Partie 1

Rappel de C

23 septembre 2019

Plan

1. Un tour rapide du C via un exemple

2. Rappel sur les pointeurs

Plan

1. Un tour rapide du C via un exemple

2. Rappel sur les pointeurs

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 :

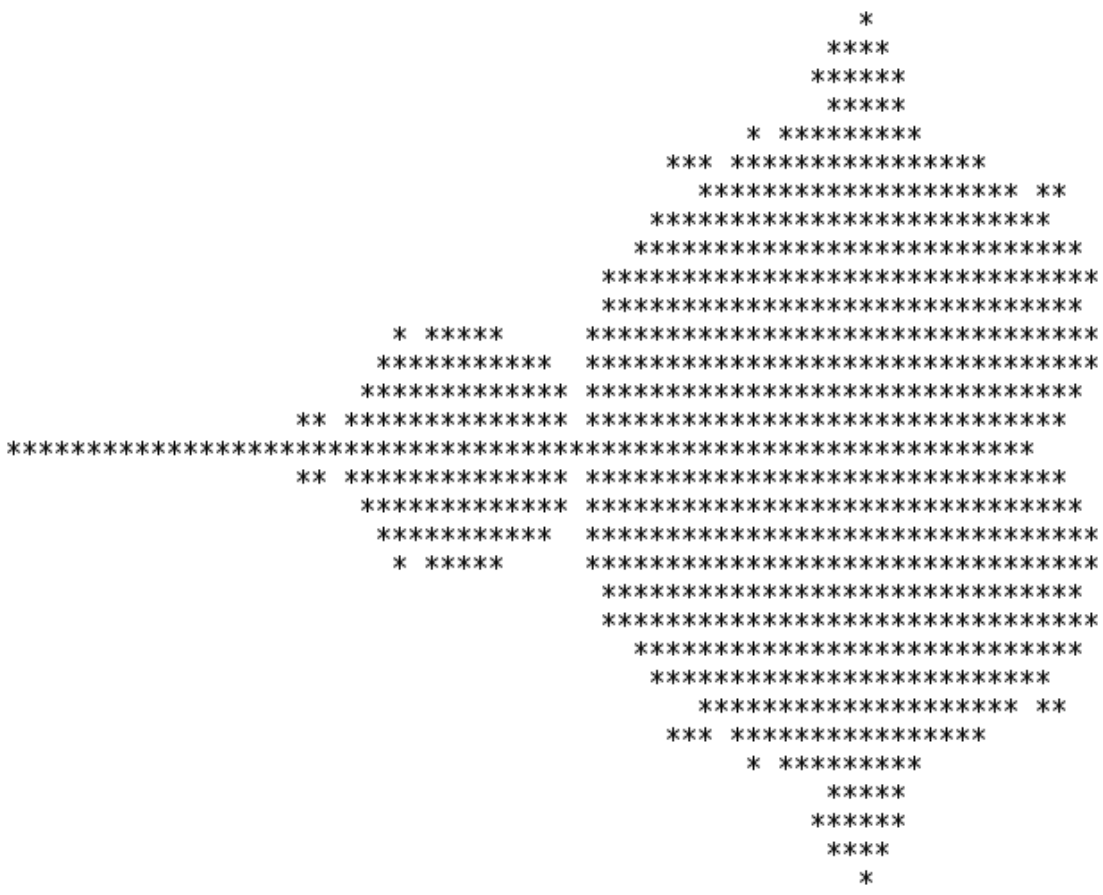
$$\begin{cases} z_0 = 0, \\ z_{n+1} = z_n^2 + c \ (n > 0). \end{cases}$$

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.

La première représentation de l'ensemble (en 1978)



https://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot

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 :

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

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 C mais la suite peut se réécrire comme suit sur base des parties réelles et imaginaires :

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

où $\mathrm{Re}(z)$ et $\mathrm{Im}(z)$ 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 $\mathcal{M}$:

```c
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 ?

Vérifier l'appartenance à $\mathcal{M}'$ (en C) 2/3

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

```c
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 précédence)

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

```c
#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 pour une specification 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 !

Vérifier l'appartenance à $\mathcal{M}'$ (en C) 3/3

On peut emballer ce code dans une fonction

```c
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

```c
#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;
}
```

```c
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;
}
```

