fixed_point Library

John McFarlane

CppCon2016
2016-06-08

Anatomy of a Fixed-Point Number

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

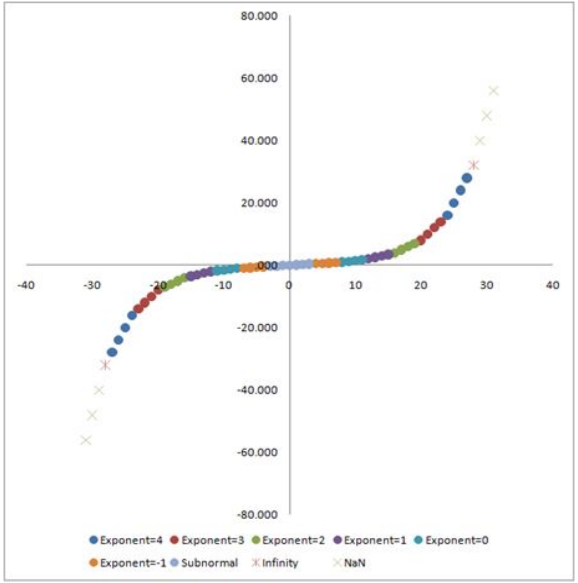
128	64	32	16	8	4	2	1		1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
I	I	I	I	I	I	I	I	.	F	F	F	F	F	F	F	F

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

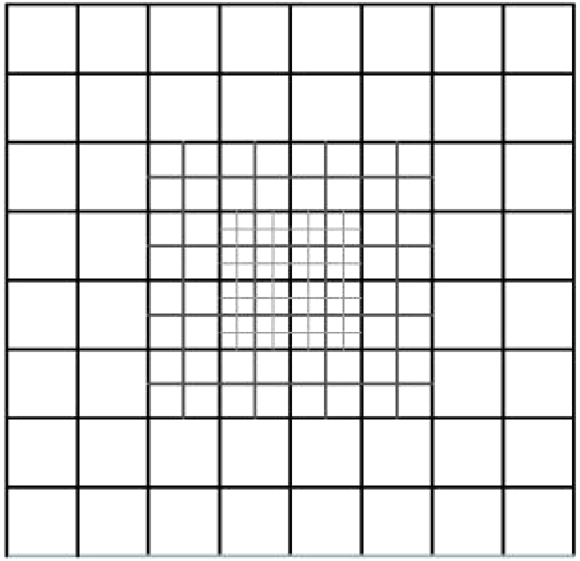
256 - 1/256 = 255.99609375

= 65535 / 256

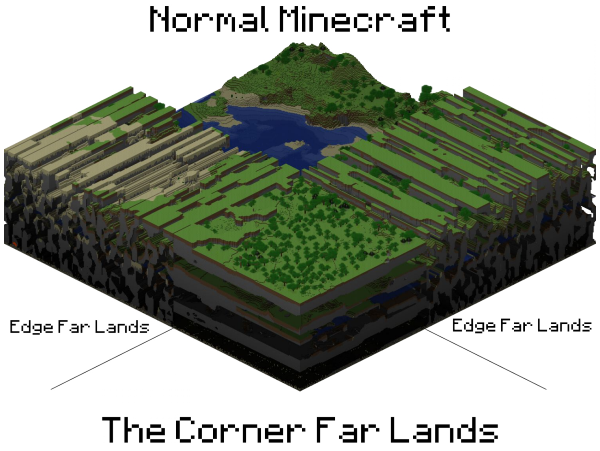
What's Wrong With Floating-Point?



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

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;
}
```

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~~

Hypothesis

Most problems with C++’s built-in fixed-point types can best be addressed **individually**.

Details:

1. Each solution involves a **literal class template**.
2. They can be instantiated with build-in types to produce numeric types which solve a **single** problem.
3. They can be combined to instantiate types which are responsible for addressing **multiple** problems.
4. This can be done at zero run-time cost.
5. This approach can minimize compile-time cost.

Suggestions

checked_integer<> - throws on errors, e.g. overflow

widening_integer<> - results of arithmetic operations widened

rounded_integer<> - better results from operations and cast from floating-point

fixed_point<> - sub-unit precision

`sg14::fixed_point<>` Class Template

Paper: P0037
Library: https://github.com/johnmcfarlane/fixed_point

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>;
```

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!

// fixed-point multiplication operator widens result
auto xx = x*x;

// x * x has type fixed_point<uint16_t, -8>
static_assert(is_same<decltype(xx), fixed_point<uint16_t, -8>>::value, "");
cout << setprecision(12) << xx << endl;  // "254.00390625" - correct

// for maximum efficiency, use named functions:
auto named_xx = multiply(x, x);

// multiply result is same as underlying representation's operation
static_assert(is_same<decltype(named_xx), fixed_point<int, -8>>::value, "");
cout << named_xx << endl;  // "254.00390625" - also correct but prone to overflow
```

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;
```

Elastication™

```
// Consider an integer type which keeps count of the bits that it uses.
auto a = elastic_integer<6, int8_t>{ 63 };

// Results of its operations widen as required.
auto aa = a*a;
static_assert(is_same<decltype(aa), elastic_integer<12, int8_t >> ::value, "");

// Obviously, this type no longer fits in a byte.
static_assert(sizeof(aa)==2, "");

// Addition requires smaller results
auto a2 = a+a;
static_assert(is_same<decltype(a2), elastic_integer<7, int8_t >> ::value, "");
```

Elastication™ + `fixed_point`

```
// Such a type can be used to specialize fixed_point.
template<int IntegerDigits, int FractionalDigits, typename Archetype>
using elastic = fixed_point<elastic_integer<IntegerDigits+FractionalDigits,
Archetype>, -FractionalDigits>;

// Now arithmetic operations are more efficient and less error-prone.
auto b = elastic<4, 28, unsigned>{15.9375};
auto bb = b*b;

cout << bb << endl;  // "254.00390625"
static_assert(is_same<decltype(bb), elastic<8, 56, unsigned>>::value, "");
```

Safety

```cpp
// a safe, 8-bit fixed-point type with range -8 <= x < 7.9375
using safe_byte = make_fixed<3, 4, boost::numeric::safe<int>>;

// prints "-8"
try {
   auto a = safe_byte{-8};
   cout << a << endl;
}
catch (std::range_error e) {
   cout << e.what() << endl;
}

// prints "Value out of range for this safe type"
try {
   auto b = safe_byte{10};
   cout << b << endl;
}
catch (std::range_error e) {
   cout << e.what() << endl;
}
```

Language Features

C++11/14

- `constexpr` - literal classes
- `auto` - novel types as results of arithmetic operations
- `decltype` - API authoring
- user-defined literals?

C++17

- template argument deduction?

C++??

- concepts

James McNellis
@JamesMcNellis

Following

Oh, thank goodness you fixed it! I hadn't even noticed the point was broken! #cppcon

> **John McFarlane** @JSAMcFarlane
> Talking about fixed-point @ 2pm on Monday at CppCon 2016 with @robertramey1 sched.co/7nMA @cppcon #cppcon

RETWEET	LIKES
1	7

5:31 PM - 21 Jul 2016

1 7