جامعة مولاي إسماعيل
UNIVERSITÉ MOULAY ISMAÏL

كلية العلوم والتقنيات
FACULTÉ DES SCIENCES ET TECHNIQUES

Support du cours :
Algorithmique & Programmation II
Filière : MIP (S3)

Module : I132

Responsable : Pr. FATIMA AMOUNAS

Groupe ROI, Département d'Informatique

Faculté des Sciences et Techniques

Errachidia - Maroc

A-U : 2020-2021

Table des matières

3. Les tableaux

- 3.1 Les tableaux à une dimension
  - 3.1.1. Définition
  - 3.1.2 Déclaration et mémorisation
  - 3.1.3. Accès aux composantes
  - 3.1.4. Initialisation et réservation automatique
  - 3.1.5. Opérations élémentaires sur un tableau 1D
  - 3.1.6. Traitement opérant sur les tableaux
  - 3.1.7 Tri d'un tableau
- 3.2. Les tableaux à deux dimensions
  - 3.2.1. Déclaration et mémorisation
  - 3.2.2. Initialisation et réservation automatique
  - 3.2.3. Opérations élémentaires sur un tableau 2D
- 3.3 Les chaînes de caractères
  - 3.3.1. Déclaration et mémorisation
  - 3.3.2. Initialisation de chaînes de caractères
  - 3.3.3. Fonctions standards de manipulation des chaînes
  - 3.3.4. Autres fonctions de manipulation des chaînes

4. Les pointeurs

- 4.1 Introduction
- 4.2 Déclaration d'un pointeur
- 4.3 Opérateurs de manipulation des pointeurs
- 4.4 Allocation dynamique de mémoire
- 4.5 Pointeurs et tableaux
  - 4.5.1 Pointeurs et tableaux à une dimension
  - 4.5.2 Pointeurs et tableaux à plusieurs dimensions
  - 4.5.3 Pointeurs et chaînes de caractères

5. Les fonctions

- 5.1 Introduction
- 5.2 Définition d'une fonction
- 5.3 Déclaration d'une fonction
- 5.4 Appel d'une fonction
- 5.5 Transmission des paramètres d'une fonction
- 5.6 Tableaux et Fonctions
- 5.7 Fonctions récursives

6. Les types composés

- 6.1 Introduction
- 6.2 Les structures
- 6.3 Les unions
- 6.4 Les énumérations
- 6.5 Nommage des types

Avant propos

Ce polycopié a pour objectif d'être un support de cours d'initiation à l'algorithmique et la programmation en langage C. Ce cours est destiné essentiellement aux étudiants de la deuxième année, parcours MIP (Mathématique Informatique et Physique).
Ce polycopié est organisé en deux parties:
Dans la 1ere partie, nous allons rappeler les notions de base sur l'algorithmique. Nous commençons par les étapes de résolution d'un problème informatique. Ensuite, nous allons donner la structure d'un algorithme ainsi que les instructions de base.
La deuxième partie est structurée en six chapitres:
- Dans le 1er chapitre, nous allons introduire le langage C, puis connaître la structure générale d'un programme C. Ensuite, nous allons décrire les concepts de base suivie par les types de données manipulés en langage C. Les instructions de base : l'affectation, les instructions d'Entrées/Sorties standards vont être aussi traitées dans ce chapitre.
- Dans le 2eme chapitre, nous allons voir les structures de contrôle, à savoir les instructions alternatives et les instructions répétitives. Les instructions de branchement non conditionnel vont être abordées dans ce chapitre.
- Dans le 3eme chapitre, nous allons aborder les structures tabulaires, à savoir les tableaux unidimensionnels, les tableaux multidimensionnels et les chaînes de caractères. Nous allons nous restreindre aux tableaux à 2dim qui sont beaucoup utilisés en pratique.
- Dans le 4eme chapitre, nous allons aborder la notion de pointeur suivi par les opérations sur les pointeurs. Nous allons ensuite traiter l'allocation dynamique de la mémoire qui constitue le processus de base pour créer les structures dynamiques.
- Dans le 5eme chapitre, nous allons initier à la programmation modulaire et voir comment décomposer un problème en plusieurs modules. Ensuite, le passage des arguments d'une fonction va être traité dans ce chapitre.
- Dans le 6eme chapitre, nous allons montrer comment le programmeur peut définir ses propres types afin de pouvoir manipuler des valeurs adaptées au problème à résoudre. Dans ce cadre, nous aborderons les structures, qui sont des types complexes hétérogènes, les unions qui sont des types simples hétérogènes et les énumérations qui sont des types simples homogènes. Nous verrons également comment associer un nouvel identificateur à un type personnalisé.

Partie 1. Algorithmique

Chapitre 1
Initiation à l'algorithmique

- Introduction
- Etapes de résolution d'un problème
- Structure d'un Algorithme
- Types de base
- Opérateurs
- Instructions de base

1.1 Introduction

Le terme informatique est un néologisme proposé en 1962 pour caractériser le traitement automatique de l'information. Il est construit sur la contraction de l'expression « information automatique ».

L'informatique traite deux aspects complémentaires : les programmes immatériels (logiciel, software) qui décrivent un traitement à réaliser et les machines (matériel, hardware) qui exécutent ce traitement. Dans ce contexte, l'ordinateur désigne un équipement informatique permettant de traiter les informations selon des séquences d'instructions (opérations).

Etant donné un problème à résoudre, son analyse consiste à le décomposer en sous problèmes plus simples jusqu'à atteindre des problèmes élémentaires que l'ordinateur est capable à résoudre. Le langage de programmation permet de transcrire le résultat de l'analyse dans un formalisme compréhensible par l'ordinateur.

Il existe différentes classes de langages de programmation :

- langages procéduraux : Fortran, Pascal, C, …
- Langages fonctionnels : Lisp, Prolog, …
- Langages objets : C++, Java, …

Dans les langages procéduraux, on distingue deux éléments fondamentaux :

- Les données : qui correspondent en quelques sortes aux variables en mathématiques.
- Les traitements : qui s'appliquent à des données et correspondent en quelques sortes aux fonctions mathématiques.

1.2 Les étapes de résolution d'un problème

La résolution d'un problème passe par 3 étapes:

- la pré-analyse qui consiste à identifier et comprendre le problème.

- l'analyse qui consiste à collecter les données nécessaires pour la résolution du problème.

- l'élaboration de l'algorithme qui s'agit de l'ensemble d'étapes à suivre pour résoudre le problème

Exemple 1: Construction géométrique

- Placer deux points A et B dans le plan.
- Tracer le cercle de centre A passant par B.

- Tracer le cercle de centre B passant par A.
- Tracer la droite passant par les points d'intersections C et D de ces deux cercles.

Exemple 2: Problème: résolution de l'équation ax + b =0

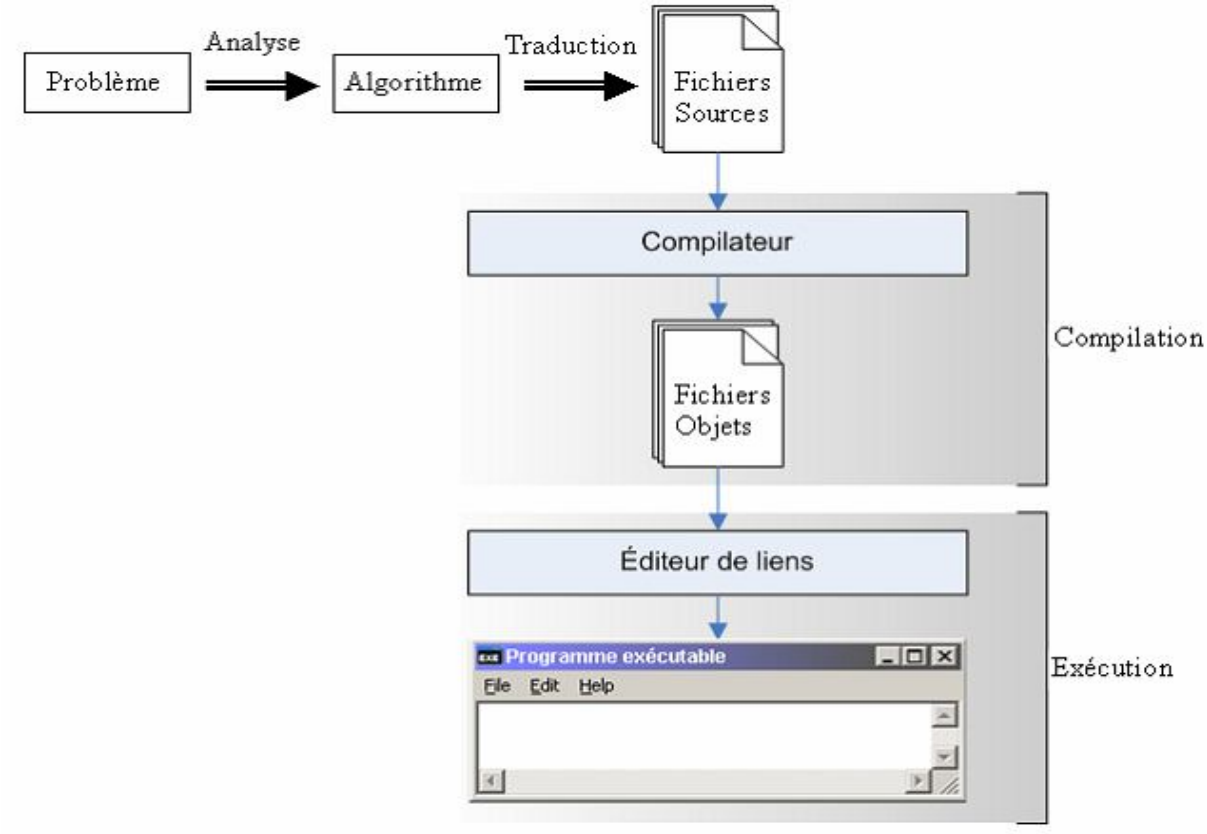
- La pré-analyse: il s'agit de chercher la solution (valeur de x).

