Boost.Lexical_Cast 1.0

Copyright © 2000-2005 Kevlin Henney
Copyright © 2006-2010 Alexander Nasonov
Copyright © 2011-2014 Antony Polukhin

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Motivation 2
Examples 3
    Strings to numbers conversion 3
    Numbers to strings conversion 3
    Converting to string without dynamic memory allocation 3
    Converting part of the string 3
    Generic programming (Boost.Fusion) 4
    Generic programming (Boost.Variant) 4
Synopsis 6
    lexical_cast 6
    bad_lexical_cast 7
    try_lexical_convert 7
Frequently Asked Questions 9
Changes 11
Performance 13
    Tests description 13
    Clang version 3.0 (tags/RELEASE_30/final) 14
    GNU C++ version 4.6.3 21
    GNU C++ version 4.5.3 28
    GNU C++ version 4.4.7 35
    Microsoft Visual C++ version 11.0 42

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Motivation

Sometimes a value must be converted to a literal text form, such as an `int` represented as a `std::string`, or vice-versa, when a `std::string` is interpreted as an `int`. Such examples are common when converting between data types internal to a program and representation external to a program, such as windows and configuration files.

The standard C and C++ libraries offer a number of facilities for performing such conversions. However, they vary with their ease of use, extensibility, and safety.

For instance, there are a number of limitations with the family of standard C functions typified by `atoi`:

- Conversion is supported in one direction only: from text to internal data type. Converting the other way using the C library requires either the inconvenience and compromised safety of the `sprintf` function, or the loss of portability associated with non-standard functions such as `itoa`.
- The range of types supported is only a subset of the built-in numeric types, namely `int`, `long`, and `double`.
- The range of types cannot be extended in a uniform manner. For instance, conversion from string representation to complex or rational.

The standard C functions typified by `strtol` have the same basic limitations, but offer finer control over the conversion process. However, for the common case such control is often either not required or not used. The `scanf` family of functions offer even greater control, but also lack safety and ease of use.

The standard C++ library offers `stringstream` for the kind of in-core formatting being discussed. It offers a great deal of control over the formatting and conversion of I/O to and from arbitrary types through text. However, for simple conversions direct use of `stringstream` can be either clumsy (with the introduction of extra local variables and the loss of infix-expression convenience) or obscure (where `stringstream` objects are created as temporary objects in an expression). Facets provide a comprehensive concept and facility for controlling textual representation, but their perceived complexity and high entry level requires an extreme degree of involvement for simple conversions, and excludes all but a few programmers.

The `lexical_cast` function template offers a convenient and consistent form for supporting common conversions to and from arbitrary types when they are represented as text. The simplification it offers is in expression-level convenience for such conversions. For more involved conversions, such as where precision or formatting need tighter control than is offered by the default behavior of `lexical_cast`, the conventional `std::stringstream` approach is recommended. Where the conversions are numeric to numeric, `boost::numeric_cast` may offer more reasonable behavior than `lexical_cast`.

For a good discussion of the options and issues involved in string-based formatting, including comparison of `stringstream`, `lexical_cast`, and others, see Herb Sutter's article, The String Formatters of Manor Farm. Also, take a look at the Performance section.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Examples

Strings to numbers conversion

The following example treats command line arguments as a sequence of numeric data

```
#include <boost/lexical_cast.hpp>
#include <vector>

int main(int /*argc*/, char * argv[])
{
    using boost::lexical_cast;
    using boost::bad_lexical_cast;

    std::vector<short> args;

    while (*++argv)
    {
        try
        {
            args.push_back(lexical_cast<short>(*argv));
        }
        catch(const bad_lexical_cast &)
        {
            args.push_back(0);
        }
    }

    // ...
}
```

Numbers to strings conversion

The following example uses numeric data in a string expression:

```
void log_message(const std::string &);

void log_errno(int yoko)
{
    log_message("Error " + boost::lexical_cast<std::string>(yoko) + ": " + strerror(yoko));
}
```

Converting to string without dynamic memory allocation

The following example converts some number and puts it to file:

```
void number_to_file(int number, FILE* file)
{
    typedef boost::array<char, 50> buf_t; // You can use std::array if your compiler supports it
    buf_t buffer = boost::lexical_cast<buf_t>(number); // No dynamic memory allocation
    fputs(buffer.begin(), file);
}
```

Converting part of the string

The following example takes part of the string and converts it to `int`:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
int convert_strings_part(const std::string& s, std::size_t pos, std::size_t n)
{
    return boost::lexical_cast<int>(s.data() + pos, n);
}
```

Generic programming (Boost.Fusion)

In this example we'll make a `stringize` method that accepts a sequence, converts each element of the sequence into string and appends that string to the result.

Example is based on the example from the Boost C++ Application Development Cookbook by Antony Polukhin, ISBN 9781849514880.

Step 1: Making a functor that converts any type to a string and remembers result:

```
#include <boost/lexical_cast.hpp>

struct stringize_functor {
private:
    std::string& result;

public:
    explicit stringize_functor(std::string& res)
        : result(res)
    {}

    template <class T>
    void operator()(const T& v) const {
        result += boost::lexical_cast<std::string>(v);
    }
};
```

Step 2: Applying `stringize_functor` to each element in sequence:

```
#include <boost/fusion/include/for_each.hpp>
template <class Sequence>
std::string stringize(const Sequence& seq) {
    std::string result;
    boost::fusion::for_each(seq, stringize_functor(result));
    return result;
}
```

Step 3: Using the `stringize` with different types:

```
#include <cassert>
#include <boost/fusion/adapted/boost_tuple.hpp>
#include <boost/fusion/adapted/std_pair.hpp>

int main() {
    boost::tuple<char, int, char, int> decim('-', 10, 'e', 5);
    assert(stringize(decim) == "-10e5");

    std::pair<short, std::string> value_and_type(270, "Kelvin");
    assert(stringize(value_and_type) == "270Kelvin");
}
```

Generic programming (Boost.Variant)

In this example we'll make a `to_long_double` method that converts value of the Boost.Variant to `long double`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/lexical_cast.hpp>
#include <boost/variant.hpp>
#include <cassert>

struct to_long_double_functor: boost::static_visitor<long double> {
    template <class T>
    long double operator()(const T& v) const {
       // Lexical cast has many optimizations including optimizations for situations that usually
        // occur in generic programming, like std::string to std::string or arithmetic type to ↲
arithmetic type conversion.
        return boost::lexical_cast<long double>(v);
    }
};

// Throws `boost::bad_lexical_cast` if value of the variant is not convertible to `long double`
template <class Variant>
long double to_long_double(const Variant& v) {
    return boost::apply_visitor(to_long_double_functor(), v);
}

int main() {
    boost::variant<char, int, std::string> v1('0'), v2("10.0001"), v3(1);

    long double sum = to_long_double(v1) + to_long_double(v2) + to_long_double(v3);
    assert(sum > 11 && sum < 11.1);
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

Library features defined in boost/lexical_cast.hpp:

```
namespace boost
{
    class bad_lexical_cast;

    template<typename Target, typename Source>
      Target lexical_cast(const Source& arg);

    template <typename Target>
      Target lexical_cast(const AnyCharacterType* chars, std::size_t count);

    namespace conversion
    {
        template<typename Target, typename Source>
            bool try_lexical_convert(const Source& arg, Target& result);

        template <typename AnyCharacterType, typename Target>
          bool try_lexical_convert(const AnyCharacterType* chars, std::size_t count, Target& res↲
ult);

    } // namespace conversion
} // namespace boost
```

lexical_cast

```
template<typename Target, typename Source>
  Target lexical_cast(const Source& arg);
```

Returns the result of streaming arg into a standard library string-based stream and then out as a Target object. Where Target is either `std::string` or `std::wstring`, stream extraction takes the whole content of the string, including spaces, rather than relying on the default `operator>>` behavior. If the conversion is unsuccessful, a `bad_lexical_cast` exception is thrown.

