Compte rendu - TP1

Berquier Raphaël - B2

Exercice 1

Recherche

Consignes : Calculer la somme des n premiers nombres impairs
Le programme demandera à l'utilisateur de saisir la valeur de n et affichera le résultat.

La construction de ce programme à commencé par la recherche d'un algorithme pour trouver les nombres impairs.
Les premiers nombres impairs sont :

- 1
- 3
- 5
- 7
- ...

Une façon d'écrire les nombres impairs est de prendre un nombre pair et d'y ajouter un nombre impair.
Un nombre pair est un multiple de 2, alors tout nombre pair peut s'écrire sous la forme : $n \times 2$.
Pour tout n entier.
Ex :

- $0 = 0 \times 2$ -> n = 0
- $2 = 1 \times 2$ -> n = 1
- $4 = 2 \times 2$ -> n = 2

Ainsi, il suffit d'ajouter un nombre impair à $n \times 2$ pour avoir un nombre impair, la solution la plus évidente est d'ajouter 1. Une formule pour trouver un nombre impair est : $n \times 2 + 1$.

J'ai commencé par afficher dans la console tous les nombres impairs inférieur à 10 à l'aide de l'algorithme suivant :

```c
int main() {
    for (int n = 0; n < 10; n++)
    {
        printf("%d\n", n*2 + 1);
    }
}
```

Console :

```
1
3
5
7
9
11
13
15
17
19
```

Le programme renvoie les 10 premiers nombres impairs, alors l'algorithme fonctionne correctement.

> **Remarque** :
> Une autre formule pour trouver la somme des nombres impairs est n², cette solution ne sera pas exploré dans ce compte rendu.

Creation du programme

Une fois l'algorithme créé, j'ai repris le code précédent et au lieu d'afficher le résultat dans la console je l'ajoute dans une variable de type entier nommé `sum`.
Pour rajouter l'interaction avec l'utilisateur, au début du programme je demande le nombre de nombre négatif à ajouter, cette valeur stocké dans `n` est ensuite utilisé en tant que limite pour la boucle au lieu de 10.

Pour améliorer la robustesse du programme, j'ai rajouter une boucle du type `do while` qui demande à l'utilisateur la valeur de n tant qu'elle est inférieur à 0. Cela permet de m'assurer que n est positif.

Ici, une valeur de 0 ne pose aucun problème puisque la boucle qui fait la somme ne s'execute pas, renvoyant la valeur par défaut de `sum`, qui est ici 0. Et 0 est bien la somme des 0 premier nombre impair.

Programme final :

```c
#include <stdio.h>

int main() {
    int n = 0;
    do
    {
        printf("Entrer n : ");
        scanf("%d", &n);
        if (n < 0)
        {
            printf("Veuillez entrer un nombre positif.\n");
        }

    } while (n< 0);

    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        sum += (i*2) + 1;
    }

    printf("La somme des %d premiers nombres impairs est : %d \n", n, sum);
    return 0;
}
```

Exercice 2

Consigne : Recherche des nombres parfaits inférieure à une valeur limite saisie par l'utilisateur.

Recherche

Pour commencer, j'ai recherché la définition d'un nombre parfait, et les premiers nombres parfait pour comprendre l'énoncé.

Définition

> Nombre parfait — Wikipédia (wikipedia.org)
>
> En arithmétique, un **nombre parfait** est un entier naturel égal à la moitié de la somme de ses diviseurs ou encore à la somme de ses diviseurs stricts. Plus formellement, un nombre parfait *n* est un entier tel que σ(*n*) = 2*n* où σ(*n*) est la somme des diviseurs positifs de *n*. Ainsi 6 est un nombre parfait car ses diviseurs entiers sont 1, 2, 3 et 6, et il vérifie bien 2 × 6 = 12 = 1 + 2 + 3 + 6, ou encore 6 = 1 + 2 + 3.

Exemples

Les premiers nombres parfaits sont :

- 6
- 28
- 496
- 8128

Recherche des diviseurs

Pour savoir si un nombre est parfait ou pas, il me faut la liste des ses diviseurs pour pouvoir les ajouter et vérifier que cette somme est égal au nombre de départ.

