Floating-Point <charconv>: Making Your Code 10x Faster With C++17's Final Boss

Stephan T. Lavavej "Steh-fin Lah-wah-wade"

Principal Software Engineer, Visual C++ Libraries

stl@microsoft.com

@StephanTLavavej

Getting Started

- Please hold your questions until the end
  - Write down the slide numbers
- Open source: github.com/microsoft/STL
- Assumptions:
  - `float` is 32-bit (IEEE 754 single-precision)
  - `double` is 64-bit (IEEE 754 double-precision)
  - `long double` will be ignored here
    - In MSVC's ABI, it's 64-bit
    - Other platforms may need 80-bit or 128-bit codepaths

Floating-Point Basics

Fixed-Point vs. Floating-Point

- Decimal fixed-point, 5 digits:
  - `123.45` stored as `12345` (i.e. `12345 *` 10^{-2})
  - `0.01` stored as `1` (i.e. `1 *` 10^{-2})
  - `999.99` stored as `99999` (i.e. `99999 *` 10^{-2})
- Decimal floating-point, 4 digits and 1 digit:
  - `17.29` stored as `1729` and `-2` (i.e. `1729 *` 10^{-2})
  - `0.000000001` stored as `1` and `-9` (i.e. `1 *` 10^{-9})
  - `9999000000000` stored as `9999` and `9` (i.e. `9999 *` 10^{9})
- Floating-point dramatically increases range
  - Mostly increases precision, but in a variable way

Binary Floating-Point

- Humans ♡ decimal
  - 3.875 = 3875 * 10^{-3}
- Computers ♡ binary
  - 3.875 = 31 * 2^{-3}
  - Exactly representable by storing 31 and -3 (encoded)
- Decimal vs. binary is simple for integers
  - Limited range, e.g. [0, 65535] or [-32768, 32767]
- Decimal vs. binary is surprising for fractions
  - Because math

Base 10 vs. Base 2

- `1/7` has a terminating expansion in base 7: $\texttt{0.1}_7$
  - But not in base 10: `0.142857…`
- Base 10 is neither "fuzzy" nor "non-deterministic"
  - 7 and 10 simply have totally different prime factors
- Decimal/binary compatibility is one-way
  - Decimal can exactly represent every binary fraction
  - Binary can't exactly represent most decimal fractions
- Base 2 is neither "fuzzy" nor "non-deterministic"
  - Binary floating-point values are exact real numbers

Base Conversions

- Vaguely like quantizing an analog signal
- How close can we get to `0.1` in binary?
  - `float: 0.100000001490116119384765625`
  - `double: 0.1000000000000000055511151231257827021181583404541015625`
- How close can we get to `0.01` in binary?
  - `float: 0.009999999776482582092285156 25`
  - `double: 0.01000000000000000020816681711721685132943093776702880859375`
- These are not "garbage digits"

Compilers Are Programs Too

- Value-modifying base conversions:
  - `double x = strtod("0.1", nullptr);`
  - `double y = 0.1; // compiler effectively calls strtod()`
- Converting from decimal usually rounds values
- Converting to decimal rounds values when you ask it to
  - This talk is all about asking for "some digits" or "all digits"
- Floating-point values are exact and crystalline
- Floating-point math isn't covered by this talk
  - Transcendental functions can be inexact, etc.
  - Epsilon comparisons can be useful, etc.

IEEE Representations

- `float`/`double` store: sign bit, exponent bits, mantissa bits
  - Sign bit: `0` is positive, `1` is negative (allows negative zero)
  - Exponent bits: Special values for zero/subnormals/infinity/NaN
  - Also encoded with "exponent bias" to handle negative exponents
  - Mantissa bits: Encoded with "implicit bit" for normals
- These are implementation details; would be a different talk
- Subnormals may be special to FPUs, but not here
  - They also represent exact real numbers
  - Some special-casing in `from_chars()`, almost none in `to_chars()`
- Infinity is easy to handle, NaN is weird
- Hexadecimal floating-point is "human-readable" IEEE