- L'analyse: il s'agit de collecter les données nécessaires pour calculer x:

  - Les valeurs de a et b
  - Vérifier si $a \neq 0$
  - Calculer la valeur de x : -b/a

- L'élaboration de l'algorithme: il s'agit de:
  - Donner les valeurs de a et b.
  - Calculer la valeur de x.
  - Afficher le résultat.

Le schéma suivant illustre les étapes de résolution informatique d'un problème.



→ **Un algorithme** représente l'enchaînement des actions (instructions) nécessaires pour faire exécuter une tâche par ordinateur (résoudre un problème). Un algorithme s'écrit le plus souvent en **pseudo-langage** de programmation (appelé langage algorithmique).
Un algorithme est un ensemble d'étapes successives, finies et ordonnées et qui visent à résoudre un problème bien défini.
→ L'algorithmique est la logique d'écrire des algorithmes.

Exemples:

- Algorithme de retrait d'argent auprès d'un guichet automatique
- Algorithme de calcul de la moyenne de 3 notes à coefficient égal à 1.

→ **Un programme** est un ensemble d'instructions (actions) qui aboutissent à l'exécution d'un traitement souhaité.

La programmation recouvre l'ensemble des techniques permettant de résoudre des problèmes à l'aide de programmes s'exécutant sur un ordinateur. Une étape essentielle consiste en l'écriture d'un texte de programme dans un langage particulier (un langage de programmation).
L'écriture du programme est un processus de programmation que l'on peut décomposer de la manière suivante :

1. **L'analyse et la spécification du problème**, qui permettent de préciser le problème à résoudre. Dans cette phase, on détermine quelles sont les données et leurs propriétés, quels sont les résultats attendus, ainsi que les relations exactes entre les données et les résultats.
2. **La conception (ou modélisation)** : il s'agit généralement de déterminer la méthode de résolution du problème (un algorithme) et d'en identifier les principales étapes.
3. **L'implantation (ou codage)** dans un ou plusieurs langages de programmation particuliers. Il s'agit de traduire la méthode et les algorithmes préconisés en un ou plusieurs textes de programmes.

1.3 Structure d'un algorithme

```
ALGORITHME <Nom>
<Déclaration des variables>
Debut
  <Actions>
Fin
```

Un algorithme est constitué:

- d'une **entête** composé du mot réservé ALGORITHME et d'un nom de l'algorithme à réaliser.
- d'une zone de **déclaration** des variables utilisés dans l'algorithme.
- **d'un corps** délimité par deux mots réservés DEBUT et FIN. C'est ici qu'on écrit les actions de l'algorithme.

1.3.1) Notion de Variable

Un programme s'exécute en manipulant des données se trouvant dans la mémoire centrale. Une variable est donc un nom d'un emplacement de la mémoire centrale qui contient des données. Ces données varient au cours de l'exécution du programme. Chaque variable possède un type et est rangée en mémoire à une adresse précise.
Elle est caractérisée par :

- **un nom** (c'est un identificateur)
  q=quotient, R=Reste, Moy=Moyenne, ….

- **Un type** (nature de l'objet : entier, caractère…). Un type détermine en particulier les valeurs possibles de l'objet et les opérations primitives applicables à l'objet.
  R: entier
  CAR: caractère;
  ADR: chaine de caractères
- **Une valeur** (contenu de l'objet). Cette valeur peut varier au cours de l'algorithme ou d'une exécution à l'autre.

Dans les cas contraires (valeur fixe) ce sont des constantes. Tous les objets manipulés par un algorithme doivent être clairement définis :

Mot clé :
CONST : PI=3.14

1.3.2) Les types de base

Lors de la déclaration d'une variable, on doit spécifier son type.

- Entier

Le type entier comprend les valeurs numériques entières, positives ou négatives.

Exemple:

X: entier
Age: entier
Annee, mois, jour: entier

- Réel

Le type réel comprend les valeurs décimales positives ou négatives à virgule.

Exemple:

X: réel
Note1 : réel

- Caractère

Une variable de type caractère accepte les 26 lettres majuscules et les 26 lettres minuscules, les 10 chiffres et les caractères spéciaux.

Exemple:

C: caractère
C1, C2: caractère

- Chaîne de caractères

Les chaînes de caractères sont délimitées par des guillemets.

Exemple: "ali","123"

Nom: chaîne de caractères

- Booléen

Il accepte deux valeurs: vrai ou faux (true ou false).

Exemple :

trouve : booléen

1.3.3) Les opérateurs

- Les opérations arithmétiques:

L'addition (+), la soustraction (-), la division réelle (/), la division entière (div), la multiplication (*), le reste de la division (mod).

- Les opérations logiques:

- NON pour la négation
- ET pour la conjonction
- OU pour la disjonction

Var1	Var2	NON var1	Var1 ET var2	Var1 ou var2
Vrai	Vrai	Faux	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai
Faux	Vrai	Vrai	Faux	Vrai
Faux	Faux	Vrai	Faux	Faux

- Les opérations de comparaison:

Ces opérateurs qui s'appliquent sur les variables de type booléen sont:
< , >, <= , >= , =

1.4) Les instructions de base

1.4.1) Instruction d'affectation

L'affectation consiste à placer une valeur dans une variable.

Syntaxe :

Variable ← valeur

Ou

Variable ← expression

Exemple:

A ← 5
B ← A+2

1.4.2) Instructions d'entrée/sortie

a) Instruction d'écriture (Sortie)

L'instruction d'écriture "écrire" a pour rôle d'afficher des informations sur un périphérique de sortie (écran).

Exemple :

écrire (var1) ➔ affiche sur l'écran le contenu de la variable var1
écrire ("100") ➔ affiche sur l'écran 100
écrire ("Bonjour") ➔ affiche sur l'écran le texte Bonjour var
écrire ("Bonjour", var) ➔ affiche sur l'écran le texte Bonjour puis le contenu de la variable var1.

b) Instruction de lecture (Entrée)

L'instruction de lecture "lire" a pour rôle de permettre à l'utilisateur d'entrer des valeurs au programme à partir de l'entrée standard (le clavier).

Exemple :

lire (var1) ➔ lire une valeur à partir du clavier et la mettre dans la variable var1.

1.5) Instructions de contrôle

Les structures de contrôle sont les éléments du langage qui déterminent l'ordre dans lequel les instructions sont exécutées.

Séquence : exécution dans l'ordre, l'une après l'autre.

Test : exécution d'une instruction sous une certaine condition.

Boucle : exécution d'une instruction plusieurs fois.

1.5.1) Enchaînement séquentiel

Les instructions sont exécutées dans l'ordre où elles apparaissent.

Instruction 1
Instruction 2
...
Instruction n

Exemple :

tmp ← a première instruction
a ← b deuxième instruction
b ← tmp troisième instruction

1.5.2) Instructions conditionnelles

La forme générale de cette instruction est:

a) Instruction à un seul choix

Syntaxe:

```
Si (condition) alors
Instructions
Fin si
```

Le bloc d'instructions est composé d'une ou de plusieurs instruction(s). Cette partie est exécutée si la condition est vraie.

Exemple :

Écrire un algorithme qui lit un nombre entier au clavier et affiche « pair » si il est pair.

```
Si (n mod 2=0) Alors
        Écrire ("pair")
FinSi
```

b) Instruction à deux choix :

Syntaxe:

```
Si (condition) alors
        Instructions1
Sinon
        Instructions2
 Fin si
```

Instruction1 est exécutée si la condition est satisfaite (vrai).
Instruction2 est exécutée si la condition est fausse.

Exemple 1 **:** Affiche le maximum de deux entiers a et b.

```
Si (a>b) alors
                ecrire (a ) ;
sinon
                ecrire (b ) ;
Finsi
```

Les **tests imbriqués** sont courants lorsqu'il y a plus de deux possibilités.

Exemple 2 **:** Indiquer le signe d'un nombre donné.

```
si (a>0) alors
        ecrire ("Ce nombre est positif") ;
sinon

si (a=0) alors
        ecrire ("Ce nombre est nul") ;
sinon
        ecrire ("Ce nombre est négatif") ;
    finsi
finsi
```

c) Instruction à plusieurs choix

La structure **Selon** permet d'effectuer tel ou tel traitement en fonction de la valeur introduite.

Syntaxe :

```
Selon (variable)
Valeur 1: Instructions1
Valeur 2: Instructions2
…
Valeur n:  Instructions_n
Sinon
        Autre Instructions
Fin selon
```

Dès qu'il y a correspondance (variable=valeur_i), les comparaisons sont arrêtées et la séquence associée est exécutée. Les différents choix sont donc exclusifs. Si aucun choix ne correspondant, alors la séquence associée au « *Sinon* », si elle existe, est exécutée.

Exemple : feux de signalisation

```
Selon couleur
'vert': ecrire("passer")
'Jaune': ecrire("attention")
'Rouge': ecrire("stop")
Sinon
        ecrire("erreur")
finselon
```

1.5.3) Instructions Répétitives

Le traitement en boucle permet de réaliser une série d'opérations un nombre de fois ou sous certain condition.

a) Instruction « Pour … »

Il est fréquent que le **nombre de répétitions (n)** soit **connu** à l'avance. Le mécanisme permettant répéter une ou plusieurs instructions n fois est la **boucle POUR.**

La forme générale de cette instruction est:

Syntaxe :

```
Pour variable de vi à vf pas= p Faire
Instructions
Fin pour
```

Exemple 1 : **affiche « bonjour » 50 fois**

```
Pour variable i de 1 à 50  Faire
   Ecrire ("bonjour")
Fin pour
```

Exemple 2 **: affiche les lettres de l'alphabet z y x …a**

```
Pour variable i de 'z' à 'a' pas=-1  Faire
   Ecrire (i, "  ")
Fin pour
```

Exemple 3 : **affiche la suite 0 3 6 9 12 ….27**

```
Pour variable i de 0 à 27 pas=3  Faire
   Ecrire (i, "  ")
Fin pour
```

b) Instruction « tant que… »

La forme générale de cette instruction est:

Syntaxe:

```
Tant que (Condition_Entrée)  faire
        Instructions
Fin tant que
```

Le contenu de la structure « tant que » **peut ne jamais être exécuté.** Donc cette structure permet en réalité de répéter un traitement **0, 1 ou plusieurs fois**.

