1

Algorithmique/programmation renforcés
Langage C
1ère année Licence Informatique

PHILIPPE HUNEL

https://ecursus.univ-antilles.fr/

CLÉ : **INFO2526**

1

2

Objectif

- Familiariser les étudiants avec les techniques et outils permettant
  - de concevoir et de comprendre,
  - de réaliser puis traduire
  - des programmes en langage C
- ⇒ rigueur de la démarche scientifique
- 14 heures de cours / TD
- 14 heures de TP

2

3

1.1 – Qu'est-ce que le C ?

- **Langage de bas niveau** (proche du matériel) mais **portable**.
- **Utilisations** : Systèmes embarqués, noyaux de systèmes d'exploitation (Linux, Windows), drivers, applications performantes.
- **Caractéristiques** :
  - **Rapidité** : Compilé en code machine.
  - **Contrôle fin** : Gestion manuelle de la mémoire, pointeurs.
  - **Bibliothèque standard riche** (stdio.h, stdlib.h, etc.).

3

4

1.2 – Premier Programme en C

```c
#include <stdio.h>
int main(){
    printf("Bonjour super beau prof");
        return 0;
}
```

- **Explication** :
  - #include <stdio.h> : Inclut la bibliothèque standard pour l'entrée/sortie.
  - int main() : Point d'entrée du programme.
  - printf() : Affiche du texte.
  - return 0 : Indique une exécution réussie.

4

2 – Bases du Langage C

5

- Les variables doivent faire l'objet d'une déclaration de type de la forme:

  type liste_des_variables ;

- Le type d'une donnée détermine :
  - l'ensemble des valeurs admissibles,
  - le nombre d'octets à réserver en mémoire
  - l'ensemble des opérateurs qui peuvent y être appliqués

5

2.1 – En C : types simples

6

- Le langage C est faiblement typé (*i.e. flexible*).
- ⇒ permet d'utiliser des opérandes de différents types dans un même calcul
- Le risque :
  - Conversion de type automatique
  - Arrondis
  - Résultats incorrects, inexplicables

6

2.1 – En C : types simples

- Ensembles de nombres et leur représentation
  - En mathématiques : $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$
- En informatique :
  - Ordinateur ne peut traiter aisément que des nombres entiers d'une taille limitée
  - Utilisation du système binaire pour calculer et sauvegarder ces nombres
  - Valeurs correctement approchées des entiers très grands, des réels ou des rationnels à partie décimale infinie sont obtenues par des ruses de calcul et de représentation

2.1 – En C : types simples

- Obligation du programmeur :
  - Il n'est pas nécessaire qu'il connaisse les détails des méthodes de codage et de calcul
  - Il doit cependant pouvoir :
    - choisir un type numérique approprié à un problème donné
    - choisir un type approprié pour la représentation sur l'écran
    - prévoir le type résultant d'une opération entre différents types numériques;
    - prévoir et optimiser la précision des résultats intermédiaires au cours d'un calcul complexe;

2.1 – En C : types simples

9

- Exemple :
  - Supposons que le type choisi ne propose que 5 positions décimales

$(1.00001*10^7 + 33) – 1*10^7 = 100$

$(1.00001*10^7 – 1*10^7) + 33 = 133$

2.2 – En C : types entiers

10

- Caractéristiques des types numériques entiers

Définition	Description	Val min	Val max	Nb. Octets
char	Caractère	-128	127	1
short	Entier court	-32768	32767	2
int	Entier standard	-2147483648	2147483647	4
long	Entier long	-2147483648	2147483647	4

2.3 – En C : types rationnels

11

- Souvent appelés des flottants (*virgule flottante*)

  <+|-> <mantisse> * 10<exposant>

- **<+|->** : signe positif ou négatif du nombre
- ***<mantisse>*** décimal positif avec un seul chiffre devant la virgule
- *<exposant>* : entier relatif

11

2.3 – En C : types rationnels

12

- Types C :
  - **float** sur 4 octets
  - **double** sur 8 octets
  - **long double** sur 12 octets

Exemples :
123.449997
1.234500e+02

Type	Taille (octets)	Plage de valeurs	Exemple
`float`	4	1.2E-38 à 3.4E+38	`float pi = 3.14;`
`double`	8	2.3E-308 à 1.7E+308	`double x = 1.23;`

12

13

2.2 – En C : types entiers

- Remarque :
  - Le type int est le type de base pour les calculs avec les entiers
    - Le codage du type int est dépend de la machine :
      - short = 2 octets ; **int = 4 octets** ; long = 4 octets
      - short = 2 octets ; **int = 2 octets** ; long = 4 octets

13

14

2.2 – En C : types entiers

- Les modificateurs signed/unsigned

Définition	Description	Val min	Val max	Nb. Octets
unsigned char	Caractère	0	255	1
unsigned short	Entier court	0	65535	2
unsigned int	Entier standard	0	4294967295	4
unsigned long	Entier long	0	4294967295	4

14

2.4 – En C : Opérateurs

15

- Arithmétiques : +, –, *, /, % (modulo)
- Logiques : && (ET), || (OU), ! (NON)
- Comparaison : ==, !=, <, >, <=, >=

2.5 – En C : lecture - écriture

16

- Ecriture `printf` : syntaxe

```
int printf(const char *format, ...);
```

  - format : Chaîne de formatage contenant des spécificateurs (%d, %s, etc.).
  - ... : Liste d'arguments correspondants aux spécificateurs.
  - Retourne : Le nombre de caractères affichés.

17

2.5 – En C : lecture - écriture

- Ecriture `printf` : Spécificateurs de Format Courants

Spécificateur	Type	Exemple
`%.2f`	Flottant (2 décimales)	`printf("%.2f", 3.14159);` → `3.14`
`%%`	Affiche `%`	`printf("%%");` → `%`
`%c`	Caractère (`char`)	`printf("%c", 'A');` → `A`
`%d`	Entier (`int`)	`printf("%d", 42);` → `42`
`%f`	Flottant (`float`)	`printf("%f", 3.14);` → `3.140000`
`%p`	Pointeur	`printf("%p", &variable);` → `0x7ffd42a1b2ac`
`%s`	Chaîne (`char*`)	`printf("%s", "Bonjour");` → `Bonjour`

17

18

2.5 – En C : lecture - écriture

- Ecriture `printf` : Flags de Formatage

Flag	Signification	Exemple
-	Alignement à gauche	`%-5d` → `42` (si 42)
0	Remplit avec des zéros	`%05d` → `00042`
+	Affiche le signe (+ ou -)	`%+d` → `+42` ou `-42`

18

19

2.5 – En C : lecture - écriture

- Ecriture `printf` : exemples

```
3   int nombre = 42;
4   printf("Nombre: |%5d| |%-5d| |%05d|\n", nombre, nombre, nombre);
```

```
Nombre: |   42| |42   | |00042|
```

- Précision : Nombre de décimales pour les flottants ou nombre maximal de caractères pour les chaînes.

```
printf("%.2f\n", 3.14159);
printf("%.5s\n", "Bonjour");
```

```
3.14
Bonjo
```


19

20

2.5 – En C : lecture - écriture

- Ecriture `printf` : exemple complet

```c
#include <stdio.h>

int main() {
    int age = 25;
    float taille = 1.754568;
    char initiale = 'A';

    printf("Âge: %d ans, Taille: %.2f m, Initiale: %c\n", age, taille, initiale);

    return 0;
}
```

```
Âge: 25 ans, Taille: 1.75 m, Initiale: A
```


20

2.5 – En C : lecture - écriture

21

- Ecriture `putchar` : syntaxe (permet d'afficher un caractère)

```
int putchar(int c);
```

  - c : Caractère à afficher (passé comme int pour compatibilité avec EOF).
  - Retourne : Le caractère affiché, ou EOF en cas d'erreur.
  - Exemple :

```
putchar('A');  // Affiche 'A'
```


21

2.5 – En C : lecture - écriture

22

- `printf` versus `putchar`

putchar	printf
Affiche **un seul caractère**.	Affiche des **chaînes formatées**.
Plus **rapide** pour les caractères individuels.	Plus **flexible** (nombres, flottants, etc.).
Retourne le caractère ou `EOF`.	Retourne le nombre de caractères affichés.

22

2.5 – En C : lecture - écriture

23

- `scanf` : Lecture Formatée, Syntaxe

```
int scanf(const char *format, ...);
```

  - format : Chaîne de formatage avec des spécificateurs (%d, %f, etc.).
  - ... : Adresses des variables où stocker les valeurs lues (utilisez &).
  - Retourne : Le nombre de variables correctement lues.

23

2.5 – En C : lecture - écriture

24

- `scanf` :

Spécificateur	Type	Exemple
`%d`	Entier (`int`)	`scanf("%d", &age);`
`%f`	Flottant (`float`)	`scanf("%f", &taille);`
`%lf`	Double (`double`)	`scanf("%lf", &pi);`
`%c`	Caractère (`char`)	`scanf(" %c", &car);`
`%s`	Chaîne (`char*`)	`scanf("%49s", nom);`
`%ld`	Long (`long`)	`scanf("%ld", &grand_nombre);`

24

2.5 – En C : lecture - écriture

25

- `scanf` : Pièges Courants
- Oublier & devant les variables :

```
scanf("%d", age);    // ❌ Erreur : doit être &age
```

- Lire une chaîne sans limiter la taille :

```
char nom[10];
scanf("%s", nom);    // ❌ Risque de débordement
```

- // ✅ Correct :

```
scanf("%9s", nom);    // Lit max 9 caractères
```


25

2.5 – En C : lecture - écriture

26

- `scanf` : exemple

```c
#include <stdio.h>

int main() {
    int age;
    float taille;
    printf("Entrez votre âge et taille: ");
    scanf("%d %f", &age, &taille);
    printf("Vous êtes agé de %d ans et votre taille est %f m\n",age,taille);
    return 0;
}
```


26

27

2.5 – En C : lecture - écriture

- `getchar` : Lire un Caractère, Syntaxe

```
int getchar(void);
```

  - Retourne :
    - Le code ASCII du caractère lu (comme int).
    - EOF (-1) en cas d'erreur ou de fin de fichier (Ctrl+D sous Linux/Mac, Ctrl+Z sous Windows).

28

2.5 – En C : lecture - écriture

- `getchar` : exemple

```
int c = getchar();
printf("Vous avez entré: %c\n", c);
```

```
d
Vous avez entré: d
```

```
printf("Entrez une ligne: ");
int c;
while ((c = getchar()) != '\n' && c != EOF) {
    putchar(c);
}
putchar('\n');
```

```
Entrez une ligne: Le prof est beau
Le prof est beau
```

2.4 – En C : Opérateurs

29

- Comparaison des Fonctions

Fonction	Utilisation Typique	Avantages	Inconvénients
`printf`	Affichage formaté (nombres, chaînes, etc.).	Flexible, puissant.	Plus lent pour les caractères individuels.
`putchar`	Affichage caractère par caractère.	Rapide, simple.	Limité aux caractères.
`scanf`	Lecture formatée (utilisateur/fichier).	Pratique pour les données structurées.	Risque de débordement, moins flexible.
`getchar`	Lecture caractère par caractère.	Sûr, simple, permet de traiter les entrées.	Lent pour les grandes entrées.

29

3 – La séquence en C

30

- Chaque instruction se termine par ;
- Toutes les variables utilisées doivent obligatoirement être déclarées au préalable

30

31

3.1 – Déclaratives

- Il est aussi nécessaire de préciser ce que les variables utilisées contiendront comme type de données.
- Il peut s'agir de nombres entiers, de nombres réels, de chaînes de caractères, ...
- Il faut faire précéder la description de l'algorithme par une partie dite **déclarative** où l'on regroupe les caractéristiques des variables manipulées.

