

INF 1241 - Algorithmique et programmation 2

LGI – Semestre 2

Département informatique

UFR des Sciences et technologies

Université de Thiès

Overview

1. PRÉSENTATION DU COURS
2. TYPES DE DONNEES COMPOSEES
3. POINTEURS
4. SOUS PROGRAMMES
5. RECURSIVITE
6. ALGORITHMES DE TRI
7. FICHIERS

Overview

1. PRÉSENTATION DU COURS
2. TYPES DE DONNEES COMPOSEES
3. POINTEURS
4. SOUS PROGRAMMES
5. RECURSIVITE
6. ALGORITHMES DE TRI
7. FICHIERS

Chapitre 1

PRESENTATION DU COURS

Sub-Overview

1. Le cours
2. Les objectifs
3. Les Prérequis
4. Les références

Vocabulaires du cours

- **Unité d'Enseignement**
 - **Titre** : INFORMATIQUE
 - **Sigle** : INF 124
- **Élément constitutif**
 - **Titre** : Algorithmique et programmation 2
 - **Sigle** : INF 1241
- **Autres éléments constitutifs de l'UE (1)**
 - **Titre** : Architecture des ordinateurs
 - **Sigle** : INF 1242

Volume horaire & évaluation

- **CM : 25H**
- **TD/TP : 25H**
- **TPE : 50H**
- **Coefficient de l'UE : 4**
- **Crédits de l'UE : 10**
- **Evaluation**
 - **Contrôle des connaissances : 40%**
 - **Examen écrit : 60%**

Responsables

- **Magistral**

Pr Mouhamadou THIAM

Maître de conférences en Informatique

Intelligence Artificielle : Sémantique Web

Email : mthiam@univ-thies.sn

- **Travaux dirigés et pratiques**

M. Papa DIOP

Ingénieur Systèmes Informatiques et Bases de Données

Diplômé de l'Université Gaston Berger de Saint-Louis

Email : papaddiop@gmail.com

Objectifs

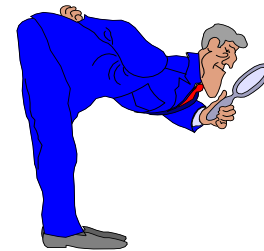
- Connaître les notions d'enchaînement, de test, de boucle
- Connaître les types de données composées
- Connaître les notions de pointeurs
- Connaître les notions de sous-programmes (fonction, procédure)

Objectifs (suite)

- Connaître les notions de récursivité
- Connaître les algorithmes de tris standards
- Pouvoir manipuler correctement les fichiers
- Savoir appliquer les concepts de base ci-dessus à la programmation en langage C

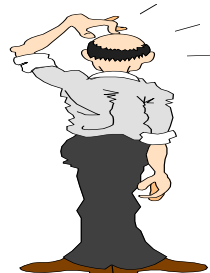
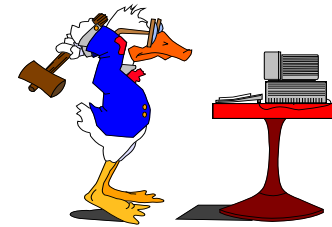
Prérequis : résolution de problèmes

- **Lire** l'énoncé du problème, être certain de bien le comprendre
 - utiliser une loupe
 - ressortir les informations pertinentes
 - données ? résultats ?



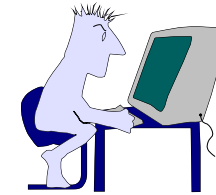
Prérequis : résolution de problèmes (suite)

- **Réfléchir** à la résolution du problème en ignorant l'existence de l'ordinateur
 - déterminer les points principaux à traiter
 - exploiter l'ensemble de vos connaissances
 - adapter ou réutiliser des recettes existantes
 - encore difficile ?
- Décomposer le problème!! **Analyse descendante!!**



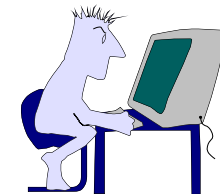
Prérequis : méthodologie

- **Écrire** formellement la solution (algorithme)
sur papier
 - utiliser un pseudo langage
- **Vérifier** votre solution sur un exemple
 - preuve formelle de l'algorithme : encore mieux !!



Prérequis : méthodologie (fin)

- **Traduire** dans un langage de programmation
- **Tester** le programme sur différents jeux de tests



- le jeu de tests doit être « suffisant »
- ne pas oublier les cas particuliers, ...

From scratch ...

Résoudre un problème

Indiquer où vous habitez



Décrire le cheminement permettant d'arriver à la solution d'un problème décrit par un énoncé

Décrire le cheminement à emprunter pour arriver à votre habitation telle que décrit par votre adresse

Quelques problèmes

- Un nombre N est-il divisible par 4 ?

- Soit une série de nombre, trier ces nombres

- Soit le problème classique de la tour de Hanoi, afficher la liste des mouvements nécessaires pour le résoudre.

- Un voyageur de commerce désire faire sa tournée, existe-t-il une tournée de moins de 50 km ?

David Hilbert & son problème n°10

- 1862 - 1943
- Liste des 23 problèmes de Hilbert (1900)
- Problème numéro 10 :
« Trouver un algorithme déterminant si une équation diophantienne à des solutions »



Équation diophantienne

- Une équation diophantienne, en mathématiques, est une équation dont les **coefficients** sont des **nombre entiers** et dont les solutions recherchées sont également **entières**. Le terme est aussi utilisé pour les équations à coefficients **rationnels**. Les questions de cette nature entrent dans une branche des mathématiques appelée **arithmétique**.

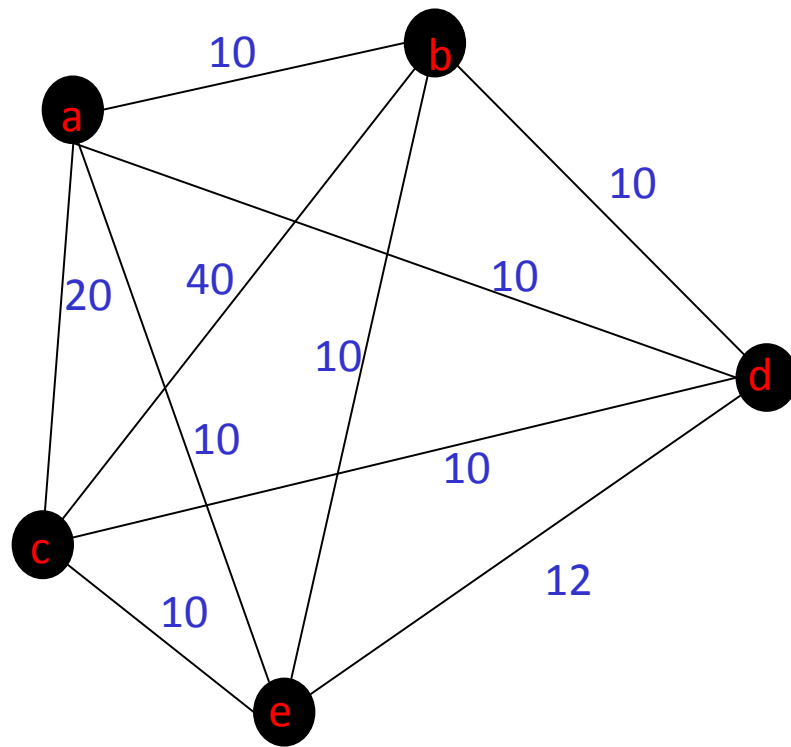
Équation diophantienne

- Carl Friedrich Gauss, un mathématicien du XIX^e siècle, disait : « Leur charme particulier vient de la simplicité des énoncés jointe à la difficulté des preuves »
- Identité de **Bézout** : $a x + b y = c$
- Théorème de **Wilson** : $(x - 1)! + 1 = y x$
- Triplet **pythagoricien** : $x^2 + y^2 = z^2$
- Last **Fermat** theorem ($n=4$) : $x^4 + y^4 = z^4$

Exemple de problème

Le voyageur de commerce

- Un voyageur de commerce désire faire sa tournée, existe-t-il une tournée de moins de 50 km ?



a-b-e-c-d-a

50km

Algorithmique (1)

- L'**algorithmique** est la science des **algorithmes**, visant à étudier les opérations nécessaires à la réalisation d'un calcul.

René Descartes dans le Discours de la Méthode :

- « *diviser chacune des difficultés que j'examinerois, en autant de parcelles qu'il se pourroit, et qu'il seroit requis pour les mieux résoudre.* ».

Algorithmique (2)

- Un **algorithme** est une méthode de résolution de problème énoncée sous la forme d'une série d'opérations à effectuer.
- La mise en œuvre de l'algorithme consiste en l'écriture de ces opérations dans un langage de programmation et constitue alors la brique de base d'un programme informatique (implémentation, « codage »)
- L'algorithme devra être plus ou moins détaillé selon le niveau d'abstraction du langage utilisé ; autrement dit, une recette de cuisine doit être plus ou moins détaillée en fonction de l'expérience du cuisinier.

Langage de programmation

- « Un langage de programmation est une *convention* pour donner des *ordres* à un *ordinateur*. Ce n'est pas censé être *obscur*, *bizarre* et plein de *pièges* subtils. Ca, ce sont les caractéristiques de la *magie*. »

Dave Small



Quelques algorithmes

- « *résous le problème* » du super chammmmpion
- « *trouve le chemin tout seul* » au touriste
- « *débrouillez vous pour que ça marche* »
notice logiciel
- les ordinateurs ont le bon goût d'être tous
strictement aussi **idiots** les uns que les autres.

Matheux pour être bon en algo ?

- Faut-il être « **bon en maths** » pour expliquer correctement son chemin à quelqu'un ?
 - Intuitif
 - méthodique et rigoureux
- « *Si on ment à un compilateur, il prendra sa revanche.* » - **Henry Spencer.**

Références

- [1] Web
- [2] Aho et al. *Structures de donnees et algorithmes*, *Addisson-Wesley / InterEditions*. 1989.
- [3] Aho et Ullman. *Concepts fondamentaux de l'informatique*, Dunod. 1993.
- [4] Sedgewick. *Algorithmes en C*. Addisson-Wesley / InterEditions. 1991.
- [5] Jacques Courtin, Irène Kowarski, « *Initiation à l'algorithmique et aux structures de données* », volume 1 et 2, Dunod
- [6] Guy Chaty, Jean Vicard, « *Programmation : cours et exercices* », Ellipses

Overview

1. PRÉSENTATION DU COURS
2. TYPES DE DONNEES COMPOSEES
3. POINTEURS
4. SOUS PROGRAMMES
5. RECURSIVITE
6. ALGORITHMES DE TRI
7. FICHIERS

Chapitre 2

TYPES DE DONNEES COMPOSEES

Sub-Overview

1. Enumérations
2. Tableaux
3. Chaîne
4. Structures
5. Compléments

Introduction

- A partir des types de base
 - caractère,
 - entier,
 - Réel
- créer de nouveaux types, appelés *types composés*,
- permettre de représenter des ensembles de données organisés.

ENUMERATIONS

Pourquoi les énumérations?

- L'utilisation de valeurs alphanumériques (lettres ou chiffres) est courante pour coder de l'information, par exemple :
 - 'F' pour féminin, 'M' pour masculin,
 - **note[0]** pour la note d'examen intermédiaire
 - **note[1]** pour la note d'examen final
 - **note[2]** pour la note du premier travail pratique
 - ...
 - **note[5]** pour la note globale des travaux pratiques
 - **note[6]** pour la note globale du cours

Cependant ...

- Le **C** permet de donner plus de clarté au codage de l'information en utilisant le type énumération :
 - 1. `char poste = Programmeur ;`
est plus clair que `Poste = 'P' ;` (est-il **polygame?**)
 - 2. `note[tps] ;`
est plus significative que `note[5]`

Définition

- Une énumération est un type permettant de définir un ensemble de constantes, parmi lesquelles les variables de ce type prendront leur valeur.
- Pour déclarer une variable de type énuméré, il faut d'abord créer le type.

En algorithmique

- type

$\text{nom_type} = \{\text{constante}_1, \text{constante}_2, \dots, \text{constante}_N\}$

- nom_type = identificateur du nouveau type
- $\text{constante}_1, \text{constante}_2, \dots, \text{constante}_N$ = liste identificateurs donnant l'ensemble des valeurs de ce type.

Exemple

- **type**

// définition du type couleur

couleur = {*bleu, blanc, rouge, vert, jaune, noir*}

// définition du type jour

jour = {*lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche*}

Déclaration de variables

- Définition type énuméré → utiliser comme un type normal
- Déclarer une ou plusieurs variables de ce type
- **Exemple :**

...

variable

c : couleur // déclaration de la variable c de type couleur

Début

...

c ← bleu // utilisation de la variable c

Fin

En langage C

- Définition type énuméré
 - mot-clé **enum**

- Syntaxe

enum nom_type {*constante₁*, *constante₂*, ..., *constante_N*};

Exemples

- **enum** *jours* {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche};
- *jours* **jour**; /*la *variable* jour est de type jours */
- **enum** *couleur* {bleu, blanc, rouge, vert, jaune, noir} ;

Déclaration de variables

- Après avoir défini un type énuméré, on peut l'utiliser pour déclarer une ou plusieurs variables de ce type.
- **Exemples:**
 - `enum jour j1;`
 - `enum jour j2 = Mardi;`

Compréhension

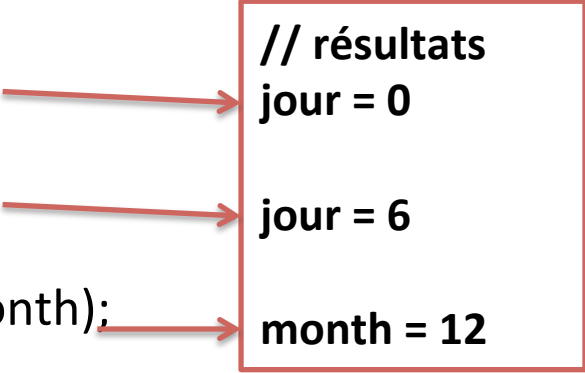
- Instances de la liste d'énumération considérées comme des **constantes entières**
- Valeurs sont numérotées à partir de **zéro** par défaut, avec un **pas de 1**.
- Exemple précédent
 - lundi = 0; mardi = 1;...

