```
#ifndef COMPLEJO_H
#define COMPLEJO_H

#include <iostream>
/**
 * Clase representativa de un número complejo. Posee algunas de las operaciones más
 * comunes que se pueden llegar a necesitar para operar con numeros complejos.
 */
class complejo {

    //Parte real del número complejo
    double re_;

    //Parte imaginaria del número complejo
    double im_;

public:
    /**
     * Constructor sin parámetros. Este constructor permite inicializar un
     * complejo en (0,0).
     */
    complejo();

    /**
     * Constructor. Este constructor permite crear un complejo a partir de
     * un número real. La parte imaginaria queda inicializada en cero.
     */
    complejo(double);

    /**
     * Constructor. Este constructor permite crear un complejo a partir de
     * dos números reales. El primero corresponde a la parte real y el
     * segundo a la parte imaginaria.
     */
```

```
    complejo(double,double);

    /**
     * Constructor por copia. Construye un complejo a partir de la copiar de otro.
     */
    complejo(const complejo &);

    /**
     * Sobrecarga operador asignación. Asigna parte real a parte real
     * y parte imaginaría a parte imaginaría.
     */
    complejo const &operator=(complejo const &);

    /**
     * Destructor. Destructor para un numero complejo.
     */
    ~complejo();

    /**
     * Parte real de un complejo. Este método devuelve la parte real de
     * un número complejo
     */
    double re() const;

    /**
     * Parte imaginaria de un complejo. Este método devuelve la parte
     * imaginaria de un número complejo.
     */
    double im() const;

    /**
     * Módulo de un número complejo. Este método devuelve el modulo de
     * un complejo.
     */
    double abs() const;
```

```
/**
 * Fase de un número complejo. Este método devuelve la fase de un número
 * complejo.
 */
double phase() const;

/**
 * Conversor de complejo de forma polar a forma cartesiana. Este método
 * recibe por parámetro el modulo y la fase de un complejo y retorna
 * un número complejo en su forma cartesiana.
 */
static complejo fromPolarToRectangular(double,double);

/**
 * Sobrecarga operador suma. Este método suma dos números complejos.
 */
friend complejo const operator+(complejo const &, complejo const &);

/**
 * Sobrecarga operador resta. Este método resta dos números complejos.
 */
friend complejo const operator-(complejo const &, complejo const &);

/**
 * Sobrecarga operador multiplicación. Este método multiplica dos
 * números complejos.
 */
friend complejo const operator*(complejo const &, complejo const &);

/**
 * Sobrecarga operador división. Este método divide un complejo con
 * un número real.
 */
friend complejo const operator/(complejo const &, double);
```

```
/**
 * Sobrecarga operador potenciación. Este método potencia un número
 * complejo con un numero entero.
 */
friend complejo const operator^(complejo const &,int);

/**
 * Sobrecarga operador igual. Este método compara la parte real de un
 * un complejo con un numero real. Además verifica que la parte
 * imaginaria del complejo sea cero.
 */
friend bool operator==(complejo const &, double);

/**
 * Sobrecarga operador igual. Este método compara dos números complejos.
 * Parte real con parte real y parte imaginaria con parte imaginaria.
 */
friend bool operator==(complejo const &, complejo const &);

/**
 * Sobrecarga operador escritura. Este operador escribe un complejo
 * en formato (Re,Img) al flujo de salida.
 */
friend std::ostream &operator<<(std::ostream &, const complejo &);

/**
 * Sobrecarga operador lectura. Este operador lee del flujo de entrada
 * un número complejo. Acepta números reales individuales a los cuales
 * se los toma como la parte real del numero complejo que retorna.
 * En el caso de que reciba al numero complejo como par ordenado
 * devuelve un complejo con la parte real y la parte imaginaria
 * obtenida del flujo de entrada.
