Composite Arithmetic Types Are > the + of Their Parts

John McFarlane

A9.com

Background

Background

- Game Development

Background

- Game Development
- Study Group 14 - Low Latency (Games, Embedded, HFT, etc.)

Background

- Game Development
- Study Group 14 - Low Latency (Games, Embedded, HFT, etc.)
  - P0037: "Fixed-Point Real Numbers"

Background

- Game Development
- Study Group 14 - Low Latency (Games, Embedded, HFT, etc.)
  - P0037: "Fixed-Point Real Numbers"
- Study Group 6 - Numerics

Background

- Game Development
- Study Group 14 - Low Latency (Games, Embedded, HFT, etc.)
  - P0037: "Fixed-Point Real Numbers"
- Study Group 6 - Numerics
  - Platforms: PCs, Cloud, Mobile devices, Embedded systems, GPUs / FPGAs,

Background

- Game Development
- Study Group 14 - Low Latency (Games, Embedded, HFT, etc.)
  - P0037: "Fixed-Point Real Numbers"
- Study Group 6 - Numerics
  - Platforms: PCs, Cloud, Mobile devices, Embedded systems, GPUs / FPGAs,
  - Applications: Simulation, Image Processing, Machine Learning, Information Security

Background

- Game Development
- Study Group 14 - Low Latency (Games, Embedded, HFT, etc.)
    - P0037: "Fixed-Point Real Numbers"
- Study Group 6 - Numerics
    - Platforms: PCs, Cloud, Mobile devices, Embedded systems, GPUs / FPGAs,
    - Applications: Simulation, Image Processing, Machine Learning, Information Security
    - P0554: "Composition of Arithmetic Types"

Background

- Game Development
- Study Group 14 - Low Latency (Games, Embedded, HFT, etc.)
    - P0037: "Fixed-Point Real Numbers"
- Study Group 6 - Numerics
    - Platforms: PCs, Cloud, Mobile devices, Embedded systems, GPUs / FPGAs,
    - Applications: Simulation, Image Processing, Machine Learning, Information Security
    - P0554: "Composition of Arithmetic Types"
- Should SG14 Produce a Low-latency Library?

Disclaimer

In the interest of time, this talk does not mention:

- user-defined literals,
- integral constants
- class template deduction
- operator overload resolution
- `<=>`
- `std::common_type`
- `noexcept`
- `constexpr`
- trig functions
- UB, nasal demons or why `signed>unsigned`
- Unum
- decimal representation
- rationals
- variable-width integers
- two's complement, ternary architecture or qubits

The Pitch

The Pitch

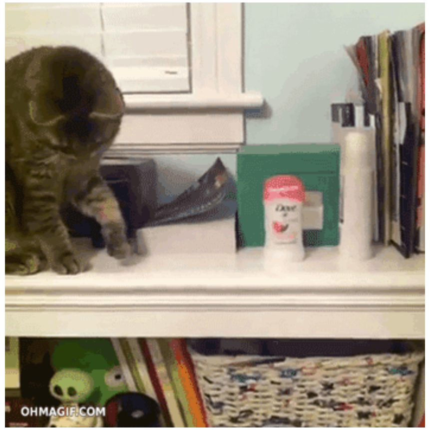
Do for `int` what the STL did for `[]`

The Pitch

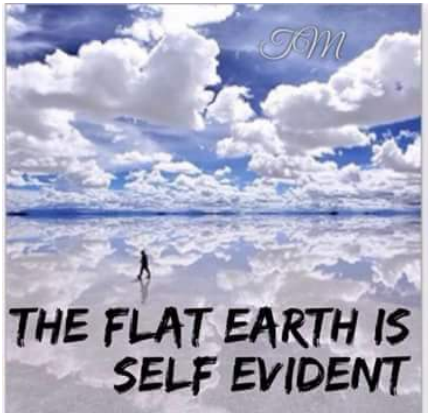
> Do for `int` what the STL did for `[]`

```
template<typename T>
using Composite = map<string, vector<unique_ptr<T>>>
```

Composability is a GOOD THING



+



=

How can I tell if my type is composite?

How can I tell if my type is composite?

Four telltale signs:

How can I tell if my type is composite?

Four telltale signs:

1. Can be composed from fundamental arithmetic types

How can I tell if my type is composite?

