# Руководство по реализации криптографии на эллиптических кривых

Санан Корняков

2024

# Содержание

**Аннотация**

Работа является пошаговым руководством по реализации криптографии на эллиптических кривых. Реализованы объекты длинной арифметики, полей и эллиптических кривых. Изучены и имплементированы алгоритмы шифрования и дешифрования, электронной цифровой подписи, подсчёт количества точек на эллиптической кривой, быстрого умножения и деления длинных чисел. Протестированы объекты и алгоритмы по скорости, сравнивая с готовыми решениями. Руководство параллельно с имплементацией объясняет и рассказывает, что и зачем было реализовано.

*Ключевые слова: эллиптические кривые, шифрование и дешифрование, криптография, ECDSA, ECC, длинная арифметика, FFT, C++, конечные поля, оптимизация, Schoof's algorithm*

# 1 Введение

Современная криптография с нынешними вычислительными мощностями требует значительных ухищрений в шифровке сообщений, и шифрование с помощью эллиптических кривых - один из мощнейших инструментов. Но доступных и полных объяснений от начала до конца по шифрованию на них ничтожно мало, поэтому я решил сделать руководство для людей, которые хотят ознакомиться с данным видом криптографии.

## 1.1 Условия игры

Если вы искали данное руководство, то скорее всего где-то слышали/читали об эллиптических кривых и о возможности криптографии на них, поэтому я рассчитываю на базовое понимание математики и алгоритмов.

Здесь не будет дотошного доказательства теорем или строгости в описании математических объектов — в первую очередь акцент делается именно на имплементации (на языке C++). Данный язык был выбран в качестве общеизвестного языка среди программистов. Выберем C++20 для удобного использования шаблонов.

В данном руководстве мы будем стараться использовать как можно меньше готовых библиотек, чтобы не было огромных black box-ов в нашем коде. Это улучшит понимание и возможности алгоритмов.

## 1.2 База

Знаменитая формула

$$y^2 = x^3 + ax + b$$

обычно является самой первой, которую вы увидите при описании криптографии эллиптических кривых. Появляется несколько вопросов:

- Что такое $x, y, a, b$? Где лежат данные числа?

  Данные числа являются элементами некоего поля $\mathbb{F}$, над которым построенна эллиптическая кривая, характеристики больше 3 (забьём на последние слова, так как мы будем работать с полями достаточно больших характеристик). Поле поддерживает все стандартные математические операции: сложение, вычитание, деление на ненулевой элемент, умножение, поэтому можно пока считать его $\mathbb{R}$.

- Что такое эллиптическая кривая?

  Это группа точек в $\mathbb{F}^2$, координаты которых удовлетворяют данному уравнению, и ещё точка бесконечности $\mathcal{O}$, которая является своеобразным нулём группы. Сложение в группе происходит по специальным формулам на координаты, которые будут рассмотрены позже. Умножение точки на натуральное число приравнивается к сложению точки с собой это число раз.

- Как это используют для шифрования?

  Обычно выбирается эллиптическая кривая $\mathbb{E}$ над неким полем $\mathbb{F}$, точка $P$ на ней и производится умножение точки на натуральное число $k$. Криптографическая стойкость достигается сложностью нахождения числа $k$ по точкам $P$ и $kP$.

- Чем это лучше других методов шифрования?

  Тем, что данный способ шифрования можно реализовать так, что он будет выполнятся быстрее других алгоритмов при аналогичной задаче и данных. Также, для одинаковых показателей криптографической стойкости, криптография на эллиптических кривых требует ключей (чисел для шифрования) меньшей длины, чем другие алгоритмы.

### 1.3 Постановка задачи

Начитавшись статей на хабре, мы воодушевились и решили написать свою криптографию на эллиптических кривых. Сначала надо определить, какие объекты нам надо реализовать:

- Нам надо реализовать эллиптическую кривую. Но эллиптическая кривая никто без поля, значит нам надо реализовать поле.

- Так как поля бывают бесконечными, а мы работаем на компьютере с числами, то ограничимся на простые поля $\mathbb{F}_p$, которые представим в виде вычетов по простому модулю $p$. Но этот простой модуль и числа в поле надо представить в виде целых чисел, а в криптографии обычно используются числа из более чем 200 битов. Целые числа такого размера не поддерживаются языком C++, поэтому нам надо реализовать класс целых чисел и длинную арифметику на них.

Итого 3 объекта: целые числа, поле, эллиптическая кривая. Приступим наконец к реализации!

