LNUM

Long numbers
Edition 1.0
23 May 2020

В этом руководстве-туториале описывается, как устроена библиотека lnum, а также процесс её установки и использования.

Для понимания устройства данной библиотеки необходимы базовые познания в языке ассемблера, а так же знания языка C++.

Поддержка продолжается, планируются будущие обновления.

Глазачев Михаил Васильевич,
Новиков Дмитрий Алексеевич,
Панов Михаил Федорович

2020 год.

Содержание

1 Устройство и подключение

Будем использовать два рабочих файла lnum.h и lnum.cpp. В заголовочном определим структуру числа, его конструкторы, деструкторы, и объявим внутренние функции и перегруженные операторы. В файле lnum.cpp опередим перегруженные операторы и внутренние функции. Чтобы воспользоваться библиотекой, создаём новый проект в VS, и добавляем оба файла в проект. Далее в основном файле проекта подключаем нашу библиотеку:

```
#include "lnum.h"
using namespace lnum;
```

Здесь и далее длинные числа имеют тип number.

2 Загаловочный файл

Подключены следующие заголовочные файлы стандартной библиотеки C++:

```
#pragma once
#include <stdexcept>
#include <iostream>
#include <iomanip>
```

2.1 Препроцессорные директивы.

Библиотека адаптируется к размеру unsigned int на данной платформе. На данный момент библиотека поддерживается только на Visual C++.
WIDTH_INT - размер unsigned int в полубайтах.
TMP_INT_SIZE - размер unsigned int в байтах.
Выбор регистров происходит в зависимости от разрядности системы.

Для 16-битных систем:

```
#if UINT_MAX == 0xFFFF
#define WIDTH_INT 4
#define TMP_INT_SIZE 2
#define ax_r ax
#define bx_r bx
#define cx_r cx
#define dx_r dx
#define si_r si
#define di_r di
```

Для 32-битных систем:

```
#elif UINT_MAX == 0xFFFFFFFF
#define WIDTH_INT 8
#define TMP_INT_SIZE 4
#define ax_r eax
#define bx_r ebx
#define cx_r ecx
#define dx_r edx
#define si_r esi
#define di_r edi
```

Для 64-битных систем:

```
#elif UINT_MAX == 0xFFFFFFFFFFFFFFFF
#define WIDTH_INT 16
#define TMP_INT_SIZE 8
#define ax_r rax
#define bx_r rbx
#define cx_r rcx
#define dx_r rdx
#define si_r esi
#define di_r edi
```

Если ничего не подошло:

```
#else
#error "Unknown OS bit depth"
#endif
```

Ограничение только на Visual C++

```
#ifndef _MSC_VER
#error "Only for Visual C++"
#endif
```

2.2 Пространство имён и его устройство.

Определим пространство имён и главную структуру следующим образом:

```
namespace lnum {
        struct number_s final {
        ...Constructors
        ...Internal structure
        ......Memory management
        ......Destructor
        ...Declaring redefinition operators
        };
```

```
        typedef number_s number;
        The remaining redefinition operators

}
```

2.3 Конструкторы

Все методы, конструкторы и деструктор определены в файле lnum.cpp, а в lnum.h только объявлены, но для удобства и целостности картины представим их конечный вид здесь.

2.3.1 Пустой конструктор

Этот конструктор вызывается при объявлении объекта класса и прямой инициализации через данный конструктор. Он выделяет память для одного unsigned int и помещает туда 0.

```
number_s() {};
```

2.3.2 Конструктор от стандартного натурального числа

Если при объявлении объекта класса ему задаётся значение типа unsigned long long и типов, которые могут в него преобразовываться, то вызывается данный конструктор, который обрабатывает число, выделяет необходимую память и заполняет массив нужными данными.

```
number_s(const unsigned long long& num) : number_num(sizeof
    (num) / sizeof(unsigned int)) {
        unsigned int i = 0;
        if (!num) this->number_num.number[0] = 0;
        for (; num >> sizeof(unsigned int) * 8 * i; ++i) {
                unsigned long long t1 = num >> sizeof(
                    unsigned int) * 8 * i,
                        t2 = (num >> sizeof(unsigned int) *
                             8 * (i + 1)) << sizeof(
                            unsigned int) * 8;
                this->number_num[i] = t1 - t2;
        }
        if (!i) this->number_num.resize(1);
        else if (i < this->number_num.size) this->
            number_num.resize(i);
}
```

