Building a class for modular arithmetic

Writing modern C++ is not easy.

Toon Baeyens

toonijn.be
toon.baeyens@gmail.com
github.com/toonijn/modulo/

About me

I worked at

- think-cell (2023-2024)
  - C++ codebase for their PowerPoint add-in
- Ghent University (2018-2023)
  - Teaching assistant
  - PhD in mathematics: Algorithms for time-independent Schrödinger equations

I love exploring

- numerical mathematics
- beautiful visualizations
- the obscure corners of C++

Toon Baeyens
toon.baeyens@gmail.com

Why?

Polynomials of Fibonacci Numbers

Problem 435

The **Fibonacci numbers** $\{f_n, n \ge 0\}$ are defined recursively as $f_n = f_{n-1} + f_{n-2}$ with base cases $f_0 = 0$ and $f_1 = 1$.

Define the polynomials $\{F_n, n \ge 0\}$ as $F_n(x) = \sum_{i=0}^{n} f_i x^i$.

For example, $F_7(x) = x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + 13x^7$, and $F_7(11) = 268\,357\,683$.

Let $n = 10^{15}$. Find the sum $\sum_{x=0}^{100} F_n(x)$ and give your answer modulo $1\,307\,674\,368\,000\ (= 15!)$.

N^{th} Digit of Reciprocals

Problem 820

Let $d_n(x)$ be the n^{th} decimal digit of the fractional part of x, or 0 if the fractional part has fewer than n digits.

For example:

- $d_7(1) = d_7\left(\frac{1}{2}\right) = d_7\left(\frac{1}{4}\right) = d_7\left(\frac{1}{5}\right) = 0$
- $d_7\left(\frac{1}{3}\right) = 3$ since $\frac{1}{3} = 0.3333333333...$
- $d_7\left(\frac{1}{6}\right) = 6$ since $\frac{1}{6} = 0.1666666666...$
- $d_7\left(\frac{1}{7}\right) = 1$ since $\frac{1}{7} = 0.1428571428...$

Let $S(n) = \sum_{k=1}^{n} d_n\left(\frac{1}{k}\right)$.

You are given:

$$S(7) = 0 + 0 + 3 + 0 + 0 + 6 + 1 = 10$$
$$= 418$$

Square + 1 = Squarefree

Problem 864

Let $C(n)$ be the number of squarefree integers of the form $x^2 + 1$ such that $1 \le x \le n$.

For example, $C(10) = 9$ and $C(1000) = 895$.

Find $C(123567101113)$.

projecteuler.net

Table Of Content

Building a class for modular arithmetic

- Known modulus
  - member type
  - `operator==`
  - `operator+`, `operator+=`
  - `operator*`
- Unknown modulus
  - dependent types
  - selecting the type
  - optimizations
  - powers

On the way we will touch

operator precedence `std::visit` casting integers

concepts comparing integers `__uint128_t`

value categories argument-dependent lookup

`MUL`, `UMULH`, `FMUL` fixed-point arithmetic micro bechmarking

dependent types `[[assume(...)]]`

Design goals

Easy to use

constexpr modulus

```
using Mod = Modulo<17>;
```

run-time modulus

```
// Something magical to get a Mod type
```

Usage

```
auto a = Mod(3);
a -= 2;
auto b = 3 + a * 7
a += a * b;
if(a == b) {
  // ...
}
```

Design goals

Goals

- Correctness
- Easy to use
- Performance

Non-goals

- Easy to understand
- Mathematical optimizations
- Perfection

Known modulus, first design

```
template<int modulus>
struct Modulo {
  int value;

  friend Modulo<modulus> operator+(Modulo<modulus> lhs, Modulo<modulus> rhs) {
    return Modulo<modulus>((lhs.value + rhs.value) % modulus);
  }

  friend Modulo<modulus> operator*(Modulo<modulus> lhs, Modulo<modulus> rhs) {
    return Modulo<modulus>(lhs.value * rhs.value % modulus);
  }
};
```

Known modulus, first design

```
template<int modulus>
struct Modulo {
  int value;

  friend Modulo<modulus> operator+(Modulo<modulus> lhs, Modulo<modulus> rhs) {
    return Modulo<modulus>((lhs.value + rhs.value) % modulus);
  }

  friend Modulo<modulus> operator*(Modulo<modulus> lhs, Modulo<modulus> rhs) {
    return Modulo<modulus>(lhs.value * rhs.value % modulus);
  }
};
```