Compréhension (suite)

- Instruction
 - jour = mardi;
- Équivalente à l'instruction
 - jour = 1;
- Initialisation *explicite* du 1^{er} élément de la liste avec valeur entière \neq de zéro autorisée
 - **Exemple** : *enum* mois { Janvier = 1, Février, Mars = 10, ..., Décembre };

Exemple complet

```
int main (void){
    enum jours {lundi, mardi, mercredi, jeudi,vendredi, samedi, dimanche};
    enum mois {Janvier = 1, Février, Mars, Avril, Mai, Juin, Juillet, Août,
        Septembre, Octobre, Novembre, Décembre} month;
    enum jours jour;
    jour = lundi;
    printf ("jour = %d\n",jour);
    jour = dimanche;
    printf ("jour = %d\n",jour);
    month = Décembre;
    printf ("month = %d\n",month);
}
```



// résultats
jour = 0
jour = 6
month = 12

Remarques

- Les valeurs associées aux constantes ne sont pas modifiables dans le programme.
- Après avoir défini un type énuméré, on peut l'utiliser pour déclarer une ou plusieurs variables de ce type

Remarques (fin)

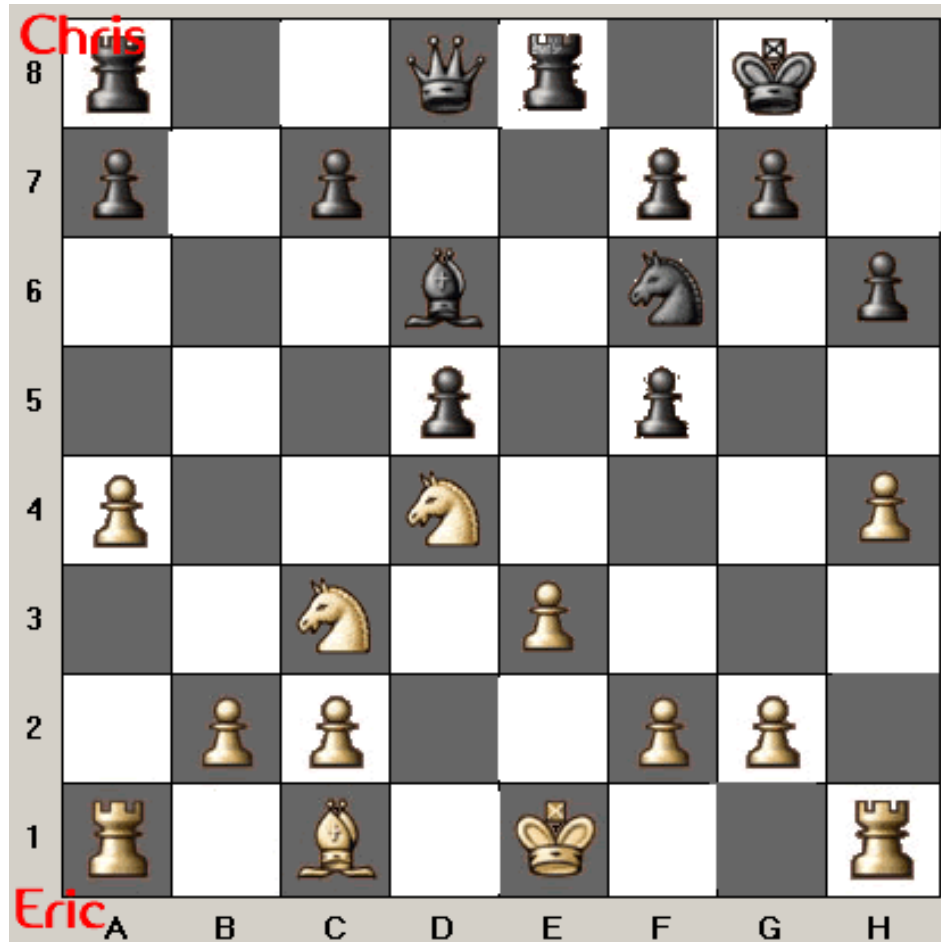
- Chaque constante est associée à un entier
 - son numéro d'ordre (partant de 0 pour le 1^{er})
 - on peut aussi spécifier la valeur associée.
- **Exemple:**
 - **enum jour** {**Lundi=1**, Mardi, **Mercredi=5**, Jeudi, Vendredi, Samedi, Dimanche};

Exercice : donnez le résultat

```
enum Jour {lundi=1,mardi,mercredi,jeudi,vendredi,samedi,dimanche};
int main() {
    enum Jour un_jour;
    printf("Donnez un numéro de jour entre 1 et 7") ;
    scanf ("%d", &un_jour) ;
    switch (un_jour ) {
        case lundi :
            printf ("C'est Lundi"); break ;
        case mardi :
            printf ("C'est Mardi"); break ;
        case mercredi :
            printf ("C'est Mercredi"); break ;
        default :
            printf ("Ce n'est ni Lundi ni Mardi ni Mercredi");
    }
    return 0;
}
```

TABLEAUX 1 ET PLUSIEURS DIMENSIONS

Motivation



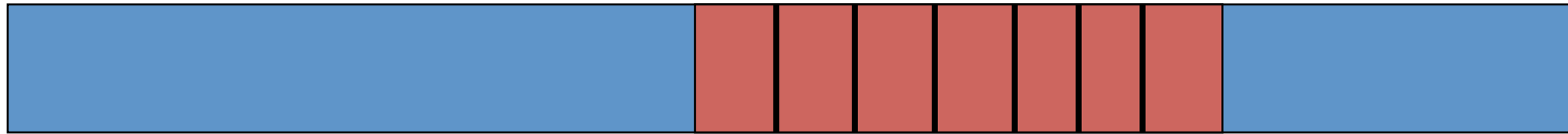
Structure de donnée:

- tableau a 2 dimension

Algorithmes:

- surtout I.A.

Les tableaux



- Accès indexé (de 0 à $n-1$ pour un tableau de n éléments)
- Stockage compact
- Taille fixe, en général
- Réajustement de taille coûteux en temps
- Insertion d'élément onéreuse en temps.

Problématique

- Supposons que nous ayons **2** notes
 - Déclarer **2** variables; OK
- Supposons que nous ayons **20** notes
 - Déclarer **20** variables; ???
- Supposons que nous ayons **200** notes
 - Déclarer **200** variables; ????



Problématique (suite)

- Supposons que nous ayons 12 notes
- Déclarer 12 variables
 - Succession de 12 instructions **lire**
 - $\text{moy} \leftarrow (n_1 + n_2 + \dots + n_{12}) / 12$

Problématique (fin)

- Avec des centaines ou milliers de valeurs

→ suicide direct



- rassembler toutes ces variables en une seule

→ tableau

Définitions

- Une **variable indicée**
- Un **vecteur**
 - Ensemble de valeurs portant le même nom de variable et repérées par un nombre
 - Le nombre qui, au sein d'un tableau, sert à repérer chaque valeur s'appelle **indice**.
 - Désigner un élément du tableau, on fait figurer le nom du tableau, suivi de l'indice de l'élément

TABLEAUX À UNE DIMENSION

Définition

- Exemple

10000	8500	12300	13000	6500	9800
1 ^{ère} case	2 ^{ème} case	3 ^{ème} case	4 ^{ème} case	5 ^{ème} case	6 ^{ème} case

- Nom : salaire
- Taille (nombre d'éléments) : 6

- Remarques

- « case contenant une valeur » doit faire penser à celle de variable
- éléments d'un tableau correspondent à des emplacements contiguës en mémoire

Déclaration

Variable

nom_tableau : **tableau** [*N*] de *type*

Où

- *nom_tableau* est l'identificateur du tableau
- *N* est la taille du tableau et doit être une constante entière et positive.
- *type* est le type des éléments du tableau

Exemple

- **variable**
 - salaire : tableau[6] de réel
 - nom_clients: tableau [20] de caractere
 - notes : tableau[8] d'entier
- 1^{er} élément du tableau → 0 et est désigné par *nom_tableau* [0]
- 2^e élément du tableau → 1 et est désigné par *nom_tableau* [1]
- ...
- Le dernier élément du tableau → N-1 et est désigné *nom_tableau* [N-1]

Exemple (suite)

10000	8500	12300	13000	6500	9800
0	1	2	3	4	5

- Pour désigner un élément, l'indice peut être écrit sous forme de :
 - Nombre en clair, exemple: **salaire [4]**
 - Variable, exemple: **salaire [i]**, avec **i** de type entier
 - Expression entière exemple : **salaire [k+1]**, avec **k** de type entier

Remarques

- ∇ sa forme la valeur de l'indice est
 - entière
 - comprise entre les valeurs minimale et maximale
- Exemple, avec le tableau salaire, interdit d'écrire
 - `salaire[10]` ou `salaire[15]`.
 - Expressions font référence à des éléments qui n'existent pas

Remarques (suite)

- Mélanger indice et valeur
 - 3^e maison de la rue a forcément 3 habitants
 - 113^e maison de la rue a forcément 113 habitants
- Aucun lien en **i** et **salaire[i]**

Exemples

Algorithme gestab

variable

Note : Tableau [12] de réel

Moy, Som : réel

Début

Pour $i \leftarrow 0$ à 11 faire

Ecrire ("Entrez la note n°", i)

Lire Note(i)

Finpour

Som $\leftarrow 0$

Pour $i \leftarrow 0$ à 11 faire

 Som \leftarrow Som + Note(i)

Finpour

Moy \leftarrow Som / 12

Fin

Algorithme gestab

variable

Note : Tableau [12]
de réel

Moy, Som : réel

Début

Som $\leftarrow 0$

Pour $i \leftarrow 0$ à 11 faire

Ecrire ("Entrez la note n°", i)

Lire Note(i)

 Som \leftarrow Som + Note(i)

Finpour

Moy \leftarrow Som / 12

Fin

En Langage C

- Syntaxe
 - ***type*** *nom_tableau* [*N*] ;
 - *nom_tableau* = identificateur du tableau
 - *N* = taille tableau est constante entière positive

Exemples

- Exemple
 - **int tab[10] ;** *//déclare une variable tableau de 10 entiers appelée tab*
 - Réservation emplacement mémoire stocker 10 entiers
 - tab[0] désigne le premier élément du tableau tab
 - tab[9] désigne le dernier élément du tableau tab

Exemple (suite)

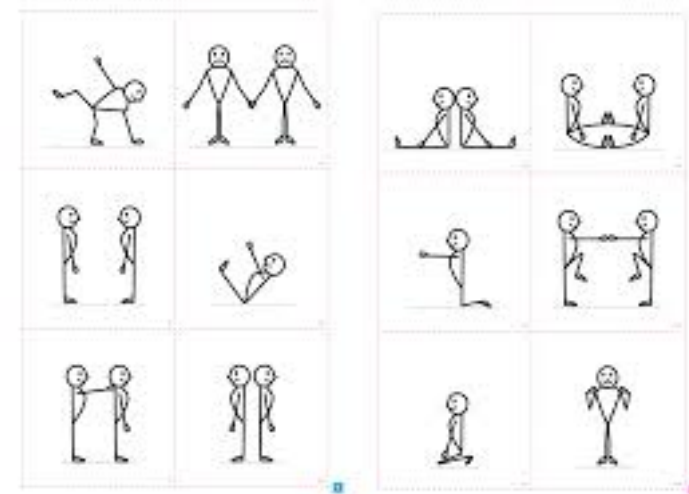
```
int main() {  
    float Note [12];  
    float Moy, Som;  
  
    for (int i=0; i<12; i++) {  
        printf("Entrez la note n°%d", i);  
        scanf( "%f", &Note[i]);  
    }  
    Som = 0;  
    for (int i=0; i<12; i++ )  
        Som = Som + Note[i];  
  
    Moy = Som / 12;  
    printf("la moyenne est = %f", Moy);  
    return 0;  
}
```


Initialisation

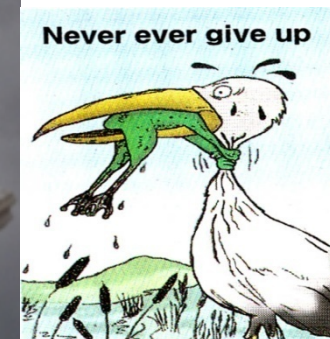
- *int T[] = {4,5,8,12,-3};*
 - Déclaration du tableau **T** de taille fixée à **5**.
 - initialisation des éléments du tableau
 - Indication du nombre d'éléments à l'intérieur des crochets non obligatoire

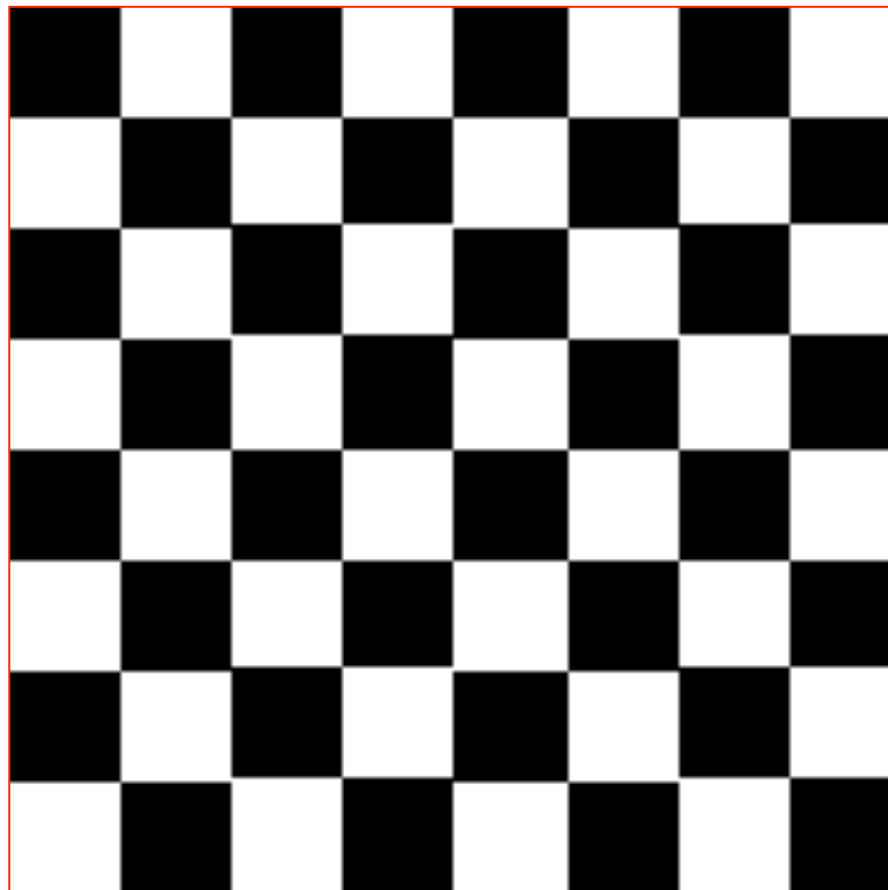
Exemple

```
int main() {  
  
    float Note [] = {12, 3, 14, 9};  
    float Moy, Som;  
  
    Som = 0;  
    for (int i=0; i<4; i++ )  
        Som = Som + Note[i];  
  
    Moy = Som / 4;  
    printf("la moyenne est = %f", Moy);  
  
    return 0;  
}
```



TABLEAUX À PLUSIEURS DIMENSIONS





Problématique



- Plus adapté à certains problèmes !
- Jeu de dames sur un damier de 64 cases
- Modélisé par un tableau de 64 valeurs
- Bien sûr, on peut programmer tout un jeu
- Pas simple et plus facile de modéliser un damier par un... damier !