31

32

Chapitre 4 : La structure de choix

LS1 - Algorithmique/programmation renforcés ;
langage C - Ph. Hunel
2025-2026

32

33

4.1 - La structure alternative

- La structure alternative se présente en général sous la forme :

```
si expression alors
    première séquence d'instructions
sinon
    deuxième séquence d'instructions
fsi
```


34

4.2. L'alternative en C

- Si se traduit en if
- La condition doit être entre parenthèses
- Sinon en else

si expression **alors**	if (expression) {
séquence d'instructions	séquence d'instructions
sinon	} else {
séquence d'instructions	séquence d'instructions
fsi	}

4.2 - L'alternative en C

35

si expression **alors**
séquence d'instructions
fsi
	if (expression) {
	séquence d'instructions
	}

si expression **alors**
une instruction
sinon
séquence d'instructions
fsi
	if (expression)
	une instruction
	else {
	séquence d'instructions
	}

4.2 - L'alternative en C

36

```
#include <stdio.h>

int NA, NB, reste;
main () {
  printf("Introduisez le nombre de doigts montrés par le joueur A : ");
  scanf("%d",&NA);
  printf("Introduisez le nombre de doigts montrés par le joueur B : ");
  scanf("%d",&NB);
  reste = (NA+NB) % 2;
  if (reste==0) {
     printf("Le joueur A a gagné\n");
  } else {
     printf("Le joueur B a gagné\n");
  }
  printf("Bravo pour le gangnant\n");
}
```

4.3 - Le choix multiple

37

- Supposons que l'on veuille demander à l'utilisateur de choisir dans un menu une des 3 possibilités offertes.
- Le choix présenté ne se limite pas à une alternative (soit - soit).
- Mais plutôt à une expression du type « selon que… »

4.3 - Le choix multiple

38

- En LDA :

```
entier i
lire i
selon que
    i=1  faire bloc1
    ou que i=2  faire bloc2
     ou que i=3  faire bloc3
    autrement écrire "Mauvais choix"
Fselon
```

- **autrement** est comme dans l'alternative facultative

4.5 - Le choix multiple

39

- Peut toujours s'écrire avec des alternatives :

```
entier i
lire i
Si i=1 alors
        bloc1
sinon
        si i=2 alors
                bloc2
        sinon
                si i=3 alors
                        bloc3
                sinon
                        écrire "Mauvais choix"
                Fsi
        Fsi
Fsi
```


4.4 - Le choix multiple en C

40

- La traduction du choix multiple en C est assez restrictive, puisque la valeur de l'expression conditionnant le choix doit être entière (char, short, int).
- L'instruction switch permet de mettre en place une structure d'exécution qui permet des choix multiples parmi des cas de même type et faisant intervenir uniquement des **valeurs constantes entières**.

41

4.4 - Le choix multiple en C

```
switch ( <expression entière> ) {
    case <constante entière>:
     <instruction 1>
     …
     <instruction N>
    …
    default :
     <instruction 1>
     …
     <instruction N>
}
```

ATTENTION : si la dernière instruction n'est pas l'instruction ***break*** les autres « ***case*** » ainsi que le « ***default*** » seront exécutés

41

42

4.4 - Le choix multiple en C

```
#include <stdio.h>
int i;
main() {
  printf("Entrez votre choix : ");
  scanf("%d",&i);
  switch(i) {
  case 1:printf("premier choix\n")
              break;
  case 2:printf("deuxième choix\n");
  case 3:printf("troisième choix\n");
  default:printf("Autre choix que choix 1, 2 ou 3\n");
  }
}
```

Entrez votre choix : 1
premier choix

Entrez votre choix : 2
deuxième choix
troisième choix
Autre choix que choix 1, 2 ou 3

42

43

4.4 - Le choix multiple en C

```
#include <stdio.h>
int i;
main() {
  printf("Entrez votre choix : ");
  scanf("%d",&i);
  switch(i) {
  case 1:printf("premier choix\n");
            break;
  case 2:printf("deuxième choix\n");
            break;
  case 3:printf("troisième choix\n");
            break;
  default:printf("Autre choix que choix 1, 2 ou 3\n");
  }
}
```


43

```
duree : entier
montant : entier
Taux : réel
Lire montant, duree
Si duree = 2 alors
| Si montant < 10 000 alors
| |  taux ← 5 %
| sinon Si montant < 20 000 alors
| |        | taux ← 10 %
| |       sinon Si montant < 25 000 alors
| |        |        |  taux ← 15 %
| |        |        sinon taux ← 17 %
| |        |        FSi
| |        FSi
| FSi
sinon Si montant < 100 000 alors
|        |  taux ←   8 %
|        sinon
|        | taux ← 4 %
|        FSi
FSi
```

Exercice

44

45

Exercice

- Écrire l'algorithme qui permet de calculer le maximum de deux entiers quelconques.

45

46

Correction

```
Titre : Maximum
Variable a ,b, max : entier
Début
   Écrire ("Saisir deux entiers a et b ")
   Lire a
   Lire b
   Si (a > b) alors
         max ⟵ a
   Sinon
      max ⟵ b
   Finsi
   Écrire ('le maximum de ' , a , ' et  de ' , b, ' est : '  , max)
Fin
```


46

47

Exercices

- Nombre positif ou négatif
  - Lire un nombre et afficher s'il est positif, négatif ou nul.
- Pair ou impair
  - Lire un entier et afficher s'il est pair ou impair.
- Majeur ou mineur
  - Lire l'âge d'une personne et afficher si elle est majeure (≥18 ans) ou mineure

47

48

Exercice

- Écrire l'algorithme qui permet de déterminer le salaire mensuel d'un commercial sachant que ce salaire comporte un montant fixe de 4000 € et une commission qui représente 5% du chiffre d'affaires réalisé par mois si ce chiffre est < 30000 et de 10 % dans le cas contraire .

48

49

Correction

… Suite de l'algorithme

```
Si  (CA < 30000) alors
    Com ⟵ CA * 0.05
Sinon
    Com ⟵ CA * 0.1
FSI
Sal ⟵ Com + 4000
Écrire ('Le salaire mensuel est de : ', Sal , '€')
FIN
```


50

Exercice

- Compliquons un peu l'énoncé:
- La commission est calculée de la manière suivante :
  - Commission = 15% du CA quand CA > 100000
  - Commission = 10% du CA quand 30000 < CA < = 100000
- Dans le cas contraire pas de commission
- Écrire l'algorithme qui permet de déterminer le salaire mensuel.

51

Correction

… Suite de l'algorithme

```
    Si  (CA > 100000) alors
        Com ⟵ CA * 0.15
    Sinon
        Si (CA > 30000) alors
         Com ⟵ CA * 0.1
        Sinon
         Com ⟵ 0
        Finsi
    Finsi
    Sal ⟵ Com + 4000
    Écrire ('Le salaire mensuel est de : ', Sal , '€')
FIN
```


51

52

Compilateur C

52

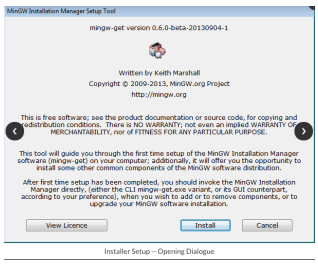
Installation

53

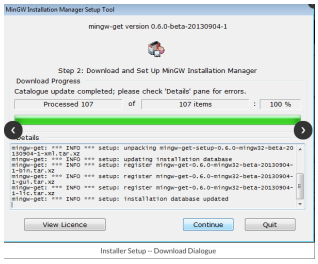
1. Télécharger MinGW

   https://sourceforge.net/projects/mingw/files/latest/download

2. Exécuter le fichier mingw-get-setup



Installer Setup – Opening Dialogue



Installer Setup – Download Dialogue

Sélectionnez juste le package de base et dans le menu Installation sélectionnez « Apply change »

Installation

54

1. Modifier la variable d'environnement Path en ajoutant le chemin : c:\MinGW\bin

2. Rajouter l'extension C/C++ dans votre Visual Studio code

55

Chapitre 5 : La structure répétitive

55

56

5.1 - La boucle " tant que "

- Fait répéter une séquence d'instructions aussi longtemps qu'une condition est **VRAI**
- En LDA :

```
Tant que condition faire
    séquence d'instructions
Ftq
```


56

57

5.2 - La boucle " pour faire"

- Lorsque le nombre d'itération est connu
- Exemple de la table de multiplication
- En LDA :

pour var_de_crt ← prem_val à dern_val faire

séquence d'instructions

fpour

57

58

5.3 – Les boucles en C

- En C, la boucle tant que se traduit par

```
while ( <expression logique>)
    <instruction>
```

- ou

```
while ( <expression logique>) {
        <séquence d'instructions>
}
```


58

5.3 – Les boucles en C

59

```
#include <stdio.h>
int m,n,a,b,r,PGCD;
main(){
  printf("Nous allons calculer le PGCD de 2 nombres\n");
  printf("Introduisez le premier nombre : ");
  scanf("%d",&m);
  printf("Introduisez le second nombre : ");
  scanf("%d",&n);
  a=m;   b=n;
  while (b!=0) {
    r=a % b;
    a=b;     b=r;
  }
  PGCD=a;
  printf("Le PGCD de %d et %d est %d\n",m,n,PGCD);
}
```


5.3 – Les boucles en C

60

- En C, la boucle pour se traduit par

for (**exp_init ; exp_cond ; exp_evol**)

    <instruction>

- Ou

for (**exp_init ; exp_cond ; exp_evol**) {

    <séquence d'instructions>

}

5.3 – Les boucles en C

61

- **exp_init** :
  - est une instruction d'initialisation ; elle est exécutée avant l'entrée dans la boucle
- **exp_cond** :
  - est la condition de continuation ; elle est testée à chaque passage, y compris lors du premier ; l'instruction ou les instructions composant le corps du for sont répétées tant que le résultat de l'expression **exp_cond** est VRAI
- **exp_evol** :
  - Est une instruction de rebouclage ; elle fait avancer la boucle ; elle est exécutée en fin de boucle avant le nouveau test de passage.

61

5.3 – Les boucles en C

62

```
#include <stdio.h>
main(){
  int i,n;
  printf("Quelle table                    );
  scanf("%d",&n);
  for (i=1;i<=10;i++)
    printf("%d fois %d font %d\n",n,i,n*i);
}
```

++ : opérateur d'incrémentation

i++ ⇔i=i+1

-- : opérateur de décrémentation

i-- ⇔i=i-1

62

5.3 – Les boucles en C

63

++ : opérateur d'incrémentation

i++ ⇔i=i+1

-- : opérateur de décrémentation

i-- ⇔i=i-1

```
#include <stdio.h>

  main() {
    int i;
    for (i=1;i<=10;i++)
      printf("%d : The teacher is the
  best\n",i);
    printf("-------------------------\n");
    for (i=10;i>=1;i--)
      printf("%d : The teacher is the first
  best\n",i);
  }
```


64

```
1 : The teacher is the best
2 : The teacher is the best
3 : The teacher is the best
4 : The teacher is the best
5 : The teacher is the best
6 : The teacher is the best
7 : The teacher is the best
8 : The teacher is the best
9 : The teacher is the best
10 : The teacher is the best
-------------------------
10 : The teacher is the first  best
9 : The teacher is the first  best
8 : The teacher is the first  best
7 : The teacher is the first  best
6 : The teacher is the first  best
5 : The teacher is the first  best
4 : The teacher is the first  best
3 : The teacher is the first  best
2 : The teacher is the first  best
1 : The teacher is the first  best
```

5.3 – Les boucles en C

65

- Somme des 10 premiers entiers : comparaison entre l'utilisation et de la boucle « tant que » de la boucle « pour »

```
somme = 0;
i=0;
while (i<10) {
   somme = somme + i;
   i = i + 1;
}
```

```
somme = 0;
for (i=0;i<10;i++)
   somme = somme + i;
```


5.3 – Les boucles en C

66

- Le langage C propose également une autre forme de la boucle tant que qui permet d'exécuter au moins une fois le corps de la boucle :

do
  <instruction>
while (<expression logique>)

- OU

do {
  <séquence d'instructions>
} while (<expression logique>)

5.3 – Les boucles en C

67

```
#include <stdio.h>

  main() {
    int i, somme, N;
    somme=0;
    printf("Entrez le nombre d'élément que vous voulez sommer : ");
    scanf("%d",&N);
    i=1;
    while (i<N) {
      somme = somme+i;
      i=i+1;
    }
    printf("Somme des %d premiers entiers est : %d\n",N,somme);
  }
```


5.3 – Les boucles en C

68

```
#include <stdio.h>

  main() {
    int i, somme, N;
    somme=0;
    printf("Entrez le nombre d'élément que vous voulez sommer : ");
    scanf("%d",&N);
    i=1;
    do {
      somme = somme+i;
      i=i+1;
    } while (i<N);
    printf("Somme des %d premiers entiers est : %d\n",N,somme);
  }
```

5.4 – Exercices

69

- Ecrire un programme C qui demande un nombre de départ, et qui affiche ensuite les dix nombres suivants.
- Par exemple, si l'utilisateur entre le nombre 17, le programme affichera les nombres de 18 à 27.

5.4 – Exercices

70

- Ecrire un programme C qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne.
- En cas de réponse supérieure à 20, on fera apparaître un message : « *Entrez un nombre plus petit !* »,
- et inversement, « *Entrez un nombre plus grand !* » si le nombre est inférieur à 10.

5.4 – Exercices

71

- Ecrire un programme C qui génère 2 nombres aléatoires qui ne seront pas connu de l'utilisateur.
  - Le premier nombre sera généré entre 1 et 50
  - Et le second entre 50 et 100
- Ecrire un programme C qui demande un nombre jusqu'à ce que la réponse soit comprise entre le premier et le deuxième nombre aléatoire généré.
- En cas de réponse supérieure à premier nombre aléatoire, on fera apparaître un message : « *Entrez un nombre plus petit !* »,
- et inversement, « *Entrez un nombre plus grand !* » si le nombre est inférieur au second nombre aléatoire.
- Si l'utilisateur saisie 0, le programme s'arrête.

