#pragma once

#include<iostream>

#include <sstream>

//the degree of detail of the output

enum class output\_mode

{

FULL,

ABBREVIATED,

SHORT

};

template<typename P> class polynomial;

template<typename P> std::ostream& operator<<(std::ostream& out, const polynomial<P>& plnm);

template<typename P> std::istream& operator>>(std::istream& in, polynomial<P>& plnm);

template<typename P> class polynomial

{

private:

//degree of the polynomial

uint64\_t deg;

public:

//an array of coefficients

P\* ptr;

//default output mode

static output\_mode default\_output\_mode;

//class realization output mode

output\_mode outm\_E;

//CONSTRUCTORS\DESTRUCTORS

polynomial() :polynomial(0) {};

polynomial(P number, output\_mode mode) :polynomial(number) { outm\_E = mode; }

polynomial(P number);

polynomial(const polynomial<P>& other, output\_mode mode) :polynomial(other) { outm\_E = mode; }

polynomial(const polynomial<P>& other);

~polynomial();

//DATA ACCESS

//get the degree of the polynomial

uint64\_t get\_deg() { return deg; };

//set the degree of the polynomial

void set\_deg(uint64\_t newdeg) { deg = newdeg; };

//ARITHMETIC OPERATORS

//addition of polynomials

polynomial<P> operator+(const polynomial<P>& other)const;

//subtruction of polynomials

polynomial<P> operator-(const polynomial<P>& other)const;

//binary division of a polynomial by a polynomial

polynomial<P> operator/(const polynomial<P>& other)const;

//the remainder of the division of a polynomial by a polynomial

polynomial<P> operator%(const polynomial<P>& other)const;

//binary polynomial multiplication

polynomial<P> operator\*(const polynomial<P>& other)const;

//binary polynomial multiplication by an element from the field

polynomial<P> operator\*(const P& other)const;

// I/O OPERATIONS

//the output operator (with the degree of detail specified in the outm\_E field)

friend std::ostream& operator<<<>(std::ostream& out, const polynomial<P>& plnm);

//the input operator

friend std::istream& operator>><>(std::istream& in, polynomial<P>& plnm);

//CHARACTER SHIFTS

//coefficient shift (increase). or (plnm\*x^n)

polynomial<P> operator>>(const uint64\_t power)const;

//coefficient shift(decrease). or the whole part of (plnm\* x^(-n))

polynomial<P> operator<<(const uint64\_t power)const;

//COMPARISON OPERATORS

//is it true if the polynomials are equal

bool operator==(const polynomial<P>& other)const;

//is it false if the polynomials are equal

bool operator!=(const polynomial<P>& other)const;

//is it true if this->deg > other.deg;

bool operator>(const polynomial<P>& other)const;

//is it true if this->deg < other.deg;

bool operator<(const polynomial<P>& other)const;

//COMPARISON OPERATORS WITH ZERO

//isZero()?(all values==0 so polynomial have zero coefficent in any position)

bool operator==(int64\_t zero)const;

//!isZero()?(all values==0 so polynomial have zero coefficent in any position)

bool operator!=(int64\_t zero)const { return not((\*this) == zero); }

//SPECIAL METHODS

//the access operator to the polynomial coefficient

P& operator[](uint64\_t index);

//equalization operator(low coefficient=other )

polynomial<P>& operator=(const P other);

//the equalization operator

polynomial<P>& operator=(const polynomial<P>& other);

//reduction of the polynomial (due to the higher zero coefficients)

polynomial cutbag()const;

//reallocate memory with zeros (old data is not saved)

void newsize(uint64\_t size);

//set the output mode via a variable of type: uint64\_t newmode

void output\_mode\_set(uint64\_t newmode);

//set the output mode via a variable of type: uint64\_t newmode

void output\_mode\_set(output\_mode new\_outm\_E);

//set the output mode via a variable of type: std::string newmode

void output\_mode\_set(std::string newmode);

};

template<typename P> output\_mode polynomial<P>::default\_output\_mode = output\_mode::SHORT;

#include "polynomial.cpp"