

---

# Boost.Lexical\_Cast 1.0

Copyright © 2000-2005 Kevlin Henney  
Copyright © 2006-2010 Alexander Nasonov  
Copyright © 2011 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](http://www.boost.org/LICENSE_1_0.txt))

## Table of Contents

Motivation .....	2
Examples .....	3
Synopsis .....	4
lexical_cast .....	4
bad_lexical_cast .....	4
Frequently Asked Questions .....	5
Changes .....	6
Performance .....	7
Tests description .....	7
clang-linux-2.8 .....	8
gcc-4.4 .....	13
gcc-4.5 .....	18
gcc-4.6 .....	23

## 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.

## Examples

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

```
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(bad_lexical_cast &)
        {
            args.push_back(0);
        }
        ...
    }
}
```

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));
}
```

# 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);
}
```

## 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.

The requirements on the argument and result types 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`. Source types that require wide-character streaming are `wchar_t`, `wchar_t *`, and `std::wstring`. Target types that require wide-character streaming are `wchar_t` and `std::wstring`.

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.

## 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>>` conform to the standard. 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.

# Changes

- **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.
  - 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.

# 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	<pre>_out = boost::lexical_cast&lt;OUTTYPE&gt;(_in);</pre>
std::stringstream with construction	<pre>std::stringstream ss; ss &lt;&lt; _in; if (ss.fail()) throw std::logic_error(descr); ss &gt;&gt; _out; if (ss.fail()) throw std::logic_error(descr);</pre>
std::stringstream without construction	<pre>ss &lt;&lt; _in; // ss is an instance of std::string stream if (ss.fail()) throw std::logic_error(descr); ss &gt;&gt; _out; if (ss.fail()) throw std::logic_error(descr); /* resetting std::stringstream to use it again */ ss.str(std::string()); ss.clear();</pre>
scanf/printf	<pre>typename OUTTYPE::value_type buffer[500]; sprintf( (char*)buffer, conv, _in); _out = buffer;</pre>

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

## clang-linux-2.8



**Table 2. Performance Table (clang-linux-2.8)**

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
string->char	!!! <1 !!!	148	14	12
string->signed char	!!! <1 !!!	97	8	7
string->unsigned char	!!! <1 !!!	90	8	13
string->int	!!! 4 !!!	102	19	15
string->short	!!! 4 !!!	105	20	15
string->long int	!!! 4 !!!	105	19	15
string->long long	!!! 4 !!!	115	19	14
string->unsigned int	!!! 4 !!!	102	18	14
string->unsigned short	!!! 4 !!!	101	19	15
string->unsigned long int	!!! 3 !!!	107	20	14
string->unsigned long long	!!! 3 !!!	103	20	14
string->bool	!!! <1 !!!	97	16	8
string->float	!!! 21 !!!	170	61	32
string->double	!!! 18 !!!	206	93	58
string->long double	135	221	94	!!! 57 !!!
char->string	!!! 7 !!!	100	17	13
unsigned char->string	!!! 7 !!!	99	18	16
signed char->string	!!! 7 !!!	101	17	12
int->string	!!! 13 !!!	110	23	15
short->string	!!! 13 !!!	112	24	18
long int->string	!!! 13 !!!	119	23	17
long long->string	!!! 13 !!!	110	23	18
unsigned int->string	!!! 14 !!!	113	24	17
unsigned short->string	!!! 13 !!!	108	24	17
unsigned long int->string	!!! 13 !!!	109	24	16

