

# Partie 1

## Rappel

25 septembre 2018

# Plan

1. Un tour rapide du C via un exemple
2. Rappel sur les pointeurs
3. Construction et correction d'algorithmes itératifs

# Plan

1. Un tour rapide du C via un exemple
2. Rappel sur les pointeurs
3. Construction et correction d'algorithmes itératifs

## Illustration : l'ensemble de Mandelbrot

En mathématique, l'ensemble de Mandelbrot  $\mathcal{M}$  est défini comme l'ensemble des points  $c \in \mathbb{C}$  du plan complexe pour lesquels la suite de nombres définie par la récurrence suivante :

$$\begin{cases} z_0 = 0, \\ z_{n+1} = z_n^2 + c \quad (n > 0). \end{cases}$$

est bornée.

Étudié depuis le début du vingtième siècle en dynamique complexe mais représenté pour la première fois à la fin des années septante grâce à l'ordinateur.

On aimerait implémenter un programme en C permettant de visualiser cet ensemble dans le plan complexe.



## Vérifier l'appartenance à $\mathcal{M}$

Impossible à vérifier formellement mais on peut néanmoins démontrer que si la suite des modules devient strictement supérieure à 2 pour un certain indice  $n$ , alors, la suite est croissante et tend vers l'infini à partir de cet indice.

On peut donc calculer un surensemble de  $\mathcal{M}$  en testant la contrainte  $|z_n| \leq 2$  pour des valeurs aussi grande que possible de  $n$ .

En pratique, on est obligé de se limiter à un nombre maximum  $N$  d'itérations, si on veut implémenter ce test empiriquement sur ordinateur.

Notre programme vérifiera donc en fait l'appartenance à l'ensemble  $\mathcal{M}' \subseteq \mathcal{M}$  défini comme suit :

$$\mathcal{M}' = \{c \in \mathbb{C} \mid \forall n, 0 \leq n \leq N : |z_n| \leq 2\}$$

## Vérifier l'appartenance à $\mathcal{M}'$

L'idée du programme est donc de calculer  $z_n$  pour des valeurs de  $n$  allant de 0 à  $N$ , en s'arrêtant dès que  $|z_n| > 2$ .

A priori, on ne peut pas manipuler des nombres complexes directement en  $\mathbb{C}$  mais la suite peut se réécrire comme suit sur base des parties réelles et imaginaires :

$$\begin{cases} \operatorname{Re}(z_0) = 0, \operatorname{Im}(z_0) = 0, \\ \operatorname{Re}(z_{n+1}) = \operatorname{Re}(z_n)^2 - \operatorname{Im}(z_n)^2 + \operatorname{Re}(c), \\ \operatorname{Im}(z_{n+1}) = 2\operatorname{Re}(z_n)\operatorname{Im}(z_n) + \operatorname{Im}(c), \end{cases}$$

où  $\operatorname{Re}(z)$  et  $\operatorname{Im}(z)$  désigne resp. les parties réelle et imaginaire d'un complexe  $z$ .

## Vérifier l'appartenance à $\mathcal{M}'$ (en C)

1/3

En supposant que les variables  $cr$  et  $ci$  (double) contiennent les parties imaginaires et réelles de  $c$ , le code suivant vérifie la non-appartenance de  $c$  à  $\mathcal{M}$  :

```
double zr = 0;
double zi = 0;
int n = 0;

while((n < N) && (zr*zr + zi*zi <= 4.0)) {
    double temp;
    temp = zr*zr - zi*zi + cr;
    zi = 2*zr*zi + ci;
    zr = temp;
    n++;
}
if (zr*zr + zi*zi <= 4.0)
    printf("c belongs to M'\n");
else
    printf("c does not belong to M'\n");
```

*Quel est l'invariant de cette boucle ?*

En utilisant un for à la place d'un while.

```
double zr = 0;
double zi = 0;

for(int n = 0; (n < N) && (zr*zr + zi*zi <= 4.0); n++) {
    double temp;
    temp = zr*zr - zi*zi + cr;
    zi = 2*zr*zi + ci;
    zr = temp;
}
if (zr*zr + zi*zi <= 4.0)
    printf("c belongs to M'\n");
else
    printf("c does not belong to M'\n");
```

## Rappel de C : déclaration et types primitifs

```
int a = 1, b, c;  
float e;
```

- Toute variable doit être déclarée en spécifiant son type et peut être initialisée au moment de sa déclaration
- Type primitif :

bool	true ou false (ISO-C99, avec stdbool.h)
char	caractère signé
int	entier signé
size_t	entier non-signé représentant une taille ou un indice
float	nombre réel (précision simple)
double	nombre réel (précision double)
- Le typage du C est **statique** (le type d'une variable est déterminé à la compilation) et **faible** (une valeur peut être convertie implicitement vers le type adéquat).

# Rappel de C : Opérateurs

(par ordre de précedence)

postfixe	[] . -> expr++ expr--
préfixe	++expr --expr +expr -expr ~ ! &expr *expr sizeof (type)expr
multiplicatifs	* / %
additifs	+ -
décalages	<< >>
comparaisons	< > <= >=
égalité	== !=
ET binaire	&
OU exclusif binaire	^
OU binaire	
ET logique	&&
OU logique	
conditionnel	?:
affectations	= += -= *= /= %= <<= >>= &= ^=  =

# Rappel de C : choix conditions

## Choix binaire

```
if (expr) {  
    ...  
}
```

```
if (expr) {  
    ...  
} else {  
    ...  
}
```

## Choix multiple

```
switch(expr) {  
    case const1 : instr1 break;  
    case const2 : instr2 break;  
    ...  
    default : instr  
}
```

## Expression conditionnelle

```
expr1 ? expr2 : expr3;
```

## Rappel de C : boucles

```
for (expr1; expr2; expr3) {  
    ...  
}  
  
while (expr) {  
    ...  
}  
  
do {  
    ...  
} while (expr);
```

Interruption :

- L'instruction `break` permet de quitter la boucle courante.
- L'instruction `continue` permet de passer à l'itération suivante, sans exécuter le restant de l'itération courante.

