Les bases : exercices corrigés
Corrigé

Objectifs

– Raffiner des problèmes simples ;
– Écrire quelques algorithmes simples ;
– Savoir utiliser les types de base ;
– Savoir utiliser les instructions « élémentaires » : d'entrée/sortie, affichage, bloc...
– Manipuler les conditionnelles ;
– Manipuler les répéritions.

Exercice 1 : Résolution d'une équation du second degré
Soit l'équation du second degré $ax^2 + bx + c = 0$ où a, b et c sont des coefficients réels. Écrire un programme qui saisit les coefficients et affiche les solutions de l'équation.
Indication : Les solutions sont cherchées dans les réels. Ainsi, dans le cas général, en considérant le discriminant $\Delta = b^2 - 4ac$, on a :

$$\begin{cases} \Delta < 0 & \text{pas de solutions réelles.} \\ \Delta = 0 & \text{une solution double : } \frac{-b}{2a} \\ \Delta > 0 & \text{deux solutions : } x_1 = \frac{-b-\sqrt{\Delta}}{2a} \text{ et } x_2 = \frac{-b+\sqrt{\Delta}}{2a} \end{cases}$$

Quelles sont les solutions de l'équation si le coefficient a est nul ?
Solution : Voici un raffinage possible :

```
R0 : Résoudre l'équation du second degré

R1 : Raffinage De « Résoudre l'équation du second degré »
     | Saisir les 3 coefficients          a, b, c: out RÉEL -- les coefficients
     | Calculer et afficher les solutions

R2 : Raffinage De « Saisir les 3 coefficients »
     | Écrire("Entrer␣les␣valeurs␣de␣a,␣b␣et␣c␣:␣")
     | Lire(a, b, c)
```

```
R2 : Raffinage De « Calculer et afficher les solutions »
     |  Si équation du premier degré Alors
     |      Résoudre l'équation du premier degré bx + c = 0
     |  Sinon
     |      Calculer le discriminant
     |          --> delta: RÉEL          -- le discriminant de ax^2 + bx + c = 0
     |      Si discriminant nul Alors            -- une solution double
     |          Écrire("Une␣solution␣double␣:␣", - B / (2 * A))
     |      SinonSi discriminant positif Alors   -- deux solutions distinctes
     |          Écrire("Deux␣solutions␣:␣")
     |          Écrire((-B + √Delta) / (2*A))
     |          Écrire((-B - √Delta) / (2*A))
     |      Sinon       { discriminant négatif }        -- pas de solution dans IR
     |          Écrire("Pas␣de␣solution␣réelle")
     |      FinSi
     |  FinSi

R3 : Raffinage De « Résoudre l'équation du premier degré bx + c = 0 »
     |  Si B = 0 Alors
     |      Si C = 0 Alors
     |          Écrire("Tout␣IR␣est␣solution")
     |      Sinon
     |          Écrire("Pas␣de␣solution")
     |      FinSi
     |  Sinon
     |      Écrire("Une␣solution␣:␣", - C / B)
     |  FinSi
```

Remarque : Notons que nous n'avons pas un premier niveau de raffinage en trois étapes, *saisir*, *calculer*, *afficher* car il est difficile de dissocier les deux dernières étapes car la forme des racines de l'équation du second est très variable et ne peut pas être facilement capturée par un type (deux solution distinctes, une solution double, pas de solution, une infinité de solution). Aussi, nous avons regroupé calcul et affichage.