	Operator	Assoc.
3	`+a` `-a`	←
5	`a*b` `a/b` `a%b`	→
6	`a+b` `a-b`	→
10	`==` `!=`	→

16	`=`
	`+=` `-=`
	`*=` `/=` `%=`
		←

General datatype

```cpp
template<[...] modulus>
struct Modulo {
  using ValueType = [...];
  ValueType value;
}
```

- What should `ValueType` be?
  - Signedness?
  - `value` needs to fit.
    - Does `modulus` need to fit?
    - Does `value + value` need to fit?
    - Does `value * value` need to fit?
  - Do we care for `__uint128`?

General datatype

```
#include <type_traits>
#include <limits>
#include <cstdint>

template<std::uint64_t modulus, typename FirstAlt=void, typename... Alts>
struct ValueTypeHelper {
  using type = typename std::conditional_t<
    modulus < std::numeric_limits<FirstAlt>::max()/2,
    std::type_identity<FirstAlt>, ValueTypeHelper<modulus, Alts...>
  >::type;
};

template<std::uint64_t modulus>
using ValueType = typename ValueTypeHelper<modulus,
    std::uint8_t, std::uint16_t, std::uint32_t, std::uint64_t>::type;

static_assert(std::is_same_v<ValueType<300>, std::uint16_t>);

static_assert(std::is_same_v<ValueType<10'000'000'000>, std::uint64_t>);
```

exit code: 0
clang++

General datatype

```
#include <type_traits>
#include <limits>
#include <cstdint>

template<std::uint64_t modulus, typename FirstAlt=void, typename... Alts>
struct ValueTypeHelper {
  using type = typename std::conditional_t<
    modulus < std::numeric_limits<FirstAlt>::max()/2,
    std::type_identity<FirstAlt>, ValueTypeHelper<modulus, Alts...>
  >::type;
};

template<std::uint64_t modulus>
using ValueType = typename ValueTypeHelper<modulus,
    std::uint8_t, std::uint16_t, std::uint32_t, std::uint64_t>::type;

static_assert(std::is_same_v<ValueType<300>, std::uint16_t>);

static_assert(std::is_same_v<ValueType<10'000'000'000>, std::uint64_t>);
```

Trade-offs

- Storage
- Caching
- Arithmetic speed
- Vectorization
- Unaligned loads
- Even larger moduli

Run benchmarks!

clang++

The constructor

Let's review!

```
template<std::uint64_t modulus_>
struct Modulo {
  using ValueType = [...];
  ValueType value;

  template<typename V>
  Modulo(V raw_value)
    : value(raw_value % modulus_) {}
}
```

The constructor

Let's review!

```
template<std::uint64_t modulus_>
struct Modulo {
  using ValueType = [...];
  ValueType value;

  template<typename V>
  Modulo(V raw_value)
    : value(raw_value % modulus_) {}
}
```

- Should we avoid the implicit one-argument constructor?
- What about the copy and move constructor?
- Are the arithmetic conversions correct in *all* cases?

The constructor

```
#include "mod/value_type.h"

template<std::uint64_t modulus_> requires (0 != modulus_)
struct Modulo {
  ValueType<modulus_> value; // unsigned and 2*modulus_ fits
  static constexpr ValueType<modulus_> modulus = modulus_;

  template<std::integral I>
  Modulo(I const& raw_value)
    : value(static_cast<ValueType<modulus_>>(
        std::cmp_less(raw_value,  0)
        ? modulus - 1u - (~static_cast<std::make_unsigned_t<I>>(raw_value)) % modulus
        : std::cmp_less(raw_value,  modulus) ? raw_value : raw_value % modulus
    )) {}
};
```

Since C++20:

- Two's complement
- `std::integral`
- `std::cmp_less`

Operator overloading

```
using Mod = Modulo<17>;
Mod a{7};
```

```
Mod a{7};
a += 3;
a *= -a;
```

```
Mod a{7};
Mod b{4};
auto c = b - (a + 3) * 7;
```