Exemple :

```
i ← 1
tantque ( i≤5) faire
        ecrire (i*i, "  ")
        i ← i+1
Fintantque
```

c) Instruction « Repeter…»

La forme générale de cette instruction est:

```
Répéter
        Instructions
jusqu'à (Condition_d_arrêt)
```

Il y a toujours **au moins une exécution du corps**. La structure **Répéter** permet de répéter un traitement **1** ou **plusieurs fois**.

Exemple : **la saisie de l'age d'une personne.**

```
Répéter
        Lire(age)
jusqu'à (age>0)
```

Synthèse

Dans cette section introductive, nous avons abordé quelques concepts généraux relatifs à l'algorithmique et au langage de programmation. Nous présentons ensuite plus particulièrement le langage C, avant d'aborder ses caractéristiques plus en détails dans les sections suivantes.

Partie 2. Programmation

Chapitre 1.
Les bases de la programmation en C

- Introduction
- Composants élémentaires du C
- Structure d'un programme C
- Types prédéfinis
- Fonctions d'entrées-sorties classiques

1.1. Introduction

Le langage C a été développé par Kernighan et Ritchie pour écrire le système Unix dans un langage portable. Il a été largement utilisé tant pour l'écriture du noyau que pour les principaux outils du système. Le langage est très populaire, structuré, compilé, évolué (proche du langage naturel et scientifique), procédural (modulaire) et portable (réutilisé et réintégrable dans plusieurs plates formes).

1.2. Les composants élémentaires du C

Un programme en langage C est constitué de cinq groupes de composants élémentaires suivants :

- Les identificateurs,
- Les mots-clés,
- Les variables,
- Les constantes,
- Les opérateurs,

On peut ajouter à ces groupes les commentaires, qui sont enlevés par le préprocesseur.

1.2.1 Les identificateurs

Le rôle d'un identificateur est de donner un nom à une entité du programme. Plus précisément, un identificateur peut désigner :

- un nom de variable ou de fonction,
- un type défini par typedef, struct, union ou enum,
- une étiquette.

Un identificateur est une suite de caractères parmi :

- les lettres (minuscules ou majuscules, mais non accentuées),
- les chiffres,
- le ``blanc souligné" (_).

Le premier caractère d'un identificateur ne peut pas être un chiffre.
Par exemple : var1, tab_23.

1.2.2 Les mots-clés

Un certain nombre de mots, appelés mots-clés, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs.
L'ANSI C compte 32 mots clés :

auto	const	double	float	int	short	struct	unsigned	
break	continue	else	for	long	signed	switch	void	
case	default	enum	goto	register	sizeof	typedef	volatile	
char	do	extern	if	return	static	union	while	

1.2.3 Les commentaires

En langage C, un commentaire débute par /* et se termine par */.

Exemple : /* ceci est un commentaire
sur plusieurs lignes */

Il est possible d'employer le symbole // (ce type de commentaire est réservé au langage C++).

// Ceci est un commentaire sur une seule ligne

1.2.4 Les variables

Une instruction de déclaration permet de définir une ou plusieurs variables et de leur associer un espace mémoire afin de stocker une ou plusieurs informations.

Syntaxe de déclaration :

```
Type identificateur;
Type identificateur1, identificateur2, …, identificateurN;
```

<Type> : int → entier (2 ou 4 octets)
short → entier court (2 octets)
long ou long int → entier long (4 octets)
char → caractère (1 octet)
float → réel (4 octets)
double → réel ou double précision (8 octets)

Exemples :

```
char c ;
float x ;
int i, j ;
```

Remarque :

- Le type booléen est déclaré comme étant un entier où la valeur 0 correspond à la valeur «faux » et la valeur 1 est supposée « vrai ».
- La valeur d'une variable du type caractère (char) doit être mise entre quottes (' ').
  Exemples : 'a', 'h', 'E'.

1.2.5 Les opérateurs

- Opérateurs arithmétiques

+ Addition 5 + 8
- Soustraction 5 - 8
* Multiplication 5 * 8
/ Division 5 / 8
% Reste de la division euclidienne 5 % 8

- Opérateurs relationnels

= = Egal
!= Différent
> Strictement supérieur
< Strictement inférieur
>= Supérieur ou égal
>= Inférieur ou égal

- Opérateurs logiques

! Non
&& Et
| | Ou

- L'opérateur d'affectation

L'affectation est une opération interne qui permet d'attribuer une valeur ou le résultat d'une expression à une variable déjà déclarée.

Syntaxe:

variable = expression ;

Une expression peut prendre deux formes différentes :

expression : <opérande><opérateur><opérande>
<opérateur><opérande>

Exemple:

```
i = 5 ;
c= 'a' ;
j = 10 ;
k = 10+5 ;
k = k + 1 ;
```

On peut initialiser les variables dès leurs déclarations :

Exemple :

```
int k=0 ;
char c='A' ;
```

- Opérateurs d'incrémentation et de décrémentation

x++ ;	x=x+1 ;
x--;	x=x-1 ;

- Simplification des opérateurs arithmétiques

x += y ;	x = x + y ;
x -= y ;	x = x - y ;
x *= y ;	x = x * y ;
x /= y ;	x = x / y ;
x %= y ;	x = x % y ;

- Priorité des opérateurs

Entre les opérateurs que nous venons de présenter, nous pouvons distinguer les classes de priorités suivantes :

Priorité 1 (la plus forte): ( )
Priorité 2: !, ++, --, -
Priorité 3: *, /, %
Priorité 4: +, -
Priorité 5: <, <=, >, >=
Priorité 6: ==, !=
Priorité 7: &&
Priorité 8: ||
Priorité 9 (la plus faible): =, +=, -=, *=, /=, %=

- Dans chaque classe de priorité, les opérateurs ont la même priorité. Si nous avons une suite d'opérateurs binaires de la même classe, l'évaluation se fait en passant de la gauche vers la droite dans l'expression.

- Pour les opérateurs unaires (!, ++, --) et pour les opérateurs d'affectation (=, += , -=, *=, /=, %=), l'évaluation se fait de droite à gauche dans l'expression.

1.2.6 Les Constantes

Une constante est une donnée dont la valeur reste fixe tout au long de l'exécution d'un programme (la valeur ne peut pas être modifiée).

Syntaxe :

- Soit hors de la fonction principale main() :

```
# define nom_constante valeur
```

- Soit dans le corps de la fonction principale main() :

```
const  type nom-constante = valeur ;
```

Exemple :

```
#define pi  3.14
const pi=3.14 ;
```

1.3. Structure d'un programme C

De manière générale, un programme C consiste en la construction de blocs individuels appelés fonctions qui peuvent s'invoquer l'un à l'autre. Chaque fonction réalise une certaine tâche.

Pour pouvoir s'exécuter, un programme C doit contenir une fonction spéciale appelée **main** qui sera **le point d'entrée** de l'exécution (c'est à dire la première fonction invoquée au démarrage de l'exécution). Toutes les autres fonctions sont en fait des sous-routines.

```
/* Déclaration des bibliothèques : directives */
#include<nomfichier1.h>
#include<nomfichier2.h>
#include ″Nom_fichier.h″
…
/* Déclaration des constantes & des macros*/
#define cons1 val1
#define exp1 exp2
…
/* Déclaration des variables globales*/
 type variable1, variable2 ;
…
/* Déclaration des fonctions */
//*****************************************************
type-de-retour  Nom-fonction1(type1 arg1, ... , typeN argN)
{
Type_variable11variable11 ; /* Déclaration des variables locales*/
/* Début du corps de la fonction1 */
 instruction1 ;
 instruction2 ;
 ..
} /* Fin du corps de la fonction1 */
//*****************************************************
```

```
type-de-retour Nom-fonction2(type1 arg1, ... , typeN argN)
{
 Type_variable12variable12 ; /* Déclaration des variables locales*/
/* Début du corps de la fonction2 */
 instruction1 ;
 instruction2 ;
 …
} /* Fin du corps de la fonction2 */
...
//*****************************************************
int main ( )              /* Fonction principale */
{
Type variable3, variable4; /* Déclaration des variables locales*/
instruction1 ;
 instruction2 ;
 …
return 0 ;
}
```

- **Directives**: permettent d'incorporer un ensemble de fonctions (routines) prêtes à l'emploi. Il s'agit d'un fichier d'entête (bibliothèque) ou un fichier utilisateur.

 # include <Nom_bibliothèque.h>

 # include ″Nom_fichier.h″

Exemple:

stdio.h, math.h, string.h, … (bibliothèques)

- **Macros** : permettent de définir des constantes symboliques ou des expressions paramétrées.

Exemple :

```
# define N 10
# define carre(a)  a*a
```

- **La déclaration des variables**:

On distingue deux types:

- ***Variables globales***

Les variables déclarées au début du fichier, à l'extérieur de toutes les fonctions sont disponibles à toutes les fonctions du programme.

- ***Variables locales***

On appelle variable locale, une variable qui est déclarée au début d'un bloc d'instruction. On ne peut alors utiliser cette variable que dans ce bloc. En particulier, une variable déclarée dans une fonction ne peut être utilisée par une autre fonction.

Règles:

- Chaque instruction en C doit se terminer par un point virgule « ; ».
- Les accolades «{» et «}» correspondent au début et à la fin du bloc ou du corps des fonctions.
- Une fonction peut ne pas contenir des paramètres.
- Les commentaires en C (texte non interprété) sont soit délimités par « /* » et « */ » soit débutés par « // » dans le cas où ils s'écrivent sur une seule ligne.