2.3.3 Конструктор копирования

Если при объявлении объекта класса ему присваивается объект этого же класса, то конструктор копирования создаст полную копию длинного числа (исключением будет лишь адрес в памяти).

```
number_s(const number_s& num) : number_num(num.number_num.
    size) {
        unsigned int *ths = this->number_num.number, *nm =
            num.number_num.number,
                *nmend = num.number_num.number + num.
                    number_num.size;
        __asm {
                mov di_r, nm
                mov si_r, ths
                mov cx_r, nmend
                copyloop :
                mov ax_r, [di_r]
                        mov[si_r], ax_r
                        add si_r, TMP_INT_SIZE
                        add di_r, TMP_INT_SIZE
                        cmp di_r, cx_r
                        jl copyloop
        }
}
```

2.4 Внутренняя структура и управление памятью.

Внутренняя структура выполнена в следующем виде:

```
struct number_vector {
/fields and methods/

/memory allocation and reallocation here/

/Destructor/
} number_num;
```

2.4.1 Внутренние поля и вспомогательные методы

```
mutable size_t size;
mutable unsigned int * number;
inline unsigned int& operator[](size_t n){
    return number[n];}
inline unsigned int operator[](size_t n) const {
    return number[n];
}
```

```
void resize(size_t new_size) {
        number = (unsigned int *)realloc(number, new_size *
            sizeof(unsigned int));
        if (number == nullptr) throw std::bad_alloc();
        if (this->size < new_size)
                for (size_t i = this->size; i < new_size;
                    ++i)
                        this->number[i] = 0;
        this->size = new_size;
}
```

Здесь всё интуитивно понятно по названиям, так что не будем заострять особое внимание.

2.4.2 Конструкторы и memory management

Теперь разберёмся на что "ссылаются" конструкторы из главной структуры.
Пустой конструктор:

```
number_vector() : size(1) {
        if ((this->number = (unsigned int*)malloc(sizeof(
            unsigned int))) == nullptr)
                throw std::bad_alloc();
        this->number[0] = 0;
}
```

Конуструкторы у которых в списке инициализации указан не единичный размер.

```
number_vector(unsigned int n) : size(n) {
        if ((this->number = (unsigned int*)malloc(n *
            sizeof(unsigned int))) == nullptr)
                throw std::bad_alloc();
        unsigned int* num_num = this->number, *num_num_end
            = this->number + n;
        __asm {
        mov si_r, num_num
        mov cx_r, num_num_end
        zeroloop :
        mov[si_r], 0
                add si_r, TMP_INT_SIZE
                cmp si_r, cx_r
                jl zeroloop
        }
}
```

В обоих случаях мы выделяем достаточно памяти для массива и заполняем элементы нулями.

2.4.3 Деструктор

```
~number_vector() { free(this->number); }
```

Возвращаем память обратно в кучу.

2.5 Внутренние методы и объявление перегруженных операторов

```
size_t size() const { return this->number_num.size; }
const unsigned int& operator[](size_t n) const {
 return this->number_num[n];
}

number_s& operator+=(const number_s&);
number_s& operator-=(const number_s&);
number_s& operator*=(const number_s&);
number_s& operator/=(const number_s&);
number_s& operator%=(const number_s&);
number_s& operator&=(const number_s&);
number_s& operator|=(const number_s&);
number_s& operator>>=(const number_s&);
number_s& operator<<=(const number_s&);
number_s& operator=(const number_s&);


const bool operator==(const number_s&) const;
const bool operator!=(const number_s&) const;
const bool operator>(const number_s&) const;
const bool operator<(const number_s&) const;
const bool operator<=(const number_s&) const;
const bool operator>=(const number_s&) const;

const number_s operator-() const;
const number_s& operator+() const;
const number_s operator~() const;
const bool operator!() const;
number_s& operator--();
const number_s operator--(int);
number_s& operator++();
const number_s operator++(int);

explicit operator bool() const { return !this->is_zero(); }
```