Round-Trip Conversions

printf() Format Examples

- `%f` – fixed notation, like `"1729.531250"`
  - Precision = digits after decimal point, default 6
- `%e` – scientific notation, like `"1.729531e+03"`
  - Precision = digits after decimal point, default 6
- `%g` – general notation, like `"1729.53"`
  - Switches between fixed and scientific, trims zeroes
  - Precision = significant digits, default 6
- `%a` – hexfloat, like `"0x1.b062000000000p+10"`
  - Precision = hexits after decimal point, default "exact"

`printf()` Precision Examples

- Printing `1729.53125`
  - `%.2f` prints `"1729.53"`
  - `%.2e` prints `"1.73e+03"`
  - `%.2g` prints `"1.7e+03"`
  - `%.2a` prints `"0x1.b0p+10"`
- Want all decimal digits? Easy!
  - Use general notation with huge precision, it'll trim zeroes
  - `%.1000g` is sufficiently huge (exactly `%.767g` for `double`)

Rounding

- The CRT supports many rounding modes
  - Default is best: round to nearest, tiebreak to even
  - `<charconv>` always uses this mode
  - Note: MSVC UCRT recently fixed tiebreaking
- Examples with `"%.1f"`:
  - `0.1484375 prints "0.1" (rounds down)`
  - `0.25      prints "0.2" (tie rounds down to even)`
  - `0.75      prints "0.8" (tie rounds up to even)`
  - `0.8515625 prints "0.9" (rounds up)`
  - `0.8671875 prints "0.9" (rounds up)`

Floating-Point Neighbors

- What `floats` are nearest to `0.1`?
  - `0.0999999865889549255371093750`
  - `0.099999994039535522460937500`
  - `0.100000001490116119384765625`
  - `0.10000000894069671630859375`
  - `0.100000016391277313232421875`
- How much can they be rounded and remain distinct?
  - `"0.09999999"`
  - `"0.099999994"`
  - `"0.1"`
  - `"0.10000001"`
  - `"0.10000002"`

Shortest vs. Worst-Case Round-Trip

- Binary-decimal-binary round-trip conversions
  - Can recover all bits given limited decimal digits
- Shortest round-trip needs special algorithms
  - Not available via `printf()`
  - Output is mathematically determined, given representation
- Worst-case round-trip precision is provable
  - 9 significant digits for `float`: `"%.9g"` and `"%.8e"`
  - 17 significant digits for `double`: `"%.17g"` and `"%.16e"`
  - Reported by `numeric_limits<T>::max_digits10`
  - (`digits10` is different, for decimal-binary-decimal)

Shortest Round-Trip Is Nice

- Start with a human-friendly decimal (few digits):
  - `"17.29"`
- Convert to nearest floating-point value (quantize):
  - `17.2900009155273437 5 (float)`
- Print as shortest round-trip decimal:
  - `"17.29"`
- This can recover all of the floating-point bits
  - Because it originally generated the floating-point bits!
- Worst-case round-trip precision isn't nice
  - `"%.9g"` prints `"17.2900009"`

Quantization Before Rounding

- Apparently nice behavior:
  - `pi, real: 3.14159265358979323846264338…`
  - `   float: 3.1415927410125732421875`
  - `shortest: 3.1415927`
- Potentially surprising behavior:
  - ` e, real: 2.71828182845904523536028 7…`
  - `   float: 2.7182817459106445312 5`
  - `shortest: 2.7182817`
- Floating-point is quantized
  - It can't "remember" what real number you wanted

<charconv> Overview

<charconv> Overview

- `from_chars()` and `to_chars()`
- Integer and floating-point
- Low-level: no whitespace, no locales, few options
- Bounds-checked
- No null termination for input or output
- No dynamic memory allocation
- No exceptions
- Amazing performance

<charconv>, Accidental Final Boss

- We thought it would be a moderate-size feature!
  - Seemed like a few weeks at most
  - Seemed similar to `stof()/stod()/to_string()`
  - It's less than 3 pages of Standardese
- Why was it so much work?
  - Can't use the CRT directly (null termination, other issues)
  - Many corner cases (e.g. parsing 768 digits and a bit more)
  - Different codepaths required for various formats
  - Performing conversions by repeatedly multiplying/dividing by `10` is extremely wrong