1.4 Les types de base

Le C est un langage typé. Cela signifie en particulier que toute variable, constante ou fonction est d'un type précis. Le type d'un objet définit la façon dont il est représenté en mémoire.

Les types de base en C concernent les caractères, les entiers et les flottants (nombres réels). Ils sont désignés par les mots-clés suivants: **char int float double**.

2 modificateurs de type

Signe (pour les entiers seulement *char* et *int* : où *b* représente le nombre de bit)

- **signed** (par défaut) : valeur $\in [-2^{b-1}, 2^{b-1} - 1]$
- **unsigned** : valeur $\in [0, 2^{b} - 1]$

Taille mémoire

short : réduit le domaine des valeurs possibles (uniquement pour **int)**

long : augmente l'intervalle [min, max]

a) Entier : int

Codé sur 4 octets selon la plate-forme utilisée (UNIX, Windows ..).

Le nombre d'octets utilisés pour coder un entier peut être un paramètre de compilation, selon la plate-forme utilisée.

Valeurs: de -32768 à 32767 sur 2 octets et
de -2147483648 à 2147483647 sur 4 octets.

Format d'affichage: **%d.**

Les types dérivés :

- **unsigned int** ou **unsigned** : **entier non signé. Valeur de 0 à 65535 (**si sur 2 octets).
  Format d'affichage : %u
- **long int** ou **long** : entier sur 4 octets
  Format d'affichage: %ld.
- **unsigned long int** ou **unsigned long** : entier long positif.
  Format d'affichage: %lu.

Le mot clé **unsigned** permet de supprimer le bit de signe.

- **short int** ou **short** : 2 octets
  Format d'affichage: %d (ou %hd)

b) Flottant ou réel : **float** (4 octets) ou **double** (8 octets).

*Format d'affichag*e: **%f**.

Les constantes numériques sont par défaut de type double.

c) Caractère : char (1 octet)

Valeurs : -128 à 127 ou 0 à 255 pour unsigned char.

Format d'affichage : **%c** pour les caractères et **%s** pour les chaînes de caractères.

Les types de base permettent de définir des objets comme entiers, flottant ou caractère:

Type	*Description*	*Taille*	*Format*
void	type générique		
char	caractère	1 octet	%c
int	entier signé	2 ou 4 octets	%d
float	flottant	4 octets	%f
double	flottant	8 octets	%lf

Taille des types :

L'espace qu'occupent les différents types en mémoire dépend de la machine sur laquelle est implanté le compilateur. Le choix est laissé aux concepteurs des compilateurs.

Exemple:

sizeof (int)

sizeof (float)

où **sizeof** est *un opérateur* qui donne la taille en nombre d'octets du type dont le nom est entre parenthèses.

1.5 Les Fonctions d'Entrées/ Sorties

- Bibliothèque adoptée

A chaque utilisation d'une fonction d'entrée ou de sortie (lecture, écriture), il faut inclure le fichier « stdio.h » à l'entête du programme.
Ceci se traduit par l'instruction suivante : **#include <stdio.h>**
« stdio.h » est le nom d'un fichier de définitions, des fonctions et des constantes utilisées dans un programme pour des entrées ou des sorties standards.

Std → standard
i → input
o → output
h → header (entête)

1.5.1 Impression et lecture de caractères

Le mécanisme d'entrée le plus simple consiste à lire un caractère provenant de *l'entrée standard*.
Les fonctions **getchar** et **putchar** permettent respectivement de lire et d'imprimer des caractères. Il s'agit de fonctions d'entrées-sorties non formatées.

a) Fonction getchar ( )

Elle permet de renvoyer le code d'un caractère saisi à partir de l'organe standard d'entrée à savoir le clavier.
Exemple :

```
char c ;
c=getchar( ) ;
```

int c ;
c=getchar( ) ; // retourne le code ASCII du caractère entré au clavier.

b) Fonction putchar ( )

Elle permet d'afficher le caractère mis en paramètre sur l'organe standard de sortie à savoir l'écran.
Exemple :

```
char c ;
c='a' ;
putchar(c) ;
```

L'instruction putchar('a') transfère le caractère a vers le fichier standard de sortie stdout.

1.5.2 La fonction de lecture scanf

La fonction scanf permet de saisir des données au clavier et de les stocker aux adresses spécifiées par les arguments de la fonction.

Syntaxe :

scanf(<format>, <liste des adresses de variables>) ;

<format> : chaîne de caractères délimitée par **""** définissant les spécifications de conversion de types à appliquer au niveau de la lecture.
Une spécification de conversion commence par le caractère % suivi d'un type de conversion comme suit :

%d ou %i	entier relatif (int)
%u	entier naturel (unsigned), non signé
%ld	long
%x	entier en hexadécimal
%o	entier en octal
%f	float
%lf	double
%c	char
%s	chaines de caractères

Exemple :

```
char c,y ;
float x ;
int i ;
scanf(" %c ", & c) ; /* saisir un caractère et le stocker dans la variable c */
scanf(" %f%c%d",&x, &y, &i) ;
```

1.5.3 La fonction d'écriture printf

La fonction **printf** est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi.
Elle permet d'afficher sur l'écran les variables mises en paramètre selon le type concerné.
Syntaxe :

printf(<format>, <liste d'arguments>) ;

ou

printf(<chaine de controle>, <liste d'arguments>) ;

<format> : chaîne de caractères délimitée par **""** définissant les spécifications de conversion de types à appliquer au niveau de l'affichage.
La chaine de contrôle contient le texte à afficher et les spécifications de format correspondant à chaque variable.
Exemple :

```
int i ;
char c ;
i=5;
c='a' ;
printf ("%d %c", i, c) ; /* permet d'afficher l'entier 5 et le caractère a sur l'écran */
```

On peut toutefois fixer le nombre de caractères de la donnée à afficher.
Par exemple :

%3s pour une chaîne de 3 caractères,
%10d pour un entier qui s'étend sur 10 chiffres, signe inclus.

En plus du caractère donnant le type des données, on peut éventuellement préciser certains paramètres du format d'impression, qui sont spécifiés entre le % et le caractère de conversion dans l'ordre suivant :

• ***largeur minimale du champ d'impression*** : %10d spécifie qu'au moins 10 caractères seront réservés pour imprimer l'entier.

Par défaut, la donnée sera cadrée à droite du champ.
Le signe - avant le format signifie que la donnée sera cadrée à gauche du champ (%-10d).

• ***précision*** : **%.2f** signifie qu'un flottant sera imprimé avec 2 chiffres après la virgule.

NB : Lorsque la précision n'est pas spécifiée, elle correspond par défaut à 6 chiffres après la virgule.

De même %10.2f signifie que l'on réserve 12 caractères (incluant le caractère**.**) pour imprimer le flottant et que 2 d'entre eux sont destinés aux chiffres après la virgule.
Pour une chaîne de caractères, la précision correspond au ***nombre de caractères imprimés*** : %30.4s signifie que l'on réserve un champ de 30 caractères pour imprimer la chaîne mais que seulement les 4 premiers caractères seront imprimés (suivis de 26 blancs).

Chapitre 2.
Les structures de contrôle

- Introduction
- Les structures conditionnelles
- Les boucles
- La démarche itérative

2.1 Introduction

En programmation procédurale comme en algorithmique (qui respecte les contraintes fondamentales de la programmation), **l'ordre des instructions** est primordial.
Le processeur exécute les instructions dans l'ordre dans lequel elles apparaissent dans le programme. On dit que l'exécution est **séquentielle.** Une fois que le programme a fini une instruction, il passe à la suivante.
Les instructions de contrôle servent à contrôler le déroulement de l'enchaînement des instructions à l'intérieur d'un programme.
Il existe deux grands types de structures de contrôle:
- **les *structures conditionnelles*** vont permettre de n'exécuter certaines instructions que sous certaines conditions
- **les *structures répétitives***, encore appelées boucles, vont permettre de répéter des instructions un certain nombre de fois, sous certaines conditions

2.2 Les structures conditionnelles

Les structures conditionnelles permettent d'exécuter des instructions différentes en fonction de certaines conditions.
Une condition (encore appelée expression conditionnelle ou logique) est évaluée, c'est à dire qu'elle est jugée vrai ou fausse. Si elle est vraie, un traitement (une ou plusieurs instructions) est réalisé; si la condition est fausse, une autre instruction va être exécutée, et ensuite le programme va continuer normalement.
Il existe 2 types principaux de structures conditionnelles
- **les *structures alternatives*** (Si…Alors…Sinon)
- **les *structures conditionnelles*** au sens strict (Si…Alors)

L'instruction conditionnelle permet d'évaluer une condition et d'exécuter en conséquence un bloc d'instructions.

2.2.1) L'instruction conditionnelle « if »

L'opérateur de test se présente sous les deux formes :
Syntaxe :

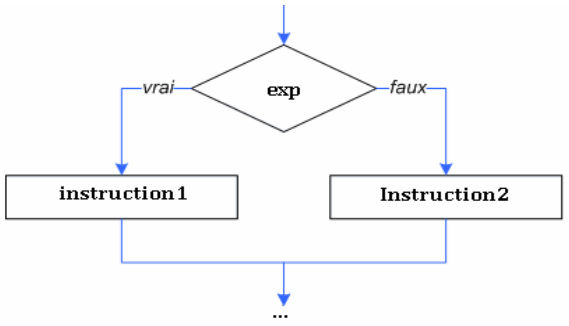
```
if (<condition>)
{
    Instructions ;
}
```

ou

```
if (<condition>)
{
     Instructions 1 ;
}
else {
     Instructions 2 ;
}
```

La condition (expression) est évaluée. Si le résultat est vrai, alors l'instruction qui suit le « if » est exécutée, et on n'exécute pas la partie du « else». Si l'expression rend un résultat égal à faux, c'est l'instruction qui suit le else (si elle existe) qui est exécutée.