Four telltale signs:

1. Can be composed from fundamental arithmetic types
2. Can be substituted for fundamental arithmetic types

How can I tell if my type is composite?

Four telltale signs:

1. Can be composed from fundamental arithmetic types
2. Can be substituted for fundamental arithmetic types
3. Can be used to compose other arithmetic types

How can I tell if my type is composite?

Four telltale signs:

1. Can be composed from fundamental arithmetic types
2. Can be substituted for fundamental arithmetic types
3. Can be used to compose other arithmetic types
4. Separation of concerns

Example Numeric Types

Example Numeric Types

Example Type #1: `safe_integer<>`:

```
template<typename Rep>
class safe_integer {
public:
    // ...
private:
    Rep _rep;
};
```

Example Numeric Types

Example Type #1: `safe_integer<>`:

```
template<typename Rep>
class safe_integer {
public:
    // ...
private:
    Rep _rep;
};
```

```
// multiplication of safe_integer<int> cannot exceed numeric limits
auto a = safe_integer<int>{numeric_limits<int>::max()} * 2;  // exception!

// difference from safe_integer<unsigned> cannot be negative
auto b = safe_integer<unsigned>{0} - 1;  // exception!

// conversion to safe_integer<char> cannot exceed numeric limits
auto c = safe_integer<char>{numeric_limits<double>::max()};  // exception!

// value of safe_integer<int> cannot be indeterminate
auto d = safe_integer<int>{};  // compiler error? exception? zero-initialization?
```

Example Numeric Types

Example Numeric Types

Example Type #2: `elastic_integer<>`:

```
template<int Digits, typename Narrowest>
class elastic_integer {
  // ...
  private:
    Rep _rep;  // Narrowest or something wider
};
```

Example Numeric Types

Example Type #2: `elastic_integer<>`:

```
template<int Digits, typename Narrowest>
class elastic_integer {
  // ...
  private:
    Rep _rep;  // Narrowest or something wider
};
```

```
// elastic_integer holding 4 digits
auto a = elastic_integer<4, unsigned>{10};

// result of addition is 1 digit wider
auto b = a+a;  // elastic_integer<5, unsigned>;

// result of subtraction is signed
auto c = -b;  // elastic_integer<5, signed>;

// run-time overflow is not the concern of elastic_integer
auto d = elastic_integer<8, signed>{256};
```

How can I tell if my type is composite?

Four telltale signs:

1. **Can be composed from fundamental arithmetic types**
2. Can be substituted for fundamental arithmetic types
3. Can be built from composite arithmetic types
4. Separation of concerns

Telltale Sign #1

Can be composed from fundamental arithmetic types

```
// good
template<typename Rep>
class safe_integer;
```

Telltale Sign #1

Can be composed from fundamental arithmetic types

```
// good
template<typename Rep>
class safe_integer;
```

```
// bad
template<int Digits, bool IsSigned>
class safe_integer;
```

Telltale Sign #1

Can be composed from fundamental arithmetic types

```cpp
// good
template<typename Rep>
class safe_integer;
```

```cpp
// bad
template<int Digits, bool IsSigned>
class safe_integer;
```

```cpp
using good = safe_integer<int>;
```

Telltale Sign #1

Can be composed from fundamental arithmetic types

```
// good
template<typename Rep>
class safe_integer;
```

```
// bad
template<int Digits, bool IsSigned>
class safe_integer;
```

```
using good = safe_integer<int>;
```

```
using bad = safe_integer<31, true>;
```

Telltale Sign #1

Can be composed from fundamental arithmetic types

```
// good
template<typename Rep>
class safe_integer;
```

```
// bad
template<int Digits, bool IsSigned>
class safe_integer;
```

```
using good = safe_integer<int>;
```

```
using bad = safe_integer<31, true>;
```

```
using bad = safe_integer<numeric_limits<int>::digits, true>;
```

Telltale Sign #1

Can be composed from fundamental arithmetic types

```cpp
// good
template<typename Rep>
class safe_integer;
```

```cpp
// bad
template<int Digits, bool IsSigned>
class safe_integer;
```

```cpp
using good = safe_integer<int>;
```

```cpp
using bad = safe_integer<31, true>;
```

```cpp
using bad = safe_integer<numeric_limits<int>::digits, true>;
```

```cpp
using good = safe_integer<int32_t>;
```

How can I tell if my type is composite?