<charconv> Timeline

Committed	Shipped	Implemented
2018-01-30	VS 2017 15.7	Integer `from_chars()/to_chars()`
2018-05-24	VS 2017 15.8	Floating-point `from_chars()`
2018-09-01	VS 2017 15.9	Fixed/scientific/general shortest (Ryu)
2018-10-31	VS 2019 16.0	Hex shortest
2018-11-20	VS 2019 16.0	Hex precision
2019-04-30	VS 2019 16.2	Fixed/scientific precision (Ryu Printf)
2019-08-17	VS 2019 16.4	General precision

<charconv> Sizes

- Header-only implementation
  - Intentional design choice, for now
- 606 KB source code (incl. comments, whitespace)
  - 221 KB for actual logic (~5,300 editor lines)
- 121 KB compiled lookup tables
  - Space-time tradeoff
- 4.9 MB tests
  - 58% of the STL's primary test suite

from_chars() Usage

from_chars(), Part 1/3

```
#include <charconv>      // std::from_chars()
#include <system_error> // std::errc
#include <stdio.h>       // Test code
#include <string_view>  // Test code
void test(const std::string_view sv) {
  const char * const first = sv.data();
  const char * const last = first + sv.size();
  double dbl;
  const std::from_chars_result res
    = std::from_chars(first, last, dbl);
```

from_chars(), Part 2/3

```
  printf("Parsed %td chars, ", res.ptr - first);
  if (res.ec == std::errc{}) {
    printf("success: %g\n", dbl);
  } else if (res.ec == std::errc::result_out_of_range) {
    printf("result_out_of_range: %g\n", dbl); // LWG 3081
  } else if (res.ec == std::errc::invalid_argument) {
    printf("invalid_argument\n");
  } else {
    printf("can't happen\n");
  }
}
```

from_chars(), Part 3/3

```
int main() {
  test("3.875");
  test("1e9999");
  test("meow");
}
```

- Output:

```
Parsed 5 chars, success: 3.875
Parsed 6 chars, result_out_of_range: inf
Parsed 0 chars, invalid_argument
```

from_chars() Formats

- `chars_format::general` is the default
  - Accepts both fixed and scientific notation (but not hex)
- `chars_format::scientific` requires exponents
- `chars_format::fixed` doesn't consider exponents
- `chars_format::hex` parses hexfloats without "`0x`"
- This behavior is unlike `strtof()/strtod()`
  - They always accept fixed, scientific, and hex with "`0x`"

to_chars() Usage

to_chars(), Part 1/3

```
#include <charconv>     // std::to_chars()
#include <system_error> // std::errc
#include <stdio.h>      // Test code
#include <type_traits>  // Test code
template <typename T> void test(const T t) {
  static_assert(std::is_floating_point_v<T>);
  constexpr bool IsFloat
    = std::is_same_v<T, float>;
```

to_chars(), Part 2/3

```
// "-1.23456735e-36", "-1.2345678901234567e-100"
constexpr size_t Size = IsFloat ? 15 : 24;
char buf[Size];
const std::to_chars_result res
  = std::to_chars(buf, buf + Size, t);
if (res.ec == std::errc{}) {
  printf("success: %.*s\n",
    static_cast<int>(res.ptr - buf), buf);
```

to_chars(), Part 3/3

```
  } else if (res.ec == std::errc::value_too_large) {
    printf("value_too_large\n");
  } else {
    printf("can't happen\n");
  }
}
int main() {
  test(17.2900009155273437 5f);
  test(1.2339999999999999857891452847979962825775146484375);
}
```