А следующие методы и оператор объявим с модификатором доступа protected: (Данные методы используются в файле lnum.cpp для перегрузки операторов)

```
friend inline number_s& __additing_operator_s(number_s&,
   const number_s&);
friend inline number_s& __substruct_operator_s(number_s&,
   const number_s&);
friend inline number_s& __multiple_operator_s(number_s&,
   const number_s&);
```

```
friend inline number_s& __division_operator_s(number_s&,
    const number_s&);
friend inline const bool __equal_operator_s(const number_s&
     ths, const number_s& num);
friend inline const bool __less_equal_operator_s(const
    number_s& ths, const number_s& num);
friend inline const bool __greater_equal_operator_s(const
    number_s& ths, const number_s& num);
friend inline number_s& __bit_and_operator_s(number_s& ths,
     const number_s& num);
friend inline number_s& __bit_or_operator_s(number_s& ths,
    const number_s& num);
friend inline number_s& __mod_operator_s(number_s& ths,
    const number_s& num);
friend inline number_s& __shift_right_operator_s(number_s&
    ths, const number_s& num);
friend inline number_s& __shift_left_operator_s(number_s&
    ths, const number_s& num);
inline bool is_zero() const { return this->number_num.size
    == 1 && this->number_num[0] == 0; }
friend std::ostream& operator<<(std::ostream&, const
    number_s& num);
```

И отдельно от пространства имён объявим следующие операторы:

```
const lnum::number_s operator+(const lnum::number_s& ths,
    const lnum::number_s& num);
const lnum::number_s operator-(const lnum::number_s& ths,
    const lnum::number_s& num);
const lnum::number_s operator*(const lnum::number_s& ths,
    const lnum::number_s& num);
const lnum::number_s operator/(const lnum::number_s& ths,
    const lnum::number_s& num);
const lnum::number_s operator%(const lnum::number_s& ths,
    const lnum::number_s& num);
const lnum::number_s operator&(const lnum::number_s& ths,
    const lnum::number_s& num);
const lnum::number_s operator|(const lnum::number_s& ths,
    const lnum::number_s& num);
const lnum::number_s operator>>(const lnum::number_s& ths,
    const lnum::number_s& num);
const lnum::number_s operator<<(const lnum::number_s& ths,
    const lnum::number_s& num);
```

3 Файл lnum.cpp

Устройство этого файла гораздо проще предыдущего: пространство имён lnum и 9 перегруженных операторов вне его:

```
const lnum::number_s operator+(const lnum::number_s& ths,
    const lnum::number_s& num) { lnum::number_s n = ths;
    return n += num; }
const lnum::number_s operator-(const lnum::number_s& ths,
    const lnum::number_s& num) { lnum::number_s n = ths;
    return n -= num; }
const lnum::number_s operator*(const lnum::number_s& ths,
    const lnum::number_s& num) { lnum::number_s n = ths;
    return n *= num; }
const lnum::number_s operator/(const lnum::number_s& ths,
    const lnum::number_s& num) { lnum::number_s n = ths;
    return n /= num; }
const lnum::number_s operator%(const lnum::number_s& ths,
    const lnum::number_s& num) { lnum::number_s n = ths;
    return n %= num; }
const lnum::number_s operator&(const lnum::number_s& ths,
    const lnum::number_s& num) { lnum::number_s n = ths;
    return n &= num; }
const lnum::number_s operator|(const lnum::number_s& ths,
    const lnum::number_s& num) { lnum::number_s n = ths;
    return n |= num; }
const lnum::number_s operator>>(const lnum::number_s& ths,
    const lnum::number_s& num) { lnum::number_s n = ths;
    return n >>= num; }
const lnum::number_s operator<<(const lnum::number_s& ths,
    const lnum::number_s& num) { lnum::number_s n = ths;
    return n <<= num; }
```

Это стандартные арифметические и побитовые операции. При чём определены они через сокращённые операторы, а не наоборот.
Для удобства будем перегружать эти операторы, используя функции следующим образом:

```
number_s& number_s::operator+=(const number_s& num) {
    return __additing_operator_s(*this, num); }
number_s& number_s::operator-=(const number_s& num) {
    return __substruct_operator_s(*this, num); }
number_s& number_s::operator*=(const number_s& num) {
    return __multiple_operator_s(*this, num); }
number_s& number_s::operator/=(const number_s& num) {
    return __division_operator_s(*this, num); }
number_s& number_s::operator%=(const number_s& num) {
    return __mod_operator_s(*this, num); }
number_s& number_s::operator&=(const number_s& num) {
    return __bit_and_operator_s(*this, num); }
```

```
number_s& number_s::operator|=(const number_s& num) {
    return __bit_or_operator_s(*this, num); }
number_s& number_s::operator>>=(const number_s& num) {
    return __shift_right_operator_s(*this, num); }
number_s& number_s::operator<<=(const number_s& num) {
    return __shift_left_operator_s(*this, num); }
```