Organigramme :



L'opérateur conditionnel ternaire

L'opérateur conditionnel '?:' est un opérateur ternaire.
Sa syntaxe est la suivante :

condition ? expression1 : expression2

Cette expression est égale à expression1 si condition est satisfaite, et à expression2 sinon.
Par exemple, l'expression

x >= 0 ? x : -x

Correspond à la valeur absolue d'un nombre. De même l'instruction

m = (a > b) ? a : b;

affecte à m le maximum de a et de b.

2.2.2) L'instruction conditionnelle « switch »

Lorsque les alternatives sont plus nombreuses, on peut utiliser la primitive **switch** qui désigne la valeur à étudier, suivie des instructions à exécuter en fonction des divers cas (commande **case**). Plus précisément, l'exécution des commandes commence lorsque la valeur étudiée apparaît, jusqu'à la fin (ou jusqu'au **break**). Si la valeur n'apparaît jamais, le cas default englobe toutes les valeurs :

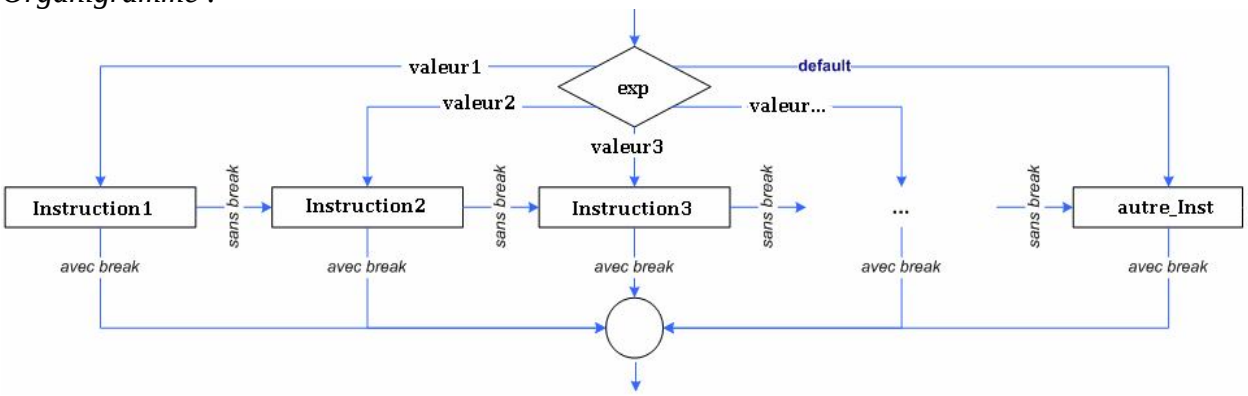
Syntaxe :

```
switch (expression) {
        case valeur1 :  Instructions1; break ;
        case valeur2 :  Instructions2; break ;
        …
        case valeurN :  InstructionN; break ;
        default :
                        autre_insts ;
        }
```

L'exécution du switch est réalisée selon les règles suivantes :

1. L'expression est évaluée comme une valeur entière ;
2. Les valeurs des « case » sont évaluées comme des constantes entières ;
3. L'exécution se fait à partir du « case » dont la valeur correspond à l'expression. Elle s'exécute en séquence jusqu'à la rencontre d'une instruction « break » ;
4. Les instructions qui suivent la condition default sont exécutées lorsque aucune constante des « case » n'est égale à la valeur retournée par l'expression ;

Organigramme :



2.3 Les boucles

Les structures itératives sont utilisées pour décrire les répétitions d'une instruction ou d'une suite d'instructions. Toute répétition d'instructions, appelée aussi **boucle** d'instructions, doit être finie et celle-ci sera contrôlée à l'aide d'une condition dont le changement de valeur provoque l'arrêt de la répétition ou la poursuite de l'exécution de ces instructions.
On distingue 3 formes de boucles: boucle for, boucle while et boucle do ..while.

2.3.1 La boucle « for … »

Elle permet d'exécuter des instructions plusieurs fois sans avoir à écrire toutes les itérations. Dans ce genre de boucle, on doit savoir *le nombre d'itérations* avant d'être dans la boucle.
Syntaxe :

```
for ( Initialisation(s) ; Test ; Modification(s))
{
   bloc d'instructions ;
}
```

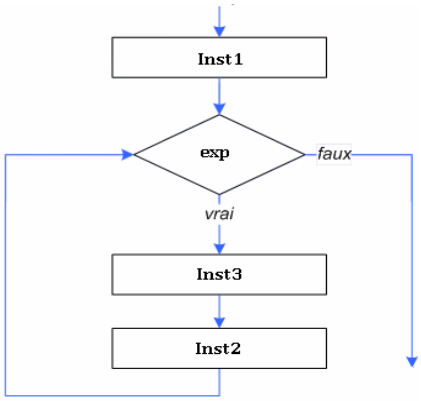
- ***Initialisation(s)*** : séparées par une virgule, ces instructions définissent les initialisations de la boucle. Elles sont exécutées une seule fois avant la première itération.

- ***Test*** : expression qui doit être égale à « faux » pour stopper la boucle. Elle est évaluée avant chaque itération.

- ***Modification(s)***: séparées par virgule, elles définissent la séquence d'instructions à réaliser entre deux itérations (avant d'effectuer l'évaluation de <test>).
- Tant que le <test> est évalué à vrai, le bloc d'instructions est exécuté.

Organigramme:

```
for (inst1 ; exp ; Inst2)
  Inst3 ;
```



Exemple :

```
for (i=0 ; i<N ; i++) {
  printf(" bonjour ");
  }
```

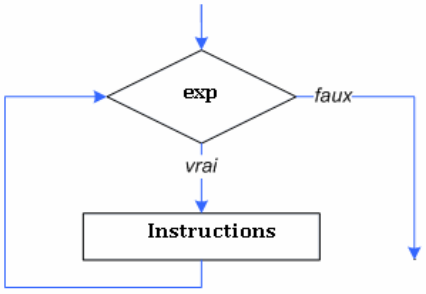
Dans cette boucle, i est le **compteur**. Il ne doit pas être modifié dans le bloc d'instructions.

2.3.2 La boucle « while … »

Contrairement à la boucle *for*, on n'est pas obligés ici de connaître le nombre d'itérations. Il n'y a pas de compteur.

```
while (expression) {
    instructions;
 }
```

L'expression est évaluée à chaque itération. Tant qu'elle est vraie, les instructions sont exécutées.
Si dès le début elle est fausse, les instructions ne sont jamais exécutées.
Organigramme:



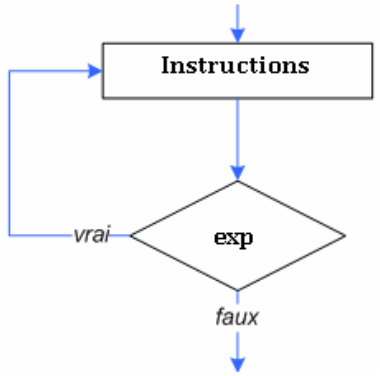
2.3.3 La boucle « do ... while… »

À la différence d'une boucle *while*, les instructions sont exécutées au moins une fois : l'expression est évaluée en fin d'itération.

```
do {
    Instruction;
    …
   } while (expression) ;
```

- La condition (expression) est évaluée à la fin de chaque itération.
- Le bloc d'instructions est exécuté au moins une fois.
- Tant que la condition est évaluée à vrai, le bloc d'instructions est exécuté.

Organigramme:



2.3 Les instructions de branchement non conditionnel

2.3.1 Branchement non conditionnel break

Il arrive fréquemment qu'en évaluant un test à l'intérieur d'une **boucle**, on aimerait arrêter brutalement la boucle ou alors « sauter » certains cas non significatifs. C'est le rôle de l'instruction « break » :

Syntaxe :

break;

On peut l'utiliser plus généralement au sein de n'importe quelle boucle (instructions for, while ou do...while) pour interrompre son déroulement et passer directement à la première instruction qui suit la boucle. C-à-d elle permet de sortir immédiatement d'une boucle for, while ou do.. while.

Exemple :

```
while (1) {
...
if (command = = ESC) break; // Sortie de la boucle
...
}
```

2.3.2 Branchement non conditionnel continue

L'instruction continue ne peut être utilisée que dans le corps d'une **boucle** (instruction for, while ou do). Elle permet de passer directement à l'itération suivante.

Syntaxe :

continue ;

Exemple :

```
for (i = -5; i < 5; i++) {
if (i = = 0)
 continue; // passer à 1 sans exécuter la suite
S=s+ 1/i ;
}
```

2.3.3 Branchement non conditionnel goto

Syntaxe :

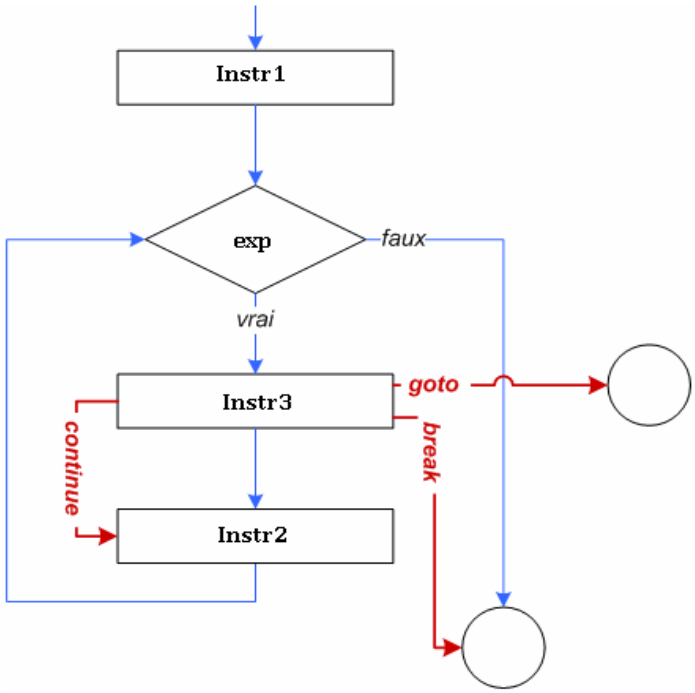
goto etiquette ;

La directive **goto** permet de brancher directement `a n'importe quel endroit de la fonction courante identifie par une étiquette. Une étiquette est un identificateur suivi du signe ":".