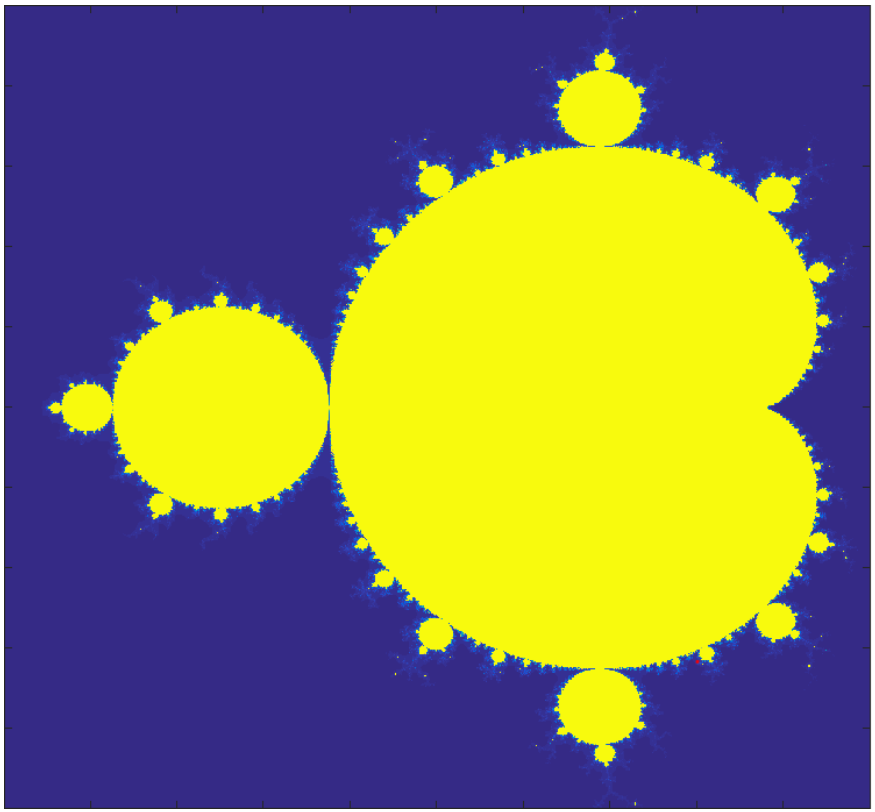
Utilisation :

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

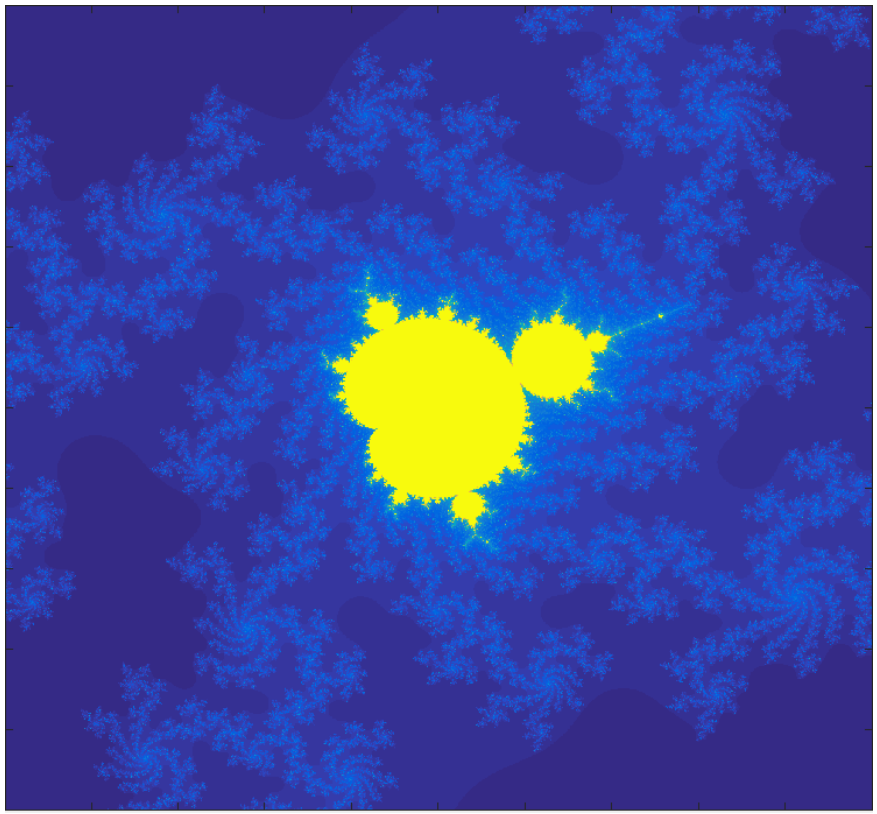
Pour voir l'image avec matlab

```
> 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

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

```c
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 libraire `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 multidimensionel
```

- 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.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);
...
```

complex.c

```
#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

```c
// 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 '.'.

