LENZEN Martial
Etudiant en licence de Mathématiques

Année universitaire : 2002/2003

PROJET D'INFORMATIQUE

Sujet donné les jeudi 12 et Vendredi 13 décembre 2002
Soutenance : durant les séances de TP des deux premières semaines de janvier 2003

Détails sur la soutenance :

Ce projet ne donnera pas lieu à un rapport écrit, mais uniquement à un passage en machine où il sera demandé une démonstration du programme et une vis ualisation des sources. Le projet est individuel et sera complété par une question supplémentaire différente pour chaque groupe de TP. Cette dernière devra être réalisée en 1 heure.
La note ne comptera pour un tiers de la note finale que si elle est supérieure à la note de l'examen de janvier.

Objectif :

Réaliser une petite calculatrice utilisant la notation infixe (habituelle, pas la notation polonaise inversée), mais travaillant sur des nombres exacts et de taille illimitée (ou du moins très grandes).

SOMMAIRE

Question 1 : **On réalisera une classe de nombres entiers très grands. Pour cela, on décomposera les nombres en leurs chiffres individuels que l'on stockera d'abord dans un tableau de chiffres. Puis faire une deuxième version utilisant une liste chaînée de chiffres. On écrira toutes les fonctions habituelles sur ces grands entiers (constructeurs à partir d'une entier, d'une chaîne de caractères, affichage, saisie, opérateurs +, –, *, /, % (modulo), << et >>). On tiendra également compte du cas des grands entiers invalides comme on l'a vu pour la classe valeur. Prévoir également un petit programme `main` permettant de tester ces grands entiers.**

GrandEntier.h

```
//  * * * * * * * * * * * * * * * GrandEntier.h * * * * * * * * * * * * * * *
//  *   Fichier : GrandEntier.h                                               *
//  *   Date    : 19.12.2002                                                  *
//  *   Copyright : (C) 2002 by LENZEN Martial. Tous droits réservés          *
//  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

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

#define  MAX   1024

class GrandEntier
{    int GE[MAX];
     int dim;
     bool signe;   // true -> + ; false -> -
     bool valide;
    public :
                GrandEntier() { GE[0]=0; dim=1; signe=true;
valide=false;}
                GrandEntier(int n);
                GrandEntier(char buf[MAX]);
     GrandEntier operator+(GrandEntier b);
     GrandEntier operator-(GrandEntier b);
     GrandEntier operator*(GrandEntier b);
     GrandEntier operator/(GrandEntier b);
     GrandEntier operator%(GrandEntier b);
     GrandEntier operator^(GrandEntier b);
     GrandEntier operator-();
     GrandEntier operator~();    // valeur absolue
     GrandEntier DecalageGauche(int PuissanceDeDix);
     GrandEntier DecalageDroite(int PuissanceDeDix);
     GrandEntier Nettoyage();
     bool        operator<(GrandEntier b);


     // fonctions amies :
     friend ostream& operator<<(ostream& o, GrandEntier  P);
     friend istream& operator>>(istream& i, GrandEntier& P);
}; // fin de la classe 'GrandEntier'


// ******************** Déclaration des constructeurs
*******************
```



```
GrandEntier::GrandEntier(int n)
{    int i=0, TMP[MAX]; signe = true;
     if (n < 0)           { n = -n; signe = false; }
     else if (n == 0)     { GE[0] = 0; i++;        }
     while (n != 0)
     {    GE[i] = n%10;
          n /= 10;
          i++;
     } dim = i;
     // Le nombre n est stocké 'à l'envers' dans GE...
     valide = true;
}

GrandEntier::GrandEntier(char buf[MAX])
{    int i=0, j=0, TMP[MAX];
     valide = true;
     if (buf[0] == '-')    { signe = false; i++; j++; }
     else                  signe = true;
     while (buf[i] != '\0')
     {    if ((int(buf[i])>int('9'))||(int(buf[i])<int('0'))) valide =
false;
          GE[i-j] = int(buf[i]) - int('0'); i++;
     } dim = i;
     for (i=0; i<dim; i++)        TMP[i] = GE[dim-i-1];
     for (i=0; i<dim; i++)        GE[i] = TMP[i];
}


// ******************** Déclaration des fonctions amies
*******************

ostream& operator<<(ostream& o, GrandEntier P)
{    if (P.valide)
     {    if (P.signe == false)  o << "-";
          for (int i=0; i<P.dim; i++)     o << P.GE[P.dim-i-1];
     } else o << "<INVALIDE>";
     // on affiche du dernier au 1er pour avoir le nombre dans le bon sens
!
     return o;
}

istream& operator>>(istream& i, GrandEntier& P)
{    char buf[MAX];
     cout << "Entrer un entier : ";
     cin.getline(buf, MAX);
          int j=0;
          P.valide = true;
          while (buf[j] != '\0')   j++;
          if (buf[0] == '-') { j--; P.dim=j; j=1; P.signe = false; }
          else               { P.dim=j; j=0; P.signe = true; }
          while (buf[j] != '\0')
          {    if ((int(buf[j])>int('9')) || (int(buf[j])<int('0')))
                    P.valide = false;
               if (P.signe) P.GE[P.dim-j-1] = int(buf[j]) - int('0');
               else         P.GE[P.dim-j] = int(buf[j]) - int('0');
               j++;
          }
     return i;
}
```

```
// ************ Déclaration des fonctions membres non-operator
************

GrandEntier GrandEntier::DecalageGauche(int PuissanceDeDix)
{   GrandEntier RES, NUL(0);
    RES.valide = valide;
    if (dim-PuissanceDeDix < 1)   return NUL;
    else                          RES.dim = dim-PuissanceDeDix;
    for (int i=0; i<dim-PuissanceDeDix; i++)
    if (i+1 < dim)
        for (int j=i+1; j<dim; j++) RES.GE[j-i-1] = GE[j];
    else return NUL;

    RES.signe = signe;
    return RES;
}

GrandEntier GrandEntier::DecalageDroite(int PuissanceDeDix)
{   GrandEntier RES;
    RES.valide = valide;
    for (int i=0; i<PuissanceDeDix; i++)  RES.GE[i]=0;
    for (int i=0; i<dim; i++)             RES.GE[i+PuissanceDeDix] =
GE[i];
    RES.dim = dim+PuissanceDeDix; RES.signe = signe;
    return RES;
}

GrandEntier GrandEntier::Nettoyage()
{   GrandEntier RES;
    int i=dim-1; RES.dim = dim;
    RES.valide = valide;
    while ((GE[i] == 0) && (i>0))     { RES.dim--; i--; }
    if (RES.dim == 1)   RES.GE[0] = GE[0];
    else
        while (i>=0)      { RES.GE[i] = GE[i]; i--; }
    RES.signe = ((RES.dim == 1) && (RES.GE[0] == 0) ? true : signe);
    return RES;
}


// ************** Déclaration des fonctions membres operator
**************

bool GrandEntier::operator<(GrandEntier b)
{   int i=0;
         if (!signe && b.signe)  return true;
    else if (signe && !b.signe)  return false;
    else if (signe && b.signe)
    {        if (dim < b.dim)  return true;
        else if (dim > b.dim)  return false;
        else
        {   while ((GE[dim-i-1] == b.GE[b.dim-i-1]) && (i<dim-1))   i++;
            return (GE[dim-i-1] < b.GE[b.dim-i-1]);
        }
    }
    else
    {        if (dim < b.dim)  return false;
        else if (dim > b.dim)  return true;
        else
        {   while ((GE[dim-i-1] == b.GE[b.dim-i-1]) && (i<dim-1))   i++;
            return (GE[dim-i-1] > b.GE[b.dim-i-1]);
```



```
        }
    }
}

GrandEntier GrandEntier::operator-()
{   GrandEntier RES = *this;
    RES.signe = !signe;
    return RES;
}

GrandEntier GrandEntier::operator~()
{   GrandEntier NUL(0), RES=*this;
    if (NUL < *this)    return RES;
    else                return -RES;
}

GrandEntier GrandEntier::operator+(GrandEntier b)
{   int retenue=0;
    GrandEntier RES;

         if ( signe &&  b.signe)   RES.signe = true;
    else if (!signe && !b.signe)   RES.signe = false;
    else if ( signe && !b.signe)   return (*this-(-b));
    else                           return (b-(-*this));

    if (*this < b)    return (b + *this);
    else
    {   RES.dim = dim;
        for (int i=0; i<b.dim; i++)
        {   RES.GE[i] = (GE[i]+b.GE[i]+retenue)%10;
            retenue = (GE[i]+b.GE[i]+retenue)/10;
        }
        for (int i=b.dim; i<dim; i++)
        {   RES.GE[i] = (GE[i]+retenue)%10;
            retenue = (GE[i]+retenue)/10;
        }
        if (retenue == 1)   { RES.GE[dim]=1; RES.dim++; }
    }

    RES.valide = (valide &&  b.valide);
    return RES;
}

GrandEntier GrandEntier::operator-(GrandEntier b)
{   int retenue=0;
    GrandEntier RES;

         if ( signe && !b.signe)    return (*this + (-b));
    else if (!signe &&  b.signe)    return -((-*this) + b);
    else if (!signe && !b.signe)    return ((-b)-(-*this));

    if (*this < b)   return -(b - *this);
    else
    {   RES.dim = dim;
        for (int i=0; i<b.dim; i++)
        {   RES.GE[i] = GE[i]-b.GE[i]-retenue;
            retenue = 0;
            if (RES.GE[i] < 0)  { RES.GE[i] += 10; retenue = 1; }
        }
        for (int i=b.dim; i<dim; i++)
        {   RES.GE[i] = GE[i]-retenue;
```

```
            retenue = 0;
            if (RES.GE[i] < 0)  { RES.GE[i] = 9; retenue = 1; }
        }
        RES.signe = true;
    }
    RES.valide = (valide &&  b.valide);

    return RES.Nettoyage();
}

GrandEntier GrandEntier::operator*(GrandEntier b)
{   int retenue=0;
    GrandEntier RES, TMP[MAX];

    for (int j=0; j<b.dim; j++)
    {   TMP[j].dim = dim + j;
        for (int i=0; i<j; i++)  TMP[j].GE[i] = 0;
        for (int i=0; i<dim; i++)
        {   TMP[j].GE[i+j] = (b.GE[j]*GE[i]+retenue)%10;
            retenue = (b.GE[j]*GE[i]+retenue)/10;
        }
        if (retenue > 0)   { TMP[j].GE[TMP[j].dim]=retenue; TMP[j].dim++;
}
        retenue = 0;
    }

    for (int i=0; i<b.dim; i++)  RES = RES + TMP[i];

    RES.signe = ((signe &&  b.signe) || (!signe && !b.signe));
    RES.valide = (valide &&  b.valide);

    return RES.Nettoyage();
}

GrandEntier GrandEntier::operator/(GrandEntier b)
{   GrandEntier X=*this, NUL(0), RES;
    int m=0, P=dim-b.dim-1;

    if (valide && b.valide)  RES.valide = true;
    else return (valide ? b : *this);

         if (~(*this) < ~b)          return NUL;
    else if (signe  && !b.signe)   { return -(*this/(-b)); }
    else if (!signe &&  b.signe)   { return -(-*this/b); }
    else if (!signe && !b.signe)   { return -(-*this/(-b)); }
    if (P < 2)
    {   while (!(X < b))      { m++; X = X-b; }
        return GrandEntier(m);
    }

    GrandEntier n = DecalageGauche(P);
    GrandEntier M=n/b;
    RES = M.DecalageDroite(P);
    RES = RES+(*this-b*RES)/b;
    return RES;
}

GrandEntier GrandEntier::operator%(GrandEntier b)
{   GrandEntier RES=*this;
    RES.valide = (valide && b.valide);
    return RES-(RES/b)*b;
```



```
}

GrandEntier GrandEntier::operator^(GrandEntier b)
{   GrandEntier NUL(0), RES(0), TMP=~*this, Q;
    if (!valide || !b.valide || b<NUL || ((!(*this<NUL) &&
            !(NUL<*this)) && (!(b<NUL) && !(NUL<b))))
    {   NUL.valide = false;
        return NUL;
    }

    if (!(b<GrandEntier(1)) && !(GrandEntier(1)<b))  return *this;
    RES.signe = (signe || b.GE[dim-1]%2==0);
    RES.valide = true;

    if (b.GE[dim-1]%2==0)     { Q = TMP^(b/2); RES = Q*Q; }
    else                      { Q = TMP^((b-1)/2); RES = TMP*Q*Q; }
    return RES;
}
```

