```
// Complexe.h
// Version MM 2004

#ifndef _COMPLEXE_H_
#define _COMPLEXE_H_

#include <iostream>
#include "util.h" // string, intToString, floatToString, etc...

class Complexe{
 private:
  float m_reel, m_img;
  static const int PRECISION;
 public:
  Complexe();
  Complexe(float a, float b);
  ~Complexe();
  Complexe(const Complexe & z);
  string toString() const;
  void setReel(float a);
  void setImg(float b);
  bool egal(const Complexe& autre) const;
  void additionner(const Complexe& z, Complexe& somme) const;
  void multiplier(const Complexe& z, Complexe& produit) const;
  void soustraire(const Complexe& z, Complexe& difference) const;
  void diviser(const Complexe& z, Complexe& quotient) const;
  void inverse(Complexe& inv) const;
  void conjugue(Complexe& z_barre) const;
  float module() const;
  Complexe& operator=(const Complexe& autre);
  Complexe operator+(const Complexe&) const;
  Complexe operator*(const Complexe&) const;
  Complexe inverse() const;
  Complexe conjugue() const;
  bool operator==(const Complexe&) const;

  // tests idiots...
  ostream& operator>>(ostream& out);
  istream& operator<<(istream& in);

};

#endif
```



```
// Complexe.cc
// Version MM 2004

#include <iostream>
#include <string>
#include <cmath>
#include "Complexe.h"

using namespace std;

const int Complexe::PRECISION=2;

Complexe::Complexe(){
  m_reel = m_img = 0;
}
Complexe::Complexe(float a, float b){
  m_reel = a;
  m_img = b;
}

Complexe::~Complexe(){}

Complexe::Complexe( const Complexe & z ){
  cout << "Complexe::constructeur par copie" << endl;
  m_reel = z.m_reel;
  m_img = z.m_img;
}

void Complexe::setReel(float a){ m_reel = a; }

void Complexe::setImg(float b){ m_img = b; }

bool Complexe::egal(const Complexe& autre) const{
  return m_reel == autre.m_reel && m_img == autre.m_img;
}

string Complexe::toString() const {

  string s = "("+floatToString(m_reel, PRECISION) ;

  if(m_img>=0)
    s+= " + "+floatToString(m_img, PRECISION)+"i)";
  else s+= " - "+floatToString(-m_img, PRECISION)+"i)";

  return s;
}

void Complexe::additionner(const Complexe& autre, Complexe& somme) const{
  somme.m_reel = m_reel + autre.m_reel;
  somme.m_img = m_img + autre.m_img;
}

void Complexe::multiplier(const Complexe& autre, Complexe& produit) const{
  produit.m_reel = m_reel * autre.m_reel - m_img * autre.m_img;
  produit.m_img = m_reel * autre.m_img + m_img * autre.m_reel;
}

void Complexe::soustraire(const Complexe& autre, Complexe& difference) const{
  difference.m_reel = m_reel - autre.m_reel;
  difference.m_img = m_img - autre.m_img;
}

void Complexe::inverse(Complexe& inv) const{
// 1/(a + ib) = a/(a^2 + b^2) - b/(a^2 + b^2)
  float module_carre = pow(m_reel,2) + pow(m_img,2);
  inv.m_reel = m_reel/module_carre;
  inv.m_img = -m_img/module_carre;
```

```
}

void Complexe::diviser(const Complexe& autre, Complexe& quotient) const{
  Complexe inv_autre;
  autre.inverse(inv_autre);
  multiplier(inv_autre, quotient);
}

void Complexe::conjugue(Complexe& z_barre) const{
// conjugué de a+ib = a - ib
  z_barre.m_reel = m_reel;
  z_barre.m_img = -m_img;
}

float Complexe::module() const{
// |z| = racine carrée de a^2 + b^2
  return sqrt(pow(m_reel,2) + pow(m_img,2));
}

Complexe& Complexe::operator=(const Complexe & z){
  cout << "Complexe::Operateur affectation" << endl;
  if ( this != &z ) {
    m_reel = z.m_reel;
    m_img = z.m_img;
  }
  return *this;
}


bool Complexe::operator==(const Complexe & autre) const{
  return m_reel==autre.m_reel && m_img==autre.m_img;
}

Complexe Complexe::operator+(const Complexe& autre) const{
  Complexe somme(m_reel + autre.m_reel,m_img + autre.m_img);

  return somme;
}

Complexe Complexe::operator*(const Complexe& autre) const{

  Complexe produit(m_reel * autre.m_reel - m_img * autre.m_img,
                   m_reel * autre.m_img + m_img * autre.m_reel);

  return produit;
}

Complexe Complexe::inverse() const{
// 1/(a + ib) = a/(a^2 + b^2) - b/(a^2 + b^2)

  float module_carre = pow(m_reel,2) + pow(m_img,2);
  Complexe inv(m_reel/module_carre, -m_img/module_carre);

  return inv;

}

Complexe Complexe::conjugue() const{
// conjugué de a+ib = a - ib
  Complexe zb(m_reel, -m_img);
  return zb;
}

// A NE PAS FAIRE, MAIS CA MARCHE
ostream& Complexe::operator>>(ostream& out){
  out << floatToString(m_reel, PRECISION) <<((m_img<0)?" - ":" + ")
      << floatToString(((m_img<0)?-m_img:m_img),PRECISION) << "i";
```



```
  return out;
}

istream& Complexe::operator<<(istream& in){
  in >> m_reel >> m_img;
  return in;
}

/*
Complexe Complexe::operator/(const Complexe& autre) const{
  cout << " / membre " << endl;
  return *this * autre.inverse();
}
*/
```

```
#ifndef _UTIL_H
#define _UTIL_H

#include <string>

using namespace std;

int pgcd(int a, int b);
string intToString(int n);
string floatToString(float x, int precision);

#endif
```



```
#include <string>
#include <cmath>
#include <cassert>
#include "util.h"

using namespace std;

int pgcd(int a, int b){

  // a et b >= 0
  assert(a>=0 && b>=0);

  //pgcd(0,x)=pgcd(x,0)=0
  if(a*b==0)
    return 0;

  // trois cas : a > b ; a = b ; a < b
  if(a>b)
    return pgcd(b, a-b);

  if(a==b)
    return a;

  return pgcd(a, b-a);
}

string intToString(int x){
  string x_s;
  int a;

  if(x==0)
    x_s="0";
  else{
     x_s="";
    if(x<0)
      a=-x;
    else
      a=x;
    while(a>0){
      x_s= ((char)('0'+a%10))+x_s;
      a/=10;
    }
    if(x<0)
      x_s="-"+x_s;
  }
  return x_s;
}

string floatToString(float x, int precision){
  string x_s;

  if(x==0){
    x_s="0";
    for(int i=0;i<precision;i++)
      x_s+="0";
  }
  else{
    int partie_conservee = (int)(x * pow(10.0,(double)precision));
    x_s = intToString(partie_conservee);
  }
  if(precision > 0)
    x_s.insert(x_s.length() - precision, ".");

  return x_s;
}
```

```
#include <iostream>
#include <cmath>
#include "Complexe.h"

using namespace std;

// fonctions non membres

ostream & operator<<(ostream& out, const Complexe & z){
    out << z.toString();
    return out;
}

Complexe operator/(const Complexe& z1, const Complexe& z2){
  cout << " / non membre" << endl;
  return z1 * z2.inverse();
}

Complexe operator-(const Complexe& z1, const Complexe& z2){
  // pas de get, il faut utiliser les services proposés par la classe
  Complexe difference;
  z1.soustraire(z2, difference);
  return difference;
}



int main(){
  Complexe z1(-1.0,1.0), z2((float)(sqrt(2.0)), (float)(-sqrt(3.0))), z3;

  cout << z1 << " + " << z2
       << " = " << z1+z2 << endl;
  // test de >> cout (surchargé dans la classe)
  z1 >> cout ;
  cout << " + (pipo) ";
  z2 >> cout;
  cout << " = ";
  z1+z2 >> cout ;
  cout << endl;

  // test de << cin (surchargé dans la classe)
  Complexe z23;
  z23 << cin;
  z23 >> cout ;


  Complexe z4 = z2+z1;
  cout << z4 << endl;
  z3= z4+z2;
  cout << z3 << " = " << z4 << " + " << z2 << endl;
  Complexe zz(2, 3);
  Complexe zz2(1, 2);
  cout << zz << " * " << zz2 << " = " << zz*zz2 << endl;
  cout << zz << " / " << zz2 << " = " << zz/zz2 << endl;


  cout << "conjugué de " << zz << " = " << zz.conjugue() << endl;
  Complexe zb=zz.conjugue();
  Complexe r(zz*zb);
  cout << zz << " * " << zb << " = " << r << endl;
  cout << "|" << zz << "| = " << zz.module() << endl;

}
```