3.1 Функции перегружающие арифметические и битовые операции

3.1.1 Функция перегружающая оператор +=

Для начала сравниваем размеры чисел, если первое меньше второго, то подгоняем размер первого числа ко второму, затем запоминаем адреса начала и конца обоих чисел. Далее реализуем ассемблерную вставку: Помещаем адреса в регистры, далее реализуем обычный алгоритм столбика, при этом не забываем учитывать переполнение. Для перехода к следующему разряду увеличиваем адреса. Когда достигаем конца правого числа (для этого сравниваем текущий адрес с адресом конца), цикл прекращается. При возникновении переполнения в первом цикле запускается второй, который работает так же, как и предыдущий, но прибавляет фиксированную единицу. В случае возникновения переполнения и во втором цикле, расширяем число и помещаем единицу на новое место. Исходный код:

```
inline number_s& __additing_operator_s(number_s& ths, const
    number_s& num) {
        unsigned int t = 0;
        if (ths.number_num.size < num.number_num.size) ths.
            number_num.resize(num.number_num.size);
        unsigned int *ths_num = ths.number_num.number, *
            num_num = num.number_num.number,
                 *ths_num_end = ths_num + ths.number_num.
                    size, *num_num_end = num_num + num.
                    number_num.size;
        __asm {
                 mov di_r, num_num
                 mov si_r, ths_num
                 mov cx_r, num_num_end
                 mov bx_r, ths_num_end
                 xor dx_r, dx_r
                 loop_add :
                 mov ax_r, [di_r]
                         add[si_r], ax_r
                         jc g1
                         add[si_r], dx_r
```

```
                        jc nextloop_add
                        mov dx_r, 0
                        jmp nextloop_add
                        g1 :
                add[si_r], dx_r
                        mov dx_r, 1
                        nextloop_add :
                        add si_r, TMP_INT_SIZE
                        add di_r, TMP_INT_SIZE
                        cmp di_r, cx_r
                        jl loop_add

                        cmp dx_r, 0
                        je end_add
                        loop_add_1 :
                cmp si_r, bx_r
                        je end_add
                        add[si_r], dx_r
                        jnc end_loop_add_1
                        add si_r, TMP_INT_SIZE
                        jmp loop_add_1
                        end_loop_add_1 :
                mov dx_r, 0


                        end_add :
                        mov t, dx_r
        }
        if (t) {
                ths.number_num.resize(ths.number_num.size +
                    1);
                ths.number_num[ths.number_num.size - 1] =
                   1;
        }
        return ths;
}
```

3.1.2 Функция перегружающая оператор -=

Исходный код:

```
inline number_s& __substruct_operator_s(number_s& ths,
    const number_s& num) {
        if (ths < num)
                throw std::invalid_argument("Too big number
                    are subtracted.");
        int tmp = 0, new_size;
        if (ths.number_num.size < num.number_num.size) ths.
            number_num.resize(num.number_num.size);
        unsigned int *ths_num = ths.number_num.number, *
            num_num = num.number_num.number;
        unsigned int *ths_num_end = ths_num + ths.
            number_num.size,
                *num_num_end = num_num + num.number_num.
                    size, sz = ths.number_num.size;
        _asm {
                mov di_r, num_num
                mov si_r, ths_num
                mov cx_r, num_num_end
                xor dx_r, dx_r

                subloop :
                mov ax_r, [si_r]
                        mov bx_r, [di_r]
                        cmp ax_r, bx_r
                        jb cf1
                        sub ax_r, bx_r
                        cmp ax_r, dx_r
                        jb cf2
                        sub ax_r, dx_r
                        xor dx_r, dx_r
                        jmp nextsubloop

                        cf1 :
                sub bx_r, ax_r
                        mov ax_r, UINT_MAX
                        sub ax_r, bx_r
                        inc ax_r
                        sub ax_r, dx_r
                        mov dx_r, 1
                        jmp nextsubloop

                        cf2 :
                mov ax_r, UINT_MAX

                        nextsubloop :
                mov[si_r], ax_r
```

```
                        add si_r, TMP_INT_SIZE
                        add di_r, TMP_INT_SIZE
                        cmp di_r, cx_r
                        jl subloop

                        mov si_r, ths_num_end
                        mov bx_r, sz
                        mov cx_r, ths_num
                        xor dx_r, dx_r
                        endloop :
                sub si_r, TMP_INT_SIZE
                        cmp si_r, cx_r
                        je fromthis
                        mov ax_r, [si_r]
                        cmp ax_r, 0
                        jne fromthis
                        dec bx_r
                        jmp endloop


                        fromthis :
                mov new_size, bx_r
        }
        if (new_size != sz) ths.number_num.resize(new_size)
            ;
        return ths;
}
```