## 2 Длинная арифметика

**Если не хотите запариваться и сделать основную рабочую лошадку блэкбоксом, то можно просто установить длинку boost::multiprecision и скипнуть данную часть.**

Так как мы хотим реализовать вычеты по большому модулю, то достаточно реализовать беззнаковые длинные целые числа.

Основной приём для имплементации длинной арифметики - хранение чисел в основании $2^{32}$ или $2^{64}$. То есть просто массив из целых чисел, которые представляют цифры данного числа в соответствующих основаниях. Есть несколько видов данного представления:

1. Количество цифр в числе меняется в зависимости от размера числа. Нет математического ограничения длины числа, только аппаратное.

2. Количество цифр в числе фиксировано и не меняется от числа.

Последний вариант можно видеть, например, в типе uint64_t — присутствуют сразу все 64 бита не зависимо от содержащихся данных. Данный вид целых чисел является наиболее удобным в реализации и использовании по назначению, поэтому будем имплементировать его.

### 2.1 Каркас

Так как количество бит в числе может разительно отличатся от задачи к задаче, то общим решением будет создать шаблонный класс по количеству содержащихся в нём бит:

```
template<size_t c_bits>
class uint_t {
};
```

Теперь надо определиться с представлением цифр в нашем классе. Из-за того, что для алгоритма деления, который будет позже, потребуется деление по две цифры, то возьмём за цифру uint32_t, чтобы можно было спокойно делить в uin64_t.

С помощью constexpr определим размеры и длину необходимого массива. Так как длина не изменяется во время жизни объекта, то возмём std::array за контейнер. Он будет гарантировать, что длина массива сохраняет свой инвариант. Итого получилось:

```
template<size_t c_bits>
class uint_t {
    using digit_t = uint32_t;
    using double_digit_t = uint64_t;

    static constexpr size_t c_bits_in_byte = 8;
    static constexpr size_t c_digit_size = sizeof(digit_t) * c_bits_in_byte;
    static constexpr size_t c_digit_number = c_bits / c_digit_size;
    static constexpr size_t c_double_digit_size = sizeof(double_digit_t) * c_bits_in_byte;
    static constexpr size_t c_double_digit_number = c_bits / c_double_digit_size;

    template<size_t V>
```

```cpp
13     friend class uint_t;
14 
15     using digits = std::array<digit_t, c_digit_number>;
16     digits m_digits = {};
17 };
```

Дали псевдонимы используемым типам, чтобы улучшить читаемость и не менять все типы, если вдруг захотим использовать за цифру uint16_t и какой-то другой тип. В m_digits храним число в основании $2^{32}$ в little-endian. 12-13 строчкой мы подружили все шаблоны друг с другом для общего взаимодействия.

## 2.2 Методы

Теперь мы хотим как-то общаться с нашими данными. Желательно имплементировать все операции, которые можно применять к обычным типам, таким как uint32_t, uint64_t.

- Конструирование: Есть несколько сценариев:

  1. Хотим сконструироваться от целочисленного типа. Заметим, что количество бит в нём может быть как больше 32, так и меньше, поэтому ограничиться назначением m_blocks[0] нельзя. Тогда определим вспомогательный приватный метод, который будет определять через концепты отношения количества бит:

```cpp
template<typename From, typename To>
concept is_upcastable_to = sizeof(From) <= sizeof(To) && is_convertible_to<From, To>;

template<typename From, typename To>
concept is_downcastable_to = sizeof(From) > sizeof(To) && is_convertible_to<From, To>;

template<typename T>
requires std::numeric_limits<T>::is_integer && is_upcastable_to<T, digit_t>
static constexpr digits split_into_digits(T value) {
    return {static_cast<digit_t>(value)};
}

template<typename T>
requires std::numeric_limits<T>::is_integer && is_downcastable_to<T, digit_t>
static constexpr digits split_into_digits(T value) {
    digits result = {};

    for (size_t i = 0; i < c_digit_number; ++i) {
        result[i] = static_cast<digit_t>(value);
        value >>= c_digit_size;

        if (value == 0) {
            break;
        }
    }

    return result;
}
```

  2. Хотим сконструироваться от контейнера целых чисел, например как наш uint_t. Для этого пишем концепт, который определяет, что данный тип действительно является контейнером целых чисел, и копируем его данные:

```cpp
template<typename Container, typename T>
concept is_convertible_container = requires(Container t, size_t i) {
    { t[i] } -> is_convertible_to<T>;
    { t.size() } -> std::same_as<size_t>;
};
        template<typename T>
requires is_convertible_container<T, digit_t> || requires(T x) {
    { uint_t {x} } -> std::same_as<T>;
}
static constexpr digits split_into_digits(const T& other) {
    const size_t min_size = std::min(size(), other.size());
    digits result = {};

```

```
    for (size_t i = 0; i < min_size; i++) {
        result[i] = static_cast<digit_t>(other[i]);
    }

    return result;
}
```

3. Хотим сконструироваться от строк (C-строк). Действительно, это единственный удобный способ задать необходимое нам число с количеством бит больше чем у size_t.