Pour deux nombres n_1 et n_2, pour savoir si n_1 est divisible par n_2 on fait la division euclidienne des deux nombres et on regarde le reste, s'il est égal à 0, alors n_1 est divisible par n_2 , sinon n_1 n'est pas divisible par n_2.

En C, et dans la plupart des langages de programmation il existe le modulo qui permet d'obtenir le reste de la division euclidienne entre 2 nombres.
Après quelques recherche je trouve que pour utiliser le modulo il faut utiliser le symbole `%` entre deux nombres.
Source : Modulo Operator (%) in C/C++ with Examples - GeeksforGeeks

J'ai fait des tests afin de m'assurer du bon fonctionnement du modulo :

```c
int main() {
    printf("%d\n", 3%2);
    printf("%d\n", 6%3);
    printf("%d\n", 2%1);
}
```

Et j'ai obtenu les résultats suivant dans la console :

```
1
0
0
```

Une fois mes tests validés. J'ai commencé l'élaboration d'une fonction permettant de trouver tous les diviseurs :

```c
int main() {
    int number = 0;
    printf("n : ");
    scanf("%d", &number);
    for (int i = 1 ; i < number ; i++)
    {
        if ((number%i)==0)
            printf("%d \n", i);

    }
}
```

Console :

```
n : 6
1
2
3
```

Dans la console on obtient bien les diviseurs de 6 (sauf 6).

Il restait à ajouter tous ces diviseurs et à vérifier l'égalité avec le nombre de départ pour savoir si un nombre est parfait.
J'ai décidé de faire une fonction nommé `IsPerfectNumber` (en anglais pour suivre les conventions classiques).

Cette fonction renvoie 1 si le nombre en argument est parfait, sinon 0. (Une variante avec des booléen est tout à fait envisageable est permettrait sûrement une meilleure lisibilité de la fonction)

```
int IsPerfectNumber(int number){
    int sum = 0;
    for (int i = 1 ; i < number ; i++)
    {
        if ((number%i)==0)
            sum += i;

    }

    if (sum == number)
        return 1;

    return 0;
}
```

Recherche de tous les nombres parfaits

Une fois cette fonction faite, il me restait à demander à l'utilisateur un nombre comme limite de recherche et dans un boucle d'appeler la fonction `IsPrimeNumber` avec en argument l'index de la boucle.

Code final :

```c
#include <stdio.h>

int IsPerfectNumber(int number){
    int sum = 0;
    for (int i = 1 ; i < number ; i++)
    {
        if ((number%i)==0)
            sum += i;
    }

    if (sum == number)
        return 1;

    return 0;
}

int main()
{
    int n = 0;
    do
    {
        printf("Entrer n : ");
        scanf("%d", &n);
        if (n < 0)
            printf("Veuillez entrer un nombre positif.\n");

    } while (n< 0);

    printf("La liste des nombres parfaits inférieurs à %d est :\n", n);
    for (int i = 1; i < n; i++)
    {
        if(IsPerfectNumber(i) == 1)
            printf("- %d \n", i);
    }
}
```

> Remarque :
> L'utilisation d'un prototype (déclaration d'une fonction avant la fonction main) n'est pas nécessaire puisque dans l'execution du script, le fonction `main` appelle uniquement la fonction `IsPerfectNumber`. Si ces fonctions venaient à appeler d'autres fonctions, ces prototypes auraient sûrement été nécessaire.

Pour m'assurer de la robustesse du programme j'ai ajouté comme à l'exercice précédent une boucle `do while` permettant de m'assurer que la valeur de n est bien positive.

J'ai aussi fait commencer la boucle à 1, puisque 0 n'est pas considéré comme un nombre parfait, même si la somme de ses diviseurs (0) est bien égal à lui même.

Exercice 3

Consignes : Résolution d'une équation du second degré Ecrire un programme permettant de résoudre une équation du second degré. Les paramètres de ce programme seront les coefficients a, b et c de l'équation ax2+bx+c=0. Vous envisagerez tous les cas de figure, y compris les cas ou certains coefficients sont nuls.