```
template <typename Target>
  Target lexical_cast(const AnyCharacterType* chars, std::size_t count);
```

Takes an array of `count` characters as input parameter and streams them out as a Target object. If the conversion is unsuccessful, a `bad_lexical_cast` exception is thrown. This call may be useful for processing nonzero terminated array of characters or processing just some part of character array.

The requirements on the argument and result types for both functions are:

- Source is OutputStreamable, meaning that an `operator<<` is defined that takes a `std::ostream` or `std::wostream` object on the left hand side and an instance of the argument type on the right.
- Target is InputStreamable, meaning that an `operator>>` is defined that takes a `std::istream` or `std::wistream` object on the left hand side and an instance of the result type on the right.
- Target is CopyConstructible [20.1.3].
- Target is DefaultConstructible, meaning that it is possible to default-initialize an object of that type [8.5, 20.1.4].

The character type of the underlying stream is assumed to be `char` unless either the `Source` or the `Target` requires wide-character streaming, in which case the underlying stream uses `wchar_t`. Following types also can use `char16_t` or `char32_t` for wide-character streaming:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Single character: `char16_t`, `char32_t`
- Arrays of characters: `char16_t *`, `char32_t *`, `const char16_t *`, `const char32_t *`
- Strings: `std::basic_string`, `boost::containers::basic_string`
- `boost::iterator_range<WideCharPtr>`, where `WideCharPtr` is a pointer to wide-character or pointer to const wide-character
- `boost::array<CharT, N>` and `std::array<CharT, N>`, `boost::array<const CharT, N>` and `std::array<const CharT, N>`

Important

Many compilers and runtime libraries fail to make conversions using new Unicode characters. Make sure that the following code compiles and outputs nonzero values, before using new types:

```
std::cout
    << boost::lexical_cast<std::u32string>(1.0).size()
    << "  "
    << boost::lexical_cast<std::u16string>(1.0).size();
```

Where a higher degree of control is required over conversions, `std::stringstream` and `std::wstringstream` offer a more appropriate path. Where non-stream-based conversions are required, `lexical_cast` is the wrong tool for the job and is not special-cased for such scenarios.

bad_lexical_cast

```
class bad_lexical_cast : public std::bad_cast
{
public:
    ... // same member function interface as std::exception
};
```

Exception used to indicate runtime lexical_cast failure.

try_lexical_convert

`boost::lexical_cast` remains the main interface for lexical conversions. It must be used by default in most cases. However some developers wish to make their own conversion functions, reusing all the optimizations of the `boost::lexical_cast`. That's where the `boost::conversion::try_lexical_convert` function steps in.

`try_lexical_convert` returns `true` if conversion succeeded, otherwise returns `false`. If conversion failed and `false` was returned, state of `result` output variable is undefined.

Actually, `boost::lexical_cast` is implemented using `try_lexical_convert`:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
template <typename Target, typename Source>
inline Target lexical_cast(const Source &arg)
{
    Target result;

    if (!conversion::try_lexical_convert(arg, result))
        throw bad_lexical_cast();

    return result;
}
```

`try_lexical_convert` relaxes the CopyConstructible and DefaultConstructible requirements for `Target` type. Following requirements for `Target` and `Source` remain:

- Source must be OutputStreamable, meaning that an `operator<<` is defined that takes a `std::ostream` or `std::wostream` object on the left hand side and an instance of the argument type on the right.
- Target must be InputStreamable, meaning that an `operator>>` is defined that takes a `std::istream` or `std::wistream` object on the left hand side and an instance of the result type on the right.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Frequently Asked Questions

- **Question:** Why does `lexical_cast<int8_t>("127")` throw `bad_lexical_cast`?
    - **Answer:** The type `int8_t` is a `typedef` to `char` or `signed char`. Lexical conversion to these types is simply reading a byte from source but since the source has more than one byte, the exception is thrown. Please use other integer types such as `int` or `short int`. If bounds checking is important, you can also call `boost::numeric_cast`: `numeric_cast<int8_t>(lexical_cast<int>("127"));`

- **Question:** Why does `lexical_cast<unsigned char>("127")` throw `bad_lexical_cast`?
    - **Answer:** Lexical conversion to any char type is simply reading a byte from source. But since the source has more than one byte, the exception is thrown. Please use other integer types such as `int` or `short int`. If bounds checking is important, you can also call `boost::numeric_cast`: `numeric_cast<unsigned char>(lexical_cast<int>("127"));`

- **Question:** What does `lexical_cast<std::string>` of an `int8_t` or `uint8_t` not do what I expect?
    - **Answer:** As above, note that int8_t and uint8_t are actually chars and are formatted as such. To avoid this, cast to an integer type first: `lexical_cast<std::string>(static_cast<int>(n));`

- **Question:** The implementation always resets the `ios_base::skipws` flag of an underlying stream object. It breaks my `operator>>` that works only in presence of this flag. Can you remove code that resets the flag?
    - **Answer:** May be in a future version. There is no requirement in Lexical Conversion Library Proposal for TR2, N1973 by Kevlin Henney and Beman Dawes to reset the flag but remember that Lexical Conversion Library Proposal for TR2, N1973 is not yet accepted by the committee. By the way, it's a great opportunity to make your `operator>>` more general. Read a good C++ book, study `std::sentry` and `ios_state_saver`.

- **Question:** Why `std::cout << boost::lexical_cast<unsigned int>("-1");` does not throw, but outputs 4294967295?
    - **Answer:** `boost::lexical_cast` has the behavior of `std::stringstream`, which uses `num_get` functions of `std::locale` to convert numbers. If we look at the Programming languages — C++, we'll see, that `num_get` uses the rules of `scanf` for conversions. And in the C99 standard for unsigned input value minus sign is optional, so if a negative number is read, no errors will arise and the result will be the two's complement.

- **Question:** Why `boost::lexical_cast<int>(L'A');` outputs 65 and `boost::lexical_cast<wchar_t>(L"65");` does not throw?
    - **Answer:** If you are using an old version of Visual Studio or compile code with /Zc:wchar_t- flag, `boost::lexical_cast` sees single `wchar_t` character as `unsigned short`. It is not a `boost::lexical_cast` mistake, but a limitation of compiler options that you use.