Problématique (suite)

- **Pourquoi?** Modélisation de damier en une dimension
 - 1 à 8 : première ligne
 - 9 à 16 : deuxième ligne
 - ...
 - 57 à 64 : dernière ligne
- Dans case on met **1** si présence d'un pion et **0** sinon
 - $\text{case}(i) \rightarrow \text{case}(i+7), \text{case}(i+9), \text{case}(i-7), \text{case}(i-9)$.
- On peut résoudre le problème plus simplement
→ modéliser un damier par un damier

Le damier

- Tableaux à 2 dimensions
 - Valeurs repérées par deux coordonnées
- Déclaration
 - **Damier : tableau[8][8] d'entier**
 - Réservation de **64 entiers (8x8)**
 - **Damier (i, j) → Damier (i-1, j-1), Damier (i-1, j+1), Damier (i+1, j-1) et Damier (i+1, j+1)**

1 DIM vs. 2 DIM

- Aucune différence qualitative entre un tableau à **1** dimension et un autre à **2**
- Pas de **lignes** ni de **colonnes**
- Tableaux à **n** dimensions
 - **mingming**(3, 5, 4, 4) $\rightarrow 3 \times 5 \times 4 \times 4 = 240$ valeurs.
Chaque valeur y est repérée par **4** coordonnées.
 - Rarement **n > 3**
 - Matheux allez y car vous n'êtes pas comme nous
 - Habitues à l'espace à plusieurs dimensions

Création

- Déclaration
 - Plusieurs indices pour désigner un élément
 - Paire de crochets et taille pour chaque dimension
- Exemple
variable
nom_tableau: tableau[N1][N2] de type

Exemple

2	1
-4.5	23
0	9

- Matrice = tableau à 2 dimensions
- `tab[0][0]` → élém à la ligne 0 et à la colonne 0
 - Vaut 2
- `tab[2][1]` → élém à la ligne 2 et à la colonne 1
 - Vaut 9

Remarques

- Utiliser **deux boucles Pour** imbriquées pour parcourir tous les éléments d'un tableau à deux dimensions : la 1^{ère} boucle pour une dimension et la 2^e pour l'autre dimension
- Possible définir un type tableau et l'utiliser

- **Exemple :**

type

tab : tableau[3] [6] de Réel

variable

T₁, T₂ : tab // T₁ et T₂ sont des variables de type tab

Exemple : Résultat?

- Variables
 - X : Tableau [2,3] en Entier;
 - i, j, val en Entier;
- Début
 - val \leftarrow 1 ;
 - Pour i \leftarrow 0 à 1
 - Pour j \leftarrow 0 à 2
 - X(i,j) \leftarrow val;
 - val \leftarrow val + 1 ;
 - Finpour
 - Finpour
 - Pour i \leftarrow 0 à 1
 - Pour j \leftarrow 0 à 2
 - Ecrire X(i,j) ;
 - Finpour
 - Finpour
- Fin

- Variables
 - X : Tableau [2,3] en Entier;
 - i, j, val en Entier;
- Début
 - val \leftarrow 1 ;
 - Pour i \leftarrow 0 à 1
 - Pour j \leftarrow 0 à 2
 - X(i,j) \leftarrow val;
 - val \leftarrow val + 1 ;
 - Finpour
 - Finpour
 - Pour j \leftarrow 0 à 2
 - Pour i \leftarrow 0 à 1
 - Ecrire X(i,j) ;
 - Finpour
 - Finpour
- Fin

En Langage C

- Ajouter paire de crochets et une taille pour chaque dimension.

- Syntaxe

type nom_tableau[N1][N2];

- **Exemple :**

- **double m[10][20];** */*m est une matrice de réels*/*
- **m[0][0]** désigne l'élément à la ligne 0, colonne 0
- **m[9][19];** désigne l'élément à ligne 9, colonne 19

Exemples : initialiser et compter le nombre de zéros dans la matrice

```
Int main(){
    Float mat[4][5];
    Int i=0, j, zero;
    For(; i<4; i++){
        For (j=0; j<5; j++)
            Mat[i][j] = j-i;
    }
    Zero = 0;
    For (i=0; i<4; i++){
        For (j=0; j<5; j++)
            If (mat[i][j] == 0)
                zero+=1;
    }
    Printf("le nombre d'éléments nuls est %d", zero);
    Return 0
}
```

Produit matricielle

- $\forall i,j : c_{ij} = \sum_{k=1}^n a_{ik} * b_{kj}$

- Exemple

$$\begin{pmatrix} 1 & 0 \\ -1 & 3 \end{pmatrix} \times \begin{pmatrix} 3 & 1 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} (1 \times 3 + 0 \times 2) & (1 \times 1 + 0 \times 1) \\ (-1 \times 3 + 3 \times 2) & (-1 \times 1 + 3 \times 1) \end{pmatrix} = \begin{pmatrix} 3 & 1 \\ 3 & 2 \end{pmatrix}$$

CHAÎNES DE CARACTÈRES

Introduction

- Chaîne de caractère gérée en langage C comme un tableau contenant des caractères
- Particularité : la dernière case du tableau utilisée pour la chaîne contient le **caractère spécial \0** appelé **caractère nul**
- Caractère représente la fin de la chaîne

Déclaration

- Tableau de caractères
- **Syntaxe**
 - `char nom_chaine [N] ;`
 - (dans ce cas limitée à N-1 caractères (+ '\0'))
- **Exemple**
 - Déclarer chaîne de 5 caractères
 - utiliser un tableau de taille 6
 - `char ch[6] ;`
 - **ch** est dans ce cas limitée à 5 caractères (+ \0)

Exemple

- Comme les tableaux numériques, on peut initialiser un tableau de caractères (ou chaîne de caractères) lors de sa déclaration
- **Exemple**
 - `char ch[] = "bonjour";`
 - le compilateur réserve un tableau de 8 octets
 - 7 octets pour `"bonjour"`
 - 1 octet pour le caractère de fin de chaîne `'\0'`.

Exemple (suite)

'b'	'o'	'n'	'j'	'o'	'u'	'r'	\0
-----	-----	-----	-----	-----	-----	-----	----

- On peut aussi écrire:

```
char ch[] = {'b','o','n','j','o','u','r'};
```

Fonctions des chaînes (1)

- La bibliothèque standard **string.h**
 - **strlen** → longueur (nombre caractères chaîne)
Cette fonction ne prend pas en compte le caractère \0
 - **strlen("bonjour"); // retourne 7**

Fonctions des chaînes (1)

- La bibliothèque standard **string.h**
 - **strcpy** → affectation
 - **strcpy**(ch, "hello") ; // ch="hello";
 - **strncpy** → affectation de **lgmax** caractères de "hello" à **ch** (en complétant par **\0** éventuellement)
 - **strncpy**(ch, "hello everybody", lgmax) ; // ch="hello e" si lgmax = 7

Fonctions des chaînes (2)

- `ch="hello "`
- **`strcat`** → concaténation de deux chaînes
 - **`strcat`** (`ch`, `"world"`); // `ch = "hello world"`
- **`strncat`** → concaténation de deux chaînes
 - `ch="hello "`
 - **`strncat`** (`ch`, `"world"`, `lgmax`); // `ch = "hello wo"` si `lgmax = 2`

Fonctions des chaînes (2)

- **strcmp** → comparaison de deux chaînes
 - **n = strcmp(ch1, ch2) ;**
 - **n** vaudra :
 - un **nombre négatif** si **ch1 < ch2** au sens syntaxique
 - **0** si **ch1** et **ch2** sont identiques
 - un **nombre positif** si **ch1 > ch2**
- **strncmp** → comme strcmp mais se limite à lgmax caractères
 - **n = strncmp(ch1, ch2, lgmax) ;**

STRUCTURES

Introduction

- Une structure désigne sous un seul nom un ensemble d'éléments pouvant être de types différents.
- Elle est un agrégat de données de types plus simples.
- Elle permette de construire des types complexes à partir des types de base ou d'autres types complexes

Construction

- Composée d'éléments appelés champ ou membre désignés par des identificateurs
- Types donnés dans la déclaration de la structure
- Types peuvent être n'importe quel autre type, même une structure.
- Les variables de type structure sont aussi appelées structures

En algorithmique

Déclaration

Type

nom_type = **Structure**

champ₁ : type₁

...

champ_N : type_N

FinStructure

- où

- *nom_type* est l'identificateur du nouveau type

- *type*₁, ..., *type*_N sont les types respectifs des champs
*champ*₁, ..., *champ*_N

Exemples

- Les types **date** et **complexe** :

- **type**

- date = **Structure**

- jour : **entier**

- mois : **chaîne**

- année : **entier**

- FinStructure**

- complexe = **Structure**

- re : **réel**

- im : **réel**

- FinStructure**

Déclaration variable

- Comme tous les autres types
- Déclarer des variables
- Exemples:

Variable

d : date // d est une variable de type date

z : complexe // z est une variable de type complexe

Opérations

- Accès aux champs grâce à l'opérateur *point* **'.'**
- Exemple: **$x.champ_1$** = $champ_1$ d'une variable structure x
- Opérations valides sur le champ = opérations valides sur le type du champ
- Opérateur d'affectation valide (# tableau)
 - Copier tous les champs de la structure

Exemple

Programme date

Type

date = **Structure**

jour : **entier**

mois : **chaîne**

année : **entier**

FinStructure

Variable

d1, d2 : date

année : **entier**;

Début

// initialiser la date d1

d1.jour ← 23

d1.mois ← "Novembre"

d1.année ← 2000

// initialiser la date d2 à partir de d1

d2 ← d1

// afficher la date d2

Ecrire(d2.jour, '/', d2.mois, '/', d2.année)

// copier le champ année de d2 dans la variable année

année ← d2.année

// saisir le champ année de d2

Lire(d2.année)

Fin

Langage C

Création de structure

- Mot-clé **struct**

- Syntaxe

```
struct nom_type{  
    Type1 nom_champ1;  
    ...  
    TypeN nom_champN;  
};
```

- Exemple

```
struct complexe  
{  
    float réelle ;  
    float imaginaire ;  
};
```

Déclaration variable

- **Syntaxe :** `struct complexe c;`
 - Accès champs d'une variable structure avec `.`
 - Nom de la variable suivi de `.` + nom du champ
- **Exemple:**
 - *c.réelle = 0;*
 - *c.imaginaire = 1;*

Remarque 1

- Possible de déclarer variable structure sans créer au préalable le type structure
- Syntaxe

```
struct nomStructure  
{  
    Type1 champ1;  
    Type2 champ2;  
    ...  
} nomVariable;
```

Remarque 2

- Mêmes règles d'initialisation lors de la déclaration que pour les tableaux.

- **Exemple 1**

`struct complexe z = {2. , 2.};`

- **Exemple 2**

`struct complexe z1 = {2. , 2.};`

`struct complexe z2;`

`z2 = z1;`

COMPLÉMENTS

Opérateur **typedef**

- Alléger l'écriture des programmes
- Attribuer un nouvel identificateur à un type existant à l'aide du mot clé **typedef**
- Types de base | énumérations | structures
- Syntaxe
typedef type synonyme;

Exemple 1

- ***enum*** *couleur* {bleu, blanc, rouge, vert, jaune, noir} ;
- ***typedef*** *enum couleur* **couleur** ; // **couleur** devient le **synonyme** de **enum couleur**

```
int main()  
{  
    couleur c ; // c est une variable de type couleur  
    ...  
}
```

Exemple 2

```
struct complexe  
{  
    double reelle;  
    double imaginaire;  
};
```

```
typedef struct complexe complexe; //complexe est synonyme de  
struct complexe
```

```
int main()  
{  
    complexe z;  
    ...  
}
```

Type tableau

- Possibilité de définir un type tableau de la même manière qu'on déclare une variable tableau mais en écrivant d'abord le mot **typedef**.

- **Exemple :**

```
typedef int tab[10]; // tab est un "type" tableau  
de 10 entiers
```

```
tab T; // T est une variable tableau de 10 entiers
```

Opérateur sizeof

- Argument
 - type ou
 - nom de variable
- Retour
 - taille (en octets) de l'espace mémoire nécessaire pour stocker une valeur de ce type.
 - **int i = sizeof(int); /* i vaut 4 */**

Fonctions Maths

- **SIN** double sin (double x)
- **COS** double cos (double x)
- **TAN** double tan (double x)
- **ASIN** double asin (double x)
- **ACOS** double acos (double x)
- **ATAN** double atan (double x)
- **ATAN2** double atan2 (double y, double x)
 - Fournit la valeur de $\arctan(y/x)$

Fonctions Maths

- **SINH double sinh (double x)**
 - Fournit la valeur de $\text{sh}(x)$
- **COSH double cosh (double x)**
 - Fournit la valeur de $\text{ch}(x)$
- **TANH double tanh (double x)**
 - Fournit la valeur de $\text{th}(x)$
- **EXP double exp (double x)**
- **LOG double log (double x)**
 - Fournit la valeur du logarithme népérien de x : $\text{Ln}(x)$ (ou $\text{Log}(x)$)
- **LOG10 double log10 (double x)**
 - Fournit la valeur du logarithme à base 10 de x : $\text{log}(x)$

Fonctions Maths

- **POW double pow (double x, double y)**
 - Fournit la valeur de x^y
- **SQRT double sqrt (double x)**
- **CEIL double ceil (double x)**
 - Fournit (sous forme d'un double) le plus petit entier qui ne soit pas inférieur à x .
- **FLOOR double floor (double x)**
 - Fournit (sous forme d'un double) le plus grand entier qui ne soit pas supérieur à x .
- **FABS double fabs (double x)**
 - Fournit la valeur absolue de x .

