TD XV

PILE D'APPEL, ENSEMBLE DE MANDELBROT

1 Retour sur `scanf`

Exercice XV.1

Si vous n'avez pas eu le temps de la traiter, reprendre la partie sur `scanf` du TP précédent.

2 Quelques expériences sur la pile et les pointeurs

Exercice XV.2 p. 8

On considère le programme suivant :

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

double expo(double x, int n){
    double res = 1.0;
    for (int i = 0; i < n; i = i + 1){
        res = res * x;
    }
    return res;
}

double f(double x, double y, int n){
    double xn = expo(x, n);
    double yn = expo(y, n);
    return xn + yn;
}

int main(int argc, char* argv[]){
    if (argc != 4) return -1;                                   // *
    int n = atoi(argv[1]);
    double x = atof(argv[2]);                                   // *
    double y = atof(argv[3]);
    printf("%f^%d + %f^%d = %f\n", x, n, y, n, f(x, y, n)); // *
    return 0;                                                   // *
}
```

1. Combien d'arguments ce programme attend-il en ligne de commande ? De quels types sont-ils ?

On suppose dans la suite que les fonctions de la bibliothèque standard appelées (`atoi`, `atof`, `printf`) n'appellent pas elles-mêmes de fonctions.

2. Lors de l'exécution de la ligne 20, quelle est la hauteur de la pile (en nombre de blocs d'activation) ?

3. Quelle est la hauteur maximale de la pile lors de l'exécution de la ligne 22 ?
4. Même question pour la ligne 24.
5. Même question pour la ligne 25.

Exercice XV.3

p. 8

On considère le code suivant :

```c
#include <stdio.h>

void f(int n, int* nmax){
    printf("Début de l'appel de f(%d, _)\n", n); // *
    printf("n     = %d\n", n);
    printf("&n    = %p\n", &n);
    printf("nmax  = %p\n", nmax);
    printf("*nmax = %d\n", *nmax);
    printf("&nmax = %p\n", &nmax);
    if (n < *nmax) f(n + 1, nmax);
    printf("Fin de l'appel de f(%d, _)\n", n);  // *
}

int main(void){
    int N = 2;
    f(0, &N);
    return 0;
}
```

1. En oubliant les `printf` situés ailleurs qu'aux lignes 4 et 11, prévoir l'affichage produit par la fonction.
2. Parmi les autres lignes provoquant un affichage, lesquelles :
   - affichent toujours la même chose ?
   - affichent des choses différentes, mais que l'on peut prévoir parfaitement en regardant le code ?
   - affichent des choses différentes et partiellement imprévisibles ?
3. Prévoir le plus complètement possible l'affichage produit par le programme.
4. Exécuter ce programme (dans le terminal et éventuellement avec *C Tutor*) et observer l'affichage produit.
5. Quelle est la taille (en octets) du bloc d'activation de `f` ?

Exercice XV.4

p. 9

1. Écrire une fonction `incremente` qui prend en entrée un pointeur vers un entier et incrémente la valeur de cet entier de 1 (cette fonction ne renverra rien).
2. Dans tous les exemples suivants, on souhaite qu'un appel `f(px, py)` incrémente celle des valeurs pointées par `px` et `py` qui est la plus petite (ou celle pointée par `px` en cas d'égalité). Par exemple :

```c
#include <stdio.h>

int main(void){
    int x = 4;
    int y = 3;
    printf("x = %d, y = %d\n", x, y);
    f(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    f(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

doit donner l'affichage suivant :

```
x = 4, y = 3
x = 4, y = 4
x = 5, y = 4
```

Dans chaque cas, dire si :

- il y a un problème de type (et si oui, où) ;
- les fonctions ont bien le comportement attendu (uniquement dans les cas où il n'y a pas de problème de type). Si ce n'est pas le cas, on expliquera d'où vient le problème.