- **Question:** Why `boost::lexical_cast<double>("-1.#IND");` throws `boost::bad_lexical_cast`?
    - **Answer:** `"-1.#IND"` is a compiler extension, that violates standard. You shall input `"-nan"`, `"nan"`, `"inf"` , `"-inf"` (case insensitive) strings to get NaN and Inf values. `boost::lexical_cast<string>` outputs `"-nan"`, `"nan"`, `"inf"`, `"-inf"` strings, when has NaN or Inf input values.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- **Question:** What is the fastest way to convert a non zero terminated string or a substring using `boost::lexical_cast`?
  - **Answer:** Use `boost::iterator_range` for conversion or `lexical_cast` overload with two parameters. For example, if you whant to convert to `int` two characters from a string `str`, you shall write `lexical_cast<int>(make_iterator_range(str.data(), str.data() + 2));` or `lexical_cast<int>(str.data(), 2);`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Changes

- **boost 1.56.0 :**
  - Added `boost::conversion::try_lexical_convert` functions.
- **boost 1.54.0 :**
  - Fix some issues with `boost::int128_type` and `boost::uint128_type` conversions. Notify user at compile time if the `std::numeric_limits` are not specialized for 128bit types and `boost::lexical_cast` can not make conversions.
- **boost 1.54.0 :**
  - Added code to convert `boost::int128_type` and `boost::uint128_type` types (requires GCC 4.7 or higher).
  - Conversions to pointers will now fail to compile, instead of throwing at runtime.
  - Restored ability to get pointers to `lexical_cast` function (was broken in 1.53.0).
- **boost 1.53.0 :**
  - Much better input and output streams detection for user defined types.
- **boost 1.52.0 :**
  - Restored compilation on MSVC-2003 (was broken in 1.51.0).
  - Added `lexical_cast(const CharType* chars, std::size_t count)` function overload.
- **boost 1.51.0 :**
  - Better performance, less memory usage for `boost::array<character_type, N>` and `std::array<character_type, N>` conversions.
- **boost 1.50.0 :**
  - `boost::bad_lexical_cast` exception is now globaly visible and can be catched even if code is compiled with -fvisibility=hidden.
  - Now it is possible to compile library with disabled exceptions.
  - Better performance, less memory usage and bugfixes for `boost::iterator_range<character_type*>` conversions.
- **boost 1.49.0 :**
  - Restored work with typedefed wchar_t (compilation flag /Zc:wchar_t- for Visual Studio).
  - Better performance and less memory usage for `boost::container::basic_string` conversions.
- **boost 1.48.0 :**
  - Added code to work with Inf and NaN on any platform.
  - Better performance and less memory usage for conversions to float type (and to double type, if `sizeof(double) < sizeof(long double)`).
- **boost 1.47.0 :**
  - Optimizations for "C" and other locales without number grouping.
  - Better performance and less memory usage for unsigned char and signed char conversions.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  - Better performance and less memory usage for conversions to arithmetic types.
  - Better performance and less memory usage for conversions from arithmetic type to arithmetic type.
  - Directly construct Target from Source on some conversions (like conversions from string to string, from char array to string, from char to char and others).
- **boost 1.34.0 :**
  - Better performance for many combinations of Source and Target types. For more details refer to Alexander Nasonovs article Fine Tuning for lexical_cast, Overload #74, August 2006 (PDF).
- **boost 1.33.0 :**
  - Call-by-const reference for the parameters. This requires partial specialization of class templates, so it doesn't work for MSVC 6, and it uses the original pass by value there.
  - The MSVC 6 support is deprecated, and will be removed in a future Boost version.
- **Earlier :**
  - The previous version of lexical_cast used the default stream precision for reading and writing floating-point numbers. For numerics that have a corresponding specialization of `std::numeric_limits`, the current version now chooses a precision to match.
  - The previous version of lexical_cast did not support conversion to or from any wide-character-based types. For compilers with full language and library support for wide characters, `lexical_cast` now supports conversions from `wchar_t`, `wchar_t *`, and `std::wstring` and to `wchar_t` and `std::wstring`.
  - The previous version of `lexical_cast` assumed that the conventional stream extractor operators were sufficient for reading values. However, string I/O is asymmetric, with the result that spaces play the role of I/O separators rather than string content. The current version fixes this error for `std::string` and, where supported, `std::wstring`: `lexical_cast<std::string>("Hello, World")` succeeds instead of failing with a `bad_lexical_cast` exception.
  - The previous version of `lexical_cast` allowed unsafe and meaningless conversions to pointers. The current version now throws a `bad_lexical_cast` for conversions to pointers: `lexical_cast<char *>("Goodbye, World")` now throws an exception instead of causing undefined behavior.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Performance

In most cases `boost::lexical_cast` is faster than `scanf`, `printf`, `std::stringstream`. For more detailed info you can look at the tables below.

Tests description

All the tests measure execution speed in milliseconds for 10000 iterations of the following code blocks:

Table 1. Tests source code

Test name	**Code**
lexical_cast	`_out = boost::lexical_cast<OUTTYPE>(_in);`

std::stringstream with construction	`std::stringstream ss;`
	`ss << _in;`
	`if (ss.fail()) throw std::logic_error(descr);`
	`ss >> _out;`
	`if (ss.fail()) throw std::logic_error(descr);`

std::stringstream without construction	`ss << _in; // ss is an instance of std::stringstream`
	`if (ss.fail()) throw std::logic_error(descr);`
	`ss >> _out;`
	`if (ss.fail()) throw std::logic_error(descr);`
	`/* reseting std::stringstream to use it again */`
	`ss.str(std::string());`
	`ss.clear();`

scanf/printf	`typename OUTTYPE::value_type buffer[500];`
	`sprintf( (char*)buffer, conv, _in);`
	`_out = buffer;`

Fastest results are highlitened with "!!! **x** !!!". Do not use this results to compare compilers, because tests were taken on different hardware.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Clang version 3.0 (tags/RELEASE_30/final)

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 2. Performance Table (Clang version 3.0 (tags/RELEASE_30/final))

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
string->char	!!! **<1** !!!	169	9	10
string->signed char	!!! **<1** !!!	108	8	10
string->unsigned char	!!! **<1** !!!	103	9	10
string->int	!!! **6** !!!	117	24	24
string->short	!!! **7** !!!	115	20	24
string->long int	!!! **7** !!!	115	19	22
string->long long	!!! **8** !!!	116	21	23
string->unsigned int	!!! **6** !!!	121	18	23
string->unsigned short	!!! **6** !!!	116	19	22
string->unsigned long int	!!! **7** !!!	117	23	21
string->unsigned long long	!!! **8** !!!	118	19	34
string->float	!!! **13** !!!	201	55	41
string->double	!!! **14** !!!	151	54	41
string->long double	195	231	67	!!! **42** !!!
string->array<char, 50>	!!! **<1** !!!	121	18	12
string->string	!!! **1** !!!	124	27	---
string->container::string	!!! **3** !!!	114	25	---
string->char	7	111	25	!!! **7** !!!
string->signed char	!!! **6** !!!	112	30	26
string->unsigned char	!!! **6** !!!	113	25	24
int->string	!!! **12** !!!	126	36	21
short->string	!!! **11** !!!	135	30	21
long int->string	!!! **11** !!!	128	28	21
long long->string	!!! **12** !!!	126	32	24
unsigned int->string	!!! **11** !!!	131	36	22
unsigned short->string	!!! **11** !!!	130	28	22