Opérateur conditionnel

- `if (a>b)`
 - `max = a ;`
- `else`
 - `max = b ;`
- `max = a>b ? a : b // affectation`
- `a>b ? i++ : i-- ; // sans affectation`

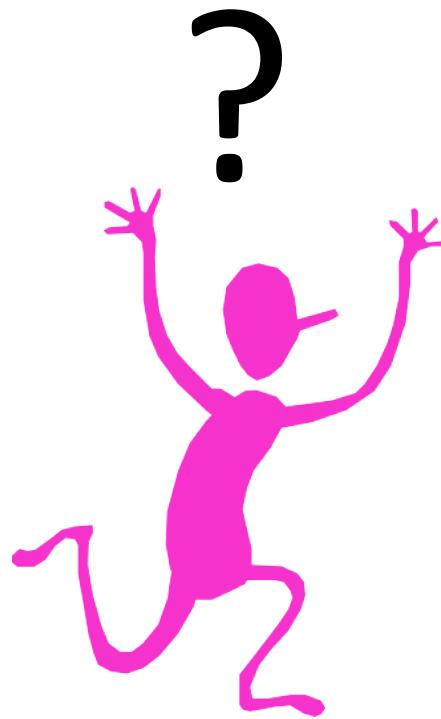
Exemple

- Calcul de la valeur absolue d'une expression
 - $3*a+1 > 0 ? 3*a+1 : -3*a-1$
- Cas où parenthèse obligatoire
 - $z = (x=y) ? a : b$
 - Affecter y à x ($x \leftarrow y$)
 - Si cette valeur # zéro ($z \leftarrow a$)
 - Sinon ($z \leftarrow b$)

Exemple (suite)

- $z = x = y ? a : b$
 - Est évaluée comme
- $z = x = (y ? a : b)$

END



CHAINES

STRUCTURES

COMPLEMENTS : TYPEDEF, SIZEOF

Overview

1. PRÉSENTATION DU COURS
2. TYPES DE DONNEES COMPOSEES
3. **POINTEURS**
4. SOUS PROGRAMMES
5. RECURSIVITE
6. ALGORITHMES DE TRI
7. FICHIERS

Chapitre 3

POINTEURS

Intuition

- ***Kernighan et Ritchie dans "programming in C"***

*« ... Les pointeurs étaient mis dans le même sac que l'instruction **goto** comme une excellente technique de formuler des programmes incompréhensibles. Ceci est certainement vrai si les pointeurs sont employés négligemment, et on peut facilement créer des pointeurs qui pointent 'n'importe où'. Avec une certaine discipline, les pointeurs peuvent aussi être utilisés pour programmer de façon claire et simple. C'est précisément cet aspect que nous voulons faire ressortir dans la suite. ... »*

Définition

- **Pointeur** : *une variable spéciale qui peut contenir l'**adresse** d'une autre variable.*
- La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de ***pointeurs***.

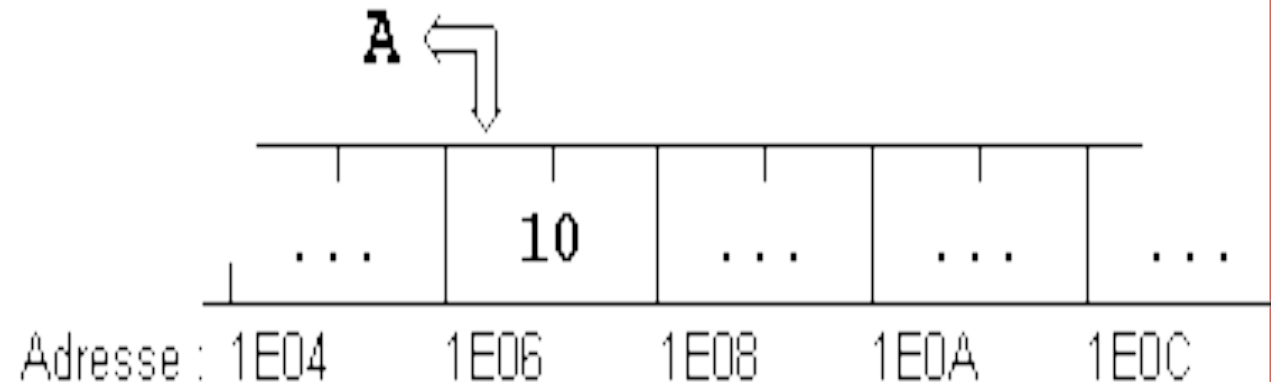
Cas du langage C

- Ils jouent un rôle primordial dans la définition de fonctions
 - passage des paramètres fait toujours par valeur
 - pointeurs sont le seul moyen de passer des variables par adresse.
 - traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs

Adressage de variables (1)

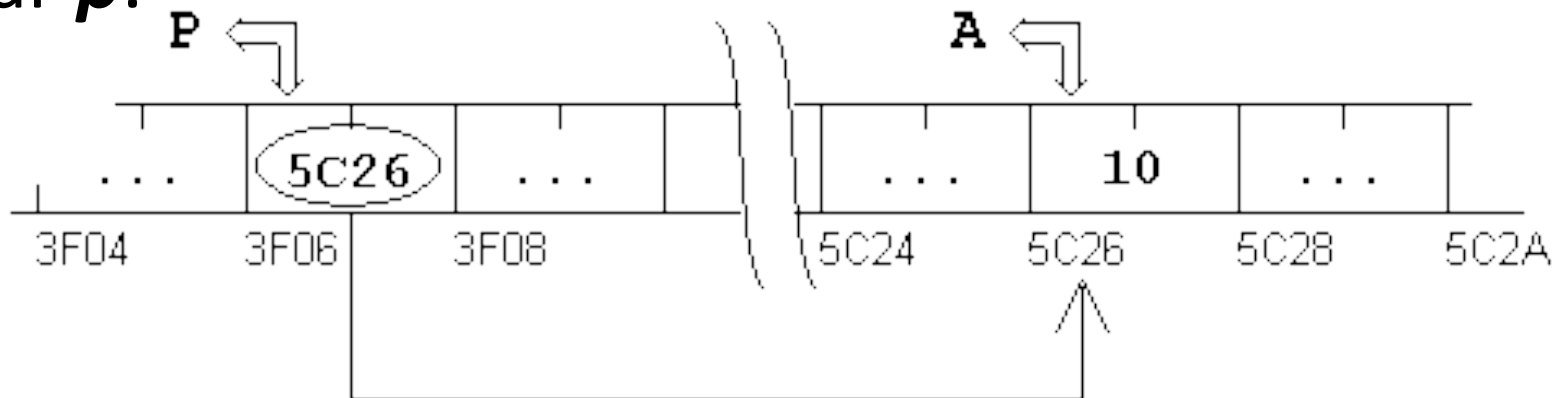
- **Adressage direct** : Accès au contenu d'une variable par le nom de la variable.
- **Exemple** :

```
short A;  
A = 10;
```



Adressage de variables (2)

- **Adressage indirect** : copier l'adresse d'une variable *a* (dont nous ne pouvons ou ne voulons pas utiliser) dans un pointeur *p*, on peut accéder à l'information de *a* en passant par *p*.



Utilisation de Pointeurs

- Limité à un type de données
- Si p contient l'adresse de a alors " p pointe sur a "
- Un pointeur peut pointer sur différentes adresses
- Le nom d'une variable reste lié à la même adresse

Opérateurs sur les Pointeurs

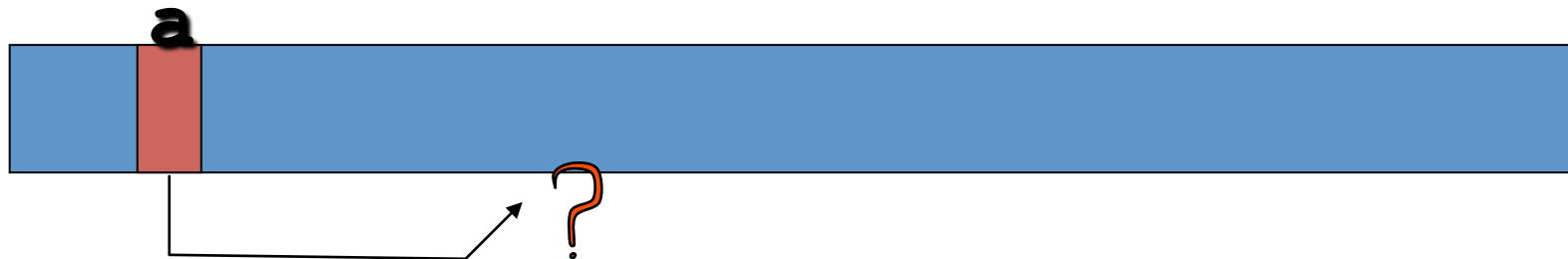
- Opérateurs de base
 - Adresse de : **&**
 - Contenu de : *****

Déclaration de pointeur (1)

- Déclaration:
 - L'instruction **<Type> * <nom_pointeur>**
 - Déclare un pointeur **nom_pointeur**
 - Reçoit des adresses de variables de type **<Type>**
 - Exemple : **int * P;**

Déclaration de pointeur (2)

~~int* a;~~

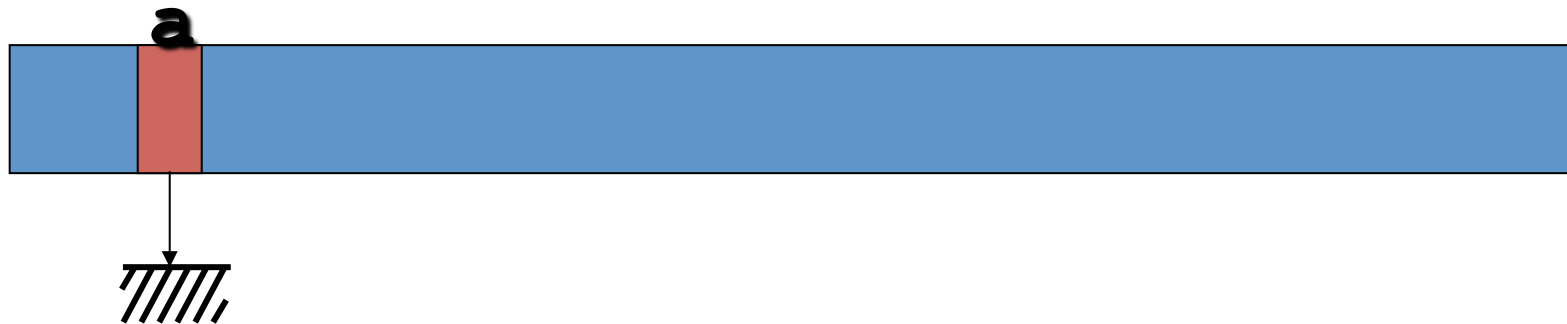


- Déclaration d'un pointeur vers un entier

Déclaration de pointeur (fin)

~~int* a;~~

int* a = NULL;



- ***Déclaration*** d'un pointeur vers un entier ***et initialisation*** à "NULL"

Utilisation de pointeurs (1)

- ***<nom_pointeur>** désigne le contenu de l'adresse référencée par le pointeur.
- Exemple
 - Int * P;
 - Int A=10, B;
 - P=&A;
 - **B=*P;** // B=10;
 - *P=90; // A = 90

Utilisation de pointeurs (2)

- **&<nom_var>** → l'adresse de la variable
 - S'applique à des variables et des tableaux.
 - Non à des constantes ou expressions
- **Exemple :**
 - *int N;*
 - *printf("Entrez un nombre entier : ");*
 - *scanf("%d", &N);*

Opérations sur les pointeurs

- Opérations élémentaires sur pointeurs
 - Priorité de **&** et ***** : même priorité que les opérateurs unaires (**!**, **++**, **--**).
 - Dans une expression ils sont tous évalués de droite à gauche.
 - Si **P** pointe sur **X** alors ***P** et **X** sont interchangeables

Incrémentation de pointeur

- Exemple : après l'instruction **P = &X;**
 - **Y = *P+1** **Y=X+1;**
 - ***P = *P+10** **X=X+10;**
 - **++*P** **++X**
 - **(*P)++** **X++ (parenthèses obligatoires)**

Pointeur *NULL*

- Zéro (0) est utilisé pour dire qu'un pointeur ne pointe "nulle part"
- Exemple: *int * p; p = 0;*
- Les pointeurs sont des variables
 - *int * p1, *p2; → p1=p2;*

Allocation d'espace (1)

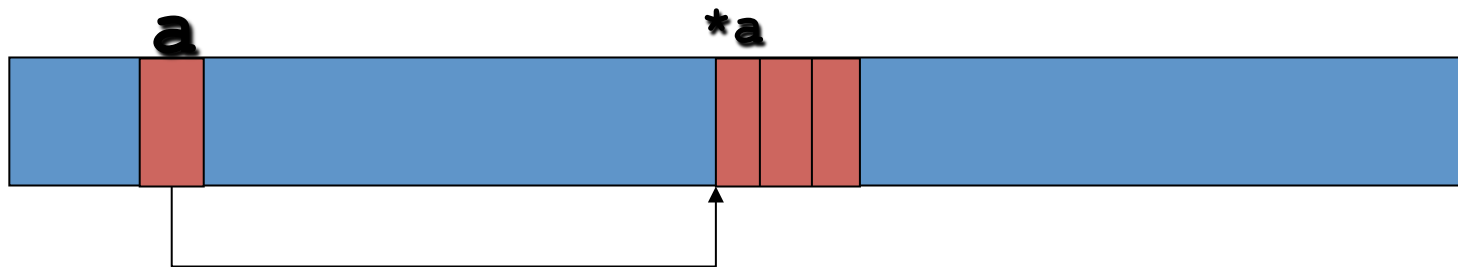
- `malloc(3*sizeof(int)) ;`



- ***Allocation dynamique*** de place mémoire (pour 3 entiers)

Allocation d'espace (2)

- ~~int* a = malloc(3*sizeof(int));~~
- int* a = (int*)malloc(3*sizeof(int));

