Fixed-point Arithmetic

John McFarlane

Silicon Valley Association of C/C++ Users
2016-06-08

About Me

Studies

- AI / ALife Studies (Middlesex, Sussex)

Career

- Game Industry Experience (Creative Assembly, Computer Artworks, Lionhead, Surreal, Snowblind, Z2)
- Internet Startup (Cookbrite)
- Autonomous Vehicles (Zoox Inc.)

SG6/SG14 Proposals

- P0037R2 - Fixed-Point Real Numbers
- P0381R0 - Numeric Width

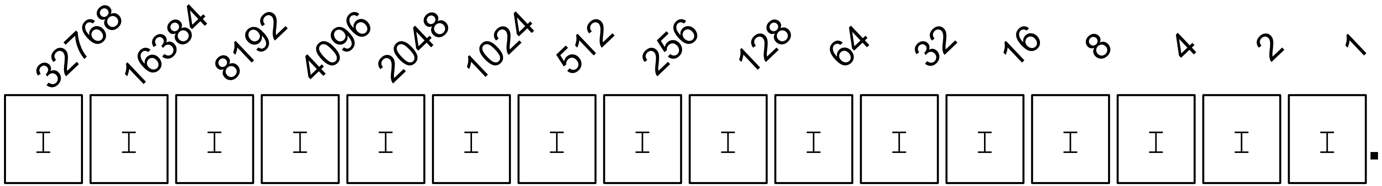
Contents

What is Fixed-Point Arithmetic?

- Floating-point without the ‘float’; exponent is determined ahead of time
- Primarily a method for representing real numbers using integers
- Popular before FPUs and on embedded systems with limited transistor counts
- Sometimes have dedicated instructions on DSPs

Anatomy of a Fixed-Point Number

uint16_t = Unsigned, 16 Integer Digits



Anatomy of a Fixed-Point Number

u8:8 = Unsigned, 8 Integer Digits, 8 Fractional Digits

128	64	32	16	8	4	2	1		1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
I	I	I	I	I	I	I	I	.	F	F	F	F	F	F	F	F

$$2^a-2^{-b}$$

256 - 1/256 = 255.99609375

= 65535 / 256

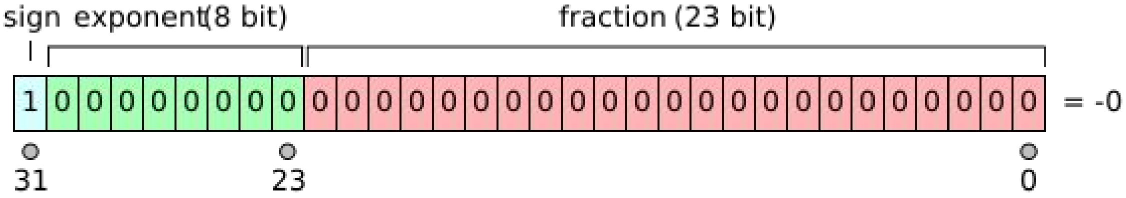
Why Not Just Use Floating Point?

Likely answer is “**you probably should**”.

1. Versatility
2. Ease of use
3. Good support of IEEE 754 standard

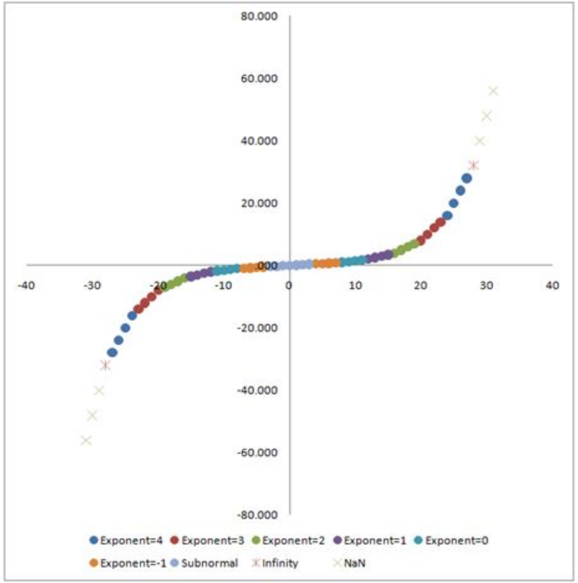
Why Use Fixed-point?