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
unsigned long long->string	!!! 13 !!!	110	23	17
bool->string	!!! 7 !!!	105	24	12
float->string	70	192	94	!!! 49 !!!
double->string	106	217	122	!!! 76 !!!
long double->string	120	219	123	!!! 80 !!!
char*->char	!!! 2 !!!	90	9	8
char*->signed char	!!! 2 !!!	87	10	7
char*->unsigned char	!!! 3 !!!	90	10	13
char*->int	!!! 6 !!!	107	21	15
char*->short	!!! 6 !!!	110	19	14
char*->long int	!!! 6 !!!	103	19	14
char*->long long	!!! 7 !!!	104	20	15
char*->unsigned int	!!! 6 !!!	101	20	15
char*->unsigned short	!!! 7 !!!	100	20	14
char*->unsigned long int	!!! 6 !!!	105	22	15
char*->unsigned long long	!!! 7 !!!	106	21	14
char*->bool	!!! 2 !!!	99	18	7
char*->float	!!! 22 !!!	159	67	33
char*->double	!!! 20 !!!	205	94	58
char*->long double	140	214	95	!!! 58 !!!
unsigned char*->char	!!! 2 !!!	92	9	7
unsigned char*->signed char	!!! 2 !!!	89	10	7
unsigned char*->unsigned char	!!! 2 !!!	89	10	14
unsigned char*->int	!!! 6 !!!	104	20	14
unsigned char*->short	!!! 6 !!!	106	21	14

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
unsigned char*->long int	!!! 6 !!!	105	19	14
unsigned char*->long long	!!! 6 !!!	106	20	15
unsigned char*->unsigned int	!!! 7 !!!	105	19	14
unsigned char*->unsigned short	!!! 6 !!!	103	19	14
unsigned char*->unsigned long int	!!! 6 !!!	106	19	14
unsigned char*->unsigned long long	!!! 6 !!!	104	21	15
unsigned char*->bool	!!! 2 !!!	102	18	7
unsigned char*->float	!!! 23 !!!	160	66	32
unsigned char*->double	!!! 20 !!!	201	95	58
unsigned char*->long double	144	221	95	!!! 60 !!!
unsigned char*->string	!!! 12 !!!	104	23	---
signed char*->char	!!! 2 !!!	90	9	7
signed char*->signed char	!!! 2 !!!	89	9	7
signed char*->unsigned char	!!! 2 !!!	89	10	13
signed char*->int	!!! 6 !!!	106	19	15
signed char*->short	!!! 6 !!!	107	20	15
signed char*->long int	!!! 6 !!!	103	19	14
signed char*->long long	!!! 6 !!!	103	19	14
signed char*->unsigned int	!!! 6 !!!	101	19	15
signed char*->unsigned short	!!! 6 !!!	101	19	16
signed char*->unsigned long int	!!! 6 !!!	105	22	15

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
signed char*->unsigned long long	!!! 6 !!!	104	21	15
signed char*->bool	!!! 2 !!!	100	18	7
signed char*->float	!!! 23 !!!	161	62	32
signed char*->double	!!! 20 !!!	207	102	57
signed char*->long double	144	216	96	!!! 63 !!!
signed char*->string	!!! 12 !!!	104	23	---
int->int	!!! <1 !!!	110	22	---
float->double	!!! <1 !!!	223	113	---
double->double	!!! <1 !!!	227	111	---
int->int	!!! <1 !!!	231	122	---
int->int	!!! <1 !!!	229	121	---
char->unsigned char	!!! <1 !!!	90	8	---
char->signed char	!!! <1 !!!	88	8	---
unsigned char->char	!!! <1 !!!	89	8	---
signed char->char	!!! <1 !!!	91	9	---

