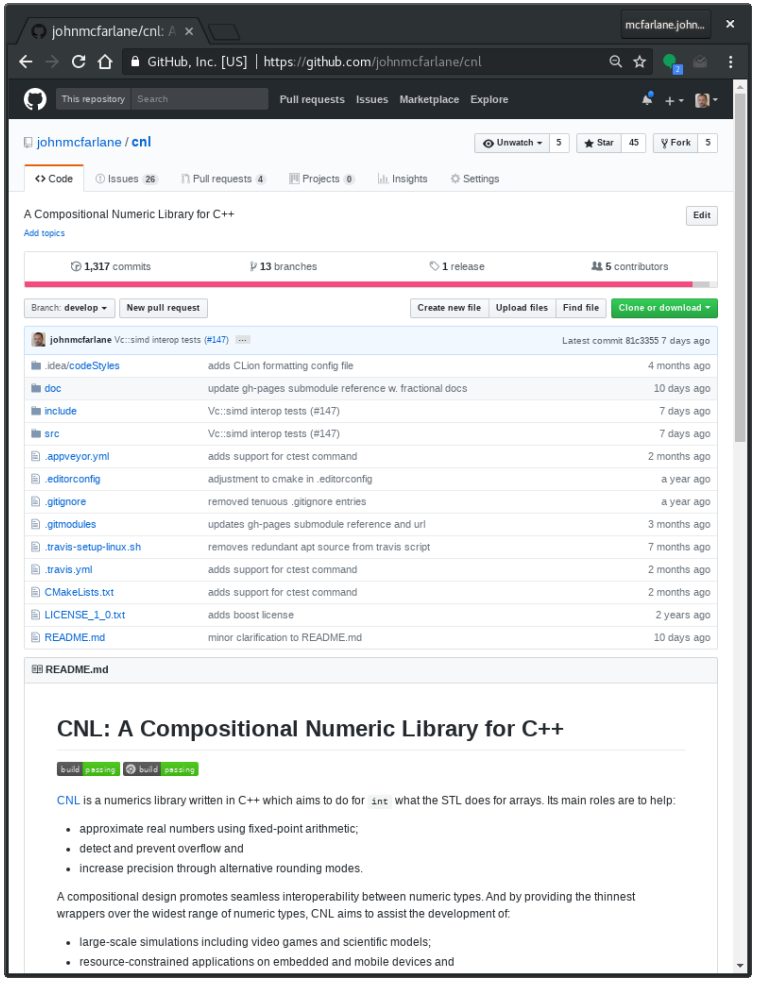
Better Numeric Types in C++

John McFarlane

ACCU Bay Area - 2018-05-09

Background

Background



← github.com/johnmcfarlane/cnl

The Problem with Integers

The Problem with Integers

1. Fixed resolution

The Problem with Integers

1. Fixed resolution
2. Limited range

The Problem with Integers

1. Fixed resolution
2. Limited range
3. 'Interesting' arithmetic behavior

The Problem with Floating-Point

The Problem with Floating-Point

1. Complicated
    - ± 1.significand * 2^{exponent}
    - special values, denormalized values, -0

The Problem with Floating-Point

1. Complicated
    - ± 1.significand * 2^{exponent}
    - special values, denormalized values, -0
2. Occasional weirdness can surprise:
    - determinism, associativity, commutativity and ordering

The Problem with Floating-Point

1. Complicated
    - ± 1.significand * 2 $^{\text{exponent}}$
    - special values, denormalized values, -0
2. Occasional weirdness can surprise:
    - determinism, associativity, commutativity and ordering
3. `<cmath>` functions lack `constexpr`

The Problem with Floating-Point

1. Complicated
    - ± 1.significand * 2 $^{\text{exponent}}$
    - special values, denormalized values, -0
2. Occasional weirdness can surprise:
    - determinism, associativity, commutativity and ordering
3. `<cmath>` functions lack `constexpr`
4. Variable resolution

The Problem with Floating-Point

1. Complicated
    - ± 1.significand * 2 $^{\text{exponent}}$
    - special values, denormalized values, -0
2. Occasional weirdness can surprise:
    - determinism, associativity, commutativity and ordering
3. `<cmath>` functions lack `constexpr`
4. Variable resolution
5. Costly in energy and silicon

Analysis

Analysis

- Floating-point problems are not *so* bad.

Analysis

- Floating-point problems are not *so* bad.
- Integers are a powerful abstraction over registers.

Analysis

- Floating-point problems are not *so* bad.
- Integers are a powerful abstraction over registers.
- But we can do a lot better.