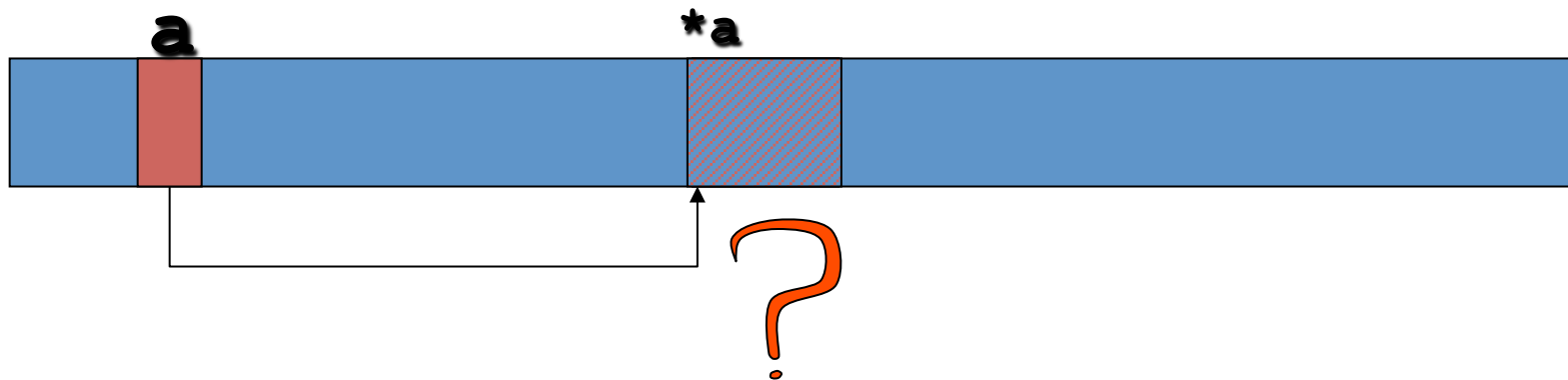


- ***Allocation dynamique*** de place mémoire (pour 3 entiers) et **assignation**

Libération de la mémoire

```
free(a) ;
```

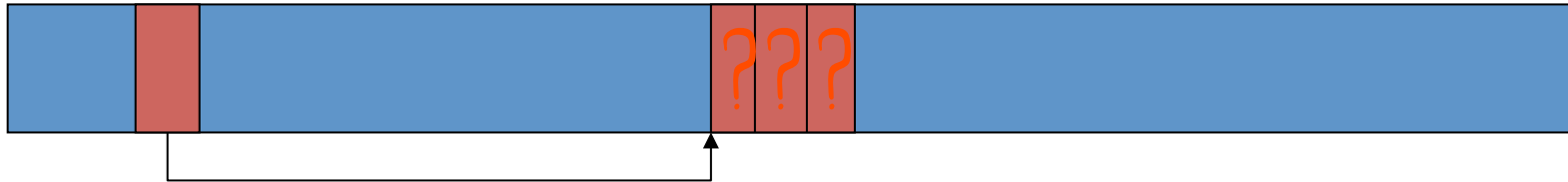
```
a = NULL ;
```



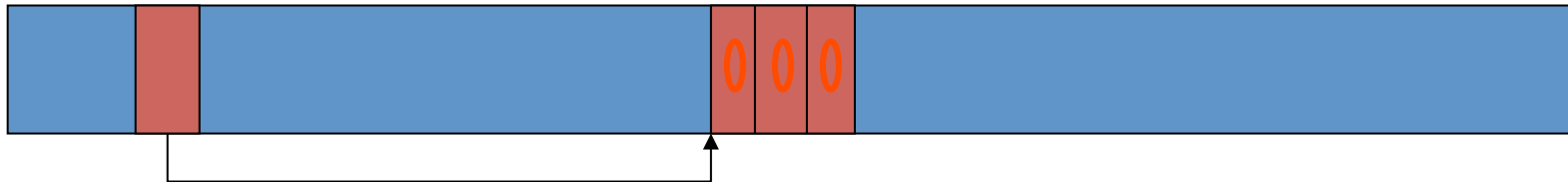
Désallocation dynamique

Réallocation de mémoire

- `int* a = (int*)malloc(3*sizeof(int));`



- `int* a = (int*)calloc(3, sizeof(int));`



- `a = (int*)realloc(4*sizeof(int));`



Remarques

- Pb de **malloc** pour les tableaux: pas d'initialisation. → **calloc**
- Coût mémoire de **realloc**:
 - Alloue un nouveau bloc de mémoire
 - Copie les valeurs de l'ancien bloc dans le nouveau
 - Libère l'ancien bloc
 - Retourne l'adresse du nouveau bloc

Pointeurs et tableaux (1)

- Étroite relation entre pointeur et tableau
- Toute opération avec indices peut être faite à l'aide de pointeurs
- Adressage des composantes d'un tableau
 - nom Tableau = adresse de son premier élément
 - ***&tableau[0]*** et ***tableau*** sont une seule et même adresse
 - Nom tableau = pointeur constant sur son premier élément

Pointeurs et tableaux (2)

- **Exemple :**
 - **A** tableau de type **int** : **int A[10]**
 - **P** pointeur sur **int** : **int *P;**
 - **P = A;** équivalente à **P = &A[0];**



Pointeurs et tableaux (3)

- Si p pointe sur une composante du tableau
 - $P+1$ pointe sur la composante suivante
 - $P+i$ pointe sur la i -ième composante derrière P
 - $P-i$ pointe sur la i -ième composante devant P .
- Ainsi, après l'instruction $P = A;$ (A un tableau)
 - le pointeur P pointe sur $A[0]$, et
 - $*(P+1)$ désigne le contenu de $A[1]$
 - $*(P+2)$ désigne le contenu de $A[2]$
 - ...
 - $*(P+i)$ désigne le contenu de $A[i]$

Pointeurs et tableaux (fin)

- Différences entre un pointeur et un tableau
 - Un pointeur est une variable
 - $P=A$ et $P++$ sont des opérations permises
 - Un nom de tableau est une constante
 - $A=P$ ou $A++$ ne sont pas permises

Intérêts des pointeurs

- Gestion de l'espace mémoire en cours d'exécution
- Modifications de variables passées en paramètres de fonction
- Représentation de tableaux: accès direct et indexé
- Références croisées
- Fonctions virtuelles en programmation objet

Overview

1. PRÉSENTATION DU COURS
2. TYPES DE DONNEES COMPOSEES
3. POINTEURS
4. SOUS PROGRAMMES
5. RECURSIVITE
6. ALGORITHMES DE TRI
7. FICHIERS

Chapitre 4

SOUS PROGRAMMES

Ce qui a été vu

- Structures de données
- Tableaux à
 - Une dimension
 - deux dimensions
- Pointeurs
- **Plusieurs** algorithmes

Problème

- Dès qu'on commence à écrire des programmes sophistiqués, il devient difficile
 - d'avoir une vision globale sur son fonctionnement
 - de trouver des erreurs
- Solution : décomposer le problème en sous problèmes
 - Trouver une solution à chacun
 - Chaque solution partielle donne lieu à un sous-programme

Programmation procédurale

- Principe:
 - écrire des programmes en utilisant des sous-programmes
- Forme générale d'un programme

Programme P

Sous-programme SP_1

...

Sous-programme SP_n

FinP

Exemple

- Algorithme qui teste si M est la matrice inverse de N

Algorithme A

A1: Lecture de M

A2: Lecture de N

A3: Affecter à O le résultat de $M * N$

A4: Tester si O est l'identité

A5: Affichage du message adéquat

Fin A

Procédures & Fonctions

- En algorithmique, on distingue deux types de sous-programmes
 - Les procédures
 - Les fonctions

Présentation

- ***Fonctions***
 - Paramètres
 - Type retourné
- ***Procédures***
 - Paramètres
 - Appel par variable
 - Appel par valeur

Fonctions : structure

- Une fonction est un sous-programme qui :
 - A un nom
 - Peut avoir des paramètres
 - Qui retourne une valeur d' un certain type
 - Qui peut avoir besoin de variables
 - Qui est composé d' instructions

Fonctions : déclaration

Fonction **nomf** (<paramètres>): type

Déclaration des variables

Début

instructions

nomf ← expression

Fin fonction

Fonctions : Exemple

- Fonction qui retourne le carré d' un entier :

Fonction ***carré***(n : entier): entier

Début

carré $\leftarrow n * n$

fin fonction

Fonction: utilisation dans un algorithme

Algorithme ex1

Variable i, j: entier

Fonction carré(n : entier): entier

Début

carré \leftarrow n * n

fin fonction

Début

Lire (i)

Ecrire(carré(i))

~~j \leftarrow carré(i)~~

~~Ecrire(j)~~

Fin

Fonctions: A retenir

- Une fonction retourne toujours une valeur
- Une fonction **NomF** contient toujours une instruction de la forme
NomF \leftarrow Expression
- Il ne faut jamais utiliser d'instructions de la forme
NomF (paramètres) \leftarrow expression
- En général, l'utilisation d'une fonction se fait
 - Soit par une affectation: $v \leftarrow \mathbf{NomF}(\text{paramètres})$
 - Soit dans l'écriture: Ecrire (**NomF** (paramètres))

Fonctions : quelques exercices

- Ecrire une fonction qui
 - Prend un tableau de 5 entiers, puis
 - Retourne la valeur Vraie ou Faux selon que le tableau est trié par ordre croissant ou non

Fonctions : Principe du test

- On suppose d'abord que le tableau est trié
- Ensuite on compare chaque case à sa suivante:
 - Si l'ordre n'est pas respecté alors on conclut que le tableau n'est pas trié

Fonction : test du tri

Fonction **trié**(T: Tableau[5] d' entiers): Booléen

Variable i : entier

Variable b : booléen

Début

 b ← Vrai

 Pour i = 1 à 4

 Si $T(i) > T(i+1)$ alors

 b ← Faux

 FinSi

 FinPour

trié ← b

Fin Fonction

Fonction : exemple

- Ecrire un *algorithme* qui
 - lit un tableau de 5 entiers puis
 - teste s'il est trié ou pas

Fonction : exemple

Algorithme ex1

Variable T1 : tableau[5] d'entier

Variable i : entier

Fonction Trié(...)

...

Fin fonction

Début

Pour i = 1 à 5

Lire(T1(i))

Fin Pour

Si Trié(T1) = Vrai Alors

Ecrire("c' est trié")

FinSi

Sinon

Ecrire(" Non trié ")

FinSinon

Fin

Fonctions : en C

```
int trié(int t[]) {  
    int b, i;  
    b = 1;  
    for (i = 0; i < 4; i++)  
        if (t[i] > t[i + 1])  
            b = 0;  
    return b;  
}
```

Fonctions : en C

```
int trié(int t[]) {  
    int b=1, i=0;  
    while (i < 4 && b!=0) {  
        if (t[i] > t[i + 1])  
            b = 0;  
        i++;  
    }  
    return b;  
}
```

Fonctions : utilisation dans un programme C

```
int main (){  
    int T1[5], i ;  
    for (i = 0; i<5; i++) {  
        printf("donner un entier");  
        scanf("%d", T1+i);  
    }  
    if trié(T1)  
        printf ("trié");  
    else  
        printf ("non trié")  
}
```

Fonctions : C

- C propose plusieurs fonctions qui sont déjà définies et qu'on peut utiliser dans nos programmes
- Exemples :
 - Sqrt: retourne la racine carrée
 - `sqrt(4)` retourne 2
 - `exp` : retourne l'exponentielle d'un nombre
 - `exp(0)` retourne 1
 - `printf`, `scanf`, `malloc`, `calloc`, `sizeof`
 - ...

Fonctions : en C

```
Char * permute (char * s, int i, int j) {  
    char x=*(s+i); // x=s[i]  
  
    s[i]=*(s+j); // s[i]=s[j]  
    s[j]=x;  
  
    return s;  
}
```

Fonctions : en C

```
Char ** ranger(char * s, char ** m, int t) {  
    int i=0;  
    while (i<t && strcmp(s, m+i) != 0)  
        i++;  
    if (i==t)  
        strcpy(m[t], s);  
  
    return m;  
}
```

Procédures : définition

- Une procédure est un sous-programme qui ne retourne pas de valeur
- C' est donc un type particulier de fonction
- En général, une procédure modifie la valeur de ses paramètres
 - Je dis bien « en général », ce n' est pas toujours le cas

Procédures : structure

- Tout comme les fonctions, une procédure est un sous-programme qui :
 - A un nom
 - Peut avoir des paramètres
 - ~~– Qui retourne une valeur d'un certain type~~
 - Qui peut avoir besoin de variables
 - Qui est composé d'instructions

Procédures : déclaration

Procédure nomf (<paramètres>)

Déclaration des variables

Début

instructions

Fin procédure

Procédures : exemple

- Une procédure qui ajoute 2 à un entier

procédure **aug2**(n : entier)

Début

$n \leftarrow n+2$

Fin Procédure

Procédures : dans les algorithmes

- Ecrire un algorithme qui
 - Lit un entier positif n puis
 - Affiche tous les nombres impaires inférieurs à n

Procédure : dans les algorithmes

Algorithme ex1

Variable i,n: entier

Procédure Aug2(..)

...

Fin Procédure

Début

Lire(n)

$i \leftarrow 1$

Tant que $i \leq n$

Ecrire(i)

aug2(i)

Fin TantQue

Fin

Procédures : A retenir

- Une procédure ne retourne pas de valeur
- Il est donc faux de l' affecter à une variable
 - Ne jamais écrire : $j \leftarrow \text{aug2}(i)$

Procédures : en C (????)

```
void aug2(int n){  
    n = n + 2;  
}
```

- Tous les programmes qu'on a écrits jusqu'à présent étaient en fait
 - Des procédures sans paramètres

Procédures : appel en C

```
void aug2(int n){  
    n = n +2;  
}
```

```
void main(){  
    int i, n;  
    printf("donner un entier");  
    scanf("%d", &n);  
    i = 1;  
    while (i <= n) {  
        printf(" %d", i);  
        aug2(i);  
    }  
}
```

Procédures : en C

```
void aug2(int * n){  
    *n = *n +2;  
}
```

- Tous les programmes qu'on a écrits jusqu'à présent étaient en fait
 - Des procédures sans paramètres

Procédures : appel en C

```
void aug2(int * n){  
    *n = *n +2;  
}
```

```
void main(){  
    int i, n;  
    printf("donner un entier");  
    scanf("%d", &n);  
    i = 1;  
    while (i <= n) {  
        printf(" %d", i);  
        aug2(&i);  
    }  
}
```

Procédures & fonctions : appels imbriqués

- Dans la définition d'une procédure, on peut faire appel à une autre procédure ou fonction déjà définie
- Même remarque pour les fonctions

Procédures & fonctions : appels imbriqués

Procédure aug4(n : entier)

Début

 aug2(n)

 aug2(n)

Fin Procédure

Procédures & fonctions : appels imbriqués

Fonction Puiss4(n : entier) : entier

Début

Puiss4 \leftarrow Carré(Carré(n))

Fin Fonction

Procédure : Appel par variable versus appel par valeur

- En général, les procédures modifient leurs paramètres.
- Ceci à cause du fait que *par défaut*, elles travaillent sur les variables elles même
- Dans certains cas, on ne veut pas que la procédure modifie ses paramètres
 - on lui précise qu' elle doit travailler sur leurs **valeurs**
 - Dans ce cas, la procédure travaille sur une **copie** des paramètres

Procédure: mode d'appel

- Exemple : on veut écrire un algorithme qui
 - saisit un tableau d'entiers puis
 - affiche ses éléments dans l'ordre croissant

Procédure : mode d'appel

- Idée :
 - On lit le tableau T
 - On fait appel à une procédure qui trie T
 - On parcourt ensuite les éléments du premier jusqu'au dernier
- Il ne faut par contre pas que le tableau soit trié définitivement
- La procédure doit donc travailler sur une « copie » non pas sur le tableau lui même

Procédure : Appel par valeur

- Si on veut qu'un des paramètres ne soit pas modifié par la procédure, il faut le faire précéder par le terme

- Val

- Exemple:

Procédure TrierEtAfficher(val t:tableau[5] d'entiers)

...

