Chapitre 1 :
Les concepts de base

Data Structures & Algorithms

Université Mohamed 1er
Faculté des sciences
Oujda

Prof. A. DARGHAM
Printemps 2017

Introduction aux structures de données

- **Programme = Structures de données + Algorithmes**
- **Définition** :
  - Une **structure de données** est une **représentation des relations logiques** qui existent entre les **éléments individuels** d'une **catégorie de données**.

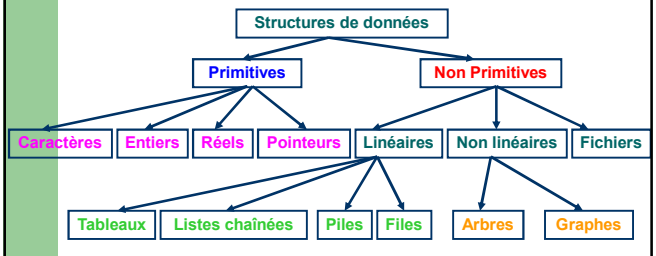
Introduction aux structures de données

- Les **structures de données** peuvent être :
  - **Linéaires** :
    - **Tableaux**
    - **Listes chaînées**
    - **Piles**
    - **Files**
  - **Non linéaires** :
    - **Arborescentes** : Arbres
    - **Relationnelles** : Graphes.

Introduction aux structures de données

- Les **structures de données** peuvent être :
  - **Primitives** (individuels) : entiers, réels, caractères, booléens, pointeurs.
  - **Non primitives** (composés) : tableaux, listes, fichiers, …etc.

Introduction aux structures de données



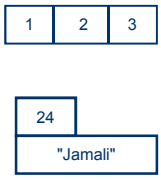
Introduction aux structures de données

- Organisation des données en mémoire :
  - **Contiguë** : les éléments se trouvent en des **adresses consécutives**, l'un après l'autre.
  - **Non contiguë** : les éléments se trouvent en des **adresses dispersées**.

Introduction aux structures de données

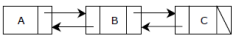
- **Exemples (structures contiguës)**

- **int** a[3] = {1, 2, 3};
- **struct** etudiant
  {
  int age;
  char nom[50];
  } etud1 = {24, "Jamali"};

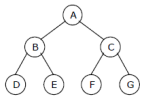


Introduction aux structures de données

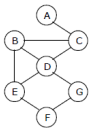
- **Exemples (structures non contiguës) :**



(1) Liste



(2) Arbre



(3) Graphe

Notion de typage

- **Définition :**
  - Un **type de données** est l'association à un objet (une variable) :
    - D'un **ensemble de valeurs possibles**,
    - D'un **ensemble d'opérations** admissibles sur ces valeurs.

Notion de typage

- **Types scalaires :**
  - Une **seule information élémentaire**.
  - Se divisent en 2 catégories :
    - **Types discrets** : énumérables (dénombrables) + nombre fini de valeurs.
    - **Type réel** : non énumérable (non dénombrable).

Notion de typage

- **Types discrets :**
  - **Entiers** : type **int** du C (4 octets), avec les opérations arithmétiques et logiques et de comparaison.
  - **Types énumérés** : booléens (pas en C), les caractères (type **char** en C, 1 octet) et les énumérations définies par l'utilisateur (**enum** en C). Exemple :
  - **Intervalles** : générés en restreignant l'ensemble des valeurs possibles d'un type de base.

Notion de typage

- **Exemple du type énumération en C :**

```
typedef enum {False, True} BOOL;
…
BOOL find = False;
…
if(find == False)
  printf("Not found\n");
```

Notion de typage

- **Types réels du C** :
  - **float** : 4 octets (VF simple précision).
  - **double** : 8 octets (VF double précision).
- **Méthodes de codage du type réel** :
  - **Virgule fixe**
  - **Virgule flottante**
  - **Décimal codé binaire** (*Binary Coded Decimal*)

Les structures statiques

- **Caractérisation** :
  - **Ne changent que de valeurs**, **jamais de structure** ou **d'ensemble de valeurs de base**.
- **Conséquence** :
  - **L'espace mémoire** qu'elles occupent **reste constant**.

Les structures statiques

- **Structures statiques simples** :
  - **Les tableaux à une seule dimension** :
    - **Agrégat** d'éléments de **même type de base**.
    - **Sélecteur** : **indice** (entier).
  - **Les enregistrements** (**records**) :
    - **Agrégat** d'éléments de **types différents** (**champs**, **membres**).
    - **Sélecteur** : **champ**.

Les structures statiques

- **Structures statiques complexes** :
  - **Les tableaux à plusieurs dimensions**.
  - **Les tableaux d'enregistrements**.

Les structures dynamiques

- **Caractérisation** :
  - Structures dont la **taille peut varier en cours d'exécution**.
- **Classification** :
  - **Structures linéaires** : fichiers, tableaux, chaînes de caractères, listes chaînées, piles et files.
  - **Structures arborescentes** : arbres.
  - **Structures relationnelles** : graphes et dictionnaires.

Les structures récursives

- **Caractérisation** :
  - Structure dont la **définition comporte une référence à elle-même**.
  - Elle est généralement basée sur un type enregistrement.
  - S'il n'y a qu'une seule référence, cette structure est linéaire, et s'il y en a plusieurs, elle est non linéaire.
  - Pour beaucoup de langages, ces **références** se font à l'aide du **type pointeur**.

Les pointeurs

- **Définition** :
  - Une **variable de type pointeur** est une **variable dont la valeur indique l'emplacement** (l'**adresse**) en mémoire d'une autre variable **crée dynamiquement lors de l'exécution** et ayant un type de base précis (**type de l'objet pointé**).

Les pointeurs

- **Rôle des pointeurs** :
  - Permettre l'accès à une structure qui serait crée lors de l'exécution.
- **Pointeurs en C** :
  - On peut définir un pointeur vers n'importe quel type (prédéfini ou non).
  - On peut même définir des **pointeurs vers des fonctions**.

Les pointeurs

- **Fonctions de la gestion de la mémoire dynamique** :
  - Elles sont disponibles dans <stdlib.h> :
    - **malloc**() : allouer un espace mémoire.
    - **calloc**() : allouer un tableau en mémoire.
    - **realloc**() : réallocation de l'espace mémoire.
    - **free**() : libération de l'espace mémoire.

Les enregistrements en C (struct)

- **Définition** :
  - Une **structure** (ou enregistrement) est une **collection de valeurs** qui **représentent une unité logique**.
  - Les éléments d'une structure sont appelés **membres** (ou **champs**) et peuvent être de **types différents**.

Les enregistrements en C (struct)

- **Caractéristiques** :
  - Chaque **champ** d'une structure porte un **nom**.
  - Les **champs** d'une structure sont stockés en mémoire dans un **emplacement contigu**.
  - La **taille d'une structure** est la **somme des tailles** de ses **champs**.

Les enregistrements en C (struct)

- **Déclaration d'une structure (Première forme)** :

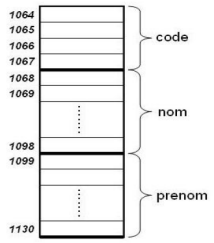
```
struct
{
  int  code;
  char nom[31];
  char prenom[31];
} employe1, employe2;
/* employe1 et employe2 sont deux structures ayant trois
   champs : code, nom et prenom */
```

Les enregistrements en C (struct)

Caractéristiques

- **L'adresse en mémoire** d'une structure est l'adresse du **1er octet de son 1er champ**.
- Si le champ **code** de la variable **employe1** est stocké à l'adresse 1064 :
- Le champ **nom** de la même variable sera stocké à l'adresse 1064 + 4 = 1068
- Le champ **prenom** de la même structure sera stocké à l'adresse 1068 + 31 = 1099.