Analysis

- Floating-point problems are not *so* bad.
- Integers are a powerful abstraction over registers.
- But we can do a lot better.
- so …

Goal of CNL

"**Do for `int` what the STL did for `[]`.**"

Goal of CNL

"**Do for `int` what the STL did for `[]`.**"

- Provide zero-cost abstractions over language-level features:

```
std::array<T, N> a;  // T a[N]
std::array<T, N>::iterator i = std::begin(a); // T* i
```

Goal of CNL

"**Do for `int` what the STL did for `[]`.**"

- Provide zero-cost abstractions over language-level features:

```
std::array<T, N> a;  // T a[N]
std::array<T, N>::iterator i = std::begin(a); // T* i
```

- Maintain a familiar interface:

```
auto const& third = a[2];
for (auto const& element : a) { /* ... */ }
```

Goal of CNL

"**Do for `int` what the STL did for `[]`.**"

- Provide zero-cost abstractions over language-level features:

```
std::array<T, N> a;  // T a[N]
std::array<T, N>::iterator i = std::begin(a); // T* i
```

- Maintain a familiar interface:

```
auto const& third = a[2];
for (auto const& element : a) { /* ... */ }
```

- Allow users to opt in to positive-cost functionality:

```
std::array<T, N> a;
auto const& bad_element = a.at(N);  // throws std::out_of_range!
```

Goal of CNL

"**Do for `int` what the STL did for `[]`.**"

- Provide zero-cost abstractions over language-level features:

```
std::array<T, N> a;  // T a[N]
std::array<T, N>::iterator i = std::begin(a); // T* i
```

- Maintain a familiar interface:

```
auto const& third = a[2];
for (auto const& element : a) { /* ... */ }
```

- Allow users to opt in to positive-cost functionality:

```
std::array<T, N> a;
auto const& bad_element = a.at(N);  // throws std::out_of_range!
```

- And most importantly...

Goal of CNL

"**Do for `int` what the STL did for `[]`.**"

- Compose!

```
using fs_cache = unordered_map<filesystem::path, vector<byte>>;
```

Non-Goal

"**Don't do for `int` what STL doesn't do for `[]`.**"

Non-Goal

"**Don't do for `int` what STL doesn't do for `[]`.**"

- Don't make the user pay for what they don't use.

Fixed-Point Arithmetic

Fixed-Point Arithmetic

```
namespace cnl {

    template<typename Rep=int, int Exponent=0, int Radix=2>
    class fixed_point {
        // ...
    private:
        Rep r;
    };

}
```

Fixed-Point Arithmetic

```
namespace cnl {

    template<typename Rep=int, int Exponent=0, int Radix=2>
    class fixed_point {
        // ...
    private:
        Rep r;
    };

}
```

Example usage:

```
#include <cnl/fixed_point.h>

void f() {
    auto n = cnl::fixed_point<int, -8>{ 0.25 };
    std::cout << n * 5; // prints "1.25"
}
```

The Good

The Good

```
// what the programmer writes
bool foo(float f) {
    auto fixed = fixed_point<int, -16>{f};
    auto fixed_plus_one = fixed + 1;
    return fixed_plus_one > fixed;
}
```

The Good

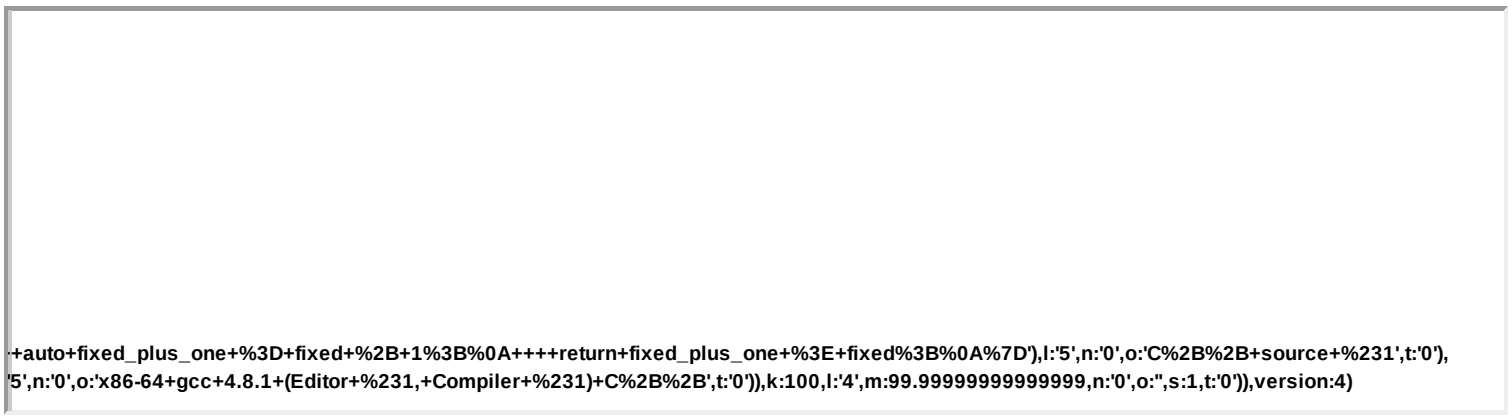
```
// what the programmer writes
bool foo(float f) {
    auto fixed = fixed_point<int, -16>{f};
    auto fixed_plus_one = fixed + 1;
    return fixed_plus_one > fixed;
}
```

```
// what the compiler sees
bool foo(float) {
    return true;
}
```