1. Predictability / determinism
2. All bits devoted to mantissa

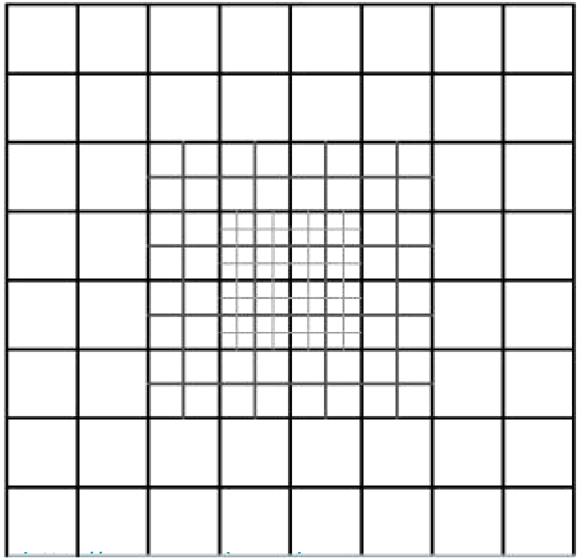


3. 8-bit and 16-bit width
4. Even distribution

Floating Point Distribution



https://blogs.msdn.microsoft.com/dwayneneed/2010/05/06/fun-with-floating-point/



http://www.pathengine.com/Contents/Overview/FundamentalConcepts/WhyIntegerCoordinates/page.php

http://minecraft.gamepedia.com/File:Far_Lands_Cartograph.png

Contents

fixed_point.h (version 0)

```
#include <cinttypes>

using u8_8 = std::uint16_t;

constexpr u8_8 float_to_fixed(float f)
{
   return f*256;
}

constexpr float fixed_to_float(u8_8 i)
{
   return i/256.f;
}

constexpr u8_8 add(u8_8 a, u8_8 b)
{
   return a+b;
}

constexpr u8_8 multiply(u8_8 a, u8_8 b)
{
   return (uint32_t(a)*uint32_t(b))/256;
}
```

test_fixed_point.cpp

```cpp
#include <sg14/fixed_point.h>

constexpr auto float_a{3.75f};
constexpr auto float_b{17.125f};

constexpr auto fixed_a = float_to_fixed(float_a);
constexpr auto fixed_b = float_to_fixed(float_b);

static_assert(fixed_to_float(fixed_a) == float_a, "");
static_assert(fixed_to_float(fixed_b) == float_b, "");

// test: add
constexpr auto fixed_sum = add(fixed_a, fixed_b);
constexpr auto float_sum = fixed_to_float(fixed_sum);
static_assert(float_sum==float_a+float_b, "");

// test: multiply
constexpr auto fixed_product = multiply(fixed_a, fixed_b);
constexpr auto float_product = fixed_to_float(fixed_product);
static_assert(float_product==float_a*float_b, "");
```

Criticisms?

- Type Safety - float and fixed values have different meanings
- Generality - only u8.8 supported
- Usability - arithmetic operators might be nice
- Overflow Safety - `255 * 255 = ?`
- Fidelity - rounding tends towards zero or negative infinity
- Predictability - types keep changing to `int` under our noses
- Portability - because `int` isn’t a known size, behavior may vary

Criticisms (that cannot also be levelled at integers)?

- Type Safety - float and fixed values have different meanings
- Generality - only u8.8 supported
- Usability - arithmetic operators might be nice
- ~~Overflow Safety - `255 * 255 = ?`~~
- ~~Fidelity - rounding tends towards zero or negative infinity~~
- ~~Predictability - types keep changing to `int` under our noses~~
- ~~Portability - because `int` isn't a known size, behavior may vary~~

sg14::fixed_point<> Class Template

Definition:

```
namespace sg14 {
  template<class Rep = int, int Exponent = 0>
  class fixed_point;
}
```

Usage:

```
#include <sg14/fixed_point.h>
using u8_8 = sg14::fixed_point<uint16_t, -8>;
```