   Для этого нам нужно сначала написать парсер строки, которая представляет число в двоичном, шестнадцатиричном и десяточном форматах. Так как мы хотим использовать методы uint_t, то нам нужно парсить в любой класс, который удовлетворяет критериям целочисленных классов:

```
template<typename T>
concept is_integral = std::is_integral_v<T> || requires(T t, T* p, void (*f)(T)) {
    f(0);
    p + t;
};
template<typename T>
requires is_integral<T>
constexpr T parse_into_uint(const char* str) {
    assert(str != nullptr && "parse_into got nullptr");

    T value = 0;
    uint16_t radix = 10;

    if (str[0] == '0' && str[1] == 'x') {
        radix = 16;
        str += 2;
    } else if (str[0] == '0' && str[1] == 'b') {
        radix = 2;
        str += 2;
    } else if (str[0] == '0') {
        radix = 8;
        ++str;
    }

    while (*str != '\0') {
        value *= static_cast<T>(radix);
        uint16_t symbol_value = radix + 1;

        if (*str >= '0' && *str <= '9') {
            symbol_value = static_cast<uint16_t>(*str - '0');
        } else if (*str >= 'a' && *str <= 'f') {
            symbol_value = static_cast<uint16_t>(*str - 'a') + 10;
        } else if (*str >= 'A' && *str <= 'F') {
            symbol_value = static_cast<uint16_t>(*str - 'A') + 10;
        }

        if (symbol_value >= radix) {
            assert(false && "parse_into got incorrect string");
        }

        value += static_cast<T>(symbol_value);
        ++str;
    }

    return value;
}
```

Теперь мы готовы определить конструкторы класса:

```
constexpr uint_t() = default;

template<typename T>
constexpr uint_t(const T& value) : m_digits(split_into_digits<T>(value)) {}

constexpr uint_t(const char* str) : m_digits(parse_into_uint<uint_t>(str).m_digits) {};

```

```
constexpr uint_t& operator=(const uint_t& value) = default;
```

Специально не делаем их explicit для неявных конвертаций. С помощью шаблонного конструктора мы можем конструироваться от других инстансов нашего класса, например:

```
uint_t<128> a = ...;
uint_t<160> b(a);
```

- Сложение: Самое простое, но тем не менее лучшее решение - это сложение в столбик. Используем стандартизированное переполнение беззнаковых типов в C++ для определения, есть ли остаток от сложения наших 32-битных чисел:

```
constexpr uint_t& operator+=(const uint_t& other) {
    digit_t carry = 0;

    for (size_t i = 0; i < c_digit_number; ++i) {
        digit_t sum = carry + other[i];
        m_digits[i] += sum;
        carry = (m_digits[i] < sum) || (sum < carry);
    }

    return *this;
}
```

Используем ключевое слово constexpr для вычисления значения некоторых констант во время компиляции.

Теперь мы хотим определить простое сложение, т.е. operator+. Его можно было бы сделать через

```
constexpr uint_t operator+(const uint_t& other) const {
    uint_t result = *this;
    result += other;
    return result;
}
```

Обратите внимание, что мы не пишем:

```
constexpr uint_t operator+(const uint_t& other) const {
    uint_t result = *this;
    return result += other;
}
```

так как тогда мы будем возвращать ссылку uint_t&, что не затриггерит NRVO.