## Rappel de C : entrées-sorties

```
#include <stdio.h>
...
int a, b;

printf("Entrez une première valeur: ");
scanf("%d", &a);
printf("Entrez une seconde valeur: ");
scanf("%d", &b);

printf("%d + %d = %d\n", a, b, a + b);
...
```

- `printf` prend comme premier argument une chaîne formatée. Les arguments suivant sont les valeurs affectées aux spécificateurs de format (c.f. [http://en.wikipedia.org/wiki/Printf\\_format\\_string#Format\\_placeholders](http://en.wikipedia.org/wiki/Printf_format_string#Format_placeholders) pour une spécification complète).
- `scanf` permet d'entrer une valeur au clavier et de la stocker à l'adresse spécifiée. Attention : la gestion propre des erreurs est difficile !

On peut emballer ce code dans une fonction

```
int mandelbrotSet(double cr, double ci) {
    double zr = 0;
    double zi = 0;
    int n = 0;

    while(n < N && ((zr*zr + zi*zi) <= 4.0)) {
        double temp;
        temp = zr*zr - zi*zi + cr;
        zi = 2*zr*zi + ci;
        zr = temp;
        n++;
    }
    return (zr*zr + zi*zi <= 4.0);
}
```

## Rappel de C : fonctions et procédures

```
int fct1(int a, int b) {
    ...
    return 4;
}
int fct2(void) {
    int a,b;
    ...
    return a+b;
}
void fct3(float b) {
    ...
    [return;]
}
```

- Une fonction peut prendre zéro, un ou plusieurs arguments.
- Les arguments sont passés par valeur.
- Chaque fonction renvoie une valeur d'un type donné, ou void.
- Si une fonction renvoie une valeur, elle doit posséder un instruction return correspondant au bon type.

# Un programme C complet pour visualiser l'ensemble

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

#define N 1000 // Maximum number of iterations
#define W 1000 // Width/height of plot (pixels)

static int mandelbrotSet(double xc, double yc);

int main(int argc, char *argv[]) {
    double x = atof(argv[2]);
    double y = atof(argv[3]);
    double size = atof(argv[4]);

    FILE *fp = fopen(argv[1], "w");

    for (int i = 0; i < W; i++) {
        for (int j = 0; j < W; j++) {
            double cr = x - size/2 + size*j/W;
            double ci = y + size/2 - size*i/W;
            fprintf(fp, "%d ", mandelbrotSet(cr, ci));
        }
        fprintf(fp, "\n");
    }

    fclose(fp);

    return 0;
}
```

```
static int mandelbrotSet(double cr, double ci) {
    double zr = 0;
    double zi = 0;
    int n = 0;

    while(n < N && ((zr*zr + zi*zi) <= 4.0)) {
        double temp;
        temp = zr*zr - zi*zi + cr;
        zi = 2*zr*zi + ci;
        zr = temp;
        n++;
    }
    return n;
}
```

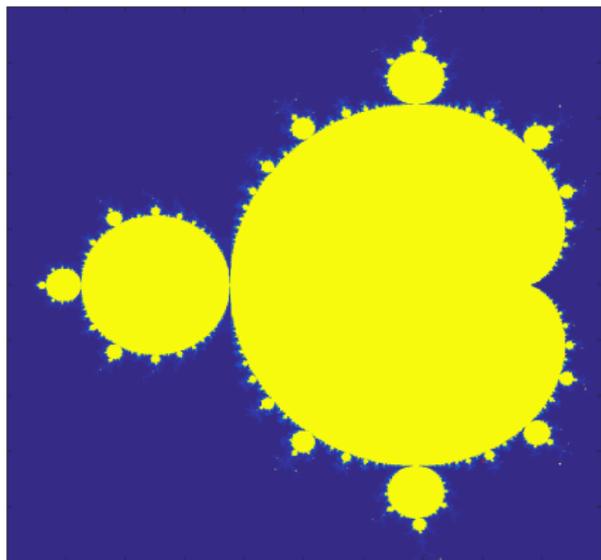
Utilisation :

```
> gcc mandelbrot.c -o mandelbrot
> ./mandelbrot m.amat -0.5 0.0 2.0
```

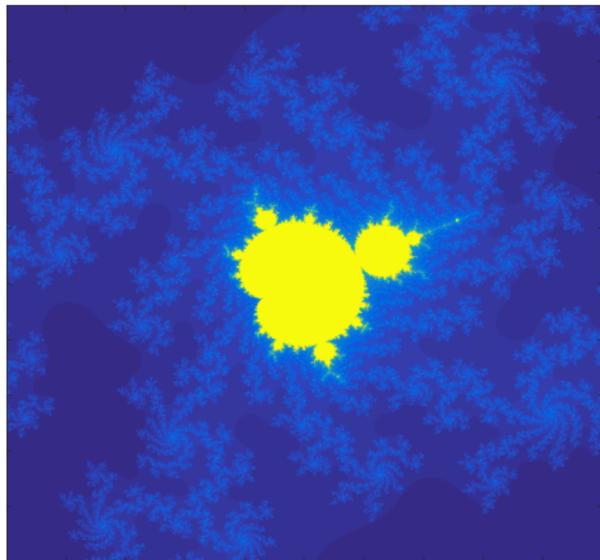
Pour voir l'image avec matlab

```
> M = load('m.amat');
> imagesc(M);
```

# L'ensemble de Mandelbrot



```
./mandelbrot m.amat -0.5 0.0 2.0
```



```
./mandelbrot m.amat 0.1015 -0.633 0.001
```

Pour en voir plus : <http://tilde.club/~david/m>

## Rappel de C : fonction main

