mechanism: constant and document strings are not heap-allocated
10. XPath: optimized evaluation allocation mechanism: all temporaries' allocations use fast stack-like allocator

- Compatibility:
  1. Removed wildcard functions (`xml_node::child_w`, `xml_node::attribute_w`, etc.)
  2. Removed `xml_node::all_elements_by_name`
  3. Removed `xpath_type_t` enumeration; use `xpath_value_type` instead
  4. Removed `format_write_bom_utf8` enumeration; use `format_write_bom` instead
  5. Removed `xml_document::precompute_document_order`, `xml_attribute::document_order` and

`xml_node::document_order` functions; document order sort optimization is now automatic

6. Removed `xml_document::parse` functions and `transfer_ownership` struct; use `xml_document::load_buffer_inplace` and `xml_document::load_buffer_inplace_own` instead
7. Removed `as_utf16` function; use `as_wide` instead

v0.9 1.07.2010

Major release, featuring extended and improved Unicode support, miscellaneous performance improvements, bug fixes and more.

- Major Unicode improvements:
  1. Introduced encoding support (automatic/manual encoding detection on load, manual encoding selection on save, conversion from/to UTF8, UTF16 LE/BE, UTF32 LE/BE)
  2. Introduced `wchar_t` mode (you can set `PUGIXML_WCHAR_MODE` define to switch pugixml internal encoding from UTF8 to `wchar_t`; all functions are switched

to their Unicode variants)

3. Load/save functions now support wide streams

- Bug fixes:

1. Fixed document corruption on failed parsing bug
2. XPath string/number conversion improvements (increased precision, fixed crash for huge numbers)
3. Improved DOCTYPE parsing: now parser recognizes all well-formed DOCTYPE declarations
4. Fixed `xml_attribute::as_uint()` for large numbers (i.e. 2^{32}-1)
5. Fixed `xml_node::first_element_by_path` for path components that are prefixes of node names, but are not exactly equal to them.

- Specification changes:

1. `parse()` API changed to `load_buffer` / `load_buffer_inplace` / `load_buffer_inplac` `load_buffer` APIs do not require zero-terminated strings.

2. Renamed `as_utf16` to `as_wide`
3. Changed `xml_node::offset_debug` return type and `xml_parse_result::offset` type to `ptrdiff_t`
4. Nodes/attributes with empty names are now printed as `:anonymous`

- Performance improvements:
  1. Optimized document parsing and saving
  2. Changed internal memory management: internal allocator is used for both metadata and name/value data; allocated pages are deleted if all allocations from them are deleted
  3. Optimized memory consumption: `sizeof(xml_node_struct)` reduced from 40 bytes to 32 bytes on x86
  4. Optimized debug mode parsing/saving by order of magnitude
- Miscellaneous:
  1. All STL includes except `<exception>` in `pugixml.hpp` are

replaced with forward declarations

2. `xml_node::remove_child` and `xml_node::remove_attribute` now return the operation result

- Compatibility:

1. `parse()` and `as_utf16` are left for compatibility (these functions are deprecated and will be removed in version 1.0)

2. Wildcard functions, `document_order` / `precompute_document_order` functions, `all_elements_by_name` function and `format_write_bom_utf8` flag are deprecated and will be removed in version 1.0

3. `xpath_type_t` enumeration was renamed to `xpath_value_type`; `xpath_type_t` is deprecated and will be removed in version 1.0

v0.5 8.11.2009

Major bugfix release. Changes:

XPath bugfixes:

1. Fixed `translate()`, `lang()` and `concat()` functions (infinite loops/crashes)
2. Fixed compilation of queries with empty literal strings (`""`)
3. Fixed axis tests: they never add empty nodes/attributes to the resulting node set now
4. Fixed string-value evaluation for node-set (the result excluded some text descendants)
5. Fixed `self::` axis (it behaved like `ancestor-or-self::`)
6. Fixed `following::` and `preceding::` axes (they included descendent and ancestor nodes, respectively)
7. Minor fix for `namespace-uri()` function (namespace declaration scope includes the parent element of namespace declaration attribute)
8. Some incorrect queries are no longer parsed now (i.e. `foo:*`)

9. Fixed `text()`/etc. node test parsing bug (i.e. `foo[text()]` failed to compile)
10. Fixed root step (`/`) - it now selects empty node set if query is evaluated on empty node
11. Fixed string to number conversion (`"123 "` converted to NaN, `"123 .456"` converted to 123.456 - now the results are 123 and NaN, respectively)
12. Node set copying now preserves sorted type; leads to better performance on some queries

Miscellaneous bugfixes:

1. Fixed `xml_node::offset_debug` for PI nodes
2. Added empty attribute checks to `xml_node::remove_attribute`
3. Fixed `node_pi` and `node_declaration` copying
4. Const-correctness fixes

Specification changes:

1. `xpath_node::select_nodes()` and related functions now

throw exception if expression return type is not node set (instead of assertion)

2. `xml_node::traverse()` now sets depth to -1 for both `begin()` and `end()` callbacks (was 0 at `begin()` and -1 at `end()`)
3. In case of non-raw node printing a newline is output after PCDATA inside nodes if the PCDATA has siblings
4. UTF8 □ `wchar_t` conversion now considers 5-byte UTF8-like sequences as invalid

New features:

1. Added `xpath_node_set::operator[]` for index-based iteration
2. Added `xpath_query::return_type()`
3. Added getter accessors for memory-management functions

v0.42 17.09.2009

Maintenance release. Changes:

Bug fixes:

1. Fixed deallocation in case of custom allocation functions or if `delete[]` / `free` are incompatible
2. XPath parser fixed for incorrect queries (i.e. incorrect XPath queries should now always fail to compile)
3. Const-correctness fixes for `find_child_by_attribute`
4. Improved compatibility (miscellaneous warning fixes, fixed `<cstring>` include dependency for GCC)
5. Fixed iterator begin/end and print function to work correctly for empty nodes

New features:

1. Added `PUGIXML_API` / `PUGIXML_CLASS` / `PUGIXML_FUNCTION` configuration macros to control class/function attributes
2. Added `xml_attribute::set_value` overloads for different types

v0.41 8.02.2009

Maintenance release. Changes:

Bug fixes:

1. Fixed bug with node printing (occasionally some content was not written to output stream)

v0.4 18.01.2009

Changes:

Bug fixes:

1. Documentation fix in samples for `parse()` with manual lifetime control
2. Fixed document order sorting in XPath (it caused wrong order of nodes after `xpath_node_set::sort` and wrong results of some XPath queries)

Node printing changes:

1. Single quotes are no longer escaped when printing nodes
2. Symbols in second half of ASCII table are no longer escaped

when printing nodes; because of this, `format_utf8` flag is deleted as it's no longer needed and `format_write_bom` is renamed to `format_write_bom_utf8`.

3. Reworked node printing - now it works via `xml_writer` interface; implementations for `FILE*` and `std::ostream` are available. As a side-effect, `xml_document::save_file` now works without STL.

New features:

1. Added unsigned integer support for attributes (`xml_attribute::as_uint`, `xml_attribute::operator=`)
2. Now document declaration (`<?xml ...?>`) is parsed as node with type `node_declaration` when `parse_declaration` flag is specified (access to encoding/version is performed as if they were attributes, i.e. `doc.child("xml").attribute("version").as_float()`); corresponding flags for node printing were also added
3. Added support for custom memory management (see `set_memory_management_functions` for details)
4. Implemented node/attribute copying (see

`xml_node::insert_copy_*` and `xml_node::append_copy` for details)

5. Added `find_child_by_attribute` and `find_child_by_attribute_w` to simplify parsing code in some cases (i.e. COLLADA files)
6. Added file offset information querying for debugging purposes (now you're able to determine exact location of any `xml_node` in parsed file, see `xml_node::offset_debug` for details)
7. Improved error handling for parsing - now `load()`, `load_file()` and `parse()` return `xml_parse_result`, which contains error code and last parsed offset; this does not break old interface as `xml_parse_result` can be implicitly casted to `bool`.

v0.34 31.10.2007

Maintenance release. Changes:

Bug fixes:

1. Fixed bug with loading from text-mode iostreams
2. Fixed leak when `transfer_ownership` is true and parsing is failing
3. Fixed bug in saving (`\r` and `\n` are now escaped in attribute values)
4. Renamed `free()` to `destroy()` - some macro conflicts were reported

New features:

1. Improved compatibility (supported Digital Mars C++, MSVC 6, CodeWarrior 8, PGI C++, Comeau, supported PS3 and XBox360)
2. `PUGIXML_NO_EXCEPTION` flag for platforms without exception handling

v0.3 21.02.2007

Refactored, reworked and improved version. Changes:

Interface:

1. Added XPath
2. Added tree modification functions
3. Added no STL compilation mode
4. Added saving document to file
5. Refactored parsing flags
6. Removed `xml_parser` class in favor of `xml_document`
7. Added transfer ownership parsing mode
8. Modified the way `xml_tree_walker` works
9. Iterators are now non-constant

Implementation:

1. Support of several compilers and platforms
2. Refactored and sped up parsing core
3. Improved standard compliancy
4. Added XPath implementation
5. Fixed several bugs

v0.2 6.11.2006

First public release. Changes:

Bug fixes:

1. Fixed `child_value()` (for empty nodes)
2. Fixed `xml_parser_impl` warning at W4

New features:

1. Introduced `child_value(name)` and `child_value_w(name)`
2. `parse_eol_pcdata` and `parse_eol_attribute` flags + `parse_minimal` optimizations
3. Optimizations of `strconv_t`

v0.1 15.07.2006

First private release for testing purposes

10. API Reference

This is the reference for all macros, types, enumerations, classes and functions in pugixml. Each symbol is a link that leads to the relevant section of the manual.

10.1. Macros

```
#define PUGIXML_WCHAR_MODE
#define PUGIXML_NO_XPATH
#define PUGIXML_NO_STL
#define PUGIXML_NO_EXCEPTIONS
#define PUGIXML_API
#define PUGIXML_CLASS
#define PUGIXML_FUNCTION
#define PUGIXML_MEMORY_PAGE_SIZE
#define PUGIXML_MEMORY_OUTPUT_STACK
#define PUGIXML_MEMORY_XPATH_PAGE_SIZE
#define PUGIXML_HEADER_ONLY
#define PUGIXML_HAS_LONG_LONG
```

10.2. Types

```
typedef configuration-defined-type char_t;
typedef configuration-defined-type string_t;
typedef void* (*allocation_function)(size_t size);
typedef void (*deallocation_function)(void* ptr);
```

10.3. Enumerations

```
enum xml_node_type
    node_null
    node_document
    node_element
    node_pcdata
    node_cdata
    node_comment
    node_pi
    node_declaration
    node_doctype

enum xml_parse_status
    status_ok
    status_file_not_found
    status_io_error
```

```
    status_out_of_memory
    status_internal_error
    status_unrecognized_tag
    status_bad_pi
    status_bad_comment
    status_bad_cdata
    status_bad_doctype
    status_bad_pcdata
    status_bad_start_element
    status_bad_attribute
    status_bad_end_element
    status_end_element_mismatch
    status_append_invalid_root
    status_no_document_element

enum xml_encoding
    encoding_auto
    encoding_utf8
    encoding_utf16_le
    encoding_utf16_be
    encoding_utf16
    encoding_utf32_le
    encoding_utf32_be
    encoding_utf32
    encoding_wchar
    encoding_latin1
```

```
enum xpath_value_type
    xpath_type_none
    xpath_type_node_set
    xpath_type_number
    xpath_type_string
    xpath_type_boolean
```

10.4. Constants

```
// Formatting options bit flags:
const unsigned int format_default
const unsigned int format_indent
const unsigned int format_no_declaration
const unsigned int format_no_escapes
const unsigned int format_raw
const unsigned int format_save_file_text
const unsigned int format_write_bom

// Parsing options bit flags:
const unsigned int parse_cdata
const unsigned int parse_comments
const unsigned int parse_declaration
const unsigned int parse_default
const unsigned int parse_doctype
const unsigned int parse_eol
```

```
const unsigned int parse_escapes
const unsigned int parse_fragment
const unsigned int parse_full
const unsigned int parse_minimal
const unsigned int parse_pi
const unsigned int parse_trim_pcdata
const unsigned int parse_ws_pcdata
const unsigned int parse_ws_pcdata_single
const unsigned int parse_wconv_attribute
const unsigned int parse_wnorm_attribute
```

10.5. Classes

```
class xml_attribute
    xml_attribute();

    bool empty() const;
    operator unspecified_bool_type() const;

    bool operator==(const xml_attribute& r) const;
    bool operator!=(const xml_attribute& r) const;
    bool operator<(const xml_attribute& r) const;
    bool operator>(const xml_attribute& r) const;
    bool operator<=(const xml_attribute& r) const;
    bool operator>=(const xml_attribute& r) const;
```

```
    size_t hash_value() const;

    xml_attribute next_attribute() const;
    xml_attribute previous_attribute() const;

    const char_t* name() const;
    const char_t* value() const;

    const char_t* as_string(const char_t* def = "")
const;
    int as_int(int def = 0) const;
    unsigned int as_uint(unsigned int def = 0) const;
    double as_double(double def = 0) const;
    float as_float(float def = 0) const;
    bool as_bool(bool def = false) const;
    long long as_llong(long long def = 0) const;
    unsigned long long as_ullong(unsigned long long def =
0) const;

    bool set_name(const char_t* rhs);
    bool set_value(const char_t* rhs);
    bool set_value(int rhs);
    bool set_value(unsigned int rhs);
    bool set_value(double rhs);
    bool set_value(float rhs);
    bool set_value(bool rhs);
```

```
    bool set_value(long long rhs);
    bool set_value(unsigned long long rhs);

    xml_attribute& operator=(const char_t* rhs);
    xml_attribute& operator=(int rhs);
    xml_attribute& operator=(unsigned int rhs);
    xml_attribute& operator=(double rhs);
    xml_attribute& operator=(float rhs);
    xml_attribute& operator=(bool rhs);
    xml_attribute& operator=(long long rhs);
    xml_attribute& operator=(unsnigned long long rhs);

class xml_node
    xml_node();

    bool empty() const;
    operator unspecified_bool_type() const;

    bool operator==(const xml_node& r) const;
    bool operator!=(const xml_node& r) const;
    bool operator<(const xml_node& r) const;
    bool operator>(const xml_node& r) const;
    bool operator<=(const xml_node& r) const;
    bool operator>=(const xml_node& r) const;

    size_t hash_value() const;
```

```
    xml_node_type type() const;

    const char_t* name() const;
    const char_t* value() const;

    xml_node parent() const;
    xml_node first_child() const;
    xml_node last_child() const;
    xml_node next_sibling() const;
    xml_node previous_sibling() const;

    xml_attribute first_attribute() const;
    xml_attribute last_attribute() const;

    implementation-defined-type children() const;
    implementation-defined-type children(const char_t*
name) const;
    implementation-defined-type attributes() const;

    xml_node child(const char_t* name) const;
    xml_attribute attribute(const char_t* name) const;
    xml_node next_sibling(const char_t* name) const;
    xml_node previous_sibling(const char_t* name) const;
    xml_node find_child_by_attribute(const char_t* name,
const char_t* attr_name, const char_t* attr_value) const;
    xml_node find_child_by_attribute(const char_t*
attr_name, const char_t* attr_value) const;
```

```
    const char_t* child_value() const;
    const char_t* child_value(const char_t* name) const;
    xml_text text() const;

    typedef xml_node_iterator iterator;
    iterator begin() const;
    iterator end() const;

    typedef xml_attribute_iterator attribute_iterator;
    attribute_iterator attributes_begin() const;
    attribute_iterator attributes_end() const;

    bool traverse(xml_tree_walker& walker);

    template <typename Predicate> xml_attribute
find_attribute(Predicate pred) const;
    template <typename Predicate> xml_node
find_child(Predicate pred) const;
    template <typename Predicate> xml_node
find_node(Predicate pred) const;

    string_t path(char_t delimiter = '/') const;
    xml_node xml_node::first_element_by_path(const
char_t* path, char_t delimiter = '/') const;
    xml_node root() const;
    ptrdiff_t offset_debug() const;
```

```
    bool set_name(const char_t* rhs);
    bool set_value(const char_t* rhs);

    xml_attribute append_attribute(const char_t* name);
    xml_attribute prepend_attribute(const char_t* name);
    xml_attribute insert_attribute_after(const char_t*
name, const xml_attribute& attr);
    xml_attribute insert_attribute_before(const char_t*
name, const xml_attribute& attr);

    xml_node append_child(xml_node_type type =
node_element);
    xml_node prepend_child(xml_node_type type =
node_element);
    xml_node insert_child_after(xml_node_type type, const
xml_node& node);
    xml_node insert_child_before(xml_node_type type,
const xml_node& node);

    xml_node append_child(const char_t* name);
    xml_node prepend_child(const char_t* name);
    xml_node insert_child_after(const char_t* name, const
xml_node& node);
    xml_node insert_child_before(const char_t* name,
const xml_node& node);
```

```
    xml_attribute append_copy(const xml_attribute&
proto);
    xml_attribute prepend_copy(const xml_attribute&
proto);
    xml_attribute insert_copy_after(const xml_attribute&
proto, const xml_attribute& attr);
    xml_attribute insert_copy_before(const xml_attribute&
proto, const xml_attribute& attr);

    xml_node append_copy(const xml_node& proto);
    xml_node prepend_copy(const xml_node& proto);
    xml_node insert_copy_after(const xml_node& proto,
const xml_node& node);
    xml_node insert_copy_before(const xml_node& proto,
const xml_node& node);

    xml_node append_move(const xml_node& moved);
    xml_node prepend_move(const xml_node& moved);
    xml_node insert_move_after(const xml_node& moved,
const xml_node& node);
    xml_node insert_move_before(const xml_node& moved,
const xml_node& node);

    bool remove_attribute(const xml_attribute& a);
    bool remove_attribute(const char_t* name);
    bool remove_child(const xml_node& n);
    bool remove_child(const char_t* name);
```

```
    xml_parse_result append_buffer(const void* contents,
size_t size, unsigned int options = parse_default,
xml_encoding encoding = encoding_auto);

    void print(xml_writer& writer, const char_t* indent =
"\t", unsigned int flags = format_default, xml_encoding
encoding = encoding_auto, unsigned int depth = 0) const;
    void print(std::ostream& os, const char_t* indent =
"\t", unsigned int flags = format_default, xml_encoding
encoding = encoding_auto, unsigned int depth = 0) const;
    void print(std::wostream& os, const char_t* indent =
"\t", unsigned int flags = format_default, unsigned int
depth = 0) const;

    xpath_node select_node(const char_t* query,
xpath_variable_set* variables = 0) const;
    xpath_node select_node(const xpath_query& query)
const;
    xpath_node_set select_nodes(const char_t* query,
xpath_variable_set* variables = 0) const;
    xpath_node_set select_nodes(const xpath_query& query)
const;

class xml_document
    xml_document();
    ~xml_document();
```

```
    void reset();
    void reset(const xml_document& proto);

    xml_parse_result load(std::istream& stream, unsigned
int options = parse_default, xml_encoding encoding =
encoding_auto);
    xml_parse_result load(std::wistream& stream, unsigned
int options = parse_default);

    xml_parse_result load_string(const char_t* contents,
unsigned int options = parse_default);

    xml_parse_result load_file(const char* path, unsigned
int options = parse_default, xml_encoding encoding =
encoding_auto);
    xml_parse_result load_file(const wchar_t* path,
unsigned int options = parse_default, xml_encoding
encoding = encoding_auto);

    xml_parse_result load_buffer(const void* contents,
size_t size, unsigned int options = parse_default,
xml_encoding encoding = encoding_auto);
    xml_parse_result load_buffer_inplace(void* contents,
size_t size, unsigned int options = parse_default,
xml_encoding encoding = encoding_auto);
    xml_parse_result load_buffer_inplace_own(void*
```

```
contents, size_t size, unsigned int options =
parse_default, xml_encoding encoding = encoding_auto);

    bool save_file(const char* path, const char_t* indent
= "\t", unsigned int flags = format_default, xml_encoding
encoding = encoding_auto) const;
    bool save_file(const wchar_t* path, const char_t*
indent = "\t", unsigned int flags = format_default,
xml_encoding encoding = encoding_auto) const;

    void save(std::ostream& stream, const char_t* indent
= "\t", unsigned int flags = format_default, xml_encoding
encoding = encoding_auto) const;
    void save(std::wostream& stream, const char_t* indent
= "\t", unsigned int flags = format_default) const;

    void save(xml_writer& writer, const char_t* indent =
"\t", unsigned int flags = format_default, xml_encoding
encoding = encoding_auto) const;

    xml_node document_element() const;

struct xml_parse_result
    xml_parse_status status;
    ptrdiff_t offset;
    xml_encoding encoding;
```

```
    operator bool() const;
    const char* description() const;

class xml_node_iterator
class xml_attribute_iterator

class xml_tree_walker
    virtual bool begin(xml_node& node);
    virtual bool for_each(xml_node& node) = 0;
    virtual bool end(xml_node& node);

    int depth() const;

class xml_text
    bool empty() const;
    operator xml_text::unspecified_bool_type() const;

    const char_t* xml_text::get() const;

    const char_t* as_string(const char_t* def = "")
const;
    int as_int(int def = 0) const;
    unsigned int as_uint(unsigned int def = 0) const;
    double as_double(double def = 0) const;
    float as_float(float def = 0) const;
    bool as_bool(bool def = false) const;
    long long as_llong(long long def = 0) const;
```

```
    unsigned long long as_ullong(unsigned long long def =
0) const;

    bool set(const char_t* rhs);

    bool set(int rhs);
    bool set(unsigned int rhs);
    bool set(double rhs);
    bool set(float rhs);
    bool set(bool rhs);
    bool set(long long rhs);
    bool set(unsigned long long rhs);

    xml_text& operator=(const char_t* rhs);
    xml_text& operator=(int rhs);
    xml_text& operator=(unsigned int rhs);
    xml_text& operator=(double rhs);
    xml_text& operator=(float rhs);
    xml_text& operator=(bool rhs);
    xml_text& operator=(long long rhs);
    xml_text& operator=(unsigned long long rhs);

    xml_node data() const;

class xml_writer
    virtual void write(const void* data, size_t size) =
0;
```

```
class xml_writer_file: public xml_writer
    xml_writer_file(void* file);

class xml_writer_stream: public xml_writer
    xml_writer_stream(std::ostream& stream);
    xml_writer_stream(std::wostream& stream);

struct xpath_parse_result
    const char* error;
    ptrdiff_t offset;

    operator bool() const;
    const char* description() const;

class xpath_query
    explicit xpath_query(const char_t* query,
xpath_variable_set* variables = 0);

    bool evaluate_boolean(const xpath_node& n) const;
    double evaluate_number(const xpath_node& n) const;
    string_t evaluate_string(const xpath_node& n) const;
    size_t evaluate_string(char_t* buffer, size_t
capacity, const xpath_node& n) const;
    xpath_node_set evaluate_node_set(const xpath_node& n)
const;
    xpath_node evaluate_node(const xpath_node& n) const;
```

```
    xpath_value_type return_type() const;

    const xpath_parse_result& result() const;
    operator unspecified_bool_type() const;

class xpath_exception: public std::exception
    virtual const char* what() const throw();

    const xpath_parse_result& result() const;

class xpath_node
    xpath_node();
    xpath_node(const xml_node& node);
    xpath_node(const xml_attribute& attribute, const
xml_node& parent);

    xml_node node() const;
    xml_attribute attribute() const;
    xml_node parent() const;

    operator unspecified_bool_type() const;
    bool operator==(const xpath_node& n) const;
    bool operator!=(const xpath_node& n) const;

class xpath_node_set
    xpath_node_set();
```

```
    xpath_node_set(const_iterator begin, const_iterator
end, type_t type = type_unsorted);

    typedef const xpath_node* const_iterator;
    const_iterator begin() const;
    const_iterator end() const;

    const xpath_node& operator[](size_t index) const;
    size_t size() const;
    bool empty() const;

    xpath_node first() const;

    enum type_t {type_unsorted, type_sorted,
type_sorted_reverse};
    type_t type() const;
    void sort(bool reverse = false);

class xpath_variable
    const char_t* name() const;
    xpath_value_type type() const;

    bool get_boolean() const;
    double get_number() const;
    const char_t* get_string() const;
    const xpath_node_set& get_node_set() const;
```

```
    bool set(bool value);
    bool set(double value);
    bool set(const char_t* value);
    bool set(const xpath_node_set& value);

class xpath_variable_set
    xpath_variable* add(const char_t* name,
xpath_value_type type);

    bool set(const char_t* name, bool value);
    bool set(const char_t* name, double value);
    bool set(const char_t* name, const char_t* value);
    bool set(const char_t* name, const xpath_node_set&
value);

    xpath_variable* get(const char_t* name);
    const xpath_variable* get(const char_t* name) const;
```

10.6. Functions

```
std::string as_utf8(const wchar_t* str);
std::string as_utf8(const std::wstring& str);
std::wstring as_wide(const char* str);
std::wstring as_wide(const std::string& str);
void set_memory_management_functions(allocation_function
```

```
allocate, deallocation_function deallocate);
allocation_function get_memory_allocation_function();
deallocation_function get_memory_deallocation_function();
```

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

Last updated 2015-04-10 20:49:27 PDT