5.4 – Exercices

72

- Ecrire un programme C qui affiche la figure suivante :

```
*******
*******
*******
*******
*******
*******
*******
```

5.4 – Exercices

73

- Ecrivez un programme C affichant un triangle d'étoiles. L'utilisateur entrera le nombre initial d'étoiles, et le programme affichera les lignes les unes à la suite des autres, chaque ligne perdant une étoile à chaque fois :

Nombre initial d'étoiles : 7

```
*
**
***
****
*****
******
*******
```


5.4 – Exercices

74

- Ecrivez un programme C affichant un triangle d'étoiles. L'utilisateur entrera le nombre initial d'étoiles, et le programme affichera les lignes les unes à la suite des autres, chaque ligne perdant une étoile à chaque fois :

Nombre initial d'étoiles : 7

```
*******
******
*****
****
***
**
*
```

75

5.4 – Exercices

- Il s'agit d'afficher les différents termes de la suite définie de la manière suivante :

- Le premier terme U_1 est compris entre 4 et 10 ($4 \leq U1 \leq 10$)
- Le terme général de la suite est :
  - $U_{n+1} = U_n/3$ si U_n est divisible par 3
  - $U_{n+1} = U_n+1$ si le reste de la division de U_n par 3 est 1
  - $U_{n+1} = U_n-2$ si le reste de la division de U_n par 3 est 2
- La suite s'arrête quand $U_n == 0$

75

76

Chapitre 6 : Les tableaux

LS1 - Algorithmique/programmation renforcés ; langage C - Ph. Hunel
2025-2026

76

77

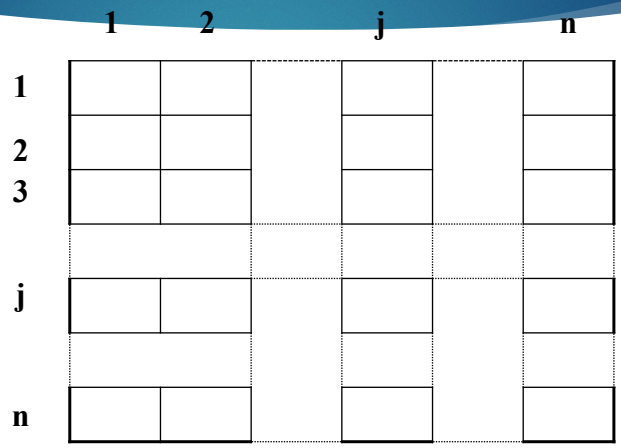
6.1 – Tableaux à un indice

- Un tableau (encore appelé table ou variable indicée) est un ensemble de données, qui sont toutes de même type, désigné par un identificateur unique (le nom du tableau), et qui se distinguent les unes des autres par leur numéro d'indice
- Exemple : les températures sous abri à 15h00 des jours d'une semaine seront les 7 valeurs de la variable température, qui est un tableau de 7 éléments (variables) de type réel désigné par :
  - Température[1], Température[2], …, Température[7],

77

78

6.2 – Tableaux à plusieurs indices



78

6.3 - Les tableaux en C

79

- Rappel :
  - Comme toute variable, un tableau doit être déclaré avant utilisation
- Principe :

  <type><identificateur>[taille$_1$][taille$_2$]…[taille$_k$];
- Exemple :

```
int a[13];
char b[8][5][10];
float d [6][15][9];
```

- N'importe quelle référence à une case peut être utilisé comme une simple variable :

```
int i,j,k;
a[i]
b[i][j][k]
```


6.3 - Les tableaux en C

80

- La taille correspond au nombre de cases du tableau
- Attention : les indices, permettant de localiser le contenu d'une case d'un tableau, varient entre

  0 et taille-1
- Il est possible d'affecter un tableau à un ensemble de valeurs dès sa déclaration par :

  <type><identificateur>[taille$_1$][taille$_2$]…={val$_1$, val$_2$, …};
- Exemple :

```
int matrice[2][3]={1,2,3,4,5,6}
```

⇔ $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$

Exemple de programme C

81

```
#include <stdio.h>

  main(){
    int i, somme, temperature[7];
    float moyenne;
    for (i=0;i<7;i++){
      printf("Temperature[%d]=",i);
      scanf("%d",&temperature[i]);
    }
   somme=0;
    for (i=0;i<7;i++)
      somme=somme+temperature[i];
    moyenne=somme/7;
    printf("la température moyenne de la semaine est %f\n",moyenne);
  }
```


Nombre d'élément d'un tableau en C

82

- Pour calculer automatiquement la taille d'un tableau :

```
int taille = sizeof(tableau) / sizeof(tableau[0]);
```

- Cela fonctionne uniquement dans la fonction où le tableau est déclaré, pas dans une fonction qui reçoit le tableau en paramètre !

83

6.4 – Les chaînes de caractères

- Il n'existe pas de type spécial « chaîne de caractère »
- Une chaîne de caractère est vue comme un tableau de caractère à une dimension
- Il existe cependant une notation particulière et de nombreuses fonctions spécifiques pour les chaînes de caractère

84

6.4.1 – Déclaration de chaîne de caractères

- Déclaration de chaînes de caractères en LDA

```
chaîne NomVariable
```

- Déclaration de chaînes de caractères en C

```
char NomVariable [<Longueur_optionnelle>];
```

- Exemples

```
char NOM [20];
char PRENOM [20];
char PHRASE [300];
```

6.4.1 – Déclaration de chaîne de caractères

85

- Espace à réserver
  - •Malheureusement, le compilateur C ne contrôle pas si un octet a été réservé pour le symbole de fin de chaîne;
  - •l'erreur se fera seulement remarquer lors de l'exécution du programme.
  - Pour un texte de **n** caractères, il faut donc prévoir **n+1** octets.

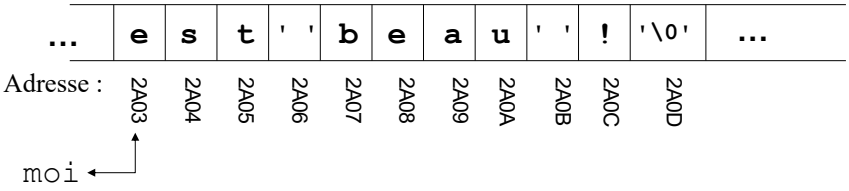
85

6.4.1 – Déclaration de chaîne de caractères

86

- Mémorisation

```
char moi[11] = "est beau !"
```



- L'adresse de la chaîne de caractères (tableau de caractères) est l'adresse du premier élément du tableau

86

6.4.1 – Déclaration de chaîne de caractères

87

- Initialisation
  - Initialisation classique d'un tableau :

```
Char moi[] = {'b','e','a','u','\0'};
```

- Facilité pour les chaînes de caractères :

```
Char moi[] = "beau";
```

- A l'initialisation par `[]`, le système réserve automatiquement le nombre d'octets nécessaires
  - *i.e.* taille de la chaîne + 1
  - Pour l'exemple moi : 4 + 1 = 5 octets

6.6.1 – Déclaration de chaîne de caractères

88

- Exemple

```
Char moi[] = "beau";
```

moi :	'b'	'e'	'a'	'u'	'\0'

```
Char moi[5] = "beau";
```

moi :	'b'	'e'	'a'	'u'	'\0'

```
Char moi[7] = "beau";
```

moi :	'b'	'e'	'a'	'u'	'\0'	0	0

6.6.1 – Déclaration de chaîne de caractères

89

- Exemple

```
Char moi[4] = "beau";
```

moi : | 'b' | 'e' | 'a' | 'u' | ☠ |

↳Erreur pendant l'exécution

```
Char moi[3] = "beau";
```

↳Erreur pendant la compilation

6.4.2 – Accès aux éléments

90

- L'accès à un élément d'une chaîne de caractères peut se faire de la même façon que l'accès à un élément d'un tableau

```
char moi[5];
```

Défini un tableau de 5 éléments :

`moi[0], moi[1], moi[2], moi[3], moi[4],`

```
char B[5] = "beau";
```

B : | 'b' | 'e' | 'a' | 'u' | '\0' |

B[0] B[1] B[2] B[3] B[4]

6.4.3 – Fonctions sur de chaîne de caractères

91

- **strlen(<s>)** : fournit la longueur de la chaîne ***sans*** compter le '\0' final
- **strcpy(<s>, <t>)** : copie <t> vers <s>
- **strcat(<s>, <t>)** : ajoute <t> à la fin de <s>
- **strcmp(<s>, <t>) :** compare <s> et <t> lexicographiquement et fournit un résultat:
  - Négatif : si <s> précède <t> Zéro : si <s> est égal à <t>
  - Positif : si <s> suit <t>
- **strncpy(<s>, <t>, <n>) :** copie au plus <n> caractères de <t> vers <s>
- **strncat(<s>, <t>, <n>)** : ajoute au plus <n> caractères de <t> à la fin de <s>

6.5 – Exercices

92

- Créer une liste contenant les 50 premiers entiers puis afficher leur somme.
- Écrire un algorithme qui indique le plus grand élément d'un tableau et la position à laquelle il se trouve.

```
Le maximum de T=[1,6,45,2,1,6,15,8,6,14] est 45
et il est à l'indice 3
```

- Écrire un programme qui inverse une liste

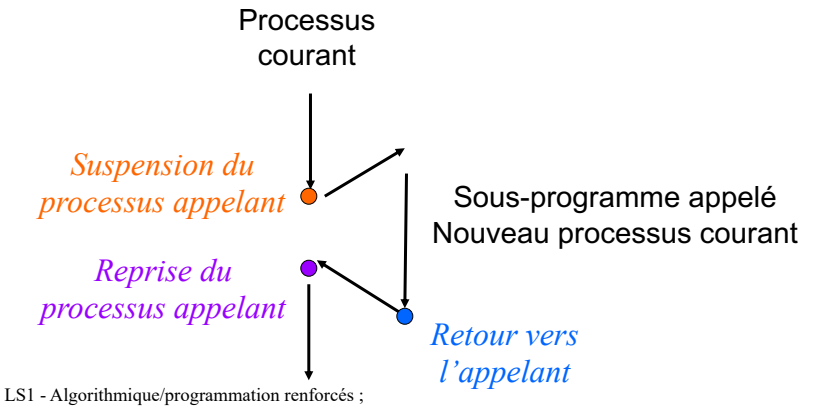
Chapitre 7 : Les sous-programmes

Exemple simple d'appels de sous-programmes


```
#include <stdio.h>
 main(){
  float X, Y;
  printf("Entrez la valeur : ");appel du sous-programme d'affichage
  scanf(''%f'',&X);     appel du sous-programme de lecture
  Y = cos(X);         appel du sous-programme cos
  printf(" %f" ,Y);       appel du sous-programme
 d'affichage
 }
```

95

Fonctionnement



95

96

7.2 – Environnement d'un sous-programme

- L'environnement d'un sous-programme est l'ensemble des variables accessibles et des valeurs disponibles dans ce sous-programme, en particulier celles issues du programme appelant.
- Trois sortes de variables sont accessibles dans le sous-programme :
  - Les variables dites globales
  - Les variables dites locales
  - Les paramètres formels

96

7.2 – Environnement d'un sous-programme

97

- Les variables dites globales :
  - Celles définies à l'extérieur de tout sous-programme et qui seront disponibles dans n'importe quel sous-programme
  - Elles peuvent donc être référencées partout dans le programme appelant et dans le sous-programme appelé
  - Leur portée est globale

97

7.2 – Environnement d'un sous-programme

98

- Les variables dites locales
  - Celles définie dans le corps du sous-programme, utiles pour son exécution propre et accessibles seulement dans ce sous-programme
  - Elles sont donc invisibles pour le programme appelant
  - Leur portée est locale

98

7.2 – Environnement d'un sous-programme

99

- Les paramètres formels
  - Les variables identifiées dans le sous-programme qui servent à l'échange d'information entre les programmes appelant et appelé
  - Leur portée est également locale.

99

7.2 - Paramètres formels & paramètres effectifs

100

- Lors de la déclaration de la procédure, les paramètres sont appelés des paramètres ***formels*** dans la mesure où ils ne possèdent pas encore de valeurs.
- Lors de l'utilisation de la procédure dans l'algorithme appelant, les paramètres sont appelés des paramètres ***effectifs*** (ou réels).
- **Remarque importante:** Dans une procédure, le nombre de paramètres formels est exactement égal au nombre paramètres effectifs.
- De même à chaque paramètre formel doit correspondre un paramètre effectif de même type.

100

101

7.3 – Procédures vs Fonctions

- **Définitions**
  - **Procédure** : Un bloc de code qui effectue une tâche spécifique **sans retourner de valeur**.
  - En C, une procédure est une fonction qui retourne **`void`**.
  - **Fonction** : Un bloc de code qui effectue une tâche spécifique et **retourne une valeur**.