```
int main() {  
    ...  
    return 0;  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    return 0;  
}
```

- Tout programme doit contenir une fonction main.
- La valeur de retour (entière) est rendue au système d'exploitation. Par convention, 0 signifie que tout s'est bien passé.
- On peut récupérer les paramètres passés au programme au moment de l'exécution en ajoutant deux arguments à la fonction :
  - ▶ `argc` : le nombre de paramètres passés (y compris le nom de l'exécutable)
  - ▶ `argv` : un tableau de chaînes de caractères dont les éléments contiennent ces arguments
- Ces arguments peuvent être analysés à l'aide des fonctions de la librairie `stdlib` (p.ex., `atoi`, `atof`).

## Rappel de C : tableaux

```
int[5] a;  
a[0] = 1;  
a[4] = 42;  
a[-1] = 10; // Bug!  
a[5] = -5; // Bug!  
  
char id[] = "texte"; // Chaîne de caractères  
  
int mat[3][4]; // Tableau multidimensionnel
```

- Un tableau est un type de données indexable contenant des éléments du même type.
- Les éléments sont indexés à partir de 0 et jusqu'à N-1.
- Les tableaux sont passés aux fonctions par **pointeurs** (voir plus loin). Leurs modifications sont donc répercutées à l'appelant.
- Une chaîne de caractère est un tableau de `char` terminé par un caractère null `'\0'`.

## Rappel de C : organisation d'un programme

Organisation possible d'un programme C en un seul fichier (voir `mandelbrot.c`) :

- Directive d'inclusion (`#include`)
- Définition de constantes et macro (`#define`)
- Définitions de types (`typedef`, voir plus loin)
- Déclarations de variables globales
- Prototypes des fonctions autres que la fonction `main`
- Définition de la fonction `main`
- Définition des autres fonctions

Seul contrainte forte : toute fonction/variable/constante doit être définie avant d'être utilisée.

## Autre variante utilisant un nouveau type

1/2

On peut simplifier le code principal et améliorer sa lisibilité en définissant un nouveau type de données pour les nombres complexes.

complex.c

complex.h

```
typedef struct {
    double re, im;
} complex;

complex complex_new(double, double);
complex complex_sum(complex, complex);
complex complex_product(complex, complex);
double complex_modulus(complex);
...
```

```
#include <math.h>
#include "complex.h"

complex complex_new(double re, double im) {
    complex c;
    c.re = re;
    c.im = im;
    return c;
}

complex complex_sum(complex a, complex b) {
    complex c;
    c.re = a.re + b.re;
    c.im = a.im + b.im;
    return c;
}

complex complex_product(complex a, complex b) {
    complex c;
    c.re = a.re * b.re - a.im * b.im;
    c.im = a.re * b.im + a.im * b.re;
    return c;
}

double complex_modulus(complex c) {
    return sqrt(c.re*c.re + c.im*c.im);
}
...
```

## Rappel de C : structures

```
// définition d'une structure
struct complex_t {
    double re, im;
};
// Définition d'un nouveau type
typedef struct {
    double re, im;
} complex;

// Utilisation
struct Complex_t a, c = {1.2, 3.4};
complex b = {.im = 1.0, .re = 2.0};
b.re = 1.0;
a.im = 3.4;
```

- Une structure est un type de données composé, dont les éléments peuvent être de types différents.
- Les éléments de la structure sont accessibles par leurs noms via l'opérateur '.'.

```
#include <stdio.h>
#include <stdlib.h>
#include "complex.h"

#define N 1000 // Maximum number of iterations
#define W 1000 // Width/Height of the plot

static int mandelbrotSet(complex c);

int main(int argc, char *argv[]) {
    double x = atof(argv[2]);
    double y = atof(argv[3]);
    double size = atof(argv[4]);

    FILE *fp = fopen(argv[1], "w");

    for (int i = 0; i < W; i++) {
        for (int j = 0; j < W; j++) {
            double cr = x - size/2 + size*j/W;
            double ci = y + size/2 - size*i/W;
            complex c = complex_new(cr, ci);
            fprintf(fp, "%d ", mandelbrotSet(c));
        }
        fprintf(fp, "\n");
    }

    fclose(fp);

    return 0;
}
```

```
static int mandelbrotSet(complex c) {
    complex z = complex_new(0,0);
    int n = 0;
    while ((n < N) && (complex_modulus(z) <= 2.0)) {
        z = complex_plus(complex_product(z,z),c);
        n++;
    }
    return n;
}
```

Pour compiler :

```
> gcc mandelbrot.c complex.c -o mandelbrot
```

## Autre variante utilisant la librairie complex du C99

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

#define N 1000 // Maximum number of iterations
#define W 1000 // Width/Height of the plot

static int mandelbrotSet(double _Complex c);

int main(int argc, char *argv[]) {
    double x = atof(argv[2]);
    double y = atof(argv[3]);
    double size = atof(argv[4]);

    FILE *fp = fopen(argv[1], "w");

    for (int i = 0; i < W; i++) {
        for (int j = 0; j < W; j++) {
            double cr = x - size/2 + size*j/W;
            double ci = y + size/2 - size*i/W;
            fprintf(fp, "%d ", mandelbrotSet(cr+I*ci));
        }
        fprintf(fp, "\n");
    }

    fclose(fp);

    return 0;
}
```

```
static int mandelbrotSet(double _Complex c) {
    double _Complex z = 0;
    int n = 0;
    while ((n < N) && (cabs(z) <= 2.0)) {
        z = z*z+c;
        n++;
    }
    return n;
}
```

Pour compiler :

```
> gcc mandelbrot.c -o mandelbrot
```

# Le langage C

Le C est un langage de bas niveau (très proche du matériel) qui est très permissif (peu de choses sont interdites).

Avantages : efficacité, puissance, flexibilité.

Inconvénients : code souvent sujet aux erreurs (bugs) et parfois difficile à comprendre et à maintenir.

Conseil pour ce cours : éviter les “trucs” de programmation et privilégier la lisibilité à la compacité, voire l'efficacité<sup>1</sup>, du code.