Four telltale signs:

1. Can be composed from fundamental arithmetic types
2. **Can be substituted for fundamental arithmetic types**
3. Can be built from composite arithmetic types
4. Separation of concerns

Telltale Sign #2

Can be substituted for fundamental arithmetic types

Telltale Sign #2

Can be substituted for fundamental arithmetic types

```cpp
// pch.h
#include <cstdint>
#include <safe_integer.h>

namespace acme {
#if defined(NDEBUG)
    template<typename Rep>
    using integer = Rep;
#else
    template<typename Rep>
    using integer = safe_integer<Rep>;
#endif
}
```

Telltale Sign #2

Can be substituted for fundamental arithmetic types

```
// pch.h
#include <cstdint>
#include <safe_integer.h>

namespace acme {
#if defined(NDEBUG)
    template<typename Rep>
    using integer = Rep;
#else
    template<typename Rep>
    using integer = safe_integer<Rep>;
#endif
}
```

```
auto square(acme::integer<short> f)
{
  return f * f;
}
```

Writing Transparent Operators

Writing Transparent Operators

```
template<typename Rep>
auto operator*(safe_integer<Rep> const& a, safe_integer<Rep> const& b)
{
    Rep product = a.data() * b.data();

    // do some overflow checking

    return safe_integer<Rep>{product};
}
```

Writing Transparent Operators

```
template<typename Rep>
auto operator*(safe_integer<Rep> const& a, safe_integer<Rep> const& b)
{
    Rep product = a.data() * b.data();

    // do some overflow checking

    return safe_integer<Rep>{product};
}
```

```
safe_integer<short>{2} * safe_integer<short>{3};
```

Writing Transparent Operators

```
template<typename Rep>
auto operator*(safe_integer<Rep> const& a, safe_integer<Rep> const& b)
{
    Rep product = a.data() * b.data();

    // do some overflow checking

    return safe_integer<Rep>{product};
}
```

```
safe_integer<short>{2} * safe_integer<short>{3};
```

```
safe_integer<short>{6} * safe_integer<int>{7};
```

Writing Transparent Operators

```
template<typename Rep1, typename Rep2>
auto operator*(safe_integer<Rep1> const& a, safe_integer<Rep2> const& b)
{
    auto product = a.data()*b.data();

    // do some overflow checking

    return safe_integer<decltype(product)>{product};
}
```

```
safe_integer<short>{2} * safe_integer<short>{3};
```

```
safe_integer<short>{6} * safe_integer<int>{7};
```

Friendly Advice

When you use a type's operator, don't assume its return type.

Friendly Advice

> When you use a type's operator, don't assume its return type.

```
// oh crap
auto c = safe_integer<simd::pack<int>>{} != safe_integer<simd::pack<int>>{}
```

(Courtesy of Joël Falcou)

Telltale Sign #2

Can be substituted for fundamental arithmetic types

```
// pch.h
#include <cstdint>
#include <safe_integer.h>

namespace acme {
#if defined(NDEBUG)
    template<typename Rep>
    using integer = Rep;
#else
    template<typename Rep>
    using integer = safe_integer<Rep>;
#endif
}
```

```
auto square(acme::integer<short> f)
{
  return f * f;
}
```

The Prime Directive

> "The Prime Directive is not just a set of rules. It is a philosophy, and a very correct one. History has proved again and again that whenever mankind interferes with a less developed civilization, no matter how well intentioned that interference may be, the results are invariably disastrous."

— Jean-Luc Picard

How can I tell if my type is composite?

Four telltale signs:

1. Can be composed from fundamental arithmetic types
2. Can be substituted for fundamental arithmetic types
3. **Can be built from composite arithmetic types**
4. Separation of concerns

Telltale Sign #3

Can be built from composite arithmetic types

```
template<typename Rep>
class safe_integer;

template<int Digits, typename Narrowest = int>
class elastic_integer;
```

Telltale Sign #3

Can be built from composite arithmetic types

```
template<typename Rep>
class safe_integer;

template<int Digits, typename Narrowest = int>
class elastic_integer;
```

```
template<int Digits, typename Narrowest = int>
class safe_elastic_integer;
```

Telltale Sign #3

Can be built from composite arithmetic types

```
template<typename Rep>
class safe_integer;

template<int Digits, typename Narrowest = int>
class elastic_integer;
```

```
template<int Digits, typename Narrowest = int>
using safe_elastic_integer =
  safe_integer<elastic_integer<Digits, Narrowest>>;
```

safe_elastic_integer

```
template<typename Rep1, typename Rep2>
auto operator*(safe_integer<Rep1> const& a, safe_integer<Rep2> const& b)
{
    auto product = a.data()*b.data();

    // do some overflow checking

    return safe_integer<decltype(product)>{product};
}
```

```
template<int Digits, typename Narrowest = int>
using safe_elastic_integer =
  safe_integer<elastic_integer<Digits, Narrowest>>;
```

```
auto a = safe_elastic_integer<4, int>{14} * safe_elastic_integer<3, int>{6};
```

safe_elastic_integer

```
template<typename Rep1, typename Rep2>
constexpr auto operator*(safe_integer<Rep1> const& a, safe_integer<Rep2> const& b)
{
    auto product = a.data()*b.data();

    if (numeric_limits<Rep1>::digits+numeric_limits<Rep2>::digits
            >numeric_limits<decltype(product)>::digits) {
        // do some overflow checking
    }

    return safe_integer<decltype(product)>{product};
}
```

```
template<int Digits, typename Narrowest = int>
using safe_elastic_integer =
  safe_integer<elastic_integer<Digits, Narrowest>>;
```

```
auto a = safe_elastic_integer<4>{14}*safe_elastic_integer<3>{6};
```

safe_elastic_integer

```
template<typename Rep1, typename Rep2>
constexpr auto operator*(safe_integer<Rep1> const& a, safe_integer<Rep2> const& b)
{
    auto product = a.data()*b.data();

    if (numeric_limits<Rep1>::digits+numeric_limits<Rep2>::digits
            >numeric_limits<decltype(product)>::digits) {
        // do some overflow checking
    }

    return safe_integer<decltype(product)>{product};
}
```

```
template<int Digits, typename Narrowest = int>
using safe_elastic_integer =
  safe_integer<elastic_integer<Digits, Narrowest>>;
```

```
auto a = safe_elastic_integer<4>{14}*safe_elastic_integer<3>{6};
```

```
auto b = safe_integer<short>{14} * safe_integer<short>{6};
```

How can I tell if my type is composite?

Four telltale signs:

1. Can be composed from fundamental arithmetic types
2. Can be substituted for fundamental arithmetic types
3. Can be built from composite arithmetic types
4. **Separation of concerns**

Let's Keep Going...

Let's Keep Going...

```
template<class Rep, int Exponent>
class fixed_point;
```

Let's Keep Going...

```
template<class Rep, int Exponent>
class fixed_point;
```

```
template<int Digits, class Narrowest = int>
class elastic_integer;
```

Let's Keep Going...

```
template<class Rep, int Exponent>
class fixed_point;
```

```
template<int Digits, class Narrowest = int>
class elastic_integer;
```

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

Let's Keep Going...

```
template<class Rep, int Exponent>
class fixed_point;
```

```
template<int Digits, class Narrowest = int>
class elastic_integer;
```

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

```
template<class Rep = int, class OverflowTag = throwing_overflow_tag>
class safe_integer;
```

precise_safe_elastic_fixed_point

```
// precise safe elastic fixed-point
template<
        int IntegerDigits,
        int FractionalDigits = 0,
        class OverflowTag = throwing_overflow_tag,
        class RoundingTag = rounding_closest_tag,
        class Narrowest = int>
using precise_safe_elastic_fixed_point = fixed_point<
        elastic_integer<
                IntegerDigits+FractionalDigits,
                precise_integer<
                        safe_integer<
                                Narrowest,
                                OverflowTag
                        >,
                        RoundingTag
                >
        >,
        -FractionalDigits
>;
```

fixed_point + elastic_integer

fixed_point + elastic_integer

```
// square a number using 15:16 fixed-point arithmetic
float square_int(float input)
{
    // user must scale values by the correct amount
    auto fixed = static_cast<int32_t>(input * 65536.f);

    // user must remember to widen the result to avoid overflow
    auto prod = int64_t{fixed} * fixed;

    // user must remember that the scale also was squared
    return prod / 4294967296.f;
}

// same function with added type safety
float square_fixed_point(float input)
{
    // alias to fixed_point<elastic_integer<31, int>, -16>
    auto fixed = elastic_fixed_point<15, 16>{input};

    // concise, safe and zero-cost!
    auto prod = fixed * fixed;

    return static_cast<float>(prod);
}
```