101

102

7.3 – Procédures vs Fonctions

- **Exemple de procédure en C**

```c
#include <stdio.h>

// Déclaration et définition d'une procédure
void BonjourProf() {
    printf("Bonjour Monsieur le beau Prof !\n");
}

int main() {
    // Appel de la procédure
    BonjourProf();
    return 0;
}
```


102

103

7.3 – Procédures vs Fonctions

- **Exemple de fonction en C**

```c
#include <stdio.h>

// Déclaration et définition d'une fonction
int carre(int x) {
    return x * x;
}

int main() {
    int resultat = carre(5); // Appel de la fonction
    printf("Le carré de 5 est %d\n", resultat);
    return 0;
}
```


103

104

7.4 – Déclaration des procédure et fonction

- **Syntaxe de la déclaration**

```c
type_de_retour nom_de_la_fonction(type_param1 nom_param1, type_param2 nom_param2, ...) {
    // Corps de la fonction
    return valeur; // Si la fonction retourne une valeur
}
```


104

105

7.4 – Appel de Fonctions

- **Passage par valeur**
  - En C, les **paramètres sont passés par valeur** : une copie de la valeur est transmise à la fonction.

```c
#include <stdio.h>

void incrementer(int x) {
    x = x + 1;
    printf("Dans la fonction : %d\n", x);
}

int main() {
    int a = 5;
    incrementer(a);
    printf("Dans main : %d\n", a); // 'a' reste inchangé
    return 0;
}
```

```
Dans la fonction : 6
Dans main : 5
```


105

106

7.4 – Appel de Fonctions

- **Passage par référence (pointeurs)**
- En C, pour modifier une variable dans une fonction, on utilise des **pointeurs**.

```c
#include <stdio.h>

void incrementer(int *x) {
    (*x) = (*x) + 1;
}

int main() {
    int a = 5;
    incrementer(&a);
    printf("a = %d\n", a); // 'a' est modifié
    return 0;
}
```

```
a = 6
```


106

Représentation mémoire

107

Sens de transmission des données

…	A[1]	A[2]	A[3]	.....	X[1]	X[2]	X[3]	…
…	15	8	2	.....	15	8	2	…

Passage par valeur

…	…	…	…	.....	X[1] A[1]	X[2] A[2]	X[3] A[3]	…
…	…	…	…	.....	15	8	2	…

Passage par adresse

107

7.5 – Portée et Durée de Vie des Variables

108

- **Variables locales**
  - Déclarées à l'intérieur d'une fonction ou d'un bloc.
  - **Portée** : Accessibles uniquement dans le bloc où elles sont déclarées.
  - **Durée de vie** : Existent uniquement pendant l'exécution du bloc.

108

7.5 – Portée et Durée de Vie des Variables

- **Variables locales**

```c
#include <stdio.h>

void maFonction() {
    int x = 10; // Variable locale
    printf("x = %d\n", x);
}

int main() {
    maFonction();
    // printf("%d", x); // Erreur : 'x' n'est pas accessible ici
    return 0;
}
```


7.5 – Portée et Durée de Vie des Variables

- **Variables globales**
  - Déclarées en dehors de toute fonction.
  - **Portée** : Accessibles dans tout le programme.
  - **Durée de vie** : Existent pendant toute l'exécution du programme.

111

7.5 – Portée et Durée de Vie des Variables

- **Variables globales**

```c
#include <stdio.h>

int x = 10; // Variable globale

void maFonction() {
    printf("Dans maFonction, x = %d\n", x);
}

int main() {
    printf("Dans main, x = %d\n", x);
    maFonction();
    return 0;
}
```

```
Dans main, x = 10
Dans maFonction, x = 10
```

2025-2026

111

112

7.5 – Portée et Durée de Vie des Variables

- **Variables statiques**
  - Déclarées avec le mot-clé **`static`**.
  - **Portée** : Locale à la fonction ou au fichier.
  - **Durée de vie** : Existent pendant toute l'exécution du programme.

112

113

7.5 – Portée et Durée de Vie des Variables

- **Variables statiques**

```c
#include <stdio.h>

void maFonction() {
    static int x = 0; // Variable statique
    x++;
    printf("x = %d\n", x);
}

int main() {
    maFonction(); // x = 1
    maFonction(); // x = 2
    maFonction(); // x = 3
    return 0;
}
```

```
x = 1
x = 2
x = 3
```

2025-2026

113

114

7.6 – Passage de tableaux à une fonction

- En C, les tableaux sont **passés par référence** (adresse du premier élément).

```c
#include <stdio.h>

void afficherTableau(int tableau[], int taille) {
    for (int i = 0; i < taille; i++) {
        printf("%d ", tableau[i]);
    }
    printf("\n");
}

int main() {
    int monTableau[] = {1, 2, 3, 4, 5};
    afficherTableau(monTableau, 5);
    return 0;
}
```

```
1 2 3 4 5
```

2025-2026

114

115

7.6 – Passage de tableaux à une fonction

- **Modification d'un tableau dans une fonction**

```c
#include <stdio.h>

void doublerTableau(int tableau[], int taille) {
    for (int i = 0; i < taille; i++) {
        tableau[i] *= 2;
    }
}

int main() {
    int monTableau[] = {1, 2, 3, 4, 5};
    doublerTableau(monTableau, 5);
    for (int i = 0; i < 5; i++) {
        printf("%d ", monTableau[i]);
    }
    return 0;
}
```

```
2 4 6 8 10
```

Un tableau étant un pointeur, sa modification à l'intérieur de la fonction entraîne sa modification à l'extérieur

2025-2026

115

116

7.6 – Passage de tableaux à une fonction

- Pour garantir la non-modification d'un tableau dans une fonction faire précéder le paramètre tableau de `const`

LS1 - Algorithmique/programmation renforcés ; langage C - Ph. Hunel

2025-2026

116

7.6 –

117

```c
#include <stdio.h>

// Fonction qui affiche un tableau sans le modifier
void afficherTableau(const int tableau[], int taille) {
    for (int i = 0; i < taille; i++) {
        printf("%d ", tableau[i]);
    }
    printf("\n");
    // tableau[0] = 10; // Erreur de compilation
}

void doublerTableau(int tableau[], int taille) {
    for (int i = 0; i < taille; i++) {
        tableau[i] *= 2;
    }
}

int main() {
    int monTableau[] = {1, 2, 3, 4, 5};

    // Appel de la fonction qui n'a pas le droit de modifier le tableau
    afficherTableau(monTableau, 5);

    // Appel de la fonction qui modifie le tableau
    doublerTableau(monTableau, 5);

    afficherTableau(monTableau, 5); // Affiche 2, 3, 4, 5, 6

    return 0;
}
```

LS1 - Algorithmiq
Hunel
26

117

118

7.7 – Exercice

- Fonction permettant de calculer e^x pour x donnée avec une précision 10^{-3} et sachant que :

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}$$

LS1 - Algorithmique/programmation renforcés ; langage C - Ph. Hunel
2025-2026

118

7.7 – Exercice

119

```
Constante utilisée
E=0.001
Variables
Réel x
fonction expo(Donnée x: réel): réel
Entier i
réel f, ex
Début
  f ←1    ex ←1    i ←1
  Tant que f ≥ E faire
    f ← f * ( x / i )
    ex ← ex + f
    i ← i + 1
  ftq
  Retour ex;
Fin
```


119

7.4.3 – Traduction en C

120

```
#include <stdio.h>
#define E 0.001

float expo(float x) {
  int i;   float f,ex;
  f=1;   ex=1;   i=1;
  while (f>=E) {
    f=f*(x/i);
    ex=ex+f;
    i++;
  }
  return ex;
}
```


120

7.4.3 – Traduction en C

```
/*Programme principal */
main() {
  float x;
  printf("Donner la valeur de x : ");
  scanf("%f",&x);
  printf("L'exponentiel de x=%f est  %f\n",x,expo(x));
}
```


121

Chapitre 8 : Types évolués

LS1 - Algorithmique/programmation renforcés ; langage C - Ph. Hunel 2025-2026

122

123

8.1 – Les structures

- Les tableaux sont des paquets de données de même type.
- Les structures sont des ensembles de données non homogènes.
- Les données peuvent avoir des types différents.
- Les structures sont déclarées ou définies selon le modèle :

123

124

8.1 – Les structures

- Les structures sont déclarées ou définies selon le modèle :

```
struct nom_de_structure_facultatif {
   la liste des données contenues dans la
   structure
} la liste des variables construites selon ce modèle.
```


124

8.1 – Les structures

125

- Exemple :

```
struct etd {
   char nom[20];
   char prenom[20];
   float note_info;
   float note_phi;
   float moyenne;
} Honorer;
```


8.1 – Les structures

126

- `etd` est un nom de modèle de structure
- `Honorer` est un objet de type `struct etd`.
- Les différentes parties de la structure `Honorer` sont accessibles par :
  - `Honorer.nom`
  - `Honorer.prenom`
  - `Honorer.note_info`
  - `Honorer.note_phi`
  - `Honorer.moyenne`

8.2 – Les unions

127

- Les **unions** permettent l'utilisation d'un même espace mémoire par des données de types différents à des moments différents.
- Syntaxe de la définition d'une union

```
union nom_union {
   type1 nom_champ1;
   type2 nom_champ2;
    …
   typeN nom_champN;
} variables;
```


8.2 – Les unions

128

- Exemple :

```
union zone {
    int entier;
    float flottant;
} z1;
```

- Utilisation
  - z1.entier = 1;
  - z1.flottant = 2.5;

129

8.3 – Opération sur les structures et les unions

- L'affectation d'un objet de type structure est réalisé par l'opérateur égal

  =

  - réalise une affectation membre à membre des membres de l'objet situé à gauche du signe = par ceux de l'expression situé à droite.
  - **ATTENTION** :
    - Lorsque l'un des membres est un pointeur, le pointeur est affecté par un autre pointeur.
    - Les deux membres pointeurs des deux objets différents pointent sur la même donnée.
    - pas de duplication de la zone pointée.

129

130

8.3 – Opération sur les structures et les unions

- Les objets de type structure peuvent être passés en argument d'une fonction.
- Une fonction peut recevoir en argument un pointeur sur un objet de type structure.
- Une fonction peut retourner un objet de type structure.

130

131

8.4 – Opérateur typedef

- Le mot-clef **typedef** permet de renommer un type.

  typedef type nouveau_type ;

  - permet de simplifier la lisibilité de certaines déclaration ou de rendre un programme plus portable en renommant certains types susceptibles de varier d'un compilateur à l'autre.
  - Exemple :

    `typedef unsigned short int usint ;`

    - renomme le type `unsigned short int` en `usint`.

131

132

8.5 – exemple de programme

```
#include <stdio.h>

  typedef struct ed {
    char nom[20];
    char prenom[20];
    int analyse, info2;
    int *math1;
    float moyenne;
  } etudiant;

  void affiche(etudiant e) {
    printf("Nom : %s\n", e.nom);
    printf("Prénom : %s\n", e.prenom);
    printf("Note d'analyse : %d\n", e.analyse);
    printf("Note d'informatique 2 : %d\n", e.info2);
    printf("Note de maths 1 : %d\n", *e.math1);
    printf("Moyenne : %2.2f\n", e.moyenne);

  }
```


132

133

8.5 – exemple de programme

```
void main() {
     etudiant e1,e2;
     printf("e1 nom : "); scanf("%s",e1.nom);
     printf("e1 prénom : "); scanf("%s",e1.prenom);
     printf("e1 analyse : "); scanf("%d",&e1.analyse);
     printf("e1 info2 : "); scanf("%d",&e1.info2);
     printf("e1 math1 : "); scanf("%d",e1.math1);
     e1.moyenne = (e1.analyse + e1.info2 + *e1.math1)/3;
     printf("\n---- e1 ----\n");
     affiche(e1);
     e2=e1;
     printf("\n---- e2 ----\n");
     affiche(e2);
     *e2.math1=100;
     strcpy(e2.nom,"LE PROF");
     printf("\n---- e2 après modification de e2 ----\n");
     affiche(e2);
     printf("\n---- e1 ----\n");
      affiche(e1);
   }
```


133

134

8.6 – Exercices

- Définissez un type Point représentant les coordonnées (abscisse et ordonnée) réelles d'un point dans un plan
- Donnez une fonction qui renvoie dans Point les coordonnées lues au clavier
- Donnez la fonction qui calcule la distance euclidienne entre deux points

134

135

Chapitre 9 : Les pointeurs

135

9.1 - Introduction

136

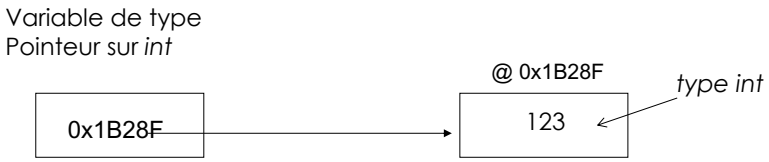
- A la compilation d'un programme, l'ordinateur réserve dans sa mémoire une place pour chaque variable déclarée.
- C'est à cette place que la valeur de la variable est stockée.
  - Association du nom de la variable à l'adresse de stockage.
  - Pendant le déroulement du programme, quand l'ordinateur rencontre un nom de variable, il va chercher à l'adresse correspondante la valeur en mémoire.