Recherche

J'ai d'abord commencé par définir les étapes de résolution d'une équation du second degré :

- Demande à l'utilisateur les valeurs de a, b, c
- Calcul du discriminant (delta Δ)
- 3 cas de figures selon la valeur de Δ
  - Strictement positif
  - Egal à 0
  - Strictement négatif

J'ai séparé en 3 cas, car pour chacun des ces 3 cas, la solution de l'équation est différente.

- $\Delta > 0$:
  - $Solution1 = \frac{-b - \sqrt{\Delta}}{2a}$
  - $Solution2 = \frac{-b + \sqrt{\Delta}}{2a}$
- Δ=0 :
  - $Solution = \frac{-b}{2a}$
- $\Delta < 0$:
  - $Solution1 = \frac{-b - i\sqrt{\Delta}}{2a}$
  - $Solution2 = \frac{-b + i\sqrt{\Delta}}{2a}$

Demandes à l'utilisateur

Pour demander à l'utilisateur j'ai choisi l'utilisation du type `double`, car ce type permet de stocker des nombres flottants négatif et positif avec comme seul contrainte d'une place plus important dans la mémoire vive. Dans notre cas, ce programme n'a pas pour but d'être installé sur des machines disposant de mémoire vive très faible (ce qui est le cas pour des Arduino ou raspberry pi par exemple). Ainsi l'utilisation du type `double` ne pose aucun soucis.

```
    double a = 0;
    double b = 0;
    double c = 0;

    printf("Entrer a : ");
    scanf("%lf", &a);

    printf("Entrer b : ");
    scanf("%lf", &b);

    printf("Entrer c : ");
    scanf("%lf", &c);

    printf("Resolution de l equation : %gx^2 + %gx + %g = 0\n", a, b ,c);
```

> Remarque :
> %g permet d'afficher un flottant sans afficher les 0 inutiles par exemple au lieu d'afficher 1.000000, cela affichera 1. Cela permet une meilleure lisibilité dans le terminal pour l'utilisateur.

Delta

Le discriminant ou delta est défini par la formule
$\Delta = b^2 - 4ac$

Calcul Δ :

```
double d = b*b - 4*a*c;
```

Une fois le discriminant obtenu, il reste à faire les 3 cas précédemment appelé. A l'aide des conditions cela se fait très simplement.

La plus grande problématique c'est posé lors du calcul des solutions puisqu'elles nécessitent l'utilisation de la racine carré ($\sqrt{\Delta}$). Cependant pour avoir cette opération l'ajout d'une librairie est nécessaire.

Import de la librairie de maths :

```c
#include <math.h>
```

Lors de la compilation une erreur se produisait, elle venait de l'utilisation de la fonction `sqrt` provenant de `math.h` (sqrt signifie square root, soit racine carré en français).
Après quelques recherches sur internet, j'ai trouvé qu'il fallait rajouter `-lm` en argument lors de la compilation. Cela est nécessaire car l'implémentation de GCC utiliser sur l'ordinateur ne lie pas automatiquement `math.h`.

Code pour trouver les solutions :

```c
if (d > 0) // 2 solutions
    {
        float s1 = (- b - sqrt(d))/(2 * a);
        float s2 = (- b + sqrt(d))/(2 * a);

        printf("Les 2 solutions de l equation sont : \n");
        printf("%.5g \n", s1);
        printf("%.5g \n", s2);

    }
    else if (d == 0) // 1 solution
    {
        float s = -b/(2 * a);

        printf("La solution de l equation est : \n");
        printf("%.5g \n", s);

    }
    else // 2 solutions dans C
    {
        float m = (- b)/(2 * a);
        float q = (sqrt(-d))/(2 * a);

        printf("Les 2 solutions de l equation sont : \n");
        printf("%.5g - %.5gi \n", m, q);
        printf("%.5g + %.5gi \n", m, q);
    }
```

> Remarque :
> %.5g, permet d'afficher un flottant sans les zéros et avec un maximum de 5 nombres après la virgule.

Les solutions pour $\Delta = 0$ ou $\Delta > 0$ sont plutôt évidente puisqu'il suffit d'appliquer les formules en faisant attention aux priorités des opérations.