Les enregistrements en C (struct)



Les enregistrements en C (struct)

- **Déclaration d'une structure (Deuxième forme) :**

```
struct employe
{
  int  code;
  char nom[31];
  char prenom[31];
} e1, e2;
/* e1 et e2 sont deux variables de type struct employe
   */
```


Les enregistrements en C (struct)

- **Déclaration d'une structure (Troisième forme) :**

```
typedef struct employe   {
  int code;
  char nom[31];
  char prenom[31];
} employe;       /* Aléas : un autre nom de type */
// Déclaration de variables de ce type :
struct employe e1, e2; /* valable dans ce cas */
employe e3, e4;
```


Les enregistrements en C (struct)

- **Initialisation d'une structure (1) :**

```
struct
{
  int code;
  char nom[31];
  char prenom[31];
} e1 = {117, "Ben Ali", "Farida"};
```


Les enregistrements en C (struct)

- **Initialisation d'une structure (2) :**

```
struct employe
{
  int code;
  char nom[31];
  char prenom[31];
} e1 = {117, "Ben Ali", "Farida"};

struct employe e2 = {118, "Ben Yahia", "Said"};
```

Les enregistrements en C (struct)

- **Initialisation d'une structure (3) :**

```
typedef struct employe   {
  int code;
  char nom[31];
  char prenom[31];
} employe;  /* pas d'initialisation ici */
employe e1 = {117, "Ben Ali", "Farida"},
struct employe e2 = {118, "Ben Yahia", "Said"};
```


Les enregistrements en C (struct)

- **Initialisation d'une structure (4) :**

```
struct
{
  int code;
  char nom[31];
  char prenom[31];
} e = {.prenom = "Farida", .code = 117};
```


Les enregistrements en C (struct)

- **Accès au membre d'une structure :**
  - Les **membres d'une structure** sont **accessibles via leurs noms**.
  - L'opérateur d'accès au membre d'une structure est « **.** » (**point**), via la syntaxe :

```
nameOfStructure.nameOfField
```

  - Les **membres d'une structure** doivent être **manipulées comme des variables**.

Les enregistrements en C (struct)

- **Exemple :**

```
#include <stdio.h>

typedef struct employe
{
  int code;
  char nom[31];
  char prenom[31];
} employe;
```


Les enregistrements en C (struct)

- **Exemple (suite) :**

```
main() {
  employe e1;
  e1.code = 117;
  /* Accès au champ code de la structure e1 */
  gets(e1.nom);
  /* Lecture du champ nom de la structure e1 */
  printf("Prenom de e1 : %s\n", e1.prenom);
  /* Affichage du champ prenom de la structure e1 */
  system("pause");
}
```


Les enregistrements en C (struct)

- **Affectation globale entre structures :**
  - Par opposition aux tableaux, on peut effectuer une **affectation globale** d'une structure à une autre.
  - Cela entraîne la **recopie des valeurs** des champs de la structure source dans les champs correspondants de la structure cible.
  - L'affectation s'effectue via **l'opérateur "="** selon la syntaxe habituelle.

Les enregistrements en C (struct)

- **Exemple** :

```
// … définition du type employe ici
main()  {
  employe e1 = {117, "Meftah", "Yasser"};
  employe e2;

  e2 = e1; /* Affectation globale de structures */
  system("pause");
}
```


Les enregistrements en C (struct)

- **Tableaux de structures** :

```
typedef struct employe  {
  int code;
  char nom[31];
  char prenom[31];
} employe;
employe tabEmployes[100];
```


Les enregistrements en C (struct)

- **Tableaux de structures** :
  - Pour un entier i, `tabEmployes[i]` désigne le (i+1)ieme employé et `tabEmployes[i].code` est son code.
  - Comme exemple, le code suivant permet d'affecter la valeur 117 au code du premier employé et changer le prénom du dernier employé à "Fayrouz" :

Les enregistrements en C (struct)

- **Tableaux de structures** :

```
tabEmployes[0].code = 117;
strcpy(tabEmployes[99].prenom,"Fayrouz");
```


Les enregistrements en C (struct)

- **Pointeurs vers structures** :

```
typedef struct employe  {
  int code;
  char nom[31];
  char prenom[31];
} employe;
employe *pE;
/* pE : pointeur vers la structure employe */
```


Les enregistrements en C (struct)

- **Pointeurs vers structures** :

```
// ……

(*pE).code = 117;

/* Accès au champ code de la structure
 pointée par pE */
```

Les enregistrements en C (struct)

- **Pointeurs vers structures** :
  - L'opérateur "**->**" permet **d'accéder** au **membre d'une structure désignée par un pointeur**.
  - Son opérande gauche doit être un **pointeur vers une structure**.
  - Par exemple, au lieu d'écrire (***pE**).**code**, on peut utiliser la notation abrégée équivalente : **pE->code**

Les enregistrements en C (struct)

- **Pointeurs vers structures** :

```
pE->code = 117;
/* ou bien (*pE).code = 117; */
gets(pE->nom);
/* ou bien gets((*pE).nom); */
puts(pE->prenom);
/* ou bien puts((*pE).prenom); */
```


Les enregistrements en C (struct)

- **Imbrication de structures** :
  - Le **champ** d'une structure peut être de **type structure** lui aussi.
  - Il est donc possible d'**imbriquer** une structure dans une autre.
  - Par exemple, pour l'adresse d'un employé est une structure comportant : un numéro de rue, un nom de ville et un numéro de maison (au lieu d'une chaîne de caractères).

Les enregistrements en C (struct)

- **Exemple** :

```
/* Déclaration de la structure adresse */
typedef struct adresse {
  int num_rue;
  int num_maison;
  char ville[25];
} adresse;
```


Les enregistrements en C (struct)

- **Exemple (suite)** :

```
/* Déclaration de la structure employe */
typedef struct employe  {
  int code;
  char nom[31];
  char prenom[31];
  adresse adr;
} employe;
```


Les enregistrements en C (struct)

- **Exemple (suite)** :

```
main()  {
  employe e1;
  adresse a1 = {25, 107, "Oujda"};
  e1.code = 117;
  strcpy(e1.nom, "Yazidi");
  strcpy(e1.prenom, "Malika");
  e1.adr = a1;
  puts(e1.adr.ville);
}
```

Les unions

- **Définition** :
  - Une **union** est une **structure spéciale** qui utilise **un même espace mémoire** pour stocker les différents champs.
  - Les champs d'une union partagent alors le même emplacement mémoire.
  - La **taille d'une union** est la **taille de son plus grand champ**.

Les unions

- **Caractéristiques** :
  - Un **seul champ** est **actif** à un moment donné.
  - L'affectation d'une nouvelle valeur à un champ **détruit automatiquement** les valeurs des autres champs.
  - Les situations d'utilisation des unions sont rares en général.
  - Un bon exemple de leur utilisation est la **table des symboles** d'un compilateur.

Les unions

- **Exemple** :

```
typedef union nombre  {
  int ival;       /* 4 octets */
  double fval; /* 8 octets */
} nombre;       /* taille globale : 8 octets */
nombre x;
x.ival = 125;
x.dval = -75.899;   /* la valeur de ival est détruite */
```


Les unions

- **Utilisation des unions** :
  - Comme les structures :
    - On peut déclarer un **tableau d'unions** ou un **pointeur vers une union**.
    - Une **fonction** peut avoir comme argument une **union** ou un **pointeur vers une union**.
    - Une **fonction** peut retourner une **union** ou un **pointeur vers une union**.

Les unions

- **Utilisation des unions** :
  - Il est également possible **d'initialiser** une **union** au moment de sa déclaration, mais la liste d'initialisation ne doit comporter qu'**un seul initialisateur.**
  - Si aucun membre n'est désigné explicitement, c'est le **premier membre** de l'union qui sera concerné par l'initialisation.

Les unions

- **Utilisation des unions** :

```
nombre x = {99.99};
/* c'est le membre ival de x qui recevra la valeur
   99*/
```

Les énumérations

- **Définition** :
  - Un **type énuméré** permet de représenter un **domaine de valeurs de taille petite**.
  - En C, un type énuméré est un **sous-ensemble fini de valeurs** de type **int**.
  - Exemples de tels domaines : booléens, jours de la semaine, mois de l'année, situation de famille, …etc.

Les énumérations

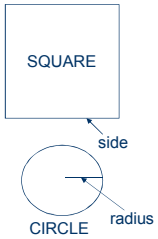
- **Exemple d'utilisation** :

```
enum {JUIN; JUILLET, AOUT} mois_ete;
mois_ete = JUIN;  /* ou bien mois_ete = 0; */
enum feu {ROUGE; ORANGE, VERT} f1, f2;
enum feu f3 = VERT;
typedef enum forme {CERCLE, CARREE, TRIANGLE}
   forme;
forme fr1 = CARREE;
enum forme fr2 = fr1;
```


Structures, unions et énumérations

- **Exemple reliant structures, unions et énumérations** :

```
#include <stdio.h>
#include <math.h>
typedef struct figure {
  enum {SQUARE, CIRCLE} form;
  union {
    double side;
    double radius;  }
} figure;
```



Structures, unions et énumérations

```
double getArea(figure f)
{
  switch(f.form)
  {
    case SQUARE :
       return pow(f.value.side, 2.0);
    case CIRCLE :
       return 3.14159 * pow(f.value.radius, 2.0);
  }
}
```


Structures, unions et énumérations

```
main()
{
  figure f1 = {SQUARE, {10.0}};
  figure f2 = {CIRCLE, {.radius = 5.0}};
  double a1 = getArea(f1);
  double a2 = getArea(f2);

  printf("Area of the square : %.2lf\n", a1);
  printf("Area of the circle : %.2lf\n", a2);
  system("pause");
}
```


Les fonctions

- **Réalisation des opérations via des fonctions** :
  - **Type** = **Données** + **Opérations**
  - Les **données** sont définies en utilisant des **structures de données** et les **opérations** en utilisant des **fonctions**.
  - Le **mécanisme** utilisé en C pour implémenter de nouvelles opérations sur les données est la **fonction**.