---

1. à complexité constante, cf. Partie 3

# Un mauvais exemple

Que fait ce code ?

```
#include <stdio.h>
main() {
    float C,l,c,o,I=-20;char _;for(;I++<20;puts(""))
    for(O=-46;O<14;putchar(_?42:32),O++)for(C=l=_=0;o
    =l*l,c=C*C,l=2*C*l+I/20,C=c-o+O/20,o+c<4&&++_);
}
```

Source : <https://www.codeproject.com/Articles/2228/Obfuscating-your-Mandelbrot-code>



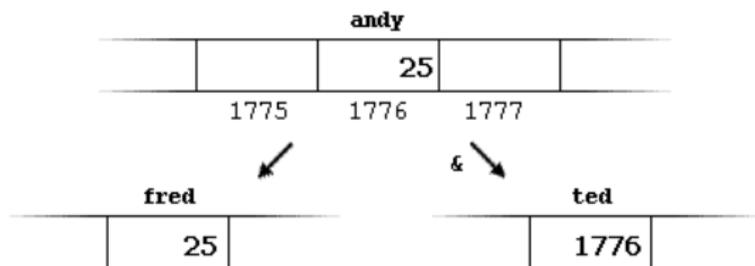
# Plan

1. Un tour rapide du C via un exemple
2. Rappel sur les pointeurs
3. Construction et correction d'algorithmes itératifs

# Variables et adresses

- L'identifiant d'une variable correspond à un emplacement mémoire, situé à une certaine adresse, contenant une valeur d'un certain type.
- Un pointeur est une variable dont la valeur est une adresse.
- Le type d'un pointeur est le type de la valeur pointée suivi de \* (e.g., `int*` pour un pointeur vers un entier).

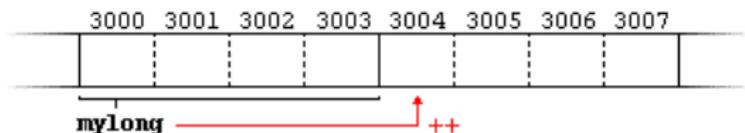
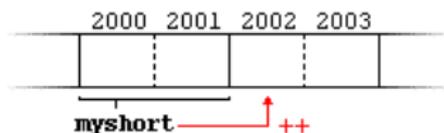
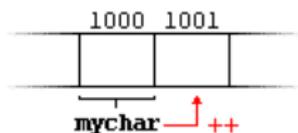
```
int andy = 25;  
int fred = andy;  
int *ted = &andy; // & dénote l'adresse de la variable andy
```



# Arithmétique sur les pointeurs

- L'addition et la soustraction sont autorisées sur des pointeurs.
- $p + 1$  correspond à l'emplacement mémoire suivant  $p$ , du même type.
- $p + n$  correspond au  $n$ -ème emplacement mémoire après  $p$ , du même type.

```
char* mychar;  
short* myshort;  
long* mylong;  
mychar = mychar + 1;  
myshort++;  
mylong++;
```



# Tableaux et pointeurs

L'identifiant d'un tableau est équivalent à un pointeur pointant vers le premier élément de ce tableau.

```
int a[5];  
int* p;  
  
p = a;  
a[0] = 10;  
*p = 10;           // Ces deux expressions sont équivalentes  
a[2] = 42;  
*(p + 2) = 42;    // Ces deux expressions sont aussi équivalentes
```

# Où est le bug ?

```
#include <stdio.h>
int main() {
    int s[4], t[4];
    for (int i = 0; i <= 4; i++) {
        s[i] = t[i] = i;
    }
    printf("i:s:t\n");
    for (int i = 0; i <= 4; i++) {
        printf("%d:%d:%d\n", i, s[i], t[i]);
    }
    return 0;
}
```

Sortie :

```
i:s:t
0:4:0
1:1:1
2:2:2
3:3:3
4:4:4
```

## Allocation / désallocation de mémoire

- Un bloc mémoire peut être alloué dynamiquement avec la fonction `malloc`.
- Renvoie `NULL` (=0) en cas d'échec, qui représente un pointeur vers rien.
- Tout bloc alloué dynamiquement **doit** être libéré explicitement avec la fonction `free`.

```
int* p = (int*) malloc(sizeof(int)); // Alloue un bloc de la taille d'un int
if (!p) {                             // Toujours vérifier le succès de malloc
    printf("Error");
    return 1;
}
free(p);                                // On libère le bloc

int* q = (int*) malloc(10 * sizeof(int)); // Alloue un bloc pour 10 int
q[0] = 42;
free(q);
```

# Structures et pointeurs

```
typedef struct {
    double re,im;
} complex;
// cette fonction n'a aucun effet sur la donnée
void test1(complex a) {
    a.re = a.re + 1.0;
    a.im = a.im + 1.0;
}
// cette fonction modifie la donnée reçue en argument
void test1(complex *a) {
    a->re = a->re + 1.0;
    a->im = a->im + 1.0;
}
```

- Les structures sont passées (et renvoyées) par **valeurs** aux fonctions. En conséquence :
  - ▶ Les modifications des champs ne sont pas répercutées vers le code appelant.
  - ▶ Les structures sont copiées intégralement lors des appels de fonctions
- On manipule souvent les structures par l'intermédiaire des pointeurs.
- Si  $p$  est un pointeur vers une donnée possédant un champ  $c$ , alors la notation  $p->c$  est une abréviation pour  $(*p).c$ .