3.1.3 Функция перегружающая оператор *=

Исходный код:

```
inline number_s& __multiple_operator_s(number_s& ths, const
    number_s& num) {
        if(ths.is_zero() || num.is_zero()) return ths = 0;
        number_s tmp = ths;
        ths = 0;
        ths.number_num.resize(num.number_num.size + tmp.
           number_num.size);
        unsigned int *ths_num = ths.number_num.number,
                     *num_num = num.number_num.number,
                     *tmp_num = tmp.number_num.number,
                        tmp_cf = 0;
        for (unsigned int i = 0; i < tmp.number_num.size;
           ++i) {
                for (unsigned int j = 0; j < num.number_num
                   .size; ++j) {
                        __asm {
                                mov si_r, tmp_num
                                mov di_r, num_num

                                mov bx_r, TMP_INT_SIZE
                                mov ax_r, j
                                mul bx_r
                                push ax_r
                                add di_r, ax_r
                                mov ax_r, i
                                mul bx_r
                                push ax_r
                                add si_r, ax_r

                                mov ax_r, [si_r]
                                mov dx_r, [di_r]
                                mul dx_r
                                add ax_r, tmp_cf
                                jnc nxt
                                inc dx_r
                                nxt:

                                mov si_r, ths_num
                                pop bx_r
                                pop cx_r
                                add si_r, bx_r
                                add si_r, cx_r

                                add [si_r], ax_r
                                jnc end
                                inc dx_r
```

```
                                    end:
                                    mov tmp_cf, dx_r
                            }
                    }
                    ths.number_num.number[i + num.number_num.
                        size] += tmp_cf;
                    tmp_cf = 0;
            }
            if (!ths.number_num.number[ths.number_num.size -
                1]) ths.number_num.resize(ths.number_num.size -
                 1);
            return ths;
}
```

3.1.4 Функция перегружающая оператор /=

Исходный код:

```
inline number_s& __division_operator_s(number_s& ths, const
    number_s& num) {
        if (num.is_zero()) throw std::invalid_argument("
            Division by zero");
        if (num > ths) { ths.number_num.resize(1); ths.
            number_num[0] = 0; return ths; }
        unsigned int new_size = ths.number_num.size - num.
            number_num.size + 1;
        number_s n; n.number_num.resize(new_size);
        for (size_t i = new_size - 1; true; --i) {
                for (unsigned int k = sizeof(unsigned int)
                    * 8 - 1, t = 1U << k; true; t = 1U <<
                    (--k)) {
                        n.number_num.number[i] += t;
                        if (ths < (n*num)) n.number_num.
                            number[i] -= t;
                        if (!k) break;
                }
                if (!i) break;
                if (i == new_size - 1 && !n.number_num.
                    number[i]) n.number_num.resize(i);
        }
        return ths = n;
}
```

3.1.5 Функция перегружающая оператор %=

Исходный код:

```
inline number_s& __mod_operator_s(number_s& ths,
const number_s& num) {
        return ths -= (ths / num)*num;
}
```

3.1.6 Функция перегружающая оператор &=

Исходный код:

```
inline number_s& __bit_and_operator_s(number_s& ths, const
    number_s& num) {
        unsigned int* ths_num = ths.number_num.number, *
            num_num = num.number_num.number,
                sz = ths.number_num.size, *ths_end =
                    ths_num + sz, new_size;
        __asm {
                mov si_r, ths_num
                mov di_r, num_num
                mov cx_r, ths_end
                loopad :
                mov ax_r, [di_r]
                        and [si_r], ax_r
                        add si_r, TMP_INT_SIZE
                        add di_r, TMP_INT_SIZE
                        cmp si_r, cx_r
                        jb loopad


                        mov cx_r, ths_num
                        xor dx_r, dx_r
                        zeroloop :
                sub si_r, TMP_INT_SIZE
                        cmp[si_r], 0
                        jne end
                        cmp si_r, cx_r
                        jna end
                        inc dx_r
                        jmp zeroloop
                        end :
                mov new_size, dx_r
        }
        if (new_size) ths.number_num.resize(num.number_num.
            size - new_size);
        else if (sz != num.number_num.size) ths.number_num.
            resize(num.number_num.size);
        return ths;
}
```