**gcc-4.4**

**Table 3. Performance Table (gcc-4.4)**

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
string->char	!!! <1 !!!	90	7	7
string->signed char	!!! <1 !!!	88	7	8
string->unsigned char	!!! <1 !!!	88	8	14
string->int	!!! 3 !!!	103	18	15
string->short	!!! 3 !!!	105	20	15
string->long int	!!! 3 !!!	101	18	16
string->long long	!!! 3 !!!	101	18	15
string->unsigned int	!!! 3 !!!	98	23	14
string->unsigned short	!!! 3 !!!	100	17	14
string->unsigned long int	!!! 3 !!!	100	21	15
string->unsigned long long	!!! 3 !!!	99	19	15
string->bool	!!! <1 !!!	95	16	8
string->float	!!! 13 !!!	160	61	33
string->double	!!! 14 !!!	206	93	59
string->long double	128	217	96	!!! 61 !!!
char->string	!!! 7 !!!	100	17	12
unsigned char->string	!!! 7 !!!	109	17	16
signed char->string	!!! 7 !!!	99	17	12
int->string	!!! 13 !!!	110	21	15
short->string	!!! 14 !!!	110	22	17
long int->string	!!! 14 !!!	109	21	16
long long->string	!!! 13 !!!	114	20	17
unsigned int->string	!!! 13 !!!	109	23	15
unsigned short->string	!!! 14 !!!	109	23	17
unsigned long int->string	!!! 13 !!!	112	23	16

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
unsigned long long->string	!!! 14 !!!	109	21	17
bool->string	!!! 7 !!!	108	23	11
float->string	63	185	92	!!! 50 !!!
double->string	106	216	116	!!! 75 !!!
long double->string	118	219	119	!!! 80 !!!
char*->char	!!! 1 !!!	93	9	9
char*->signed char	!!! 1 !!!	92	9	9
char*->unsigned char	!!! 1 !!!	92	9	14
char*->int	!!! 4 !!!	107	19	15
char*->short	!!! 5 !!!	109	19	15
char*->long int	!!! 4 !!!	113	19	15
char*->long long	!!! 4 !!!	108	20	15
char*->unsigned int	!!! 4 !!!	106	19	15
char*->unsigned short	!!! 4 !!!	106	18	15
char*->unsigned long int	!!! 4 !!!	103	22	15
char*->unsigned long long	!!! 4 !!!	105	20	15
char*->bool	!!! 1 !!!	104	18	8
char*->float	!!! 15 !!!	164	62	33
char*->double	!!! 16 !!!	203	97	58
char*->long double	132	223	98	!!! 60 !!!
unsigned char*->char	!!! 2 !!!	90	9	8
unsigned char*->signed char	!!! 2 !!!	92	10	8
unsigned char*->unsigned char	!!! 2 !!!	91	9	14
unsigned char*->int	!!! 6 !!!	106	20	15
unsigned char*->short	!!! 6 !!!	106	21	15

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
unsigned char*->long int	!!! 6 !!!	111	19	15
unsigned char*->long long	!!! 6 !!!	107	20	15
unsigned char*->unsigned int	!!! 6 !!!	105	19	15
unsigned char*->unsigned short	!!! 6 !!!	103	18	15
unsigned char*->unsigned long int	!!! 6 !!!	106	22	14
unsigned char*->unsigned long long	!!! 6 !!!	105	20	14
unsigned char*->bool	!!! 2 !!!	106	18	8
unsigned char*->float	!!! 15 !!!	167	68	33
unsigned char*->double	!!! 17 !!!	203	99	58
unsigned char*->long double	129	216	97	!!! 61 !!!
unsigned char*->string	!!! 13 !!!	111	23	---
signed char*->char	!!! 2 !!!	92	9	8
signed char*->signed char	!!! 2 !!!	91	9	8
signed char*->unsigned char	!!! 2 !!!	91	9	14
signed char*->int	!!! 6 !!!	107	19	15
signed char*->short	!!! 6 !!!	109	24	14
signed char*->long int	!!! 6 !!!	112	19	15
signed char*->long long	!!! 5 !!!	107	20	15
signed char*->unsigned int	!!! 6 !!!	108	20	15
signed char*->unsigned short	!!! 6 !!!	104	18	15
signed char*->unsigned long int	!!! 6 !!!	102	22	15