Вместо определения метода класса, мы напишем 4 дружественных функции:

```
friend constexpr uint_t operator+(const uint_t& lhs, const uint_t& rhs) {
    uint_t result = lhs;
    result += rhs;
    return result;
}

friend constexpr uint_t operator+(uint_t&& lhs, const uint_t& rhs) {
    lhs += rhs;
    return lhs;
}

friend constexpr uint_t operator+(const uint_t& lhs, uint_t&& rhs) {
    rhs += lhs;
    return rhs;
}

friend constexpr uint_t operator+(uint_t&& lhs, uint_t&& rhs) {
    lhs += rhs;
    return lhs;
}
```

Тут есть два оптимизационных момента:

1. Эффективно используется то, что нам передали r – value, и не копируем данные. Обычно это возникает при многократном сложении или в других сложных формулах:

```
uint_t a,b,c = ...;
uint_t result = a + b + c;
```

2. Теперь можно неявно заапкастить другие типы к uint_t, чтобы применить данное сложение. Это позволяет писать:

```
uint_t a = ...;
uint_t b = 3 + a;
```

что было бы невозможно при внутреннем методе. Везде далее будем использовать по возможности внешние friend функции для возможности неявного апкаста других типов.

- Вычитание: Оно абсолютно аналогично делается через вычитание в столбик:

```
constexpr uint_t& operator-=(const uint_t& other) {
    digit_t remainder = 0;

    for (size_t i = 0; i < c_digit_number; ++i) {
        digit_t prev = m_digits[i];
        digit_t sum = other[i] + remainder;
        m_digits[i] -= sum;
        remainder = (m_digits[i] > prev) || (sum < remainder);
    }

    return *this;
}
```

Так как мы хотим писать такие конструкции как:

```
uint_t a = ...;
uint_t b = -a;
```

то нужно определить отрицание. В компьютерах отрицательные целые числа представляются как флипнутые биты $+$ 1. Рассмотрим на примере:

$$00000000000000000101$$
$$+$$
$$11111111111111111011$$
$$=$$
$$00000000000000000000$$

Определим inplace отрицание как приватный метод:

```
constexpr void negative() {
    for (size_t i = 0; i < c_digit_number; ++i) {
        m_digits[i] = ~(m_digits[i]);
    }

    ++*this;
}
```

Значит отрицанием будет:

```
constexpr uint_t operator-() const {
    uint_t result = *this;
    result.negative();
    return result;
}
```

Определяем внешние friend для вычитания:

```
friend constexpr uint_t operator-(const uint_t& lhs, const uint_t& rhs) {
    uint_t result = lhs;
    result -= rhs;
    return result;
}

friend constexpr uint_t operator-(uint_t&& lhs, const uint_t& rhs) {
    lhs -= rhs;
    return lhs;
}

friend constexpr uint_t operator-(const uint_t& lhs, uint_t&& rhs) {
    rhs -= lhs;
    rhs.negative();
    return rhs;
}

friend constexpr uint_t operator-(uint_t&& lhs, uint_t&& rhs) {
    lhs -= rhs;
    return lhs;
}
```

- Умножение: Стабильным и надёжным способом будет умножение в столбик. TODO
- Деление: Чтобы поделить два целочисленных длинных числа используем алгоритм-D Кнута:

  Задача - поделить два длинных числа, представленных цифрами с основанием $b$, где $b$ в имплементации $2^{32}$ или $2^{64}$.

  Рассмотрим сначала $u = (u_n u_{n-1} \dots, u_0)_b$ и $v = (v_{n-1} \dots v_0)_b$, где $u/v < b$. Найдём алгоритм для вычисления $q := \lfloor u/v \rfloor$:

  Заметим, что $u/v < b \Leftrightarrow u/b < v \Leftrightarrow \lfloor u/b \rfloor < v$, а это условие того, что

  $$(u_n u_{n-1} \dots, u_1)_b < (v_{n-1} \dots v_0)_b$$

  Если обозначить $r := u - qv$, то $q$ - это уникальное число, такое что $0 \leqslant r < v$. Пусть

  $$\hat{q} := \min\left(\left\lfloor \frac{u_n b + u_{n-1}}{v_{n-1}} \right\rfloor, b - 1\right)$$

  Т.е. мы получаем гипотетическое значение $q$, поделив первые две цифры $u$ на первую цифру $v$, а если результат деления больше или равен $b$, то берём $b - 1$. Для такого $\hat{q}$ выполняются две теоремы:

  **Теорема 1.** $\hat{q} \geqslant q$

  **Теорема 2.** *Если* $v_{n-1} \geqslant \lfloor b/2 \rfloor$, *то* $\hat{q} - 2 \leqslant q \leqslant \hat{q}$.

  Существенно ограничили нашу гипотезу. Умножив $u$ и $v$ на $\lfloor b/(v_{n-1} + 1) \rfloor$, мы не изменим длину числа $v$ и результат деления. После этого умножения станет выполнятся вторая из данных теорем.