3.1.7 Функция перегружающая оператор |=

Исходный код:

```
inline number_s& __bit_or_operator_s(number_s& ths, const
    number_s& num) {
        size_t v = ths.number_num.size, vn = num.number_num
            .size;
        if (v < vn) ths.number_num.resize(v = vn);
        unsigned int* ths_num = ths.number_num.number,
                *num_num = num.number_num.number, *num_end
                    = num_num + vn;
        __asm {
                mov si_r, ths_num
                mov di_r, num_num
                mov cx_r, num_end
                lp :
                mov ax_r, [di_r]
                        or [si_r], ax_r
                        add si_r, TMP_INT_SIZE
                        add di_r, TMP_INT_SIZE
                        cmp di_r, cx_r
                        jb lp
        }
        return ths;
}
```

3.1.8 Функция перегружающая оператор ≫=

Исходный код:

```
inline number_s& __shift_right_operator_s(number_s& ths,
    const number_s& num) {
     if (num >= ths.number_num.size*TMP_INT_SIZE * 8) {
         ths.number_num.resize(1);
         ths.number_num.number[0] = 0;
         return ths;
     }
     else if (ths.is_zero()) return ths;
     unsigned int shl = (num / (TMP_INT_SIZE * 8)).
        number_num.number[0];
     if (shl) {
     for (unsigned int* i = ths.number_num.number;
         i + shl < ths.number_num.number + ths.number_num.
            size; ++i)
             *i = *(i + shl);
     ths.number_num.resize(ths.number_num.size - shl);
     }
     shl = (num % (TMP_INT_SIZE * 8)).number_num.number[0];
         if (shl) {
         unsigned int prev = 0, next = 0;
         for (unsigned int* i = ths.number_num.number + ths.
            number_num.size;
         --i >= ths.number_num.number;) {
             next = *i % (1 << shl);
             *i >>= shl;
             *i += (prev << (TMP_INT_SIZE * 8 - shl));
             prev = next;
         }
     }
     if (!ths.number_num.number[ths.number_num.size - 1])
         ths.number_num.resize(ths.number_num.size - 1);
     return ths;
}
```

3.1.9 Функция перегружающая оператор ≪=

Исходный код:

```
inline number_s& __shift_left_operator_s(number_s& ths,
    const number_s& num) {
    if (ths.is_zero()) return ths;
    unsigned int shl = (num / (TMP_INT_SIZE * 8)).
        number_num.number[0];
    ths.number_num.resize(ths.number_num.size + shl + 1);
    if (shl) {
        for (unsigned int* i = ths.number_num.number + ths.
            number_num.size - 1;
             --i - shl >= ths.number_num.number;) {
            *i = *(i - shl);
        }
        for (unsigned int* i = ths.number_num.number + shl;
             --i >= ths.number_num.number;) {
            *i = 0;
        }
    }
    unsigned int shlm = (num % (TMP_INT_SIZE * 8)).
        number_num.number[0];
    if (shlm) {
        unsigned int prev = 0, next = 0;
        for (unsigned int* i = ths.number_num.number + shl;
             i < ths.number_num.number + ths.number_num.size
                 ; ++i) {
                next = *i / (1 << (TMP_INT_SIZE * 8 - shlm)
                    );
                *i <<= shlm;
                *i += prev;
                prev = next;
        }
    }
    if (!ths.number_num.number[ths.number_num.size - 1])
        ths.number_num.resize(ths.number_num.size - 1);
    return ths;
}
```

3.2 Функции реализующие операторы сравнения

Точно так же перегрузим операторы с помощью вспомогательных функций:

```
const bool number_s::operator==(const number_s& num) const
   { return __equal_operator_s(*this, num); }
const bool number_s::operator!=(const number_s& num) const
   { return !__equal_operator_s(*this, num); }
const bool number_s::operator> (const number_s& num) const
   { return !__less_equal_operator_s(*this, num); }
const bool number_s::operator< (const number_s& num) const
   { return !__greater_equal_operator_s(*this, num); }
const bool number_s::operator<=(const number_s& num) const
   { return __less_equal_operator_s(*this, num); }
const bool number_s::operator>=(const number_s& num) const
   { return __greater_equal_operator_s(*this, num); }
```