From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
signed char*->unsigned long long	!!! <b>6</b> !!!	104	20	15
signed char*->bool	!!! <b>2</b> !!!	104	18	8
signed char*->float	!!! <b>16</b> !!!	165	63	33
signed char*->double	!!! <b>16</b> !!!	203	98	59
signed char*->long double	129	215	98	!!! <b>61</b> !!!
signed char*->string	!!! <b>13</b> !!!	109	21	---
int->int	!!! <b>&lt;1</b> !!!	109	21	---
float->double	!!! <b>&lt;1</b> !!!	221	102	---
double->double	!!! <b>&lt;1</b> !!!	223	103	---
int->int	!!! <b>&lt;1</b> !!!	231	115	---
int->int	!!! <b>&lt;1</b> !!!	231	115	---
char->unsigned char	!!! <b>&lt;1</b> !!!	92	8	---
char->signed char	!!! <b>&lt;1</b> !!!	88	8	---
unsigned char->char	!!! <b>&lt;1</b> !!!	88	7	---
signed char->char	!!! <b>&lt;1</b> !!!	89	8	---

**gcc-4.5**

**Table 4. Performance Table (gcc-4.5)**

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
string->char	!!! <1 !!!	91	8	7
string->signed char	!!! <1 !!!	91	8	7
string->unsigned char	!!! <1 !!!	90	8	13
string->int	!!! 3 !!!	100	20	14
string->short	!!! 3 !!!	106	20	14
string->long int	!!! 3 !!!	100	18	14
string->long long	!!! 9 !!!	100	18	15
string->unsigned int	!!! 3 !!!	97	20	14
string->unsigned short	!!! 3 !!!	102	17	14
string->unsigned long int	!!! 3 !!!	97	21	14
string->unsigned long long	!!! 3 !!!	97	19	14
string->bool	!!! <1 !!!	95	16	7
string->float	!!! 15 !!!	157	63	32
string->double	!!! 17 !!!	203	95	59
string->long double	129	216	93	!!! 58 !!!
char->string	!!! 8 !!!	100	17	10
unsigned char->string	!!! 8 !!!	96	18	16
signed char->string	!!! 8 !!!	96	18	10
int->string	!!! 14 !!!	105	22	15
short->string	!!! 14 !!!	107	23	17
long int->string	!!! 14 !!!	109	22	17
long long->string	!!! 14 !!!	105	22	18
unsigned int->string	!!! 14 !!!	105	25	15
unsigned short->string	!!! 15 !!!	105	23	17
unsigned long int->string	!!! 14 !!!	109	24	17

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
unsigned long long->string	!!! 14 !!!	102	23	17
bool->string	!!! 8 !!!	104	23	12
float->string	66	181	92	!!! 49 !!!
double->string	107	215	120	!!! 75 !!!
long double->string	117	221	125	!!! 79 !!!
char*->char	!!! 1 !!!	89	9	7
char*->signed char	!!! 1 !!!	90	9	7
char*->unsigned char	!!! 2 !!!	90	9	13
char*->int	!!! 7 !!!	103	20	15
char*->short	!!! 6 !!!	102	29	14
char*->long int	!!! 7 !!!	101	20	15
char*->long long	!!! 6 !!!	102	20	14
char*->unsigned int	!!! 6 !!!	99	19	14
char*->unsigned short	!!! 6 !!!	101	18	14
char*->unsigned long int	!!! 6 !!!	102	22	14
char*->unsigned long long	!!! 6 !!!	101	21	14
char*->bool	!!! 3 !!!	98	18	7
char*->float	!!! 18 !!!	162	63	31
char*->double	!!! 17 !!!	203	96	58
char*->long double	135	214	98	!!! 58 !!!
unsigned char*->char	!!! 2 !!!	87	9	7
unsigned char*->signed char	!!! 2 !!!	87	9	7
unsigned char*->unsigned char	!!! 3 !!!	87	9	13
unsigned char*->int	!!! 6 !!!	105	20	14
unsigned char*->short	!!! 6 !!!	102	21	14