  **Алгоритм** $D$**:** Дано неотрицательное целое число $u = (u_{m+n-1}, \dots, u_1, u_0)_b$ и $v = (v_{n-1}, \dots, v_1, v_0)_b$, где $v_{n-1} \neq 0$ и $n > 1$. Мы хотим посчитать $\lfloor u/v \rfloor = (q_m, q_{m-1}, \dots, q_0)_b$ и остаток $u \bmod v = (r_{n-1}, \dots, r_0)_b$:

  1. $d := \lfloor b/(v_{n-1} + 1) \rfloor$. Тогда пусть $(u_{m+n} u_{m+n-1} \dots u_1 u_0)_b := (u_{m+n-1} \dots u_1 u_0)_b \cdot d$, аналогично, $(v_{n-1}, \dots, v_1, v_0)_b = (v_{n-1}, \dots, v_1, v_0)_b \cdot d$. Заметим, что новая цифра могла появиться только у $u$
  2. Итерироваться будем по $j$, которая в начале равна $m$ (Делить в следующих шагах будем $(u_{j+n} \dots u_{j+1} u_j)_b$ на $(v_{n-1} \dots v_1 v_0)_b$ чтобы получить цифру $q_j$)
  3. $\hat{q} := \left\lfloor \frac{u_{j+n} b + u_{j+n-1}}{v_{n-1}} \right\rfloor$ и пусть $\hat{r}$ будет остатком, т.е. $\hat{r} := u_{j+n} b + u_{j+n-1} \pmod{v_{n-1}}$
  4. Если $\hat{q} \geqslant b$ или $\hat{q} v_{n-2} > b\hat{r} + u_{j+n-2}$, то уменьшаем $\hat{q}$ на 1 и увеличиваем $\hat{r}$ на $v_{n-1}$. Если $\hat{r} < b$, то повторяем данный шаг

5. Заменим $(u_{j+n} \dots u_{j+1} u_j)_b$ на

$$(u_{j+n} \dots u_{j+1} u_j)_b - \hat{q}(0 v_{n-1} \dots v_1 v_0)_b$$

6. Назначаем $q_j = \hat{q}$
7. Если число $u$ на 5 шаге получилось отрицательным, то добавляем к нему $b^{n+1}$ и переходим к шагу 8, иначе переходим к шагу 9.
8. (Вероятность данного шага крайне мала, за счёт чего достигается асимптотическая быстрота алгоритма) Уменьшаем $q_j$ на 1 и добавляем $(0 v_{n-1} \dots v_1 v_0)_b$ к $(u_{j+n} \dots u_{j+1} u_j)_b$ (при сложении появится цифра $u_{j+n+1}$, её следует проигнорировать)
9. Уменьшаем $j$ на 1. Если $j \geqslant 0$, то возвращаемся на шаг 3
10. Теперь $q = (q_m \dots q_1 q_0)$ - это искомое частное, а искомый остаток можно получить, поделив $(u_{n-1} \dots u_1 u_0)$ на $d$ наивным способом
11. Возвращаем $(q, r)$

Алгоритм сверху применяется только при размере делителя больше 1 цифры и не больше количества цифр в делимом, так как при меньших размерах есть более быстрые оптимизации, значит нам понадобится приватный метод определения количества цифр в числе:

```
constexpr size_t actual_size() const {
    size_t result = c_digit_number;

    while (result > 0 && m_digits[result - 1] == 0) {
        --result;
    }

    return result;
}
```

Для удобной работы с m_digits определим приватные методы для operator[], которые будут проталкивать его внутрь:

```
constexpr const digit_t& operator[](size_t pos) const {
    return m_digits[pos];
}

constexpr digit_t& operator[](size_t pos) {
    return m_digits[pos];
}
```