136

137

9.1 - Introduction

- Un pointeur est une variable qui contient l'**adresse en mémoire** d'un objet d'un type donné.
  - Autrement dit, une valeur de type pointeur repère une variable. Cela signifie qu'une valeur de type pointeur est l'adresse d'une variable d'un type donné.
  - On parle de « pointeur sur int » ou de « pointeur sur double ».

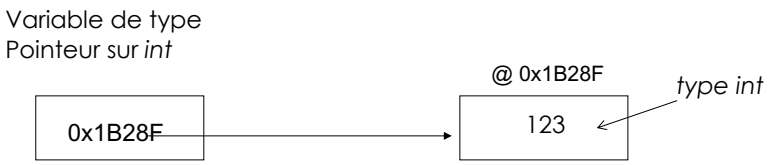


137

138

9.1 - Introduction

- **Analogie**
  - Une variable : une boîte contenant une valeur.
  - Un pointeur : une flèche pointant vers une boîte.

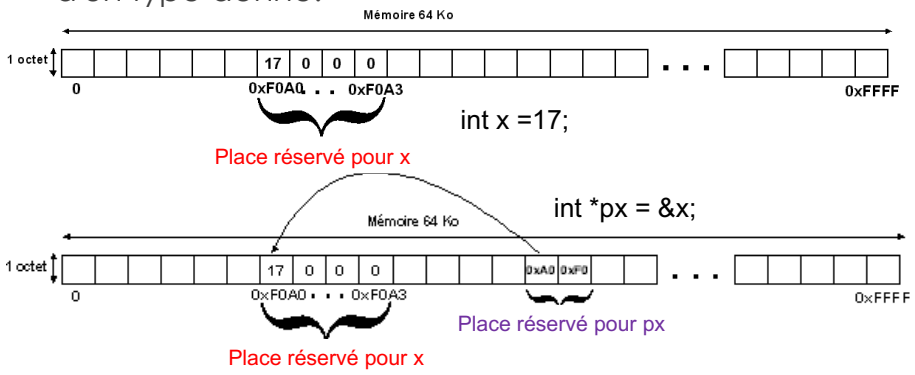


138

139

9.1 - Introduction

- Un pointeur contient l'**adresse en mémoire** d'un objet d'un type donné.



139

140

9.1 - Introduction

- Définition d'un pointeur
  - partir de la déclaration d'une variable ayant un type de base ;
  - ajouter le signe * devant le nom de la variable.
- Exemple :
  - `int *p;` /* p est un pointeur sur un *int* */
  - `char *c;` /* c est un pointeur sur *char* */

140

141

9.1 - Introduction

- Remarque :
  - Dans la définition d'un pointeur, le signe * est rattaché au nom de la variable qui suit et non au type.
  - Exemple :

```
int *p, i ;     /* p est un pointeur sur un int */
                /* i est simplement une variable entière */
```


141

142

9.1 - Introduction

- Opérateur *adresse de* : &
  - L'opérateur & permet de récupérer l'adresse en mémoire d'un objet.
  - Cette adresse pourra être stockée dans un pointeur.
  - Exemple :

```
int i=10;
int *pi;
pi = &i ; /* le pointeur pi repère la variable i */
```


142

9.1 - Introduction

143

- Opérateur *d'indirection* : *
  - L'opérateur * permet d'accéder au contenu de l'adresse mémoire pointée par un pointeur.
  - Exemple :

```
int *pi, i ;
pi = &i ; /* le pointeur pi repère la variable i */
*pi = 1 ; /* ⇒ i a la valeur 1 */
```


143

9.1 – Introduction – exemple complet

144

```c
#include <stdio.h>

int main() {
    int *ptr = &x;

    printf("Valeur de x : %d\n", x);          // 10
    printf("Adresse de x : %p\n", &x); // Adresse de x
    printf("Valeur de ptr : %p\n", ptr); // Adresse de x
    printf("Valeur pointée par ptr : %d\n", *ptr); // 10

    *ptr = 20; // Modifie x via le pointeur
    printf("Nouvelle valeur de x : %d\n", x); // 20

    return 0;
}
```

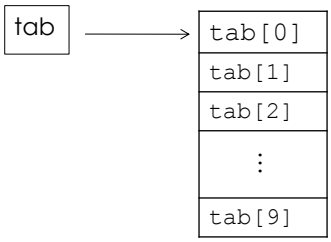
```
Valeur de x : 10
Adresse de x : 0x16badaf88
Valeur de ptr : 0x16badaf88
Valeur pointée par ptr : 10
Nouvelle valeur de x : 20
```


144

9.2 – Pointeur et tableau

145

- Vous avez déjà manipulé des pointeurs quand vous avez manipulé les tableaux.
- En fait, le nom seul du tableau est une constante qui contient l'adresse du premier élément du tableau comme le schématise la figure suivante



145

9.2 – Pointeur et tableau

146

- On peut récupérer l'adresse de n'importe quel objet.
- Par exemple, il est possible d'obtenir l'adresse d'un élément d'un tableau
  - ```
    double a[20];
    double *p;
    p = &(a[10]);
    ```
- Par convention, le nom d'un tableau est une constante égale à l'adresse du premier élément du tableau.
  - Par conséquent :
  - ```
    p = &a[0];
    p = a; /* sont équivalentes */
    ```


146

147

9.3 – Pointeur et calcul

- Il est possible de faire des calculs sur les pointeurs.
- On peut ajouter ou soustraire une *valeur entière* à un pointeur.
- Par extension, il possible de comparer les valeurs de deux pointeurs à l'aide des opérateurs ==, !=, <, <=, > et >=.
- De telles opérations n'ont de sens cependant, que si le pointeur repère un élément d'un tableau.
- En fait, les calculs sur pointeurs et l'utilisation de l'opérateur `[]` d'accès à un élément d'un tableau peuvent être considérés comme équivalent.
- Soit **`tab`**, un tableau de **`int`**, les deux lignes suivantes sont équivalentes :
  - `tab[i]`
  - `*(tab + i)`

147

148

9.3 – Pointeur et calcul

- **Incrémentation et décrémentation**
  - `ptr++` : déplace le pointeur vers l'adresse suivante (selon le type).
  - `ptr--` : déplace le pointeur vers l'adresse précédente.

148

149

9.3 – Pointeur et calcul - exemple

```c
#include <stdio.h>

int main() {
    int tab[3] = {10, 20, 30};
    int *ptr = tab; // ptr pointe vers le premier élément

    printf("*ptr initial = %d\n", *ptr); // 10
    ptr++; // ptr pointe vers le deuxième élément
    printf("*ptr après ++ = %d\n", *ptr); // 20
    ptr--; // ptr pointe à nouveau vers le premier élément
    printf("*ptr après -- = %d\n", *ptr); // 10

    return 0;
}
```

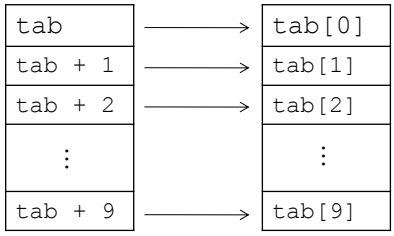
```
*ptr initial = 10
*ptr après ++ = 20
*ptr après -- = 10
```


149

150

9.2 – Pointeur et tableau

- `tab` est égal à `&tab[0]`
- `*tab` est égal à `tab[0]`
- L'élément `tab[i]` est équivalent à `*(tab+i)`
- On a donc les correspondances suivantes :



150

9.2 – Pointeur et tableau

151

```
int tab[3] = { 1 , 2, 3} ;
int *pta ;
int *ptb ;
pta = tab ;
ptb = pta+2 ;
```

- Dessiner le schéma de représentation
- Quelle est la valeur retournée par **`*pta`** et **`*ptb`**

151

9.2 – Pointeur et tableau

152

```
int tab[10];
int *pt ;
pt = tab ;
```

- On aura les équivalences suivantes :

  `*tab` ⇔ `tab[0]` ⇔ `*pt` ⇔ `pt[0]`

- Et plus généralement pour i de 0 à 9 :

  `*(tab+i)` ⇔ `tab[i]` ⇔ `*(pt+i)` ⇔ `pt[i]`

152

9.2 – Pointeur et tableau

153

- Un tableau à plusieurs dimensions est un tableau dont les éléments sont eux-mêmes des tableaux.
- Le tableau défini par

```
int tab[4][5]
```

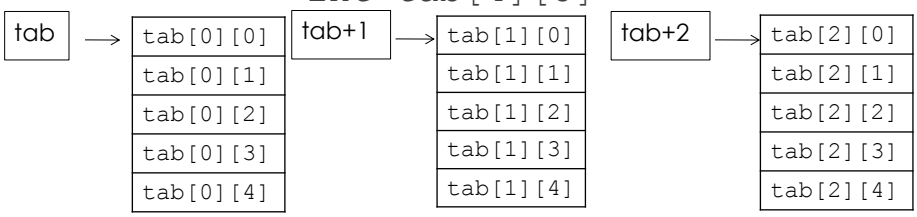
  - contient 4 tableaux de 5 entiers chacun.
  - `tab` donne l'adresse du 1er sous-tableau,
  - `tab+1` celle du 2ème sous-tableau et ainsi de suite

153

9.2 – Pointeur et tableau

154

`int tab[4][5]`



- l'opération `tab+2` n'ajoute pas `2` à la valeur de `tab`
- mais ajoute 2 fois le nombre d'octets correspondant à un tableau de 5 entiers, à savoir

  5*4 = 20 octets.

154

9.2 – Pointeur et tableau

155

- Tableau de pointeurs :
  - Se définit comme un tableau classique de variables avec une * devant
  - Exemple :

```
int *pttab[4] ;
/* pttab est un tableau de 4 pointeurs */
/* d'entiers */
```

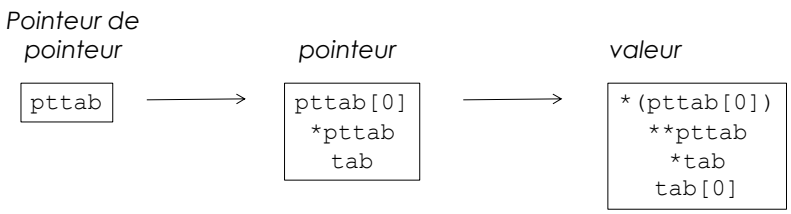

155

9.2 – Pointeur et tableau

156

- Exemple d'utilisation des tableaux de pointeurs :

```
int tab[4] = {1, 2, 3, 4};
int *pttab[4] = {tab, tab + 1, tab + 2, tab + 3};
```

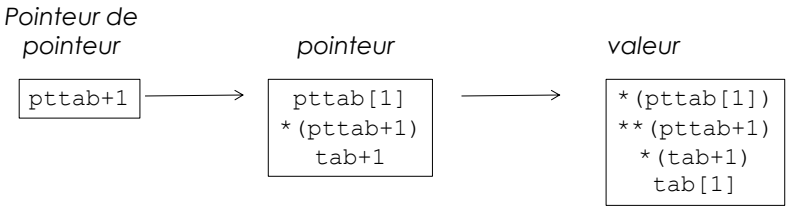


156

9.2 – Pointeur et tableau

- Exemple d'utilisation des tableaux de pointeurs :

```
   int tab[4] = {1, 2, 3, 4};
int *pttab[4] = {tab, tab + 1, tab + 2, tab + 3};
```



157

9.2 – Pointeur et tableau

- Exemple d'utilisation des tableaux de pointeurs :

```
int T1[4] = { 1 , 2 , 3 , 4 } ;
int T2[3] = { 5 , 6 , 7 } ;
int T3[1] = { 8 } ;
int *pt[3] = { T1 , T2 , T3 } ;
```


158

9.3 – Pointeur et calcul

159

```c
#include <stdio.h>
int main() {
    char mess[] = "Le beau prof !";
    char *p;
    int nbcar;
    nbcar = 0;
    p = mess;
    while (*p != '\0') {
        nbcar++;
        p++;
    }
    printf("\"%s\" possède %d caractères\n", mess,nbcar);
}
```

```
"Le beau prof !" possède 14 caractères
```


159

9.4 – Pointeur et passage de paramètre

160

- Deux modes de passage de paramètre :
  - **Passage par valeur**
    - passer une valeur qui sera exploitée par l'algorithme de la procédure
  - **Passage par adresse**
    - passer une référence à une variable, de manière à permettre à la procédure de modifier la valeur de cette variable
    - ⇔ passer l'adresse de la variable
    - ⇒ utilisation des pointeurs

