Домашние задание

Как известно, в языках Си и C++ не имеется типов данных, способных вмещать в себя целые числа неограниченной длины (тип `long long` вмещает в себя всего лишь числа от -2^{63} до $2^{63}-1$, а хотелось бы производить операции с числами длиной хотя бы в 100000 тысяч десятичных цифр). Язык Си достаточно мощен дл того, чтобы расширить его таким типом данных. Вам предлагается реализовать **арифметику высокой точности**, используя конструкции языка Си. Нам не удастся так же легко оперировать с этими числами, как это получилось бы в C++, но что-то сделать можно.

Вам представляется *интерфейс* набора библиотечных функций, основанный на типе данных `struct bn_s`, который мы для краткости будем называть `bn`, реализующий необходимое множество операций. Так как в языке Си нет *перегрузки* типов операций, как в C++ и мы не можем писать `bn a,b,c; a = b + c;`, то нам придётся использовать набор функций как для создания объектов типа «большое число», так и для операций над этими объектами. Набор необходимых функций приведён ниже.

```
#pragma once
// Файл bn.h
struct bn_s;
typedef struct bn_s bn;

enum bn_codes {
   BN_OK, BN_NULL_OBJECT, BN_NO_MEMORY, BN_DIVIDE_BY_ZERO
};

bn *bn_new();   // Создать новое BN
bn *bn_init(bn const *orig); // Создать копию существующего BN

// Инициализировать значение BN десятичным представлением строки
int bn_init_string(bn *t, const char *init_string);

// Инициализировать значение BN представлением строки
// в системе счисления radix
int bn_init_string_radix(bn *t, const char *init_string, int radix);

// Инициализировать значение BN заданным целым числом
int bn_init_int(bn *t, int init_int);

// Уничтожить BN (освободить память)
int bn_delete(bn *t);
```

```
// Операции, аналогичные +=, -=, *=, /=, %=
int bn_add_to(bn *t, bn const *right);
int bn_sub_to(bn *t, bn const *right);
int bn_mul_to(bn *t, bn const *right);
int bn_div_to(bn *t, bn const *right);
int bn_mod_to(bn *t, bn const *right);

// Возвести число в степень degree
int bn_pow_to(bn *t, int degree);

// Извлечь корень степени reciprocal из BN (бонусная функция)
int bn_root_to(bn *t, int reciprocal);

// Аналоги операций x = l+r (l-r, l*r, l/r, l%r)
bn* bn_add(bn const *left, bn const *right);
bn* bn_sub(bn const *left, bn const *right);
bn* bn_mul(bn const *left, bn const *right);
bn* bn_div(bn const *left, bn const *right);
bn* bn_mod(bn const *left, bn const *right);

// Выдать представление BN в системе счисления radix в виде строки
// Строку после использования потребуется удалить.
const char *bn_to_string(bn const *t, int radix);

// Если левое меньше, вернуть <0; если равны, вернуть 0; иначе  >0
int bn_cmp(bn const *left, bn const *right);

int bn_neg(bn *t); // Изменить знак на противоположный
int bn_abs(bn *t); // Взять модуль
int bn_sign(bn const *t); //-1 если t<0; 0 если t = 0, 1 если t>0
```

Вы видели содержимое файла `bn.h`, которое вы не имеете права менять. В нём — *декларации* функций, которые вам нужно реализовать и *декларация* типа `bn`, как структуры `struct bn_s`.

Ваша цель — написать файл `bn_ваша_фамилия_латинскими_буквами.c`, который должен будет содержать *определения* структуры `struct bn_s` и *определения* всех интерфейсных функций из файла. Ваш файл может содержать любое количество *неэкспортируемых* вспомогательных данных и функций. Файл должен быть должным образом оформлен, документирован. Предоставленный файл будет использоваться в программе тестирования в автоматическом режиме в одном из контестов, о котором будет объявлено позже.