# Une ré-implémentation des complexes par pointeur

## complex.h

```
typedef struct {
    double re, im;
} complex;

complex *complex_new(double, double);
double complex_real_part(complex *);
double complex_imgry_part(complex *);
void complex_sum(complex *, complex *);
void complex_difference(complex *, complex *);
void complex_product(complex *, complex *);
double complex_modulus(complex *);
double complex_distance(complex *, complex *);
```

## complex.c

```
#include <math.h>
#include "complex.h"

complex *complex_new(double re, double im) {
    complex *c = (complex *)malloc(sizeof(complex));
    c->re = re;
    c->im = im;
    return c;
}

void complex_sum(complex *a, complex *b) {
    a->re = a->re + b->re;
    a->im = a->im + b->im;
}

void complex_product(complex *a, complex *b) {
    double tmp_re, tmp_im;
    tmp_re = a->re * b->re - a->im * b->im;
    tmp_im = a->re * b->im + a->im * b->re;
    a->re = tmp_re;
    a->im = tmp_im;
}

...
```

# Plan

1. Un tour rapide du C via un exemple
2. Rappel sur les pointeurs
3. Construction et correction d'algorithmes itératifs
  - Construction de programmes
  - Construction d'algorithmes itératifs
  - Correction d'algorithmes itératifs

# Construction de programmes

Pour résoudre un problème de programmation complexe, on le découpe généralement en **sous-problèmes** plus simples à appréhender :

- Exemples de sous-problèmes pour afficher l'ensemble de Mandelbrot :
  - ▶ vérifier l'appartenance d'un point à l'ensemble,
  - ▶ calculer le module d'un nombre complexe,
  - ▶ produire l'image en parcourant le plan complexe.

Avantages d'un découpage :

- Facilite l'implémentation : on peut résoudre chaque sous-problème indépendamment,
- Généralité : on peut partager du code entre différents programmes,
- Lisibilité et facilité de maintenance du code.

# Construction de programmes

Résoudre un sous-problème élémentaire :

- Certains sont triviaux et requièrent d'établir une séquence simple d'instructions (p.ex., calculer le module d'un nombre complexe)
- D'autres sont plus complexes et nécessitent de **répéter** une séquence d'instructions selon un schéma dépendant des données (p.ex., déterminer l'appartenance à l'ensemble de Mandelbrot)

Deux types de solutions pour ces derniers cas :

- Solutions **itératives**, basées sur des boucles (cf. INFO2009 et ce rappel)
- Solutions **récurives**, basées sur des fonctions qui s'invoquent elles-mêmes (cf. Partie 2)

# Construction d'algorithmes itératifs

Concevoir un algorithme itératif<sup>2</sup> peut être un exercice très compliqué, surtout si on cherche une solution efficace.

Deux difficultés principales :

- Imaginer le **schéma itératif** permettant de résoudre le problème.
- Générer le **code** implémentant ce schéma en évitant les bugs.

Le premier problème est de loin le plus compliqué et cette compétence s'acquière quasi uniquement via la pratique (cf. INFO0902 pour des techniques génériques néanmoins).

La **technique de l'invariant** permet d'aborder formellement le second problème.

---

2. C'est le cas aussi des algorithmes récursifs

# Invariant de boucle

Un **invariant de boucle** est une propriété définie sur les variables du programme qui définit précisément ce qui doit être calculé à chaque itération pour arriver au résultat escompté. Il résume l'état courant des calculs.

Identifier l'invariant revient à imaginer le schéma itératif de résolution du problème et est parfois **non trivial**.

Une fois l'invariant établi, implémenter la boucle peut par contre se faire de manière relativement **automatique**.

L'utilisation de l'invariant permet donc d'éviter les erreurs d'implémentation.

## Invariant de boucle : plus formellement

Une **assertion** est une relation entre les variables et les données utilisées par le programme qui est vraie à un moment donné lors de l'exécution du programme.

Deux assertions particulières :

- **Pré-condition  $P$**  : condition que doivent remplir les entrées valides du programme
- **Post-condition  $Q$**  : condition qui exprime que le résultat du programme est celui attendu.

On cherche donc à écrire un programme, noté  $S$ , dont l'exécution dans **tous** les cas où  $P$  est vrai mène à ce que  $Q$  soit **toujours** vrai.

Lorsque c'est le cas, on dira que le **triplet**  $\{P\}S\{Q\}$  est **correct**.

**Exemple** : Si  $P = \{x \geq 0\}$  et  $Q = \{y^2 = x\}$ , le code  $S = "y = \text{sqrt}(x);"$  rend le triplet  $\{P\}S\{Q\}$  correct.

## Invariant de boucle : plus formellement

```
{P}  
INIT  
while (B)  
    CORPS  
FIN  
{Q}
```

```
{P}  
INIT  
{I}  
while (B)  
    {I et B} CORPS {I}  
    {I et non B}  
FIN  
{Q}
```

Dans le cas où le programme nécessite une boucle :

- On met en évidence une assertion particulière  $I$ , l'**invariant de boucle**, qui décrit l'état du programme pendant la boucle.
- On détermine ensuite le gardien  $B$  et les codes  $INIT$ ,  $CORPS$  et  $FIN$  tels que les triplets suivants soient corrects :
  - ▶  $\{P\}$  INIT  $\{I\}$
  - ▶  $\{I \text{ et } B\}$  CORPS  $\{I\}$
  - ▶  $\{I \text{ et non } B\}$  FIN  $\{Q\}$

Si on a plusieurs boucles imbriquées, on les traite séparément.

# Illustration 1 : appartenance à l'ensemble de Mandelbrot

Pré et post-conditions :

- $P = \{cr \in \mathbb{R}, ci \in \mathbb{R}\}$
- $Q = \{r = 1 \text{ si } \forall n, 0 \leq n \leq N : |z_n| \leq 2, 0 \text{ sinon}\}$

où  $cr$  et  $ci$  sont les entrées du programmes,  $r$  le résultat,  $N$  une constante, et  $z_n$  est la  $n$ -ème valeur de la suite définie précédemment avec  $c \in \mathbb{C}$  tel que  $c = cr + ici$ .

Schéma de la boucle :

- on va calculer  $z_n$  pour des valeurs croissantes de  $n$  et s'arrêter dès que soit  $|z_n| > 2$ , soit  $n = N$ .