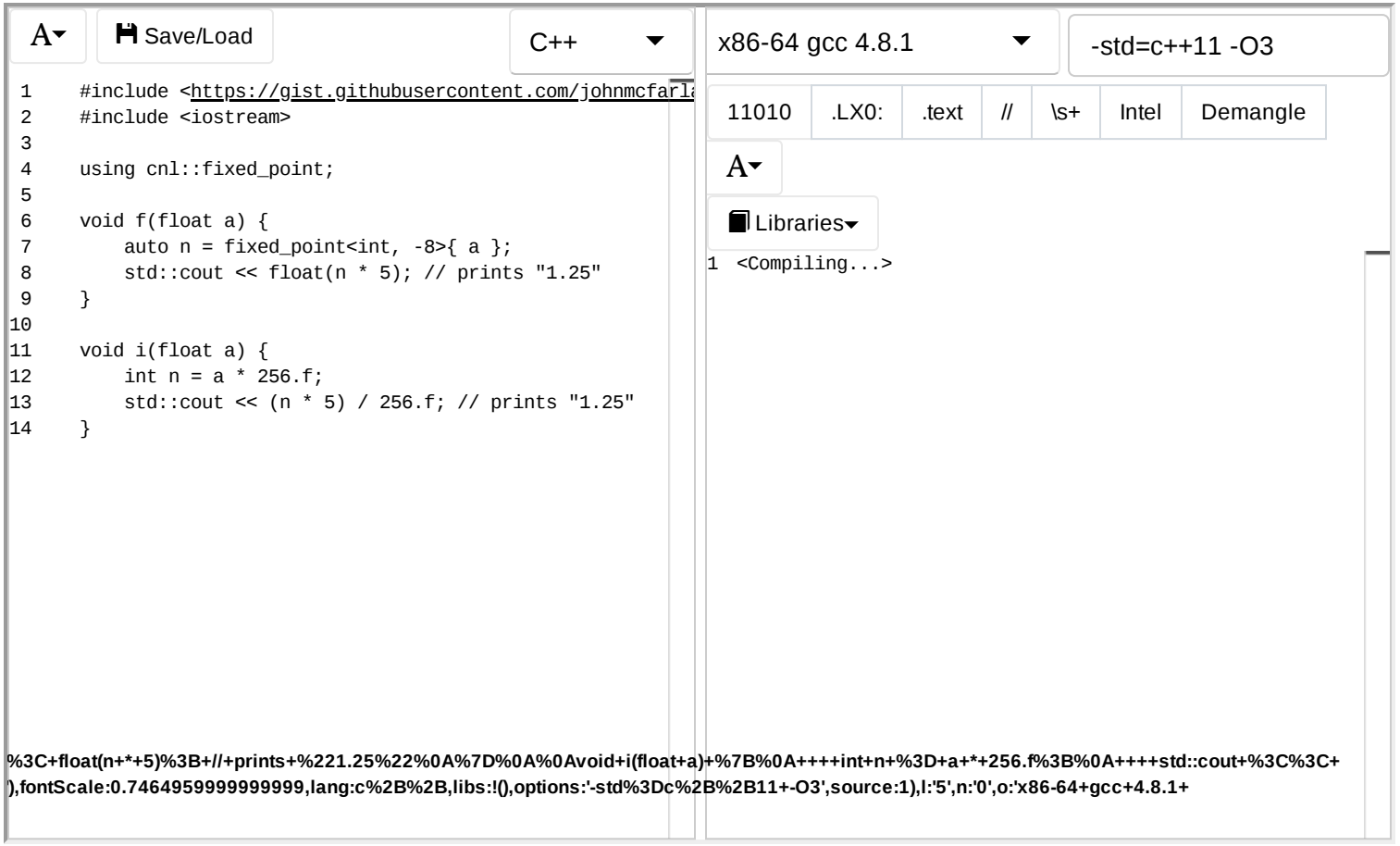
The Good

```
// what the programmer writes
bool foo(float f) {
    auto fixed = fixed_point<int, -16>{f};
    auto fixed_plus_one = fixed + 1;
    return fixed_plus_one > fixed;
}
```

```
// what the compiler sees
bool foo(float) {
    return true;
}
```