160

9.4 – Pointeur et passage de paramètre

161

```
#include <stdio.h>
void add(int a, int b, int *c) {
/* c repère l'entier où on désire mettre le résultat */
/* on définit donc un pointeur pour dire qu'il s'agit de passer l'adresse */
    *c = a + b;
}
void main() {
    int i,j,k;
/* on passe les valeurs de i et j comme premiers paramètres */
/* on passe l'adresse de k comme troisième paramètre */
    add(i,j,&k);
}
```


161

9.4 – Pointeur et passage de paramètre

162

```
#include <stdio.h>
void add(int a, int b, int c) {
   c = a + b;
}
void main() {
   int i,j,k;
   i=2; j=3;
   k=1;
   add(i,j,k);
/* Quelle serait la valeur de k dans ce cas ? */
}
```


162

9.4 – Pointeur et passage de paramètre

163

```
#include <stdio.h>
void add(int a, int b, int *c) {
   *c = a + b;
}
void main() {
   int i,j,k;
   i=2; j=3;
   k=1;
   add(i,j,&k);
/* Quelle serait la valeur de k dans ce cas ? */
}
```


9.4 – Pointeur et passage de paramètre

164

- Ecrire une procédure qui permute le contenu de deux variables :

```
void permute (int a , int b){
   int buf ;
   buf = a ;
   a = b ;
   b = buf ;
}
```

9.4 – Pointeur et passage de paramètre

165

- Ecrire une procédure qui permute le contenu de deux variables :

```
void permute (int *a , int *b){
  int buf ;
  buf = *a ;
  *a = *b ;
  *b = buf ;
}
```


165

9.4 – Pointeur et passage de paramètre

166

- L'appel de la procédure `permute` dans le `main` se fera de la façon suivante :

```
int main(int argc, char** argv) {
  int toto, titi ;
  toto = 1 ;
  titi = 20 ;
  permute(&toto, &titi) ;
}
```


166

167

9.5 – Pointeurs et Structures

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

struct Personne {
    char nom[50];
    int age;
};

int main() {
    struct Personne p1 = {"Alice", 25};
    struct Personne *ptr = &p1;

    printf("Nom : %s, Age : %d\n", ptr->nom, ptr->age);
    strcpy(ptr->nom, "Bob");
    ptr->age = 30;
    printf("Nouveau nom : %s, Nouvel âge : %d\n", p1.nom, p1.age);

    return 0;
}
```

```
Nom : Alice, Age : 25
Nouveau nom : Bob, Nouvel âge : 30
```


167

168

Opérateurs d'incrémentation

- **++a (Incrémentation préfixée)**
  - Incrémente d'abord la variable a, puis retourne la nouvelle valeur.
  - Effet immédiat : La valeur de a est augmentée de 1 avant toute autre opération.
- **a++ (Incrémentation postfixée)**
  - Retourne d'abord la valeur actuelle de a, puis incrémente a après.
  - Effet différé : La valeur de a est augmentée de 1 après que sa valeur initiale ait été utilisée.

168

169

Opérateurs d'incrémentation

```c
#include <stdio.h>

int main() {

    int a,b;
    // Incrémentation préfixée
    a=1;
    b=++a;
    printf("a=%d - b=%d\n",a,b);

    // Incrémentation postfixée
    a=1;
    b=a++;
    printf("a=%d - b=%d\n",a,b);
    return 0;
}
```

```
a=2 - b=2
a=2 - b=1
```


169

Opérateurs d'incrémentation

- Si opération seule :

Incrémentation préfixée ⇔ Incrémentation postfixée

`++a ; ⇔ a++;`

```c
#include <stdio.h>

int main() {
    int a;
    a=1;
    printf("a=%d\n",a);
    ++a;
    printf("Valeur de a après ++a=%d\n",a);
    a=1;
    printf("a=%d\n",a);
    a++;
    printf("Valeur de a après a++=%d\n",a);
    return 0;
}
```

```
a=1
Valeur de a après ++a=2
a=1
Valeur de a après a++=2
```


170

171

Exercices

- On s'amuse avec les pointeurs...
  - Compléter le tableau en indiquant les valeurs des déférentes variables au terme de chaque instruction du programme suivant (on peut aussi indiquer sur quoi pointent les pointeurs) :

programme	a	b	c	p1	*p1	p2	*p2
`int a, b, c, *p1, *p2 ;`	?	?	?	?	?	?	?
`a = 1, b = 2, c = 3 ;`							
`p1 = &a, p2 = &c ;`							
`*p1 = (*p2)++ ;`							
`p1 = p2 ;`							
`p2 = &b ;`							
`*p1 -= *p2 ;`							
`++*p2 ;`							
`*p1 *= *p2 ;`							
`a = ++*p2 * *p1 ;`							
`p1 = &a ;`							
`*p2 = *p1 /= *p2 ;`							

171

172

Exercices

- Qu'affiche le programme suivant :

```
#include <stdio.h>

   int main() {
        int a = 10;
        int *p = &a;
        printf("%d\n", *p);
        return 0;
   }
```


172

173

Exercices

- Qu'affiche le programme suivant :

```
#include <stdio.h>

  int main() {
       int a = 5;
       int *p = &a;
       *p = 20;
       printf("%d\n", a);
       return 0;
  }
```


173

174

Exercices

- Qu'affiche le programme suivant :

```
#include <stdio.h>

  int main() {
       int a = 30;
       int *p = &a;
       int **pp = &p;
       printf("**pp=%d\n", **pp);
       return 0;
  }
```


174

Exercices

- Qu'affiche le programme suivant :

```
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3};
    int *p = arr;
    printf("%d\n", *(p + 1));
    return 0;
}
```


9.6 – Pointeur et allocation dynamique

- La déclaration d'une variable ou d'un tableau réalise automatiquement
  - la réservation d'un emplacement de la mémoire dont la taille correspond à la taille nécessaire au stockage des données
  - l'association de cette zone mémoire à l'identificateur de la variable.
- La zone reste réservée tout au long de la durée de vie de la variable.
- L'allocation de la mémoire est alors dite statique.

177

9.5 – Pointeur et allocation dynamique

- Il est parfois nécessaire d'allouer de la mémoire dynamiquement.
  - Taille du tableau non connu au moment de l'écriture du programme
    - Soit on choisit une taille très grande
    - Soit on choisit une taille adaptée au moment de l'utilisation
- Ceci est réalisé à l'aide des fonctions **`malloc`**, **`calloc`**, **`realloc`** définies dans le fichier d'en-tête <stdlib.h>.

177

178

9.5 – Pointeur et allocation dynamique

- D'une machine à une autre, la taille réservée pour un int, un float,... change
- L'utilisation de l'opérateur **`sizeof`** permet de connaître la taille d'un objet.
  - `sizeof nom_variable` fournit la taille de la variable *nom_variable*
  - `sizeof nom_constante` fournit la taille de la constante *nom_constante*
  - `sizeof (type)` fournit la taille pour un objet du type *type*

178

```c
#include <stdio.h>
int main() {
    int a[10] ;
    char b[5][10] ;
    printf("taille de a : %d\n",sizeof a) ;
    printf("taille de b : %d\n",sizeof b) ;
    printf("taille de 4.25 : %d\n",sizeof 4.25) ;
    printf("taille de Bonjour ! : %d\n",sizeof "Bonjour !") ;
    printf("taille d'un float : %d\n",sizeof(float)) ;
    printf("taille d'un double : %d\n",sizeof(double)) ;

}
```

179

- Produiront à l'exécution sur un PC ?

```
taille de a : 40
taille de b : 50
taille de 4.25 : 8
taille de Bonjour ! : 10
taille d'un float : 4
taille d'un double : 8
```


179

9.5 – Pointeur et allocation dynamique

180

- La fonction **malloc** reçoit en argument la taille de l'emplacement à réserver et retourne l'adresse de l'emplacement (de type void *) réservé ou NULL si l'allocation échoue.
- La fonction **calloc** reçoit en argument un entier n et la taille d'un élément et retourne l'adresse d'un emplacement réservé (de taille n*taille_element) ou NULL si l'allocation échoue.
- La fonction **calloc** initialise aussi tous les éléments de la zone allouée à 0.

180

9.5 – Pointeur et allocation dynamique

- La fonction **realloc** réalise le redimensionnement d'un emplacement précédemment alloué dynamiquement.
- Elle reçoit l'adresse de la zone à redimensionner et la nouvelle taille de l'emplacement.
- Elle retourne l'adresse du nouvel emplacement (éventuellement inchangée) ou NULL si l'allocation échoue.
- La fonction **free** réalise la libération de la zone mémoire allouée située à l'adresse qui lui est passée en argument.

9.5 – Pointeur et allocation dynamique

- La fonction **malloc** reçoit en argument la taille de l'emplacement à réserver et retourne l'adresse de l'emplacement (de type void *) réservé ou NULL si l'allocation échoue.

```
int *pi;
pi = (int *)malloc(sizeof(int));
float *pf;
pf = (float *)malloc(sizeof(float));
```

183

9.5 – Pointeur et allocation dynamique

- La fonction **calloc** reçoit en argument un entier n et la taille d'un élément et retourne l'adresse d'un emplacement réservé (de taille n*taille_element) ou NULL si l'allocation échoue.

```
int *tab, nb;
printf("Nombre d'éléments du tableau : ");
scanf("%d", &nb);
tab = (int *)calloc(nb,sizeof(int));
```

⇔

```
tab = (int *)malloc(nb*sizeof(int));
```


183

184

9.5 – Pointeur et allocation dynamique

- La fonction **realloc** réalise le redimensionnement d'un emplacement précédemment alloué dynamiquement.

```
int *tab, nb;
printf("Nombre d'éléments du tableau : ");
scanf("%d", &nb);
tab = (int *)calloc(nb,sizeof(int));

tab = (int *)realloc(tab,(nb+10)*sizeof(int));
```


184

185

9.5 – Pointeur et allocation dynamique

- Cet opérateur est donc souvent utilisé avec les fonctions d'allocation de mémoire.

```
int n ;
int *tab ;
printf("taille du tableau : ") ;
scanf("%d", &n) ;
tab = (int *)malloc(n*sizeof(int)) ;
```

- Les **(int *)** devant le malloc s'appelle un **cast**
- La mémoire allouée de cette façon doit être libérée après utilisation par la fonction **free**.

185

186

9.5 – Pointeur et allocation dynamique - exemple

- Exemple :

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

int main() {
    int * p;
    int i;
    p = (int *)malloc(sizeof(int));
    *p = 1;
    i = *p + 1;
    printf("*p = %d, i = %d\n", *p, i);
    free(p);
}
```

```
*p = 1, i = 2
```


186

187

9.5 – Pointeur et allocation dynamique - exemple

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int * pt;
    int * p;
    int i;
    int nb;
    printf("Nombre d'éléments du tableau : ");
    scanf("%d", &nb);
    pt = (int *)calloc(nb,sizeof(int));
    for(i=0;i<nb;i++) {
        printf("Entrez l'élément d'indice %d : ",i);
        scanf("%d",pt+i);
    }
    for(p=pt;p<pt+nb;p++)
    printf("L'élément d'indice %d est %d\n",p-pt,*p);
    free(pt);
}
```

```
Nombre d'éléments du tableau : 5
Entrez l'élément d'indice 0 : 1
Entrez l'élément d'indice 1 : 5
Entrez l'élément d'indice 2 : 10
Entrez l'élément d'indice 3 : 15
Entrez l'élément d'indice 4 : 20
L'élément d'indice 0 est 1
L'élément d'indice 1 est 5
L'élément d'indice 2 est 10
L'élément d'indice 3 est 15
L'élément d'indice 4 est 20
```


187

188

9.6 – Les tableaux de pointeurs

`int *D[];`

- Si `D[i]` pointe dans un tableau,
- `D[i]` : désigne l'adresse de la première composante
- `D[i]+j` : désigne l'adresse de la j-ième composante
- `*(D[i]+j)` : désigne le contenu de la j-ième composante
- Exemple
- `char *JOUR[] = {"dimanche", "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi"};`

188

189

9.6 – Les pointeurs vers des pointeurs

- Un pointeur vers un pointeur est une variable qui contient l'adresse d'un autre pointeur
- Exemple

```c
#include <stdio.h>

int main() {
    int x = 10;
    int *ptr = &x;
    int **ptr_ptr = &ptr; // Pointeur vers un pointeur

    printf("Valeur de x : %d\n", **ptr_ptr); // 10
    return 0;
}
```

```
Valeur de x : 10
```


189

190

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

int main() {
    int **matrice;

    // Allocation mémoire
    matrice = (int**)malloc(lignes * sizeof(int*));
    for (int i = 0; i < lignes; i++) {
        matrice[i] = (int*)malloc(colonnes * sizeof(int));
    }
    // Initialisation
    for (int i = 0; i < lignes; i++) {
        for (int j = 0; j < colonnes; j++) {
            matrice[i][j] = i + j;
        }
    }

    // Affichage
    for (int i = 0; i < lignes; i++) {
        for (int j = 0; j < colonnes; j++) {
            printf("%d ", matrice[i][j]);
        }
        printf("\n");
    }

    // Libération de la mémoire
    for (int i = 0; i < lignes; i++) {
        free(matrice[i]);
    }
    free(matrice);
}
```

tion dans les tableaux dynamiques

```
0 1
1 2
2 3
```


190

191

9.6 – Les tableaux de pointeurs

- Considérez les déclarations de `NOM1` et `NOM2`:

```
char *NOM1[] = {"Kilo", "Loui-
  Mich",   "Renardo", "Pont-Dalmas", "Boulotte"};
char NOM2[][16]={"Kilo", "Loui-
  Mich",   "Renardo", "Pont-Dalmas", "Boulotte"};
```

a) Représenter graphiquement la mémorisation des deux variables `NOM1` et `NOM2`.