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
unsigned char*->long int	!!! 6 !!!	101	20	14
unsigned char*->long long	!!! 6 !!!	102	20	14
unsigned char*->unsigned int	!!! 6 !!!	99	19	14
unsigned char*->unsigned short	!!! 6 !!!	100	18	14
unsigned char*->unsigned long int	!!! 6 !!!	101	24	14
unsigned char*->unsigned long long	!!! 6 !!!	100	20	14
unsigned char*->bool	!!! 3 !!!	99	18	8
unsigned char*->float	!!! 17 !!!	164	64	32
unsigned char*->double	!!! 18 !!!	201	94	58
unsigned char*->long double	133	217	95	!!! 60 !!!
unsigned char*->string	!!! 14 !!!	103	23	---
signed char*->char	!!! 3 !!!	88	10	8
signed char*->signed char	!!! 2 !!!	87	10	7
signed char*->unsigned char	!!! 3 !!!	87	9	13
signed char*->int	!!! 6 !!!	104	20	14
signed char*->short	!!! 6 !!!	105	21	14
signed char*->long int	!!! 6 !!!	104	20	15
signed char*->long long	!!! 6 !!!	106	20	14
signed char*->unsigned int	!!! 6 !!!	99	20	14
signed char*->unsigned short	!!! 6 !!!	100	18	14
signed char*->unsigned long int	!!! 6 !!!	102	23	14

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
signed char*->unsigned long long	!!! <b>6</b> !!!	103	20	14
signed char*->bool	!!! <b>3</b> !!!	99	18	7
signed char*->float	!!! <b>18</b> !!!	159	60	32
signed char*->double	!!! <b>18</b> !!!	203	95	57
signed char*->long double	129	213	97	!!! <b>56</b> !!!
signed char*->string	!!! <b>14</b> !!!	105	22	---
int->int	!!! <b>&lt;1</b> !!!	109	22	---
float->double	!!! <b>&lt;1</b> !!!	226	104	---
double->double	!!! <b>&lt;1</b> !!!	229	103	---
int->int	!!! <b>&lt;1</b> !!!	225	115	---
int->int	!!! <b>&lt;1</b> !!!	227	115	---
char->unsigned char	!!! <b>&lt;1</b> !!!	90	8	---
char->signed char	!!! <b>&lt;1</b> !!!	84	8	---
unsigned char->char	!!! <b>&lt;1</b> !!!	88	8	---
signed char->char	!!! <b>&lt;1</b> !!!	89	8	---

**gcc-4.6**

**Table 5. Performance Table (gcc-4.6)**

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
string->char	!!! <1 !!!	94	8	7
string->signed char	!!! <1 !!!	96	9	7
string->unsigned char	!!! <1 !!!	96	8	13
string->int	!!! 3 !!!	110	18	16
string->short	!!! 3 !!!	111	18	16
string->long int	!!! 3 !!!	109	18	15
string->long long	!!! 3 !!!	111	18	15
string->unsigned int	!!! 3 !!!	110	20	15
string->unsigned short	!!! 3 !!!	111	18	15
string->unsigned long int	!!! 3 !!!	109	18	15
string->unsigned long long	!!! 3 !!!	114	19	15
string->bool	!!! <1 !!!	106	17	8
string->float	!!! 13 !!!	175	70	33
string->double	!!! 14 !!!	182	81	58
string->long double	118	190	87	!!! 58 !!!
char->string	!!! 8 !!!	118	21	12
unsigned char->string	!!! 8 !!!	109	18	16
signed char->string	!!! 8 !!!	108	18	12
int->string	20	121	21	!!! 16 !!!
short->string	!!! 15 !!!	120	22	17
long int->string	!!! 15 !!!	120	22	16
long long->string	!!! 15 !!!	120	22	17
unsigned int->string	!!! 15 !!!	120	22	16
unsigned short->string	!!! 15 !!!	120	22	18
unsigned long int->string	16	118	22	!!! 15 !!!