a.

```c
int plus_petit(int x, int y){
    if (x <= y) return x;
    return y;
}

void f(int* px, int* py){
    incremente(plus_petit(*px, *py));
}
```

b.

```c
int* plus_petit(int x, int y){
    if (x <= y) return &x;
    return &y;
}

void f(int* px, int* py){
    incremente(plus_petit(*px, *py));
}
```

c.

```c
int* plus_petit2(int* x, int* y){
    if (*x <= *y) return x;
    return y;
}

void f(int* px, int* py){
    incremente(plus_petit(px, py));
}
```

d.

```c
int* plus_petit(int* x, int* y){
    int a = *x;
    int b = *y;
    if (a <= b) return &a;
    return &b;
}

void f(int* px, int* py){
    incremente(plus_petit(px, py));
}
```

3 Fractale de Mandelbrot

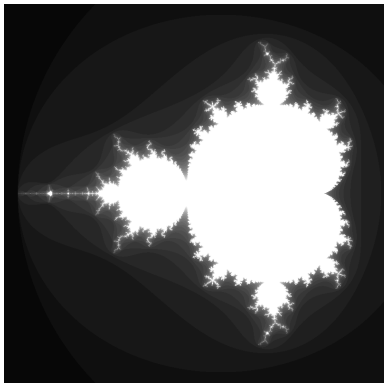

Figure XV.1 – L'ensemble de Mandelbrot.

Pour $c \in \mathbb{C}$, on considère la fonction :

$$\begin{array}{rccc} f_c : & \mathbb{C} & \to & \mathbb{C} \\ & z & \mapsto & z^2 + c \end{array}$$

À partir de cette fonction, on définit la suite $(z_n(c))$ par :

$$\begin{cases} z_0(c) = 0 \\ z_{n+1}(c) = f_c(z_n(c)) \quad \text{pour } n \geqslant 0 \end{cases}$$

L'*ensemble de Mandelbrot* est l'ensemble des complexes c tels que $(u_n(c))$ soit bornée. Dans la suite, on note $\mathcal{M}$ cet ensemble.

Exercice XV.5 – Bornes

Cette question est purement mathématique : il peut être pertinent de la traiter plus tard, chez vous, et d'admettre le résultat de la dernière question pour la suite.

1. Soit $c \in \mathbb{C}$ et $n \in \mathbb{N}$. On note $z_n = z_n(c)$ et l'on suppose $|z_n| > 2$ et $|z_n| > c$.
   a. Montrer que pour $m \geqslant 0$, on a $|z_{n+m}| \geqslant |c| + 2^m (|z_n| - |c|)$.
   b. En déduire que $c \notin \mathcal{M}$.
2. Montrer que si $|c| > 2$, alors $|z_1| > |c|$.
3. Montrer que $c \in \mathcal{M} \iff (\forall n \in \mathbb{N}, |z_n| \leqslant 2)$.

On définit désormais $\mathcal{M}_N = \{c \in \mathbb{C} \mid \forall n \leqslant N,\ |z_n(c)| \leqslant 2\}$. D'après ce qui précède, on a :

- $\forall N \in \mathbb{N}$, $\mathcal{M} \subset \mathcal{M}_N$;
- $\mathcal{M} = \bigcap_{N \in \mathbb{N}} \mathcal{M}_N$

Pour N suffisamment grand, $\mathcal{M}_N$ sera une bonne approximation de $\mathcal{M}$. Pour déterminer si $c \in \mathcal{M}$, on pourra donc se fixer un entier `itermax` puis calculer les valeurs successives de $z_n(c)$:

- dès que l'une de ces valeurs a un module strictement supérieur à 2, on sait que $c \notin \mathcal{M}$;
- si toutes les valeurs jusqu'à $n = $ `itermax` ont un module inférieur ou égal à 2, alors $c \in \mathcal{M}_{\texttt{itermax}}$ et l'on considère que $c \in \mathcal{M}$ (ce qui n'est pas forcément vrai).