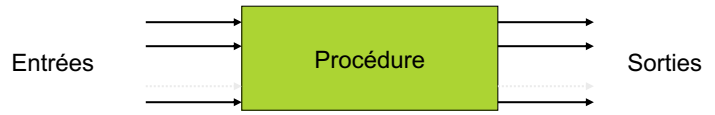
b) Imaginez que vous devez écrire un programme pour chacun des deux tableaux qui trie les chaînes selon l'ordre lexicographique. En supposant que vous utilisez le même algorithme de tri pour les deux programmes, lequel des deux programmes sera probablement le plus rapide?

191

192

193

7.3 – Les procédures



193

194

7.3 – Les procédures

- La définition d'une procédure comprend trois parties:
- Partie 1 : L'entête

  Procédure *Nom_procédure* **(***Liste de paramètres formels***);**
- Partie 2 : Déclaration des variables locales

  type_variable_i Nom_variable_i
- Partie 3 : Corps de la procédure

  Début

  Instructions

  Fin

194

7.3 – Les procédures

195

- **Exemple :**
  - Calcul de la somme de deux matrices carrées A et B.
  - Pour calculer la somme de deux matrices il faut lire d'abord ces deux matrices.
  - Ainsi, au lieu d'écrire deux fois le sous-programme de lecture d'une matrice, il est possible d'écrire plutôt une procédure et de l'utiliser pour saisir A et B.

195

7.3.1 - Paramètres formels & paramètres effectifs

196

- Lors de la déclaration de la procédure, les paramètres sont appelés des paramètres ***formels*** dans la mesure où ils ne possèdent pas encore de valeurs.
- Lors de l'utilisation de la procédure dans l'algorithme appelant, les paramètres sont appelés des paramètres ***effectifs*** (ou réels).
- **Remarque importante:** Dans une procédure, le nombre de paramètres formels est exactement égal au nombre paramètres effectifs.
- De même à chaque paramètre formel doit correspondre un paramètre effectif de même type.

196

197

7.3.2 - Passage des paramètres

- Lors de la définition d'une procédure il est possible de choisir entre trois modes de transmission de paramètres:
  - **Mode donnée** *(En C on parle de Passage par valeur)*
  - **Mode Donnée/Résultat** (*En C on parle de Passage par adresse*):

197

198

7.3.2.1 – Mode données

- Rappel
  - Déclarer une variable de nom X ⇒ Réserver un emplacement mémoire pour X
- Lors de l'appel d'une procédure un emplacement mémoire est réservé pour chaque paramètre formel.
- De même, un emplacement mémoire est aussi réservé pour chaque paramètre effectif lors de sa déclaration.
- Cependant, lors de l'appel de la procédure, les valeurs des paramètres effectifs sont copiées dans les paramètres formels.

198

199

7.3.2.1 – Mode données

- L'exécution des instructions de la procédure se fait avec les valeurs des paramètres formels et toute modification de ces derniers ne peut affecter en aucun cas celles des paramètres effectifs.
- Dans ce type de passage de paramètres, les valeurs des paramètres effectifs sont connues avant le début de l'exécution de la procédure et jouent le rôle uniquement d'entrées de la procédure.

199

200

7.3.2.1 – Mode données

- En C, en principe il n'existe que des fonctions. Une fonction qui ne retourne aucun résultat (type `void`) peut traduire la notion de procédure.

Procédure Afficher(**Données** Y:Matrice, **Données** n:entier)

- Se traduit en C par :

```
void Afficher(Matrice Y, int n);
```

Il y a eu initialisation des paramètres formels par des valeurs lors de l'appel de la procédure

200

201

7.3.2.1 – Mode données

Procédure carrés(**Données** X,Y :**entiers**)
Entiers A,B
Début
A ← X
B ← Y
A ← A∗A
B ← B∗B
Fin

```
Entiers M1,M2
Début
  Lire M1, M2
  carrés(M1,M2)
  écrire « M1= »,M1, « M2= »,M2
Fin
```


201

202

7.3.2.1 – Mode données

Procédure carrés_bis(**Données** X,Y :**entiers**)
Début
X ← X∗X
Y ← Y∗Y
écrire « X= »,X, « Y=
Fin

```
Entiers M1,M2
Début
  Lire M1, M2
  carrés_bis(M1,M2)
  écrire « M1= »,M1, « M2= »,M2
Fin
```


202

7.3.2.2 – Données/Résultats

203

- Passage par adresse, par variable
- La différence principale entre le passage par valeur et le passage par adresse c'est que dans ce dernier **un seul emplacement mémoire** est réservé pour le paramètre formel et le paramètre effectif correspondant.
- Dans ce cas chaque paramètre formel de la procédure utilise **directement** l'emplacement mémoire du paramètre effectif correspondant.
- Toute modification du paramètre formel entraîne la même modification du paramètre effectif correspondant.

203

7.3.2.2 – Mode Données/Résultats

204

Procédure Somme(**Données** D,F:entier, **Données-Résultat** R:entier)

- Se traduit en C par :

```
void Somme(int D, int F, int *R);
```

- Contrairement au passage par valeur, dans ce cas à l'appel de la procédure les valeurs des paramètres effectifs ne sont pas copiées dans les paramètres formels.
- Ces derniers utilisent directement l'emplacement mémoire (ou l'adresse) des paramètres effectifs.

204

Représentation mémoire

205

Sens de transmission des données

…	A[1]	A[2]	A[3]	…….	X[1]	X[2]	X[3]	…
…	15	8	2	…….	15	8	2	…

Passage par valeur

…	…	…	…	……	X[1] A[1]	X[2] A[2]	X[3] A[3]	…
…	…	…	…	……	15	8	2	…

Passage par adresse

205

7.4 – Les fonctions

206

- Une fonction est un sous-programme qui calcule, à partir de paramètres éventuels, une valeur d'un certain type utilisable dans une expression du programme appelant.
- Les langages de programmation proposent de nombreuses fonctions prédéfinies : partie entière d'un nombre, racine carrée, fonctions trigonométriques, etc.

206

207

7.4 – Les fonctions

- Les fonctions sont comparables aux procédures à deux exceptions près :
  - Leur appel ne constitue pas à lui seul une instruction, mais figure dans une expression ;
  - Leur exécution produit un résultat qui prend la place de la fonction lors de l'évaluation de l'expression.

207

208

7.4 – Les fonctions

- La définition d'une fonction comprend aussi trois parties:
- Partie 1 : L'entête

Fonction *Nom_fonction(Liste de paramètres formels)*: **Type_fonction**

- Partie 2 : Déclaration des variables locales

type_variable_i Nom_variable_i

- Partie 3 : Corps de la fonction

Début

Instructions

Retour ***valeur retournée***

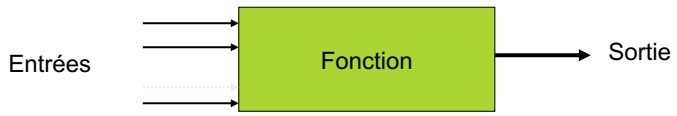
Fin

208

209

7.4 – Les fonctions

- Il s'agit donc de sous-programmes *nommés* et *typés* qui calculent et retournent une et une seule valeur.
- Le type de cette valeur est celui de la fonction.



209

210

7.4.1 - Passage des paramètres

- Lors de la définition d'une fonction il ne doit, normalement, être utilisé qu'un seul mode de transmission de paramètres :
  - **Mode donnée** *(En C on parle de Passage par valeur)*
- *Cependant les langages de programmation permettent d'utiliser les autres modes.*

210

211

7.4.2 – Fonction : exemple

- Fonction permettant de calculer e^x pour x donnée avec une précision 10^{-3} et sachant que :

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}$$

211

212

Constante utilisée
E=0.001
Variables
Réel x
fonction expo(**Donnée** x: réel): réel
Entier i
réel f, ex
Début
f ←1 ex ←1 i ←1
Tant que f ≥ E **faire**
f ← f * (x / i)
ex ← ex + f
i ← i + 1
ftq
Retour ex;
Fin

212

213

Algorithme principal

Début

Écrire « Donner la valeur de x: »

Lire x

Écrire « L'exponentielle de x= », x, « est : », expo(x)

Fin

213

214

7.4.3 – Traduction en C

```
#include <stdio.h>
#define E 0.001

float expo(float x) {
  int i;   float f,ex;
  f=1;   ex=1;   i=1;
  while (f>=E) {
    f=f*(x/i);
    ex=ex+f;
    i++;
  }
  return ex;
}
```


214

7.4.3 – Traduction en C

```
/*Programme principal */
main() {
   float x;
   printf("Donner la valeur de x : ");
   scanf("%f",&x);
   printf("L'exponentiel de x=%f est
   %f\n",x,expo(x));
}
```


7.5 – Fonctions récursives

- Expression d'algorithme sous forme récursive :
  - définition plus concise
  - Démonstration par récurrence
- Principe :
  - Utiliser pour décrire l'algorithme sur une donnée *d*, l'algorithme lui même appliqué à un sous-ensemble de *d* ou une donnée *d'* plus petite
- Une fonction récursive possède donc la propriété de s'appeler elle-même dans sa définition
- Permet de coder des fonctions définies à partir de relations de récurrence

7.5 – Fonctions récursives

- Exemple calcul du factoriel d'un nombre entier :
  - 1ère définition :
    - n! = 1 * 2 * 3 * 4 * … * n
    - Traduction par une boucle :

Je vous écoute

  - 2ème définition :
    - $n! = \begin{cases} 1 \text{ si } n=0 \\ n * (n-1)! \end{cases}$

217

7.5 – Fonctions récursives


```
fonction factoriel(Donnée n: entier): entier
Entier fac
Début
  si n = 0 alors
    fac ← 1
  sinon
    fac ← n * factoriel(n-1)
  fsi
  Retour fac;
Fin
```


218

7.5 – Fonctions récursives

219

- Dérouler cet algorithme pour :
  factoriel(5)
- Factoriel(5) → 5 * 24 =
  - Factoriel(4) → 4 * 6 = 24
    - Factoriel(3) → 3 * 2 = 6
      - Factoriel(2) → 2 * 1 = 2
        - . Factoriel(1) → 1 * 1 = 1
          - - Factoriel(0) → 1

7.5 – Exercices

220

- Écrire une fonction minimum3(a, b, c) qui retourne le plus petit de trois nombres.
- Ecrire une fonction qui renvoie le nombre d'éléments pairs d'un tableau
- Écrire une fonction qui renvoie le nombre d'occurrences de la valeur *val* dans un tableau de N éléments entiers.
- Écrire une fonction qui renvoie le nombre de bigrammes d'un tableau de caractères
  - Exemple le tableau de caractères contenant la phrase « elle m'appelle pour partir avec la folle » contient 3 bigrammes « ll ».
- Écrire une fonction qui recherche si un sous-tableau existe dans un tableau donné

221

Chapitre 8 : Les algorithmes de tri

222

8.1 - Introduction

- Un tableau T est dit « **trié en ordre croissant** » si tous les éléments consécutifs du tableau vérifient :

$$T[i-1] \leq T[i]$$

- Il est admis qu'un
  - tableau vide est trié
  - tableau ne contenant qu'un seul élément est trié

223

8.1 - Introduction

- D'où la définition :
  - Un tableau vide (n=0) est ordonné (trié),
  - Un tableau contenant un seul élément (n=1) est ordonné,
  - Un tableau T[1..n], n>1, est ordonné si

$$\forall\, i \in [2..n],\ T[i-1] \leq T[i]$$

224

8.1 - Introduction

- Tri d'un tableau
  - Soit un vecteur (tableau à une dimension) T[1..n] à valeurs dans un ensemble E de valeurs muni d'une relation d'ordre notée <
  - Trier le vecteur T consiste à construire un vecteur T'[1..n] tel que :
    - T' soit trié,
    - T' et T contiennent les mêmes éléments.
  - Le plus souvent T et T' sont le même vecteur ; T' est construit en permutant entre eux les éléments de T .