- Which operators?
  - `operator+`, `operator-`, `operator*`, `operator/`
  - `operator+=`, `operator-=`, `operator*=`, `operator/=`
  - `operator+()`, `operator-()`
  - `operator++()`, `operator--()`, `operator++(int)`, `operator--(int)`
  - `operator<=>`, `operator==`
- What about assignment operators?
- What about conversion operators?
  - `operator int()`
  - `operator bool()`
- Can we reuse code between e.g. `operator+` and `operator+=`?
- Can we default some operators?
- Stream operators `operator<<()`, `operator>>()`?>

operator+= and operator+

```
#include "mod/value_type.h"

template<std::uint64_t modulus_>
struct Modulo {
  static constexpr ValueType<modulus_> modulus = modulus_;
  ValueType<modulus_> value;

  Modulo<modulus_> operator+=(Modulo<modulus_> const& rhs) {
    value += rhs.value;
    if(value >= modulus) value -= modulus;
    return *this;
  }

  template<typename Lhs, typename Rhs>
  friend decltype(auto) operator+(Lhs&& lhs, Rhs&& rhs) {
    if constexpr(std::is_same<decltype(lhs), Modulo<modulus_>&&>::value) {
      return lhs += std::forward<Rhs>(rhs);
    } else {
      return Modulo<modulus_>(std::forward<Lhs>(lhs)) += std::forward<Rhs>(rhs);
    }
  }
};
```

exit code: 0
clang++

Does this break everything?

```
struct Modulo {
  template<typename Lhs, typename Rhs>
  friend Modulo operator+(Lhs&& lhs, Rhs&& rhs) { ... }
};
```

A name first declared in a friend declaration within a class or class template X becomes a member of the innermost enclosing namespace of X, but is not visible for lookup (except argument-dependent lookup that considers X) unless a matching declaration at namespace scope is provided [...].

cppreference.com/w/cpp/language/friend#Notes

Does this break everything?

```
struct Modulo {
  template<typename Lhs, typename Rhs>
  friend Modulo operator+(Lhs&& lhs, Rhs&& rhs) { ... }
};
```

A name first declared in a friend declaration within a class or class template X becomes a member of the innermost enclosing namespace of X, but is not visible for lookup (except argument-dependent lookup that considers X) unless a matching declaration at namespace scope is provided [...].

cppreference.com/w/cpp/language/friend#Notes

```
struct Modulo {
  template<typename Lhs, typename Rhs>
  requires (std::is_same_v<std::decay_t<Lhs>, Modulo> || std::is_same_v<std::decay_t<Rhs>, Modulo>)
  friend Modulo operator+(Lhs&& lhs, Rhs&& rhs) { ... }
};
```

operator*

```cpp
#include <cstdint>
using u64 = std::uint64_t;

template<u64 modulus>
u64 plus(u64 lhs, u64 rhs) {
  lhs += rhs;
  if(modulus <= lhs) lhs -= modulus;
  return lhs;
}

template u64 plus<711>(u64, u64);



template<u64 modulus>
u64 product(u64 lhs, u64 rhs) {
  return lhs * rhs % modulus;
}

template u64 product<711>(u64, u64);
```

```asm
_Z4plusILm711EEmmm:                     # @_Z4plusILm711EEm
# %bb.0:
        lea     rcx, [rsi + rdi]
        cmp     rcx, 711
        lea     rax, [rsi + rdi - 711]
        cmovb   rax, rcx
        ret
                                        # -- End function
_Z7productILm711EEmmm:                  # @_Z7productILm711
# %bb.0:
        imul    rdi, rsi
        movabs  rcx, -5163012757620535543
        mov     rax, rdi
        mul     rcx
        shr     rdx, 9
        imul    rax, rdx, 711
        sub     rdi, rax
        mov     rax, rdi
        ret
                                        # -- End function
```

Aside: compilers hate div

Example: 16-bit division by 5

To calculate `x / 5`:

- Multiply `x` and `(0011 0011 0011 0011)₂`

```
1 / 5 == (0.0011 0011 0011...)₂
```

- The high 16-bit are the quotient `q`
- `x - 5 * q` is the remainder

Aside: compilers hate div

Example: 16-bit division by 5

To calculate `x / 5`:

- Multiply `x` and `(0011 0011 0011 0011)₂`

```
1 / 5 == (0.0011 0011 0011...)₂
```

- The high 16-bit are the quotient `q`
- `x - 5 * q` is the remainder

In hardware:

- x86: `MUL` computes both low and high bits
- ARM: `MUL` computes low bits, `UMULH` computes high bits
- AVR: `MUL` computes low bits, `FMUL` computes high bits

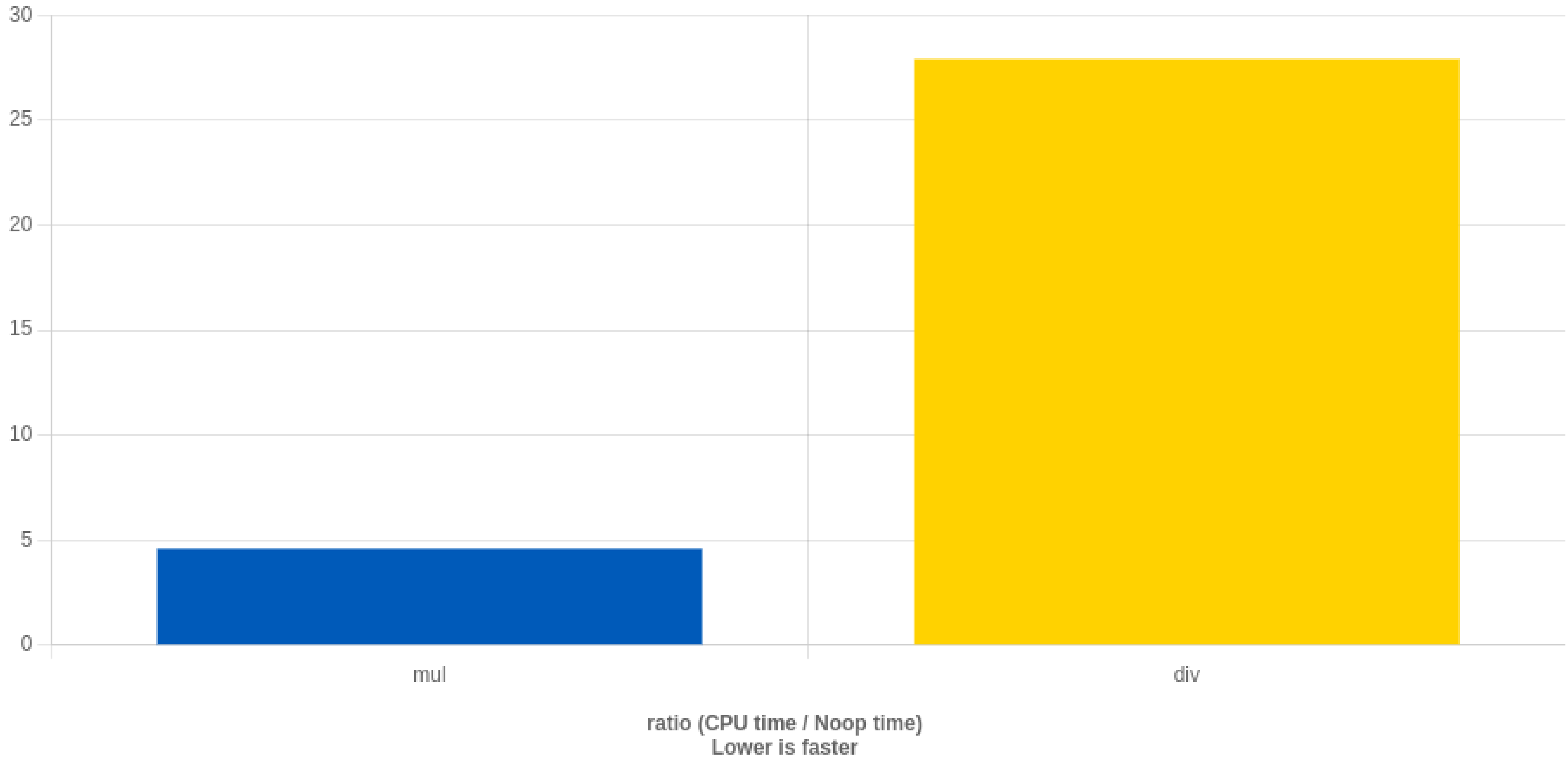
Aside: compilers hate div

```cpp
std::uint64_t volatile zero = 0;
static void mul(benchmark::State& state) {
  std::uint64_t lhs = zero;
  std::uint64_t rhs = zero;
  std::uint64_t constexpr modulus = 711;

  for (auto _ : state) {
    benchmark::DoNotOptimize(lhs * rhs % modulus);
    ++lhs, ++rhs;
  }
}
```

```cpp
std::uint64_t volatile zero = 0;
static void div(benchmark::State& state) {
  std::uint64_t lhs = zero;
  std::uint64_t rhs = zero;
  std::uint64_t modulus = zero + 711;

  for (auto _ : state) {
    benchmark::DoNotOptimize(lhs * rhs % modulus);
    ++lhs, ++rhs;
  }
}
```