- Output:

```
success: 17.29
success: 1.234
```

to_chars() Formats

- Default: "plain" shortest
  - Selects scientific if shorter, otherwise prefers fixed
- `chars_format` shortest
  - `chars_format::scientific`
  - `chars_format::fixed`
    - "Shortest" means fractional part; whole part can be very long!
  - `chars_format::general`
    - Selects scientific/fixed according to `printf()`'s criterion
  - `chars_format::hex` (no `"0x"`, zero-trims hexits)
- `chars_format fmt, int precision`
  - Like `printf()` (but as always, no null terminators or `"0x"`)

```
plain    | general      ||    plain | general
3.14e-05 | 3.14e-05     ||    7e-05 | 7e-05
0.000314 | 0.000314     ||    7e-04 | 0.0007
0.00314  | 0.00314      ||    0.007 | 0.007
0.0314   | 0.0314       ||    0.07  | 0.07
0.314    | 0.314        ||    0.7   | 0.7
3.14     | 3.14         ||    7     | 7
31.4     | 31.4         ||    70    | 70
314      | 314          ||    700   | 700
3140     | 3140         ||    7000  | 7000
31400    | 31400        ||    70000 | 70000
314000   | 314000       ||    7e+05 | 700000
3140000  | 3.14e+06     ||    7e+06 | 7e+06
31400000 | 3.14e+07     ||
3.14e+08 | 3.14e+08     ||
```

<charconv> Algorithms

Shortest Round-Trip

- Dragon4 (Steele and White, 1990)
  - Slow (uses bignums), complete
  - Improved by Gay (1990), Burger and Dybvig (1996)
- Grisu2 (Loitsch, 2010 – intermediate result)
  - Incorrect output – don't use!
- Grisu3 (Loitsch, 2010)
  - Fast, incomplete (needs fallback algorithm)
- Errol3 (Andrysco, Jhala, and Lerner, 2016)
  - Moderate speed, complete
- Ryu (Ulf Adams, 2018)
  - Fastest, complete

Ryu Techniques

- Ulf Adams' magic is still beyond my understanding
  - Read his paper and code, watch his talk
- Wide multiplications (64x128 for Ryu, 64x192 for Ryu Printf) followed by shifts
  - Multiplying by constants stored in large tables
  - Adams proved that arbitrary precision is not necessary
- Produces integers (e.g. `1729`), writes `"17.29"`
  - Core algorithm is so fast, this step is relatively costly!
- Only integer operations; cold FPU transistors

Performance Tricks

- Fixed shortest: Switch between Ryu, Ryu Printf
  - Lookup table to determine when Ryu rounds large integers
  - Fallbacks: Ryu Printf for `double`, long division for `float`
- General precision: Avoid trial formatting
  - C Standard needs to know scientific exponent X
  - Use a set of lookup tables; reasonably small
  - Stack buffer for final formatting and zero-trimming
- Hex precision: Use the ALU to perform rounding
  - Instead of marching through hexits backwards
  - Rounding away hexits is very weird, but I'll do it fast

<charconv> Performance

Desktop: Intel Core i7-4790 (Haswell Refresh)

Comparisons Are Complicated

- Many interesting dimensions
  - Compiler: MSVC (C1XX/C2), Clang/LLVM
  - Architecture: x86, x64, ARM, ARM64
  - Library: CRT, STL
  - Floating-point type: `float`, `double`
  - Format: plain, scientific, fixed, general, hex
- All times are nanoseconds per floating-point value
  - All speedup ratios are `CRT_time/STL_time`
- Comparing CRT precision 🍎 vs. STL shortest 🍕
  - CRT general precision vs. STL plain shortest
  - CRT fixed precision (lossy) vs. STL fixed shortest (lossless)

strtof()/strtod() vs. from_chars()

MSVC	CRT x86	STL x86	Speedup x86	CRT x64	STL x64	Speedup x64
float scientific	284	168	1.69	161	126	1.28
double scientific	523	381	1.37	318	268	1.19
float hex	127	58	2.20	84	40	2.11
double hex	174	60	2.89	127	44	2.87

sprintf_s() vs. to_chars() Precision

MSVC	CRT x86	STL x86	Speedup x86	CRT x64	STL x64	Speedup x64
float scientific 8	580	134	4.3	366	72	5.1
double scientific 16	1032	171	6.0	610	87	7.0
float fixed 6 (lossy)	603	100	6.0	355	52	6.9
double fixed 6 (lossy)	2745	373	7.4	1381	146	9.5
float general 9	696	154	4.5	427	91	4.7
double general 17	2855	206	13.9	1324	108	12.3
float hex 6	261	30	8.8	188	27	7.0
double hex 13	318	111	2.9	223	34	6.5