From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
unsigned long long->string	!!! 15 !!!	117	21	17
bool->string	!!! 8 !!!	117	23	10
float->string	77	218	105	!!! 50 !!!
double->string	108	247	129	!!! 73 !!!
long double->string	120	250	131	!!! 79 !!!
char*->char	!!! 2 !!!	99	9	7
char*->signed char	!!! 2 !!!	98	9	8
char*->unsigned char	!!! 2 !!!	98	9	13
char*->int	!!! 6 !!!	115	22	15
char*->short	!!! 6 !!!	114	22	15
char*->long int	!!! 6 !!!	114	22	16
char*->long long	!!! 6 !!!	119	22	15
char*->unsigned int	!!! 6 !!!	114	20	15
char*->unsigned short	!!! 6 !!!	116	20	15
char*->unsigned long int	!!! 6 !!!	117	22	15
char*->unsigned long long	!!! 6 !!!	118	22	15
char*->bool	!!! 3 !!!	113	18	8
char*->float	!!! 15 !!!	180	78	32
char*->double	!!! 16 !!!	185	89	58
char*->long double	119	193	91	!!! 60 !!!
unsigned char*->char	!!! 2 !!!	99	9	8
unsigned char*->signed char	!!! 2 !!!	99	10	8
unsigned char*->unsigned char	!!! 2 !!!	100	9	15
unsigned char*->int	!!! 6 !!!	118	22	15
unsigned char*->short	!!! 6 !!!	117	26	15

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
unsigned char*->long int	!!! 6 !!!	119	21	15
unsigned char*->long long	!!! 6 !!!	118	21	14
unsigned char*->unsigned int	!!! 6 !!!	115	22	14
unsigned char*->unsigned short	!!! 6 !!!	117	20	15
unsigned char*->unsigned long int	!!! 6 !!!	115	21	15
unsigned char*->unsigned long long	!!! 6 !!!	117	22	15
unsigned char*->bool	!!! 3 !!!	112	18	8
unsigned char*->float	!!! 15 !!!	181	78	33
unsigned char*->double	!!! 16 !!!	185	92	59
unsigned char*->long double	120	190	89	!!! 58 !!!
unsigned char*->string	!!! 14 !!!	121	22	---
signed char*->char	!!! 2 !!!	99	9	9
signed char*->signed char	!!! 2 !!!	98	9	8
signed char*->unsigned char	!!! 2 !!!	98	9	14
signed char*->int	!!! 6 !!!	119	22	16
signed char*->short	!!! 6 !!!	115	22	15
signed char*->long int	!!! 6 !!!	119	22	15
signed char*->long long	!!! 6 !!!	117	22	15
signed char*->unsigned int	!!! 6 !!!	117	23	15
signed char*->unsigned short	!!! 6 !!!	117	21	14
signed char*->unsigned long int	!!! 7 !!!	119	24	15

From->To	lexical_cast	std::stringstream with construction	std::stringstream without construction	scanf/printf
signed char*->unsigned long long	!!! <b>6</b> !!!	116	22	15
signed char*->bool	!!! <b>3</b> !!!	111	18	8
signed char*->float	!!! <b>16</b> !!!	180	78	33
signed char*->double	!!! <b>16</b> !!!	185	89	59
signed char*->long double	120	191	91	!!! <b>59</b> !!!
signed char*->string	!!! <b>14</b> !!!	122	23	---
int->int	!!! < <b>1</b> !!!	120	22	---
float->double	!!! < <b>1</b> !!!	242	115	---
double->double	!!! < <b>1</b> !!!	243	115	---
int->int	!!! < <b>1</b> !!!	265	141	---
int->int	!!! < <b>1</b> !!!	266	140	---
char->unsigned char	!!! < <b>1</b> !!!	95	8	---
char->signed char	!!! < <b>1</b> !!!	95	8	---
unsigned char->char	!!! < <b>1</b> !!!	94	8	---
signed char->char	!!! < <b>1</b> !!!	94	8	---