Les fonctions

- **Définition** :
  - Une **fonction** est un **bloc de code** C qu'on peut utiliser dans un programme en **évoquant son nom** et en lui fournissant des **paramètres**.
  - Toute **fonction** est essentiellement un **petit programme** avec ses propres **déclarations** et **instructions**.

Les fonctions

- **Avantages** :
  - **Modularité** : les fonctions nous permettent de diviser un programme en un certain nombre de petites fonctions qui sont **faciles à comprendre** et à **modifier**.
  - **Réutilisabilité** : on peut prendre une fonction déjà écrite et l'utiliser dans n'importe quel autre programme.

Prototypes

Les fonctions

- **Déclaration d'une fonction** :

```
int add(int a, int b);  /* ou : int add(int, int);  */
/* calcule la somme de deux entiers */

void printInt(int n);
/* affiche un entier */

int readInt(void);
/* lit un entier au clavier et le renvoie */
```


Les fonctions

- **Types de retour d'une fonction C** :
  - Types de retour d'une fonction C : **types de base** (char, int, float, double) **pointeurs**, **structures** **unions** ou **énumérations**.
  - Cependant, une **fonction C** **ne peut jamais retourner un tableau**.
  - Le type **void** est utilisé pour indiquer que la fonction ne retourne aucune valeur : il s'agit plutôt d'une **procédure**.

Les fonctions

- **Définition d'une fonction** :
  - **Définir une fonction** = **écrire son code** (ses instructions) en utilisant la syntaxe du C.
  - Tout programme C doit contenir une **définition de la fonction main** (c'est le point d'entrée du programme).

Les fonctions

- **Définition d'une fonction** :
  - Le **corps d'une fonction** est identique à celui de la fonction **main** et contient donc :
    - Des **déclarations de variables locales**.
    - Des **instructions exécutables**.
  - Les traitements effectués par une fonction doivent s'exprimer en fonction des **paramètres formels** et des **variables locales** de celle-ci.

Les fonctions

- **Instruction return** :
  - L'instruction **return** permet d'achever (**terminer**) l'exécution de la fonction et permet aussi de **fournir la valeur de retour** au programme appelant.

Les fonctions

- **Appel d'une fonction** :
  - Une fonction est **appelée** en écrivant son **nom**, suivi de la **liste des valeurs des arguments** (**paramètres effectifs**).
  - Lorsqu'une fonction est appelée, le système interrompue l'exécution du programme appelant et **exécute les instructions de la fonction**.

Les fonctions

- **Exemple** :

```
#include <stdio.h>
#define N  50
/* Déclaration des fonctions */
int add(int, int);
int readInt();
```


Les fonctions

- **Exemple (suite)** :

```
/* Le programme principal */
main()  {
  int a, b, c;

  c = add(10, 20);
  // un appel de la fonction add
```


Les fonctions

- **Exemple (suite)** :

```
  printf("Resultat = %d\n", c);
  // affiche : Resultat = 30
  c = add(N, 20);
  printf("res = %d\n", c); // affiche : res = 70
  a = readInt();
  b = readInt();
  // deux appels de la fonction readInt
```


Les fonctions

- **Exemple (suite)** :

```
  c = add(a, b);
  printf("Resultat = %d\n", c);
  c = add(add(10, 20), N * 2);
  printf("Resultat = %d\n", c);
  // affiche : Resultat = 130
  system("pause");
}
```

Les fonctions

- **Exemple (suite)** :

```
/* Définitions des fonctions */
int add(int a, int b)
{
  return a + b;
}
```


Les fonctions

- **Exemple (suite)** :

```
int readInt()
{
  int x; /* variable locale */
  printf("Donner un entier : ");
  scanf("%d", &x);
  return x;
}
```


Les fonctions

- **Paramètres effectifs d'une fonction** :
  - **Valeur constante**.
  - **Constante symbolique**.
  - **Variable**.
  - **Expression combinant ces trois éléments**.

Les fonctions

- **Paramètres effectifs d'une fonction** :
  - En fin de compte, c'est la **valeur de cette expression** qui sera **transmise à la fonction** lors de son appel.
  - Attention, les paramètres effectifs doivent "correspondre" aux paramètres formels :
    - **Même ordre de passage** des paramètres.
    - **Mêmes types**.

Les fonctions

- **Modes de passage des arguments** :
  - Le **passage par valeur** :
    - La fonction reçoit une ***copie de la valeur du paramètre effectif***.
  - Le **passage par adresse** (***via un pointeur***) :
    - La fonction reçoit ***l'adresse du paramètre effectif***.

Les fonctions

- **Modes de passage des arguments** :
  - Les paramètres **passés par valeur** **ne peuvent jamais être modifiés** par une fonction.
  - Les **paramètres passés par adresse** **peuvent être modifiés** par une fonction.

Les fonctions

- **Exemples** :

```
#include <stdio.h>
int numberOfDigits(int n) {
/* le paramètre n est passé par valeur */
  int k = 1;
  while(n >= 10)  {
    k++;
    n /= 10;        }
  return k;
}
```


Les fonctions

- **Exemples (suite)** :

```
void twice(int *p)
{
  /* le paramètre p est passé par adresse */
  *p = 2 * (*p);
}
main()  {
  int n = 107, k;
```


Les fonctions

- **Exemples (suite)** :

```
  k = numberOfDigits(n);
  printf("Nombre de chiffres de %d : %d\n", n, k);
  printf("Valeur de n = %d\n", n);
  /* affiche 107 et non pas 1 */
  twice(&n);
  printf("Valeur de n = %d\n", n);
  /* affiche 214 et non pas 107 */
  system("pause");
}
```


Les fonctions

- **Cas des tableaux** :
  - Les tableaux passés en paramètres **peuvent être modifiés** par une fonction.
  - En effet, le **nom d'un tableau** est **l'adresse du premier octet de son premier élément** (un tableau étant un **pointeur constant**, mais ses éléments ne le sont pas).

Les fonctions

- **Exemple** :

```
#include <stdio.h>
void clearAll(int a[ ], int n)
{
  int i;
  for(i = 0; i < n; i++)
      a[i] = 0;
 }
```


Les fonctions

- **Exemple (suite)** :

```
main()  {
  int a[5] = {-1, 2, 0, 3, 7};
  int n = 5, i;

  printf("Tableau avant appel :\n");
  for(i = 0; i < n; i++)
     printf("%d ", a[i]);
```

Les fonctions

- **Exemple (suite)** :

```
printf("\n");
clearAll(a, n);  /* appel de la fonction */
printf("Tableau apres appel :\n");
for(i = 0; i < n; i++)
    printf("%d ", a[i]);
printf("\n");
system("pause");
}
```


Les fonctions

- **Cas des structures** :
  - Une **structure** peut être **passée à une fonction** (soit par valeur, soit par adresse).
  - Lors d'un **passage par valeur**, **tous les champs** de la **structure représentant le paramètre effectif** seront **recopies** dans les **champs correspondants de la structure représentant le paramètre formel** (**recopie membre à membre**).

Les fonctions

- **Cas des structures** :
  - Cela génère un **coût supplémentaire**.
  - Lors d'un **passage par adresse**, **seule l'adresse de la structure** représentant le paramètre effectif **sera transmise** : **il n'y aura pas de recopie des champs** de la structure qui représente le paramètre effectif.
  - Une **fonction** peut également **retourner une structure** ou un **pointeur vers une structure**.

Les fonctions

- **Exemples** :

```
typedef struct point
{
  double x;   /* Abscisse du point*/
  double y;   /* Ordonné du point*/
} point;
```


Les fonctions

- **Exemples (suite)** :

```
void print(point A)  {
  printf("Je suis en (%.3lf, %.3lf)\n", A.x, A.y);
}
void move(point *A, double dx, double dy) {
  A->x += dx;
  A->y += dy;
}
```


Les fonctions

- **Exemples (suite)** :

```
double distance(const point *A, const point *B)
{
  double a = pow(A->x - B->x, 2.0);
  double b = pow(A->y - B->y, 2.0);
  return sqrt(a + b);
}
```

Les fonctions

- **Exemples (suite)** :

```
point* middle(const point *A, const point *B) {
  point *M = (point*) malloc(sizeof(point));

  M->x = (A->x + B->x) / 2.0;
  M->y = (A->y + B->y) / 2.0;
  return M;
}
```


Les fonctions

- **Paramètres à la fois résultats** :
  - Si un argument d'une fonction est **aussi résultat** de cette fonction, cet argument doit être passé par **adresse**.
  - **Exemple** : la fonction suivante prend un premier argument de type double et calcule sa partie entière dans un second argument de type long et sa partie fractionnaire dans un troisième argument de type double.