The Good


A
Save/Load
C++
x86-64 gcc 4.8.1
-std=c++11 -O3
11010
.LX0:
.text
//
\s+
Intel
Demangle
A
Libraries
1 <Compiling...>
#include <https://gist.githubusercontent.com/johnmcfarl
#include <iostream>
using cnl::fixed_point;
void f(float a) {
auto n = fixed_point<int, -8>{ a };
std::cout << float(n * 5); // prints "1.25"
}
void i(float a) {
int n = a * 256.f;
std::cout << (n * 5) / 256.f; // prints "1.25"
}
%3C+float(n+*+5)%3B+//+prints+%221.25%22%0A%7D%0A%0Avoid+i(float+a)+%7B%0A++++int+n+%3D+a+*+256.f%3B%0A++++std::cout+%3C%3C+
'),fontScale:0.7464959999999999,lang:c%2B%2B,libs:!(),options:'-std%3Dc%2B%2B11+-O3',source:1),l:'5',n:'0',o:'x86-64+gcc+4.8.1+

The Bad

The Bad

```
// range exceeded! (undefined behavior)
auto a = numeric_limits<int>::max() + 1;
```

The Bad

```
// range exceeded! (undefined behavior)
auto a = numeric_limits<int>::max() + 1;
```

```
// also undefined behavior
auto b = numeric_limits<fixed_point<int, -16>>::max() + 1;
```

The Bad

```
// range exceeded! (undefined behavior)
auto a = numeric_limits<int>::max() + 1;
```

```
// also undefined behavior
auto b = numeric_limits<fixed_point<int, -16>>::max() + 1;
```

```
// compiles
static_assert(1 == 1, "this does compile");

// error: static assertion failed: this does not compile
static_assert(1 != 1, "this does not compile");

// error: left shift count >= width of type
static_assert(1 << 1000, "this does not compile");
```

The Bad

```
// range exceeded! (undefined behavior)
auto a = numeric_limits<int>::max() + 1;
```

```
// also undefined behavior
auto b = numeric_limits<fixed_point<int, -16>>::max() + 1;
```

```
// compiles
static_assert(1 == 1, "this does compile");

// error: static assertion failed: this does not compile
static_assert(1 != 1, "this does not compile");

// error: left shift count >= width of type
static_assert(1 << 1000, "this does not compile");
```

```
// compiles
static_assert(numeric_limits<fixed_point<int, -16>>::max() - 1, "this compiles");

// fatal error: static_assert expression is not an integral constant expression
static_assert(numeric_limits<fixed_point<int, -16>>::max() + 1, "this does not!");
```

The Bad

```
// compiles
static_assert(unsigned{1} < signed{-1}, "wat?");
```

The Bad

```
// compiles
static_assert(unsigned{1} < signed{-1}, "wat?");
```

```
// compiles
static_assert(fixed_point<unsigned>{1} < fixed_point<signed>{-1}, "huh?");
```

The Bad

```
// compiles
static_assert(unsigned{1} < signed{-1}, "wat?");
```

```
// compiles
static_assert(fixed_point<unsigned>{1} < fixed_point<signed>{-1}, "huh?");
```

```
// compiles (C++17)
static_assert(fixed_point{1U} < fixed_point{-1});
```

The Ugly

The Ugly

```
// multiplication
auto n = cnl::fixed_point<int, -8>{1.5};
auto nn = n * n;    // type?
```

The Ugly

```
// multiplication
auto n = cnl::fixed_point<int, -8>{1.5};
auto nn = n * n;    // type?
```

```
// (-8) + (-8) = -16
static_assert(std::is_same_v<decltype(nn), cnl::fixed_point<int, -16>>);
```

The Ugly


t%3Cint%3E(f+*+256.f)%3B%0A++++auto+nn+%3D+n+*+n%3B%0A++++return+nn+/+65536.f%3B%0A%7D'),l:'5',n:'0',o:'C%2B%2B+source+%231',t:'0'),

The Ugly

```
constexpr auto n = cnl::fixed_point<int, -8>{1.5};
constexpr auto d = cnl::fixed_point<int, -8>{2.25};
```

The Ugly

```
constexpr auto n = cnl::fixed_point<int, -8>{1.5};
constexpr auto d = cnl::fixed_point<int, -8>{2.25};
```

```
// (-8) - (-8) = 0
constexpr auto q = n / d;    // cnl::fixed_point<int, 0>;
```

The Ugly

```
constexpr auto n = cnl::fixed_point<int, -8>{1.5};
constexpr auto d = cnl::fixed_point<int, -8>{2.25};
```

```
// (-8) - (-8) = 0
constexpr auto q = n / d;    // cnl::fixed_point<int, 0>;
```

```
constexpr auto r = n % d;    // cnl::fixed_point<int, -8>;
```

The Ugly

```
constexpr auto n = cnl::fixed_point<int, -8>{1.5};
constexpr auto d = cnl::fixed_point<int, -8>{2.25};
```

```
// (-8) - (-8) = 0
constexpr auto q = n / d;    // cnl::fixed_point<int, 0>;
```

```
constexpr auto r = n % d;    // cnl::fixed_point<int, -8>;
```

```
// (-8) - (23)
constexpr auto q = cnl::divide(n, d);    // cnl::fixed_point<long, -31>;
```

How Do You Solve a Problem Like Division?