Et l'algorithme correspondant :

```
Algorithme second_degré

    -- Résoudre l'équation du second degré

Variables
    a, b, c: Réel        -- les coefficients de l'équation
    delta: Réel          -- le discriminant

Début
        -- Saisir les 3 coefficients
        Écrire("Entrer␣les␣valeurs␣de␣a,␣b␣et␣c␣:␣")
        Lire(a, b, c)

        -- Calculer et afficher les solutions
```

```
        Si a = 0 Alors          -- équation du premier degré
            -- Résoudre l'équation du premier degré bx + c = 0
            Si B = 0 Alors              -- équation constante
                Si C = 0 Alors
                    Écrire("Tout IR est solution")
                Sinon
                    Écrire("Pas de solution")
                FinSi
            Sinon                       -- équation réellement du premier degré
                Écrire("Une solution : ", - C / B)
            FinSi

        Sinon                   -- équation réellement du second degré
            -- Calculer le discriminant delta
            delta ← b*b - 4*a*c

            -- Déterminer et afficher les solutions
            Si delta = 0 Alors          -- une solution double
                Écrire("Une solution double : ", - B / (2 * A))
            SinonSi delta > 0 Alors     -- deux solutions distinctes
                Écrire("Deux solutions : ")
                Écrire((-B + sqrt(delta)) / (2*A))
                Écrire((-B - sqrt(delta)) / (2*A))
            Sinon       { discriminant négatif }        -- pas de solution dans IR
                Écrire("Pas de solution réelle")
            FinSi
        FinSi
Fin.
```

```
/*********************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.3
 *  Objectif : Résolution de l'équation du seond degré.
 ********************************************************************/

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

/* Demande à l'utilisateur les coefficients a, b et c de l'équation du
 * second degré  ax^2 + bx + c* , calcule et affiche les solutions
 * réelles de cette équation.
 */
int main()
{
    /* Déclaration des variables (C oblige) */
    float a, b, c;      /* coefficients de l'équation   */

    /* Poser le problème */
    printf("Résolution de l'équation du second degré :\n");
```

```
    printf("\t␣␣␣␣a.x2␣+␣b.x␣+c␣=␣0\n");

    /* Saisir la valeur des coefficients */
    printf("Donnez␣la␣valeur␣des␣coefficients␣a,␣b␣et␣c␣:␣");
    scanf("%f%f%f", &a, &b, &c);

    /* Résoudre l'équation et afficher les résultats */
    if (a == 0) {        /*{ équation du premier degré }*/
        /* résoudre l'équation du premier degré */
        if (b == 0) {
            if (c == 0) {
                printf("Tout␣les␣réels␣sont␣solution");
            }
            else {
                printf("Pas␣de␣solution");
            }
        }
        else {
            printf("Une␣seule␣solution␣:␣%f", -c/b);
        }
    }
    else {
        /* cas général  */
        float delta = b*b - 4*a*c;             /* le discriminant */
        if (delta > 0) {
            /* deux solutions réelles   */
            printf("Deux␣solutions␣:\n");
            printf("\tsolution␣1␣=␣%f\n", (-b + sqrt(delta))/2/a);
            printf("\tsolution␣2␣=␣%f\n", (-b - sqrt(delta))/2/a);
        }
        else if (delta == 0) {
            /* une solution double      */
            printf("Une␣solution␣double␣:␣%f\n", (-b/2/a));
        }
        else {
            /* pas de solutions réelles */
            printf("Pas␣de␣solutions␣réelles");
        }
    }
    printf("\n");

    return EXIT_SUCCESS;
}
```

Exercice 2 : Le nombre d'occurrences du maximun
Compléter le programme de l'exercice 2 (Exercices résolus, Semaine 1) qui calcule les statistiques sur une série de valeurs réelles pour qu'il affiche également le nombre d'occurrences de la plus grande valeur, c'est-à-dire le nombre de valeurs de la série qui correspondent à la valeur maxiamle.

Par exemple, pour la série 1 2 3 1 2 3 3 2 3 1 0, le max est 3 et il y a quatre occurrences de 3. Dans la série 1 2 3 3 3 3 1 2 4 1 2 3 0, il y a une seule occurrence du maximun qui est 4.

Solution : Le principe est d'ajouter une nouvelle variable d'accumulation, nb_max, qui comptabilise le nombre d'occurrences du max rencontrées. Elle est initialisée à 1 sur la première valeur. Elle est remise à 1 dès qu'un nouveau maximum est identifié. Elle est augmentée de 1 si la valeur lue est égale au maximum précédent.

```
/****************************************************************************
 *  Auteur  :  Xavier Crégut <cregut@enseeiht.fr>
 *  Version :  1.1
 *
 *  Objectif :
 *      Afficher la moyenne, la plus grande et la plus petite valeur et le
 *      nombre d'occurences de la plus grande valeur d'une série de valeurs
 *      réelles lues au clavier.
 *
 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main()
{
    double x;           /* un réel lu au clavier */
    int nb;             /* le nombre de valeurs lues de la série */
    double somme;       /* la somme des valeurs lues de la série */
    double min;         /* la plus petite des valeurs lues de la série */
    double max;         /* la plus grande des valeurs lues de la série */
    double nb_max;      /* le nombre d'occurrences du max */

    /* afficher la consigne */
    printf("Donnez une série d'entiers qui se termine par 0.\n");

    /* lire le premier réel */
    scanf("%lf", &x);

    if (x == 0) {       /* Pas de valeur dans la séries */
        printf("La série est vide.  ");
        printf("Les statistiques demandées n'ont pas de sens !\n");
    }
    else {
        /* initialiser les variables statistiques avec x */
        max = x;
        min = x;
        somme = x;
```

```
        nb = 1;
        nb_max = 1;

        /* lire une nouvelle valeur x */
        scanf("%lf", &x);

        while (x != 0) {
            /* mettre à jour les variables statistiques */
            nb++;
            somme += + x;
            if (x > max) {
                max = x;
                nb_max = 1;
            }
            else if (x == max) {
                nb_max++;
            }
            else if (x < min) {
                min = x;
            }

            /* lire une nouvelle valeur x */
            scanf("%lf", &x);
        }

        /* afficher les statistiques */
        printf("Moyenne = %f\n", somme / nb);
        printf("Plus petite valeur = %f\n", min);
        printf("Plus grande valeur = %f\n", max);
        printf("Nb d'occurrences du max = %f\n", nb_max);
    }

    return EXIT_SUCCESS;
}

/*                      moyenne, min, max,      nb_max
    1 2 3 0     -->     2,       1,     3       1
    2 -2 0      -->     0,       -2,    2       1
    -4 -2 0     -->     -3,      -4,    -2      1
    13 0        -->     13,      13,    13      1
    0           -->     Non défini
    1 3 1 3 0   -->     2        1      3       2
    3 3 3 3 0   -->     3        3      3       4
*/
```

Exercice 3 : Nombres amis
Deux nombres N et M sont amis si la somme des diviseurs de M (en excluant M lui-même) est égale à N et la somme des diviseurs de N (en excluant N lui-même) est égale à M.