Les fonctions

- **Exemple** :

```
#include <stdio.h>
void decompose(double x, long *partInt,
   double *partFrac)
 {
   *partInt = (long) x;
   *partFrac = x - *partInt;
}
```


Les fonctions

- **Exemple** :

```
main()  {
  long i;
  double x;

  decompose(3.14159, &i, &y);
   /* i = 3 et y = 0.14159 */
  system("pause");
 }
```


Les fonctions

- **Fonctions retournant un pointeur** :
  - Une fonction peut également **retourner un pointeur**.
- **Exemple** :

```
int* middle(int a[ ], int n)
{
  return &a[n / 2];
}
```


Les fonctions

- **Pointeurs vers fonctions** :
  - En C, le **nom d'une fonction** est un **pointeur constant vers la fonction**.
  - Sa valeur correspond à l'adresse du codage en mémoire de la fonction.
  - Par exemple, le nom **scanf** est un pointeur sur la fonction **scanf** qui lit une valeur et la place dans une variable :

scanf == &scanf

Les fonctions

- **Pointeurs vers fonctions** :
  - Un **avantage des pointeurs vers une fonction** est la possibilité d'écrire une **fonction ayant comme argument une fonction**.

Les fonctions

- **Pointeurs vers fonctions** :
  - Par exemple, pour calculer le **taux d'accroissement**

    (f(b) - f(a)) / (b − a)

    de n'importe quelle **fonction f** sur un intervalle [a, b], on peut utiliser une fonction ayant comme premier argument un pointeur vers une fonction retournant un double et ayant comme argument un double.

Les fonctions

- **Exemple** :

```
#include <stdio.h>
#include <math.h>
double Accroissement(double (*pFct)
   (double), double a, double b)
{
  return (pFct(b) - pFct(a)) / (b - a);
}
```


Les fonctions

- **Exemple (suite)** :

```
main()  {
  double a = 2.0, b = 3.0, t1, t2, t3;

  t1 = Accroissement(sin, a, b);
  t2 = Accroissement(cos, a, b);
  t3 = Accroissement(sqrt, a, b);
```


Les fonctions

- **Exemple (suite)** :

```
  puts("Accroissement de sin sur [2, 3] :");
  printf("%lf\n", t1);  /* t1 = -0.768177 */
  puts("Accroissement de cos sur [2, 3] :");
  printf("%lf\n", t2);  /* t2 = -0.573846 */
  puts("Accroissement de sqrt sur [2, 3] :");
  printf("%lf\n", t3);  /* t3 = 0.317837 */
  system("pause");
}
```


Les fonctions

- **Remarque** :
  - Une **fonction** peut avoir comme argument un pointeur vers une fonction, mais **elle ne peut jamais renvoyer un pointeur vers une fonction**.

Les fonctions récursives

- **Définition** :
  - Une **fonction récursive** est une fonction qui dans sa définition, **appelle elle-même** (directement ou indirectement).

Les fonctions récursives

- **Conception d'une fonction récursive** :
  - Une fonction récursive met en jeu deux éléments :
    - Une **base** : la fonction n'effectue aucun appel récursif et procède d'une manière directe. Elle définit aussi la **condition d'arrêt** de la fonction.
    - Une **récurrence** : la fonction effectue au moins un appel récursif sur un paramètre plus petit.

Les fonctions récursives

- **Exemple 1 (récursivité terminale)** :

```
int fact(int n)
{
  if(n == 0)  // la base
    return 1;
  return n * fact(n - 1);  // la récurrence
}
```


Les fonctions récursives

- **Exemple 1 (récursivité terminale)** :

```
int fibo(int n)
{
  if(n == 0 || n == 1)  // la base
    return 1;
  return fibo(n - 1) + fibo(n - 2);  // la récurrence
}
```


Les fonctions récursives

- **Exemple 3 (récursivité non terminale)** :

```
int getMin(int a[ ], int start, int end)  {
  if(start == end)
    return a[start];
  else {
    int middle, min1, min2;
    middle = (start + end) / 2;
    min1 = getMin(a, start, middle);
    min2 = getMin(a, middle + 1, end);
    return ((min1 <= min2) ? Min1 : min2);  }
}
```


Les fonctions récursives

- **Exemple 4 (récursivité croisée)** :

```
#include <stdio.h>
typedef enum {False, True } BOOL;

/* Déclaration obligatoire */

BOOL isEven(int n); // est pair
BOOL isOdd(int n);  // est impair
```

Les fonctions récursives

- **Exemple 4 (récursivité croisée)** :

```
BOOL isEven(int n)
{
  if(n == 0)
    return True;
  return isOdd(n - 1);
}
```


Les fonctions récursives

- **Exemple 4 (récursivité croisée)** :

```
BOOL isOdd(int n)
{
  if(n == 0)
    return False;
  return isEven(n - 1);
}
```


Les fonctions récursives

- **Récursive sur les données** :
  - La **récursivité** est un concept général fortement lié à la notion de récurrence en mathématiques.
  - Les **données peuvent également être de nature récursive**.

Les fonctions récursives

- **Récursive sur les données** :
  - Par exemple, **l'ensemble des nombres entiers** est **récursif** :
    - 0 est un nombre entier (la base).
    - Si n est un nombre entier, alors n + 1 est aussi un nombre entier (la récurrence).

Les fonctions récursives

- **Récursive sur les données** :
  - Un autre exemple est **l'ensemble des mots palindromes (mots qui coïncident avec leurs inverses) non vides sur l'alphabet {a,b}** est **récursif** :
    - a et b sont deux palindromes (la base).
    - Si x est un palindrome, alors axa et bxb sont des palindromes.

Les déclarations complexes

- **Les déclarateurs** :
  - Un **déclarateur** C consiste en un **identificateur** (le nom de la variable ou de la fonction à déclarer) éventuellement précédé d'un symbole « * » ou suivi d'un **[ ]** pour indiquer un **tableau**, ou d'un **( )** pour indiquer une **fonction**.
  - En combinant ces trois symboles avec un nom de type de base, on peut définir des **déclarations complexes**.

Les déclarations complexes

- **Méthode pour déchirer une déclaration complexe** :
  1. Repérer tout d'abord l'**identificateur**.
  2. Lorsqu'il y a un choix, **favoriser [ ]** et **( )** sur *.
  3. Si l'identificateur est enfermé dans une parenthèse, commencer par les symboles enfermés dans ces parenthèses.

Les déclarations complexes

- **Exemples** :

```
int a[10];
/* a : un tableau de 10 entiers */
int (*p)[10];
/* p : un pointeur vers un tableau de 10 entiers */
int *tp[10];
/* tp : un tableau de 10 pointeurs vers entiers */
```


Les déclarations complexes

- **Exemples** :

```
int (*pf) (int, int);
/* pf : un pointeur vers une fonction ayant comme arugments deux int et retournant un int */
int *(*x[10]) (int *);
/* x : un tableau de 10 pointeurs vers une fonction ayant comme argument un pointeur vers int et retournant un pointeur vers int */
```


Les déclarations complexes

- **Usage de typedef** :
  - Le mot-clé **typedef** permet de **nommer** un type existant (**réalise un aléas de type**).
  - Il est préférable d'utiliser **typedef** dans une série de déclarations élémentaires pour augmenter la lisibilité et la compréhensibilité d'une déclaration complexe.

Les déclarations complexes

- **Exemples de l'usage de typedef** :

```
typedef int entier;
typedef int tabDix[10];
// int *(*x[10]) (int *);
typedef int * Fct (int*);
typedef Fct *PtrFct;
typedef PtrFct TabDixPtrFct[10];
TabDixPtrFct x;
```


Les modules

- **Notion de module** :
  - La **modularité** est l'un des **mécanismes fondamentaux** de la **génie logicielle** et de la **programmation**.
  - Un **logiciel** (ou un **programme**) est **modulaire** s'il est découpé en **plusieurs unités largement indépendantes**.
  - Ces **unités** sont aussi appelées des **modules**.

Les modules

- **Définition** :
  - Un **module** est un **ensemble de fonctions** (ou services) traitant des **données communes**.
  - Il est composé de deux éléments :
    - L'**interface**.
    - L'**implémentation**.

Les modules

- **Interface** :
  - Elle est stockée dans un fichier à part appelé **fichier d'en-tête** et porte l'extension "**.h**".
  - Ce fichier regroupe les **déclarations de constantes**, de **variables**, de **types de données** et de **fonctions visibles au monde extérieur** (c'est-à-dire accessibles depuis l'extérieur du module).