# Illustration 1 : appartenance à l'ensemble de Mandelbrot

Invariant :

$$I = \{(\forall n' : 0 \leq n' < n : |z_{n'}| \leq 2) \text{ et } (z_r + iz_i = z_n) \text{ et } (0 \leq n \leq N)\},$$

où  $n$  est le compteur de boucle et  $z_r$  et  $z_i$  sont deux variables qui contiendront les résultats intermédiaires.

$$\overbrace{z_0, z_1, \dots, z_{n-1}}^{|\dots| \leq 2}, \underbrace{z_n}_{z_r + iz_i = z_n}, \overbrace{z_{n+1}, \dots, z_N}^{|\dots| ?}$$

Gardien :

$$B = \{n < N, z_r^2 + z_i^2 \leq 4\}.$$

# Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \dots, z_{n-1}}^{|\dots| \leq 2}, \underbrace{z_n}_{z_r + iz_i = z_n}, \overbrace{z_{n+1}, \dots, z_N}^{|\dots| ?}$$

```
{P}
INIT
{I}
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
    {I et B}
    CORPS
    {I}
}
{I et non B}
FIN
{Q}                                     {r = 1 si  $\exists n : 0 \leq n \leq N : |z_n| > 2, 0$  sinon}
```

{cr ∈ ℝ, ci ∈ ℝ}

# Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \dots, z_{n-1}}^{|\dots| \leq 2}, \underbrace{z_n, z_{n+1}, \dots, z_N}_{z_r + iz_i = z_n}^{|\dots| ?}$$

{P} {cr ∈ ℝ, ci ∈ ℝ}

```
double zr = 0;  
double zi = 0;  
int n = 0;
```

{I}

```
while((n < N) && (zr*zr + zi*zi <= 4.0)) {  
    {I et B}  
    CORPS  
    {I}
```

```
}
```

{I et non B}

FIN

{Q}

{r = 1 si ∃n : 0 ≤ n ≤ N : |zn| > 2, 0 sinon}

# Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \dots, z_{n-1}}^{|\dots| \leq 2}, \underbrace{z_n}_{z_r + iz_i = z_n}, \overbrace{z_{n+1}, \dots, z_N}^{|\dots| ?}$$

{P} {cr ∈ ℝ, ci ∈ ℝ}

```
double zr = 0;  
double zi = 0;  
int n = 0;
```

{I}

```
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
```

{I et B}

```
double temp;  
temp = zr*zr - zi*zi + cr;  
zi = 2*zr*zi + ci;  
zr = temp;  
n++;
```

{I}

}

{I et non B}

**FIN**

{Q}

{r = 1 si ∃n : 0 ≤ n ≤ N : |zn| > 2, 0 sinon}

# Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \dots, z_{n-1}}^{|\dots| \leq 2}, \underbrace{z_n}_{z_r + iz_i = z_n}, \overbrace{z_{n+1}, \dots, z_N}^{|\dots| ?}$$

```
{P} {cr ∈ ℝ, ci ∈ ℝ}  
double zr = 0;  
double zi = 0;  
int n = 0;  
{I}  
while((n < N) && (zr*zr + zi*zi <= 4.0)) {  
    {I et B}  
    double temp;  
    temp = zr*zr - zi*zi + cr;  
    zi = 2*zr*zi + ci;  
    zr = temp;  
    n++;  
    {I}  
}  
{I et non B}  
int r = (zr*zr+zi*zi <= 4.0);  
{Q} {r = 1 si ∃n : 0 ≤ n ≤ N : |zn| > 2, 0 sinon}
```

## Illustration 1 : appartenance à l'ensemble de Mandelbrot

```
double zr = 0;
double zi = 0;
int n = 0;
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
    double temp;
    temp = zr*zr - zi*zi + cr;
    zi = 2*zr*zi + ci;
    zr = temp;
    n++;
}
int r = (zr*zr+zi*zi <= 4.0);
```

## Illustration 2 : tri par insertion

On souhaite écrire une fonction pour trier un tableau  $A$  de valeurs entières. Le tri doit être effectué dans le tableau lui-même, via échanges d'éléments.

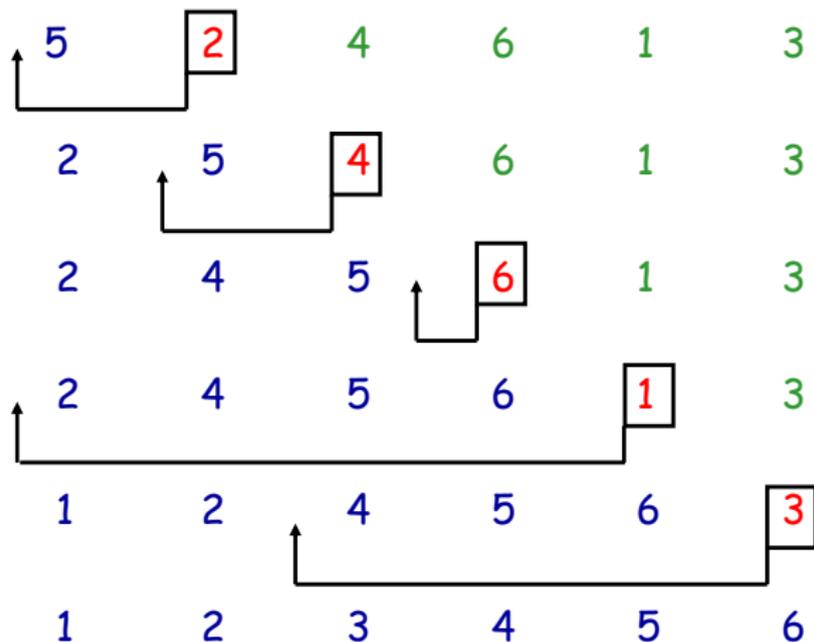
```
void insertion_sort(int A[], int N);
```

Principe de l'algorithme :

- On parcourt le tableau de gauche à droite en triant successivement les préfixes du tableau de tailles  $2, 3, \dots, N$ .
- A chaque itération, on augmente la taille du préfixe trié en insérant le nouvel élément  $A[i]$  à sa position dans le sous-tableau  $A[0..i-1]$  précédemment ordonné.