Quant aux nombres complexes cas où $\Delta < 0$, il n'est pas nécessaire d'utiliser de librairie pour la gestion des nombres complexes, car aucune calcul entre complexe est effectué.
On peut d'abord calculer une solution ($z = m + iq$) et en deuxième solution le conjugué de cette solution : $\bar{z} = m - iq$.

Exercice 4

L'exercice 4 consiste est le débogage de 2 programmes.

eq2degree.c

Ce programme est similaire à l'exercice 3, il a pour but d'afficher les solutions d'une équation du 2nd degré.

Dans un premier temps j'ai essayer de compiler le programme pour voir s'il y a des erreurs.
Dans la console j'ai obtenu :

```
q2degre.c: Dans la fonction « main »:
eq2degre.c:30:23: erreur: expected « ) » before « printf »
   30 |                 if(d<d
      |                   ~   ^
      |                       )
   31 |                         printf("\n 2 solutions complexes conjuguées:
(-%d+i racine(%d)) /%d et (-%d-i racine(%d)) /%d",b,-d,2*a,b,-d,2*a);
      |                         ~~~~~~
eq2degre.c:38:9: erreur: expected expression before « } » token
   38 |         }
      |
```

Ce message indique que dans la fonction main, à la ligne 30, caractère 23 il y a une erreur.

L'erreur provient d'un d qui remplace `)`
Correction :

```
    if(d<0d // correction : if d<0)
                printf("\n 2 solutions complexes conjuguées: (-%d+i
racine(%d)) /%d et (-%d-i racine(%d)) /%d",b,-d,2*a,b,-d,2*a);
        else
```

Ensuite le programme se compilait sans erreur.
En regardant le programme plus en détail j'ai remarque qu'il manquait un '=' dans une condition.

J'ai ensuite exécuté le programme et après avoir entré la valeur de A, j'ai eu l'erreur suivante :

```
Erreur de segmentation (core dumped)
```

Cette erreur es dû à l'oublie du caractère `&` devant la valeur de A. Il manquait aussi ce caractère devant B et C.
La dernière erreur est lors de l'affichage des solutions, dans les racines il y a des 0. Pour trouver la source du problème, j'ai d'abord vérifié si D était à 0 à l'aide d'un print, mais il avait une valeur correcte, cela m'a permet de déduire que le problème venait du print. D est du type `double` mais lors de l'affichage par le print il est interprété comme un `int` puisqu'il est écrit `%d`. Pour régler il faut remplacer `%d` par `%f`, à tous les endroits au D est affiché.

Ensuite j'ai testé le programme dans tous les cas de D différent, et toutes les valeurs étaient correctes.

Code corrigé :

```
// resolution d'une equation du 2e degre
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main()
{
        //déclaration des variables
        int a,b,c;
        float d = 0;
        //acquisition des variables
        printf("\n Ax²+Bx+C=0");
        printf("\n entrez A: ");
        scanf("%d",&a); // & manquant
        printf("\n entrez B: ");
        scanf("%d",&b); // & manquant
        printf("\n entrez C: ");
        scanf("%d",&c); // & manquant

        //Condition invalidant la suite du prgm
        if(a==0) // Oublie d'un = apres le a
        {
                printf("\n ce n'est plus du 2nd degré");
                return 0;
        }
        else
        {
                d=(b*b)-(4.0*a*c);
                printf("delta : %d", d);
                if(d<0) // d remplacé par )
                        printf("\n 2 solutions complexes conjuguées: (-%d+i
racine(%f)) /%d et (-%d-i racine(%f)) /%d",b,-d,2*a,b,-d,2*a); // modification
de %f dans les racines car d est un double
                else
                        if(d>0)
                                printf("\n 2 solutions réelles: (-
%d+racine(%f)) /%d et (-%d-racine(%f)) /%d",b,d,2*a,b,d,2*a); // modification
de %f dans les racines car d est un double
                        else
                                printf("\n 1 solution double :-%d /%d",b,2*a);

        }
        printf("\n");
```

```
        return 1;
}
```

monnaie.c

Comme pour le premier code je l'ai compilé et j'ai réglé les erreurs :

- `scanf("%fls",billet);` ajout de & devant la variable `billet`
- `scanf("%f",&billet);` lettre non nécessaire après le `%f`
- `printf("\nValeur saisie : %f", &billet);` signe & en trop
- Rajout de `int` devant la fonction main pour indiquer le type de sortie (non nécessaire mais permet de clarifier le programme)
- `return 1;` -> `return 0` plus logique pour dire qu'il n'y a pas d'erreur
- `while (billet)` ; transformé en -> `while(billet > 0);` pour rend plus clair la condition de la boucle.