Main correspondant :

```
#include <iostream.h>
#include <stdlib.h>
#include "GrandEntier.h"

int main()
{   GrandEntier a, b;
    GrandEntier c("4567"), d(4567);

    cin >> a; cout << "a = " << a << endl;
    cin >> b; cout << "b = " << b << endl << endl;
    cout << "a / 100 = " << a.DecalageGauche(2) << endl;
    cout << "d * 100 = " << d.DecalageDroite(2) << endl;
    cout << "a / 100000 = " << a.DecalageGauche(5) << endl;
    cout << "d * 100000 = " << d.DecalageDroite(5) << endl << endl;

    cout << "a + b = " << a+b << endl;
    cout << "a - b = " << a-b << endl;
    cout << "a * b = " << a*b << endl;
    cout << "a / b = " << a/b << endl;
    cout << "a % b = " << a%b << endl;
    cout << "a ^ b = " << (a^b) << endl;

    return 0;
}
```

Version avec des listes chaînées :

GrandEntier2.h

```
//  * * * * * * * * * * * * * * * GrandEntier2.h  * * * * * * * * * * * * * *
//  *   Fichier : GrandEntier2.h                                              *
//  *   Date    : 23.12.2002                                                  *
//  *   Copyright : (C) 2002 by LENZEN Martial. Tous droits réservés.         *
//  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <stdlib.h>
```

```
#include <math.h>

#define  MAX   1024

struct Chiffre
{   int c;
    struct Chiffre *suivant;
};

class GrandEntier
{   struct Chiffre *premier;
    bool signe;       // true -> +  ;  false -> -
    bool valide;

   public :
               GrandEntier();
               GrandEntier(int n);
               GrandEntier(char buf[MAX]);
      GrandEntier operator+(GrandEntier b);
      GrandEntier operator-(GrandEntier b);
      GrandEntier operator*(GrandEntier b);
      GrandEntier operator/(GrandEntier b);
      GrandEntier operator%(GrandEntier b);
      GrandEntier operator^(GrandEntier b);
      GrandEntier operator-();
      GrandEntier operator~();    // valeur absolue
      GrandEntier DecalageGauche(int PuissanceDeDix);
      GrandEntier DecalageDroite(int PuissanceDeDix);
      GrandEntier Nettoyage();
      GrandEntier Inverser();
      bool        operator<(GrandEntier b);

    friend ostream& operator<<(ostream& o, GrandEntier  P);
    friend istream& operator>>(istream& i, GrandEntier& P);
};  // fin de la classe 'GrandEntier'


// ****************** Déclaration des constructeurs *******************

GrandEntier::GrandEntier()
{   Chiffre *TMP;
    TMP = new Chiffre;
    valide = false;
    signe = true;
        TMP->c = 0; TMP->suivant = NULL;
    premier = TMP;
}

GrandEntier::GrandEntier(int n)
{    GrandEntier X;
     Chiffre *t, *CHIFFRE;
     signe = true;
     if (n <0 )  { n = -n; signe = false; }
     if (n == 0)
     {  Chiffre *TMP; TMP = new Chiffre;
        valide = true;
        TMP->c = 0; TMP->suivant = NULL;
        premier = TMP;
     }
     else premier = NULL;
     while (n > 0)
```



```
     {   CHIFFRE = new Chiffre;
         CHIFFRE->suivant = premier;
         CHIFFRE->c = n%10; n/=10;
         premier = CHIFFRE;
     }

     *this = Inverser();
     valide = true;
}

GrandEntier::GrandEntier(char buf[MAX])
{   int i=0; Chiffre *CHIFFRE;
    valide = true;

    if (buf[0] == '-')    { signe = false; i++; }
    else                    signe = true;

    premier = NULL;
    while (buf[i] != '\0')
    {   if ((int(buf[i])>int('9'))||(int(buf[i])<int('0'))) valide =
            false;
        CHIFFRE = new Chiffre;  CHIFFRE->suivant = premier;
        CHIFFRE->c = int(buf[i]) - int('0');
        i++; premier = CHIFFRE;
    }
}


// ****************** Déclaration des fonctions amies ******************

ostream& operator<<(ostream& o, GrandEntier P)
{   Chiffre *CHIFFRE, *D;
    if (!P.valide)
    {   o << "<INVALIDE>";
        return o;
    }

    P = P.Inverser();
    if (!P.signe)   o << "-";
    CHIFFRE = new Chiffre; CHIFFRE = P.premier;
    while (CHIFFRE != NULL)
    {   o << CHIFFRE->c;
        CHIFFRE = CHIFFRE->suivant;
    } return o;
}

istream& operator>>(istream& i, GrandEntier& P)
{   char buf[MAX]; int j=0; Chiffre *CHIFFRE;
    cout << "Entrer un entier : ";
    cin.getline(buf, MAX);
        P.valide = true;
        if (buf[0] == '-')  { j=1; P.signe=false; }
        else                { j=0; P.signe=true;  }

        P.premier = NULL;
        while (buf[j] != '\0')
        {   if ((int(buf[j])>int('9')) || (int(buf[j])<int('0')))
                P.valide = false;
            CHIFFRE = new Chiffre;
            CHIFFRE->c = int(buf[j])-int('0');
            CHIFFRE->suivant = P.premier;
```

```
            P.premier = CHIFFRE; j++;
        }
    return i;
}

// ********** Declaration des fonctions membres non-operator **********

GrandEntier GrandEntier::Inverser()
{   GrandEntier TMP; Chiffre *CHIFFRE, *D;
    TMP.premier = NULL;
    CHIFFRE = new Chiffre;
    CHIFFRE = premier;
    while (CHIFFRE != NULL)
    {   D = new Chiffre; D->c = CHIFFRE->c;
        D->suivant = TMP.premier; TMP.premier = D;
        CHIFFRE = CHIFFRE->suivant;
    }
    TMP.signe = signe; TMP.valide = valide;
    return TMP;
}

GrandEntier GrandEntier::Nettoyage()
{   GrandEntier RES, TMP; Chiffre *CHIFFRE, *D;
    TMP.premier = NULL;
    CHIFFRE = new Chiffre; CHIFFRE = premier;

    while (CHIFFRE != NULL)
    {   D = new Chiffre; D->c = CHIFFRE->c;
        D->suivant = TMP.premier; TMP.premier = D;
        CHIFFRE = CHIFFRE->suivant;
    }
    while (TMP.premier->c == 0 && TMP.premier->suivant != NULL)
        TMP.premier = TMP.premier->suivant;

    RES.premier = NULL;
    CHIFFRE = new Chiffre; CHIFFRE = TMP.premier;
    while (CHIFFRE != NULL)
    {   D = new Chiffre; D->c = CHIFFRE->c;
        D->suivant = RES.premier; RES.premier = D;
        CHIFFRE = CHIFFRE->suivant;
    }

    if (RES.premier->c == 0 && RES.premier->suivant == NULL)
        RES.signe = true;
    else RES.signe = signe;
    RES.valide = valide;
    return RES;
}

GrandEntier GrandEntier::DecalageGauche(int PuissanceDeDix)
{   GrandEntier RES = *this, NUL(0);
    for (int i=0; i<PuissanceDeDix; i++)
    {   if (RES.premier->suivant != NULL)
            RES.premier = RES.premier->suivant;
        else return NUL;
    }
}

GrandEntier GrandEntier::DecalageDroite(int PuissanceDeDix)
{   GrandEntier RES = *this; Chiffre *CHIFFRE;
    for (int i=0; i<PuissanceDeDix; i++)
```



```
    {   CHIFFRE = new Chiffre; CHIFFRE->c = 0;
        CHIFFRE->suivant = RES.premier;
        RES.premier = CHIFFRE;
    }
    return RES;
}

// ************ Declaration des fonctions membres operator ************

GrandEntier GrandEntier::operator+(GrandEntier b)
{   int retenue = 0; GrandEntier RES, TMP; Chiffre *x, *y, *z;

         if ( signe &&  b.signe)  RES.signe = true;
    else if (!signe && !b.signe)  RES.signe = false;
    else if ( signe && !b.signe)  return (*this-(-b));
    else                          return (b-(-*this));

    if (~*this < ~b) return (b+*this);
    TMP.premier = NULL;
    x = new Chiffre; x = premier;
    y = new Chiffre; y = b.premier;

    while (y != NULL)
    {   z = new Chiffre;
        z->c = (x->c+y->c+retenue)%10;
        retenue = (x->c+y->c+retenue)/10;
        z->suivant = TMP.premier; TMP.premier = z;
        x = x->suivant; y = y->suivant;
    }
    while (x != NULL)
    {   z = new Chiffre;
        z->c = (x->c+retenue)%10;
        retenue = (x->c+retenue)/10;
        z->suivant = TMP.premier; TMP.premier = z;
        x = x->suivant;
    }
    if (retenue == 1)
    {   z = new Chiffre; z->c = 1;
        z->suivant = TMP.premier; TMP.premier = z;
    }

    TMP.valide = true; TMP.signe = RES.signe;
    RES = TMP.Inverser();
    RES.valide = (valide && b.valide);
    return RES;
}

GrandEntier GrandEntier::operator-(GrandEntier b)
{   int retenue=0; GrandEntier RES, TMP; Chiffre *x, *y, *z;

    if ( signe && !b.signe)     return (*this+(-b));
    if (!signe &&  b.signe)     return -((-*this)+b);
    if (!signe && !b.signe)     return ((-b)-(-*this));

    if (*this < b)   return -(b-*this);

    TMP.premier = NULL;
    x = new Chiffre; x = premier;
    y = new Chiffre; y = b.premier;

    while (y != NULL)
```

```
    {   z = new Chiffre; z->c = x->c - y->c - retenue; retenue = 0;
        if (z->c < 0)   { z->c += 10; retenue = 1; }
        z->suivant = TMP.premier; TMP.premier = z;
        x = x->suivant; y = y->suivant;
    }
    while (x != NULL)
    {   z = new Chiffre; z->c = x->c - retenue; retenue = 0;
        if (z->c < 0)   { z->c = 9; retenue = 1; }
        z->suivant = TMP.premier; TMP.premier = z;
        x = x->suivant;
    }

    RES.signe = true; TMP.valide = true;
    RES = TMP.Inverser();
    RES.valide = (valide && b.valide);
    return RES.Nettoyage();
}

GrandEntier GrandEntier::operator*(GrandEntier b)
{   int retenue=0, n=0; GrandEntier RES, TMP; Chiffre *x, *y, *t;
    RES.valide = (valide && b.valide);

    TMP.valide = true; TMP.signe = true;
    x = new Chiffre; y = new Chiffre;
    x = premier; y = b.premier;

    while (y != NULL)
    {   TMP.premier = NULL;
        for (int i=0; i<n; i++)
        {   t = new Chiffre; t->c = 0;
            t->suivant = TMP.premier;     TMP.premier = t;
        }

        while (x != NULL)
        {   t = new Chiffre;
            t->c = (x->c * y->c + retenue)%10;
            retenue = (x->c * y->c + retenue)/10;
            t->suivant = TMP.premier; TMP.premier = t;
            x = x->suivant;
        } x = premier;

        if (retenue > 0)
        {   t = new Chiffre; t->c = retenue;
            t->suivant = TMP.premier; TMP.premier = t;
        } retenue = 0; n++; y = y->suivant;

        TMP = TMP.Inverser();
        RES = RES + TMP;
    }

    RES.signe = ((signe && b.signe) || (!signe && !b.signe));
    return RES.Nettoyage();
}

GrandEntier GrandEntier::operator/(GrandEntier b)
{   GrandEntier X = *this, NUL(0), RES; int m=0, DIM=0, dim=0;
    Chiffre *x, *y;

    x = new Chiffre; x = premier;
    y = new Chiffre; y = b.premier;
```



```
    while (x != NULL)     { DIM++; x = x->suivant; }
    while (y != NULL)     { dim++; y = y->suivant; }
    int p = DIM-dim-1;

    if (valide && b.valide)       RES.valide = true;
    else                          return (valide ? b : *this);
    if (!(X<NUL) && !(NUL<X))     return NUL;
    if (~*this < ~b)              return NUL;

    if ( signe && !b.signe)     return -(*this/(-b));
    if (!signe &&  b.signe)     return -(-*this/b);
    if (!signe && !b.signe)     return -(-*this/(-b));

    if (p<2)
    {   while (!(X<b))  { m++; X = X-b; }
        return GrandEntier(m);
    }
    GrandEntier n = DecalageGauche(p);
    GrandEntier M = n/b;
    RES = M.DecalageDroite(p);
    RES = RES + (*this-b*RES)/b;
    return RES;
}

GrandEntier GrandEntier::operator%(GrandEntier b)
{   GrandEntier RES = *this;
    RES.valide = (valide && b.valide);
    return RES-(RES/b)*b;
}

GrandEntier GrandEntier::operator^(GrandEntier b)
{   GrandEntier NUL(0), RES(0), TMP=~*this, Q;
    if (!valide || !b.valide || b<NUL || ((!(*this<NUL) &&
            !(NUL<*this)) && (!(b<NUL) && !(NUL<b))))
    {   NUL.valide = false;
        return NUL;
    }

    if (!(b<GrandEntier(1)) && !(GrandEntier(1)<b))      return *this;
    RES.signe = (signe || ((b.premier->c)%2) == 0);
    RES.valide = true;

    if ((b.premier->c)%2 == 0)
    {   Q = TMP^(b/2);
        RES.premier = (Q*Q).premier;
    }
    else
    {   Q = TMP^((b-1)/2);
        RES.premier = (TMP*Q*Q).premier;
    }
    return RES;
}

GrandEntier GrandEntier::operator-()
{   GrandEntier RES = *this;
    RES.signe = !signe;
    return RES;
}

GrandEntier GrandEntier::operator~()
{   GrandEntier RES = *this;
```

```
    return (RES.signe ? RES : -RES);
}

bool GrandEntier::operator<(GrandEntier b)
{   GrandEntier A, B; int i=0, j=0; Chiffre *x, *y;
    if (!signe &&  b.signe)     return true;
    if ( signe && !b.signe)     return false;
    A = Inverser(); B = b.Inverser();
    x = new Chiffre; y = new Chiffre;
    x = premier; y = b.premier;

    while (x != NULL)   // donne la dimension de *this
    {   i++;
        x = x->suivant;
    }
    while (y != NULL)    // donne la dimension de b
    {   j++;
        y = y->suivant;
    }
    if (i<j) return  signe;
    if (j<i) return !signe;
    x = new Chiffre; y = new Chiffre;
    x = A.premier; y = B.premier;

    while (x != NULL)
    {   if (x->c < y->c) return  signe;
        if (x->c > y->c) return !signe;
        x = x->suivant; y = y-> suivant;
    }
    return false;
}
```

Main correspondant :

```
#include <iostream.h>
#include <stdlib.h>
#include "GrandEntier2.h"

int main()
{   GrandEntier a, b;
    GrandEntier c("4567"), d(4567);

    cin >> a; cout << "a = " << a << endl;
    cin >> b; cout << "b = " << b << endl << endl;
    cout << "c / 100 = " << c.DecalageGauche(2) << endl;
    cout << "d * 100 = " << d.DecalageDroite(2) << endl;
    cout << "c / 100000 = " << c.DecalageGauche(5) << endl;
    cout << "d * 100000 = " << d.DecalageDroite(5) << endl << endl;

    cout << "a + b = " << a+b << endl;
    cout << "a - b = " << a-b << endl;
    cout << "a * b = " << a*b << endl;
    cout << "a / b = " << a/b << endl;
    cout << "a % b = " << a%b << endl;
    cout << "a ^ b = " << (a^b) << endl;

    return 0;
}
```


Question 2 : On définira une nouvelle classe de grandes valeurs décimales, en utilisant l'une des deux représentation au choix (tableau de chiffres ou liste chaînée), mais en tenant maintenant compte également des chiffres après la virgule. A nouveau, écrire une fonction main simple permettant de tester ces nombres décimaux.

GrandFlottant.h

```
//  * * * * * * * * * * * * * * GrandFlottant.h * * * * * * * * * * * * * * *
//  *   Fichier : GrandFlottant.h                                             *
//  *   Date    : 19.12.2002                                                  *
//  *   Copyright : (C) 2002 by LENZEN Martial. Tous droits réservés.         *
//  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

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

#define  MAX   1024      // Partie entière et frac. seront de dim <= 1024
#define  NBD   50        // Nbre de chiffres après la , pour la division

class GrandFlottant
{    int GE_IPart[MAX]; int dim1;
     int GE_FPart[MAX]; int dim2;
     bool signe;   // true -> + ; false -> -
     bool valide;
    public :
                    GrandFlottant();
                    GrandFlottant(double n);
                    GrandFlottant(char buf[MAX]);
     GrandFlottant DivisionEntiere(GrandFlottant b);
     GrandFlottant operator+(GrandFlottant b);
     GrandFlottant operator-(GrandFlottant b);
     GrandFlottant operator*(GrandFlottant b);
     GrandFlottant operator/(GrandFlottant b);
     GrandFlottant operator^(GrandFlottant b);
     GrandFlottant operator-();
     GrandFlottant operator~();    // valeur absolue
     GrandFlottant DecalageGauche(int PuissanceDeDix);
     GrandFlottant DecalageDroite(int PuissanceDeDix);
     GrandFlottant Nettoyage();
     GrandFlottant Remplissage(int NbreDeZero);
     bool          Valide();
     bool          operator==(GrandFlottant b);
     bool          operator< (GrandFlottant b);

     friend ostream& operator<<(ostream& o, GrandFlottant  P);
     friend istream& operator>>(istream& i, GrandFlottant& P);
}; // fin de la classe 'GrandFlottant'


// ******************* Déclaration des constructeurs *******************

GrandFlottant::GrandFlottant()
{   GE_IPart[0] = 0;
    GE_FPart[0] = 0;
    dim1 = 1; dim2 = 0;
```

```
    signe = true; valide = false;
}

GrandFlottant::GrandFlottant(double n)
{   int i=0, n_IPart, TMP[MAX];
    double n_FPart;
    signe = true;
    if (n < 0)        { n = -n; signe = false; }
    if (n == 0)
    {   GE_IPart[0]=0; dim1 = 1;
        GE_FPart[0]=0; dim2 = 0;
        signe=true; valide=true;
    }
    else    n_IPart = int(n);
    n_FPart = n-double(n_IPart);

    while (n_IPart != 0)
    {   GE_IPart[i] = n_IPart%10;
        n_IPart /= 10;
        i++;
    } dim1 = i; i=0;

    if (n_FPart != 0)
    {   while (n_FPart != 0)
        {   n_FPart *= 10;
            GE_FPart[i] = int(n_FPart);
            n_FPart = (n_FPart-double(int(n_FPart)) < 1e-2 ? 0 :
                       n_FPart-double(int(n_FPart))); i++;
        }
        dim2 = i;
        for (i=0; i<dim2; i++)   TMP[i] = GE_FPart[dim2-1-i];
        for (i=0; i<dim2; i++)   GE_FPart[i] = TMP[i];
    }
    // Le nombre n est stocké 'à l'envers' dans GE_FPart...
    valide = true;
}

GrandFlottant::GrandFlottant(char buf[MAX])
{   int i=0, j=0, TMP[MAX];
    valide = true;
    if (buf[0] == '-')
    {   signe = false;
        i++; j++;
    } else signe = true;

    while (true)
    {   if ((buf[i] == ',') || (buf[i] == '.') || (buf[i] == '\0'))
        {   if ((signe == false) && (i == 1))  valide = false;
            if ((signe == true)  && (i == 0))  valide = false;
            break;
        }
        if ((int(buf[i])>int('9'))||(int(buf[i])<int('0')))
valide=false;
        GE_IPart[i-j] = int(buf[i]) - int('0');
        i++;
    } dim1 = i;
    for (j=0; j<dim1; j++)  TMP[j] = GE_IPart[dim1-1-j];
    for (j=0; j<dim1; j++)  GE_IPart[j] = TMP[j];
    dim2 = 0;

    if (buf[i] != '\0')
```



```
    {   j = i+1;
        while (buf[j] != '\0')
        {   if ((int(buf[j])>int('9'))||(int(buf[j])<int('0')))
                valide=false;
            GE_FPart[j-i-1] = int(buf[j]) - int('0');
            j++;
        } dim2 = j-i-1;
        for (i=0; i<dim2; i++)  TMP[i] = GE_FPart[dim2-1-i];
        for (i=0; i<dim2; i++)  GE_FPart[i] = TMP[i];
    }
}


// ****************** Déclaration des fonctions amies ******************

ostream& operator<<(ostream& o, GrandFlottant P)
{   if (P.valide)
    {   if (P.signe == false)  o << "-";
        for (int i=0; i<P.dim1; i++)
            o << P.GE_IPart[P.dim1-1-i];
        if (P.dim2 != 0)
        {   o << ",";
            for (int i=0; i<P.dim2; i++)
                o << P.GE_FPart[P.dim2-1-i];
        }
    } else o << "<INVALIDE>";
    // on affiche du dernier au 1er pour avoir le nombre dans le bon sens
!
    return o;
}

istream& operator>>(istream& i, GrandFlottant& P)
{   char buf[MAX];
    cout << "Entrer un flottant : ";
    cin.getline(buf, MAX);
        int k=0, j=0;
        P.valide = true;

        while (true)
        {   if ((buf[j] == ',') || (buf[j] == '.') || (buf[j] == '\0'))
            {   if (j == 0)    P.valide = false;
                break;
            }
            j++;
        }

        P.dim1 = j; P.dim2=0;
        if ((buf[j] == ',') || (buf[j] == '.'))
        {   j++;
            while (buf[j] != '\0')
            {   k++; j++;
            }
            P.dim2 = k;
        } else P.dim2 = 0;

        if (buf[0] == '-') { j=1; P.signe = false; }
        else               { j=0; P.signe = true; }

        for (int l=j; l<P.dim1; l++)
        {   if ((int(buf[l])>int('9')) || (int(buf[l])<int('0')))
                P.valide = false;
```

```
            P.GE_IPart[P.dim1-l-1] = int(buf[l]) - int('0');
        }
        if (P.dim2 != 0)
        {   for (k=1; k<=P.dim2; k++)
            {   if ((int(buf[P.dim1+k]) > int('9'))
                    || (int(buf[P.dim1+k]) < int('0')))
                    P.valide = false;
                P.GE_FPart[P.dim2-k] = int(buf[P.dim1+k]) - int('0');
            }
        }
        if (buf[0] == '-') P.dim1--;        //
    return i;
}


// *********** Déclaration des fonctions membres non-operator ***********

GrandFlottant GrandFlottant::DecalageGauche(int PuissanceDeDix)
{   GrandFlottant RES;

    // Cas où PuissanceDeDix >= dim1 (i.e. partie entière = 0) :
    if (PuissanceDeDix >= dim1)
    {   RES.dim1 = 1; RES.GE_IPart[0] = 0; RES.dim2 =
dim2+PuissanceDeDix;
        for (int i=0; i<dim2; i++)
            RES.GE_FPart[i] = GE_FPart[i];
        for (int i=0; i<dim1; i++)
            RES.GE_FPart[dim2+i] = GE_IPart[i];
        for (int i=0; i<PuissanceDeDix-dim1; i++)
            RES.GE_FPart[dim2+dim1+i] = 0;
    }
    else  // Cas où PuissanceDeDix < dim2 :
    {   RES.dim1 = dim1-PuissanceDeDix; RES.dim2 = dim2+PuissanceDeDix;
        for (int i=0; i<dim2; i++)
            RES.GE_FPart[i] = GE_FPart[i];
        for (int i=0; i<PuissanceDeDix; i++)
            RES.GE_FPart[dim2+i] = GE_IPart[i];
        for (int i=0; i<dim1-PuissanceDeDix; i++)
            RES.GE_IPart[i] = GE_IPart[PuissanceDeDix+i];
    }

    RES.valide = valide; RES.signe = signe;
    return RES.Nettoyage();
}

GrandFlottant GrandFlottant::DecalageDroite(int PuissanceDeDix)
{   GrandFlottant RES;

    // Cas où PuissanceDeDix >= dim2 (tout décaler, plus de virgule) :
    if (PuissanceDeDix >= dim2)
    {   RES.dim1 = dim1+PuissanceDeDix; RES.dim2 = 0;
        for (int i=0; i<dim1; i++)
            RES.GE_IPart[RES.dim1-1-i] = GE_IPart[dim1-1-i];
        for (int i=0; i<dim2; i++)
            RES.GE_IPart[RES.dim1-dim1-1-i] = GE_FPart[dim2-1-i];
        for (int i=0; i<PuissanceDeDix-dim2; i++)
            RES.GE_IPart[RES.dim1-dim1-dim2-1-i] = 0;
    }
    else // Cas où PuissanceDeDix < dim2 :
    {   RES.dim1 = dim1+PuissanceDeDix; RES.dim2 = dim2-PuissanceDeDix;
        for (int i=0; i<dim1; i++)
```



```
            RES.GE_IPart[RES.dim1-1-i] = GE_IPart[dim1-1-i];
        for (int i=0; i<PuissanceDeDix; i++)
            RES.GE_IPart[RES.dim1-dim1-1-i] = GE_FPart[dim2-1-i];
        for (int i=0; i<dim2-PuissanceDeDix; i++)
            RES.GE_FPart[dim2-PuissanceDeDix-1]=GE_FPart[dim2-PuissanceDeDix-
1];
    }

    RES.valide = valide; RES.signe = signe;
    return RES.Nettoyage();
}


GrandFlottant GrandFlottant::Nettoyage()
{   GrandFlottant RES;

    // On s'occupe d'abord de la partie entière :
    int i=dim1-1; RES.dim1 = dim1; RES.dim2 = dim2;
    RES.valide = valide;
    while ((GE_IPart[i] == 0) && (i>0))    { RES.dim1--; i--; }
    if (RES.dim1 == 1)   RES.GE_IPart[0] = GE_IPart[0];
    else
        while (i>=0)      { RES.GE_IPart[i] = GE_IPart[i]; i--; }


    // On s'occupe ensuite de la partie fractionnaire :
    int NbreDeZero = 0; i = 0;
    while ((GE_FPart[i] == 0) && (i < dim2))
    {   RES.dim2--; NbreDeZero++; i++;
    }
    if ((RES.dim2 != 0))
    {   for (int j=0; j<RES.dim2; j++)
            RES.GE_FPart[j] = GE_FPart[j+NbreDeZero];
    }
    RES.signe = ((RES.dim2 == 0) && (RES.dim1 == 1)
              && (RES.GE_IPart[0] == 0) ? true : signe);
    return RES;
}

GrandFlottant GrandFlottant::Remplissage(int NbreDeZero)
{   GrandFlottant RES = *this;
    RES.dim2 = dim2+NbreDeZero;
    for (int i=0; i<dim2; i++)
        RES.GE_FPart[dim2+NbreDeZero-1-i] = GE_FPart[dim2-1-i];
    for (int i=0; i<NbreDeZero; i++)
        RES.GE_FPart[NbreDeZero-1-i] = 0;
    return RES;
}

bool GrandFlottant::Valide()
{   return (*this).valide;
}

// ************* Déclaration des fonctions membres operator *************

bool GrandFlottant::operator==(GrandFlottant b)
{   GrandFlottant X = Nettoyage(), Y = b.Nettoyage();

    if ((X.dim1 != Y.dim1) || (X.dim2 != Y.dim2))  return false;
    else
    {   for (int i=0; i<X.dim1; i++)
```

```
            if (X.GE_IPart[i] != Y.GE_IPart[i]) return false;
        if (X.dim2 != 0)
            for (int i=0; i<X.dim2; i++)
                if (X.GE_FPart[i] != Y.GE_FPart[i]) return false;
    }
    return true;
}

bool GrandFlottant::operator<(GrandFlottant b)
{   int i=0;
     if (!signe && b.signe)  return true;
    else if (signe && !b.signe)  return false;
    else if (signe && b.signe)
    {    if (dim1 < b.dim1)  return true;
        else if (dim1 > b.dim1)  return false;
        else
        {   while (i<dim1)
            {   if (GE_IPart[dim1-i-1] == b.GE_IPart[b.dim1-i-1]) i++;
                else return (GE_IPart[dim1-i-1] < b.GE_IPart[b.dim1-i-
1]);
            }
            // On remplit le flottant qui a la partie frac. la + petite
            // de 0 jusqu'à atteindre la dim. de l'autre (2 1ers for),
            // puis on effectue les comparaisons (dernier for)
            // EXEMPLE : 123.456 deviendra 123.4560000 si b.dim2 = 7 :
            if (dim2 < b.dim2)
            {   for (int i=0; i<dim2; i++)
                    GE_FPart[b.dim2-1-i] = GE_FPart[dim2-1-i];
                for (int i=0; i<b.dim2-dim2; i++)
                    GE_FPart[b.dim2-dim2-1-i] = 0;
                for (int i=b.dim2-1; i>0; i--)
                {   if (GE_FPart[i] < b.GE_FPart[i])  return true;
                    if (GE_FPart[i] > b.GE_FPart[i])  return false;
                }
                return (GE_FPart[0] < b.GE_FPart[0]);
            }
            else
            {   for (int i=0; i<b.dim2; i++)
                    b.GE_FPart[dim2-1-i] = b.GE_FPart[b.dim2-1-i];
                for (int i=0; i<dim2-b.dim2; i++)
                    b.GE_FPart[dim2-b.dim2-1-i] = 0;
                for (int i=dim2-1; i>0; i--)
                {   if (GE_FPart[i] < b.GE_FPart[i])  return true;
                    if (GE_FPart[i] > b.GE_FPart[i])  return false;
                }
                return (GE_FPart[0] < b.GE_FPart[0]);
            }
        }
    } else return (-b < -*this);
}


GrandFlottant GrandFlottant::operator-()
{   GrandFlottant RES = *this;
    RES.signe = !signe;
    return RES;
}


GrandFlottant GrandFlottant::operator~()
{   GrandFlottant NUL(0.0), RES=*this;
```



```
    if (NUL < *this)    return RES;
    else                return -RES;
}


GrandFlottant GrandFlottant::operator+(GrandFlottant b)
{   int retenue=0;
    GrandFlottant RES;

         if ( signe &&  b.signe)   RES.signe = true;
    else if (!signe && !b.signe)   RES.signe = false;
    else if ( signe && !b.signe)   return (*this-(-b));
    else                           return (b-(-*this));

    if (~*this < ~b) return (b + *this);

    // On remplit le flottant de dim frac la + ptte par des 0 :
    if (dim2 < b.dim2)
         *this = Remplissage(b.dim2-dim2);
    else b = b.Remplissage(dim2-b.dim2);
    RES.dim1 = dim1; RES.dim2 = dim2;

    // On additione les parties fractionnaires :
    for (int i=0; i<dim2; i++)
    {   RES.GE_FPart[i] = (GE_FPart[i]+b.GE_FPart[i]+retenue)%10;
        retenue = (GE_FPart[i]+b.GE_FPart[i]+retenue)/10;
    }

    // On additionne les parties entières :
    for (int i=0; i<b.dim1; i++)
    {   RES.GE_IPart[i] = (GE_IPart[i]+b.GE_IPart[i]+retenue)%10;
        retenue = (GE_IPart[i]+b.GE_IPart[i]+retenue)/10;
    }
    for (int i=b.dim1; i<dim1; i++)
    {   RES.GE_IPart[i] = (GE_IPart[i]+retenue)%10;
        retenue = (GE_IPart[i]+retenue)/10;
    }
    if (retenue == 1)   { RES.GE_IPart[dim1]=1; RES.dim1++; }

    RES.valide = (valide && b.valide);
    return RES.Nettoyage();
}


GrandFlottant GrandFlottant::operator-(GrandFlottant b)
{   int retenue=0; GrandFlottant RES;

         if ( signe && !b.signe)    return (*this + (-b));
    else if (!signe &&  b.signe)    return -((-*this) + b);
    else if (!signe && !b.signe)    return ((-b)-(-*this));

    if (*this < b)   return -(b-(*this));

    // On remplit le flottant qui a la partie frac. la plus petite
    // de 0 jusqu'à atteindre la dimension de l'autre.
    RES.dim1 = dim1;
    if (dim2 < b.dim2)
         RES = Remplissage(b.dim2-dim2);
    Else RES = b.Remplissage(dim2-b.dim2);

    // Soustraction des parties fractionnaires :
```

```
    for (int i=0; i<RES.dim2; i++)
    {   RES.GE_FPart[i] = GE_FPart[i]-b.GE_FPart[i]-retenue;
        retenue = 0;
        if (RES.GE_FPart[i] < 0)   { RES.GE_FPart[i] += 10; retenue = 1 ;
}
    }

    // Soustraction des parties entières :
    for (int i=0; i<b.dim1; i++)
    {   RES.GE_IPart[i] = GE_IPart[i]-b.GE_IPart[i]-retenue;
        retenue = 0;
        if (RES.GE_IPart[i] < 0)   { RES.GE_IPart[i] += 10; retenue = 1;
}
    }
    for (int i=b.dim1; i<dim1; i++)
    {   RES.GE_IPart[i] = GE_IPart[i]-retenue;
        retenue = 0;
        if (RES.GE_IPart[i] < 0)   { RES.GE_IPart[i] += 10; retenue = 1;
}
    }

    RES.signe = true; RES.valide = (valide && b.valide);
    return RES.Nettoyage();
}


GrandFlottant GrandFlottant::operator*(GrandFlottant b)
{   int retenue=0; int DIM=dim2+b.dim2;
    GrandFlottant RES, TMP, NUL(0.0), X=*this, Y=b;

    X = X.DecalageDroite(dim2);     // Transforme X en "GrandEntier"...
    Y = Y.DecalageDroite(b.dim2);   // Transforme Y en "GrandEntier"...

    for (int j=0; j<Y.dim1; j++)
    {   TMP.dim1 = X.dim1 + j;
        TMP.dim2 = 0; TMP.valide = true;
        for (int i=0; i<j; i++)    TMP.GE_IPart[i] = 0;
        for (int i=0; i<X.dim1; i++)
        {   TMP.GE_IPart[i+j] = (Y.GE_IPart[j]*X.GE_IPart[i]+retenue)%10;
            retenue = (Y.GE_IPart[j]*X.GE_IPart[i]+retenue)/10;
        }
        if (retenue > 0)   { TMP.GE_IPart[TMP.dim1]=retenue; TMP.dim1++;
}
        retenue = 0;
        RES = RES + TMP;
    }

    RES = RES.DecalageGauche(DIM);
    RES.signe = ((signe &&  b.signe) || (!signe && !b.signe));
    RES.valide = (valide && b.valide);
    return RES;
}


GrandFlottant GrandFlottant::DivisionEntiere(GrandFlottant b)
{   GrandFlottant X=*this, Y=b, NUL;
    int i=0, m=0, P=dim1-b.dim1-1;

         if (~X < ~Y)             { NUL.valide = true; return NUL; }
    else if (signe  && !b.signe)  { return -(X.DivisionEntiere(-Y)); }
    else if (!signe &&  b.signe)  { return -((-X).DivisionEntiere(Y)); }
```



```
    else if (!signe && !b.signe)  { return (-X).DivisionEntiere(-Y); }

    GrandFlottant RES;
    if (valide && b.valide)  RES.valide = true;
    else return (valide ? b : *this);

    GrandFlottant UN("1");
    while ((Y < X) || (Y == X))   { RES = RES + UN; Y = Y+b; }
    return RES;
}

GrandFlottant GrandFlottant::operator/(GrandFlottant b)
{   GrandFlottant X=*this, Y=b, RES, RESTE, TMP;
    int i=1;

         if (!X.signe && !Y.signe)        return (-X)/(-Y);
    else if (!X.signe &&  Y.signe)        return -((-X)/Y);
    else if (X.signe  && !Y.signe)        return -(X/(-Y));

    X = X.Nettoyage(); Y = Y.Nettoyage();
    int MAXDIM = (X.dim2 > Y.dim2 ? X.dim2 : Y.dim2);

    X = X.DecalageDroite(MAXDIM);   // Transforme X en "GrandEntier"...
    Y = Y.DecalageDroite(MAXDIM);   // Transforme Y en "GrandEntier"...

    // Calcul de la partie entière (ss la forme 123,0000<-GrandFlottant)
    RES = X.DivisionEntiere(Y); TMP = RES;

    // Calcul de la partie fractionnaire :
    GrandFlottant UN("1");  // Servira à comparer si le quotient est=0.
    while (i < NBD)
    {   if (TMP < UN)    RESTE = X.DecalageDroite(1);
        else             RESTE = X-(TMP*Y);
        // Si le reste est nul, on s'arrete :
        if (RESTE < UN)  return RES;
        // Sinon, on continue les divisions successives :
        if (TMP < UN)    X = RESTE;
        else             X = RESTE.DecalageDroite(1);
        TMP = X.DivisionEntiere(Y);
        if (TMP < UN)    RES = RES.Remplissage(1);
        else             RES = RES + TMP.DecalageGauche(i);
        i++;
    }
    return RES;
}

GrandFlottant GrandFlottant::operator^(GrandFlottant b)
{   GrandFlottant X=*this, Y=b, RES=X;
    GrandFlottant NUL, ZERO(0.0), UN(1.0), DEUX(2.0);

    if (Y.dim2 != 0) return NUL;
    if (Y < ZERO)    { Y = -Y; return UN/(X^Y); }
    if (Y < UN)      return UN;

    RES.valide = true;
    if (Y < DEUX)    return RES;
    else             return RES*(X^(Y-UN));
}
```

Main correspondant :

```
#include <iostream.h>
#include <stdlib.h>
#include "GrandFlottant.h"

int main()
{   GrandFlottant a, b;
    GrandFlottant c("300"), d(123.456);
    cout << "Test de la fonction de saisie : "; cin >> a;
    cout << "                                "; cin >> b;
    cout << "Test de la fonction d'affichage : a = " << a << endl;
    cout << "                                  b = " << b << endl;
    cout << "Test du constructeur -> chaine de caracteres : c = ";
    cout << c << endl;
    cout << "Test du constructeur -> 'double'             : d = ";
    cout << d << endl << endl;
    cout << "Test de la fonction d'inegalite : a < b => ";
      if (a<b) cout << "Vrai." << endl;
      else     cout << "Faux." << endl;
    cout << "Test de la fonction d'égalité   : a = b => ";
      if (a==b) cout << "Vrai." << endl;
      else      cout << "Faux." << endl;
    cout << endl;

    cout << "Test de la fonction de decalage gauche : c/100 = ";
    cout << c.DecalageGauche(2) << endl;
    cout << "                                         d/100000 = ";
    cout << d.DecalageGauche(5) << endl;
    cout << "Test de la fonction de decalage droite : c*100 = ";
    cout << c.DecalageDroite(2) << endl;
    cout << "                                         d*100000 = ";
    cout << d.DecalageDroite(5) << endl << endl;
    cout << "Test de la fonction de Remplissage (3 zeros) : c = ";
    cout << c.Remplissage(3) << endl;
    cout << "                                    (7 zeros) : d = ";
    cout << d.Remplissage(7) << endl << endl;

    cout << "Test des fonctions 'operator' :" << endl;
    cout << "  1. a + b = " << a+b << endl;
    cout << "  2. a - b = " << a-b << endl;
    cout << "  3. a * b = " << a*b << endl;
    cout << "  4. a / b = " << a/b << endl;
    cout << "  5. a ^ b = " << (a^b) << endl;

    return 0;
}
```


Question 3 : **On réalisera maintenant une calculatrice permettant de faire des calculs sur ces grandes valeurs décimales et entières. Cette calculatrice utilisera la notation infixe. On considérera d'abord les parenthèses comme obligatoires. Réaliser une boucle d'interaction permettant d'entrer d'abord les nombres les uns après les autres (comme sur une calculatrice à quatre opérations, c'est-à-dire, d'abord 3, puis +, puis 5, puis = donnera le résultat 8). On modifiera ensuite cette boucle de manière à pouvoir entrer une expression directement (3+5= affichera le résultat 8).**

Version 1 : notation polonaise inverse

Projet_question3_PostFix.h

```
//  * * * * * * * * * * * * * * *  CaltoPostFix.cpp * * * * * * * * * * * * * * *
//  * Fichier : CaltoPostFix.cpp                                                    *
//  * Date    : 12.01.2003                                                          *
//  * Copyright : (C) by LENZEN Martial. Tous droites réservés.                     *
//  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <stdlib.h>
#include "GrandFlottant.h"

#define MAX   1024


// On initialise la classe Pile qui contient les GrandFlottant :

struct Cell
{   GrandFlottant GF;
    struct Cell *suivant;
};

class Pile
{    Cell *premier;
    public :
                    Pile() { premier = (Cell*) 0; }
     GrandFlottant  Sommet();
     void           Empiler(GrandFlottant D);
     void           Depiler();
     void           Vide();
     bool           EstVide();
     int            Hauteur();
};


// ******* Déclarations des fonctions propres à la classe 'Pile' ********

GrandFlottant Pile::Sommet()
{   if (premier == NULL) return GrandFlottant();
    else                 return premier->GF;
}

void Pile::Empiler(GrandFlottant D)
```

```
{   Cell *CELL; CELL = new Cell; CELL->GF = D;
    CELL->suivant = premier; premier = CELL;
}

void Pile::Depiler()
{   Cell *CELL; CELL = new Cell; CELL = premier;
    premier = premier->suivant;
}

void Pile::Vide()
{   Cell *CELL;
    while (premier != NULL)
    {   CELL = new Cell;
        CELL = premier;
        premier = premier->suivant;
    }
}

bool Pile::EstVide()
{   return (premier == NULL);
}

int Pile::Hauteur()
{   int dim=0;
    Cell *CELL; CELL = new Cell; CELL = premier;
    while (CELL != NULL)     { CELL = CELL->suivant; dim++; }
    return dim;
}


// On initialise maintenant la classe Calculatrice :

class Calculatrice
{    Pile P;
    public :
          Calculatrice()     { P = Pile(); }
     GrandFlottant Resultat(){ return P.Sommet(); }

     // Initialisation des fonctions "opérateurs" :
     void Addition();
     void Soustraction();
     void Multiplication();
     void Division();
     void Minus();
     void ValeurAbsolue();

     // Initialisation des fonctions servant à la boucle interactive :
     bool Boucle();
     void Afficher();
     void AfficherTout();
};


// *************** Déclaration des fonctions "opérateurs" ***************

void Calculatrice::Addition()
{   GrandFlottant A, B, RES;
    A = P.Sommet(); P.Depiler();
    B = P.Sommet(); P.Depiler();
    P.Empiler(B+A);
}
```



```
void Calculatrice::Soustraction()
{   GrandFlottant A, B;
    A = P.Sommet(); P.Depiler();
    B = P.Sommet(); P.Depiler();
    P.Empiler(B-A);
}

void Calculatrice::Multiplication()
{   GrandFlottant A, B;
    A = P.Sommet(); P.Depiler();
    B = P.Sommet(); P.Depiler();
    P.Empiler(B*A);
}

void Calculatrice::Division()
{   GrandFlottant A, B;
    A = P.Sommet(); P.Depiler();
    B = P.Sommet(); P.Depiler();
    P.Empiler(B/A);
}

void Calculatrice::Minus()
{   GrandFlottant A;
    A = P.Sommet(); P.Depiler();
    P.Empiler(-A);
}

void Calculatrice::ValeurAbsolue()
{   GrandFlottant A;
    A = P.Sommet(); P.Depiler();
    P.Empiler(~A);
}


// ***** Déclaration des fonctions servant à la boucle interactive ******

void Calculatrice::Afficher()
{   cout << ">> " << Resultat() << endl;
    return;
}

bool Calculatrice::Boucle()
{   bool ok=false; char buf[MAX];
    cout << "<< ";
    cin.getline(buf, MAX);
    if (buf[1] == '\0')
        switch (buf[0])
        {   case '+' : Addition();           ok=true; break;
            case '-' : Soustraction();       ok=true; break;
            case '*' : Multiplication();     ok=true; break;
            case '/' : Division();           ok=true; break;
            case '=' : Afficher();           ok=true; break;
            case 'd' : P.Depiler();          ok=true; break;
            case 'v' : P.Vide();             ok=true; break;
            case 'q' : return false;
        }
    if (!ok) P.Empiler(GrandFlottant(buf));
    return true;
}
```

Main correspondant :

```
#include <iostream.h>
#include "Projet_question3_PostFix.h"

int main()
{    Calculatrice CAL;
     cout << "ATTENTION : Cette calculatrice utilise la notation polonaise
               inversée," << endl;
     cout << "---------   autrement dit, la notation postfixée." << endl;
     cout << "  Vous entrez d'abord les nombres, puis les opérations." << endl;
     cout << "  A tout moment, un appui sur les touches suivantes : " << endl;
     cout << "'=' => affiche le résultat courant ;" << endl;
     cout << "'d' => dépile la pile de la calculatrice ;" << endl;
     cout << "'v' => vide la pile de la calculatrice ;" << endl;
     cout << "'q' => quitte la calculatrice." << endl << endl;
     while(CAL.Boucle());
}
```

Version 2 : notation standard avec gestion des parenthèses et priorités :

CaltoInfixe.cpp

```
//  * * * * * * * * * * * * * * CaltoInfixe.cpp * * * * * * * * * * * * * * *
//  * Fichier : CaltoInfixe.cpp                                               *
//  * Date    : 12.01.2003                                                    *
//  * Copyright : (C) by LENZEN Martial. Tous droites réservés.              *
//  * Restrictions : LE PARANTHESAGE NE FONCTIONNE QUE DANS LA LIMITE         *
//  *                ET  UNE SEULE PAIRE.                                     *
//  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream>
#include <stdlib.h>
#include <string.h>
#include "GrandFlottant.h"

#define  N   255


// Déclaration des fonctions :

GrandFlottant    Resultat(GrandFlottant Calc[], char OP[], int DimOP);
GrandFlottant    Calcul(char CALCUL[], GrandFlottant Calc[], char OP[]);
GrandFlottant    CharsToGF(char nb[]);
void             ErreurOp();
int              RecherchePriorites(GrandFlottant Calc[],char OP[],int
DimOP);
int              Decompose(char CALCUL[], GrandFlottant Calc[], char
OP[]);
int              NbDeParantheses(char CALCUL[]);
int              NbDOperateurs(char CALCUL[]);


int main()
{   char CALCUL[N], OP[N]; int DimOP=0;
    GrandFlottant Calc[N], RES(0.0);
```



```
    cout << ">> ";
    cin.getline(CALCUL, N);

    if ((CALCUL[0] == 'q') || (CALCUL[0] == 'Q')) return 0;

    cout << "<< " << CALCUL << " = ";
    RES = Calcul(CALCUL, Calc, OP);
    if (RES.Valide())    cout << RES << endl;
    return 0;
}


// Définition des fonctions :

GrandFlottant Resultat(GrandFlottant Calc[], char OP[], int DimOP)
{   GrandFlottant res = Calc[0];

    if (DimOP > 0)
    {   for (int i=0; i<DimOP; i++)
            switch (OP[i])
            {   case '+' :  { res = res+Calc[i+1]; break; }
                case '-' :  { res = res-Calc[i+1]; break; }
                case '*' :  { res = res*Calc[i+1]; break; }
                case '/' :  { res = res/Calc[i+1]; break; }
                case '^' :  { res = (res^Calc[i+1]); break; }
                default  :  { ErreurOp(); break; }
            }
    } return res;
}

void ErreurOp()
{   cout << "Erreur de l'opérande." << endl;
}

GrandFlottant CharsToGF(char nb[])
{   return GrandFlottant(nb);
}

int Decompose(char CALCUL[], GrandFlottant Calc[], char OP[])
//  fonction qui retourne la dimension de OP[]
{   int i=0, pos=0, op=0, calc=0, NbDeP=0;
    char CALC[N][N];

    if ((CALCUL[0]=='+') || (CALCUL[0]=='-') || (CALCUL[0]=='*') ||
        (CALCUL[0]=='/') || (CALCUL[0]=='^'))
        return -1;
    if (NbDeParantheses(CALCUL) < 0)  return NbDeParantheses(CALCUL);

    while (CALCUL[i] != '\0')
    {   switch (CALCUL[i])
        {   case '+' :   { OP[op]='+'; op++; calc++; pos=0; break; }
            case '-' :   { OP[op]='-'; op++; calc++; pos=0; break; }
            case '*' :   { OP[op]='*'; op++; calc++; pos=0; break; }
            case '/' :   { OP[op]='/'; op++; calc++; pos=0; break; }
            case '^' :   { OP[op]='^'; op++; calc++; pos=0; break; }
            case '(' :   {   NbDeP++; i++;
                             while ((CALCUL[i] != '\0') && (i<N))
                             {   if (CALCUL[i] == '(')     NbDeP++;
                                 if (CALCUL[i] == ')')     NbDeP--;
                                 if (NbDeP == 0)           break;
                                 pos++; CALC[calc][pos-1]=CALCUL[i];i++;
```

```
                            } break;
                        }
            default  :  { pos++; CALC[calc][pos-1]=CALCUL[i]; break; }
        } i++;
    }

    for (int j=0; j<=calc; j++)
        if ((NbDeParantheses(CALC[j])==0) && (NbDOperateurs(CALC[j])==0))
            Calc[j] = CharsToGF(CALC[j]);
        else
        {   char OP2[N]; GrandFlottant Calc2[N]; // <= pb pr moult () !!
            Calc[j] = Calcul(CALC[j], Calc2, OP2);
        }
    return op;
}

int RechercheePriorites(GrandFlottant Calc[], char OP[], int DimOP)
// Fonction qui retourne la nouvelle dim de OP, càd DimOP-1
{   int i=0;

    if ((OP[0]=='^') && (DimOP==1))   return DimOP;
    if ((OP[0]=='*') && (DimOP==1))   return DimOP;
    if ((OP[0]=='/') && (DimOP==1))   return DimOP;

    for (int i=0; i<DimOP; i++)
        if ((OP[i] == '^') && (i<N))
        {   Calc[i] = (Calc[i] ^ Calc[i+1]);
            for (int j=i+2; j<=DimOP; j++)
            {   Calc[j-1] = Calc[j];
                OP[j-2] = OP[j-1];
            } DimOP--;
            return DimOP;
        }

    for (int i=0; i<DimOP; i++)
        if (((OP[i] == '*') || (OP[i] == '/')) && (i<N))
        {   if (OP[i] == '*')
            {   Calc[i] = Calc[i]*Calc[i+1];
                for (int j=i+2; j<=DimOP; j++)
                {   Calc[j-1] = Calc[j];
                    OP[j-2] = OP[j-1];
                }
            }
            else
            {   Calc[i] = Calc[i]/Calc[i+1];
                for (int j=i+2; j<=DimOP; j++)
                {   Calc[j-1] = Calc[j];
                    OP[j-2] = OP[j-1];
                }
            } DimOP--;
        }
    return DimOP;
}

int NbDeParantheses(char CALCUL[])
{   int Verif=0, NbDeP=0, i=0;

    while ((CALCUL[i] != '\0') && (i<N))
    {   if (CALCUL[i] == '(')    { Verif++; NbDeP++; }
        if (CALCUL[i] == ')')    Verif--; i++;
    }
```



```
        if (Verif == 1)      return -2;
    else if (Verif == -1)    return -3;
    else if (Verif < 0)      return -4;
    else if (Verif > 0)      return -5;
    return NbDeP;
}

int NbDOperateurs(char CALCUL[])
{   int NB=0, i=0;
    while ((CALCUL[i] != '\0') || (i<N))
    {   if ((CALCUL[i]=='+') || (CALCUL[i]=='-') || (CALCUL[i]=='*')
             || (CALCUL[i]=='/') || (CALCUL[i]=='^'))
            NB++;
        i++;
    }
    return NB;
}

GrandFlottant Calcul(char CALCUL[], GrandFlottant Calc[], char OP[])
{   int DimOP; GrandFlottant res;
    DimOP = Decompose(CALCUL, Calc, OP);
         if (DimOP == -1)  ErreurOp();
    else if (DimOP == -2)  cout << "Paranthèse ) manquante..." << endl;
    else if (DimOP == -3)  cout << "Paranthèse ( manquante..." << endl;
    else if (DimOP == -4)  cout << "Paranthèses ) manquantes..." <<
endl;
    else if (DimOP == -5)  cout << "Paranthèses ( manquantes..." <<
endl;
    else
    {   for (int i=0; i<DimOP; i++);
            DimOP = RechercheePriorites(Calc, OP, DimOP);
        res = Resultat(Calc, OP, DimOP);
    }
    return res;
}
```

Soutenance, question supplémentaire : implémenter une fonction qui calcule la factorielle d'un grand entier (fonctionne dans GrandEntier.h ET GrandEntier2.h) :

```
GrandEntier GrandEntier::Factorielle()
{   GrandEntier ZERO(0.0), UN(1.0), DEUX(2.0), TROIS(3.0), N=*this;

    if (N < ZERO)        {N.valide = false; return N; }
        if (N < UN)      return UN ;
    else if (N < DEUX)   return UN ;
    else if (N < TROIS)  return DEUX;

    return N*(N-UN)*(N-DEUX).Factorielle();
}
```

Remarque personelle : Dire qu'on avait une heure pour faire cette fonction...