Autre variante utilisant un nouveau type 2/2

```c
#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;
}
```

```c
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 mandlebrot.c complex.c -o mandelbrot
```

Autre variante utilisant la librairie complex du C99

```c
#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;
}
```

```c
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 mandlebrot.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[1], du code.

1. à complexité constante, cf. Partie 3

Un mauvais exemple

Que fait ce code ?

```c
#include <stdio.h>
main() {
   float C,l,c,o,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

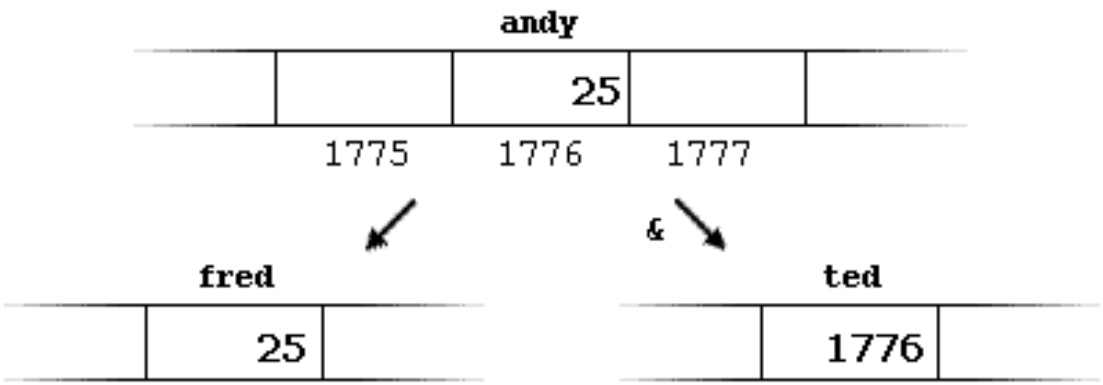
Un mauvais exemple

Plan

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).

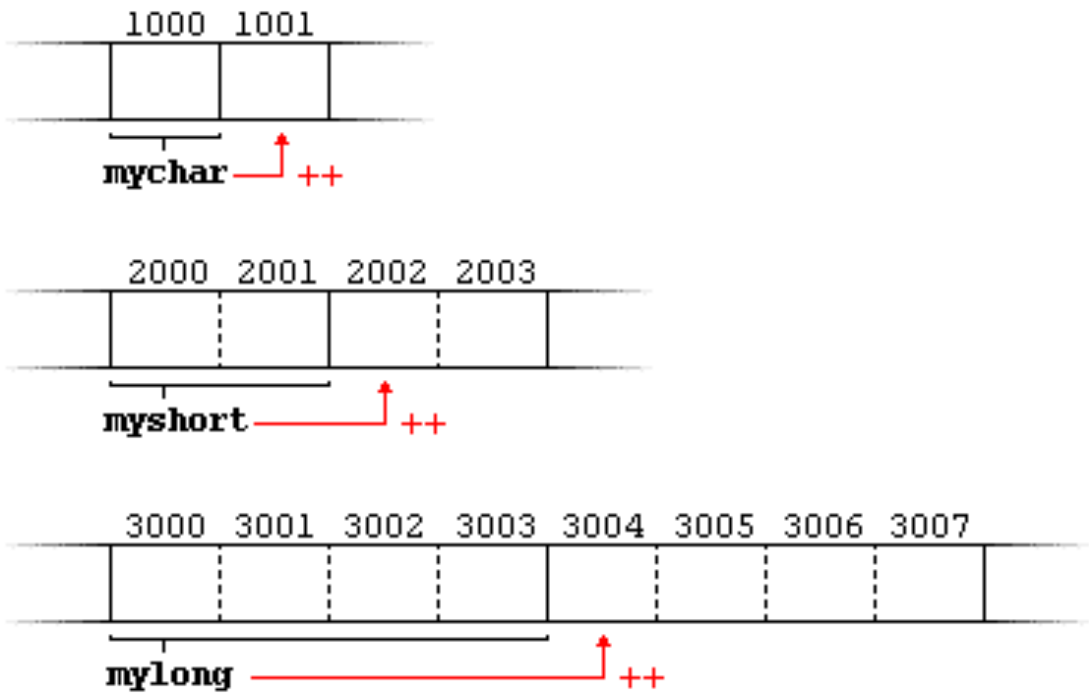
```
int andy = 25;
int fred = andy;
int *ted = &andy; // & dénote l'adresse de la variable andy
```



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 ?

```c
#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`.

```c
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

```c
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 abbréviation 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;
}

...
```