Pour m'assurer que le programme fonctionne correctement, j'ai testé avec plusieurs valeurs et le programme fonctionne correctement à part le nombre de pièce de 1€ qui n'était pas affiché. Il manquait `%d` dans le printf.

Code corrigé :

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


int main() // manque du int, créant  un warning lors de la compilation
{

        /* variables */

        float  billet ; /* billet dont on veut faire la monnaie*/
        int nbp2 ; /* nb de pieces de 2€*/
        int nbp1 ;
        int nbp50;
        int nbp20;
        int nbp10;
        int nbp5;
        int nbp2c;
        int nbp1c;
        int diviseur ;

        /* Initialisation*/



        printf(" \n\n Saisir la valeur d'un billet (entier superieur  a 2 €)
:");
        scanf("%f",&billet); // manque d'un & devant billet
        fflush(stdin);
        printf("\nValeur saisie : %f", billet); // & en trop devant billet, et
changement de %fls en %f pour correspondre à un double

        diviseur=200;
        billet=billet*100; /* pour tout convertir en centimes d'euros*/

        /* Traitement */

        do
        {

                switch (diviseur) {
                        case 200 :  /* pieces de 2€ */
                                nbp2=billet/diviseur;
                                billet=(int)billet%diviseur;
```

```
                                diviseur=100;
                                printf("\n Il y a %d pieces de 2€",nbp2);
                                break;
                        case 100 :   /* pieces de 1€ */
                                nbp1=(int)billet/diviseur;
                                billet=(int)billet%diviseur;
                                diviseur=50;
                                printf("\n Il y a %d pieces de 1€",nbp1); //
oublie de %d au milieu du printf
                                break;
                        case 50 :  /* pieces de 50 cts */
                                nbp50=(int)billet/diviseur;
                                billet=(int)billet%diviseur;
                                diviseur=20;
                                printf("\n Il y a %d pieces de 50 cts",nbp50);
                                break;
                        case 20 :   /* pieces de 20 cts */
                                nbp20=(int)billet/diviseur;
                                billet=(int)billet%diviseur;
                                diviseur=10;
                                printf("\n Il y a %d pieces de 20 cts",nbp20);
                                break;
                        case 10 :  /* pieces de 10 cts */
                                nbp10=(int)billet/diviseur;
                                billet=(int)billet%diviseur;
                                diviseur=5;
                                printf("\n Il y a %d pieces de 10 cts",nbp10);
                                break;
                        case 5 :   /* pieces de 5 cts */
                                nbp5=(int)billet/diviseur;
                                billet=(int)billet%diviseur;
                                diviseur=2;
                                printf("\n Il y a %d pieces de 5 cts",nbp5);
                                break;
                        case 2 :  /* pieces de 2 cts */
                                nbp2c=(int)billet/diviseur;
                                billet=(int)billet%diviseur;
                                diviseur=1;
                                printf("\n Il y a %d pieces de 2 cts",nbp2c);
                                break;
                        case 1 :   /* pieces de 1 cts */
                                nbp1c=(int)billet/diviseur;
                                billet=(int)billet%diviseur;
                                diviseur=1;
                                printf("\n Il y a %d pieces de 1 cts",nbp1c);
                                break;
```

```
                        default : printf("\n\n Cas imprevu !!! Il y a une
erreur");
                                }
                }
        while (billet>0); // (billet > 0) plus logique que juste (billet) meme
si les 2 fonctionne dans la cas de nombre positif

        return 0; // return 0 au lieu de 1, car il n'y a pas d'erreur

}
```