 */
friend std::istream &operator>>(std::istream &, complejo &);
```

```
};

#endif
```

```cpp
#include "complejo.h"
#include <iostream>
#include <cmath>
#include <stdlib.h>

using namespace std;

complejo::complejo() : re_(0), im_(0) {}

complejo::complejo(double r) : re_(r), im_(0){}

complejo::complejo(double r, double i) : re_(r), im_(i){}

complejo::complejo(complejo const &c) : re_(c.re_), im_(c.im_){}

complejo const & complejo::operator=(complejo const &c){
    re_ = c.re_;
    im_ = c.im_;
    return *this;
}

complejo::~complejo() {}

double complejo::re() const {
    return re_;
}

double complejo::im() const {
    return im_;
}

double complejo::abs() const {
    return std::sqrt(re_ * re_ + im_ * im_);
}
```

```
double complejo::phase() const {
    return atan(this->im_/this->re_);
}


complejo complejo::fromPolarToRectangular(double mod,double phase){
    double re = mod*cos(phase);
    double im = mod*sin(phase);
    return complejo(re,im);
}

complejo const operator+(complejo const &x, complejo const &y){
    complejo z(x.re_ + y.re_, x.im_ + y.im_);
    return z;
}

complejo const operator-(complejo const &x, complejo const &y){
    complejo r(x.re_ - y.re_, x.im_ - y.im_);
    return r;
}

complejo const operator*(complejo const &x, complejo const &y){
    complejo r(x.re_ * y.re_ - x.im_ * y.im_,
               x.re_ * y.im_ + x.im_ * y.re_);
    return r;
}


complejo const operator/(complejo const &c, double f)
{
        return complejo(c.re_ / f, c.im_ / f);
}
```

```
complejo const operator^(complejo const &c , int power){

    if(power == 0){
        return complejo(1,0);
    }

    if(power == 1){
        return c;
    }

    double module = c.abs();
    double phase = c.phase();
    if(power < 0){
        module = 1/module;
    }

    for(int i = 0; i < power - 1 ; i++){
        module = module*module;
    }
    phase = phase*power;

    return complejo::fromPolarToRectangular(module,phase);
}


bool operator== (complejo const &c, double f){
    bool b = (c.im_ != 0 || c.re_ != f) ? false : true;
    return b;
}

bool operator== (complejo const &x, complejo const &y){
    bool b = (x.re_ != y.re_ || x.im_ != y.im_) ? false : true;
    return b;
}
```

```
ostream & operator<<(ostream &os, const complejo &c){
    double reAux = floor(c.re_*100)/100;
    double imAux = floor(c.im_ *100)/100;
    return os << "(" << reAux << "," << imAux << ")";
}

istream & operator>>(istream &is, complejo &c) {
    int good = false;
    int bad  = false;
    double re = 0;
    double im = 0;
    char ch = 0;

    if (is >> ch && ch == '(') {
        if (is >> re && is >> ch && ch == ',' && is >> im && is >> ch
            && ch == ')')
                good = true;
        else
                bad = true;
    } else if (is.good()) {
        is.putback(ch);
        if (is >> re)
                good = true;
        else
                bad = true;
    }

    if (good)
        c.re_ = re, c.im_ = im;
    if (bad)
        is.clear(ios::badbit);

    return is;
}
```

```
/*
 * File:   vector.h
 * Author: Diego / Marcelo
 *
 * Created on March 21, 2016, 8:56 PM
 */

#ifndef VECTOR_H
#define VECTOR_H

#include <iostream>

using namespace std;


/**
 * Clase vector. Permite almacenar un arreglo de datos. Contiene una serie de
 * de métodos que permiten agregar, quitar y leer los distintos elementos almacenados
 * en el vector. Además la clase esta templetizada, permitiendo así crear vectores
 * de cualquier tipo.
 */
template <class T>
class vector{

    private:
        //Tamaño del vector
        int size;
        //Cantidad de espacio en memoria del vector
        int capacity;
        //Puntero a la primera posición del vector
        T* pv;

    public:
```

```
/**
 * Constructor sin parámetros. Inicializa un vector en cero. Además no
 * crea la memoria para el vector.
 */
vector(){
    this->pv = NULL;
    this->size = 0;
    this->capacity = 0;
}

/**
 * Constructor. Inicializa un vector vacío. A diferencia del constructor
 * sin parámetros, este crea la memoria para el vector de acuerdo al
 * parámetro size_.
 */
vector(int size_){
    this->pv = new T[size_];
    this->size = size_;
    this->capacity = size_;
}

/**
 * Constructor copia. Inicializa un vector a partir de otro pasado
 * por parámetro.
 */
vector(const vector<T> & cv){
    this->size = cv.size ;
    this->capacity = cv.capacity;
    T* cp = new T[ size ];
    for ( int i = 0; i < this->size; i++ ){
        cp[ i ] = cv.pv[ i ];
    }
    if(this->pv){
        delete[] this->pv;
    }
```

```
        this->pv = cp;
    }

    /**
     * Destructor de vector. borra la memoria creada para el vector.
     */
    ~vector(){
        if(this->pv){
            delete [] this->pv;
        }
    }

    /**
     * Largo del vector. Este método devuelve el largo del vector.
     */
    int length() const{
        return this->size;
    }

    /**
     * Inserta un elemento al vector. Este método inserta al final del vector
     * un elemento. Para agregar el elemento primero verifica que tenga
     * memoria, en caso de no tener crea memoria con capacidad igual al
     * doble de la que tenía.
     */
    void pushBack(T & elem){
        if(this->size == 0){
            this->pv = new T[2];
            this->capacity = 2;
        }
        else{
            if(this->capacity == this->size){
                T* aux = this->pv;
                this->pv = new T[this->capacity*2];
                this->capacity = this->capacity*2;
```

```
            for(int i = 0 ; i < this->size ; i++){
                this->pv[i] = aux[i];
            }
            delete[] aux;
        }
    }
    this->pv[this->size] = elem;
    this->size++;
}

/**
 * Operador asignación. Este operador copia los elementos del vector
 * pasado por parámetro al objeto sobre el cual se ejecuto.
 */
vector<T> & operator=(const vector<T> & rigth) {
    if (&rigth != this)
    {
        if (this->size != rigth.size) {
            T * aux;
            aux = new T[ rigth.size ];
            delete [] this->pv;
            this->size = rigth.size;
            this->pv = aux;
            for (int i = 0; i < size; i++){
                this->pv[i] = rigth.pv[i];
            }
            return *this;
        }
        else
        {
            for (int i = 0; i < this->size; i++){
                this->pv[i] = rigth.pv[i];
            }
            return *this;
        }
```

```
    }
    return *this;
}


/**
 * Operador comparación. Compara el contenido del vector pasado por
 * parámetro con el vector sobre el cual se ejecuto el operador. En el
 * caso que todos los elementos coincidan devuelve true.
 */
bool operator==(const vector<T> & rigth) const {
    if (this->size != rigth.size)
        return false; // Vectores de diferentes tamaños
    else
        for (int i = 0; i < this->size; i++)
            if (this->pv[ i ] != rigth.pv[ i ])
                return false;
    return true;
}

/**
 * Operador indexación constante. Permite obtener el objeto almacenado en una
 * determinada posición del vector.
 */
const T & operator[](int index) const {
    if(index >= this->size){
        cerr << "Indice incorrecto en const operator[]" << endl;
        abort();
    }
    return this->pv[index];
}

/**
 * Operador indexación. Permite asignarle un valor a una determinada
 * posición del vector.
```

```
         */
        T & operator[](int index) {
            if(index >= this->size){
                cout << "Indice incorrecto en operator[]" << endl;
                abort();
            }
            return this->pv[index];
        }

        /**
         * Operador lectura. Permite leer de un stream de entrada un conjunto
         * de objetos del tipo T, almacenándolos en un vector.
         */
        friend std::istream &operator>>(std::istream & is,vector<T> & vector){
            T aux;
            for(int i = 0 ; is >> aux ; i++){
                vector.pushBack(aux);
            }
            return is;
        }

        /**
         * Operador escritura. Permite escribir en un stream de salida el vector
         * del cual se llamo.
         */
        friend std::ostream &operator<<(std::ostream & os, const vector<T> & vector){
            for(int i = 0 ; i < vector.size ; i++){
                os << vector[i] << endl;
            }
            return os;
        }
};


#endif /* VECTOR_H */
```

```cpp
/*
 * File:   DFTcalculator.cc
 * Author: Diego / Marcelo
 *
 * Created on March 21, 2016, 8:09 PM
 */

#include <cstdlib>
#include "vector.h"
#include "complejo.h"
#include <fstream>
#include <iostream>
#include <cmath>
#include <string>

#ifndef M_PI
    #define M_PI 3.14159265358979323846
#endif

/**
 * Clase DFTcalculator. Esta clase contiene una serie de metodos que permiten
 * calcular la transformada discreta de fourier y la transformada inversa
 * de fourier.
 */
class DFTcalculator
{
    private:


    /**
     * Metodo privado estatico calculate. Este metodo permite calcular tanto la
     * transformada como la anti transformada de fourier. Es un metodo privado que
     * utiliza la clase.
     */
```

```
static void calculate(const vector<complejo> & data , vector<complejo> & result , string algo
    int N = data.length();
    double re = cos(2*M_PI/N);
    double im = sin(2*M_PI/N);
    int sign = 1;
    if(algorithm == "idft"){
        sign = -1;
    }
    for(int i = 0 ; i < N ; i++){
        complejo sum = 0;
        for(int j = 0 ; j < N ; j++){
            complejo aux(re,im);
            aux = aux^(sign*j*i);
            sum = sum + (data[j]*aux);
        }
        if(algorithm == "idft"){
            sum = sum / N;
        }
        result.pushBack(sum);
    }
}


public:

/**
 * Método el cual permite calcular la transformada discreta de fourier. Recibe
 * por parámetro dos vectores uno con la información y otro donde escribirá
 * el resultado.
 */
static void calculateDFT(const vector<complejo> & data , vector<complejo> & result)
{
    calculate(data , result , "dft");
}
```

```
    /**
     * Método el cual permite calcular la anti-transformada discreta de fourier.
     * Recibe dos parámetros uno con la información y otro donde la escribirá.
     */
    static void calculateIDFT(const vector<complejo> & data , vector<complejo> & result)
    {
        calculate(data , result , "idft");
    }

};
```

```
/*
 * File:   main.cpp
 * Author: Diego / Marcelo
 *
 * Created on March 21, 2016, 8:09 PM
 */

#include <cstdlib>
#include "vector.h"
#include "complejo.h"
#include <fstream>
#include <iostream>
#include <cmath>
#include "cmdline.h"
#include "DFTcalculator.h"

using namespace std;

#define OPT_DEFAULT   0
#define OPT_SEEN      1
#define OPT_MANDATORY 2


static string method;
static istream *iss = NULL;
static ostream *oss = NULL;
static fstream ifs;
static fstream ofs;

static void opt_input(string const &arg){
    if (arg == "-") {
        iss = &cin;
    }
    else {
```

```
        ifs.open(arg.c_str(), ios::in);
        iss = &ifs;
    }

    if (!iss->good()) {
        cerr << "cannot open " << arg << "." << endl;
        abort();
    }
}

static void opt_output(string const &arg){
    if (arg == "-") {
        oss = &cout;
    }
    else {
        ofs.open(arg.c_str(), ios::out);
        oss = &ofs;
    }

    if (!oss->good()) {
        cerr << "cannot open " << arg << "." << endl;
        abort();
    }
}

static void opt_help(string const &arg){
    cout << "cmdline [-m method] [-i file] [-o file]" << endl;
    cout << "The default input output are the standar IO.";
    cout << "The default method is discrete fourier transform.";
    exit(0);
}

static void opt_method(string const &arg){
    method = arg;
}
```

```
static option_t options[] = {
        {1, "i", "input", "-", opt_input, OPT_DEFAULT},
        {1, "o", "output", "-", opt_output, OPT_DEFAULT},
        {1, "m", "method", "dft", opt_method, OPT_DEFAULT},
        {0, "h", "help", NULL, opt_help, OPT_DEFAULT},
        {0, },
};

/**
 * Función main. El programa espera por comando que le configuren el stream de entrada
 * de donde tomará los datos para calcular la transformada discreta de fourier.
 * Además necesita que le configuren el stream de salida a donde escribirá los
 * datos obtenidos.
 */
int main(int argc, char** argv) {

    //Creo un vector que almacenará la información leída
    vector<complejo> data = vector<complejo>();

    //Leo las opciones con las que me ejecutan el programa
    cmdline cmdl(options);
    cmdl.parse(argc, argv);

    //Leo la información del stream de entrada
    *iss >> data;

    //Calculo la serie
    vector<complejo> result = vector<complejo>();
    if(method == "dft"){
        DFTcalculator::calculateDFT(data,result);
    }
    else if(method == "idft"){
        DFTcalculator::calculateIDFT(data,result);
    }
```

```
    //Guardo el resultado en el stream de salida
    *oss << result << endl;

    //Borramos la memoria creada.
    delete &data;
    delete &result;

    return 0;
}
```