quick-bench.com

operator*

```cpp
#include <bit>
#include <cstdint>
using u64 = std::uint64_t;

template<u64 modulus>
u64 product(u64 lhs, u64 rhs) {
  if constexpr(std::bit_width(modulus) < 32) {
    return lhs * rhs % modulus;
  } else {
    return static_cast<u64>(
      static_cast<__uint128_t>(lhs)
      * rhs % modulus
    );
  }
}

template
u64 product<711>(u64, u64);

template
u64 product<10'000'000'000>(u64, u64);
```

```asm
_Z7productILm711EEmmm:                  # @_Z7productILm71
# %bb.0:
        imul    rdi, rsi
        movabs  rcx, -5163012757620535543
        mov     rax, rdi
        mul     rcx
        shr     rdx, 9
        imul    rax, rdx, 711
        sub     rdi, rax
        mov     rax, rdi
        ret
                                        # -- End function
_Z7productILm10000000000EEmmm:          # @_Z7productILm10
# %bb.0:
        push    rax
        mov     rax, rsi
        mul     rdi
        movabs  rcx, 10000000000
        mov     rdi, rax
        mov     rsi, rdx
        mov     rdx, rcx
        xor     ecx, ecx
        call    __umodti3@PLT
        pop     rcx
        ret
                                        # -- End function
```

Table Of Content

Building a class for modular arithmetic

- Known modulus
  - member type
  - `operator==`
  - `operator+`, `operator+=`
  - `operator*`
- Unknown modulus
  - dependent types
  - selecting the type
  - optimizations
  - powers

Run-time modulus

What should `Mod` be?

```
// Something magical to get a Mod type
auto a = Mod(3);
a -= 2;
auto b = 3 + a * 7
a += a * b;
```

- Could it dependent on requested modulus?
- Should it store the modulus?
- What should the value type be?
- How do we choose it at runtime?

Compile-time modulus

```
template<typename Info>
struct ModuloImpl {
  typename Info::ValueType value;

  template<std::integral I>
  ModuloImpl(I const&);

  ModuloImpl<Info>& operator+=(
    ModuloImpl<Info> const& rhs
  ) {
    value += rhs.value;
    if(value >= Info::modulus)
      value -= Info::modulus;

    return *this;
  }
};
```

Known modulus

```
template<std::uint64_t modulus_>
struct CTModuloInfo {
  using ValueType = [...];
  static constexpr ValueType modulus = modulus_;
};

template<std::uint64_t modulus_>
using Modulo = ModuloImpl<CTModuloInfo<modulus_>>;
```

Unknown modulus

```
template<typename VT>
struct RTModuloInfo {
  using ValueType = VT;
  static ValueType modulus;
};
```

Aside: dependent type

> *In computer science and logic, a dependent type is a type whose definition depends on a value. It is an overlapping feature of type theory and type systems. [...]*
> *[A common example of dependent types are dependent functions.] The return type of a dependent function may depend on the value (not just type) of one of its arguments. For instance, a function that takes a positive integer `n` may return an array of length `n`, where the array length is part of the type of the array.*
>
> en.wikipedia.org/wiki/Dependent_type

```
// Fictional example
auto array_of_length(std::size_t n) -> std::array<double, n> {
  // ...
}
```

Aside: dependent type

In computer science and logic, a dependent type is a type whose definition depends on a value.

en.wikipedia.org/wiki/Dependent_type

```
std::variant<int, double> v = get_a_variant_from_somewhere();

std::visit([](auto&& arg) {
  std::cout << arg << std::endl;
}, v);
```

The *value* of `v` determines what the *type* of `arg` will be.