Наконец, определим приватный метод divide, который будет вычислять, в каком случае мы находимся.

```
static constexpr uint_t divide(const uint_t& lhs, const uint_t& rhs, uint_t* remainder = nullptr) {
    size_t dividend_size = lhs.actual_size();
    size_t divisor_size = rhs.actual_size();

    // CASE 0:
    if (dividend_size < divisor_size) {
        if (remainder != nullptr) {
            *remainder = lhs;
        }

        return uint_t(0);
    }

    // CASE 1:
    if (divisor_size == 1) {
        return divide(lhs, rhs[0], remainder);
    }

    // CASE 2:
    return d_divide(lhs, rhs, remainder);
}

static constexpr uint_t divide(const uint_t& lhs, const digit_t& rhs, uint_t* remainder = nullptr) {
```

```
    uint_t result;
    double_digit_t part = 0;

    for (size_t i = c_digit_number; i > 0; --i) {
        part = (part << (c_digit_size)) + static_cast<double_digit_t>(lhs[i - 1]);

        if (part < rhs) {
            continue;
        }

        result[i - 1] = static_cast<digit_t>(part / rhs);
        part %= rhs;
    }

    if (remainder != nullptr) {
        *remainder = uint_t(static_cast<digit_t>(part));
    }

    return result;
}

static constexpr uint_t d_divide(const uint_t& lhs, const uint_t& rhs, uint_t* remainder = nullptr) {
    size_t dividend_size = lhs.actual_size();
    size_t divisor_size = rhs.actual_size();

    uint_t<c_bits + c_digit_size> dividend(lhs);
    uint_t divisor(rhs);
    uint_t quotient;

    size_t shift_size = 0;
    digit_t divisor_head = divisor[divisor_size - 1];
    static constexpr double_digit_t c_HalfBlock = static_cast<double_digit_t>(1)
                                                  << (c_digit_size - 1);

    while (divisor_head < c_HalfBlock) {
        ++shift_size;
        divisor_head <<= 1;
    }

    dividend <<= shift_size;
    divisor <<= shift_size;

    double_digit_t divisor_ = divisor[divisor_size - 1];
    static constexpr double_digit_t c_Block = static_cast<double_digit_t>(1) << c_digit_size;

    for (size_t i = dividend_size - divisor_size + 1; i > 0; --i) {
        double_digit_t part =
            (static_cast<double_digit_t>(dividend[i + divisor_size - 1]) << c_digit_size)
            + static_cast<double_digit_t>(dividend[i + divisor_size - 2]);
        double_digit_t quotient_temp = part / divisor_;
        part %= divisor_;

        if (quotient_temp == c_Block) {
            --quotient_temp;
            part += divisor_;
        }

        while (part < c_Block
               && (quotient_temp * divisor[divisor_size - 2]
                   > (part << c_digit_size) + dividend[i + divisor_size - 3])) {
            --quotient_temp;
            part += divisor_;
        }

        int64_t carry = 0;
        int64_t widedigit = 0;

        for (size_t j = 0; j < divisor_size; ++j) {
            double_digit_t product =
                static_cast<digit_t>(quotient_temp) * static_cast<double_digit_t>(divisor[j]);
            widedigit = (static_cast<int64_t>(dividend[i + j - 1]) + carry) - (product & UINT32_MAX);
```

```
            dividend[i + j - 1] = static_cast<digit_t>(widedigit);
            carry = (widedigit >> c_digit_size) - static_cast<double_digit_t>(product >> c_digit_size);
        }

        widedigit = static_cast<int64_t>(dividend[i + divisor_size - 1]) + carry;
        dividend[i + divisor_size - 1] = static_cast<digit_t>(widedigit);

        quotient[i - 1] = static_cast<digit_t>(quotient_temp);

        if (widedigit < 0) {
            --quotient[i - 1];
            widedigit = 0;

            for (size_t j = 0; j < divisor_size; ++j) {
                widedigit += static_cast<double_digit_t>(dividend[i + j - 1]) + divisor[j];
                dividend[i + j - 1] = static_cast<digit_t>(widedigit);
                widedigit >>= 32;
            }
        }
    }

    if (remainder != nullptr) {
        *remainder = uint_t(0);

        for (size_t i = 0; i < divisor_size - 1; ++i) {
            (*remainder)[i] =
                (dividend[i] >> shift_size)
                | (static_cast<double_digit_t>(dividend[i + 1]) << (c_digit_size - shift_size));
        }

        (*remainder)[divisor_size - 1] = dividend[divisor_size - 1] >> shift_size;
    }

    return quotient;
}
```

Теперь можем определить операторы деления и остатка:

```
friend constexpr uint_t operator/(const uint_t& lhs, const uint_t& rhs) {
    uint_t result = divide(lhs, rhs);
    uint_t less = result * rhs;
    uint_t greater = (result + 1) * rhs;
    if (less > lhs || greater <= lhs) {
        result = 0;
    }
    return result;
}

friend constexpr uint_t operator%(const uint_t& lhs, const uint_t& rhs) {
    uint_t remainder;
    divide(lhs, rhs, &remainder);
    return remainder;
}

constexpr uint_t& operator*=(const uint_t& other) {
    return *this = *this * other;
}

constexpr uint_t& operator/=(const uint_t& other) {
    return *this = *this / other;
}
```

Так как нам не нужны r-value при делении, то не пишем оптимизации на них.