# Tri par insertion : graphiquement

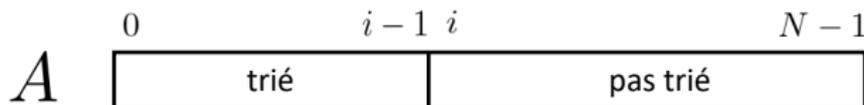


## Tri par insertion : boucle externe

Pré-condition  $P_e$  :  $A$  est un tableau d'entiers de taille  $N$ .

Post-condition  $Q_e$  :  $A$  contient les éléments du tableau de départ triés par ordre croissant.

Invariant  $I_e$  (de la boucle externe) : Le sous-tableau  $A[0..i-1]$ , avec  $1 \leq i \leq N$ , contient les  $i$  premiers éléments du tableau initial triés par ordre croissant.



Gardien  $B_e$  :  $i < N$ .

## Tri par insertion : boucle externe

$\{P_e\}$	$\{A \text{ est un tableau d'entiers de taille } N\}$
<b>INITe</b>	
$\{I_e\}$	$\{A[0..i-1] \text{ trié}\}$
<b>while</b> ( $i < N$ ) {	
$\{I_e \text{ et } B_e\}$	$\{A[0..i-1] \text{ trié et } i < N\}$
<b>CORPSe</b>	
$\{I_e\}$	$\{A[0..i-1] \text{ trié}\}$
}	
$\{I_e \text{ et non } B\}$	$\{A[0..i-1] \text{ trié et } i = N\}$
<b>FINe</b>	
$\{Q_e\}$	$\{A \text{ contient les éléments du tableau de départ triés}\}$

## Tri par insertion : boucle externe

$\{P_e\}$	$\{A \text{ est un tableau d'entiers de taille } N\}$
<code>int i = 1;</code>	
$\{I_e\}$	$\{A[0..i-1] \text{ trié}\}$
<code>while (i &lt; N) {</code>	
$\{I_e \text{ et } B_e\}$	$\{A[0..i-1] \text{ trié et } i < N\}$
<b>CORPSe</b>	
$\{I_e\}$	$\{A[0..i-1] \text{ trié}\}$
<code>}</code>	
$\{I_e \text{ et non } B\}$	$\{A[0..i-1] \text{ trié et } i = N\}$
<b>FINe</b>	
$\{Q_e\}$	$\{A \text{ contient les éléments du tableau de départ triés}\}$

## Tri par insertion : boucle externe

$\{P_e\}$	$\{A \text{ est un tableau d'entiers de taille } N\}$
<code>int i = 1;</code>	
$\{I_e\}$	$\{A[0..i-1] \text{ trié}\}$
<code>while (i &lt; N) {</code>	
$\{I_e \text{ et } B_e\}$	$\{A[0..i-1] \text{ trié et } i < N\}$
<b>CORPSe</b>	
$\{I_e\}$	$\{A[0..i-1] \text{ trié}\}$
<code>}</code>	
$\{I_e \text{ et non } B\}$	$\{A[0..i-1] \text{ trié et } i = N\}$
-	
$\{Q_e\}$	$\{A \text{ contient les éléments du tableau de départ triés}\}$

## Tri par insertion : boucle externe

$\{P_e\}$	$\{A \text{ est un tableau d'entiers de taille } N\}$
<code>int i = 1;</code>	
$\{I_e\}$	$\{A[0..i-1] \text{ trié}\}$
<code>while (i &lt; N) {</code>	
$\{I_e \text{ et } B_e\}$	$\{A[0..i-1] \text{ trié et } i < N\}$
<b>CORPSe'</b>	
$\{Q_i\}$	$\{A[0..i] \text{ trié}\}$
<code>i++;</code>	
$\{I_e\}$	$\{A[0..i-1] \text{ trié}\}$
<code>}</code>	
$\{I_e \text{ et non } B\}$	$\{A[0..i-1] \text{ trié et } i = N\}$
-	
$\{Q_e\}$	$\{A \text{ contient les éléments du tableau de départ triés}\}$

# Tri par insertion : boucle interne (CORPSe')

Pré-condition  $P_i$  :  $\{I_e \text{ et } B_e\} = \{A[0..i-1] \text{ trié et } i < N\}$

Post-condition  $Q_i$  :  $I_e = \{A[0..i] \text{ trié}\}$

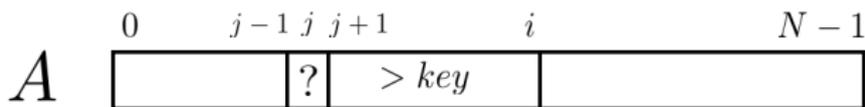
Idée de la boucle : Soit  $key = A[i]$ , la valeur à déplacer :

- On parcourt le sous-tableau  $A[0..i-1]$  de droite à gauche.
- Tant que les éléments parcourus sont supérieurs à  $key$ , on les décale d'une position vers la droite.
- On insère  $key$  à la position finalement atteinte.

# Tri par insertion : boucle interne

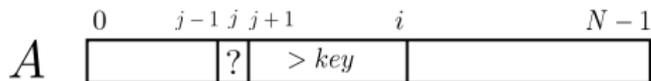
**Invariant  $I_j$**  : Soit  $j$  un nouvel indice et  $key = A[i]$  la valeur à insérer :

- $A[0..j-1]$  et  $A[j+1..i]$  sont triés par ordre croissant et ensemble contiennent tous les éléments du sous-tableau  $A[0..i]$  initial, excepté  $key$ .
- $A[j-1] \leq A[j+1]$
- $key < A[j+1]$



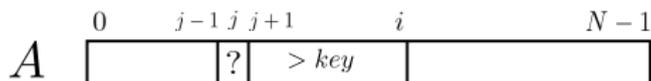
**Gardien  $B_j$**  :  $\{j > 0 \text{ et } A[j-1] > key\}$