Fin procédure

Procédures : appel par valeur en C

- Il faut fournir les paramètres par valeur et non par leurs adresses

```
void proc(int i){  
    i=sqrt(i);  
    printf("%d", i);  
}
```

Procédures & fonctions: exemple complet

- Reprendre l'algorithme de tri et le développer cette fois-ci en utilisant des fonctions et des procédures
- Procédure de saisie du tableau
- Fonction qui retourne l'indice de la valeur max dans une partie du tableau
- Procédure qui échange les valeurs de deux cases
- Procédure qui fait le tri en utilisant la fonction et les 2 procédures ci-dessus

Procédure de saisie

Procédure saisir(t:tableau[5] d' entiers)

variable i: entier

Début

 Pour i = 1 à 5

 Lire(t(i))

 Fin Pour

Fin Procédure

Fonction qui retourne l'indice de la valeur max dans une partie du tableau

Fonction IndMax (t: tableau[5] d'entiers,
i: entier)

variable j, Max: entier

Début

Max \leftarrow i

Pour j = i à 5

Si t(Max) < t(j) Alors

Max \leftarrow j

FinSi

Fin Pour

IndMax \leftarrow Max

Fin Fonction

Procédure qui échange les cases i et j

Procédure échanger(i, j : entier, t: tableau[5] d' entiers)

Variable Z : entier

Début

$Z \leftarrow t(i)$

$t(i) \leftarrow t(j)$

$t(j) \leftarrow Z$

Fin Procédure

Procédure de Tri

Procédure Trier(T: Tableau[5] d' entiers)

Variable i: entier

Debut

Pour i = 1 à 4

Echanger(i, IndMax(i,T), T)

FinPour

FinProcédure

Algorithme de saisie et de tri

Algorithme ex

Variable T: Tableau[5] d' entiers

Début

 Saisir(T)

 Trier(T)

Fin

Exercice d'application

- Traduire l'algorithme précédent (saisie et tri) en LC

Overview

1. PRÉSENTATION DU COURS
2. TYPES DE DONNEES COMPOSEES
3. POINTEURS
4. SOUS PROGRAMMES
5. **RECURSIVITE**
6. ALGORITHMES DE TRI
7. FICHIERS

Chapitre 5

RECURSIVITE

Conception d'un algorithme

- Analyse descendante (AD)
 - Décomposer le problème en sous-problèmes
 - Décomposer les sous-problèmes en sous-sous-problèmes
 - Défaut : changement en cascade
- Analyse ascendante (AS)
 - Assembler fonctions, primitives et outils
 - Défaut : tuer une mouche avec une bombe atomique
- Mélange des deux
 - Mieux
 - AD ayant à l'esprit les modules bien conçus

Qualité d'un programme

- Qualité d'écriture
- Terminaison
- Validité
- Complétude
- Performance

Récurtivité : concepts

- Un algorithme est récursif lorsqu'il s'appelle lui-même
- Elle peut être cachée si A appelle B qui appelle A
- Exemple de la somme(a, b)

Réversivité : exemple

Algorithme factorielle(entier n) : retourne un entier

// je réfléchis récursivement

Debut

si $n=0$ alors retourner 1

sinon retourner $n \times \text{factorielle}(n-1)$

Fin

//et je suis efficace mais **faites gaffe**

Paradigme « diviser pour régner »

- **Récurtivité**

De l'art d'écrire des programmes qui résolvent des problèmes que l'on ne sait pas résoudre soi-même !

- **Définition récursive.** *Une définition récursive est une définition dans laquelle intervient ce que l'on veut définir.*
- **Algorithme récursif.** *Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même.*

Paradigme « diviser pour régner »

- **Principe et dangers de la récursivité**
 - **Principe et intérêt** : ce sont les mêmes que ceux de la démonstration par récurrence en mathématiques. On doit avoir :
 - un certain nombre de cas dont la résolution est connue, ces « cas simples » formeront les cas d'arrêt de la récursion ;
 - un moyen de se ramener d'un cas « compliqué » à un cas « plus simple ».
 - La récursivité permet d'écrire des algorithmes concis et élégants.

Paradigme « diviser pour régner »

- **Difficultés :**

- la définition peut être dénuée de sens :
 - Algorithme $A(n)$
 - **renvoyer** $A(n)$
- il faut être sûrs que l'on retombera toujours sur un cas connu, c'est-à-dire sur un cas d'arrêt ; il nous faut nous assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'applications.

Paradigme « diviser pour régner »

- **Non décidabilité de la terminaison**

Question : peut-on écrire un programme qui vérifie automatiquement si un programme donné P termine quand il est exécuté sur un jeu de données D ?

- **Entrée:** Un programme P et un jeu de données D .
- **Sortie:** *vrai* si le programme P termine sur le jeu de données D , et *faux* sinon.

- **Démonstration de la non décidabilité**

- Supposons qu'il existe un tel programme, nommé *termine*, de vérification de la terminaison. À partir de ce programme on conçoit le programme Q suivant :

Paradigme « diviser pour régner »

programme Q

résultat = termine(Q)

tant que résultat = *vrai* **faire** attendre une seconde **fin tant que**
renvoyer résultat

- Supposons que le programme Q (qui ne prend pas d'arguments) termine. Donc termine(Q) renvoie *vrai*, la deuxième instruction de Q boucle indéfiniment et Q ne termine pas. Il y a donc contradiction et le programme Q ne termine pas. Donc, termine(Q) renvoie *faux*, la deuxième instruction de Q ne boucle pas, et le programme Q termine normalement. Il y a une nouvelle fois contradiction : par conséquent, il n'existe pas de programme tel que termine.
- **Le problème de la terminaison est indécidable!!**

Paradigme « diviser pour régner »

- **Diviser pour régner**
 - **Principe**

Nombres d'algorithmes ont une structure récursive : pour résoudre un problème donné, ils s'appellent eux-mêmes récursivement une ou plusieurs fois sur des problèmes très similaires, mais de tailles moindres, résolvent les sous problèmes de manière récursive puis combinent les résultats pour trouver une solution au problème initial.

Le paradigme « diviser pour régner » donne lieu à trois étapes à chaque niveau de récursivité :

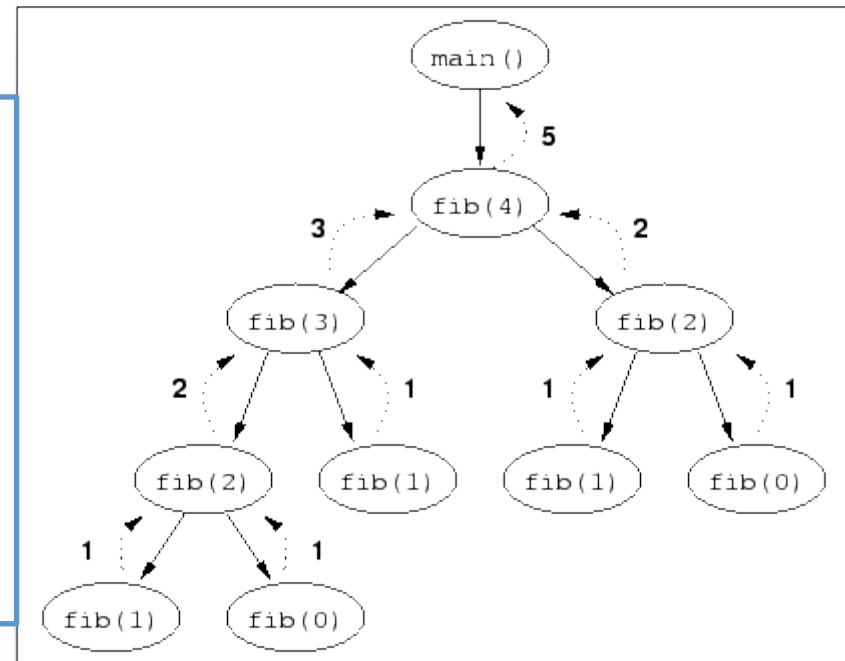
- **Diviser** : le problème en un certain nombre de sous-problèmes ;
- **Régner** : sur les sous-problèmes en les résolvant récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement ;
- **Combiner** : les solutions des sous-problèmes en une solution complète du problème initial.

Fonctionnement

- **Contexte** = paramètres formels ses variables locales...)
- Appel → dépôt (empilement) du contexte
- Fin exécution → retrait contexte de la pile (dépilement)
- Retour → contexte de la fonction appelante.
- Conclusion → empilements successifs puis dépilement

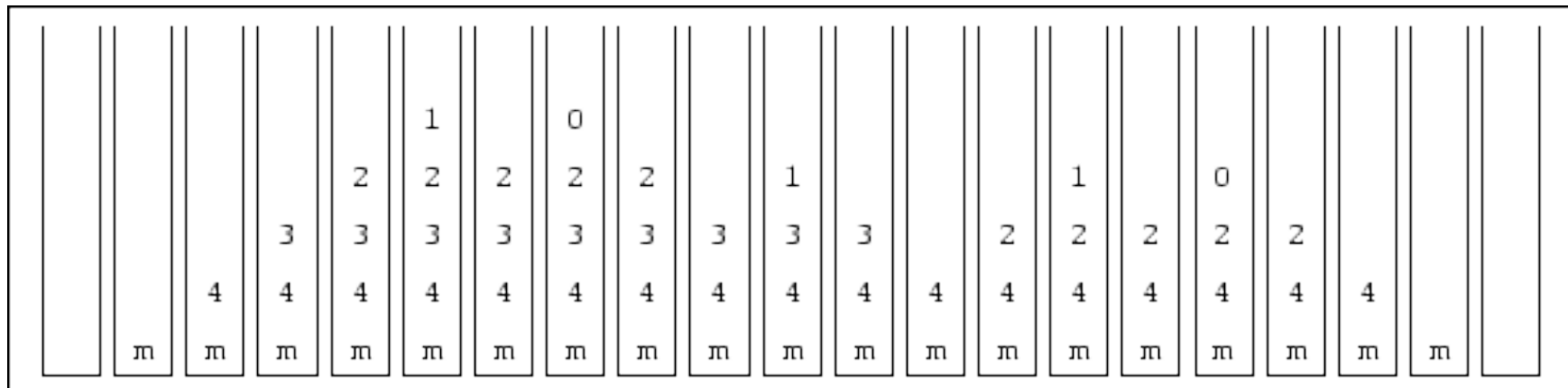
Exemple

Fonction **fib** (n : entier) : entier
Début
 si ($n \leq 1$) Alors
 retourner 1
 Sinon
 retourner **fib** ($n-1$) + **fib** ($n-2$)
 FinSi
FinFonction



États successifs de la pile d'exécution

- Appel de fib(4) dans main()
- main() → m
- fib(4) → 4
- fib(3) → 3
- fib(2) → 2
- fib(1) → 1
- fib(0) → 0



Principes

- Identifier
 - cas de base dont la résolution est connue (cas d'arrêt)
 - cas « compliqué » \rightarrow cas « plus simple ».
 - terminaison algorithme \rightarrow converge cas de base.

Avantages

- Écrire des algorithmes concis et élégants
- Transcrire quasi littéralement définitions mathématiques
 - appelées récurrences
- Risquer
 - grande occupation de la mémoire
 - longs temps d'exécution

Limites

- Écrire des algorithmes concis et élégants
- Transcrire quasi littéralement définitions mathématiques
 - appelées récurrences
- Risquer
 - grande occupation de la mémoire
 - longs temps d'exécution

Récurtivité : exemple 2

Algorithme **somme**(a,b:entiers): **entier**

// a et b étant 2 entiers ≥ 0

// calcul suivant le principe : $a+b=a+(b-1)+1$

Debut

 si $b = 0$ alors retourner a

 sinon

$\text{tmp} \leftarrow \text{somme}(a,b-1)$ //appel récursif

 retourner $\text{tmp} + 1$

 // on pouvait ecrire "*retourner 1 + somme (a,b-1)*"

 //c'était pareil, on aurait le même résultat

 finsi

fin

TYPES DE RÉCURSIVITÉ

Réversivité simple

- Exemple de la fonction puissance → un seul appel récursif

$$x^n = \begin{cases} 1 & \text{si } n = 0; \\ x \times x^{n-1} & \text{si } n \geq 1. \end{cases}$$

- Fonction PUISSANCE ($x : \text{réel}, n : \text{entier}$) : réel

Début

Si ($n = 0$) alors

retourner 1

sinon

retourner $x * \text{PUISSANCE}(x, n - 1)$

FinSi

FinFonction

Récurtivité multiple

- Exemple de la fonction combinaison avec relation Pascal

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

- Fonction COMB ($n : \text{entier}, p : \text{entier}$) : entier

Début

Si (($p = 0$) ou ($p = n$)) alors

retourner 1

sinon

retourner COMB ($n - 1, p$) + COMB ($n - 1, p - 1$)

FinSi

FinFonction

Réversivité mutuelle

$$\text{pair}(n) = \begin{cases} \text{vrai} & \text{si } n = 0; \\ \text{impair}(n-1) & \text{sinon;} \end{cases} \quad \text{et} \quad \text{impair}(n) = \begin{cases} \text{faux} & \text{si } n = 0; \\ \text{pair}(n-1) & \text{sinon.} \end{cases}$$

Fonction **PAIR** (n : entier) : booléen

Début

Si ($n = 0$) alors

retourner *vrai*

sinon

retourner **IMPAIR** ($n-1$)

FinSi

FinFonction

Fonction **IMPAIR** (n : entier) : booléen

Début

Si ($n = 0$) alors

retourner faux

sinon

retourner **PAIR** ($n-1$)