Важно! Написанные вами функции ничего не должны вводить и выводить! Конечно, при отладке вы можете использовать ввод/вывод, но в варианте,

присланном для тестирования его быть не должно. Все ошибочные ситуации должны быть обработаны без вывода чего-либо на экран. Функции, возвращающие указатели, должны при ошибке возвращать NULL, остальные функции должны возвращать 0 при успешном завершении операции и код ошибки из перечислимого типа `bn_codes` при неуспешном.

О тестировании. Часть тестов на минимальную положительную оценку будет проводиться для проверки корректности исполнения операций, при этом контроль указателей будет отключён. В остальных тестах будет проверяться корректность полученного ответа и будет проводиться жёсткая проверка обращения к памяти (`address-sanitizer`) и (`valgrind`).

Максимальная оценка будет ставиться за прохождение всех тестов. Трое, показавшие лучший результат времени в тестировании на скорость (при условии прохождения всех остальных тестов) получат дополнительные бонусы. Корректно реализовавшие функцию извлечения корня также получат бонусы.

Функции деления и нахождения остатка должны соответствовать математическому смыслу. То есть $(A/B) \cdot B + A \pmod B = A$, при этом знак остатка должен совпадать со знаком делителя.

```
17 / 10 = 1
17 % 10 = 7
-17 / 10 = -2
-17 % 10 = 3
17 / -10 = -2
17 % -10 = -3
-17 / -10 = 1
-17 % -10 = -7
```

За коллективное творчество баллы за задачу будут поровну делиться между всеми участниками коллектива.

Содержание созданного вами файла может быть примерно таким:

```
// Файл bn_krokodilov.c
#include "bn.h"

struct bn_s {
    int *body; // Тело большого числа
    int  bodysize; // Размер массива body
    int sign; // Знак числа
};

// Создать новое BN
bn *bn_new() {
```

```
   bn * r = malloc(sizeof bn);
   if (r == NULL) return NULL;
   r->bodysize = 1;
   r->sign = 0;
   r->body = malloc(sizeof(int) * r->bodysize);
   if (r->body == NULL) {
      free(r);
      return NULL;
   }
   r->body[0] = 0;
   return r;
}

// Создать копию существующего BN
bn *bn_init(bn const *orig) {
 ...
}

...
```

Далее следует простой пример, как можно использовать написанный вами код (используйте его в тестировании).

```
// Файл testbn.c
#include <stdio.h>
#include "bn.h"

int main() {
   bn *a = bn_new(); // a = 0
   bn *b = bn_init(a); // b тоже = 0
   int code = bn_init_string(a, "123456789012345678");
                          // a = 123456789012345678
   // Здесь и далее code - код ошибки.
   // 0 - всё хорошо
   // Не 0 - что-то пошло не так.
   code = bn_init_int(b, 999); // b = 999
   code = bn_add_to(a, b); // a = 123456789012346677
   code = bn_pow_to(b, 5); // b = 999^5
   code = bn_root_to(b, 5); // b = 999
   code = bn_init_int(a, 0); // a = 0
   code = bn_div_to(b,a); // OOPS, 999/0
   if (code != 0) {
```

```
        printf("bn_div_to failed, code=%d\n", code);
    }

    bn *c = bn_new();
    code = bn_init(c, 222);
    bn *d = bn_new();
    code = bn_init(d, 333);
    bn *e = bn_add(c, d); // e = 555
    bn *f = bn_mod(e, c); // f = 111

    const char *r1 = bn_to_string(f,10); // r1 -> "111"
    printf("f=%s\n", r1);
    free(r1);

    code = bn_cmp(c, d);
    if (code < 0) {
        printf("c < d\n");
    } else if (code == 0) {
        printf("c == d\n");
    } else {
        printf("c > d\n");
    }

    code = bn_neg(b); // b = -999;
    int bsign = bn_sign(b); // bsign = -1
    code = bn_abs(b); // b = 999;
    bn_delete(f); // Забудете удалить - провалите тесты
    bn_delete(e);
    bn_delete(d);
    bn_delete(c);
    bn_delete(b);
    bn_delete(a);
    return 0;
}
```