Exemple :

```
for ( ... )
for ( ... )
if ( erreur )
goto TRAITEMENT_ERREUR;
...
TRAITEMENT_ERREUR: // le traitement d'erreur est effectu´ e ici
printf("Erreur: ....");
...
```

Organigramme:



Chapitre 3.
Les tableaux

- Tableaux unidimensionnels
- Tableaux multidimensionnels
- Chaînes de caractères

3.1 Tableau à une seule dimension

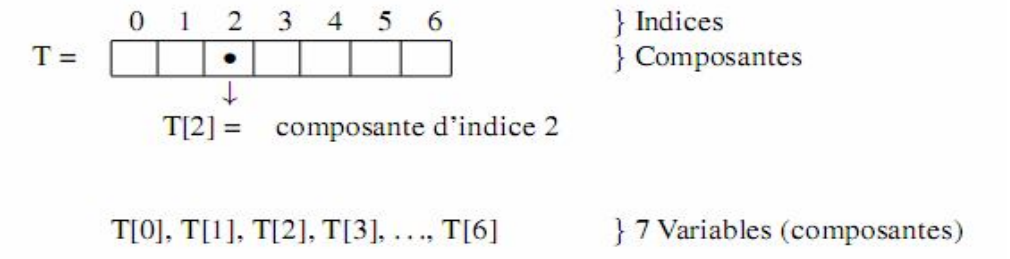
Le langage C offre des structures de données permettant de stocker les valeurs dans une entité commune. Pour traiter ces valeurs, il suffit de parcourir la structure.

3.1.1 Définition

Un tableau est un ensemble d'éléments de ***même type*** désignés par un identificateur unique ; chaque élément y est repéré par la valeur d'un entier nommé **indice**, et qui représente la position de l'élément dans le tableau.

Un tableau de base en C contient une ligne et plusieurs colonnes. Il est cependant possible de créer un tableau à plusieurs dimensions, grâce à la notion de tableaux de tableaux. Chaque élément du tableau est un tableau.

Un tableau est une structure regroupant plusieurs valeurs de *même type*, appelées *composantes* du tableau.



3.1.2 Déclaration d'un tableau

La déclaration d'un tableau provoque la réservation automatique d'une zone contigüe en mémoire.

La déclaration d'un tableau permet de :

- Spécifier un nom au tableau (Identificateur).
- Donner un type pour les éléments contenus dans le tableau.
- Eventuellement le nombre de ses éléments.

Déclaration :

<type> nomDuTableau [nombre_d_elements] ;

nomDuTableau : identificateur

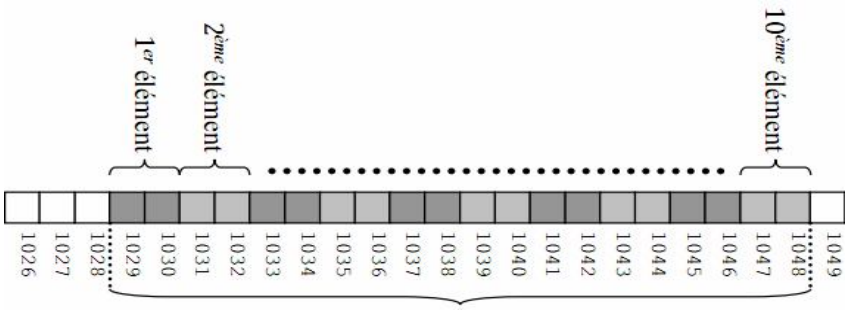
nombre_d_elements : expression constante entière et positive

- Représentation d'un tableau en mémoire :

Un tableau est représenté en mémoire de façon contiguë, c'est-à-dire que le premier élément est suivi du second, qui est directement suivi du troisième, etc.

Représentation contiguë : tous les éléments constituant le tableau sont placés en mémoire les uns après les autres.

Pour un type de données occupant p octets en mémoire, un tableau de taille n occupera n ×p octets en mémoire.
L'exemple suivant représente le tableau déclaré précédemment, qui contient 10 entiers de type int.



La dimension d'un tableau :
Dans la déclaration d'un tableau le nombre d'éléments a deux propriétés :
- il peut être une constante de type int ou un nombre entier (comme dans les exemples).

```
# define n   12
const int m = 6 ;

char t1[n] ;
char t2[m] ;
int t3[10] ;
```

- il peut être une valeur non déterminé, alors on doit maximiser lors de la déclaration.

```
int tab[100] ;
```

3.1.3 Accès aux composantes

L'accès à une composante de tableau permet de traiter cette composante comme n'importe quelle variable individuelle : on peut modifier sa valeur dans le tableau, l'utiliser pour un calcul, un affichage, etc.
L'accès d'une composante se fait via son **indice** ou position dans le tableau. En langage C, la première position a pour **indice 0**, et la dernière, à **l'indice n-1** (taille du tableau moins un).

3.1.4 Initialisation d'un tableau

On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante:

```
type nom-du-tableau [N] = {constante1, constante2, . . . , constanteN} ;
```

Par exemple:

```
#define N 5
    int t[N] = {14, 27, 3, 18, 81};
    ….
```

3.1.5 Opérations élémentaires sur un tableau 1D

Soit t un tableau de n éléments.

a. Remplissage d'un tableau

```
for(i=0 ; i<n ; i++)
scanf ("format", &t[i])
```

b. Affichage d'un tableau

```
for(i=0 ; i<n ; i++)
printf ("format   ", t[i])
```

Exemple : Ecrire un programme permettant de compter le nombre d'étudiants ayant une note sup. à 10.

```
# include<stdio.h>
# include<conio.h>
int main( )
{
        float notes[50] ;
        int n, i, cpt ;
printf(″taper le nombre d'étudiants \n″) ;
scanf(″%d″, &n) ;
printf(″taper la liste des notes \n″) ;
for ( i=0 ; i<n ;i++)
  scanf(″%f″, &notes[i]) ;
cpt=0;
for ( i=0 ; i<n ;i++)
  if(notes[i]) >10)
    cpt++;
printf(″le nombre d'étudiants ayant une note>10 est %d″, cpt) ;
getch( ) ;
return 0 ;
}
```

3.1.6 Traitement opérant sur les tableaux

En langage c, il est possible de:

- Créer des tableaux
- Remplir les valeurs dans un tableau.
- Modifier le contenu de tableau
- Trier les éléments de tableau.
- Effectuer des opérations entre les tableaux (Somme, produit, …)
- Rechercher une valeur dans un tableau.
- **…**

3.1.7 Opérations du Tri

Une opération de tri consiste à mettre en ordre une suite de valeurs stockées dans un tableau par rapport à un critère déterminé (croissant, décroissant).

Il existe de nombreux algorithmes de tri avec des performances différentes. Parmi ces algorithmes, on trouve:

- Tri par sélection
- Tri par insertion
- Tri Bulle
- Tri rapide

(Voir série 2 et 3 (TD))

3.2 Tableaux à deux dimensions

Il est possible de définir en langage C, un tableau de plusieurs dimensions grâce à la déclaration de tableaux de tableaux. L'utilisation de tableaux de plus d'une dimension est fréquemment utilisée pour les matrices.

En C, un tableau à deux dimensions (une matrice) est à interpréter comme un tableau (à une dimension) de dimension L dont chaque composante est un tableau (unidimensionnel) de dimension C.

On appelle L *le nombre de lignes*, C *le nombre de colonnes.*

L et C sont les dimensions du tableau.

Un tableau à deux dimensions contient donc L×C composantes.

3.2.1 Déclaration

int mat[L][C] ; /* L et C sont des constantes entières */

Représentation mémoire d'un tableau à deux dimensions :

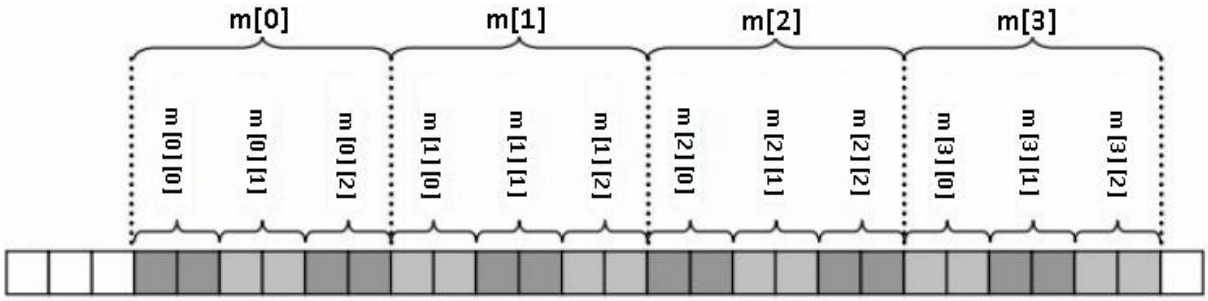
En C, un tableau multidimensionnel est considéré comme un tableau dont les éléments sont eux-mêmes des tableaux.

Nom_Tableau [*0*][*0*]	Nom_Tableau [*0*][*1*]			Nom_Tableau [*0*][*dim2-1*]
Nom_Tableau [*1*][*0*]				
Nom_Tableau [*dim1-1*][*0*]				Nom_Tableau [*dim1-1*][*dim2-1*]

Exemple : une matrice d'entiers de taille n×p :

int m[n][p];

Ici, on déclare en fait un tableau de taille n, dont chaque élément correspond à un tableau de taille p. Pour n = 4 et p = 3, ce tableau serait représenté ainsi :



3.2.2 Initialisation et réservation automatique

Lors de la déclaration d'un tableau, on peut initialiser ses composantes en indiquant la liste des valeurs respectives entre accolades.