Les modules

- **Implémentation** :
  - Elle est mémorisée dans un autre fichier (de préférable de même nom que le fichier d'en-tête) avec l'extension "**.c**" et **référence la partie interface** en incluant le fichier d'en-tête.
  - Ce fichier regroupe les **déclarations de constantes**, de **variables**, de **types** et de **fonctions locales** au module, mais aussi les **définitions de toutes les fonctions déclarées dans l'interface**.

Les modules

- **Avantages de la modularité** :
  - **Abstraction** : les utilisateurs utilisent les modules comme des **types de données abstraits** (***TDA***). Ils savent comment utiliser les services offerts par les interfaces des modules sans connaître les détails de l'implémentation.

Les modules

- **Avantages de la modularité** :
  - **Réutilisabilité** : un module présentant des **services génériques** est potentiellement réutilisable dans d'autres programmes.
  - **Maintenabilité** : il est plus facile de repérer les erreurs d'un module que celles d'un plus grand programme. Une fois les erreurs d'un module sont détectées et fixées, il ne reste qu'à recompiler le module séparément sans affecter le reste du programme.

Les modules

- **Deux propriétés importantes** :
  - **Cohésion forte** : les éléments d'un même module doivent être **fortement liés** entre eux. On doit penser aux éléments d'un module comme les pièces d'un système unique qui coopèrent pour atteindre un objectif commun. La cohésion forte **facilite l'utilisation** et la **compréhension des modules**.

Les modules

- **Deux propriétés importantes** :
  - **Couplage faible** : les modules doivent être le plus possible **indépendants** les uns des autres. Le couplage faible **facilite la maintenabilité** et la **réutilisation des modules**.

Les modules

- **Catégories de modules** :
  - Un **flaque de données** (***Data Pool***) : le module est une **simple collection de constantes et/ou de variables**. En C, un module de cette catégorie est souvent un fichier d'en-tête. Par exemple, « **float.h** » et « **limits.h** » sont deux modules de ce type.

Les modules

- **Catégories de modules** :
  - Une **librairie** : le module est une **collection de fonctions**. Par exemple, « **string.h** » est l'interface d'une librairie pour le traitement des chaînes de caractères.
  - Un **objet abstrait** : le module est une **collection de fonctions** qui **manipulent une structure de données cachée**.

Les modules

- **Catégories de modules** :
  - Un **type abstrait de données** : le module présente un **type de données** dont **l'implémentation est cachée**. Pour un utilisateur qui veut exécuter des opérations sur ces variables, il doit juste **appeler certaines des fonctions** disponibles par le module du type abstrait de données.

Les types de données abstraits

- **Définition** :
  - Un **type abstrait de données** (***TAD***) est un type de données accessible uniquement à travers une interface.
  - On fait référence au programme qui utilise un ***TAD*** comme un **client**, et au programme qui spécifie le ***TAD*** comme une **implémentation**.

Les types de données abstraits

- **Exemple : un TAD gérant des ensembles finis d'entiers**
  - Un ensemble sera codé comme un **pointeur vers une structure** ayant deux membres :
    - Un **entier** nommé "`card`" représentant le nombre des éléments de l'ensemble.
    - Un **tableau d'entiers** nommé "`elements`" et représentant les éléments effectifs de l'ensemble.

Les types de données abstraits

- **Exemple : un TAD gérant des ensembles finis d'entiers**
  - Nous traitons des ensembles de cardinal ≤ `CARD_MAX` (fixée à 100).

Les types de données abstraits

- **Services offerts par le TAD :**
  - Créer un ensemble vide.
  - Créer un ensemble de n éléments à partir d'un tableau a. Une fonction locale permet de vérifier si le tableau a ne contient pas des éléments dupliqués.
  - Tester si un ensemble est vide.
  - Vider un ensemble.
  - Afficher les éléments d'un ensemble.

Les types de données abstraits

- **Services offerts par le TAD :**
  - Calculer le cardinal d'un ensemble.
  - Tester si un élément appartient à un ensemble.
  - Ajouter un élément à un ensemble.
  - Supprimer un élément d'un ensemble.
  - Calculer la place d'un élément dans un ensemble.

Les types de données abstraits

- **Services offerts par le TAD :**
  - Calculer la réunion de deux ensembles.
  - Calculer l'intersection de deux ensembles.
  - Calculer la différence de deux ensembles.
  - Calculer la différence symétrique de deux ensembles.
  - Copier un ensemble sur un autre.

Les types de données abstraits

- **Services offerts par le TAD :**
  - Tester si un ensemble est inclus dans un autre.
  - Tester si deux ensembles sont égaux.

Les types de données abstraits

- **Interface (Fichier setOfInt.h) :**

```
#ifndef SETOFINT_H
#define SETOFINT_H
#define CARD_MAX  100
typedef enum {False, True} BOOL;
typedef struct setOfInt   {
  int elements[CARD_MAX];
  int card;
} setOfInt;
```

Les types de données abstraits

- Interface (suite) :

```
setOfInt *createEmptySet();
setOfInt *createSetFromArray(int a[], int n);
BOOL isEmpty(setOfInt A);
void removeAll(setOfInt *A);
void print(setOfInt *A);
int getCardinal(setOfInt *A);
void insertElem(int e, setOfInt *A);
BOOL belongsTo(int e, setOfInt *A);
void removeElem(int e, setOfInt *A);
```


Les types de données abstraits

- Interface (suite) :

```
int getPlace(int e, setOfInt A);
void getUnion(setOfInt *A, setOfInt *B, setOfInt *C);
void getIntersec(setOfInt *A, setOfInt *B, setOfInt *C);
void getDiff(setOfInt *A, setOfInt *B, setOfInt *C);
void getSymDiff(setOfInt *A, setOfInt *B, setOfInt *C);
void copy(setOfInt *A, setOfInt *B);
BOOL isSubSetOf(setOfInt *A, setOfInt *B);
BOOL isEqualTo(setOfInt *A, setOfInt *B);
#endif
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

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

/* Fonction verify : teste si un tableau a de n entiers est
   convenable pour être un ensemble, c'est-à-dire il ne
   contient aucun élément dupliqué */
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
BOOL verify(int *a, int n)  {
  int i, j;
  for(i = 0; i < n - 1; i++)
      for(j = i + 1; j < n; j++)
          if(a[i] == a[j])
            return False;
  return True;
}
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
setOfInt *createEmptySet()
{
  setOfInt *A;
  A = (setOfInt*) malloc(sizeof(setOfInt));
  A->card = 0;
  return A;
}
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
setOfInt *createSetFromArray(int a[ ], int n)  {
  setOfInt * A;
  A = (setOfInt*) malloc(sizeof(setOfInt));
  A->card = 0;
  if(verify(a, n) == True)  {
    int i;
    for(i = 0; i < n; i++)  A->elements[i] = a[i];
    A->card = n;  }
  return A;
}
```

Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
BOOL isEmpty(setOfInt A) {
  if(A.card == 0)
    return True;
  return False;  }
void removeAll(setOfInt *A)  {
  if(isEmpty(*A) == False)
    A->card = 0;  }
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
void print(setOfInt *A)   {
  printf("{");
  if(isEmpty(*A) == False)   {
    int i;
    for(i = 0; i < A->card - 1; i++)
        printf("%d, ", A->elements[i]);
    printf("%d", A->elements[i]); }
  puts("}");  }
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
int getCardinal(setOfInt *A)
{
  return A->card;
}
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
void insertElem(int e, setOfInt *A)   {
  if(belongsTo(e, A) == False)   {
    if(A->card < CARD_MAX)  {
      A->card++;
      A->elements[A->card - 1] = e;  }
    else
      puts("Impossible d'ajouter un autre element\n");  }
  else
    printf("L'element %d est deja dans l'ensemble\n", e);   }
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
BOOL belongsTo(int e, setOfInt *A)
{
  int i;
  for(i = 0; i < A->card; i++)
      if(A->elements[i] == e)
        return True;
  return False;
}
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
void removeElem(int e, setOfInt *A)  {
  if(isEmpty(*A) == False)  {
    if(belongsTo(e, A))   {
        int i, k = getPlace(e, *A);
        for(i = k; i < A->card - 1; i++)
            A->elements[i] = A->elements[i + 1];
        A->card--;  }
    else printf("L'element %d n'est pas dans l'ensemble\n", e);   }
  else  puts("L'ensemble est deja vide");   }
```

Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
int getPlace(int e, setOfInt A)
{
  int i;

   for(i = 0; i < A.card; i++)
       if(A.elements[i] == e)
         return i;
   return -1; }
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
void getUnion(setOfInt *A, setOfInt *B, setOfInt *C) {
  int i, j;
  C->card = 0;
  if(isEmpty(*A) == False)  {
        for(i = 0; i < A->card; i++)
            insertElem(A->elements[i], C); }
  if(isEmpty(*B) == False)  {
        for(j = 0; j < B->card; j++) {
           if(belongsTo(B->elements[j], C) == False)
             insertElem(B->elements[j], C);  } } }
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
void getIntersec(setOfInt *A, setOfInt *B, setOfInt *C)  {
  int i, j;
  C->card = 0;
  if(isEmpty(*A) == False || isEmpty(*B) == False)
  {
    for(i = 0; i < A->card; i++)
        if(belongsTo(A->elements[i], B) == True)
          insertElem(A->elements[i], C);  }
}
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
void getDiff(setOfInt *A, setOfInt *B, setOfInt *C) {
  int i, j;
  C->card = 0;
  if(isEmpty(*A) == False)
  {
    for(i = 0; i < A->card; i++)
        if(belongsTo(A->elements[i], B) == False)
          insertElem(A->elements[i], C);
  }
}
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
void getSymDiff(setOfInt *A, setOfInt *B, setOfInt *C)
{
  setOfInt E, F;

  E = *createEmptySet();
  F = *createEmptySet();
  getDiff(A, B, &E);
  getDiff(B, A, &F);
  getUnion(&E, &F, C);
}
```


Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
BOOL isSubSetOf(setOfInt *A, setOfInt *B)
{
  int i;

  for(i = 0; i < A->card; i++)
      if(belongsTo(A->elements[i], B) == False)
        return False;
  return True;
}
```

Les types de données abstraits

- Implémentation (Fichier setOfInt.c) :

```
BOOL isEqualTo(setOfInt *A, setOfInt *B) {
  return isSubSetOf(A, B) && isSubSetOf(B, A);
}
void copy(setOfInt *A, setOfInt *B)   {
  int i;
  A->card = B->card;
  for(i = 0; i < B->card; i++)
      A->elements[i] = B->elements[i];  }
```


Les types de données abstraits

- **Un programme utilisant le TAD :**
  - Le programme va :
  1. Créer un ensemble « A » formé des entiers pairs inférieurs à 10.
  2. Créer un ensemble « B » contenant les entiers impairs inférieurs à 10.
  3. Afficher A et B.
  4. Calculer C = A ∪ B, puis l'afficher.

Les types de données abstraits

- **Un programme utilisant le TAD :**

```
#include "setOfInt.h"
main() {
  setOfInt A, B, C;
  int a[5] = {0, 2, 4, 6, 8};  int b[5] = {1, 3, 5, 7, 9};
  A = *createSetFromArray(a, 5);
  B = *createSetFromArray(b, 5);
  print(&A);   print(&B);
  C = *createEmptySet();
  getUnion(&A, &B, &C);
  print(&C);
  system("pause");   }
```


Analyse d'un algorithme

- **Rappel de la notion d'algorithmes :**
  - Un **algorithme** est une **séquence finie d'étapes** (instructions) pour résoudre un problème.
  - S'il est vu comme fonction, un **algorithme est un processus qui transforme des données en entrée** et **produit des données en sortie**.
  - Un même problème peut être résolu par des algorithmes différents.

Analyse d'un algorithme

- **Principes des algorithmes :**
  - **Principe de correction** : un algorithme doit être correct (il fournit toujours la sortie valide pour n'importe quelle donnée en entrée).
  - **Principe d'effectivité** : un algorithme doit être composé d'une série d'étapes concrètes ("bien spécifiées", "compréhensibles" et "exécutables" par la machine qui va l'utiliser).

Analyse d'un algorithme

- **Principes des algorithmes :**
  - **Principe de finitude** : un algorithme doit avoir un nombre fini d'étapes de calcul.
  - **Principe de terminaison** : un algorithme doit toujours se terminer quelque soit les données en entrée.

Analyse d'un algorithme

- **Performance d'une algorithme :**
  - Un algorithme, lorsqu'il est exécuté, va utiliser des **ressources** :
    - La **mémoire centrale**.
    - Le **processeur**.
    - Les **disques**.
    - La **bande passante d'un réseau**.
    - ...etc.

Analyse d'un algorithme

- **Performance d'une algorithme :**
  - La **performance** (**efficacité**) d'un algorithme est **sa capacité à optimiser les ressources**.
  - Typiquement, la ressource critique pour un algorithme est son **temps d'exécution**.
  - Un algorithme est d'autant plus efficace si son temps d'exécution est plus rapide.

Analyse d'un algorithme

- **Performance d'une algorithme :**
  - Il est possible de **mesurer expérimentalement** le **temps d'exécution d'un algorithme**.
  - En C, on peut utiliser la bibliothèque « **time.h** » pour cet effet.

Analyse d'un algorithme

- **Calcul du temps d'exécution d'un algorithme en C :**

```
#include <time.h>
// ……..
float start, finish, elapsed;
start = clock(); /* temps de début */
// Appel de l'algorithme
finish = clock(); /* temps de fin */
elapsed = (finish - start) / CLOKS_PER_SEC;
```


Analyse d'un algorithme

- **Exemple :**

```
#include <time.h>
#include <stdio.h>
int fibo(int n)
{
  if(n <= 1)
    return 1;
  else
    return fibo(n - 1) + fibo(n - 2);
}
```


Analyse d'un algorithme

- **Exemple (suite) :**