How Do You Solve a Problem Like Division?

```
5.5 * 5.5 = 30.25
55. * .55 = 30.25
5.5 * 55. = 302.5
```

How Do You Solve a Problem Like Division?

```
5.5 * 5.5 = 30.25
55. * .55 = 30.25
5.5 * 55. = 302.5
```

```
AAA.BBBBB * CCCCCC.DD = AAACCCCCC.BBBBBDD
```

How Do You Solve a Problem Like Division?

```
5.5 * 5.5 = 30.25
55. * .55 = 30.25
5.5 * 55. = 302.5
```

```
AAA.BBBBB * CCCCCC.DD = AAACCCCCC.BBBBBDD
```

```
10 / 100 = 00.10
1 / 1000 = 0.001
```

How Do You Solve a Problem Like Division?

```
5.5 * 5.5 = 30.25
55. * .55 = 30.25
5.5 * 55. = 302.5
```

```
AAA.BBBBB * CCCCCC.DD = AAACCCCCC.BBBBBDD
```

```
10 / 100 = 00.10
1 / 1000 = 0.001
```

```
AAA.BBBBB / CCCCCC.DD = AAADD.BBBBBCCCCCC
```

How Do You Solve a Problem Like Division?

```
5.5 * 5.5 = 30.25
55. * .55 = 30.25
5.5 * 55. = 302.5
```

```
AAA.BBBBB * CCCCCC.DD = AAACCCCCC.BBBBBDD
```

```
10 / 100 = 00.10
1 / 1000 = 0.001
```

```
AAA.BBBBB / CCCCCC.DD = AAADD.BBBBBCCCCCC
```

```
1 / 3 = 0.333333333...
```

How Do You Solve a Problem Like Division?

How Do You Solve a Problem Like Division?

```
template<typename Numerator, typename Denominator>
struct fractional {
    Numerator numerator;
    Denominator denominator;
};
```

How Do You Solve a Problem Like Division?

```
template<typename Numerator, typename Denominator>
struct fractional {
    Numerator numerator;
    Denominator denominator;
};
```

```
constexpr auto n = fixed_point<int16_t, -8>{1.5};
constexpr auto d = fixed_point<int16_t, -8>{2.25};
constexpr auto f = fractional{n, d};
constexpr auto q = fixed_point{f};  // fixed_point<int32_t, -15>{.66666667}
```

Elasticity

Elasticity

```
auto n = fixed_point<uint8_t, -8>{0.99609375};
auto nn = n * n;
```

Elasticity

```
auto n = fixed_point<uint8_t, -8>{0.99609375};
auto nn = n * n;    // fixed_point<int, -16>{0.9922027587890625};
```

Elasticity

```
auto n = fixed_point<uint8_t, -8>{0.99609375};
auto nn = n * n;    // fixed_point<int, -16>{0.9922027587890625};
```

```
auto n = fixed_point<int, -31>{0.99609375};
auto nn = n * n;    // fixed_point<int, -62>{?!?!?!?!?!?!};
```

Elasticity

```
template<int Digits, class Narrowest = int>
class elastic_integer { WideEnoughInteger r; /* other stuff */ };
```

Elasticity

```
template<int Digits, class Narrowest = int>
class elastic_integer { WideEnoughInteger r; /* other stuff */ };
```

```
#include <cnl/elastic_integer.h>
using cnl::elastic_integer;
auto e = elastic_integer<31>{0x7FFFFFFF};   // r has 31 or more digits
```

Elasticity

```
template<int Digits, class Narrowest = int>
class elastic_integer { WideEnoughInteger r; /* other stuff */ };
```

```
#include <cnl/elastic_integer.h>
using cnl::elastic_integer;
auto e = elastic_integer<31>{0x7FFFFFFF};   // r has 31 or more digits
```

```
auto ee = e * e;   // elastic_integer<62>{INT64_C(0x3FFFFFFF00000001)}
```

Elasticity

```
template<int Digits, class Narrowest = int>
class elastic_integer { WideEnoughInteger r; /* other stuff */ };
```

```
#include <cnl/elastic_integer.h>
using cnl::elastic_integer;
auto e = elastic_integer<31>{0x7FFFFFFF};   // r has 31 or more digits
```

```
auto ee = e * e;   // elastic_integer<62>{INT64_C(0x3FFFFFFF00000001)}
```

```
auto _2ee = ee + ee;   // elastic_integer<63>{INT64_C(0x7FFFFFFE00000002)}
```

Elasticity

```
template<int Digits, class Narrowest = int>
class elastic_integer { WideEnoughInteger r; /* other stuff */ };
```

```
#include <cnl/elastic_integer.h>
using cnl::elastic_integer;
auto e = elastic_integer<31>{0x7FFFFFFF};   // r has 31 or more digits
```

```
auto ee = e * e;   // elastic_integer<62>{INT64_C(0x3FFFFFFF00000001)}
```

```
auto _2ee = ee + ee;   // elastic_integer<63>{INT64_C(0x7FFFFFFE00000002)}
```

```
auto fpe = fixed_point<elastic_integer<31>, -31>{0.99609375};
```

Elasticity

```
template<int Digits, class Narrowest = int>
class elastic_integer { WideEnoughInteger r; /* other stuff */ };
```

```
#include <cnl/elastic_integer.h>
using cnl::elastic_integer;
auto e = elastic_integer<31>{0x7FFFFFFF};   // r has 31 or more digits
```

```
auto ee = e * e;   // elastic_integer<62>{INT64_C(0x3FFFFFFF00000001)}
```

```
auto _2ee = ee + ee;   // elastic_integer<63>{INT64_C(0x7FFFFFFE00000002)}
```

```
auto fpe = fixed_point<elastic_integer<31>, -31>{0.99609375};
```

```
auto sq = fpe * fpe;  // fixed_point<elastic_integer<62>, -62>{0.9922027587890625}
```

Elasticity

```
template<int Digits, class Narrowest = int>
class elastic_integer { WideEnoughInteger r; /* other stuff */ };
```

```
#include <cnl/elastic_integer.h>
using cnl::elastic_integer;
auto e = elastic_integer<31>{0x7FFFFFFF};   // r has 31 or more digits
```

```
auto ee = e * e;   // elastic_integer<62>{INT64_C(0x3FFFFFFF00000001)}
```

```
auto _2ee = ee + ee;   // elastic_integer<63>{INT64_C(0x7FFFFFFE00000002)}
```

```
auto fpe = fixed_point<elastic_integer<31>, -31>{0.99609375};
```

```
auto sq = fpe * fpe;  // fixed_point<elastic_integer<62>, -62>{0.9922027587890625}
```

```
auto q = fixed_point{fractional{sq, sq}};
// fixed_point<elastic_integer<124>, -62>{1}
```

Run-time Safety

Run-time Safety

```
#include <cnl/overflow_integer.h>
using cnl::overflow_integer;
auto i = overflow_integer<uint8_t>{255};
```

Run-time Safety

```
#include <cnl/overflow_integer.h>
using cnl::overflow_integer;
auto i = overflow_integer<uint8_t>{255};
```

```
auto j = i + 1;
```

Run-time Safety

```
#include <cnl/overflow_integer.h>
using cnl::overflow_integer;
auto i = overflow_integer<uint8_t>{255};
```

```
auto j = i + 1; // overflow_integer<int>{256}
```

Run-time Safety

```
#include <cnl/overflow_integer.h>
using cnl::overflow_integer;
auto i = overflow_integer<uint8_t>{255};
```

```
auto j = i + 1; // overflow_integer<int>{256}
```

```
overflow_integer<uint8_t> k = i + 1;  // throw std::overflow_error
```

Run-time Safety

```
#include <cnl/overflow_integer.h>
using cnl::overflow_integer;
auto i = overflow_integer<uint8_t>{255};
```

```
auto j = i + 1; // overflow_integer<int>{256}
```

```
overflow_integer<uint8_t> k = i + 1;  // throw std::overflow_error
```

```
constexpr overflow_integer<uint8_t> k = i + 1;
static_assert(cnl::_impl::identical(overflow_integer<int>{256}, k));
```

```
[ 29%] Building CXX object src/test/CMakeFiles/fp_test.dir/cppcon2017.cpp.o
/home/john/cnl/src/test/cppcon2017.cpp:151:37: fatal error: constexpr variable 'k' must be i
    constexpr overflow_integer<uint8_t> k = i + 1;
                                        ^~~~~~~~
/home/john/cnl/include/cnl/overflow.h:52:40: note: subexpression not valid in a constant exp
            return condition ? value : throw std::overflow_error("");
                                       ^
```

and so on...

Deduction and UDLs

Deduction and UDLs

```
auto x = fixed_point{42UL}; // fixed_point<unsigned long, 0>{42}
```

Deduction and UDLs

```
auto x = fixed_point{42UL}; // fixed_point<unsigned long, 0>{42}
```

```
auto y = fixed_point{128};  // fixed_point<int, 0>{128}
```

Deduction and UDLs

```
auto x = fixed_point{42UL}; // fixed_point<unsigned long, 0>{42}
```

```
auto y = fixed_point{128};  // fixed_point<int, 0>{128}
```

```
using cnl::literals;
auto z = fixed_point{128_c};  // fixed_point<int, 7>{128}
```

Deduction and UDLs

```
auto x = fixed_point{42UL}; // fixed_point<unsigned long, 0>{42}
```

```
auto y = fixed_point{128};  // fixed_point<int, 0>{128}
```

```
using cnl::literals;
auto z = fixed_point{128_c};  // fixed_point<int, 7>{128}
```

```
auto a = fixed_point{0b10000000000000000000000000000000000000000_c};
// a === fixed_point<int, 40>{0b10000000000000000000000000000000000000000L}
```

Deduction and UDLs

```
auto x = fixed_point{42UL}; // fixed_point<unsigned long, 0>{42}
```

```
auto y = fixed_point{128};  // fixed_point<int, 0>{128}
```

```
using cnl::literals;
auto z = fixed_point{128_c};  // fixed_point<int, 7>{128}
```

```
auto a = fixed_point{0b10000000000000000000000000000000000000_c};
// a === fixed_point<int, 40>{0b10000000000000000000000000000000000000L}
```

```
auto b = fixed_point{0b11111111111111111111111111111111111111_c};
// b === fixed_point<long, 0>{0b11111111111111111111111111111111111111L}
```

Deduction and UDLs

Deduction and UDLs

```
auto c = elastic_integer{2018_c}; // elastic_integer<11>{2018}
```

Deduction and UDLs

```
auto c = elastic_integer{2018_c}; // elastic_integer<11>{2018}
```

```
auto e = 0x7f000_elastic; // fixed_point<elastic_integer<7>, 12>{0x7f000}
```

Deduction and UDLs

```
auto c = elastic_integer{2018_c}; // elastic_integer<11>{2018}
```

```
auto e = 0x7f000_elastic; // fixed_point<elastic_integer<7>, 12>{0x7f000}
```

```
auto s = e >> 1_c;  // fixed_point<elastic_integer<7>, 11>{0x3f800}
```

Interoperability - Boost.Multiprecision

```
#include <cnl/auxiliary/boost.multiprecision.h>
using namespace boost::multiprecision;

template<int NumBits, int Exponent = 0>
using mp_fixed_point = cnl::fixed_point<
        number<cpp_int_backend<NumBits, NumBits, signed_magnitude, unchecked, void
        Exponent>;
```

Interoperability - Boost.Multiprecision

```
#include <cnl/auxiliary/boost.multiprecision.h>
using namespace boost::multiprecision;

template<int NumBits, int Exponent = 0>
using mp_fixed_point = cnl::fixed_point<
        number<cpp_int_backend<NumBits, NumBits, signed_magnitude, unchecked, void
        Exponent>;
```

Fixed-Point + Boost.Multiprecision:

Interoperability - Boost.Multiprecision

```
#include <cnl/auxiliary/boost.multiprecision.h>
using namespace boost::multiprecision;

template<int NumBits, int Exponent = 0>
using mp_fixed_point = cnl::fixed_point<
        number<cpp_int_backend<NumBits, NumBits, signed_magnitude, unchecked, void
        Exponent>;
```

Fixed-Point + Boost.Multiprecision:

- googol (10^{100}) ✓

Interoperability - Boost.Multiprecision

```
#include <cnl/auxiliary/boost.multiprecision.h>
using namespace boost::multiprecision;

template<int NumBits, int Exponent = 0>
using mp_fixed_point = cnl::fixed_point<
        number<cpp_int_backend<NumBits, NumBits, signed_magnitude, unchecked, void
        Exponent>;
```

Fixed-Point + Boost.Multiprecision:

- googol (10^{100}) ✓
- googolth (1 / googol) ✓

Interoperability - Boost.Multiprecision

```
#include <cnl/auxiliary/boost.multiprecision.h>
using namespace boost::multiprecision;

template<int NumBits, int Exponent = 0>
using mp_fixed_point = cnl::fixed_point<
        number<cpp_int_backend<NumBits, NumBits, signed_magnitude, unchecked, void
        Exponent>;
```

Fixed-Point + Boost.Multiprecision:

- googol (10^{100}) ✓
- googolth (1 / googol) ✓
- googolplex (10^{googol}) ✗

Interoperability - Boost.SIMD

```
#include <cnl/auxiliary/boost.simd.h>
using boost::simd::pack;

template<class T, std::size_t N, int Exponent>
using fixed_point_pack = fixed_point<pack<T, N>, Exponent>;
```

Interoperability - Boost.SIMD

```cpp
#include <cnl/auxiliary/boost.simd.h>
using boost::simd::pack;

template<class T, std::size_t N, int Exponent>
using fixed_point_pack = fixed_point<pack<T, N>, Exponent>;
```

```cpp
using fpp = fixed_point_pack<int, 4, -16>;
using initializer = initializer<fpp>;

auto expected = fpp{initializer{7.9375+-1, -8.+.125, 0+-5, 3.5+-3.5}};
auto augend = fpp{initializer{7.9375, -8., 0, 3.5}};
auto addend = fpp{initializer{-1, .125, -5, -3.5}};
auto sum = augend + addend;
```

Interoperability - Boost.SIMD

CNL Today and Tomorrow

CNL Today and Tomorrow

- Arbitrary width

CNL Today and Tomorrow

- Arbitrary width
- Full Support for Rounding and Overflow

```
template<class Rep = int, class RoundingTag = closest_rounding_tag>
class precise_integer;
```