Exercice XV.6 p. 10

Écrire une fonction `int divergence(double xc, double yc, int itermax)`. Cette fonction prend en entrée un complexe c (parties réelle et imaginaire) et un entier `itermax` et doit renvoyer :

- le plus petit $n \leqslant$ `itermax` tel que $c \notin \mathcal{M}_n$, s'il en existe un ;
- `itermax + 1` sinon.

On définit deux constantes `ROWS` et `COLS` ainsi qu'un tableau bidimensionnel global :

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

#define ROWS 800
#define COLS 800

int arr[ROWS][COLS];
```

- `arr[0][0]` correspondra au pixel en haut à gauche de l'image ;
- `arr[ROWS - 1][0]` au pixel en bas à gauche ;
- `arr[0][COLS - 1]` au pixel en haut à droite ;
- `arr[ROWS - 1][COLS - 1]` au pixel en bas à droite.

Exercice XV.7 p. 10

On souhaite que l'image corresponde dans le plan complexe aux points $z = x + iy$ tels que $x_{min} \leqslant x \leqslant x_{max}$ et $y_{min} \leqslant y \leqslant y_{max}$. Écrire deux fonctions

```c
double re(int j, double xmin, double xmax);

double im(int i, double ymin, double ymax);
```

prenant en entrée un numéro de ligne ou de colonne dans le tableau `arr` et renvoyant la partie réelle ou imaginaire du complexe correspondant à ce pixel.
On souhaite bien sûr que $x_{min} + iy_{min}$ soit en bas à gauche de l'image et $x_{max} + iy_{max}$ en haut à droite.

Exercice XV.8 p. 10

Écrire une fonction