8.2 – Tri par remplacement

225

- Cette méthode simple et intuitive est malheureusement très peu performante.
- Elle consiste à construire un tableau Ttrié[1..n] à partir de T[1..n] tel que :

$$Ttrié[i-1] \leq Ttrié[i] \ , \ \forall \ i \in [2..n]$$

- Principe :
  - Identifier le maximum du tableau
  - Rechercher le minimum du tableau T
  - Recopier ce minimum dans Ttrié à la position i
  - Remplacer le minimum du tableau T par le maximum
  - Recommencer pour i+1

225

8.2 – Tri par remplacement

226

```
Procédure tri_remplacement (D/R T[1..n], R Ttrié[1..n] : tableau de …)
Entier i,j
E max
Début
  max ← maximum(T[1..n])
  i ← 1
  tant que i<n faire
    j ← indice_min(T[1.n])
    Ttrié[i] ← T[j]
    T[j] ← max
    i ← i+1
  ftq
  Ttrié[n] ← max
Fin
```

```
Fonction maximum( D T[1..n] : tableau de …):E
Entier i
E max
Début
  max ← T[1]
  Pour i ← 2 à n faire
    si T[i]>max alors
      max ← T[i]
    fsi
  fpour
  Retour max
Fin
```


226

8.2 – Tri par remplacement

227

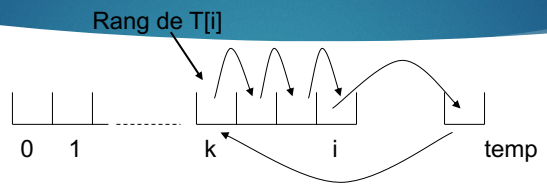
- Pour chaque élément rangé dans le tableau Ttrié, il faut parcourir tout le tableau T et non une partie du tableau T
- Nécessite un 2ème tableau, or si le nombre d'éléments à trier est important, cet algorithme requiert donc un espace mémoire double.

8.3 - Tri par insertion

228

- Cette méthode de tri insère (au i ème passage) le i ème élément T[i] à la bonne place parmi T[1],T[2]...T[i-1].
- Après l'étape i, tous les éléments entre la première et la i ème position sont triés.
- Il existe plusieurs méthode de tri par insertion selon le principe qui est utilisé pour rechercher le rang de l'élément à insérer parmi les éléments du début de la liste déjà triés

8.3 - Tri par insertion



- Principe de l'algorithme :
  - pour i ← 2 à n faire

    déplacer T[i] vers le début du tableau jusqu'à

    la position j<=i telle que

    T[j] < T[k] pour j<=k<i et (ou bien T[j]>=T[j-1] ou bien j=1).

8.3 - Tri par insertion

4	2	0	5	3	Vecteur de départ
2	4	0	5	3	Les cellules 1 à 2 sont triées
0	2	4	5	3	Les cellules 1 à 3 sont triées
0	2	4	5	3	Les cellules 1 à 4 sont triées
0	2	3	4	5	Les cellules 1 à 5 sont triées

231

8.3 - Tri par insertion

```
Procédure tri_insertion (D/R T[1..n], : tableau de …)
Entier i,j
Début
 pour i← 2 à n faire
  j ← i-1
  tant que j ≥ 1 et T[j] > T[j+1] faire
   échange(T,,j+1,,j)
   j ← j-1
  ftq
 fpour
Fin
```

Exemple visuel

232

8.4 – Tri par sélection

- Le principe du tri par sélection d'un vecteur est d'aller chercher le plus petit élément du vecteur pour le mettre en premier, puis de repartir du second, d'aller chercher le plus petit élément pour le mettre en second etc.
- Au $i^{ème}$ passage, on sélectionne l'élément ayant la plus petite valeur parmi les positions i..n et on l'échange avec T[i].

8.4 – Tri par sélection

233

4	2	0	5	3	Vecteur de départ
0	2	4	5	3	Le plus petit élément est à sa place
0	2	4	5	3	Les 2 plus petits éléments sont à leur place
0	2	3	5	4	Les 3 plus petits éléments sont à leur place
0	2	3	4	5	Les 4 plus petits éléments sont à leur place

8.4 – Tri par sélection

234

```
Procédure tri_sélection (D/R T[1..n], : tableau de …)
Entier i,j
Début
  pour i← 1 à n faire
    pour j← i+1 à n faire
     si T[i] > T[j] alors
       échange(T,i,j)
     fsi
    fpour
  fpour
Fin
```

Exemple visuel

235

8.5 – Tri à bulles

- Le principe du tri à bulles *(bubble sor*t) est de comparer deux à deux les éléments e_1 et e_2 consécutifs d'un tableau et d'effecteur une permutation si $e_1 > e_2$.
- On continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

236

8.5 – Tri à bulles

4	2	0	5	3	Vecteur de départ
4	0	2	3	5	Fin du premier passage
0	2	3	4	5	Fin du deuxième et dernier passage

237

8.5 – Tri à bulles

```
Procédure tri_bulles (D/R T[1..n], : tableau de …)
Entier i,j
Début
 pour i← 1 à n-1 faire
  pour j← n à i+1 faire {décroissant}

   si T[j-1] > T[j] alors
    échange(T,j-1,j)
   fsi
  fpour
 fpour
Fin
```


238

8.5 – Tri à bulles

- Évaluation du coût de l'algorithme
  - Les comparaisons sont effectuées à l'intérieur de la boucle la plus interne
  - Pour i=1, n-1 comparaisons sont effectuées
  - Pour i=2, n-2 comparaisons sont effectuées
  - … … … …
  - Pour i=n-1, 1 comparaison est effectuée
  - Le nombre total de comparaisons effectuées est donc : $\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$

239

8.5 – Tri à bulles

- Optimisation de l'algorithme
  - Après avoir traité i-1 éléments ($1 \leq i \leq n$)
    - Les éléments de 1..i-1 sont triés
    - Tous les éléments compris entre 1..i-1 sont inférieurs aux éléments compris entre i..n

      $T[1..i-1] \leq T[i..n]$
    - Les éléments compris entre i..n ne sont pas traités
  - Si les éléments compris entre i..n sont triés à la suite des permutations effectuées, il est inutile de poursuivre car :
    - T[1..i-1] trié ; $T[1..i-1] \leq T[i..n]$; T[i..n] trié $\Rightarrow$ T[1..n] trié
  - Exemple de BATEAUX

240

```
Procédure tri_bulles2 (D/R T[1..n], : tableau de …)
Entier i,j
Booléen onapermuté
Début
 i← 1
 onapermuté ← VRAI
 Tant que onapermuté faire
  onapermuté ← FAUX
  pour j← n à i+1 faire {décroissant}
   si T[j-1] > T[j] alors
     échange(T,j-1,j)
     onapermuté ← VRAI
   fsi
  fpour
  i← i+1
 ftq
Fin
```

241

```
PRODECURE Tri_bulle (D/R T[1..n], : tableau de ...)
Booleen permut
Début
 REPETER
   permut ← FAUX
   POUR i ← 1 à N-1 FAIRE
    SI T[i] > T[i+1] ALORS
      échange(T,i,i+1)
      permut ← VRAI
    FSi
   Fpour
 Jusqu'à permut = FAUX
FIN
```

Exemple visuel

242

8.6 – Tri indirect

- Ce tri utilise un tableau auxiliaire qui indique, pour chaque élément du tableau à trier, le rang que celui-ci devrait occuper dans le tableau trié.
- Le principe consiste à compter, pour chaque élément du tableau à trier, le nombre d'éléments qui lui sont inférieurs ou égaux. Le nombre trouvé donnera la place (l'indice) de cet élément dans le tableau trié
- Le tri se fait donc ensuite par l'intermédiaire de cet index

243

8.6 – Tri indirect

```
Procédure tri_indirect (D/R T[1..n], : tableau de …)
Entier i
Tableau d'entier ind
Début
 pour i← 1 à n faire
   ind[i] ← 1
 fpour
            COMPTAGE DES ELEMENTS
 pour i← 1 à n faire
    Ttrié[ind[i]] ← T[i]
 fpour
Fin
```


243

244

8.6 – Tri indirect

COMPTAGE DES ELEMENTS

```
pour i← 1 à n-1 faire
  pour k← 1 à n faire
    si T[k] < T[i] alors
      ind[i] ← ind[i] + 1
    fsi
  fpour
fpour
```

Exemple visuel

244

8.6 – Tri indirect

AMELIORATION DU COMPTAGE DES ELEMENTS

```
pour i← 1 à n-1 faire
  pour k← i+1 à n faire
    si T[i] < T[k] alors
      ind[i] ← ind[i] + 1
    sinon
      ind[k] ← ind[k] + 1
    fsi
  fpour
fpour
```


Chapitre 9 : Les algorithmes de recherche

LS1 - Algorithmique/programmation renforcés ;
langage C - Ph. Hunel
2025-2026

247

9.1 - Introduction

- Le but de des algorithmes de recherche est de trouver un élément donné parmi une liste d'éléments fourni par un vecteur.
- 9.2 – Recherche séquentielle dans un tableau non trié
  - On sait faire
  - Parcourir le tableau jusqu'à ce qu'on trouve l'élément recherché ou qu'on ait atteint la fin du tableau
  - Dans le pire des cas nécessitera *n* comparaisons (*n* étant le nombre d'éléments du tableau)
    - Cas où l'élément recherché n'appartient pas au tableau

248

9.3 – Recherche séquentielle dans un tableau trié

- On sait faire
  - Parcourir le tableau jusqu'à ce qu'on trouve l'élément recherché ou **un élément supérieur à l'élément recherché** ou encore qu'on ait atteint la fin du tableau
  - Dans le pire des cas nécessitera *n* comparaisons (*n* étant le nombre d'éléments du tableau)
    - Cas où l'élément recherché est plus grand que le dernier élément du tableau (*i.e* élément d'indice *n*)
  - Amélioration possible :
    - Regarder si l'élément recherché n'est pas plus grand que le dernier élément du tableau
    - Mais le problème demeure si l'élément recherché est le dernier élément du tableau

9.4 – Recherche dichotomique dans un tableau trié

249

- Existe-t-il un algorithme plus efficace (en nombre de comparaisons) qui permettrait de rechercher l'existence d'un élément donné dans un tableau trié?

9.4 – Recherche dichotomique dans un tableau trié

250

- Supposons un tableau T[1..n] trié

  $\forall i \in [2..n], T[i-1] \leq T[i]$

- Partitionnons le tableau T en 3 sous-tableaux T[1..m-1], T[m], T[m+1..n]
- En comparant l'élément recherché à T[m], s'il n'y a pas égalité, il est possible de dire à quel sous-tableau appartient l'élément recherché :
  - À T[1..m-1] si l'élément recherché < T[m],
  - À T[m+1..n] si l'élément recherché > T[m],
- On se retrouve alors dans le cas précédent mais avec les sous-tableaux T[1..m-1] ou T[m+1..n]

9.4 – Recherche dichotomique dans un tableau trié

251

- Donc :
  - Soit il a été trouvé un indice m tel que

    Élément recherché = T[m]

    - et l'algorithme est terminé
  - Soit il a été trouvé en un nombre fini d'itération un sous-tableau vide et dans ce cas

    Élément recherché $\notin$ T[1..n]

    - et l'algorithme est terminé

9.4 – Recherche dichotomique dans un tableau trié

252

- Il faut maintenant déterminer les sous-tableaux T_1, T_2, T_3, …,T_k tel que chaque sous-tableau T_i soit de taille (nombre d'éléments) strictement inférieure à la taille du sous-tableau précédent T_{i-1}.
- Remarque :
  - Si pour T_1, T_2, T_3, …,T_k , on choisit T[1..n], T[2..n], T[3..n], …, T[k..n], on se ramène à la recherche séquentiel précédente.
- Généralement l'indice m est choisit de telle sorte que la taille de T[1..m-1] et T[m+1..n] soit la même à un élément près.

253

```
fonction dichoto (D T[1..n], : tableau de …; D elem_rech: …): booléen
Entier inf, sup, m
Booléen trouvé
Début
 inf ← 1  sup ← n
 trouvé ← FAUX
 Tant que inf ≤ sup et non trouvé faire
  m ← (inf + sup) div 2
  si T[m] = elem_rech alors
   trouvé ← VRAI
  sinon si T[m] < elem_rech alors
   inf ← m+1
  sinon
   sup ← m-1
  fsi fsi
 ftq
 retour(trouvé)
Fin
```


253

254

9.4 – Recherche dichotomique dans un tableau trié

- Coût de l'algorithme :
  - La taille des tableaux de la suite $T_1, T_2, T_3, \ldots, T_k$ est divisé par 2 à chaque itération :
  - n, n/2, …, $n/2^{k-1}$
  - On aura donc la partie entière de $\log_2(n)$ sous-tableaux non vides

254


255
Fin
LS1 - Algorithmique/programmation renforcés ; langage C - Ph. Hunel
2025-2026

255