Exemple :

```
int T[5]     = {10, 20, 30, 40, 50};
float tab[5]       = { 10.1, 70.3, 30.5, 20.0, 50.4 };
int T2_dim[2][3]   = {{1,5,7},{8,4,3}};     /* 2 lignes et 3 colonnes */
```

3.2.3 Opérations élémentaires sur un tableau 2D

Soit M un tableau à deux dimensions de L lignes et C colonnes.

a. Remplissage d'un tableau

```
for(i=0 ; i<L ; i++)
        for(j=0 ; j<C ; j++)
            scanf ("Format", &M[i][j]) ;
```

b. Affichage d'un tableau

```
for(i=0; i<L ; i++)     {
        for(j=0; j<C ; j++)
              printf ("Format   ", M[i][j]);
        printf ("\n");
        }
```

Exercice*:* Donner un programme C permettant de calculer le produit matriciel de A et B.
En multipliant une matrice A de dimensions n et m avec une matrice B de dimensions m et p on obtient une matrice C de dimensions n et p:

$$A(n,m) * B(m,p) = C(n,p)$$

$$c_{ij} = \sum_{k=1}^{k=M} (a_{ik} * b_{kj})$$

3.3 Les chaînes de caractères

En lg C, il n'existe pas de type spécial chaîne ou string. Une chaîne de caractères est traitée comme un tableau à une dimension de caractères.

Une chaîne de caractères est un tableau de caractères qui se termine par le caractère '\0'. Elle peut contenir aussi bien des caractères imprimables que non imprimables.

3.3.1 Déclaration de chaînes de caractères

La déclaration d'une chaine de caractères consiste à réserver le nombre d'octets nécessaires pour la chaîne, c.-à-d. **le nombre de caractères + 1**

Syntaxe :

```
        char nom_chaine[<taille>+1] ;
```

Remarque : Le 1 ajouté à <taille> symbolise le caractère de fin de chaîne '\0'.

Exemple :

```
char chaine[5] ;
```

3.3.2 Initialisation d'une chaîne de caractères

En général, les tableaux sont initialisés par une liste de valeurs entre accolades:

```
char ch[ ] = {'H','e','l','l','o','\0'};
```

Pour les chaines de caractères, on peut l'initialiser par une chaîne de caractères constante. Il s'agit d'une chaîne définie sous la forme d'une constante littérale est notée à l'aide des guillemets "".

Exemple : "bonjour"

Syntaxe d'initialisation :

Char nom_chaine[] = "contenu de la chaîne";

ou bien

Char nom_chaine[] = {<car1>,<car2>, …,<carN>,'\0'};

Exemple :

```
char chaine[25] = "Voici une chaîne de caractères";
```

3.3.3 Fonctions standards de manipulation des chaînes

a) Fonction scanf ( )

Elle permet de lire une chaîne de caractères à partir du clavier. La spécification de conversion en chaîne de caractères est définie par **%s**.

Syntaxe :

scanf (" %s ", Nom_chaine) ;

Exemple :

```
char chaine[30] ;
scanf (" %s ", chaine) ;
/* Attention, y a pas de référence d'adresse & */
```

b) Fonction printf ( )

Elle permet d'afficher une chaîne de caractères sur l'écran. La spécification de conversion en chaîne de caractères est définie par %s.

Syntaxe :

printf(" %s ", Nom_chaine) ;

Exemple :

```
char chaine[30] ;
scanf(" %s ", chaine) ;
printf("La chaîne de caractères saisie est: %s",chaine) ;
```

c) Fonction gets( )

Elle permet de lire une chaîne de caractères à partir du clavier jusqu'à l'identification du caractère Entrée.

La valeur de retour est la chaîne lue. Si rien, la valeur NULL est retournée.

Syntaxe :

gets(Nom_chaine)

Exemple :

```
char s[20] ;
 gets(s) ;/* saisir une chaîne de caractères et la stocker dans s */
```

d) Fonction puts( )

Elle permet d'écrire une chaîne de caractères sur l'écran.

Syntaxe :

puts(Nom_chaine) ;

Exemple :

```
char s[20] ;
gets(s) ;   /* saisir une chaîne à partir du clavier */
puts(s) ;   /* l'afficher sur l'écran */
```

3.3.4 Autres fonctions de manipulation des chaînes

De nombreuses fonctions de manipulation de chaînes, se trouvant dans la bibliothèque <**string.h**>, sont directement fournies.

a) Fonction strlen (ch)

Elle permet de récupérer la longueur d'une chaîne de caractères. Elle renvoie le nombre de caractères stockés dans la chaîne.

Exemple :

```
char s[20] ;
gets(s) ;
printf("Longueur de la chaîne : %d", strlen(s));
```

s=« bonjour », ➔ affiche: 7

b) Fonction strcpy(ch1, ch2)

Elle permet de copier une chaîne de caractères ch2 dans une autre chaîne ch1.

Exemple:

```
char s1[20],s2[20] ;
gets(s1) ;    /* saisie d¶une chaîne de caractères s1 */
strcpy(s2, s1) ;    /* copie de s1 dans s2 */
puts(s2) ;     /* affichage de s2 */
```

c) Fonction strncpy(ch1, ch2, n) :

Elle permet de copier les n premiers caractères de ch2 dans ch1.

d) Fonction strcat ( )

Elle permet d'ajouter une chaîne de caractères à la fin d'une autre.

Exemple :

```
char s1[20], s2[20];
strcpy(s1, "ABCD");
strcpy(s2,"EF");
strcat (s1,s2);  /* concaténer s2 à s1 */
puts(s1);
```

e) Fonction strcmp()

Elle permet de comparer deux chaînes de caractères.

Valeur de retour : **0** **si S1 = S2.**
< 0 (-1) **si S1 < S2.**
> 0 (1) **si S1 > S2.**

Exemple :

```
strcpy ( s1, "ABCD") ;  /* copie de la première chaîne */
strcpy ( s2, "ABCE");  /* copie de la deuxième chaîne */
printf("%d", strcmp( s1, s2));          /* la valeur -1 s'affiche sur écran */
```

f) Fonction strncmp(ch1, ch2) :

Elle permet de comparer les n premiers caractères des deux chaines de caractères ch1 et ch2.

Exemple :

ch1= ″bonsoir″ , ch2=″bonjour″

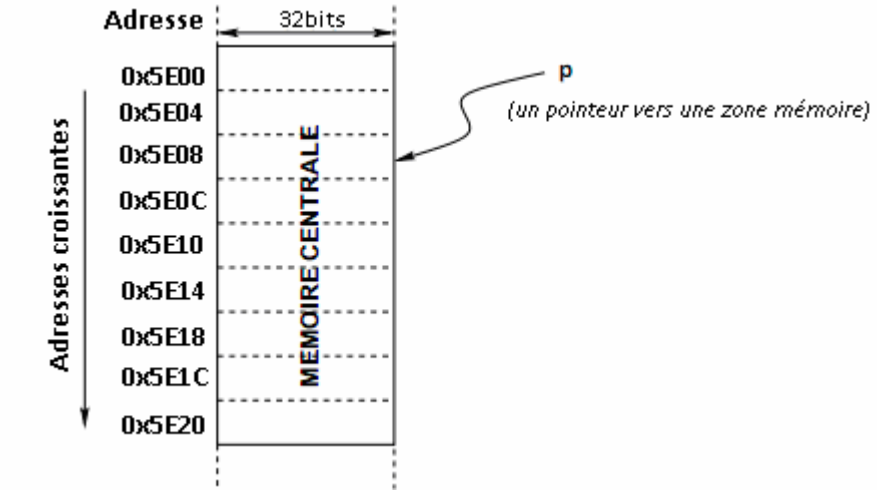
Strncmp(ch1,ch2,3) renvoie 0.

Chapitre 4
Les pointeurs

- Introduction
- Déclaration d'un pointeur
- Opérateurs de manipulation des pointeurs
- Arithmétique des pointeurs
- Allocation dynamique de mémoire
- Libération dynamique avec la fonction free
- Pointeurs et tableaux

4.1 Introduction

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle **adresse** comme l'illustre la figure suivante :



L'adressage de la mémoire centrale

Ainsi, lorsqu'on déclare une variable var de type T, l'ordinateur réserve un espace mémoire (de sizeof (T) octets) pour y stocker les valeurs de var.

Définition

Un pointeur est une variable de type adresse.

4.2 Déclaration d'un pointeur

En C, chaque pointeur est limité à un type de donnée. En effet, le type d'un pointeur dépend du type de l'objet pointé.

On déclare un pointeur par l'instruction :

type *nom-du-pointeur ;

Où type est le type de l'objet pointé.

Exemple :

```
int *pi; // pi est un pointeur vers un int
char *pc; // pc pointeur vers un char
```

4.3 Opérateurs de manipulation des pointeurs

Lors du la manipulation des pointeurs, on a besoin de:
– L'opérateur unaire **& ('adresse de'**): qui fournit l'adresse en mémoire d'une variable donnée.
– L'opérateur *** ('contenu de') :** pour accéder au contenu d'une adresse.

4.3.1 L'opérateur 'adresse de' &

L'opérateur & permet d'accéder à l'adresse d'une variable.
La syntaxe est la suivante :

&nom-variable

Dans l'exemple suivant, on définit un pointeur P qui pointe vers un entier X:

```
int * P;        // pointeur vers un entier non initialisé
int X = 14;     // variable entière initialisée par 14
P=&X;           // p pointe vers X
```

X désigne le contenu de X — &X désigne l'adresse de X
P désigne l'adresse de X — *P désigne le contenu de X

4.3.2 L'opérateur 'contenu de' : *

L'opérateur unaire d'indirection * permet d'accéder directement à la valeur de l'objet pointé.
La syntaxe est la suivante :

***nom-pointeur**

Ainsi, si P est un pointeur vers un entier X, *p désigne la valeur de X.
Remarque :
- Les opérateurs * et & ont la même priorité que les autres opérateurs unaires (la négation !, l'incrémentation ++, la décrémentation --.
- Dans une même expression, les opérateurs unaires *,&, !, ++, -- sont évalués de droite à gauche.