sprintf_s() vs. to_chars() Shortest

MSVC	CRT x86	STL x86	Speedup x86	CRT x64	STL x64	Speedup x64
float plain	696	53	13.2	427	46	9.2
double plain	2855	111	25.8	1324	54	24.7
float scientific	580	53	10.9	366	43	8.4
double scientific	1032	111	9.3	610	54	11.4
float fixed	603	67	9.0	355	53	6.6
double fixed	2745	429	6.4	1381	173	8.0
float general	696	55	12.6	427	44	9.6
double general	2855	111	25.6	1324	54	24.4
float hex	261	29	9.0	188	25	7.4
double hex	318	120	2.7	223	32	6.9

Performance Summary

- `<charconv>` is incredibly fast, as promised
  - Includes all bounds-checking costs
- x64 is significantly faster than x86
  - After thoroughly tuning the code for both architectures
  - Wide integer operations make a huge difference
- Performance is nearly uniform across formats
  - Except for fixed notation, due to large integers

Future Improvements

Must Go Faster!

- MSVC's integer `<charconv>` needs to be optimized
- MSVC, LLVM could improve codegen, bugs filed
- SIMD: Great potential, some prototype code
  - Calling all SIMD experts!
- Ryu Printf for `float`: [`ulfjack/ryu#102`](ulfjack/ryu#102)
- Can `from_chars()` avoid bignums?
  - Want more algorithmic breakthroughs

More Info

Links

- MSVC STL: [github.com/microsoft/STL]
  - `stl/inc/charconv`
  - `stl/inc/xcharconv.h`
  - `stl/inc/xcharconv_ryu.h`
  - `stl/inc/xcharconv_ryu_tables.h`
- Ryu: github.com/ulfjack/ryu
  - PLDI 2018 talk: youtu.be/kw-U6smcLzk
- C++20 WP: wg21.link/standard 20.19 [charconv]
- C11: N1570 7.21.6.1 "The `fprintf` function"
- Exploring Binary: exploringbinary.com
- Wikipedia: Single-precision, Double-precision

Questions?

stl@microsoft.com

@StephanTLavavej

Bonus Slides!

Large Integers

- `uint64_t` spends 64 bits on integers
  - Hard limit: all representable, until none
- `double` spends 52 explicit bits on mantissa
  - Soft limit: all representable, then every 2nd, 4th, 8th, etc.
  - (Imagine decimal floating-point: `1234 * 10`^{1}, `1235 * 10`^{1})
- Again, no fuzziness, no garbage digits
  - 2^{123} is `10633823966279326983230456482242756608`

Avoid Rounding Twice

- Converting `string` to `double`, then `double` to `float`, performs rounding twice, with undesirable results
- Instead, convert `string` to `float` directly
  - `from_chars()` is overloaded to do the right thing
- Full example: wg21.link/lwg2403
- (Note: Active bugs in two compiler front-ends; compilers are programs too)
- Widening `float` to `double` is lossless
  - But directly printing `float` is faster; sorry, `printf()`

Hexadecimal Floating-Point

Hexfloats `0x1.94p+6`

- `0x1.f000000000000p+1 // 3.875`
  - `31 * 2`$^{-3}$ `= 0x1F * 2`$^{-3}$ `= 0x1.F * 2`^{1}
- Other examples:
  - `0x1.0000000000000p+20   // 0x1 * 2`^{20} `= 1048576.0`
  - `0x0.0000000000001p-1022 // min subnormal`
  - `0x1.fffffffffffffp+1023 // max normal`
  - `0x1.14a3d70a3d70ap+4    // nearest value to 17.29`
- `0x` prefix: Core and `<stdio.h>` yes, `<charconv>` no

Hexfloats: Human-Readable IEEE

- `0x1.fffffffffffffp+1023`
- The `1` hexit is the implicit bit (`0` for subnormals)
- The `fffffffffffff` hexits are the explicit mantissa
  - `double`: 52 explicit bits, 13 hexits, `"%.13a"`
  - `float`: 23 explicit bits, 6 hexits (last is even), `"%.6a"`
- Can't use `'e'` for "exponent"; `'p'` means "power"
- `1023` (written in decimal) is for a power of 2
  - Despite the mantissa being in base 16