https://godbolt.org/g/C3oRXx

```
square_int(float):
        mulss   xmm0, DWORD PTR .LC0[rip]
        cvttss2si       eax, xmm0
        pxor    xmm0, xmm0
        cdqe
        imul    rax, rax
        cvtsi2ssq       xmm0, rax
        mulss   xmm0, DWORD PTR .LC1[rip]
        ret
square_fixed_point(float):
        mulss   xmm0, DWORD PTR .LC0[rip]
        cvttss2si       eax, xmm0
        pxor    xmm0, xmm0
        cdqe
        imul    rax, rax
        cvtsi2ssq       xmm0, rax
        mulss   xmm0, DWORD PTR .LC1[rip]
        ret
.LC0:
        .long   1199570944
.LC1:
        .long   796917760
```

fixed_point + Boost.Multiprecision

fixed_point + Boost.Multiprecision

```
#include <sg14/auxiliary/multiprecision.h>

void boost_example()
{
    using namespace boost::multiprecision;
    using rep = number<
        cpp_int_backend<400, 400, unsigned_magnitude, unchecked, void>>;
    using big_number = fixed_point<rep, 0>;

    auto googol = big_number{1};
    for (auto zeros = 0; zeros!=100; ++zeros) {
        googol *= 10;
    }
    cout << googol << endl;  // "1e+100"

    auto googolth = 1 / googol;
    cout << googolth << endl;  // "1e-100"
}
```

The Small Print

The Small Print

```
template<int Digits, typename Narrowest>
using make_signed_t<elastic_integer<Digits, Narrowest>>
    = elastic_integer<Digits, make_signed_t<Narrowest>>;

// elastic_integer<10, signed>
using a = make_signed_t<elastic_integer<10, unsigned>>;
```

The Small Print

```
template<int Digits, typename Narrowest>
using make_signed_t<elastic_integer<Digits, Narrowest>>
    = elastic_integer<Digits, make_signed_t<Narrowest>>;

// elastic_integer<10, signed>
using a = make_signed_t<elastic_integer<10, unsigned>>;
```

```
template<typename T>
using twice_as_wide = set_num_digits<T, numeric_limits<T>::digits * 2>;

// uint16_t
using b = twice_as_wide<uint8_t>;
```

The Small Print

```
template<int Digits, typename Narrowest>
using make_signed_t<elastic_integer<Digits, Narrowest>>
    = elastic_integer<Digits, make_signed_t<Narrowest>>;

// elastic_integer<10, signed>
using a = make_signed_t<elastic_integer<10, unsigned>>;
```

```
template<typename T>
using twice_as_wide = set_num_digits<T, numeric_limits<T>::digits * 2>;

// uint16_t
using b = twice_as_wide<uint8_t>;
```

```
auto c = safe_integer<int>{...};
auto d = multiply_overflow(saturate, c, 2);
```

numeric_traits

numeric_traits

```
namespace std {
  template<class T>
  struct numeric_traits {
    static constexpr bool is_specialized = false;
    // ...
  };

  template<>
  struct numeric_traits<int> {
    // ...
  };
}
```

numeric_traits

```
namespace std {
  template<class Rep>
  struct numeric_traits<safe_integer<Rep>> {
    static constexpr bool is_specialized = false;

    using make_signed_t = safe_integer<numeric_traits<Rep>::make_signed_t>;
    using make_unsigned_t = safe_integer<numeric_traits<Rep>::make_unsigned_t>;
    using set_width_t = safe_integer<numeric_traits<Rep>::make_unsigned_t>;

    Rep to_rep(safe_integer<Rep> const& si) {
      return si._rep;
    }

    safe_integer<Rep> from_rep(Rep const& r) {
      return safe_integer<Rep>{r};
    }

    // ...
  };
}
```

Thanks

John McFarlane, @JSAMcFarlane

Links:

- fixed-point - github.com/johnmcfarlane/fixed_point
- SG14 forum - groups.google.com/a/isocpp.org/d/forum/sg14

Papers:

- P0554: Composition of Arithmetic Types - wg21.link/p0554
- P0037: Fixed-Point Real Numbers - wg21.link/p0037
- P0101: Numeric TS Outline - wg21.link/p0101