```c
void fill_tab(double xmin, double xmax,
              double ymin, double ymax,
              int itermax);
```

qui remplit le tableau global `arr` avec les valeurs renvoyées par la fonction `divergence` pour les différents pixels.

Exercice XV.9

p. 11

On suppose dans cette question que la fonction `fill_tab` a déjà été appelée, et donc que le tableau `arr` contient les valeurs de `divergence` pour les différents pixels.

1. Écrire une fonction `void print_pixel_bw(int i, int j, int itermax)` qui affiche `255 255 255 \n` ou `0 0 0 \n` selon que le pixel (i,j) de l'image appartient ou non à $\mathcal{M}_{\texttt{itermax}}$.
2. Écrire une fonction `void print_tab(int itermax)` qui affiche le fichier PPM correspondant à l'image calculée. On pourra reprendre (et modifier !) le code écrit au TP 13.
3. Écrire la fonction `main` de manière à obtenir le comportement suivant :
   - si le programme est appelé sans argument en ligne de commande, il utilise les valeurs $x_{min} = y_{min} = -2$, $x_{max} = y_{max} = 2$ et $itermax = 20$;
   - s'il est appelé avec un argument, cet argument est utilisé pour $itermax$;
   - s'il est appelé avec cinq arguments, le premier est utilisé pour $itermax$ et les autres pour $x_{min}, x_{max}, y_{min}$ et y_{max} (dans l'ordre) ;
   - s'il est appelé avec un autre nombre d'arguments, il termine sans rien faire en affichant un message d'erreur.

   Dans tous les cas (sauf le dernier), le programme doit afficher le contenu du fichier PPM décrit dans les questions précédentes sur la sortie standard.
   On utilisera la fonction `atoi` pour convertir une chaîne de caractères en entier et `atof` pour convertir en flottant.

Exercice XV.10

p. 11

Modifier le code pour obtenir un affichage en niveaux de gris. On affichera une couleur d'autant plus sombre que le point c est rapidement sorti du disque de rayon 2 (centré en l'origine).

Remarque

Pour obtenir un dessin vraiment « joli » [1], il faut définir une notion d'itération partielle pour éviter les aplats de couleur, puis utiliser une palette cyclique et interpoler.

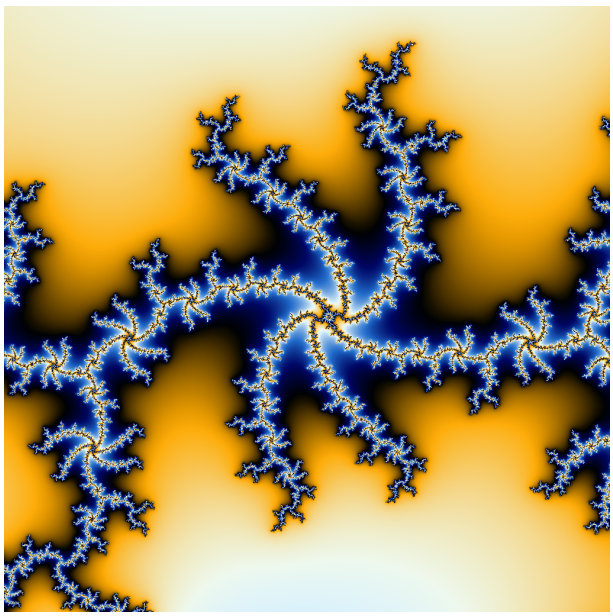

Figure XV.2 – Zoom sur la partie $0.00172 \leqslant x \leqslant 0.00184$, $-0.82258 \leqslant y \leqslant -0.82246$.

1. C'est un terme technique, dont la définition nous emmènerait trop loin du programme...

4 Pour chercher

Exercice XV.11 – Maxima par plage

p. 12

On considère un tableau $t = t_0, \dots, t_{n-1}$ et un entier $h \geqslant 1$. Pour $0 \leqslant i \leqslant n - h$, on définit :

$$m_h(i) = \max(t_i, \dots, t_{i+h-1})$$

1. Écrire un algorithme (simple) permettant de calculer tous les $m_h(i)$.
2. Déterminer la complexité de cet algorithme.
3. Trouver un algorithme permettant de trouver tous les $m_h(i)$ en temps $O(n)$. *C'est loin d'être évident ! N'hésitez pas à demander des indications.*
4. Écrire un programme en C implémentant cet algorithme.

Solutions

Correction de l'exercice XV.2 page I

1. Ce programme attend 3 arguments : un entier et deux `double`.
2. Le seul bloc d'activation est ici celui de `main` : la pile est de hauteur 1.
3. Pendant l'appel à `atof`, il y aura deux blocs : un pour `main` et un pour `atof` (il pourrait y en avoir davantage si `atof` faisait un appel, mais on suppose que ce n'est pas le cas). Donc la hauteur maximale est 2 à cette ligne.
4. `main` appelle `f` qui appelle `expo` (deux fois successivement). Pendant chacun de ces deux appels à `expo`, la pile est de hauteur 3. L'appel à `f` se termine avant que celui à `printf` ne commence, donc 3 est en fait la hauteur maximale de la pile.
5. Dans cette ligne, le seul bloc d'activation est à nouveau celui de `main`, donc la hauteur vaut 1.

Correction de l'exercice XV.3 page 2

1. On obtiendra :

```
Début de l'appel de f(0, _)
Début de l'appel de f(1, _)
Début de l'appel de f(2, _)
Fin de l'appel de f(2, _)
Fin de l'appel de f(1, _)
Fin de l'appel de f(0, _)
```

2. - Les valeurs de `nmax` et `*nmax` ne changent pas au cours des appels récursifs.
   - La valeur de `n` sera 0, puis 1, puis 2.
   - Les valeurs de `&n` et `&nmax` sont des adresses sur la pile : elle seront différentes à chaque appel, et on ne peut pas vraiment connaître leur valeur *a priori* (on peut prévoir comment elles évoluent si on connaît l'architecture).
3. Ce qu'on peut prévoir :