CNL Today and Tomorrow

- Arbitrary width
- Full Support for Rounding and Overflow

```
template<class Rep = int, class RoundingTag = closest_rounding_tag>
class precise_integer;
```

- Full complement of operators for `overflow_integer` and `precise_integer`

CNL Today and Tomorrow

- Arbitrary width
- Full Support for Rounding and Overflow

```
template<class Rep = int, class RoundingTag = closest_rounding_tag>
class precise_integer;
```

- Full complement of operators for `overflow_integer` and `precise_integer`
- Full complement of free functions

```
add(saturated_overflow, UINT32_C(0xFFFFFFFF), UINT32_C(0x12345678))
divide(closest_rounding_tag, 2, 3);
```

CNL Today and Tomorrow

- Arbitrary width
- Full Support for Rounding and Overflow

```
template<class Rep = int, class RoundingTag = closest_rounding_tag>
class precise_integer;
```

- Full complement of operators for `overflow_integer` and `precise_integer`
- Full complement of free functions

```
add(saturated_overflow, UINT32_C(0xFFFFFFFF), UINT32_C(0x12345678))
divide(closest_rounding_tag, 2, 3);
```

- Better Literals

```
auto a = 0b1111.1111_elastic;  // fixed_point<elastic_integer<8>, -4>
```

Thank You!

Now go to GitHub and try it out! github.com/johnmcfarlane/cnl

```
git clone https://github.com/johnmcfarlane/cnl.git
```

Bonus Slides - Beyond Exponents

`std::ratio` scales things statically:

```
template<int Num, int Denom = 1> class ratio;
```

Bonus Slides - Beyond Exponents

`std::ratio` scales things statically:

```
template<int Num, int Denom = 1> class ratio;
```

What if `Exponent` was replaced with a type parameter?

```
// equivalent to fixed_point<int, -8, 2>
using a = fixed_point<int, ratio<1, 256>>;
```

Bonus Slides - Beyond Exponents

`std::ratio` scales things statically:

```
template<int Num, int Denom = 1> class ratio;
```

What if `Exponent` was replaced with a type parameter?

```
// equivalent to fixed_point<int, -8, 2>
using a = fixed_point<int, ratio<1, 256>>;
```

This should be possible: simply separate `fixed_point`'s two concerns:

```
// a type which stores an integer and scales it
template<typename Rep, typename Scale>
class scaled_integer;

// a type which scales integers
template<int Exponent, int Radix> class power;
```

Bonus Slides - Beyond Exponents

Now `fixed_point` is just one of many `scaled_integer` types.

```
template<typename Rep, int Exponent, int Radix>
using fixed_point = scaled_integer<Rep, power<Exponent, Radix>>

// (Decimal fixed-point is already on the to-do list.)
template<typename Rep, int Exponent>
using decimal_fixed_point = scaled_integer<Rep, power<Exponent, 10>>
```

Bonus Slides - Beyond Exponents

Now `fixed_point` is just one of many `scaled_integer` types.

```
template<typename Rep, int Exponent, int Radix>
using fixed_point = scaled_integer<Rep, power<Exponent, Radix>>

// (Decimal fixed-point is already on the to-do list.)
template<typename Rep, int Exponent>
using decimal_fixed_point = scaled_integer<Rep, power<Exponent, 10>>
```

And a souped-up `std::ratio` is another way to scale integers.

```
template<typename Rep, int Power>
using dollar = scaled_integer<Rep, ratio<1, 100>>;

template<typename Rep, int Power>
using angle = scaled_integer<Rep, ratio<1, 360>>;

// units in the range [0, 1]
template<typename Rep>
using unit = scaled_integer<Rep, ratio<1, std::numeric_limits<Rep>::max()>>;
```

Bonus Slides - Beyond Exponents

And scaling of integers is just the beginning...

Bonus Slides - Beyond Exponents

And scaling of integers is just the beginning...

```
// type-safety prevents units from being confused
struct length_tag {};
struct time_tag {};

// individual quantities
template<typename BaseTag, int Power>
struct base_quantity;

// is quantity a scale, like exponent or std::ratio?
template<typename ... BaseQuantities>
struct quantity;

using time = quantity<base_quantity<time_tag, 1>>;
using length = quantity<base_quantity<length_tag, 1>>;

template<typename ... BaseQuantitiesA, typename ... BaseQuantitiesB>
auto operator/(quantity<BaseQuantitiesA...>, quantity<BaseQuantitiesB...>)

// quantity<base_quantity<length_tag, 1>, base_quantity<time_tag, -1>>
auto speed = length{} / time{};
```

Disclaimer: none of this compiles ... yet!

Thank You! - @JSAMcFarlane

Now go to GitHub and try it out!

```
git clone https://github.com/johnmcfarlane/cnl.git
```

slides: github.com/johnmcfarlane/presentations/tree/master/2018-05-09_ACCU_Bay_Area