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
unsigned long int->string	!!! **11** !!!	130	36	22
unsigned long long->string	!!! **11** !!!	127	43	25
float->string	53	190	83	!!! **41** !!!
double->string	59	197	82	!!! **44** !!!
long double->string	118	229	101	!!! **44** !!!
char*->char	!!! **1** !!!	105	9	9
char*->signed char	!!! **1** !!!	107	10	10
char*->unsigned char	!!! **1** !!!	106	9	11
char*->int	!!! **7** !!!	149	25	24
char*->short	!!! **7** !!!	118	20	22
char*->long int	!!! **9** !!!	117	20	28
char*->long long	!!! **9** !!!	128	23	29
char*->unsigned int	!!! **7** !!!	120	19	23
char*->unsigned short	!!! **7** !!!	125	20	22
char*->unsigned long int	!!! **8** !!!	125	21	24
char*->unsigned long long	!!! **8** !!!	130	19	22
char*->float	!!! **14** !!!	162	56	41
char*->double	!!! **16** !!!	151	54	39
char*->long double	111	176	58	!!! **42** !!!
char*->array<char, 50>	!!! **1** !!!	116	20	17
char*->string	!!! **8** !!!	125	27	---
char*->container::string	!!! **2** !!!	115	26	---
unsigned char*->char	!!! **1** !!!	101	9	9
unsigned char*->signed char	!!! **1** !!!	104	9	11
unsigned char*->unsigned char	!!! **1** !!!	103	9	13

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
unsigned char*->int	!!! **8** !!!	116	20	24
unsigned char*->short	!!! **7** !!!	121	20	26
unsigned char*->long int	!!! **8** !!!	118	20	22
unsigned char*->long long	!!! **8** !!!	122	20	23
unsigned char*->unsigned int	!!! **6** !!!	119	22	23
unsigned char*->unsigned short	!!! **7** !!!	122	20	22
unsigned char*->unsigned long int	!!! **8** !!!	125	21	22
unsigned char*->unsigned long long	!!! **8** !!!	122	19	25
unsigned char*->float	!!! **14** !!!	162	62	37
unsigned char*->double	!!! **15** !!!	151	58	39
unsigned char*->long double	116	156	58	!!! **42** !!!
unsigned char*->array<char, 50>	!!! **1** !!!	122	19	15
unsigned char*->string	!!! **8** !!!	124	27	---
unsigned char*->container::string	!!! **4** !!!	119	25	---
signed char*->char	!!! **1** !!!	107	9	9
signed char*->signed char	!!! **1** !!!	108	10	11
signed char*->unsigned char	!!! **1** !!!	106	9	11
signed char*->int	!!! **7** !!!	122	21	22
signed char*->short	!!! **7** !!!	126	20	22
signed char*->long int	!!! **8** !!!	119	20	23
signed char*->long long	!!! **8** !!!	119	21	26

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
signed char*->unsigned int	!!! **6** !!!	124	18	22
signed char*->unsigned short	!!! **7** !!!	124	21	23
signed char*->unsigned long int	!!! **8** !!!	121	24	23
signed char*->unsigned long long	!!! **8** !!!	122	20	22
signed char*->float	!!! **14** !!!	167	56	37
signed char*->double	!!! **14** !!!	162	53	40
signed char*->long double	110	152	56	!!! **42** !!!
signed char*->array<char, 50>	!!! **1** !!!	117	19	12
signed char*->string	!!! **8** !!!	132	27	---
signed char*->container::string	!!! **4** !!!	116	26	---
iterator_range<char*>->char	!!! **<1** !!!	112	14	9
iterator_range<char*>->signed char	!!! **<1** !!!	107	13	10
iterator_range<char*>->unsigned char	!!! **<1** !!!	145	15	10
iterator_range<char*>->int	!!! **6** !!!	119	22	23
iterator_range<char*>->short	!!! **6** !!!	115	22	23
iterator_range<char*>->long int	!!! **7** !!!	115	25	22
iterator_range<char*>->long long	!!! **7** !!!	117	21	23
iterator_range<char*>->unsigned int	!!! **6** !!!	118	22	22
iterator_range<char*>->unsigned short	!!! **6** !!!	117	24	22

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
iterator_range<char*>->unsigned long int	!!! **7** !!!	124	25	22
iterator_range<char*>->unsigned long long	!!! **7** !!!	119	22	22
iterator_range<char*>->float	!!! **13** !!!	159	42	41
iterator_range<char*>->double	!!! **14** !!!	152	40	40
iterator_range<char*>->long double	113	155	58	!!! **54** !!!
iterator_range<char*>->array<char, 50>	!!! **<1** !!!	127	23	13
iterator_range<char*>->string	!!! **7** !!!	132	30	---
iterator_range<char*>->container::string	!!! **3** !!!	122	24	---
array<char, 50>->char	!!! **<1** !!!	110	9	10
array<char, 50>->signed char	!!! **<1** !!!	119	9	13
array<char, 50>->unsigned char	!!! **<1** !!!	106	13	11
array<char, 50>->int	!!! **6** !!!	131	21	22
array<char, 50>->short	!!! **7** !!!	119	22	28
array<char, 50>->long int	!!! **8** !!!	133	21	26
array<char, 50>->long long	!!! **8** !!!	115	22	23
array<char, 50>->unsigned int	!!! **6** !!!	118	18	22
array<char, 50>->unsigned short	!!! **7** !!!	119	19	22
array<char, 50>->unsigned long int	!!! **7** !!!	118	23	21
array<char, 50>->unsigned long long	!!! **7** !!!	117	20	22

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
array<char, 50>->float	!!! **15** !!!	156	53	36
array<char, 50>->double	!!! **15** !!!	148	55	39
array<char, 50>->long double	110	150	56	!!! **41** !!!
array<char, 50>->array<char, 50>	!!! **<1** !!!	117	19	12
array<char, 50>->string	!!! **7** !!!	124	26	---
array<char, 50>->container::string	!!! **4** !!!	115	26	---
int->int	!!! **<1** !!!	117	24	---
float->double	!!! **<1** !!!	245	125	---
char->signed char	!!! **<1** !!!	100	9	---

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

GNU C++ version 4.6.3

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 3. Performance Table (GNU C++ version 4.6.3)

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
string->char	!!! **<1** !!!	142	10	18
string->signed char	!!! **<1** !!!	111	8	10
string->unsigned char	!!! **<1** !!!	101	8	10
string->int	!!! **7** !!!	110	20	24
string->short	!!! **6** !!!	109	20	25
string->long int	!!! **7** !!!	113	19	24
string->long long	!!! **7** !!!	116	24	23
string->unsigned int	!!! **6** !!!	110	19	23
string->unsigned short	!!! **5** !!!	116	18	23
string->unsigned long int	!!! **7** !!!	111	22	23
string->unsigned long long	!!! **7** !!!	108	20	22
string->float	!!! **11** !!!	161	54	38
string->double	!!! **11** !!!	146	56	41
string->long double	113	151	59	!!! **43** !!!
string->array<char, 50>	!!! **<1** !!!	107	18	14
string->string	!!! **2** !!!	127	24	---
string->container::string	!!! **3** !!!	142	26	---
string->char	!!! **7** !!!	110	23	17
string->signed char	!!! **7** !!!	114	23	24
string->unsigned char	!!! **7** !!!	110	25	24
int->string	!!! **12** !!!	127	31	22
short->string	!!! **13** !!!	129	31	22
long int->string	!!! **12** !!!	125	30	22
long long->string	!!! **13** !!!	127	34	24
unsigned int->string	!!! **13** !!!	127	27	21
unsigned short->string	!!! **12** !!!	127	28	22