```
main( )  {
  float t1, t2, t;
  t1 = clock();
  fibo(45);
  t2 = clock();
  t = (t2 – t1) / CLOKS_PER_SEC;
  printf("temps d'execution de fibo(45) : %.3f\n", t);
}
```

Analyse d'un algorithme

- **Inconvénients de la méthode expérimentale :**
  - Le temps d'exécution expérimental calculé **dépend d'autres facteurs** :
    - La **machine** utilisée.
    - Le **système d'exploitation** (OS) de la machine.
    - Le **compilateur** (le langage) utilisé pour coder l'algorithme.
    - ...etc.

Analyse d'un algorithme

- **Inconvénients de la méthode expérimentale**
  - Les temps d'exécutons expérimentaux de deux algorithmes sont **difficiles à comparer**, sauf si les mesures ont été effectuées dans les mêmes conditions.
  - Les expérimentations ne peuvent être effectuées que sur un **sous-ensemble fini et limité de valeurs d'entrée** (**résultats non généraux**).

Analyse d'un algorithme

- **Efficacité théorique d'un algorithme :**
  - On cherche une méthode **facile à comprendre** et à **utiliser** pour l'analyse de la performance d'un algorithme, et qui doit :
    - Permettre **l'évaluation des efficacités relatives** de deux ou plusieurs algorithmes **indépendamment de l'environnement** (matériel et logiciel).

Analyse d'un algorithme

- **Efficacité théorique d'un algorithme :**
  - Être **basée sur l'étude de la description de haut niveau de l'algorithme** sans aucun besoin à son implémentation.
  - Prendre en compte toutes les **valeurs d'entrée possibles**.

Analyse d'un algorithme

- **Efficacité théorique d'un algorithme :**
  - Cette approche s'appelle **l'analyse asymptotique de la complexité d'algorithmes**.
  - Le but de cette analyse est d'établir des **résultats plus généraux** permettant d'exprimer **l'efficacité intrinsèque** de la stratégie utilisée par un algorithme, indépendamment de son environnement d'exécution (matériel et logiciel).

Analyse d'un algorithme

- **Efficacité théorique d'un algorithme :**
  - Le type de résultat que l'on souhaite avoir est par exemple :

*"Sur toute machine, et quel que soit le langage de programmation, **l'algorithme A1 sera meilleur que l'algorithme A2** pour les données de grande taille".*

*"**L'algorithme A est optimal** en **nombre de comparaisons** pour résoudre le problème P".*

Analyse d'un algorithme

- **Calcul de l'efficacité théorique d'un algorithme :**
  - **Étape 1** : **Définir la taille d'une entrée de l'algorithme**.
  - Cette quantité dépend d'un algorithme à un autre.
  - La comparaison des algorithmes se fait donc par rapport à la taille de l'entrée.

Analyse d'un algorithme

- **Calcul de l'efficacité théorique d'un algorithme :**
  - Pour un **algorithme de tri**, une bonne mesure de la taille est le **nombre d'éléments à trier**.
  - Pour un **algorithme** qui **résout un système de n équations linéaires à n inconnus**, il est normal de prendre **n** pour la taille du problème.

Analyse d'un algorithme

- **Calcul de l'efficacité théorique d'un algorithme :**
  - D'autres **algorithmes** pourraient utiliser la **valeur d'une entrée particulière**, ou la **longueur d'une liste** en entrée du programme, ou la **taille d'un tableau** en entrée, ou une **combinaison de ces quantités**.

Analyse d'un algorithme

- **Calcul de l'efficacité théorique d'un algorithme :**
  - Il est pratique de considérer une fonction à valeurs positives **T(n)** pour représenter le **nombre d'unités de temps prises par un algorithme** sur n'importe quelle **entrée de taille n**.
  - Nous appellerons alors **T(n)** le **temps d'exécution théorique de l'algorithme**.

Analyse d'un algorithme

- **Calcul de l'efficacité théorique d'un algorithme :**
  - **2ème étape** : **Déterminer les opérations fondamentales (OF) de l'algorithme**.
  - Ces opérations sont qualifiées de **fondamentales**, car enfin de compte, le **temps d'exécution théorique de l'algorithme** sera **proportionnel à leur nombre**.

Analyse d'un algorithme

- **Exemples d'OF :**
  - Pour un algorithme de **recherche interne**, le **nombre de comparaisons** entre l'élément recherché et les autres éléments en mémoire.
  - Pour un algorithme de **recherche externe**, le **nombre d'accès à la mémoire** secondaire.

Analyse d'un algorithme

- **Exemples d'OF :**
  - Pour un algorithme de **tri interne d'une liste**, on peut considérer le **nombre de comparaisons** entre éléments et le **nombre de déplacements** d'éléments.
  - Pour un algorithme qui **multiplier deux matrices**, le **nombre de multiplications** et le **nombre d'additions**.

Analyse d'un algorithme

- **Calcul de l'efficacité théorique d'un algorithme :**
  - **3ème étape** : **Compter le nombre de ces opérations fondamentales**.

Analyse d'un algorithme

- **Règles pour compter le nombre des OF :**
  - **Séquence** : les NOF s'ajoutent.
  - **Branchements conditionnels** : on peut majorer le NOF. Si OF(X) désigne le NOF d'une construction X, alors :

$$OF(\text{si C alors A sinon B}) \leq OF(C) + \max(OF(A), OF(B))$$

Pour la séquence :

$$OF(X; Y) = OF(X) + OF(Y)$$

Analyse d'un algorithme

- **Règles pour compter le nombre des OF :**
  - **Boucles** : $\sum_i OF(i)$, où i est l'incrément de la boucle et OF(i) est le NOF lors de l'exécution de la $i^{ème}$ itération de la boucle.
  - **Procédures récursives** : le calcul du NOF donne en général lieu à la résolution de certaines équations de récurrence. Par exemple, le nombre des multiplications dans l'algorithme de la factorielle est :

Analyse d'un algorithme

$M(n) = 0$, si $n \leq 1$

$M(n) = M(n - 1) + 1$, si $n > 1$

Analyse d'un algorithme

- **Calcul de l'efficacité théorique d'un algorithme :**
  - **4ème étape** : **Choisir quelle classe de complexité**.

Analyse d'un algorithme

- **Calcul de l'efficacité théorique d'un algorithme :**
  - Il y a trois catégories de complexités théoriques :
    - La **complexité dans le pire des cas**
    - La **complexité dans le meilleur des cas**
    - La **complexité en moyenne**

Analyse d'un algorithme

- **Notations :**
  - Soit A un algorithme qui résout un problème P.
  - On note par n, la **taille d'une entrée** de l'algorithme A.
  - On définit l'ensemble :

    D_n = {d : donnée possible de taille n}.
  - $T_A(d)$: **complexité théorique en temps** de l'algorithme A, lorsqu'il traite la donnée d.

Analyse d'un algorithme

- **Complexité dans le meilleur des cas :**
  - $T_{min}(n) = Inf \{T_A(d) \mid d \in D_n\}$.
- **Complexité dans le pire des cas :**
  - $T_{max}(n) = Sup \{T_A(d) \mid d \in D_n\}$.
- **Complexité en moyenne :**
  - $T_{moy}(n) = \sum_{d \in Dn} p(d) \times T_A(d)$.

  Où p(d) est la probabilité que l'on ait la donnée d en entrée de l'algorithme A.

Analyse d'un algorithme

- **Complexité dans le pire des cas :**
  - Elle donne une borne supérieure du temps d'exécution de l'algorithme A.
  - Elle permet de donner une estimation de la taille maximale des données qui pourront être traitées par l'algorithme (en un temps raisonnable).

Analyse d'un algorithme

- **Complexité dans le meilleur des cas :**
  - Elle donne une borne inférieure du temps d'exécution de l'algorithme A.
- **Complexité dans le meilleur des cas :**
  - Elle donne une estimation du temps d'exécution moyen de l'algorithme A.
  - Elle est souvent beaucoup plus difficile à calculer !

Les notations asymptotiques

- **Comparaison de deux algorithmes :**
  - On a introduit une **mesure de la complexité d'un algorithme** comme une **fonction de la taille des données** : **T(n)**.
  - Il est donc très intéressant de connaître la **rapidité de croissance** de cette fonction lorsque la taille des données augmente.

Les notations asymptotiques

- **Comparaison de deux algorithmes :**
  - En effet, pour traiter un **problème de petite taille**, la **méthode utilisée importe peu**.
  - Cependant, pour un **problème de grande taille**, les **différences de performance entre algorithmes peuvent être énormes**.

Les notations asymptotiques

- **Comparaison de deux algorithmes :**
  - Souvent, une **simple approximation** de la fonction **T(n)** suffit pour savoir si un algorithme est **utilisable ou non**, ou pour **comparer deux différents algorithmes** traitant le même problème.

Les notations asymptotiques

- **Comparaison de deux algorithmes :**
  - Par exemple, pour n très grand, il est secondaire de savoir si un algorithme fait (n + 1) ou (n + 2) opérations : les **constantes additives sont négligeables**.
  - Parfois, même les **constantes multiplicatives** ont peu d'importance.

Les notations asymptotiques

- **Comparaison de deux algorithmes :**
  - Par exemple, si l'on a un algorithme A_1 de complexité $T_1(n) = n^2$ et un autre algorithme A_2 de complexité $T_2(n) = 2n$. A2 sera meilleur que A1 pour presque tous les n (n > 1).
  - De même, si $T_1(n) = 3n^2$ et $T_2(n) = 25n$. A2 est meilleur que A1 pour n > 8.

Les notations asymptotiques

- **Comparaison de deux algorithmes :**
  - Quelles que soient les constantes multiplicatives, l'algorithme A_2 sera toujours meilleur à partir d'un certain rand n, car la fonction $f(n) = n^2$ croit plus vite que la fonction $g(n) = n$:

$$\lim_{n \to \infty} f(n) / g(n) = 0$$

  - On dit que **l'ordre de grandeur asymptotique** de f(n) est **strictement plus grand** que celui de g(n).

Les notations asymptotiques

- **Grand O :**
  - On dit que g(n) est une **borne asymptotique supérieure** de f(n), s'il existe une constante C > 0 et un rang n_0 telles que :

$$f(n) \leq C \times g(n), \text{ pour tout } n \geq n_0.$$

  - On écrit : f(n) = O(g(n)).

Les notations asymptotiques

- **Exemples :**
  - $f(n) = 7n^2 + 5n + 3 = O(n^2)$
  - $f(n) = \sin(n) + 2 = O(1)$
  - $f(n) = n + 9 = O(n)$

Les notations asymptotiques

- **Grand O :**
  - On dit que g(n) est une **borne asymptotique inférieure** de f(n), s'il existe une constante $C > 0$ et un rang n_0 telles que :

    $f(n) \geq C \times g(n)$, pour tout $n \geq n_0$.
  - On écrit : $f(n) = \Omega(g(n))$.

Les notations asymptotiques

- **Exemples :**
  - $f(n) = 7n^2 + 5n + 3 = \Omega(n^2)$
  - $f(n) = 7n^2 + 5n + 3 = \Omega(n)$
  - $f(n) = n + 9 = \Omega(n)$

Les notations asymptotiques

- **Grand O :**
  - On dit que g(n) est une **borne asymptotique** de f(n), s'il existe deux constantes $C_1 > 0$, $C_2 > 0$ et un rang n_0 telles que :

    $C_1 \times g(n) \leq f(n) \leq C_2 \times g(n)$, pour tout $n \geq n_0$.
  - On écrit : $f(n) = \Theta(g(n))$.

Les notations asymptotiques

- **Exemples :**
  - $f(n) = 7n^2 + 5n + 3 = \Theta(n^2)$
  - $f(n) = 7n \times \log(n) + 5n + 3 = \Theta(n \times \log(n))$
  - $f(n) = n + 9 = \Theta(n)$

Les notations asymptotiques

- **Propriétés :**
  - **Transitivité** de O, Ω et Θ :
    - Si $f(n) = O(g(n))$ et $g(n) = O(h(n))$, alors $f(n) = O(h(n))$
    - Si $f(n) = \Omega(g(n))$ et $g(n) = \Omega(h(n))$, alors $f(n) = \Omega(h(n))$
    - Si $f(n) = \Theta(g(n))$ et $g(n) = \Theta(h(n))$, alors $f(n) = \Theta(h(n))$

Les notations asymptotiques

- **Propriétés :**
  - **Réflexivité** de O, Ω et Θ :
    - f(n) = O(f(n))
    - f(n) = Ω(f(n))
    - f(n) = Θ(f(n))
  - **Symétrie** de Θ :
    - f(n) = Θ(g(n)) ⇔ g(n) = Θ(f(n))

Les notations asymptotiques

- **Propriétés :**
  - **Symétrie transposée** de O et Ω :
    - f(n) = O(g(n)) ⇔ g(n) = Ω(f(n))
  - **Définition** de Θ :
    - f(n) = Θ(g(n)) ⇔ f(n) = O(g(n)) et f(n) = Ω(g(n))
  - **Loi du Max :**
    - Max(f(n), g(n)) = Θ(f(n) + g(n))

Les notations asymptotiques

- **Fonctions usuelles :**
  - **Parties entières** :
    - $\lfloor n \rfloor = \Theta(n)$
    - $\lceil n \rceil = \Theta(n)$
  - **Polynômes** :
    - $f(n) = a_d n^d + \ldots + a_1 n + a_0 = \Theta(n^d)$

Les notations asymptotiques

- **Fonctions usuelles :**
  - Toute fonction **poly-logarithmique** croit moins vite que toute fonction **polynomiale**.
  - Toute fonction **polynomiale** croit moins vite que toute fonction **exponentielle**.
  - Toute fonction **exponentielle** croit moins vite que toute fonction **factorielle**.

Les notations asymptotiques

- **Le théorème général :**
  - Soient a > 0, b > 1 et d ≥ 0 trois constantes telles que :

    $$T(n) = a \times T(n / b) + O(n^d)$$
  - Alors :
    - $T(n) = O(n^d)$, si $d > \log_b(a)$
    - $T(n) = O(n^d \times \log(n))$, si $d = \log_b(a)$
    - $T(n) = O(n^{\log(a)/\log(b)})$, si $d < \log_b(a)$

Les notations asymptotiques

- **Applications du théorème général :**
  - T(n) = 2T(n/3) + 1.
    - $T(n) = O(n^{\log(2)/\log(3)}) = O(n^{0.63093})$
  - $T(n) = 9T(n/3) + n^3$
    - $T(n) = O(n^3)$
  - $T(n) = 8T(n/2) + n^3$
    - $T(n) = O(n^3 \times \log(n))$

Classification des algorithmes

- **Les 7 fonctions fondamentales :**
  - **Constante** : $\Theta(1)$
  - **Logarithmique** : $\Theta(\log(n))$
  - **Linéaire** : $\Theta(n)$
  - **Quasi-linéaire** : $\Theta(n \times \log(n))$
  - **Carrée** : $\Theta(n^2)$
  - **Cubique** : $\Theta(n^3)$

  Polynomiale : $\Theta(n^d)$
  - **Exponentiel** : $\Theta(a^n)$, avec a > 1

Classification des algorithmes

- **Algorithme constant :**
  - Un algorithme qui effectue un **nombre constant d'opérations de base**.
- **Exemples** :
  - Échange (permutation) de deux variables
  - Addition de deux nombres
  - Minimum ou maximum de deux ou trois nombres.

Classification des algorithmes

- **Exemples** :