Declaration

```
// x is represented by an int and scaled down by 1 bit
auto x = fixed_point<int, -1>{3.5};

// another way to specify a fixed-point type is with make_fixed or make_ufixed
auto y = make_fixed<30, 1>{3.5};  // (s30:1)
static_assert(is_same<decltype(x), decltype(y)>::value, "");  // assumes that int is 32-bit

// under the hood, x stores a whole number
cout << x.data() << endl;  // "7"

// but it multiplies that whole number by 2^-1 to produce a real number
cout << x << endl;  // "3.5"

// like an int, x has limited precision
cout << x/2 << endl;  // "1.5"
```

Arithmetic Operations

Addition: Incremented Capacity

	u2:6	I I . F F F F F F
+	u5:3	I I I I I . F F F
=	u6:6	I I I I I F . F F F F F F

Multiplication: Combined Capacity

	u2:6	I I . F F F F F F
*	u5:3	I I I I I . F F F
=	u7:9	I I I I I I F . F F F F F F F F F

Arithmetic Operators

```
// define a constant signed value with 3 integer and 28 fractional bits (s3:28)
constexpr auto pi = fixed_point<int32_t, -28>{3.1415926535};

// expressions involving integers return fixed_point results
constexpr auto tau = pi*2;
static_assert(is_same<decltype(tau), decltype(pi)>::value, "");

// "6.28319"
cout << tau << endl;

// expressions involving floating-point values return floating-point results
constexpr auto degrees = tau*(180/3.1415926534);
static_assert(is_same<decltype(degrees), const double>::value, "");

// "360"
cout << degrees << '\n';
```

Arithmetic Operators - The ‘Multiply Problem’

What should `decltype(fixed_point<R, E>()*fixed_point<R, E>())` be?

- Truncate:
  - drop lower bits
    - Good for `make_fixed<0, N>`
    - Bad for `make_fixed<N, 0>`
  - drop higher bits
    - Bad for `make_fixed<0, N>`
    - Good for `make_fixed<N, 0>`
  - match operands:
    - `fixed_point<decltype(R()*R()), E>::value`
- Widen:
  - Powerful - greatly reduced risk of overflow
  - Astonishing - novel types created frequently
  - Complicated - bits must be counted, compile time suffers
  - Limited - assignment to pre-ordained type truncates

Arithmetic Functions

```
// this variable uses all of its capacity
auto x = fixed_point<uint8_t, -4>{15.9375};

// 15.9375 * 15.9375 = 254.00390625 ... overflow!
cout << fixed_point<uint8_t, -4>{x*x} << endl;  // "14" instead!

// by default, fixed-point follows similar promotion rules to native types
auto xx = x*x;

// x * x has type fixed_point<int, -4>
static_assert(is_same<decltype(xx), fixed_point<int, -4>>::value, "");
cout << x*x << endl;  // "254" - better but not perfect

// for full control, use named functions:
cout << setprecision(12)
       << multiply<fixed_point<uint16_t, -8>>(x, x) << endl;  // 254.00390625
```

Archetypes, Families and `set_width`

The two native **families** are the signed and unsigned integers. **Fast archetypes** are `signed` and `unsigned`. **Least archetypes** are `signed char` and `unsigned char`.

A helper type for choosing a member of a family based on width:

```
template<class Archetype, size_t MinNumBits>
struct set_width;
```

For example, to specify an unsigned, 16-bit native type:

```
using u16 = typename set_width<signed, 16>::type;
```

Archetypes in Action

```
template<int IntegerDigits, int FractionalDigits = 0, class Archetype = signed>
using make_fixed = fixed_point<
       set_width_t<Archetype, IntegerDigits+FractionalDigits+is_signed<Archetype>::value>,
       -FractionalDigits>;

template<int IntegerDigits, int FractionalDigits = 0, class Archetype = unsigned>
using make_ufixed = make_fixed<
       IntegerDigits,
       FractionalDigits,
       typename make_unsigned<Archetype>::type>;
```

Composition

```
// define an unsigned type with 400 integer digits and 400 fractional digits
// and use boost::multiprecision::uint128_t as the archetype for the Rep type
using big_number = make_ufixed<400, 400, boost::multiprecision::uint128_t>;
static_assert(big_number::digits==800, "");

// a googol is 10^100
auto googol = big_number{1};
for (auto zeros = 0; zeros!=100; ++zeros) {
   googol *= 10;
}

// "1e+100"
cout << googol << endl;

// "1e-100" although this calculation is only approximate
cout << big_number{1}/googol << endl;
```

Contents

Run-Time Overflow Detection

Integration with P0228R0 coming soon!

```
#include <boost/safe_numeric/safe_integer.hpp>

safe<int> f(safe<int> x, safe<int> y){
  return x + y; // throw exception if correct result cannot be returned
}
```

https://github.com/robertramey/safe_numerics

Elastication™

```
// this variable has 4 integer and 4 fractional digits
auto x = elastic<4, 4, unsigned>{15.9375};
cout << x << endl;  // "15.9375"

// unlike fixed_point, operations on elastic types often produce bigger types
auto xx = x*x;
static_assert(is_same<decltype(xx), elastic<8, 8, unsigned>>::value, "");
cout << xx << endl;  // "254.00390625"

// the 'archetype' of x is unsigned which means it uses machine-efficient types
static_assert(sizeof(x) == sizeof(unsigned), "");

// if storage is the main concern, a different archetype can be used
auto compact_x = elastic<4, 4, uint8_t>(x);
static_assert(sizeof(compact_x) == sizeof(uint8_t), "");
cout << compact_x << endl;  // "15.9375"

// but don't worry: it's a lower limit and storage still increases as required
auto compact_xx = elastic<8, 8, uint8_t>(xx);
static_assert(sizeof(compact_xx) == sizeof(uint16_t), "");
cout << compact_xx << endl;     // "254.00390625"
```

Decimalization

```
template<int Radix, class Rep = int, int Exponent = 0>
class basic_fixed_point;

template<class Rep = int, int Exponent = 0>
using fixed_point = basic_fixed_point<2, Rep, Exponent>;

template<class Rep = int, int Exponent = 0>
using decimal_fixed_point = basic_fixed_point<10, Rep, Exponent>;

template <typename Rep> using btc = decimal_fixed_point<Rep, -8>;  // bitcoin
template <typename Rep> using eur = decimal_fixed_point<Rep, -2>;  // euro
template <typename Rep> using jpy = decimal_fixed_point<Rep, 0>;   // yen
template <typename Rep> using kwd = decimal_fixed_point<Rep, -3>;  // Kiwaiti dinar

using usd_cent_hundredths = decimal_fixed_point<long long, -4>;

using gbp = make_decimal_fixed_point<6, 2>;  // all UK prices under £1M
```

Contents

Fixed-point, Floating Point and Integral

Fixed-point is ~~an alternative to floats~~ a superset of integers:

- Truncating lower bits is flawed.
- Integers already truncate upper bits.
- Behavior is least astonishing when:

```
fixed_point<Integer, 0> === Integer
```

An integer is a fixed-point type with `Exponent=0`.

Fixed-point Means Two Different Things

1. Approximation of a real number using integers:
   - extends integers the way `vector` and `array` extend arrays
2. A numeric type that has:
   - run-time error handling (esp. overflow);
   - compile-time error handling through unlimited widening (elastication);
   - a choice of rounding modes;
   - etc..

Generic Solutions Are Good

Typical:

```
Vector3 normalized(Vector3 a) {
  return a / a.magnitude();
}
```

Better:

```
template <typename V>
auto normalized(V a) {
  return a / magnitude(a);
}
```

Modern C++ Language Features for Numerics

- C++11
  - `constexpr`
  - `static_assert`
  - `auto`
  - `using`
  - `explicit` conversion operators
  - `auto`
- C++14
  - `auto`
  - variable templates

Open Design Questions

Exponent? It’s just `-fixed_point<>::fractional_digits`.

Are `Rep` and `Exponent` the right way ‘round?

How to support other radixes.

What to call `get_width`?

User-defined literals?

Aliases?

Performance / Efficiency?

Reference Implementation

github.com/johnmcfarlane/fixed_point/

- reference for P0037 and P0381
- stable, versioned API
- tests and benchmarks
- documentation
- integration with Boost.Multiprecision
- experimental elastic and integer class templates
- CMake: GCC 4.8, Clang 3.5, Visual C++ 14.0
- 128-bit integer support on GCC & Clang

Questions / Feedback

Impressions of API?

Anybody want to contribute / test / make something better / proofread paper?

Content missing from presentation?

Please tell me which 20 slides I need to delete!

Did you spot the deliberate mistake?