FinSi

FinFonction

Récurtivité imbriquée

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{sinon} \end{cases}$$

Fonction **ACKERMANN**(*m* : entier, *n* : entier): entier

Début

 si (*m* = 0) alors

 retourner *n*+1

 sinon si (*n* = 0) alors

 retourner **ACKERMANN**(*m* - 1, 1)

 sinon

 retourner **ACKERMANN**(*m* - 1, **ACKERMANN**(*m*, *n* - 1))

 FinSi

FinFonction

Diviser pour régner

- **Principe général** : Décomposition → Recomposition
- **Solution** : algorithmes intéressants et efficaces.
- **Diviser** : problème en un certain nombre de sous-problèmes
- **Régner** : sur sous-problèmes en
 - Résolvant récursivement
 - Résolvant directement si la taille d'un sous-problème est assez réduite
- **Combiner** : solutions sous-problèmes en une solution complète

Exemple : dichotomie

- recherche valeur **v** dans tableau entiers ordonné dans l'ordre **croissant**
- dichotomie → divisions successives du tableau en **2** parties
- **deb** et **fin** les indices de début et de fin du tableau **tab***
- **m** indice milieu tableau.
- Si **v = tab[m]** alors la valeur recherchée est au milieu du tableau.
- Sinon si **v < tab[m]** → **rechercher** dans intervalle **[deb, m-1]**
- Sinon **rechercher** dans intervalle **[m+1, fin]**

Algorithme de la dichotomie

Fonction **chercher** (tab : tableau[MAX] d'entier, v : entier, deb: entier, fin : entier) : booléen
variable

 m : entier

Début

 Si (deb > fin) alors

 retourner FAUX

 finSi

 m \leftarrow (deb+fin) div 2

 si (**tab[m] == v**) alors

 retourner VRAI

 sinon si (**tab[m] > v**) alors

 retourner **chercher** (tab,v,deb,m-1)

 sinon

 retourner **chercher** (tab,v,m+1,fin)

 finSi

FinFonction

Overview

1. PRÉSENTATION DU COURS
2. TYPES DE DONNEES COMPOSEES
3. POINTEURS
4. SOUS PROGRAMMES
5. RECURSIVITE
6. ALGORITHMES DE TRI
7. FICHIERS

Chapitre 6

ALGORITHMES DE TRI

Introduction

- Trier = ranger dans un certain ordre
- Généralement → un tableau
- Ordre = croissant ou décroissant ou **AUTRE**
- Plusieurs méthodes de tri classées
 - Simplicité
 - Efficacité
 - Les deux ensemble

Tris courants

1. Tri par sélection
2. Tri par insertion
3. Tri à bulles
4. Tri par fusion
5. Tri rapide

Tri par sélection

- Principe
 - Déterminer successivement l'élément devant se retrouver en 1^{ère}, en 2^{ème} position, etc.
 - Croissant → Plus petit des éléments restants, et ainsi de suite.
- Méthode
 - parcourt du tableau gauche → droite
 - A chaque position **i**, on place le **plus petit** élément qui se trouve dans le sous tableau droit (entre les positions i et N)

Algorithme

Procédure triSelection (**E-S** tab : Tableau[]d'entier , **E** N : entier)

Variable i, j, imin : entier

début

Pour i \leftarrow 0 à N-2 **Faire**

*/*on cherche l'indice du plus petit élément entre i et N-1 et on range l'élément à la position i */*

imin \leftarrow i

Pour j \leftarrow i+1 à N-1 **Faire**

Si (tab[j] < tab[imin]) **Alors**

imin \leftarrow j

FinSi

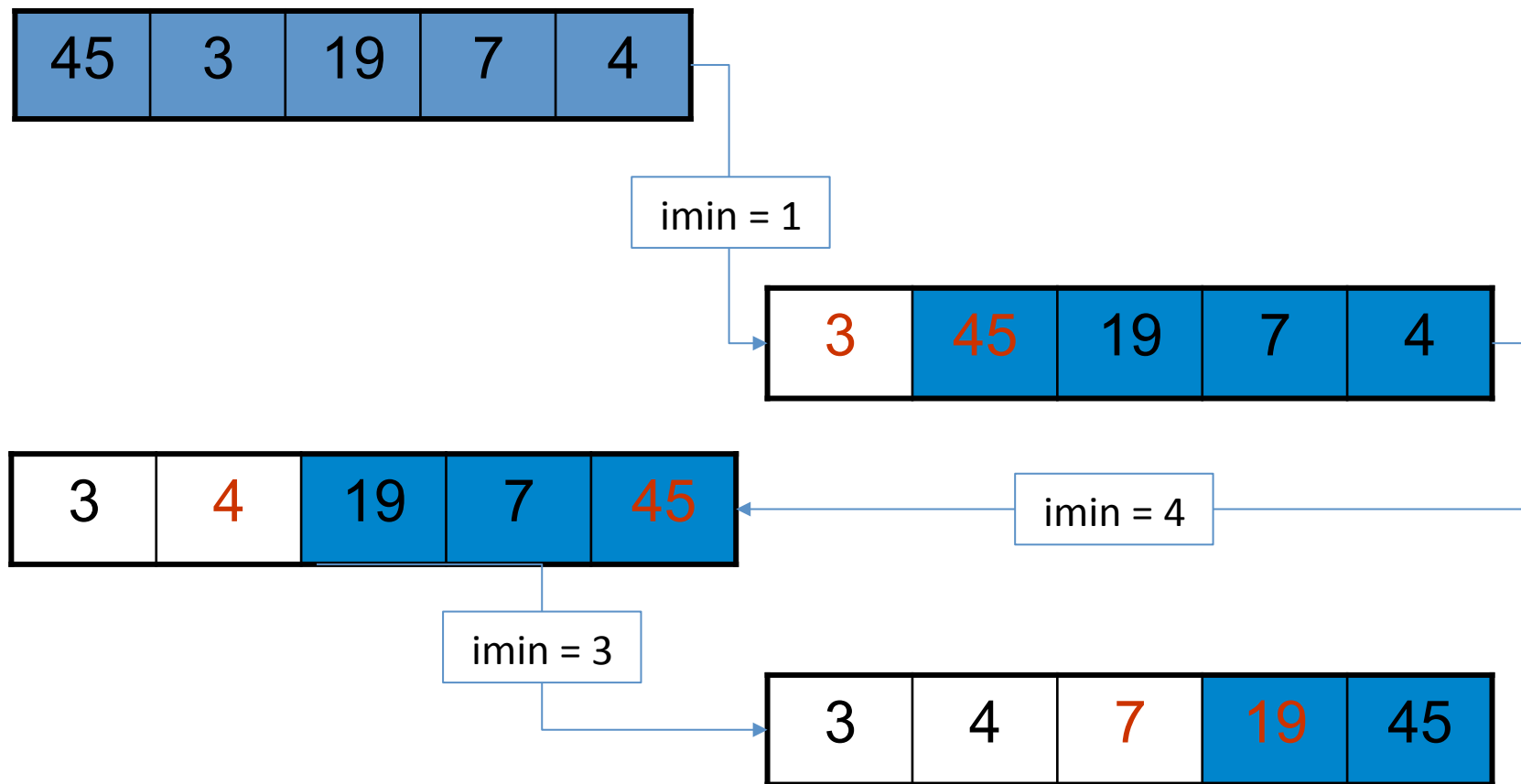
FinPour

echanger(tab[i], tab[imin])

FinPour

FinProcédure

Exemple



Fonctionnement

- **Boucle principale** : prenons comme point de départ le premier élément, puis le second, etc, jusqu'à l'avant dernier.
- **Boucle secondaire** : à partir de ce point de départ **mouvant**, recherchons jusqu'à la fin du tableau quel est le plus petit élément. Une fois que nous l'avons trouvé, nous l'échangeons avec le point de départ.

Tri par insertion

- Principe
 - On procède par étape: à l'étape **i**, on insère le **i^{ème}** élément entre les positions **1** et **i** en sachant que les **i-1** premiers éléments sont déjà triés
- Méthode
 - parcourt tableau **gauche** → **droite** à partir du 2^{ème}
 - A chaque position **i**, on décale pour placer le **i^{ème}** élément à la bonne place dans le sous tableau gauche

Algorithme

procédure triInsertion (**E-S** tab : Tableau[] d'Entier , E N : Entier)

Variable i, j : Entier

tmp : Entier

Début

Pour i \leftarrow 1 à N-1 **Faire**

tmp \leftarrow tab[i]

*/*recherche séquentielle de la position d'insertion et décalages*/*

j \leftarrow i

Tant que ((j > 0) **ET** (tab[j-1] > tmp))**Faire**

tab[j] \leftarrow tab[j-1] // décalage

j \leftarrow j-1

FinTQ

tab[j] \leftarrow tmp // insertion

FinPour

FinProcédure

Exemple

i=1	45 3 19 7 4	3 45 19 7 4
i=2	3 45 19 7 4	3 19 45 7 4
i=3 tmp=7	3 19 45 7 4	3 19 45 45 4
	3 19 19 45 4	3 7 19 45 4
i=4	3 7 19 45 4	3 7 19 45 45
	3 7 19 19 45	3 4 7 19 45

Fonctionnement

- **Boucle externe** : point de départ = indice 2nd élément, puis le suivant, etc, jusqu'au dernier. Gardons la valeur du tableau à l'indice i
- **Boucle interne** : à partir de ce point de départ **mouvant**, recherchons (en décalant vers la droite) jusqu'au début du tableau la bonne place de l'élément. Une fois que nous l'avons trouvé, nous copions l'élément gardé.

Tri à bulles

- Principe
 - déplacer les petits éléments vers le début du tableau et les grands vers la fin du tableau en effectuant des échanges successifs
- Méthode
 - A chaque étape i , on parcourt le tableau à partir de la fin en comparant les éléments consécutifs deux à deux et en les échangeant s'ils ne sont pas dans le bon ordre. Ainsi, on range à la position i le plus petit élément entre i et N .

Algorithme

procédure triABulles(E-S tab : Tableau[] d'Entier , E N : Entier)

Variable

i, j : Entier

Début

Pour i \leftarrow 0 à N - 2 **Faire**

Pour j \leftarrow N-1 à i+1 **par pas de - 1 Faire**

Si (tab[j] < tab[j - 1]) **alors**

echanger(tab[j] , tab[j-1])

FinSi

FinPour

FinPour

FinProcédure

Exemple

45	3	19	7	4
----	---	----	---	---

$i \leftarrow 0$	$j \leftarrow 4$	$4 < 7$	45	3	19	7	4	\rightarrow	45	3	19	4	7
	$j \leftarrow 3$	$4 < 19$	45	3	19	4	7	\rightarrow	45	3	4	19	7
	$j \leftarrow 2$	$4 < 3$	45	3	4	19	7	\rightarrow					
	$j \leftarrow 1$	$3 < 45$	45	3	4	19	7	\rightarrow	3	45	4	19	7
$i \leftarrow 1$	$j \leftarrow 4$	$7 < 19$	3	45	4	19	7	\rightarrow	3	45	4	7	19
	$j \leftarrow 3$	$7 < 4$	3	45	4	7	19	\rightarrow					
	$j \leftarrow 2$	$4 < 45$	3	45	4	7	19	\rightarrow	3	4	45	7	19
$i \leftarrow 2$	$j \leftarrow 4$	$19 < 7$	3	4	45	7	19	\rightarrow					
	$j \leftarrow 3$	$7 < 45$	3	4	45	7	19	\rightarrow	3	4	7	45	19
$i \leftarrow 3$	$j \leftarrow 4$	$19 < 45$	3	4	7	45	19	\rightarrow	3	4	7	19	45

Fonctionnement

- **Boucle externe** : point de départ = indice 1^{er} élément, puis le suivant, etc, jusqu'au avant dernier. Gardons la valeur du tableau à l'indice i
- **Boucle interne** : à partir du dernier élément jusqu'au suivant du point de départ, si 2 éléments consécutifs ne sont pas dans le bon ordre on les échange.

Tri par fusion

- Principe du « diviser pour régner »
 - **Diviser.** Diviser le tableau de n éléments à trier en deux sous tableaux de $n/2$ éléments.
 - **Régner.** Trier les deux sous tableaux récursivement.
 - **Combiner.** Fusionner les deux sous-tableaux triés.
- Méthode

Algorithme

procédure triFusion(**E-S** tab : **tableau** [] **d'**Entier, **E** deb: Entier, fin : Entier)

Variable

milieu : Entier

Début

si (deb < fin) **alors**

milieu = (deb + fin) div 2

triFusion(tab,deb,milieu)

triFusion(tab,milieu+1,fin)

fusionner(tab,deb,milieu,fin)

FinSi

FinProcédure

Algorithme de la fusion

Procédure fusionner(E-S t : tableau [] d'Entier, E deb: Entier, milieu: Entier, fin : Entier)

Variable

temp : tableau [fin – deb + 1] d'Entier //temp est un tableau temporaire dans lequel on met le résultat de la fusion

i, j : Entier // indice de l'élément courant dans t[deb .. milieu] et indice de l'élément courant dans t[milieu+1 .. fin]

k : Entier // indice de la position d'insertion dans temp

Début

i ← deb j ← milieu+1 k ← deb

tant que (i <= milieu) et (j <= fin) faire

si (t[i] < t[j]) alors

temp[k] ← t[i]

i ← i+1

sinon

temp[k] ← t[j]

j ← j+1

FinSi

k ← k+1

Fintq

....