```
int sumToN(int n)  {
  return n * (n + 1) / 2;
}
void exchange(int *x, int *y)
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```


Classification des algorithmes

- **Algorithme logarithmique :**
  - Un algorithme qui effectue un **nombre logarithmique d'opérations de base**.
  - Typiquement, c'est un algorithme qui résout un problème en divisant sa taille par une quantité constante.

Classification des algorithmes

- **Exemples** :
  - Recherche binaire d'un élément dans un tableau trié.
  - Détermination du nombre de chiffres d'un nombre entier dans une base donnée.

Classification des algorithmes

- **Exemple** :

```
int numberOfDigits(int n)  {
  int k = 0, b = 1;
  while(b < n)
  {
    b = b * 2;
    k++;
  }
  return k + 1;
}
```

Classification des algorithmes

- **Algorithme linéaire :**
  - Un algorithme qui effectue un **nombre d'opérations de base proportionnelle à la taille de son entrée**.
  - Typiquement, c'est un algorithme qui effectue un temps constant pour chaque donnée élémentaire d'une collection de n données.

Classification des algorithmes

- **Exemples** :
  - Recherche d'un élément dans un tableau non trié.
  - Calcul de la somme des éléments d'un tableau.
  - Calcul du plus petit élément d'une collection.

Classification des algorithmes

- **Exemple** (1) :

```
int linearSearch(int x, int a[ ], int n)  {
  int i;
  for(i = 0; i < n; i++)
      if(a[i] == x)
        return 1;
  return 0;
}
```


Classification des algorithmes

- **Exemple (2)** :

```
int sumToN(int n)
{
  if(n == 0)
    return 0;
  return sumToN(n - 1) + n;
}
```


Classification des algorithmes

- **Algorithme quasi-linéaire :**
  - Un algorithme qui effectue un **nombre d'opérations de base proportionnelle à n × log(n)**, où n est la taille de son entrée.
  - Typiquement, c'est un algorithme qui résout un problème de taille n en le divisant en deux sous-problèmes, chacun de taille n/2, puis trouve la solution générale en combinant les solutions partielles en un temps linéaire.

Classification des algorithmes

- **Exemples :**
  - Tri rapide (***quick sort***).
  - Tri par fusion (***merge sort***).

Classification des algorithmes

- **Exemple** :

```
void quickSort(int a[ ], int low, int high)  {
  int middle;

  if(low < high)
  {
     middle = split(a, low, high);
     quickSort(a, low, middle - 1);
     quickSort(a, middle, high);
  }
}
```


Classification des algorithmes

- **Exemple (suite)** :

```
int split(int a[ ], int low, int high)  {
  int part_element = a[low];
  for(;;)  {
     while(low < high && part_element <= a[high])
           high--;
     if(low >= high)   break;
     a[low++] = a[high];
```


Classification des algorithmes

- **Exemple (suite)** :

```
     while(low < high && part_element >= a[low])
           low++;
     if(low >= high)
       break;
     a[high--] = a[low];
  }
  a[high] = part_element;
  return high;
}
```


Classification des algorithmes

- **Algorithme quadratique :**
  - Un algorithme qui effectue un **nombre d'opérations de base proportionnelle à n^2**, où n est la taille de son entrée.
  - Typiquement, c'est un algorithme avec deux boucles imbriquées ou un algorithme qui effectue un traitement sur tous les couples d'une collection de n éléments.

Classification des algorithmes

- **Exemples :**
  - Tri par sélection (***selction sort***).
  - Tri par insertion (***insertion sort***).
  - Tri bulles (***Bubble sort***).
  - Test de symétrie d'une matrice carrée.

Classification des algorithmes

- **Exemple** :

```
void selectionSort(int a[ ], int n)  {
  int i, imin;
  for(i = 0; i <= n - 2; i++)
  {
    imin = i;
    for(j = i + 1; j < n; j++)
    {
      if(a[i] < a[imin])
        imin = i;
    }
```

Classification des algorithmes

- **Exemple** :

```
if(imin != i)  {
   int temp = a[imin];
   a[imin] = a[i];
   a[i] = temp;
  }
 }
}
```


Classification des algorithmes

- **Algorithme cubique :**
  - Un algorithme qui effectue un **nombre d'opérations de base proportionnelle à n^3**, où n est la taille de son entrée.
  - Typiquement, c'est un algorithme avec trois boucles imbriquées.
- **Exemple** :
  - Multiplication de deux matrices carrées d'ordre n.

Classification des algorithmes

- **Exemple** :

```
void matrixProduct(int n, int a[n][n], int b[n][n],
                   int c[n][n])  {
  int i, j, k;
  for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
       c[i][j] = 0;
       for(k = 0; k < n; k++)
          c[i][j] = c[i][j] + a[i][k] * b[k][j];  }
}
}
```


Classification des algorithmes

- **Algorithme exponentiel :**
  - Un algorithme qui effectue un **nombre d'opérations de base proportionnelle à a^n**, où n est la taille de son entrée et a > 1.
- **Exemple** :
  - L'algorithme récursive qui calcule la suite de Fibonacci.

Classification des algorithmes

- **Exemple** :

```
int fibo(int n)
{
  if(n == 0 || n == 1)
    return 1;
  return fibo(n - 1) + fibo(n - 2);
}
```


Classification des algorithmes

- **Exemple** :

```
int powerOf2(int n)
{
  if(n == 0)
    return 1;
  return powerOf2(n - 1) + powerOf2(n - 1);
}
```