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
unsigned long int->string	!!! **12** !!!	131	27	22
unsigned long long->string	!!! **12** !!!	125	28	24
float->string	51	200	81	!!! **40** !!!
double->string	56	194	82	!!! **48** !!!
long double->string	65	220	82	!!! **41** !!!
char*->char	!!! **<1** !!!	104	10	9
char*->signed char	!!! **<1** !!!	101	10	11
char*->unsigned char	!!! **<1** !!!	99	10	12
char*->int	!!! **6** !!!	112	23	24
char*->short	!!! **6** !!!	115	21	23
char*->long int	!!! **8** !!!	111	21	24
char*->long long	!!! **9** !!!	112	21	30
char*->unsigned int	!!! **7** !!!	112	22	24
char*->unsigned short	!!! **6** !!!	119	19	23
char*->unsigned long int	!!! **7** !!!	115	22	23
char*->unsigned long long	!!! **7** !!!	115	20	23
char*->float	!!! **12** !!!	153	54	39
char*->double	!!! **12** !!!	153	61	41
char*->long double	108	160	61	!!! **49** !!!
char*->array<char, 50>	!!! **<1** !!!	107	20	14
char*->string	!!! **7** !!!	123	26	---
char*->container::string	!!! **2** !!!	121	24	---
unsigned char*->char	!!! **<1** !!!	97	10	9
unsigned char*->signed char	!!! **<1** !!!	98	10	12
unsigned char*->unsigned char	!!! **<1** !!!	99	11	12

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
unsigned char*->int	!!! **6** !!!	112	22	24
unsigned char*->short	!!! **10** !!!	111	24	24
unsigned char*->long int	!!! **8** !!!	110	23	24
unsigned char*->long long	!!! **9** !!!	115	21	25
unsigned char*->unsigned int	!!! **6** !!!	111	24	23
unsigned char*->unsigned short	!!! **6** !!!	118	19	23
unsigned char*->unsigned long int	!!! **8** !!!	112	21	23
unsigned char*->unsigned long long	!!! **13** !!!	109	20	23
unsigned char*->float	!!! **12** !!!	154	56	39
unsigned char*->double	!!! **17** !!!	150	58	41
unsigned char*->long double	108	149	68	!!! **43** !!!
unsigned char*->array<char, 50>	!!! **1** !!!	107	19	15
unsigned char*->string	!!! **8** !!!	124	26	---
unsigned char*->container::string	!!! **4** !!!	121	24	---
signed char*->char	!!! **<1** !!!	99	10	9
signed char*->signed char	!!! **<1** !!!	99	10	10
signed char*->unsigned char	!!! **<1** !!!	99	10	12
signed char*->int	!!! **6** !!!	113	28	24
signed char*->short	!!! **6** !!!	110	21	25
signed char*->long int	!!! **8** !!!	110	21	24
signed char*->long long	!!! **9** !!!	116	21	24

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
signed char*->unsigned int	!!! **7** !!!	114	21	23
signed char*->unsigned short	!!! **6** !!!	116	20	23
signed char*->unsigned long int	!!! **8** !!!	113	27	23
signed char*->unsigned long long	!!! **8** !!!	110	20	23
signed char*->float	!!! **12** !!!	155	53	44
signed char*->double	!!! **13** !!!	150	60	42
signed char*->long double	108	151	62	!!! **44** !!!
signed char*->array<char, 50>	!!! **1** !!!	107	19	15
signed char*->string	!!! **8** !!!	124	26	---
signed char*->container::string	!!! **4** !!!	121	24	---
iterator_range<char*>->char	!!! **<1** !!!	103	14	10
iterator_range<char*>->signed char	!!! **<1** !!!	102	15	12
iterator_range<char*>->unsigned char	!!! **<1** !!!	102	14	12
iterator_range<char*>->int	!!! **6** !!!	115	23	24
iterator_range<char*>->short	!!! **5** !!!	110	22	24
iterator_range<char*>->long int	!!! **7** !!!	109	22	29
iterator_range<char*>->long long	!!! **7** !!!	111	24	28
iterator_range<char*>->unsigned int	!!! **6** !!!	114	22	23
iterator_range<char*>->unsigned short	!!! **5** !!!	115	20	22

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
iterator_range<char*>->unsigned long int	!!! **7** !!!	123	26	23
iterator_range<char*>->unsigned long long	!!! **7** !!!	110	23	24
iterator_range<char*>->float	!!! **11** !!!	153	38	38
iterator_range<char*>->double	!!! **11** !!!	140	43	40
iterator_range<char*>->long double	108	147	!!! **41** !!!	46
iterator_range<char*>->array<char, 50>	!!! **<1** !!!	109	22	15
iterator_range<char*>->string	!!! **8** !!!	122	29	---
iterator_range<char*>->container::string	!!! **3** !!!	117	23	---
array<char, 50>->char	!!! **<1** !!!	98	10	9
array<char, 50>->signed char	!!! **<1** !!!	99	9	12
array<char, 50>->unsigned char	!!! **<1** !!!	102	9	12
array<char, 50>->int	!!! **6** !!!	119	23	23
array<char, 50>->short	!!! **6** !!!	111	21	26
array<char, 50>->long int	!!! **7** !!!	115	20	28
array<char, 50>->long long	!!! **9** !!!	110	21	26
array<char, 50>->unsigned int	!!! **6** !!!	115	22	23
array<char, 50>->unsigned short	!!! **6** !!!	115	19	23
array<char, 50>->unsigned long int	!!! **7** !!!	118	23	23
array<char, 50>->unsigned long long	!!! **7** !!!	109	20	24

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
array<char, 50>->float	!!! **12** !!!	160	53	38
array<char, 50>->double	!!! **11** !!!	147	57	41
array<char, 50>->long double	109	154	59	!!! **42** !!!
array<char, 50>->array<char, 50>	!!! **1** !!!	105	19	14
array<char, 50>->string	!!! **8** !!!	129	26	---
array<char, 50>->container::string	!!! **4** !!!	116	25	---
int->int	!!! **<1** !!!	118	24	---
float->double	!!! **<1** !!!	242	132	---
char->signed char	!!! **<1** !!!	94	8	---

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

GNU C++ version 4.5.3

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 4. Performance Table (GNU C++ version 4.5.3)

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
string->char	!!! **<1** !!!	153	15	9
string->signed char	!!! **<1** !!!	134	8	10
string->unsigned char	!!! **<1** !!!	97	8	14
string->int	!!! **7** !!!	115	22	22
string->short	!!! **5** !!!	112	19	21
string->long int	!!! **7** !!!	110	19	24
string->long long	!!! **7** !!!	115	21	23
string->unsigned int	!!! **6** !!!	113	20	23
string->unsigned short	!!! **5** !!!	116	18	23
string->unsigned long int	!!! **7** !!!	111	20	23
string->unsigned long long	!!! **7** !!!	115	18	23
string->float	!!! **14** !!!	153	55	38
string->double	!!! **11** !!!	151	60	38
string->long double	107	151	59	!!! **44** !!!
string->array<char, 50>	!!! **<1** !!!	107	18	12
string->string	!!! **2** !!!	129	49	---
string->container::string	!!! **9** !!!	199	22	---
string->char	!!! **7** !!!	114	27	16
string->signed char	!!! **7** !!!	116	32	23
string->unsigned char	!!! **7** !!!	114	27	22
int->string	!!! **11** !!!	125	31	21
short->string	!!! **11** !!!	126	33	21
long int->string	!!! **11** !!!	126	32	22
long long->string	!!! **11** !!!	118	30	23
unsigned int->string	!!! **11** !!!	125	31	20
unsigned short->string	!!! **12** !!!	128	30	21

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
unsigned long int->string	!!! **11** !!!	131	30	21
unsigned long long->string	!!! **11** !!!	127	32	23
float->string	49	197	92	!!! **39** !!!
double->string	56	195	80	!!! **43** !!!
long double->string	60	222	88	!!! **42** !!!
char*->char	!!! **<1** !!!	100	10	9
char*->signed char	!!! **<1** !!!	99	10	10
char*->unsigned char	!!! **<1** !!!	106	10	10
char*->int	!!! **7** !!!	113	23	22
char*->short	!!! **6** !!!	113	21	23
char*->long int	!!! **8** !!!	116	21	23
char*->long long	!!! **8** !!!	115	21	21
char*->unsigned int	!!! **6** !!!	114	25	22
char*->unsigned short	!!! **6** !!!	119	20	23
char*->unsigned long int	!!! **8** !!!	114	23	23
char*->unsigned long long	!!! **7** !!!	111	20	24
char*->float	!!! **16** !!!	154	54	38
char*->double	!!! **12** !!!	149	59	40
char*->long double	107	166	62	!!! **44** !!!
char*->array<char, 50>	!!! **1** !!!	108	20	12
char*->string	!!! **8** !!!	125	28	---
char*->container::string	!!! **2** !!!	123	24	---
unsigned char*->char	!!! **<1** !!!	104	11	9
unsigned char*->signed char	!!! **<1** !!!	106	10	10
unsigned char*->unsigned char	!!! **<1** !!!	101	10	10

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
unsigned char*->int	!!! **7** !!!	117	22	24
unsigned char*->short	!!! **6** !!!	111	26	22
unsigned char*->long int	!!! **8** !!!	111	23	23
unsigned char*->long long	!!! **8** !!!	114	21	23
unsigned char*->unsigned int	!!! **7** !!!	115	20	25
unsigned char*->unsigned short	!!! **6** !!!	113	20	22
unsigned char*->unsigned long int	!!! **8** !!!	115	25	24
unsigned char*->unsigned long long	!!! **7** !!!	113	25	25
unsigned char*->float	!!! **16** !!!	158	55	38
unsigned char*->double	!!! **12** !!!	155	62	40
unsigned char*->long double	108	153	60	!!! **41** !!!
unsigned char*->array<char, 50>	!!! **1** !!!	111	19	12
unsigned char*->string	!!! **8** !!!	125	30	---
unsigned char*->container::string	!!! **4** !!!	121	23	---
signed char*->char	!!! **<1** !!!	98	14	9
signed char*->signed char	!!! **<1** !!!	98	11	10
signed char*->unsigned char	!!! **<1** !!!	99	10	10
signed char*->int	!!! **7** !!!	111	22	24
signed char*->short	!!! **6** !!!	123	22	23
signed char*->long int	!!! **8** !!!	112	21	23
signed char*->long long	!!! **8** !!!	114	24	24

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
signed char*->unsigned int	!!! **6** !!!	114	19	22
signed char*->unsigned short	!!! **6** !!!	112	21	24
signed char*->unsigned long int	!!! **8** !!!	114	23	22
signed char*->unsigned long long	!!! **8** !!!	116	22	24
signed char*->float	!!! **16** !!!	156	55	38
signed char*->double	!!! **12** !!!	151	59	39
signed char*->long double	111	159	60	!!! **44** !!!
signed char*->array<char, 50>	!!! **1** !!!	107	24	12
signed char*->string	!!! **8** !!!	122	28	---
signed char*->container::string	!!! **4** !!!	122	23	---
iterator_range<char*>->char	!!! **<1** !!!	103	13	10
iterator_range<char*>->signed char	!!! **<1** !!!	103	13	10
iterator_range<char*>->unsigned char	!!! **<1** !!!	104	14	10
iterator_range<char*>->int	!!! **6** !!!	115	23	24
iterator_range<char*>->short	!!! **7** !!!	111	21	24
iterator_range<char*>->long int	!!! **7** !!!	108	21	23
iterator_range<char*>->long long	!!! **7** !!!	114	24	23
iterator_range<char*>->unsigned int	!!! **6** !!!	111	22	23
iterator_range<char*>->unsigned short	!!! **5** !!!	114	20	23

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
iterator_range<char*>->unsigned long int	!!! **7** !!!	119	25	24
iterator_range<char*>->unsigned long long	!!! **7** !!!	110	20	24
iterator_range<char*>->float	!!! **15** !!!	148	38	40
iterator_range<char*>->double	!!! **10** !!!	146	41	40
iterator_range<char*>->long double	103	138	!!! **39** !!!	42
iterator_range<char*>->array<char, 50>	!!! **<1** !!!	109	22	13
iterator_range<char*>->string	!!! **7** !!!	121	32	---
iterator_range<char*>->container::string	!!! **3** !!!	120	24	---
array<char, 50>->char	!!! **<1** !!!	102	9	9
array<char, 50>->signed char	!!! **<1** !!!	97	9	10
array<char, 50>->unsigned char	!!! **<1** !!!	99	9	10
array<char, 50>->int	!!! **7** !!!	114	22	23
array<char, 50>->short	!!! **6** !!!	116	21	23
array<char, 50>->long int	!!! **7** !!!	109	20	23
array<char, 50>->long long	!!! **7** !!!	114	21	23
array<char, 50>->unsigned int	!!! **7** !!!	119	20	25
array<char, 50>->unsigned short	!!! **6** !!!	120	20	23
array<char, 50>->unsigned long int	!!! **7** !!!	113	20	21
array<char, 50>->unsigned long long	!!! **7** !!!	112	20	24

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
array<char, 50>->float	!!! **16** !!!	155	57	38
array<char, 50>->double	!!! **11** !!!	152	59	42
array<char, 50>->long double	107	152	60	!!! **41** !!!
array<char, 50>->array<char, 50>	!!! **1** !!!	111	20	12
array<char, 50>->string	!!! **8** !!!	123	36	---
array<char, 50>->container::string	!!! **4** !!!	128	23	---
int->int	!!! **<1** !!!	118	26	---
float->double	!!! **<1** !!!	233	120	---
char->signed char	!!! **<1** !!!	97	8	---

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

GNU C++ version 4.4.7

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 5. Performance Table (GNU C++ version 4.4.7)

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
string->char	!!! **<1** !!!	111	8	9
string->signed char	!!! **<1** !!!	100	8	10
string->unsigned char	!!! **<1** !!!	102	8	11
string->int	!!! **6** !!!	114	21	23
string->short	!!! **5** !!!	120	21	29
string->long int	!!! **7** !!!	114	22	26
string->long long	!!! **7** !!!	118	21	23
string->unsigned int	!!! **7** !!!	115	21	23
string->unsigned short	!!! **5** !!!	119	18	22
string->unsigned long int	!!! **7** !!!	115	20	23
string->unsigned long long	!!! **9** !!!	116	26	24
string->float	!!! **12** !!!	165	53	40
string->double	!!! **12** !!!	154	54	40
string->long double	112	148	61	!!! **45** !!!
string->array<char, 50>	!!! **<1** !!!	120	19	14
string->string	!!! **2** !!!	141	55	---
string->container::string	!!! **2** !!!	164	36	---
string->char	!!! **7** !!!	161	24	18
string->signed char	!!! **6** !!!	109	25	24
string->unsigned char	!!! **6** !!!	109	25	25
int->string	!!! **11** !!!	128	32	23
short->string	!!! **12** !!!	136	54	34
long int->string	!!! **15** !!!	187	41	23
long long->string	!!! **11** !!!	128	30	29
unsigned int->string	!!! **13** !!!	124	29	23
unsigned short->string	!!! **11** !!!	128	30	22

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
unsigned long int->string	!!! **11** !!!	131	30	22
unsigned long long->string	!!! **11** !!!	133	33	29
float->string	52	187	90	!!! **39** !!!
double->string	58	190	86	!!! **45** !!!
long double->string	70	218	88	!!! **47** !!!
char*->char	!!! **<1** !!!	99	11	9
char*->signed char	!!! **<1** !!!	99	11	10
char*->unsigned char	!!! **<1** !!!	100	12	10
char*->int	!!! **6** !!!	117	23	21
char*->short	!!! **6** !!!	115	28	23
char*->long int	!!! **7** !!!	119	22	24
char*->long long	!!! **7** !!!	114	23	22
char*->unsigned int	!!! **6** !!!	113	21	21
char*->unsigned short	!!! **6** !!!	120	21	21
char*->unsigned long int	!!! **7** !!!	117	25	23
char*->unsigned long long	!!! **7** !!!	119	23	21
char*->float	!!! **13** !!!	160	61	36
char*->double	!!! **13** !!!	152	54	40
char*->long double	116	173	58	!!! **43** !!!
char*->array<char, 50>	!!! **1** !!!	121	20	12
char*->string	!!! **7** !!!	126	29	---
char*->container::string	!!! **2** !!!	119	27	---
unsigned char*->char	!!! **<1** !!!	96	12	9
unsigned char*->signed char	!!! **<1** !!!	95	11	12
unsigned char*->unsigned char	!!! **<1** !!!	95	12	12

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
unsigned char*->int	!!! **6** !!!	113	27	24
unsigned char*->short	!!! **6** !!!	120	23	21
unsigned char*->long int	!!! **7** !!!	114	22	23
unsigned char*->long long	!!! **7** !!!	114	23	23
unsigned char*->unsigned int	!!! **6** !!!	115	23	23
unsigned char*->unsigned short	!!! **6** !!!	120	21	23
unsigned char*->unsigned long int	!!! **7** !!!	117	23	21
unsigned char*->unsigned long long	!!! **7** !!!	121	23	21
unsigned char*->float	!!! **12** !!!	161	58	39
unsigned char*->double	!!! **13** !!!	153	54	38
unsigned char*->long double	110	150	62	!!! **43** !!!
unsigned char*->array<char, 50>	!!! **1** !!!	113	20	12
unsigned char*->string	!!! **8** !!!	124	30	---
unsigned char*->container::string	!!! **3** !!!	118	27	---
signed char*->char	!!! **<1** !!!	99	11	9
signed char*->signed char	!!! **<1** !!!	102	12	10
signed char*->unsigned char	!!! **<1** !!!	99	12	10
signed char*->int	!!! **6** !!!	114	30	23
signed char*->short	!!! **6** !!!	118	23	23
signed char*->long int	!!! **7** !!!	119	22	21
signed char*->long long	!!! **7** !!!	114	23	26

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
signed char*->unsigned int	!!! **6** !!!	114	26	23
signed char*->unsigned short	!!! **6** !!!	121	22	23
signed char*->unsigned long int	!!! **7** !!!	126	23	21
signed char*->unsigned long long	!!! **7** !!!	114	22	21
signed char*->float	!!! **12** !!!	163	57	39
signed char*->double	!!! **13** !!!	156	53	40
signed char*->long double	112	156	56	!!! **42** !!!
signed char*->array<char, 50>	!!! **1** !!!	117	20	12
signed char*->string	!!! **8** !!!	127	28	---
signed char*->container::string	!!! **4** !!!	112	27	---
iterator_range<char*>->char	!!! **<1** !!!	103	14	9
iterator_range<char*>->signed char	!!! **<1** !!!	104	16	10
iterator_range<char*>->unsigned char	!!! **<1** !!!	103	16	10
iterator_range<char*>->int	!!! **6** !!!	121	22	21
iterator_range<char*>->short	!!! **7** !!!	112	23	23
iterator_range<char*>->long int	!!! **7** !!!	115	24	23
iterator_range<char*>->long long	!!! **7** !!!	113	24	23
iterator_range<char*>->unsigned int	!!! **6** !!!	117	26	23
iterator_range<char*>->unsigned short	!!! **5** !!!	120	20	23

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
iterator_range<char*>->unsigned long int	!!! **7** !!!	124	28	21
iterator_range<char*>->unsigned long long	!!! **7** !!!	113	22	21
iterator_range<char*>->float	!!! **11** !!!	190	58	63
iterator_range<char*>->double	!!! **20** !!!	194	44	39
iterator_range<char*>->long double	116	145	46	!!! **44** !!!
iterator_range<char*>->array<char, 50>	!!! **<1** !!!	116	23	15
iterator_range<char*>->string	!!! **7** !!!	127	33	---
iterator_range<char*>->container::string	!!! **3** !!!	112	24	---
array<char, 50>->char	!!! **<1** !!!	98	11	10
array<char, 50>->signed char	!!! **<1** !!!	99	12	15
array<char, 50>->unsigned char	!!! **<1** !!!	100	11	10
array<char, 50>->int	!!! **6** !!!	114	27	22
array<char, 50>->short	!!! **5** !!!	113	23	23
array<char, 50>->long int	!!! **7** !!!	118	22	23
array<char, 50>->long long	!!! **7** !!!	114	26	23
array<char, 50>->unsigned int	!!! **6** !!!	113	27	23
array<char, 50>->unsigned short	!!! **5** !!!	124	21	23
array<char, 50>->unsigned long int	!!! **7** !!!	116	23	21
array<char, 50>->unsigned long long	!!! **7** !!!	115	22	21

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
array<char, 50>->float	!!! **11** !!!	162	58	36
array<char, 50>->double	!!! **13** !!!	155	54	44
array<char, 50>->long double	111	149	55	!!! **42** !!!
array<char, 50>->array<char, 50>	!!! **1** !!!	114	18	14
array<char, 50>->string	!!! **7** !!!	129	29	---
array<char, 50>->container::string	!!! **3** !!!	113	26	---
int->int	!!! **<1** !!!	114	25	---
float->double	!!! **<1** !!!	236	121	---
char->signed char	!!! **<1** !!!	97	8	---

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Microsoft Visual C++ version 11.0

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 6. Performance Table (Microsoft Visual C++ version 11.0)

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
string->char	!!! **<1** !!!	43	17	7
string->signed char	!!! **<1** !!!	43	17	8
string->unsigned char	!!! **<1** !!!	42	17	8
string->int	!!! **8** !!!	71	49	10
string->short	!!! **8** !!!	72	47	10
string->long int	!!! **8** !!!	71	47	10
string->long long	!!! **8** !!!	71	47	10
string->unsigned int	!!! **8** !!!	72	46	10
string->unsigned short	!!! **8** !!!	71	47	10
string->unsigned long int	!!! **8** !!!	70	45	10
string->unsigned long long	!!! **8** !!!	70	46	10
string->float	!!! **14** !!!	586	559	37
string->double	601	618	592	!!! **37** !!!
string->long double	629	645	618	!!! **37** !!!
string->array<char, 50>	!!! **<1** !!!	52	28	11
string->string	!!! **1** !!!	59	34	---
string->container::string	!!! **2** !!!	54	31	---
string->char	!!! **2** !!!	50	24	9
string->signed char	!!! **2** !!!	50	24	13
string->unsigned char	!!! **2** !!!	50	24	13
int->string	!!! **9** !!!	86	59	13
short->string	!!! **9** !!!	86	59	13
long int->string	!!! **9** !!!	87	59	13
long long->string	!!! **9** !!!	88	62	13
unsigned int->string	!!! **9** !!!	87	60	13
unsigned short->string	!!! **9** !!!	91	63	13

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
unsigned long int->string	!!! **9** !!!	91	62	13
unsigned long long->string	!!! **9** !!!	88	60	13
float->string	73	167	137	!!! **56** !!!
double->string	77	176	144	!!! **64** !!!
long double->string	79	175	143	!!! **63** !!!
char*->char	!!! **<1** !!!	43	17	7
char*->signed char	!!! **<1** !!!	43	17	8
char*->unsigned char	!!! **<1** !!!	44	17	8
char*->int	!!! **8** !!!	70	47	10
char*->short	!!! **8** !!!	72	48	10
char*->long int	!!! **8** !!!	72	47	10
char*->long long	!!! **8** !!!	71	47	10
char*->unsigned int	!!! **8** !!!	72	46	10
char*->unsigned short	!!! **8** !!!	72	47	10
char*->unsigned long int	!!! **8** !!!	70	46	10
char*->unsigned long long	!!! **8** !!!	70	45	10
char*->float	!!! **14** !!!	586	560	37
char*->double	598	617	597	!!! **40** !!!
char*->long double	635	653	622	!!! **37** !!!
char*->array<char, 50>	!!! **1** !!!	53	28	11
char*->string	!!! **1** !!!	59	35	---
char*->container::string	!!! **3** !!!	54	30	---
unsigned char*->char	!!! **<1** !!!	41	17	7
unsigned char*->signed char	!!! **<1** !!!	42	17	8
unsigned char*->unsigned char	!!! **<1** !!!	41	17	8

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
unsigned char*->int	!!! **8** !!!	72	47	10
unsigned char*->short	!!! **8** !!!	72	47	10
unsigned char*->long int	!!! **8** !!!	72	47	10
unsigned char*->long long	!!! **8** !!!	72	47	11
unsigned char*->unsigned int	!!! **8** !!!	70	46	10
unsigned char*->unsigned short	!!! **8** !!!	72	48	10
unsigned char*->unsigned long int	!!! **8** !!!	71	46	10
unsigned char*->unsigned long long	!!! **8** !!!	70	45	11
unsigned char*->float	!!! **14** !!!	589	564	38
unsigned char*->double	601	615	588	!!! **37** !!!
unsigned char*->long double	628	644	620	!!! **38** !!!
unsigned char*->array<char, 50>	!!! **1** !!!	54	28	11
unsigned char*->string	!!! **2** !!!	59	36	---
unsigned char*->container::string	!!! **3** !!!	54	30	---
signed char*->char	!!! **<1** !!!	41	17	7
signed char*->signed char	!!! **<1** !!!	43	17	8
signed char*->unsigned char	!!! **<1** !!!	42	17	8
signed char*->int	!!! **8** !!!	71	47	10
signed char*->short	!!! **8** !!!	72	48	10
signed char*->long int	!!! **8** !!!	71	47	10
signed char*->long long	!!! **8** !!!	72	47	10

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
signed char*->unsigned int	!!! **8** !!!	70	46	10
signed char*->unsigned short	!!! **8** !!!	72	47	10
signed char*->unsigned long int	!!! **8** !!!	70	46	10
signed char*->unsigned long long	!!! **8** !!!	70	46	11
signed char*->float	!!! **14** !!!	586	562	37
signed char*->double	603	615	589	!!! **37** !!!
signed char*->long double	630	644	623	!!! **40** !!!
signed char*->array<char, 50>	!!! **1** !!!	54	28	11
signed char*->string	!!! **2** !!!	59	36	---
signed char*->container::string	!!! **3** !!!	54	30	---
iterator_range<char*>->char	!!! **<1** !!!	74	46	7
iterator_range<char*>->signed char	!!! **<1** !!!	75	46	8
iterator_range<char*>->unsigned char	!!! **<1** !!!	74	46	8
iterator_range<char*>->int	!!! **8** !!!	98	70	10
iterator_range<char*>->short	!!! **8** !!!	103	72	10
iterator_range<char*>->long int	!!! **8** !!!	111	71	10
iterator_range<char*>->long long	!!! **8** !!!	98	70	10
iterator_range<char*>->unsigned int	!!! **7** !!!	103	76	10
iterator_range<char*>->unsigned short	!!! **8** !!!	104	75	10

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
iterator_range<char*>->unsigned long int	!!! **7** !!!	104	71	10
iterator_range<char*>->unsigned long long	!!! **8** !!!	99	71	11
iterator_range<char*>->float	!!! **13** !!!	123	93	37
iterator_range<char*>->double	603	111	82	!!! **38** !!!
iterator_range<char*>->long double	629	116	83	!!! **38** !!!
iterator_range<char*>->array<char, 50>	!!! **<1** !!!	82	52	11
iterator_range<char*>->string	!!! **2** !!!	83	56	---
iterator_range<char*>->container::string	!!! **2** !!!	81	53	---
array<char, 50>->char	!!! **<1** !!!	41	17	7
array<char, 50>->signed char	!!! **<1** !!!	41	17	8
array<char, 50>->unsigned char	!!! **<1** !!!	41	17	8
array<char, 50>->int	!!! **8** !!!	73	46	10
array<char, 50>->short	!!! **8** !!!	73	47	10
array<char, 50>->long int	!!! **8** !!!	75	48	10
array<char, 50>->long long	!!! **8** !!!	73	48	11
array<char, 50>->unsigned int	!!! **8** !!!	73	47	10
array<char, 50>->unsigned short	!!! **8** !!!	74	50	10
array<char, 50>->unsigned long int	!!! **8** !!!	71	46	10
array<char, 50>->unsigned long long	!!! **8** !!!	70	47	11

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

From->To	**lexical_cast**	**std::stringstream with construction**	**std::stringstream without construction**	**scanf/printf**
array<char, 50>->float	!!! **14** !!!	586	567	37
array<char, 50>->double	599	624	590	!!! **37** !!!
array<char, 50>->long double	632	643	618	!!! **37** !!!
array<char, 50>->array<char, 50>	!!! **1** !!!	52	28	11
array<char, 50>->string	!!! **2** !!!	59	34	---
array<char, 50>->container::string	!!! **3** !!!	55	30	---
int->int	!!! **<1** !!!	105	79	---
float->double	!!! **<1** !!!	226	188	---
char->signed char	!!! **<1** !!!	40	16	---

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/