Clang/LLVM Performance

Desktop: Intel Core i7-4790 (Haswell Refresh)

Clang/LLVM 8.0.1

MSVC's STL ♡ Clang/LLVM

- "Header-only" means "you choose the compiler"
- Filed several performance bugs for MSVC and LLVM
- Added workarounds to Ryu and Ryu Printf
- MSVC optimized div/mod better, LLVM improved
  - Used MSVC to generate a 128-bit "magic multiply" constant
- LLVM's overall <charconv> codegen is better
  - Every nanosecond is precious, e.g. 111 ns vs. 85 ns
- Having multiple compilers is good!

strtof()/strtod() vs. from_chars()

Clang/LLVM	CRT x86	STL x86	Speedup x86	CRT x64	STL x64	Speedup x64
float scientific	287	142	2.01	160	118	1.36
double scientific	524	310	1.69	317	240	1.32
float hex	129	50	2.56	82	41	2.01
double hex	176	55	3.20	123	48	2.57

sprintf_s() vs. to_chars() Precision

Clang/LLVM	CRT x86	STL x86	Speedup x86	CRT x64	STL x64	Speedup x64
float scientific 8	588	129	4.6	378	65	5.8
double scientific 16	1051	163	6.5	621	75	8.3
float fixed 6 (lossy)	603	99	6.1	356	47	7.6
double fixed 6 (lossy)	2792	346	8.1	1374	113	12.1
float general 9	701	149	4.7	438	82	5.4
double general 17	2928	199	14.7	1331	97	13.8
float hex 6	271	25	11.0	189	20	9.5
double hex 13	326	30	10.8	224	35	6.5

sprintf_s() vs. to_chars() Shortest

Clang/LLVM	CRT x86	STL x86	Speedup x86	CRT x64	STL x64	Speedup x64
float plain	701	51	13.8	438	38	11.5
double plain	2928	85	34.5	1331	45	29.9
float scientific	588	47	12.6	378	36	10.7
double scientific	1051	87	12.0	621	46	13.5
float fixed	603	58	10.4	356	43	8.2
double fixed	2792	385	7.2	1374	146	9.4
float general	701	48	14.5	438	37	12.0
double general	2928	87	33.7	1331	47	28.3
float hex	271	30	9.1	189	25	7.7
double hex	326	64	5.1	224	31	7.2

glibc vs. MSVC STL Performance

Laptop: Intel Core i7-1065G7 (Ice Lake)

Changing The CRT Baseline

- Is the Windows UCRT unusually slow? No.
- glibc, Ubuntu Bionic (WSL2), GCC x64
  - Used `sprintf()` instead of `sprintf_s()`
- MSVC STL, Windows, Clang/LLVM x64
- glibc's `strtof()/strtod()` are fast for scientific
  - Same speed as `from_chars()`
- `<charconv>` remains significantly faster elsewhere
  - Still ~10x for most scenarios

strtof()/strtod() vs. from_chars()

	glibc x64	STL x64	Speedup x64
float scientific	106	102	1.03
double scientific	205	211	0.97
float hex	89	40	2.23
double hex	176	50	3.53

sprintf() vs. to_chars() Precision

	glibc x64	STL x64	Speedup x64
float scientific 8	269	64	4.2
double scientific 16	509	71	7.2
float fixed 6 (lossy)	331	47	7.1
double fixed 6 (lossy)	2148	101	21.2
float general 9	298	77	3.8
double general 17	604	90	6.7
float hex 6	99	20	4.9
double hex 13	103	32	3.2

sprintf() vs. to_chars() Shortest

	glibc x64	STL x64	Speedup x64
float plain	298	39	7.5
double plain	604	44	13.6
float scientific	269	37	7.3
double scientific	509	45	11.3
float fixed	331	45	7.3
double fixed	2148	121	17.8
float general	298	37	8.0
double general	604	44	13.7
float hex	99	25	3.9
double hex	103	31	3.4