3.2.1 Функция перегружающая оператор == и !=

Исходный код:

```
inline const bool __equal_operator_s(const number_s& ths,
   const number_s& num) {
        size_t n = ths.number_num.size, nv = num.number_num
           .size;
        if (n != nv) return false;
        unsigned int *ths_num = ths.number_num.number, *
           num_num = num.number_num.number;
        unsigned int *ths_num_end = ths_num + n,
                 *num_num_end = num_num + nv;
        bool t = 0;
        _asm {
                 mov di_r, num_num_end
                 mov si_r, ths_num_end
                 mov cx_r, ths_num
                 equalloop :
                 sub di_r, TMP_INT_SIZE
                         sub si_r, TMP_INT_SIZE
                         mov ax_r, [si_r]
                         mov bx_r, [di_r]
                         cmp ax_r, bx_r
                         jne end
                         cmp si_r, cx_r
                         ja equalloop
                         mov t, 1
                         end:
        }
        return t;}
```

3.2.2 Функция перегружающая оператор $>$ и $<=$

Исходный код:

```
inline const bool __less_equal_operator_s(const number_s&
    ths, const number_s& num) {
        size_t n = ths.number_num.size, nv = num.number_num
            .size;
        if (n > nv) return false;
        else if (n < nv) return true;
        unsigned int *ths_num = ths.number_num.number, *
            num_num = num.number_num.number;
        unsigned int *ths_num_end = ths_num + n,
                *num_num_end = num_num + nv;
        bool t = 0;
        _asm {
                mov di_r, num_num_end
                mov si_r, ths_num_end
                mov cx_r, ths_num
                equalloop :
                sub di_r, TMP_INT_SIZE
                sub si_r, TMP_INT_SIZE

                mov ax_r, [si_r]
                mov bx_r, [di_r]
                cmp ax_r, bx_r
                ja end
                cmp ax_r, bx_r
                jne next
                cmp si_r, cx_r
                jae equalloop
                next :
                mov t, 1
                end :
        }
        return t;
}
```

3.2.3 Функция перегружающая оператор < и >=

Исходный код:

```
inline const bool __greater_equal_operator_s(const number_s
    & ths, const number_s& num) {
        size_t n = ths.number_num.size, nv = num.number_num
            .size;
        if (n > nv) return true;
        else if (n < nv) return false;
        unsigned int *ths_num = ths.number_num.number, *
            num_num = num.number_num.number;
        unsigned int *ths_num_end = ths_num + n,
                *num_num_end = num_num + nv;
        bool t = 0;
        _asm {
                mov di_r, num_num_end
                mov si_r, ths_num_end
                mov cx_r, ths_num
                equalloop :
                sub di_r, TMP_INT_SIZE
                sub si_r, TMP_INT_SIZE

                mov ax_r, [si_r]
                mov bx_r, [di_r]
                cmp ax_r, bx_r
                jb end
                cmp ax_r, bx_r
                jne next
                cmp si_r, cx_r
                jae equalloop
                next :
                mov t, 1
                end :
        }
        return t;
}
```

3.3 Унарные операторы

3.3.1 Унарный плюс

Исходный код:

```
const number_s& number_s::operator+() const { return *this;
    }
```

3.3.2 Оператор отрицания

Исходный код:

```
inline const bool number_s::operator!() const { return this
   ->is_zero(); }
```

3.3.3 Префиксный декремент

Исходный код:

```
number_s& number_s::operator--() {
        if (this->is_zero())
                throw std::invalid_argument("Too big number
                    are subtracted.");
        unsigned int sz = this->number_num.size, *ths =
           this->number_num.number,
                *ths_end = this->number_num.number + sz,
                   new_size;

        __asm {
                mov si_r, ths
                pos :
                mov ax_r, [si_r]
                        sub ax_r, 1
                        jnc endpositive
                        mov[si_r], ax_r
                        add si_r, TMP_INT_SIZE
                        jmp pos


                        endpositive :
                mov[si_r], ax_r

                        mov si_r, ths_end
                        mov bx_r, sz
                        mov cx_r, ths
                        xor dx_r, dx_r
                        endloop :
                sub si_r, TMP_INT_SIZE
                        cmp si_r, cx_r
                        je fromthis
                        mov ax_r, [si_r]
                        cmp ax_r, 0
                        jne fromthis
                        dec bx_r
                        jmp endloop

                        fromthis :
                mov new_size, bx_r
        }
        if (new_size != sz) this->number_num.resize(
           new_size);
        return *this;
}
```