```
Début de l'appel de f(0, _)
n     = 0
&n    = pointeur_1
nmax  = pointeur_2
*nmax = 2
&nmax = pointeur_3
Début de l'appel de f(1, _)
n     = 1
&n    = pointeur_4
nmax  = pointeur_2
*nmax = 2
&nmax = pointeur_5
Début de l'appel de f(2, _)
n     = 2
&n    = pointeur_6
nmax  = pointeur_2
*nmax = 2
&nmax = pointeur_7
Fin de l'appel de f(2, _)
```

```
Fin de l'appel de f(1, _)
Fin de l'appel de f(0, _)
```

4.

5. On obtient l'affichage suivant (sur ma machine, en compilant avec une certaine version d'un certain compilateur, pour une certaine exécution, avec certaines options de compilation. . .) :

```
Début de l'appel de f(0, _)
n     = 0
&n    = 0x7ffe594b0ffc
nmax  = 0x7ffe594b1014
*nmax = 2
&nmax = 0x7ffe594b0ff0
Début de l'appel de f(1, _)
n     = 1
&n    = 0x7ffe594b0fdc
nmax  = 0x7ffe594b1014
*nmax = 2
&nmax = 0x7ffe594b0fd0
Début de l'appel de f(2, _)
n     = 2
&n    = 0x7ffe594b0fbc
nmax  = 0x7ffe594b1014
*nmax = 2
&nmax = 0x7ffe594b0fb0
Fin de l'appel de f(2, _)
Fin de l'appel de f(1, _)
Fin de l'appel de f(0, _)
```

On voit que les différents pointeurs obtenus pour `&n` sont à `0x20` les uns des autres, et qu'il en est de même pour les pointeurs `&nmax`. Cela indique que le bloc d'activation fait `0x20` = 32 octets.

Remarque

Ce type de raisonnement est un peu dangereux, parce que rien n'oblige le compilateur à faire quelque chose de simple (si on compile avec toutes les optimisations, on obtient quelque chose de bien plus difficile à interpréter). De toute façon je ne vous demanderai jamais de répondre à une question de ce type en évaluation.

Correction de l'exercice XV.4 page 2

1. Pas de problème :

```
void incremente(int* p){
    *p = *p + 1;
}
```

2. **a.** `incremente` attend un `int*`, on lui donne un `int` : il y a une erreur de type.

   b. Ici, la fonction `plus_petit` renvoie l'adresse de l'un de ses arguments. Or ces arguments sont des variables locales à la fonction : leur durée de vie est celle de l'appel. On passe donc à `incremente` un pointeur vers un objet qui n'existe plus : le comportement du programme est non défini.

   c. Cette version est correcte.

   d. Cette version n'est pas correcte : on renvoie encore un pointeur vers une variable locale, ce qui est illégal.

Correction de l'exercice XV.6 page 5

```c
int divergence(double xc, double yc, int itermax){
    double x = 0.;
    double y = 0.;
    int i = 0;
    while (x*x + y*y <= 4. && i <= itermax){
        double tmp = x;
        x = x*x - y*y + xc;
        y = 2. * tmp * y + yc;
        i++;
    }
    return i;
}
```

Correction de l'exercice XV.7 page 5

```c
double re(int j, double xmin, double xmax){
    double ratio = (double)j / (COLS - 1);
    return xmin + ratio * (xmax - xmin);
}

double im(int i, double ymin, double ymax){
    double ratio = (double)i / (ROWS - 1);
    return ymax + ratio * (ymin - ymax);
}
```

Correction de l'exercice XV.8 page 5

```c
void fill_tab(double xmin, double xmax, double ymin, double ymax, int itermax){
    for (int i = 0; i < ROWS; i++){
        for (int j = 0; j < COLS; j++){
            double x = re(j, xmin, xmax);
            double y = im(i, ymin, ymax);
            arr[i][j] = divergence(x, y, itermax);
        }
    }
}
```

Correction de l'exercice XV.9 page 6