- Битовые сдвиги: Нам поступает запрос на сдвиг на size_t shift бит влево или вправо. Для высокой производительности выполнение операции нужно разбить на два этапа: сдвиг цифр внутри числа, сдвиг битов внутри цифр:

```
constexpr uint_t& operator>>=(size_t shift_size) {
    size_t digit_shift = shift_size >> 5;
```

```

    if (digit_shift > 0) {
        for (size_t i = 0; i < c_digit_number; ++i) {
            if (i + digit_shift < c_digit_number) {
                m_digits[i] = m_digits[i + digit_shift];
            } else {
                m_digits[i] = 0;
            }
        }
    }

    shift_size %= c_digit_size;

    if (shift_size == 0) {
        return *this;
    }

    for (size_t i = 0; i + digit_shift < c_digit_number; ++i) {
        m_digits[i] >>= shift_size;

        if (i + 1 < c_digit_number) {
            m_digits[i] |= m_digits[i + 1] << (c_digit_size - shift_size);
        }
    }

    return *this;
}

constexpr uint_t& operator<<=(size_t shift_size) {
    size_t digit_shift = shift_size >> 5;

    if (digit_shift > 0) {
        for (size_t i = c_digit_number; i > 0; --i) {
            if (i > digit_shift) {
                m_digits[i - 1] = m_digits[i - digit_shift - 1];
            } else {
                m_digits[i - 1] = 0;
            }
        }
    }

    shift_size %= c_digit_size;

    if (shift_size == 0) {
        return *this;
    }

    for (size_t i = c_digit_number; i > digit_shift; --i) {
        m_digits[i - 1] <<= shift_size;

        if (i - 1 > 0) {
            m_digits[i - 1] |= m_digits[i - 2] >> (c_digit_size - shift_size);
        }
    }

    return *this;
}
```

В обоих методах в концах двигаем недостающие биты из соседней цифры, если она существует. Определяем внешние friend:

```
friend constexpr uint_t operator>>(const uint_t& lhs, const size_t& rhs) {
    uint_t result = lhs;
    return result >>= rhs;
}

friend constexpr uint_t operator>>(uint_t&& lhs, const size_t& rhs) {
    return lhs >>= rhs;
}

friend constexpr uint_t operator<<(const uint_t& lhs, const size_t& rhs) {
```

```
    uint_t result = lhs;
    return result <<= rhs;
}

friend constexpr uint_t operator<<(uint_t&& lhs, const size_t& rhs) {
    return lhs <<= rhs;
}
```

- Сравнение: Так как используются 20 плюсы, то можно определить оператор <=>, но мы не можем использовать = default, так как тогда сравнение будет с 0 индекса, а не с последнего:

```
friend constexpr std::strong_ordering operator<=>(const uint_t& lhs, const uint_t& rhs) {
    for (size_t i = c_digit_number; i > 0; --i) {
        if (lhs[i - 1] != rhs[i - 1]) {
            return lhs[i - 1] <=> rhs[i - 1];
        }
    }

    return std::strong_ordering::equal;
}
```

Так как мы не определили через default, нам придётся написать и оператор равенства, но он очевиден:

```
friend constexpr bool operator==(const uint_t& lhs, const uint_t& rhs) {
    return lhs.m_digits == rhs.m_digits;
}
```

- Битовые операции: Наше число является по сути большой последовательностью бит одного числа, поэтому битовые операции выполняются поэлементно:

```
constexpr uint_t& operator^=(const uint_t& other) {
    for (size_t i = 0; i < c_digit_number; ++i) {
        m_digits[i] ^= other[i];
    }

    return *this;
}

constexpr uint_t& operator|=(const uint_t& other) {
    for (size_t i = 0; i < c_digit_number; ++i) {
        m_digits[i] |= other[i];
    }

    return *this;
}

constexpr uint_t& operator&=(const uint_t& other) {
    for (size_t i = 0; i < c_digit_number; ++i) {
        m_digits[i] &= other[i];
    }

    return *this;
}
```

Определяем внешние friend:

```
friend constexpr uint_t operator^(const uint_t& lhs, const uint_t& rhs) {
    uint_t result = lhs;
    return result ^= rhs;
}

friend constexpr uint_t operator^(uint_t&& lhs, const uint_t& rhs) {
    return lhs ^= rhs;
}

friend constexpr uint_t operator^(const uint_t& lhs, uint_t&& rhs) {
    return rhs ^= lhs;
}