Type dependent on modulus

Since C++20 we can write:

```
modulo(17, []<typename Mod>() {
  auto a = Mod(3);
  a -= 8;
  std::cout << a.value << std::endl;
});
```

or

```
int modulus;
std::cin >> modulus;
bool is_special = modulo(modulus, []<typename Mod>() {
  auto a = Mod(3);
  a -= 8 * a;
  return a == -1;
});
```

But, what happens in `modulo(std::uint64_t modulus, auto lambda)` ?

Selecting the correct type

```cpp
template<typename Info>
struct ModuloImpl {
  typename Info::ValueType value;

  template<std::integral I>
  ModuloImpl(I const&);

  ModuloImpl<Info>& operator+=(
    ModuloImpl<Info> const& rhs
  ) {
    value += rhs.value;
    if(value >= Info::modulus)
      value -= Info::modulus;

    return *this;
  }
};
```

```cpp
#include "mod/modulo_impl.h"

template<typename VT>
struct Info {
  using ValueType = VT;
  inline static thread_local ValueType modulus = 0;
};

auto modulo(std::uint64_t modulus, auto fn) {
  if(std::bit_width(modulus) < 32) {
    Info<u32>::modulus = modulus;
    return fn.template operator()<ModuloImpl<Info<u32>>>();
  } else {
    assert(std::bit_width(modulus) < 64);
    Info<u64>::modulus = modulus;
    return fn.template operator()<ModuloImpl<Info<u64>>>();
  }
}

int main() {
  modulo(7, []<typename Mod>() {
    Mod m(3);
    m += 2;
  });
}
```

```
exit code: 0
```
clang++

Optimizing the constructor

```cpp
#include <cstdint>
#include <utility>
#include <concepts>
using u64 = std::uint64_t;

inline u64 modulus = 711;

template<std::integral I>
u64 construct(I const& value) {
  //[[assume(17 <= modulus)]];
  return std::cmp_less(value,  0)
    ? modulus - 1u - (
      ~static_cast<std::make_unsigned_t<I>>(value)
    ) % modulus
    : std::cmp_less(value,  modulus)
      ? value
      : value % modulus;
}

u64 f() {
  return construct(3);
}
```

```asm
_Z1fv:                                  # @_Z1fv
# %bb.0:
        mov     rcx, qword ptr [rip + modulus]
        mov     eax, 3
        cmp     rcx, 3
        ja      .LBB0_2
# %bb.1:
        mov     al, 3
        movzx   eax, al
        div     cl
        movzx   eax, ah
        ret
                                        # -- End function
modulus:
```

Optimizing the operator*

Simple

- Use `div`
- If needed use `__uint128_t`

Balanced

- Use `div`
- Avoid `__uint128_t`
- Compute `2**64 % modulus` once

```
m64 = (~0ull)%modulus + 1; // only once

// msvc: _umul128
// gcc: __uint128_t
std::tie(result, high) = mul(lhs, rhs);
while(high >= 0) {
  std::tie(low, high) = mul(high, m64);
  // msvc: _addcarry_u64
  if(__builtin_add_overflow(
    result, low, &result
  )) ++high;
}
result %= modulus;
```

Performant

- Compute magic numbers once
- Always avoid `div`

Taking powers

```cpp
#include "mod/modulo.h"
#include <iostream>

template<typename Int, typename Exp>
Int pow(Int i, Exp e) {
  Int result = 1;
  while(e) {
    if(e % 2 == 1) result *= i;
    i *= i;
    e /= 2;
  }
  return result;
}

int main() {
  for(int mod = 3; mod < 20; mod += 2)
    modulo(mod, [&]<typename Mod>() {
      auto a = Mod(3);
      std::cout << a.value << "**8 = " << pow(a, 8).value
                << " (mod " << mod <<")" << std::endl;
    });
}
```

```
0**8 = 0 (mod 3)
3**8 = 1 (mod 5)
3**8 = 2 (mod 7)
3**8 = 0 (mod 9)
3**8 = 5 (mod 11)
3**8 = 9 (mod 13)
3**8 = 6 (mod 15)
3**8 = 16 (mod 17)
3**8 = 6 (mod 19)
```

Conclussions

operator precedence `std::visit` casting integers concepts

comparing integers `__uint128_t` value categories

argument-dependent lookup `MUL`, `UMULH`, `FMUL`

fixed-point arithmetic micro bechmarking dependent types

`[[assume(...)]]`

Toon Baeyens
toon.baeyens@gmail.com