# Tri par insertion : boucle interne



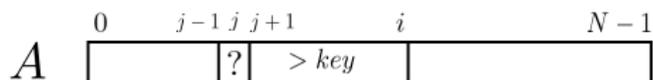
```
{ $P_i$ }
INIT $i$ 
{ $I_i$ }
while (j > 0 && A[j-1] > key) {
    { $I_i$  et  $B_i$ }
    CORPS $i$ 
    { $I_i$ }
}
{ $I_i$  et non  $B_i$ }
FIN $i$ 
{ $Q_i$ }
{ $A[0..i-1]$  trié}
```

## Tri par insertion : boucle interne



```
{ $P_i$ } { $A[0..i-1]$  trié et  $i < N$ }  
int key = A[i];  
int j = i;  
{ $I_i$ }  
while (j > 0 && A[j-1] > key) {  
    { $I_i$  et  $B_i$ }  
    CORPSi  
    { $I_i$ }  
}  
    { $I_i$  et non  $B_i$ }  
FINi  
{ $Q_i$ } { $A[0..i-1]$  trié}
```

## Tri par insertion : boucle interne



{ $P_i$ } { $A[0..i-1]$  trié et  $i < N$ }

```
int key = A[i];
```

```
int j = i;
```

{ $I_i$ }

```
while (j > 0 && A[j-1] > key) {
```

```
    { $I_i$  et  $B_i$ }
```

```
    A[j] = A[j-1];
```

```
    j--;
```

```
    { $I_i$ }
```

```
}
```

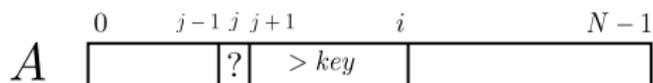
{ $I_i$  et non  $B_i$ }

**FINi**

{ $Q_i$ }

{ $A[0..i-1]$  trié}

## Tri par insertion : boucle interne



{ $P_i$ } { $A[0..i-1]$  trié et  $i < N$ }

```
int key = A[i];
```

```
int j = i;
```

{ $I_i$ }

```
while (j > 0 && A[j-1] > key) {
```

```
    { $I_i$  et  $B_i$ }
```

```
    A[j] = A[j-1];
```

```
    j--;
```

```
    { $I_i$ }
```

```
}
```

{ $I_i$  et non  $B_i$ }

```
A[j] = key;
```

```
i++;
```

{ $Q_i$ } { $A[0..i-1]$  trié}

## Tri par insertion : code complet

```
void insertion_sort(int A[], int N) {
    int i = 1;
    while (i < N) {
        int key = A[i];
        int j = i;
        while (j > 0 && A[j-1]>key) {
            A[j] = A[j-1];
            j--;
        }
        A[j] = key;
        i++;
    }
}
```

# Discussion

- L'expression d'un invariant, même informel, permet de limiter les erreurs lors de l'implémentation d'une boucle.
- Dans la suite du cours théorique, on fournira ponctuellement les invariants de boucle.
- Vous devez être capables de pouvoir exprimer l'invariant de toutes vos boucles.
- Si vous avez vous-même dérivé l'algorithme, trouver l'invariant devrait être trivial.

# Plan

1. Un tour rapide du C via un exemple
2. Rappel sur les pointeurs
3. Construction et correction d'algorithmes itératifs
  - Construction de programmes
  - Construction d'algorithmes itératifs
  - Correction d'algorithmes itératifs

## Preuve de correction d'algorithmes itératifs

- Dans certains contextes, il est crucial de prouver formellement qu'un programme est correct (p.ex. dans le domaine médical ou en aéronautique).
- On peut toujours tester son programme **empiriquement** mais en général, il est impossible de considérer tous les cas possibles d'utilisation d'un code.

*Testing can only show the presence of bugs, not their absence*

*E.W. Dijkstra*

- L'analyse de correction de triplets et la technique de l'invariant de boucle peut aussi être utilisé pour prouver formellement qu'un algorithme itératif est correct
- On peut automatiser une grosse partie de ces analyses, mais pas la dérivation des invariants de boucle.

## Terminaison de boucle

Prouver qu'un triplet  $\{P\}S\{Q\}$  est correct n'est pas suffisant dans le cas d'une boucle.

Il faut encore prouver que la boucle se termine.

Exemple : On peut prouver que le triplet ci-dessous est correct mais la boucle ne se termine pas toujours. On dira que le code est **partiellement correct**.

```
{cr ∈ ℝ, ci ∈ ℝ}
double zr = 0;
double zi = 0;

while(zr*zr + zi*zi <= 4.0) {
    double temp;
    temp = zr*zr - zi*zi + cr;
    zi = 2*zr*zi + ci;
    zr = temp;
}
int r = (zr*zr+zi*zi <= 4.0);
{r = 1 si ∀n ≥ 0 : |zn| ≤ 2, 0 sinon}
```

# Terminaison de boucle

Pour prouver qu'une boucle se termine, on cherche une **fonction de terminaison**  $f$  :

- définie sur base des variables de l'algorithme et à valeur **entière naturelle** ( $\geq 0$ )
- telle que  $f$  **décroît strictement** suite à l'exécution du corps de la boucle
- telle que  $B$  **implique**  $f > 0$

Puisque  $f$  décroît strictement, elle finira par atteindre 0 et donc à infirmer  $B$ .

Exemple :

- Mandelbrot :  $f = N - n$
- Tri par insertion (boucle externe) :  $f = N - i$

## Terminaison de boucle

Il n'est pas toujours trivial de prouver la terminaison d'une boucle.

Personne n'a pu encore prouver que la boucle suivante se terminait pour tout  $n > 1$ , bien qu'on l'ait prouvé empiriquement pour toutes les valeurs de  $N < 1,25 \cdot 2^{62}$ .

```
void Algo(int n) {
    while(n!=1) {
        if (n % 2) // n est impair
            n = 3*n+1;
        else      // n est pair
            n = n/2;
    }
}
```

[https://fr.wikipedia.org/wiki/Conjecture\\_de\\_Syracuse](https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse)