

LA CLASSE DES NOMBRES COMPLEXES

Illustration des notions de constructeurs, de surcharges d'opérateurs, de fonctions amies.

FICHIER complex.h

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

#include <iostream>
#include <math.h>
#include <assert.h>

using namespace std;

class complex
{
private:
    /** Représentation interne d'un nombre complexe par leurs parties réelle et imaginaire */
    double m_re;
    double m_im;

public:
    /** Les deux imaginaires purs i et j, ce sont des constantes
     *  tels que  $i^2 = j^2 = -1$ 
     */
    static const complex i;
    static const complex j;

    /** ##### LES CONSTRUCTEURS #####
     */

    /** Constructeur à 2 paramètres avec 2 valeurs par défaut
     *  utilisation d'une liste d'initialisation pour initialiser les attributs
     *
     *  MODE D'EMPLOI :
     *  utilisation avec 2 arguments:
     *  complex z = complex(6,9); soit z = (6,9)
     *  ou complex z(6,9); soit z = (6,9)
     *
     *  Sert aussi de constructeur de transtypage (ou de conversion) d'un double (la partie réelle)
     *  vers un complex : complex(const double re)
     *  utilisation avec 1 argument ==> le second prend la valeur par défaut 0
     *  complex z = complex(4); soit z = (4, 0)
     *
     *  Sert aussi de constructeur par défaut complex()
     *  utilisation avec 0 argument ==> les 2 arguments prennent la valeur par défaut 0
     *  complex z = complex(); soit z = (0, 0)
     */
};
```

```

complex(const double re = 0, const double im = 0): m_re(re), m_im(im)
{
    if ((re == 0) && (im == 0))
    {
        cout << "CONSTRUCTEUR PAR DEFAUT" << endl;
    }
    else if(im == 0)
    {
        cout << "CONSTRUCTEUR DE CONVERSION : TRANSTYPAGE DOUBLE -->
COMPLEXE" << endl;
    }
    else
        cout << "CONSTRUCTEUR A DEUX PARAMETRES" << endl;
}

```

/ LE CONSTRUCTEUR DE COPIE n'est volontairement pas programmé.**

* En effet le compilateur C++ est en mesure de le générer automatiquement,
* de telle sorte que la copie membre à membre d'un complexe soit réalisée.
* Les valeurs des attributs de l'instance courante, l'original donc, sont recopiées dans ceux de la
nouvelle
* instance créée, la copie donc.
* Les attributs sont de type double, la recopie par défaut, dite superficielle, ne pose aucun
* problème de partage de données : chaque copie a ses propres attributs.
* Attention : s'il y avait des attributs de type pointeur, la copie superficielle conduirait à partager
* les zones pointées.
* Par suite, toute modification de cette zone par l'une des instances, l'original ou une des copies,
* serait répercutée sur toutes ces mêmes instances. On parle dans ce cas d'effet de bord.
* Afin d'éviter cette copie superficielle, il convient de procéder à une copie dite profonde qui
* recopie les zones pointées dans de nouvelles zones allouées dynamiquement.
*/

/ ##### OPERATEUR DE TRANSTYPAGE complex --> double #####**
*/

/** Opérateur de transtypage de complex vers double
* Il est mis en oeuvre dans plusieurs cas:
* 1) lors du passage de paramètre à une fonction attendant un double en paramètre
* et recevant un complexe en argument :
* exemple : void f(double n) et appel f(z) ==> z(6.8), 12.9) converti en un double temporaire
valant 6.8
* 2) lors du retour d'une fonction devant retourner un double et à laquelle on retourne un
complexe.
* exemple : double g() { complex z(7.5, 99);; return z; }
* un double temporaire valant 7.5 sera retourné
* 3) lors d'une affectation d'un complexe à un double
* exemple : double n;
* n = z; ==> si z(6,8) alors n vaut 6
*/
operator double()
{
 cout << "TRANSTYPAGE COMPLEXE --> DOUBLE" << endl;
 return m_re;
}

```
/** ##### LES ACCESSEURS EN LECTURE ET EN ECRITURE #####  
*/
```

```
/** LES ACCESSEURS EN LECTURE
```

```
*/
```

```
double getRe() const
```

```
{
```

```
    return m_re;
```

```
}
```

```
double getIm() const
```

```
{
```

```
    return m_im;
```

```
}
```

```
/** LES ACCESSEURS EN ECRITURE
```

```
*/
```

```
void setRe(const double re)
```

```
{
```

```
    m_re = re;
```

```
}
```

```
void setIm(const double im)
```

```
{
```

```
    m_im = im;
```

```
}
```

```
/** ##### DIVERS #####  
*/
```

```
/** Le conjugué d'un complex  $a + i b$  et  $a - i b$ 
```

```
*/
```

```
complex conjugate() const
```

```
{
```

```
    return complex(m_re, -m_im);
```

```
}
```

```
/** le module d'un nombre complexe
```

```
*/
```

```
double module() const
```

```
{
```

```
    return sqrt(m_re * m_re + m_im * m_im);
```

```
}
```

```

/** ##### SURCHARGE DE L'OPERATEUR = #####
*/

/** Surcharge de l'opérateur d'affectation ( Assignment in English)
* Seul est surchargé complex & operator= (const double val)
* Cela correspond au cas  $z = 3.7 \iff z.operator+(3.7)$ 
* En effet, l'affectation entre 2 complexes se fait par copie des valeurs des membres.
* Le scénario est équivalent à celui du constructeur de copie.
* Par suite, on parle d'affectation superficielle ou profonde...
* Concrètement: la surcharge complex & operator= (const complex & other) est inutile.
*  $z2 = z1; \iff z2.operator+(z1)$  fonctionne par défaut avec une assignation superficielle
*
*/

/** INUTILE affectation superficielle générée automatiquement par le compilateur
*  $z2 = z1; \iff z2.operator+(z1)$ 
* La référence en retour sert à chaîner les affectations :  $z1 = z2 = z3 \dots$ 
*/
complex & operator= (const complex & other)
{
    /** auto-affectation détextée par égalité des pointeurs
    * rien à faire, si ce n'est retourner une référence sur l'instance courante!
    */
    if (this == &other)
        return *this;

    m_re = other.m_re;
    m_im = other.m_im;
    return *this;
}

/**  $z = 3.7 \iff z.operator+(3.7)$ 
* si cette surcharge n'existait pas, le compilateur devrait contruire un complexe (3.7, 0)
* lors du passage de paramètre de l'appel à complex & operator= (const complex & other)
* on aurait :  $z = 3.7 \iff z.operator+(complex(3.7, 0.0))$ 
* La référence en retour sert à chaîner les affectations :  $z1 = z2 = z3 \dots$ 
*/
complex & operator= (const double val)
{
    m_re = val;
    m_im = 0.;
    return *this;
}

```

```

/** ##### SURCHARGE DES OPERATEURS +=, -=, *=, /= #####
*/
/** MEME RAISONNEMENT QUE POUR LA SURCHARGE DE =
*/
complex & operator+=(const complex & other)
{
    m_re += other.m_re;
    m_im += other.m_im;
    return *this;
}

/** MEME RAISONNEMENT QUE POUR LA SURCHARGE DE =
*/
complex & operator+=(const double val)
{
    m_re += val;
    return *this;
}

/** MEME RAISONNEMENT QUE POUR LA SURCHARGE DE =
*/
complex & operator-=(const complex & other)
{
    m_re -= other.m_re;
    m_im -= other.m_im;
    return *this;
}

/** MEME RAISONNEMENT QUE POUR LA SURCHARGE DE =
*/
complex & operator-=(const double val)
{
    m_re -= val;
    return *this;
}

/** MEME RAISONNEMENT QUE POUR LA SURCHARGE DE =
*/
complex & operator*=(const complex & other)
{
    const double temp = m_re; // sauvegarde préventive
    m_re = m_re * other.m_re - m_im * other.m_im;
    m_im = m_im * other.m_re + temp * other.m_im;
    return *this;
}

/** MEME RAISONNEMENT QUE POUR LA SURCHARGE DE =
*/
complex & operator*=(const double val)
{
    m_re *= val;
    m_im *= val;
}

```

```

    return *this;
}

/** MEME RAISONNEMENT QUE POUR LA SURCHARGE DE =
 */
complex & operator/= (const complex & other)
{
    const double denominator = other.m_re * other.m_re +
                                other.m_im * other.m_im;
    assert(denominator != 0);

    const double temp = m_re; // sauvegarde préventive
    m_re = (m_re * other.m_re + m_im * other.m_im) / denominator;
    m_im = (m_im * other.m_re - temp * other.m_im) / denominator;
    return *this;
}

/** MEME RAISONNEMENT QUE POUR LA SURCHARGE DE =
 */
complex & operator/= (const double val)
{
    assert(val != 0);
    m_re /= val;
    m_im /= val;
    return *this;
}

/** ##### SURCHARGES DES OPERATEURS BINAIRES ARITHMETIQUES +, -, *, / #####
 */
/** Pour  $z1 + z2 \iff z.operator+(z2)$ 
 * Un nouveau nombre complexe temporaire, résultat de l'addition, est retourné.
 * L'associativité gauche du + est alors possible:  $z1 + z2 + z3 \iff (z1 + z2) + z3$ 
 */
complex operator+ (const complex & other) const
{
    return complex(m_re + other.m_re, m_im + other.m_im);
}

/** Pour  $z + 3.7 \iff z.operator+(3.7)$ 
 * si cette surcharge n'existait pas, le compilateur devrait contruire un complexe (3.7, 0)
 * lors du passage de paramètre de l'appel à complex operator+ (const complex & other)
 * on aurait :  $z + 3.7 \iff z.operator+(complex(3.7, 0.0))$ 
 */
complex operator+ (const double val) const
{
    return complex(m_re + val, m_im);
}

```

```

/** Il faut utiliser une fonction externe C pour pouvoir écrire 3.7 + z
 * On a alors :  $3.7 + z \iff \text{operator+}(3.7, z)$ 
 * En effet une fonction membre donnerait  $3.7.\text{operator+}(z)$  ce qui est impossible pour la raison
suivante :
 * Dans ce contexte, le double 3.7 devrait être vu comme une instance de la classe complexe.
 * On pourrait espérer une conversion du double vers un complexe par appel implicite au
constructeur de conversion.
 * Mais une instance ne peut être convertie en un autre type!
 *
 * Rendre la fonction extene amie de la classe, permet à la fonction d'accéder à la partie privée
de la classe.
 * Sans cette déclaration d'amitié, le recours aux accesseurs en lecture eût été nécessaire.
 */
friend complex operator+ (const double left, const complex & right)
{
    return complex(left + right.m_re, right.m_im);
}

```

```

/** REMARQUES
 * 1) il aurait été possible de ne définir que la fonction externe amie
 * friend complex operator+ (const complex & left, const complex & right)
 * Mais cela aurait nécessité des conversions des doubles vers les complexes lors du passage de
paramètres.
 *
 * 2) il aurait également possible de définir les 3 fonctions suivantes
 * friend complex operator+ (const complex & left, const complex & right)
 * friend complex operator+ (const complex & left, double right)
 * friend complex operator+ (double left, const complex & right)
 * Cela aurait eu pour mérite de ne pas solliciter de conversions des doubles vers les complexes
 * lors du passage de paramètres.
 */

```

```

/** MEME RAISONNEMENT QUE POUR LA SURCHARGE DE +
 */

```

```

complex operator- (const complex& other) const
{
    return complex(m_re - other.m_re, m_im - other.m_im);
}

```

```

complex operator- (const double val) const
{
    return complex(m_re - val, m_im);
}

```

```

friend complex operator- (const double left, const complex& right)
{
    return complex(left - right.m_re, -right.m_im);
}

```



```
/** #### SURCHARGES DES OPERATEURS DE PRE ET DE POST-INCREMENTATION #####
```

```
*/
```

```
complex & operator++ ()
```

```
{  
    ++m_re;  
    return *this;  
}
```

```
complex operator++ (int)
```

```
{  
    complex temp(*this);  
    m_re++;  
    return temp;  
}
```

```
complex& operator-- ()
```

```
{  
    --m_re;  
    return *this;  
}
```

```
complex operator-- (int)
```

```
{  
    complex temp(*this);  
    --m_re;  
    return temp;  
}
```

```
/** ##### SURCHARGE DES OPERATEURS BOOLEENS == et != #####
```

```
*/
```

```
/** RAISONNEMENT ANALOGUE A CELUI DE LA SURCHARGE DE L'OPERATEUR +  
 * pour les paramètres des fonctions
```

```
*/
```

```
bool operator== (const complex &other) const
```

```
{  
    return m_re == other.m_re && m_im == other.m_im;  
}
```

```
bool operator== (const double val) const
```

```
{  
    return m_re == val && m_im == 0.;  
}
```

```
friend bool operator== (const double left, const complex& right)
```

```
{  
    return left == right.m_re && right.m_im == 0.;  
}
```

```
bool operator!= (const complex &other) const
```

```
{
```

```

    return m_re != other.m_re || m_im != other.m_im;
}

bool operator!=(const double val) const
{
    return m_re != val || m_im != 0.;
}

friend bool operator!=(const double left, const complex& right)
{
    return left != right.m_re || right.m_im != 0.;
}

/** ##### SURCHARGE DES OPERATEURS D'ENTREE-SORTIE #####
*/

friend istream & operator>>(istream & is, complex & z)
{
    cout << "partie reelle : ";
    is >> z.m_re;
    cout << "partie imaginaire : ";
    is >> z.m_im;

    return is;
}

friend ostream & operator<<(ostream & os, const complex & z)
{
    os << '(' << z.m_re << ',' << z.m_im << ')';
    return os;
}
};

#endif

```

FICHIER complex.cpp

```

#include "complex.h"

/** Les deux constantes imaginaires purs i et j
 * tels que  $i^2 = j^2 = -1$ 
 * appels au constructeur à 2 paramètres
 */
const complex complex::i(0., 1.);
const complex complex::j(0., 1.);

```

FICHIER main.cpp

```
#include <iostream>
#include <iomanip>
#include "complex.h"

using namespace std;

int main()
{
    cout << "LES NOMBRES COMPLEXES" << endl;

    // pour fixer la précision à 0,01 près
    cout << fixed << setprecision(2) << endl;

    cout << "\n TESTS DE TOUS LES CONSTRUCTEURS" << endl;
    complex z0 = complex(0, 0);
    complex z1 = complex();
    complex z2;

    complex z3 = complex(3);
    complex z4(5);

    complex z5 = complex(4,7);

    cout << "\n TESTS DU CONSTRUCTEUR DE COPIE GENERE AUTOMATIQUEMENT PAR
LE COMPILATEUR " << endl;
    complex z6 = complex(z5);
    complex z7 = z5;

    cout << "\n TESTS DE LA SURCHARGE DE L'OPERATEUR <<" << endl;
    cout << "z1 = " << z1 << endl;
    cout << "z2 = " << z2 << endl;
    cout << "z3 = " << z3 << endl;
    cout << "z4 = " << z4 << endl;
    cout << "z5 = " << z5 << endl;
    cout << "z6 = " << z6 << endl;
    cout << "z7 = " << z7 << endl;

    cout << "\n TESTS DE LA SURCHARGE DE L'OPERATEUR >>" << endl;
    complex z8;
    cout << "Saisie de z8" << endl;
    cin >> z8;
    cout << "z8 = " << z8 << endl;

    cout << "\n TEST DE L'OPERATEUR DE TRANSTYPAGE d'un double vers complex" << endl;
    double x = 666.777;
    z5 = x;
    cout << "z5 = 666.777 donc z5 = " << z5 << endl;

    cout << "\n TEST OPERATEUR DE TRANSTYPAGE d'un complex en un double" << endl;
    double y;
```

```

y = z5;
cout << "y = z5 avec z5 = " << z5 << endl;
cout << " y = " << y << endl;

cout << "\n TEST DU CONJUGUE D'UN NOMBRE COMPLEXE a + i b est a - i b" << endl;
complex z6conjugate = z6.conjugate();
cout << "z6conjugue = " << z6conjugate << endl;

cout << "\n TEST DU MODULE D'UN NOMBRE COMPLEXE" << endl;
double z6module = z6.module();
cout << "z6module = " << z6module << endl;

cout << "\n TESTS DE LA SURCHARGE DE L'OPERATEUR = " << endl;
// Appel de complex & operator= (const complex & other) généré automatiquement par le
compilateur
cout << " z6 = z8 avec z8 = " << z8 << endl;
z6 = z8;
cout << "z6 = " << z6 << endl;
cout << " z5 = z3 = z8 avec z8 = " << z8 << endl;
z5 = z3 = z8;
cout << "z3 = " << z3 << endl;
cout << "z5 = " << z5 << endl;
// auto-affectation
z5 = z5 = z5;

// Appel de complex & operator= (const double val)
cout << " z1 = 2014" << endl;
z1 = 2014;
cout << "z1 = " << z1 << endl;

cout << "\n TESTS DE LA SURCHARGE DES OPERATEURS +=, -=, *=, /= " << endl;
cout << "z3 = " << z3 << endl;
cout << "z3 += " << z6 << endl;
z3 += z6;
cout << "z3 = " << z3 << endl;
cout << "z3 += " << 666 << endl;
z3 += 666;
cout << "z3 = " << z3 << endl;

cout << "\n TESTS DE LA SURCHARGE DES OPERATEURS +, -, *, /" << endl;
complex z9, z10, z11, z12, z13;
z9 = z5 + z8;
z10 = z5 - z8;
z11 = z5 * z8;
z12 = z5 / z8;

cout << z9 << " = " << z5 << " + " << z8 << endl;
cout << z10 << " = " << z5 << " - " << z8 << endl;
cout << z11 << " = " << z5 << " * " << z8 << endl;
cout << z12 << " = " << z5 << " / " << z8 << endl;

/** CE CODE FAIT PLANter LE PROGRAMME; cf assert dans les sources

```

```

complex zero;
z13 = z5 / zero;
z13 = z5 / 0.0;
*/

```

```

cout << "\n TESTS DE L' ASSOCIATIVITE GAUCHE DE LA SURCHARGE DE L'
OPERATEUR +" << endl;
z10 = z5 + z6 + z9;
cout << z10 << " = " << z5 << " + " << z6 << " + " << z9 << endl;

```

```

cout << "\n TESTS OPERATEURS DE PRE-INCREMENTATION ET DE POST
INCREMENTATION" << endl;

```

```

/** Rappel
*/
int u = 3, v = 12;
++u;
cout << u << endl; // u est pré-incrémenté et vaut 4
++u = v; // u est pré-incrémenté et vaut 5, puis vaut 12 suite à l'affectation
cout << u << endl; // u vaut 12
// u++ = v; NE COMPILE PAS car u++ n'est pas une lvalue
v = u++; // v vaut u soit 12, puis u est post-incrémenté, u vaut 13
cout << u << " " << v << endl; // u vaut 13
v = ++u; // u est pré-incrémenté et vaut 14
cout << u << " " << v << endl; // v aut 14

```

```

cout << z11 << " = " << z1 << endl;
cout << " ++z11 ==> z11 <==> " << ++z11 << " ==> " << z11 << endl;
cout << " z11++ ==> z11 <==> " << z11++ << " ==> " << z11 << endl;
z10 = ++z1;
cout << " z10 = ++z11 ==> z10 = " << z10 << " ==> z11 = " << z11 << endl;
z10 = z11++;
cout << " z10 = z11++ ==> z10 = " << z10 << " ==> z11 = " << z11 << endl;

```

```

complex z14 = complex(12,24);
cout << "z14 = " << z14 << endl;
z14++ = z10;
cout << " z14++ = z10 ==> z10 = " << z10 << " ==> z14 = " << z14 << endl;
cout << "z12 = " << z12 << endl;
++z12 = z10; // z2 est pré-incrémenté puis sa valeur écrasée par celle de z10
cout << " ++z12 = z10 ==> z10 = " << z10 << " ==> z12 = " << z12 << endl;

```

```

return 0;
}

```

TRACE D'EXECUTION (sous Windows)

```
C:\INFORMATIQUE\0 Enseignements\0 UMLV M2 C++\M2 Info C++\TD1\complex\bin\Debug\co...
CONSTRUCTEUR A DEUX PARAMETRES
CONSTRUCTEUR A DEUX PARAMETRES
LES NOMBRES COMPLEXES

TESTS DE TOUS LES CONSTRUCTEURS
CONSTRUCTEUR PAR DEFALT
CONSTRUCTEUR PAR DEFALT
CONSTRUCTEUR PAR DEFALT
CONSTRUCTEUR DE CONVERSION : TRANSTYPAGE DOUBLE --> COMPLEXE
CONSTRUCTEUR DE CONVERSION : TRANSTYPAGE DOUBLE --> COMPLEXE
CONSTRUCTEUR A DEUX PARAMETRES

TESTS DU CONSTRUCTEUR DE COPIE GENERE AUTOMATIQUEMENT PAR LE COMPILATEUR

TESTS DE LA SURCHARGE DE L'OPERATEUR <<
z1 = <0.00,0.00>
z2 = <0.00,0.00>
z3 = <3.00,0.00>
z4 = <5.00,0.00>
z5 = <4.00,7.00>
z6 = <4.00,7.00>
z7 = <4.00,7.00>

TESTS DE LA SURCHARGE DE L'OPERATEUR >>
CONSTRUCTEUR PAR DEFALT
Saisie de z8
partie reellev : 5 10
partie imaginaire : z8 = <5.00,10.00>

TEST DE L'OPERATEUR DE TRANSTYPAGE d'un double vers complex
z5 = 666.777 donc z5 = <666.78,0.00>

TEST OPERATEUR DE TRANSTYPAGE d'un complex en un double
TRANSTYPAGE COMPLEXE --> DOUBLE
y = z5 avec z5 = <666.78,0.00>
y = 666.78

TEST DU CONJUGUE D'UN NOMBRE COMPLEXE a + i b est a - i b
CONSTRUCTEUR A DEUX PARAMETRES
z6conjugue = <4.00,-7.00>

TEST DU MODULE D'UN NOMBRE COMPLEXE
z6module = 8.06

TESTS DE LA SURCHARGE DE L'OPERATEUR =
z6 = z8 avec z8 = <5.00,10.00>
z6 = <5.00,10.00>
z5 = z3 = z8 avec z8 = <5.00,10.00>
z3 = <5.00,10.00>
z5 = <5.00,10.00>
z1 = 2014
z1 = <2014.00,0.00>
```

```
C:\INFORMATIQUE\0 Enseignements\0 UMLV M2 C++\M2 Info C++\TD1\complex\bin\Debug\co...
TESTS DE LA SURCHARGE DES OPERATEURS +, -, *, /=
z3 = <5.00,10.00>
z3 += <5.00,10.00>
z3 = <10.00,20.00>
z3 += 666
z3 = <676.00,20.00>

TESTS DE LA SURCHARGE DES OPERATEURS +, -, *, /
CONSTRUCTEUR PAR DEFALT
CONSTRUCTEUR PAR DEFALT
CONSTRUCTEUR PAR DEFALT
CONSTRUCTEUR PAR DEFALT
CONSTRUCTEUR PAR DEFALT
CONSTRUCTEUR A DEUX PARAMETRES
CONSTRUCTEUR PAR DEFALT
CONSTRUCTEUR A DEUX PARAMETRES
CONSTRUCTEUR DE CONVERSION : TRANSTYPAGE DOUBLE --> COMPLEXE
<10.00,20.00> = <5.00,10.00> + <5.00,10.00>
<0.00,0.00> = <5.00,10.00> - <5.00,10.00>
<-75.00,100.00> = <5.00,10.00> * <5.00,10.00>
<1.00,0.00> = <5.00,10.00> / <5.00,10.00>

TESTS DE L' ASSOCIATIVITE GAUCHE DE LA SURCHARGE DE L' OPERATEUR +
CONSTRUCTEUR A DEUX PARAMETRES
CONSTRUCTEUR A DEUX PARAMETRES
<20.00,40.00> = <5.00,10.00> + <5.00,10.00> + <10.00,20.00>

TESTS OPERATEURS DE PRE-INCREMENTATION ET DE POST INCREMENTATION
4
12
13 12
14 14
<-75.00,100.00> = <2014.00,0.00>
++z11 ==> z11 <==> <-74.00,100.00> ==> <-74.00,100.00>
z11++ ==> z11 <==> <-74.00,100.00> ==> <-73.00,100.00>
z10 = ++z11 ==> z10 = <2015.00,0.00> ==> z11 = <-73.00,100.00>
z10 = z11++ ==> z10 = <-73.00,100.00> ==> z11 = <-72.00,100.00>
CONSTRUCTEUR A DEUX PARAMETRES
z14 = <12.00,24.00>
z14++ = z10 ==> z10 = <-73.00,100.00> ==> z14 = <13.00,24.00>
z12 = <1.00,0.00>
++z12 = z10 ==> z10 = <-73.00,100.00> ==> z12 = <-73.00,100.00>
```