```c
void print_pixel_bw(int i, int j, int itermax){
    int c = 0;
    if (arr[i][j] > itermax){
        c = 255;
    }
    for (int k = 0; k < 3; k++){
        printf("%d ", c);
    }
    printf("\n");
}

void print_tab(int itermax){
    printf("P3\n");
    printf("%d %d\n", COLS, ROWS);
    printf("255\n");
    for (int i = 0; i < ROWS; i++){
        for (int j = 0; j < COLS; j++){
            print_pixel_gs(i, j, itermax);
        }
    }
}

int main(int argc, char* argv[]){
    double xmin = -2.;
    double xmax = 2.;
    double ymin = -2.;
    double ymax = 2.;
    int itermax = 20;
    if (argc >= 2) {
        itermax = atoi(argv[1]);
    }
    if (argc == 6){
        xmin = atof(argv[2]);
        xmax = atof(argv[3]);
        ymin = atof(argv[4]);
        ymax = atof(argv[5]);
    }
    fill_tab(xmin, xmax, ymin, ymax, itermax);
    print_tab(itermax);
    return 0;
}
```

Correction de l'exercice XV.10 page 6

Pour quelque chose de basique, il n'y a pas grand chose à changer. Il suffit de définir :

```c
void print_pixel_gs(int i, int j, int itermax){
    int c = 255 * arr[i][j] / itermax;
    for (int k = 0; k < 3; k++){
        printf("%d ", c);
    }
    printf("\n");
}
```

Ensuite on change une ligne dans `print_tab`. Cela va marcher raisonnablement bien pour la fenêtre d'affichage par défaut, mais si l'on zoome on risque d'avoir un contraste très mauvais : en effet, rien ne dit que toutes les valeurs de 0 à $itermax$ seront prises. Pour éviter ce problème, on peut déterminer les valeurs minimale et maximale de `arr` et les faire correspondre à du noir et à du blanc, respectivement.

Correction de l'exercice XV.II page 7

On donne une solution efficace, qui suppose définies quelques variables globales :

- un entier `N` (la taille du tableau) ;
- un entier `H` (le h de l'énoncé) ;
- quatre tableaux d'entiers de même taille N : `arr` le tableau d'entrée, `gauche` et `droite` des tableaux auxiliaires et `result` le tableau de sortie.

Pour chaque entier i vérifiant $kh \leqslant i < (k+1)h$, on précalcule :

- $d_i = \max(arr[i], arr[i+1], \dots, arr[(k+1)h-1])$ que l'on met dans le tableau `droite`
- $g_i = \max(arr[kh], arr[kh+1], \dots, arr[i])$ que l'on met dans le tableau `gauche`.

Ces précalculs peuvent se faire en temps linéaire. On a ensuite $m_i = \max(d_i, g_{i+h-1})$. Le code calcule correctement les $m_h(i)$ pour les i situés entre $n-h$ et n (en prenant le maximum jusqu'à la fin du tableau). L'énoncé ne le demandait pas, ce qui permet de simplifier un peu.

```
void fill_droite(void){
    for (int i = 0; i < N; i = i + H){
        int j = min(i + H - 1, N - 1);
        droite[j] = arr[j];
        j--;
        for ( ; j >= i; j--){
            droite[j] = max(arr[j], droite[j + 1]);
        }
    }
}

void fill_gauche(void){
    for (int i = 0; i < N; i = i + H){
        gauche[i] = arr[i];
        for (int j = i + 1; j < min(i + H, N); j++){
            gauche[j] = max(gauche[j - 1], arr[j]);
        }
    }
}


void f(void){
    fill_gauche();
    fill_droite();
    int i;
    for (i = 0; i + H - 1 < N; i++){
        result[i] = max(droite[i], gauche[i + H - 1]);
    }
    for ( ; i < N; i++){
        result[i] = droite[i];
    }
}
```