FinProcédure

Algorithme de la fusion

Procédure fusionner(E-S t : tableau [] d'Entier, Edeb: Entier, milieu: Entier, fin: Entier)

Variable

temp : tableau [fin – deb + 1] d'Entier //temp est un tableau temporaire dans lequel on met le résultat de la fusion

i, j : Entier // indice de l'élément courant dans t[deb .. milieu] et indice de l'élément courant dans t[milieu+1 .. fin]

k : Entier // indice de la position d'insertion dans temp

Début

....

tant que (i <= milieu) faire // on copie dans temp le reste des éléments de t[deb .. milieu]

temp[k] \leftarrow t[i]

i \leftarrow i+1

k \leftarrow k+1

fintq

tant que (j <= fin) faire // on copie dans temp le reste des éléments de t[milieu+1 .. fin]

temp[k] \leftarrow t[j]

j \leftarrow j+1

k \leftarrow k+1

fintq

Pour k \leftarrow deb à fin Faire // on recopie le résultat dans le tableau original

T[k] \leftarrow temp[k]

FinPour

FinProcédure

Algorithme de la fusion

Procédure fusionner(**E-S** t : **tableau** [] d'Entier, E deb: Entier, milieu: Entier, fin : Entier)

Variable

temp : **tableau** [fin – deb + 1] d'Entier //temp est un tableau temporaire dans lequel on met le résultat de la fusion
i, j : Entier // indice de l'élément courant dans t[deb .. milieu] et indice de l'élément courant dans t[milieu+1 .. fin]
k : Entier // indice de la position d'insertion dans temp

Début

i ← deb j ← milieu+1 k ← deb

tant que (i <= milieu) **et** (j <= fin) **faire**

si (t[i] < t[j]) **alors**

 temp[k] ← t[i]

 i ← i+1

sinon

 temp[k] ← t[j]

 j ← j+1

FinSi

 k ← k+1

Fintq

tant que (i <= milieu) **faire** *// on copie dans temp le reste des éléments de t[deb .. milieu]*

 temp[k] ← t[i]

 i ← i+1

 k ← k+1

fintq

tant que (j <= fin) **faire** *// on copie dans temp le reste des éléments de t[milieu+1 .. fin]*

 temp[k] ← t[j]

 j ← j+1

 k ← k+1

fintq

Pour k ← deb à fin **Faire** *// on recopie le résultat dans le tableau original*

 T[k] ← temp[k]

FinPour

FinProcédure

Exemple $\text{tab} =$

45	3	19	7	4
----	---	----	---	---

$d \leftarrow 0$	$f \leftarrow 4$	$d < f$	$m \leftarrow 2$		45	3	19			7	4
$d \leftarrow 0$	$f \leftarrow 2$	$d < f$	$m \leftarrow 1$	45	3		19				
$d \leftarrow 0$	$f \leftarrow 1$	$d < f$	$m \leftarrow 0$	45		3					
$d \leftarrow 0$	$f \leftarrow 0$	$d < f$	faux	45							
$d \leftarrow 0$	$f \leftarrow 0$	$d < f$	faux			3					
fusionner					3	45					
						3	19	45			
$d \leftarrow 3$	$f \leftarrow 4$	$d < f$	$m \leftarrow 3$						7		4
$d \leftarrow 3$	$f \leftarrow 3$	$d < f$	faux						7		
$d \leftarrow 4$	$f \leftarrow 4$	$d < f$	Faux								4
fusionner									4	7	
fusionner	$i \leftarrow 0$	$j \leftarrow 3$	$k \leftarrow 0$	$\text{temp} \leftarrow$	3						
	$i \leftarrow 1$	$j \leftarrow 3$	$k \leftarrow 1$	$\text{temp} \leftarrow$	3	4					
	$i \leftarrow 1$	$j \leftarrow 4$	$k \leftarrow 2$	$\text{temp} \leftarrow$	3	4	7				
	$i \leftarrow 1$	$j > f$	$k \leftarrow 3$	$\text{temp} \leftarrow$	3	4	7	19			
	$i \leftarrow 2$	cp	$k \leftarrow 4$	$\text{temp} \leftarrow$	3	4	7	19	45	$\rightarrow \text{tab}$	

Fonctionnement

- Récursivement
 - Tableau divisé en 2 sous-tableaux
 - Algorithme appliqué à chaque sous-tableau
 - Fusion des sous-tableaux ordonnés

Tri rapide (quicksort)

- Principe du « diviser pour régner »
 - **Diviser**
 - Choix pivot au hasard (souvent $t[\text{deb}]$)
 - Partition en 2 sous tableaux tels que les éléments
 - Avant pivot $<$ pivot
 - Après pivot \geq pivot
 - **Régner**
 - Pivot est à sa place définitive
 - Trier de même façon les 2 tableaux placés avant et après lui.
 - **Combiner**
 - Sous-tableaux triés sur place
 - Conclusion: le tableau est trié en totalité.

Algorithme

procédure triRapide(**E-S** tab : tableau [] d'Entier, **E** deb : Entier, fin : Entier)

Variable

placePivot : Entier *//indice du pivot après partitionnement*

Début

Si (deb < fin) **Alors**

partitionner(tab, deb, fin, placePivot)

triRapide(tab, deb, placePivot-1)

triRapide(tab, placePivot+1, fin)

FinSi

FinProcédure

Algorithme du partitionnement

Procédure partitionner(**E-S** t : **tableau** [] d'Entier, E deb: Entier, fin : Entier, **E-S** placePivot : Entier)

Variable

pivot: Entier

i : Entier

Début

pivot \leftarrow t[deb]

placepivot \leftarrow deb

Pour i \leftarrow deb+1 à fin

si (t[i] < pivot) **alors**

 placepivot \leftarrow placePivot +1

 échanger(t[placePivot] , t[i])

FinSi

Finpour

échanger(t[deb], t[placePivot])

FinProcédure

Exemple

45	3	19	7	4
----	---	----	---	---

piv=45, pp=0	i←1	3 < 45	pp = 1	45	3	19	4	7
	i←2	19 < 45	pp = 2	45	3	19	4	7
	i←3	4 < 45	pp = 3	45	3	19	4	7
	i←4	7 < 45	pp = 4	45	3	19	4	7
	FinSi			7	3	19	4	45
piv=7, pp=0	i←1	3 < 7	pp = 1	7	3	19	4	
	i←2	19 < 7	faux					
	i←3	4 < 7	pp = 2	7	3	4	19	
	FinSi			4	3	7	19	
piv=4, pp=0	i←1	3 < 4	pp = 1	4	3			
	FinSi			3	4			
piv=7, pp=0	i←1	19 < 7	faux			7	19	
	FinSi					7	19	

Fonctionnement

- **Partitionnement du tableau** : choisir un pivot aléatoire, par exemple le premier élément
- **Récurtivité** : appliquer le tri sur les tableaux avant la valeur pivot et après cette dernière
- Pivot étant toujours à la bonne place ➔ à la fin le tableau est trié

A voir aussi ...

Overview

1. PRÉSENTATION DU COURS
2. TYPES DE DONNEES COMPOSEES
3. POINTEURS
4. SOUS PROGRAMMES
5. RECURSIVITE
6. ALGORITHMES DE TRI
7. FICHIERS

Chapitre 7

FICHIERS

Introduction

- **Définition** : ensemble d'informations
 - organisées et
 - stockées sur une mémoire de masse
 - disque dur,
 - disquette,
 - bande magnétique,
 - CD-ROM

En Langage C

- **Définition** : une suite d'octets
 - Les informations contenues dans un fichier ne sont pas forcément de même type
 - un char,
 - un int,
 - une structure, ...
 - Un pointeur fournit l'adresse d'une information quelconque

Types de fichiers (1)

- **Texte ou à accès séquentiel**
 - Données sans aucune structure particulière
 - Lecture caractère par caractère ou ligne par ligne.
 - séquentiel : parcourir le fichier depuis le début pour accéder à une donnée particulière.
 - ne contient pas forcément que du texte.
 - terme "texte" employé car le fichier est vu comme une suite de caractères
 - pas forcément lisibles

Types de fichiers (2)

- **binaire ou à accès direct** (aléatoire)
 - organisés par une structure sous-jacente.
 - pas de caractères mais d'enregistrements (struct)
 - En connaissant la taille d'un enregistrement, on peut se rendre directement à un endroit précis du fichier ➔ accès direct.

Manipulation (1)

- **Opérations possibles**

- Créer
- Ouvrir
- Fermer
- Lire
- Ecrire
- Détruire
- Renommer.

➔ primitives dans bibliothèque standard "**stdio.h**"

Manipulation (2)

- Manipuler fichier
 - définir un pointeur
 - pointeur fournit l'adresse d'une cellule donnée
- Déclaration
 - **FILE *fichier; /*majuscules obligatoires FILE*/**
 - déclaration fichier **AVANT** celle des autres variables

Ouverture

- Fonction qui retourne un FILE
- Prototype fonction : **FILE *fopen(char *nom, char *mode);**
 - 2 chaines en paramètres
 - **nom**: nom du fichier sur le disque, exemple : "**c:\toto.txt**"
 - **mode** d'ouverture dépend
 - TEXTE
 - BINAIRE

Mode ouverture : texte

MODE	DESCRIPTION
"r" ou "rt"	lecture seule
"w" ou "wt"	écriture seule (destruction de l'ancienne version si elle existe)
"w+" ou "wt+"	lecture/écriture (destruction ancienne version si elle existe)
"r+" ou "rt+"	lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version
"a+" ou "at+"	lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version, le pointeur est positionné à la fin du fichier.

- A l'ouverture, le pointeur est positionné au début du fichier (sauf "a+" et "ab+")
- Il retourne NULL si erreur d'ouverture du fichier

Mode ouverture : binaire

MODE	DESCRIPTION
"rb"	lecture seule
"wb"	écriture seule (destruction de l'ancienne version si elle existe)
"wb+"	lecture/écriture (destruction ancienne version si elle existe)
"rb+"	lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version
"ab+"	lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version, le pointeur est positionné à la fin du fichier.

- A l'ouverture, le pointeur est positionné au début du fichier (sauf "a+" et "ab+")
- Il retourne NULL si erreur d'ouverture du fichier

Exemple

```
FILE *fichier;  
//ouverture du fichier en mode écriture  
fichier = fopen("exemple.txt", "w");  
  
if(fichier==NULL){  
    printf("fichier non ouvert\n");  
}  
else{  
    /* suite du programme */  
    /* parmi lequel lecture des valeurs dans le fichier */  
}
```


Fermeture

- Toujours fermer fichier à la fin d'une session
- Utilisation de fonction **fclose** → un entier.
- Prototype : **int fclose (FILE *fichier);**
- Retourne
 - 0 si la fermeture s'est bien passé,
 - EOF en cas d'erreur.
- Exemple

```
FILE *fichier;  
Fichier = fopen("c:\exemple.txt", "w");  
/* Ici instructions de traitement */  
fclose (fichier);
```

Écriture (1)

- Caractère : **int putc (char c, FILE *fichier)**
 - Écriture de **c** à la position courante
 - Déplacement = **une case mémoire**
 - Retourne **EOF** en cas d'erreur
- Entier : **int putw(int n, FILE *fichier)**
 - Écriture de **n** à la position courante
 - Déplacement = **nombre de cases correspondant à la taille d'un entier.**
 - Retourne **n** si écriture réussie

Écriture (2)

- Chaîne de caractères : **int fputs (char *chaine, File *fichier)**
 - Écriture de **chaine** à la position courante
 - Déplacement = **longueur de la chaine** (**'\0'** *n'est pas rangé dans le fichier*).
 - Retourne **EOF** en cas d'erreur
 - Exemple:

```
FILE *fichier;  
Fichier = fopen("c:\exemple.txt", "w");  
fputs("Bonjour", fichier);  
fclose(fichier);
```

Écriture (3)

- Octets : **int fwrite (void *p, int taille_bloc, int nb_bloc, FILE *fichier)**
 - Écriture **nb_bloc x taille_bloc** octets lus
 - Déplacement = **nombre octets écrits**
 - Retourne **EOF** en cas d'erreur
 - p = adresse et son type non important
 - Exemple:

```
FILE *fichier;  
Fichier = fopen("c:\exemple.txt", "w");  
fwrite(p, 4, 5, fichier);  
fclose(fichier);
```

Écriture fichier ASCII

- **int fprintf (FILE *fichier, char *format, list d'expressions)**
 - Écriture de **list expressions**
 - Déplacement = **nombre octets écrits**
 - Retourne **EOF** en cas d'erreur
 - Exemple:

```
FILE *fichier;  
int n = 45;  
fichier = fopen("c:\exemple.txt", "w");  
fprintf(fichier, "%s", "il fait beau");  
fprintf(fichier, "%d", n);  
fclose(fichier);
```

Lecture caractère

- Fonction : **int getc(FILE *fichier)**
 - Lecture d'un caractère
 - Déplacement = une case mémoire
 - Retourne entier (chr(n)) si réussi
 - Retourne EOF en cas d'erreur

- Exemple

```
FILE *fichier;  
char c;  
fichier = fopen("c:\exemple.txt", "w");  
c = (char) getc(fichier);  
.....  
fclose(fichier);
```

Lecture entier

- Fonction : **int getw (FILE *fichier)**
 - Lecture d'un **entier** à la position courante
 - Déplacement = **nombre de cases correspondant à la taille d'un entier**.
 - Retourne **l'entier lu** si réussie
 - Retourne **EOF** en cas d'erreur
- Exemple

```
FILE *fichier;  
int n;  
fichier = fopen("c:\\exemple.txt", "w");  
n = getw(fichier);  
.....  
fclose(fichier);
```

Lecture plusieurs caractères

- Fonction : **char *fgets(char *chaine, int n, FILE *fichier)**
 - Lecture de **n-1 caractère**
 - Déplacement = **(n-1) x cases mémoire**
 - chaine=caractères lus + '\0'
 - Retourne **EOF** en cas d'erreur
- Exemple

```
FILE *fichier;  
char *chaine;  
fichier = fopen("c:\exemple.txt", "w");  
fgets(chaine, 15, fichier);  
.....  
fclose(fichier);
```


Lecture plusieurs blocs

- Fonction : **int fread (void *p, int taille_bloc, int nb_bloc, FILE *fichier)**
 - Lecture de **nb_bloc x taille_bloc**
 - Déplacement = **nombre octets lus**
 - Retourne **nombre d'octets lus** ou **0** en fin de fichier
 - Retourne **EOF** en cas d'erreur
- Exemple

```
FILE *fichier;  
char *chaine;  
fichier = fopen("c:\exemple.txt", "w");  
int n = fread(p, 15, 10, fichier);  
  
.....  
fclose(fichier);
```

Lecture plusieurs expressions

- **int fscanf (FILE *fichier, char *format, liste d'adresses)**
 - Lecture de **liste d'adresses**
 - Déplacement = **nombre octets lus**
 - Retourne **nombre d'octets lus** ou **0** en fin de fichier
 - Retourne **EOF** en cas d'erreur
 - Exemple:

```
FILE *fichier;  
fichier = fopen("c:\exemple.txt", "w");  
fscanf (fichier, "%s", ???);  
fscanf (fichier, "%d", ???);  
fclose(fichier);
```

Fonctions utiles ...

- Destruction
 - Prototype : *int remove(char *nom)*
 - Retourne 0 si réussi
- Renommage:
 - Prototype : *int rename (char* anciennom, char *nouveaunom)*
 - Retourne 0 si réussi

Erreurs à gérer

- **fopen**
 - retourne le pointeur NULL si erreur
 - Exemple : impossibilité d'ouvrir le fichier
- **fgets**
 - retourne le pointeur NULL en cas d'erreur ou si la fin du fichier est atteinte.
- **feof (FILE *fichier)**
 - retourne 0 tant que la fin du fichier n'est pas atteinte.