3.3.4 Постфиксный декремент

Исходный код:

```
const number_s number_s::operator--(int) { number_s num = *
    this; --*this;  return num; }
```

3.3.5 Префиксный инкремент

Исходный код:

```
number_s& number_s::operator++() {
        unsigned int t = 0, sz = this->number_num.size, *
            ths = this->number_num.number,
                 *ths_end = this->number_num.number + sz;
        __asm {
                mov si_r, ths
                mov di_r, ths_end
                pos :
                mov ax_r, [si_r]
                        add ax_r, 1
                        jnc endpos
                        mov[si_r], ax_r
                        add si_r, TMP_INT_SIZE
                        cmp si_r, di_r
                        jne pos
                        mov t, 1
                        jmp end
                        endpos :
                mov[si_r], ax_r
                        end :
        }
        if (t) { this->number_num.resize(sz + 1); this->
            number_num[sz] = 1; }
        return *this;
}
```

3.3.6 Постфиксный инкремент

Исходный код:

```
const number_s number_s::operator++(int) { number_s num = *
    this; ++*this; return num; }
```

3.4 Оператор присваивания

Исходный код:

```
number_s& number_s::operator=(const number&num) {
        this->number_num.resize(num.number_num.size);
        unsigned int *this_num = this->number_num.number,
                *this_end = this_num + this->number_num.
                    size,
                *num_num = num.number_num.number;
        __asm {
                mov si_r, this_num
                mov di_r, num_num
                mov cx_r, this_end

                g :
                mov ax_r, [di_r]
                mov[si_r], ax_r

                add si_r, TMP_INT_SIZE
                add di_r, TMP_INT_SIZE
                cmp si_r, cx_r
                jb g
        }
        return *this;
}
```

3.5 Оператор вывода

Введём следующие обозначения:
streamlink - ссылка на поток;
num - само длинное число;
Если размер числа равен 1, то выводим его в шестнадцатиричном формате с помощью команды (streamlink ≪ std::hex ≪ num.number_num[0];) (number_num[n] оператор для обращения к элементу с n-ым индексом) и возвращаем ссылку на поток. Если размер числа больше единицы: делаем правильный поэлементный вывод с учётом длины числа представленном в 16-ом формате (чтобы не выводить лишние нули в начале). Временно меняем заполнитель потока на символ нуля, выводим оставшиеся элементы в таком же виде, как и в случае единичного размера.

```
std::ostream& operator<<(std::ostream& str, const number_s&
    num) {
    if (num.number_num.size != 1) {
        unsigned int t = num.number_num[num.number_num.size
            - 1], c = WIDTH_INT - 1;
        while (!(t >> (c * 4))) --c;
        while (true) {
            str << std::hex << (t >> c * 4) % 0x10;
            if (!c--) break;
        }
        char fl = str.fill();
        str.fill('0');
        for (unsigned int i = num.number_num.size - 2; true
            ; --i) {
            str << std::hex << std::setw(WIDTH_INT) << num.
                number_num[i];
            if (!i) break;
        }
        str.fill(fl);
        return str;
    }
    else str << std::hex << num.number_num[0];
}
```

3.6 Таблица операторов

Оператор	Реализация	Сложность алгоритма
+=	Ассемблер	O(n)
-=	Ассемблер	O(n)
*=	Ассемблер	O(n(n+m))
/=	C++	O($n^2(n+m)$)
%=	C++	O($n^2(n+m)$)
<	C++	O(n)
>	C++	O(n)
<=	Ассемблер	O(n)
>=	Ассемблер	O(n)
==	Ассемблер	O(n)
!=	C++	O(n)
ostream ≪	C++	O(n)
≪=	C++	O(n)
≫=	C++	O(n)
++	Ассемблер	O(n)
- -	Ассемблер	O(n)
static_cast<bool>	C++	O(1)
=	Ассемблер	O(m)
+	Ассемблер	O(n)
-	Ассемблер	O(n)
*	Ассемблер	O(n(n+m))
/	C++	O($n^2(n+m)$)
%	C++	O($n^2(n+m)$)
≪	C++	O(n)
≫	C++	O(n)

Где n - размер левого числа, m - размер правого числа, c - некоторая константа.