```

```
friend constexpr uint_t operator^(uint_t&& lhs, uint_t&& rhs) {
    return lhs ^= rhs;
}

friend constexpr uint_t operator|(const uint_t& lhs, const uint_t& rhs) {
    uint_t result = lhs;
    result |= rhs;
    return result;
}

friend constexpr uint_t operator|(uint_t&& lhs, const uint_t& rhs) {
    lhs |= rhs;
    return lhs;
}

friend constexpr uint_t operator|(const uint_t& lhs, uint_t&& rhs) {
    rhs |= lhs;
    return rhs;
}

friend constexpr uint_t operator|(uint_t&& lhs, uint_t&& rhs) {
    lhs |= rhs;
    return lhs;
}

friend constexpr uint_t operator&(const uint_t& lhs, const uint_t& rhs) {
    uint_t result = lhs;
    result &= rhs;
    return result;
}

friend constexpr uint_t operator&(uint_t&& lhs, const uint_t& rhs) {
    lhs &= rhs;
    return lhs;
}

friend constexpr uint_t operator&(const uint_t& lhs, uint_t&& rhs) {
    rhs &= lhs;
    return rhs;
}

friend constexpr uint_t operator&(uint_t&& lhs, uint_t&& rhs) {
    lhs &= rhs;
    return lhs;
}
```

- Унарные инкремент и декремент: Определим вспомогательные приватные методы для увеличения/уменьшения числа на 1, скопировав код у $+=/-=$ соответственно:

```
constexpr void increment() {
    for (size_t i = 0; i < c_digit_number; ++i) {
        m_digits[i] += 1;

        if (m_digits[i] != 0) {
            break;
        }
    }
}

constexpr void decrement() {
    for (size_t i = 0; i < c_digit_number; ++i) {
        digit_t temp = m_digits[i];
        m_digits[i] -= 1;

        if (temp >= m_digits[i]) {
            break;
        }
    }
}
```

То есть мы пытаемся прибавить/вычесть остаток, пока не найдём хотя бы одну цифру, которая не переполниться от этой операции.

Теперь сами унарные операции. В C++ они бывают двух видов: префиксные и постфиксные. Они разделяются типом int в аргументе:

```
[[nodiscard("Optimize unary operator usage")]]
constexpr uint_t
    operator++(int) {
    uint_t result = *this;
    increment();
    return result;
}

constexpr uint_t& operator++() {
    increment();
    return *this;
}

[[nodiscard("Optimize unary operator usage")]]
constexpr uint_t
    operator--(int) {
    uint_t result = *this;
    decrement();
    return result;
}

constexpr uint_t& operator--() {
    decrement();
    return *this;
}
```

Дали атрибуты nodiscard постфиксным операторам, чтобы пользователь эффиктивно использовал унарны операции.

- Конвертация в стандартные типы: Два варианта:

  1. Хотим обрезать наш тип до стандартных целочисленных типов. Тогда заполняем требуемый тип битами из m_digits:

```
template<typename T>
requires is_convertible_to<T, digit_t>
constexpr T convert_to() const {
    size_t shift_size = sizeof(T) * c_bits_in_byte;
    size_t digits_number = shift_size / c_digit_size;

    if (digits_number == 0) {
        return static_cast<T>(m_digits[0]);
    }

    T result = 0;

    for (size_t i = 0; i < c_digit_number && i < digits_number; ++i) {
        result |= static_cast<T>(m_digits[i]) << (i * c_digit_size);
    }

    return result;
}
```

  2. Хотим получить всё число. Так как ни один стандартный тип такого размера не поддерживает, то будем переводить наше число в строку обычным делением:

```
template<typename T>
constexpr T convert_to() const;

template<>
constexpr std::string convert_to() const {
    std::string result;
    uint_t clone_of_this = *this;

```

```
    do {
        uint_t remainder;
        clone_of_this = divide(clone_of_this, 10, &remainder);
        result.push_back(remainder.m_digits[0] + '0');
    } while (clone_of_this > 0);

    std::reverse(result.begin(), result.end());
    return result;
}
```

Заметим, что мы специально сделали шаблон, а потом его специализировали, чтобы можно было использовать два варианта одинаково:

```
uint_t a = ...;
size_t b = a.convert_to<size_t>();
std::string s = a.convert_to<std::string>();
```