Écrire un programme qui affiche tous les couples (N, M) de nombres amis tels que $N \leq M \leq MAX$, MAX étant lu au clavier.

Indication : Les nombres amis compris entre 1 et 100000 sont (6, 6), (28, 28), (220, 284), (496, 496), (1184, 1210), (2620, 2924), (5020, 5564), (6232, 6368), (8128, 8128), (10744, 10856), (12285, 14595), (17296, 18416), (66928, 66992), (67095, 71145), (63020, 76084), (69615, 87633) et (79750, 88730).

Solution : L'idée est de faire un parcours de tous les couples possibles et de se demander s'ils forment un couple de nombres parfaits ou non. Pour un n et un m donnés, il s'agit alors de calculer la somme de leurs diviseurs. C'est en fait deux fois le même problème. Il s'agit de calculer la somme des diviseurs d'un entier p. Naïvement, on peut regarder si les entiers de 2 à $p - 1$ sont des diviseurs de p.

Une première optimisation consiste à remarquer que p n'a pas de diviseurs au delà de $p/2$. En fait, il est plus intéressant de remarquer que si i est diviseur de p alors p/i est également un diviseur de p. Il suffit donc de ne considérer comme diviseur potentiel que les entiers compris dans l'intervalle 2..$\sqrt{p}$. Il faut toutefois faire attention à ne pas comptabiliser deux fois $\sqrt{p}$.

```
R0 : Afficher les couples de nombres parfaits (N, M) avec 1 < N < M <= Max

R1 : Raffinage De « R0 »
      | Pour m ← 2 JusquÀ m = Max Faire
      |    | Pour n ← 2 JusquÀ n = m Faire
      |    |    | Si n et m parfaits Alors
      |    |    |    | Afficher le couple (n, m)
      |    |    | FinSi
      |    | FinPour
      | FinPour

R2 : Raffinage De « n et m parfaits »
      | Résultat ← (somme des diviseurs de N) = M
      |           Et (somme des diviseurs de M) = N

R3 : Raffinage De « somme des diviseurs de p »
      | somme ← 0
      | Pour i ← 2 JusquÀ racine carrée de p Faire
      |    | Si i diviseur de p Alors
      |    |    | somme ← somme + i + (p Div i)
      |    | FinSi
      | FinPour
      | Si p est un carré parfait Alors
      |    | somme ← somme - racine carrée de p
      | FinSi
```

L'algorithme est alors le suivant. Notez quelques optimisations qui font que l'algorithme de respecte pas complètement le raffinage.

```
Algorithme nb_amis

   -- Afficher les couples de nombres amis (N, M) avec 1 < N < M <= MAX

Variables
    n, m: Entier         -- pour représenter les couples possibles
    somme_n: Entier      -- somme des diviseurs de n
    somme_m: Entier      -- somme des diviseurs de m

Début
    Pour m ← 2 JusquÀ m = MAX Faire
        -- calculer la somme des divieurs de m
        -- Remarque : on peut déplacer cette étape à l'extérieur de la
        -- boucle car elle ne dépend pas de n (optimisation).
        somme_m ← 1
        Pour i ← 2 JusquÀ racine carrée de m Faire
            Si i diviseur de m Alors
                somme_m ← somme_m + i + (m Div i)
            FinSi
        FinPour
        Si m est un carré parfait Alors
            somme_m ← somme_m - racine carrée de m
        FinSi


        Pour n ← 2 JusquÀ n = m Faire
            -- calculer la somme des divieurs de n
            somme_n ← 1
            Pour i ← 2 JusquÀ racine carrée de n Faire
                Si i diviseur de n Alors
                    somme_n ← somme_n + i + (n Div i)
                FinSi
            FinPour
            Si n est un carré parfait Alors
                somme_n ← somme_n - racine carrée de n
            FinSi

            -- déterminer si n et m sont amis
            Si (somme_n = m) Et (somme_m = n) Alors { n et m sont amis }
                Afficher le couple (n, m)
            FinSi
        FinPour
    FinPour

Fin.
```

En C.

```
/*********************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *
```

```c
 *  Objectif : Afficher les couples de nombres amis (N, M)
 *      avec 1 < N < M <= MAX.
 *********************************************************************/

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

#define MAX 1000000

int main()
{
    int n, m;           /* pour représenter les couples possibles */
    int somme_n;        /* somme des diviseurs de n */
    int somme_m;        /* somme des diviseurs de m */
    int racine;         /* la racine carrée d'un nombre */
    int i;              /* varaible de boucle */

    for (m = 2; m <= MAX; m++) {
        /* calculer la somme des divieurs de m */
        /* Remarque : on peut déplacer cette étape à l'extérieur de la boucle
        ** car elle ne dépend pas de n (optimisation). */
        somme_m = 1;
        racine = (int) (sqrt(m));
        for (i = 2; i <= racine; i++) {
            if (m % i == 0) {   /* i diviseur de m */
                somme_m = somme_m + i + (m / i);
            }
        }
        if (m == racine * racine) {     /* m est un carré parfait */
            somme_m = somme_m - racine;
        }

        for (n = 2; n <= m; n++) {
            /* calculer la somme des divieurs de n */
            somme_n = 1;
            racine = (int) (sqrt(n));
            for (i = 2; i <= racine; i++) {
                if (n % i == 0) {       /*  i diviseur de n */
                    somme_n = somme_n + i + (n / i);
                }
            }
            if (n == racine * racine) { /* n est un carré parfait */
                somme_n = somme_n - racine;
            }

            /* déterminer si n et m sont amis */
            if (somme_n == m && somme_m == n) { /* n et m sont amis */
                printf("(%d,␣%d)\n", n, m);
            }
        }
```

```
    }

    return EXIT_SUCCESS;
}
```

Une solution plus efficace consiste à constater que pour un entier M compris entre 2 et MAX, le seul nombre amis possible est la somme de ses diviseurs que l'on note `somme_m`. Il reste alors à vérifier si la somme des diviseurs de `somme_m` est égale à M pour savoir si `somme_m` et M sont amis. Le fait de devoir afficher les couples dans l'ordre croissant nous conduit à ne considérer que les sommes de diviseurs inférieures à M.

À titre indicatif, pour Max = 1000000, ce deuxième algorithme termine en 1 minute 23 alors que dans le même temps, seuls les sept premiers résultats sont trouvés avec le premier algorithme. Les solutions suivantes sont trouvées au bout d'une minute 32 secondes, 2 minutes 46, 5 minutes 35, 20 minutes, etc.

```
R0 : Afficher les couples de nombres parfaits (N, M) avec 1 < N < M <= Max

R1 : Raffinage De « R0 »
      | Pour m ← 2 JusquÀ m = Max Faire
      |   | Déterminer la somme des diviseurs de m
      |   |                                     m: in ; somme_m: out Entier
      |   | Si somme_m <= m Alors        { somme_m est un nb amis potentiel }
      |   |   | Déterminer la somme des diviseurs de somme_m (somme_n)
      |   |   | Si somme_n = somme_m Alors        { somme_m et m amis }
      |   |   |   | Afficher le couple (somme_m, m)
      |   |   | FinSi
      |   | FinPour
      | FinPour

R2 : Raffinage De « somme des diviseurs de p »
      | somme ← 1
      | Pour i ← 2 JusquÀ racine carrée de p Faire
      |    | Si i diviseur de p Alors
      |    |    | somme ← somme + i + (p Div i)
      |    | FinSi
      | FinPour
      | Si p est un carré parfait Alors
      |    | somme ← somme - racine carrée de p
      | FinSi
```

Voici l'algorithme correspondant.

```
Algorithme nb_amis

   -- Afficher les couples de nombres amis (N, M) avec 1 < N < M <= MAX

Variables
    m: Entier              -- pour parcourir les entiers de 2 à MAX
    somme_m: Entier        -- somme des diviseurs de m
    somme_n: Entier        -- somme des diviseurs de somme_m correspondant à n
```

```
Début
    Pour m ← 2 JusquÀ m = MAX Faire
        -- Déterminer la somme des divieurs de m
        somme_m ← 1
        Pour i ← 2 JusquÀ racine carrée de m Faire
            Si i diviseur de m Alors
                somme_m ← somme_m + i + (m Div i)
            FinSi
        FinPour
        Si m est un carré parfait Alors
            somme_m ← somme_m - racine carrée de m
        FinSi

        { somme_m est la candidat pour n }


        Si somme_m <= m Alors   { on s'intéresse au cas  n <= m }
            -- Déterminer la somme des diviseurs de somme_m (somme_n)
            somme_n ← 1
            Pour i ← 2 JusquÀ racine carrée de somme_m Faire
                Si i diviseur de somme_m Alors
                    somme_n ← somme_n + i + (somme_m Div i)
                FinSi
            FinPour
            Si somme_m est un carré parfait Alors
                somme_n ← somme_n - racine carrée de somme_m
            FinSi

            Si somme_n = m Alors        { somme_m et m amis }
                Afficher le couple (somme_m, m)
            FinSi
        FinPour

    FinPour
Fin.
```

En C.

```c
/*****************************************************************************
 *  Auteur    : Xavier Crégut <cregut@enseeiht.fr>
 *  Version   : 1.1
 *
 *  Objectif : Afficher les couples de nombres amis (N, M)
 *       avec 1 < N < M <= MAX.
 ****************************************************************************/

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

#define MAX 1000000
```

```
int main()
{
    int m;              /* parcourir les entiers de 2 à MAX */
    int somme_m;        /* somme des diviseurs de m */
    int somme_n;        /* somme des diviseurs de somme_m correspondant à n */
    int racine;         /* la racine carrée d'un nombre */
    int i;              /* variable de boucle */

    for (m = 2; m <= MAX; m++) {
        /* Déterminer la somme des divieurs de m */
        somme_m = 1;
        racine = (int) sqrt(m);
        for (i = 2; i <= racine; i++) {
            if (m % i == 0) {   /* i diviseur de m */
                somme_m = somme_m + i + (m / i);
            }
        }
        if (m == racine * racine) {      /* m est un carré parfait */
            somme_m = somme_m - racine;
        }

        if (somme_m <= m) {      /* on s'intéresse au cas  n <= m */
            /* Déterminer la somme des diviseurs de somme_m (somme_n) */
            somme_n = 1;
            racine = (int) (sqrt(somme_m));
            for (i = 2; i <= racine; i++) {
                if (somme_m % i == 0) { /*  i diviseur de somme_m */
                    somme_n = somme_n + i + (somme_m / i);
                }
            }
            if (somme_m == racine * racine) {   /* carré parfait */
                somme_n = somme_n - racine;
            }

            /* déterminer si n et m sont amis */
            if (somme_n == m) {          /* n et m sont amis */
                printf("(%d, %d)\n", somme_m, m);
            }
        }
    }

    return EXIT_SUCCESS;
}

/*  Remarque : pour la structuration du programme, il serait préférable
 *  d'utiliser un sous-programme (une fonction) qui calcule la somme des
 *  diviseurs d'un entier n.
 */
```

Exercice 4 : Nombres de Amstrong

Les *nombres de Amstrong* appelés parfois *nombre cubes* sont des nombres entiers qui ont la particularité d'être égaux à la somme des cubes de leurs chiffres. Par exemple, 153 est un nombre de Amstrong car on a :

$$153 = 1^3 + 5^3 + 3^3.$$

Afficher tous les nombres de Amstrong sachant qu'ils sont tous compris entre 100 et 499.

Indication : Les nombres de Amstrong sont : 153, 370, 371 et 407.

Solution :

```
R0 : Afficher les nombres de Amstrong compris entre 100 et 499.
```

On peut envisager (au moins) deux solutions pour résoudre ce problème.

Solution 1. La première solution consiste à essayer les combinaisons de trois chiffres qui vérifient la propriété. On prend alors trois compteurs : un pour les unités, un pour les dizaines et un pour les centaines. Les deux premiers varient de 0 à 9. Le dernier de 1 à 4. Ayant les trois chiffres, on peut calculer la somme de leurs cubes puis le nombre qu'ils forment et on regarde si les deux sont égaux.

Ceci se formalise dans le raffinage suivant :

```
R1 : Raffinage De « Afficher les nombres de Amstrong »
    Pour centaine ← 1 JusquÀ centaine = 4 Faire
        Pour dizaine ← 1 JusquÀ dizaine = 9 Faire
            Pour unité ← 1 JusquÀ unité = 9 Faire
                Déterminer le cube
                Déterminer le nombre
                Si nombre = cube Alors
                    Afficher nombre
                FinSi
            FinPour
        FinPour
    FinPour
```

En C. On en déduit alors le programme C suivant.

```c
/*********************************************************************
 *  Auteur    : Xavier Crégut <cregut@enseeiht.fr>
 *  Version   : 1.1
 *  Objectif : Afficher les nombres de Amstrong
 *********************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main()
    /* Principe : on parcourt tous les trois chiffres du nombre
     * compris entre 100 et 499.
     */
{
```

```
    int nb;              /* le nombre considére */
    int centaine, dizaine, unite;      /* les trois chiffres de nb */
    int cube;            /* le somme des cubes des trois chiffres */

    for (centaine = 1; centaine <= 4; centaine++) {
        for (dizaine = 0; dizaine <= 9; dizaine++) {
            for (unite = 0; unite <= 9; unite++) {
                /* déterminer la somme des cubes */
                cube = centaine * centaine * centaine
                        + dizaine * dizaine * dizaine
                        + unite * unite * unite;

                /* déterminer le nombre */
                nb = centaine * 100 + dizaine * 10 + unite;

                if (nb == cube) {       /* c'est un nombre de Amstrong */
                    printf("%d\n", nb);
                }
            }
        }
    }

    return EXIT_SUCCESS;
}
```

On remarque que les opérations peuvent être réoganisées pour augmenter les performances en temps de calcul. En particulier, il est inutile de faire des calculs à l'intérieure d'une boucle s'ils ne dépendent pas de la boucle. Ainsi, le calcul du cube des centaines peut se faire dans la boucle la plus externe au lieu de le faire dans la plus interne.

En C. On arrive alors à une nouvelle version du programme.

```
/*********************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Afficher les nombres de Amstrong
 ********************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main()
    /* Principe : on parcourt tous les trois chiffres du nombre
     * compris entre 100 et 499.
     * On sort de la boucle interne certains calculs (centaine3,
     * dizaine3)
     */
{
    int nb;              /* le nombre considére */
    int centaine, dizaine, unite;      /* les trois chiffres de nb */
    int centaine3, dizaine3, unite3;   /* le cube des trois chiffres */
    int cube;            /* le somme des cubes des trois chiffres */
```

```
    for (centaine = 1; centaine <= 4; centaine++) {
        centaine3 = centaine * centaine * centaine;
        for (dizaine = 0; dizaine <= 9; dizaine++) {
            dizaine3 = dizaine * dizaine * dizaine;
            for (unite = 0; unite <= 9; unite++) {
                unite3 = unite * unite * unite;

                /* déterminer la somme des cubes */
                cube = centaine3 + dizaine3 + unite3;

                /* déterminer le nombre */
                nb = centaine * 100 + dizaine * 10 + unite;

                if (nb == cube) {       /* c'est un nombre de Amstrong */
                    printf("%d\n", nb);
                }
            }
        }
    }

    return EXIT_SUCCESS;
}
```

Solution 2. La deuxième solution consiste à parcourir tous les entiers compris entre 100 et 499 et à regarder s'ils sont égaux à la somme des cubes de leurs chiffres. La difficulté est alors d'extraite les chiffres. L'idée est d'utiliser la division entière par 10 et son reste.

On obtient le raffinage suivant.

```
R1 : Raffinage De « Afficher les nombres de Amstrong »
    Pour nombre ← 100 JusquÀ centaine = 499 Faire
        Déterminer les chiffres de nb
        Déterminer le cube des chiffres
        Si nombre = cube Alors
            Afficher nombre
        FinSi
    FinPour

R2 : Raffinage De « Déterminer les chiffres de nb »
    unité ← nombre Mod 10
    dizaine ← (nombre Div 10) Mod 10
    centaine ← nombre Div 100
```

En C. On en déduit alors le programme C suivant.

```
/*********************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Afficher les nombres de Amstrong
 ********************************************************************/
```

```c
#include <stdio.h>
#include <stdlib.h>

int main()
    /* Principe : on parcourt tous les entiers de 100 à 499 et on
     * vérifie s'il s'agit d'un nombre de Amstrong.
     */
{
    int nb;             /* le nombre considére */
    int centaine, dizaine, unite;       /* les trois chiffres de nb */
    int cube;           /* le somme des cubes des trois chiffres */

    for (nb = 100; nb <= 499; nb++) {
        /* déterminer les chiffres de nb */
        unite = nb % 10;
        dizaine = (nb / 10) % 10;
        centaine = nb / 100;

        /* déterminer le cube des chiffres */
        cube = centaine * centaine * centaine
               + dizaine * dizaine * dizaine
               + unite * unite * unite;

        if (nb == cube) {       /* c'est un nombre de Amstrong */
            printf("%d\n", nb);
        }
    }

    return EXIT_SUCCESS;
}
```

Exercice 5 : Puissance
Calculer et afficher la puissance entière d'un réel.

Solution : On peut, dans un premier temps, se demander si la puissance entière n d'un réel x a toujours un sens. En fait, ceci n'a pas de sens si x est nul et si n est strictement négatif. D'autre part, le cas $x = 0$, $n = 0$ est indéterminé. On choisit de contrôler la saisie de manière à garantir que nous ne sommes pas dans l'un de ces cas. Nous souhaitons donner à l'utilisateur un message le plus clair possible lorsque la saisie est invalide.

Nous envisageons les jeux de tests suivants :

```
  x   n  -->   x^n

  2   3  -->      8     -- cas nominal
  3   2  -->      9     -- cas nominal
  1   1  -->      1     -- cas nominal
  2  -3  --> 0.125      -- cas nominal (puissance négative)
  0   2  -->      0     -- cas nominal (x est nul)
  0  -2  --> ERREUR     -- division par zéro
  0   0  --> Indéterminé
1.5   2  --> 2.25       -- x peut être réel
```

Voyons maintenant le principe de la solution. Par exemple, pour calculer 2^3, on peut faire $2 * 2 * 2$. Plus généralement, on multiplie x n fois par lui-même (donc une boucle **Pour**).

Si on essaie ce principe, on constate qu'il ne fonctionne pas dans le cas d'une puissance négative. Prenons le cas de 2^{-3}. On peut le réécrire $(1/2)^3$. On est donc ramené au cas précédent. Le facteur multiplicatif est dans ce cas $1/x$ et le nombre de fois est $-n$.

Nous introduisons donc deux variables intermédiaires qui sont :

```
facteur: Réel    -- facteur multiplicatif pour obtenir les puissances
                 -- successives de x
puissance: Réel -- abs(n).  On a : facteur^puissance = x^n
```

On peut maintenant formaliser notre raisonnement sous la forme du raffinage suivant.

```
R0 : Afficher la puissance entière d'un réel

R1 : Raffinage De « R0 »
      | Saisir avec contrôle les valeurs de x et n       x: out Réel; n: out Entier
      | { (x <> 0) Ou (n > 0) }
      | Calculer x à la puisssance n                     x, n: in ; xn: out Réel
      | Afficher le résultat                             xn: in Réel

R2 : Raffinage De « Saisir avec contrôle les valeurs de x et n »
      | Répéter
      |    | Saisir la valeur de x et n            x: out Réel; n: out Entier
      |    | Contrôler x et n                      x, n: in ; valide: out Booléen
      | JusquÀ valide                              valide: in

R1 : Raffinage De « Calculer x à la puisssance n »
      | Si x = 0 Alors
      |    | xn := 0
      | Sinon
```

```
        |    | Déterminer le facteur multiplicatif et la puissance
        |    |                     x, n: in ;
        |    |                     facteur out Réel ;
        |    |                     puissance: out Entier
        |    | Calculer xn par itération (accumulation)
        |    |                     n, facteur, puissance: in ; xn : out
        | FinSi

  R3 : Raffinage De « Calculer xn par itération (accumulation) »
        | xn ← 1;
        | Pour i ← 1 JusquÀ i = puissance Faire
        |    | { Invariant : xn = facteur^i }
        |    | xn ← xn * facteur
        | FinPour
```

On peut alors en déduire l'algorithme suivant :

```
Algorithme puissance

    -- Afficher la puissance entière d'un réel

Variables
    x: Réel                 -- valeur réelle lue au clavier
    n: Entier               -- valeur entière lue au clavier
    valide: Booléen         -- x^n peut-elle être calculée
    xn: Réel                -- x à la puissance n
    facteur: Réel           -- facteur multiplicatif pour obtenir
                            -- les puissances successives
    puissance: Entier;      - abs(n).  On a facteur^puissance = x^n
    i: Entier               -- variable de boucle

Début
    -- Saisir avec contrôle les valeurs de x et n
    Répéter
        -- saisir la valeur de x et n
        Écrire("x = ")
        Lire(x)
        Écrire("n = ")
        Lire(n)

        -- contrôler x et n
        valide ← VRAI
        Si x = 0 Alors
            Si n = 0 Alors
                ÉcrireLn("x et n sont nuls. x^n est indéterminée.")
                valide ← FAUX
            SinonSi n < 0 Alors
                ÉcrireLn("x nul et n négatif. x^n n'a pas de sens.")
                valide ← FAUX
            FinSi
        FinSi
```

```
        Si Non valide Alors
            ÉcrireLn("␣␣Recommencez␣!")
        FinSi
    JusquÀ valide

    -- Calculer x à la puisssance n
    Si x = 0 Alors      -- cas trivial
        xn ← 0
    Sinon
        -- Déterminer le facteur multiplicatif et la puissance
        Si n >= 0 Alors
            facteur ← x
            puissance ← n
        Sinon
            facteur ← 1/x
            puissance ← -n
        FinSi

        -- Calculer xn par itération (accumulation)
        xn ← 1;
        Pour i ← 1 JusquÀ i = puissance Faire
            { Invariant : xn = facteur^i }
            xn ← xn * facteur
        FinPour
    FinSi

    -- Afficher le résultat
    ÉcrireLn(x, "^", n, "␣=␣", xn)
Fin.
```

En C.

```c
/*********************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Afficher la puissance entière d'un réel
 ********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main()
{
    double x;           /* valeur réelle lue au clavier */
    int n;              /* valeur entière lue au clavier */
    bool valide;        /* x^n peut-elle être calculée */
    double xn;          /* x à la puissance n */
    double facteur;     /* facteur multiplicatif pour
                         * obtenir les puissances successives
                         */
    int puissance;      /* abs(n).  On a : facteur^puissance == x^n */
```

```
int i;              /* variable de boucle */

/* Saisir avec contrôle les valeurs de x et n */
do {
    /* saisir la valeur de x et n */
    printf("x␣=␣");
    scanf("%lf", &x);
    printf("n␣=␣");
    scanf("%d", &n);

    /* contrôler x et n */
    valide = true;
    if (x == 0) {
        if (n == 0) {
            printf("x␣et␣n␣sont␣nuls.␣x^n␣est␣indéterminée.");
            valide = false;
        }
        else if (n < 0) {
            printf("x␣nul␣et␣n␣négatif.␣x^n␣n'a␣pas␣de␣sens.");
            valide = false;
        }
    }
    if (!valide) {
        printf("␣␣Recommencez␣!\n");
    }
} while (!valide);

/* Calculer x à la puisssance n */
if (x == 0) {       /* cas trivial */
    xn = 0;
}
else {
    /* Déterminer le facteur multiplicatif et la puissance */
    if (n >= 0) {
        facteur = x;
        puissance = n;
    }
    else {
        facteur = 1/x;
        puissance = -n;
    }

    /* Calculer xn par itération (accumulation) */
    xn = 1;
    for (i = 1; i <= puissance; i++) {
        /*{ Invariant : xn = facteur^{*i} }*/
        xn = xn * facteur;
    }
}

/* Afficher le résultat */
```

```
    printf("%f^%d␣=␣%f\n", x, n, xn);

    return EXIT_SUCCESS;
}
```

Exercice 6 : Amélioration du calcul de la puissance entière
On se propose d'améliorer l'algorithme de calcul de la puissance (exercice 5) en remarquant que

$$x^n = \begin{cases} (x^p)^2 & \text{si } n = 2p \\ (x^p)^2 \times x & \text{si } n = 2p + 1 \end{cases}$$

Ainsi, pour calculer 3^5, on peut faire $3 * 9 * 9$ avec bien sûr $9 = 3^2$.
Solution : Nous nous appuyons sur les mêmes variables que pour le calcul « naturel » de la puissance (exercice 5). Nous allons continuer à calculer x^n par accumulation. Nous avons l'invariant suivant :

$$x^n = \text{xn} * \text{facteur}^{\text{puissance}}$$

Lors de l'initialisation, cet invariant est vrai (xn vaut 1 et facteur et puissance sont tels qu'ils valent x^n).

D'après la formule donnée dans l'énoncé, à chaque itération de la boucle, deux cas sont à envisager :

– soit puissance est paire. On peut alors l'écrire $2 * p$. On a alors :

$$\begin{aligned} x^n &= \text{xn} * \text{facteur}^{\text{puissance}} \\ &= \text{xn} * \text{facteur}^{(2 * \text{p})} \\ &= \text{xn} * (\text{facteur}^2)^{\text{p}} \end{aligned}$$

On peut donc faire :

```
facteur <- facteur * facteur
puissance <- puissance Div 2        -- car p = puissance Div 2
```

On constate que l'invariant est préservé d'après les égalités ci-dessus et la puissance a strictement diminué (car divisée par 2).

– soit puissance est impaire. On peut alors l'écrire $2 * p + 1$. On a alors :

$$\begin{aligned} x^n &= \text{xn} * \text{facteur}^{\text{puissance}} \\ &= \text{xn} * \text{facteur}^{(2 * \text{p} + 1)} \\ &= \text{xn} * (\text{facteur} * \text{facteur}^{(2*\text{p})}) \\ &= (\text{xn} * \text{facteur}) * \text{facteur}^{(2*\text{p})} \end{aligned}$$

On peut donc faire :

```
xn <- xn * facteur
puissance <- puissance - 1
```

On constate que l'invariant est préservé d'après les égalités ci-dessus et la puissance a strictement diminué (car diminuée de 1).

On peut alors en déduire le raffinage suivant :

```
R3 : Raffinage De « Calculer xn par itération (accumulation) »
    | xn <- 1;
    | TantQue puissance > 0 Faire
    |    | { Variant : puissance }
```

```
      |   | { Invariant : x^n = xn * facteur^puissance }
      |   | Si puissance Div 2 = 0 Alors          { puissance = 2 * p }
      |   |   | puissance ← puissance Div 2
      |   |   | facteur ← facteur * facteur
      |   | Sinon                                  { puissance = 2 * p + 1 }
      |   |   | puissance ← puissance - 1
      |   |   | xn ← xn * facteur
      |   | FinSi
      | FinTQ
```

En C.

```c
/*********************************************************************
 *  Auteur    : Xavier Crégut <cregut@enseeiht.fr>
 *  Version   : 1.1
 *  Objectif : Afficher la puissance entière d'un réel
 ********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main()
{
    double x;           /* valeur réelle lue au clavier */
    int n;              /* valeur entière lue au clavier */
    bool valide;        /* x^n peut-elle être calculée */
    double xn;          /* x à la puissance n */
    double facteur;     /* facteur multiplicatif pour
                         * obtenir les puissances successives
                         */
    int puissance;      /* abs(n).  On a : facteur^puissance == x^n */

    /* Saisir avec contrôle les valeurs de x et n */
    do {
        /* saisir la valeur de x et n */
        printf("x = ");
        scanf("%lf", &x);
        printf("n = ");
        scanf("%d", &n);

        /* contrôler x et n */
        valide = true;
        if (x == 0) {
            if (n == 0) {
                printf("x et n sont nuls. x^n est indéterminée.");
                valide = false;
            }
            else if (n < 0) {
                printf("x nul et n négatif. x^n n'a pas de sens.");
                valide = false;
            }
```

```
        }
        if (!valide) {
            printf("␣␣Recommencez␣!\n");
        }
    } while (!valide);

    /* Calculer x à la puisssance n */
    if (x == 0) {       /* cas trivial */
        xn = 0;
    }
    else {
        /* Déterminer le facteur multiplicatif et la puissance */
        if (n >= 0) {
            facteur = x;
            puissance = n;
        }
        else {
            facteur = 1/x;
            puissance = -n;
        }

        /* Calculer xn par itération (accumulation) */
        xn = 1;
        while (puissance > 0) {
            /*{ Invariant : xn * facteur ^ puissance == x ^ n }*/
            /*{ Variant : puissance }*/
            if (puissance % 2 == 0) {
                facteur = facteur * facteur;
                puissance = puissance / 2;
            }
            else {
                xn = xn * facteur;
                puissance = puissance - 1;
            }
        }
    }

    /* Afficher le résultat */
    printf("%f^%d␣=␣%f\n", x, n, xn);

    return EXIT_SUCCESS;
}
```