Exemple :
Après l'instruction

```
P = &X;
```

Les expressions suivantes, sont équivalentes:

```
Y = *P+1      ⇔   Y = X+1
*P = *P+10    ⇔   X = X+10
*P += 2       ⇔   X += 2
++*P          ⇔   ++X
(*P)++        ⇔   X++
```

4.3.3 Initialisation d'un pointeur

Par défaut, lorsque l'on déclare un pointeur p sur un objet de type T, il est non initialisé.
L'initialisation peut s'effectuer de trois façons :
- *Affectation à l'adresse d'une autre variable de p.* Si la variable est un pointeur, on peut faire l'affectation directement, sinon on doit utiliser l'opérateur &.

Exemple :

```
int *p1, *p2 ;
int X = 14;
p1 = &X;
p2 = p1;
```

- *Affectation de p à la valeur NULL*: on peut dire qu'un pointeur ne pointe sur rien en lui affectant la valeur NULL (valeur constante).

Exemple:

int * p = NULL;

- *Affectation directe de *p* (la zone mémoire pointée par p). Pour cela, il faut d'abord réserver à *p un espace-mémoire de taille adéquate (celui du type pointé par p, soit sizeof(T) octets).

4.4 Allocation dynamique de mémoire

Pour permettre une utilisation efficace de la mémoire, il est primordial de disposer de moyens pour réserver et libérer de la mémoire dynamiquement au fur et à mesure de l'exécution. C'est ce qu'on appelle **une allocation dynamique de la mémoire**. Les fonctions pour la gestion dynamique de la mémoire sont déclarées dans le fichier **stdlib.h**.

4.4.1 Allocation dynamique avec la fonction malloc

La fonction **malloc** permet de demander une allocation dynamique de mémoire d'une zone de taille nb octets lors de l'exécution.

Syntaxe :

malloc(nb_octets)

ou

calloc(nb, sizeof(type))

Cette fonction retourne un pointeur générique de type void * pointant vers la zone allouée de nb octets. En pratique, on utilise la fonction sizeof ( ) pour déterminer la valeur nb octets.

Exemple :

1) Pour initialiser un pointeur vers un entier, on écrit :

```
#include <stdlib.h>
int main() {
int *p;
p = (int*) malloc(sizeof(int)); // allocation dynamique
*p = 14;
}
```

2) Pour initialiser un pointeur vers un tableau d'entiers, on écrit :

```
int *tab;
int i;
scanf("%d", &N);
tab=(int*)malloc(N*sizeof(int));  // ou bien tab=(int*)calloc(N,sizeof(int));
for (i = 0; i < N; i++)
scanf(" %d",&tab[i]);
…
```

4.4.2 Libération dynamique avec la fonction free

La fonction free permet de libérer l'espace mémoire préalablement alloué par la fonction malloc.

Syntaxe :

free(nom_pointeur);

4.5 Pointeurs et tableaux

4.5.1 Pointeurs et tableaux à une dimension

Le nom d'un tableau représente l'adresse de son premier élément. En d'autres termes, c'est un ***pointeur constant*** qui pointe sur le premier élément de tableau.

Ainsi, la déclaration int T[10]; définit un tableau de 10 entiers. T est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit,

T a pour valeur **&T[0].**

Ainsi, après l'instruction, P = T;
le pointeur P pointe sur T[0], et

*(P+1) désigne le contenu de T[1]
*(P+2) désigne le contenu de T[2]
…
*(P+i) désigne le contenu de T[i]

a) Arithmétique des pointeurs

Le langage C offre une série d'opérations arithmétiques sur les pointeurs.

- Affectation par un pointeur sur le même type

Soient P1 et P2 deux pointeurs sur le même type de données, alors l'instruction

P1 = P2; fait pointer P1 sur le même objet que P2

- Addition et soustraction d'un nombre entier

Si P pointe sur une composante quelconque d'un tableau, alors P+1 pointe sur la composante suivante.

Plus généralement, Si P pointe sur l'élément T[i] d'un tableau, alors

P+n pointe sur T[i+n] (la $n^{ème}$ composante derrière P)
P-n pointe sur T[i-n] (la $n^{ème}$ composante devant P).

- Incrémentation et décrémentation d'un pointeur

Si P pointe sur l'élément T[i] d'un tableau, alors après l'instruction :

P++; P pointe sur T[i+1]
P+=n; P pointe sur T[i+n]
P--; P pointe sur T[i-1]
P-=n; P pointe sur T[i-n]

- Soustraction de deux pointeurs

Soient P1 et P2 deux pointeurs qui pointent dans le même tableau: P1-P2 fournit le nombre de composantes comprises entre P1 et P2.

Le résultat de la soustraction P1-P2 est

- Négatif : si P1 précède P2
- zéro: si P1 = P2
- Positif : si P2 précède P1

Plus généralement, la soustraction de deux pointeurs qui pointent dans le même tableau est équivalente à la soustraction des indices correspondants.

- Comparaison de deux pointeurs

On peut comparer deux pointeurs par <, >, <=, >=, = =, !=.

La comparaison de deux pointeurs qui pointent dans le même tableau est équivalente à la comparaison des indices correspondants.

b) Parcours d'un tableau

On peut utiliser un pointeur pour parcourir les éléments du tableau:

Exemple :

```
#define N 10
int main( ) {
        int *tab,*p;
        int i;
        tab=(int*)malloc(N*sizeof(int)); // allocation dynamique

        for (i = 0; i < N; i++)
        scanf(" %d",&tab[i]);
        ...
        for (p= tab; p < tab+N; p++) {
        printf("%d ",*p);

        }
return 0;
}
```

4.5.2 Pointeurs et tableaux à plusieurs dimensions

Un tableau à deux dimensions peut être représenté par :

- *Un pointeur* sur une zone mémoire de taille le produit des deux dimensions comme suit:

```
int M[L][C];
```

M représente l'adresse du premier élément du tableau, c'est à dire &tab[0][0].

- *Un tableau de pointeurs*. Chaque élément du tableau est alors lui-même un pointeur sur un tableau. On déclare un pointeur qui pointe sur un objet de type **type *** (deux dimensions) comme suit:

```
type **nom_pointeur;
```

L'exemple suivant illustre la déclaration d'un tableau à deux dimensions (une matrice de n lignes et m colonnes à coefficients entiers) sous forme d'un tableau de pointeurs :

```
int main() {
int n=14, m=5;
// Allocation dynamique
int **tab;
tab = (int**)malloc(n * sizeof(int*));
for (i = 0; i < n; i++)
tab[i] = (int*)malloc(m * sizeof(int));
....
// Libération
for (i = 0; i < n; i++)
        free(tab[i]);
free(tab);
return 0;
}
```

La première allocation dynamique réserve pour l'objet pointé par tab l'espace mémoire correspondant à n pointeurs sur des entiers. Ces n pointeurs correspondent aux lignes de la matrice. Les allocations dynamiques suivantes réservent pour chaque pointeur tab[i] l'espace mémoire nécessaire pour stocker m entiers.

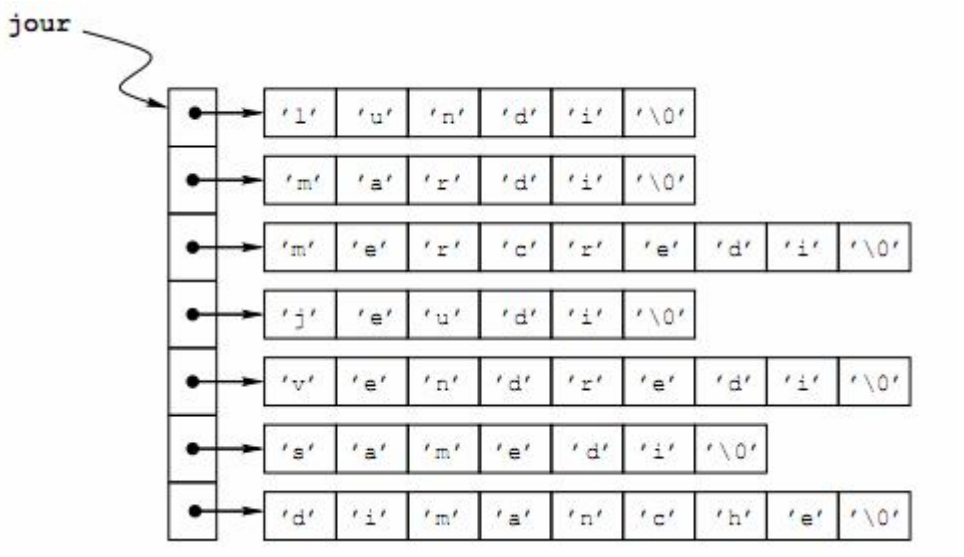
4.5.3 Pointeurs et chaînes de caractères

On peut initialiser un tableau de ce type par *Un tableau de pointeurs*. Chaque élément du tableau est alors lui-même un pointeur sur un tableau de caractères.

Exemple :

char * jour[] = {"lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"};

L'allure du tableau jour est :



Chapitre 5
Les Fonctions & les procédures

- Définition d'une fonction
- Déclaration d'une fonction
- Appel d'une fonction
- Transmission des paramètres d'une fonction
- Les tableaux et les fonctions
- La récursivité

5.1 Introduction

La conception d'un programme procède en général par des affinements successifs. On décompose le problème à résoudre en sous-problèmes, puis ces derniers à leur tour, jusqu'à obtenir des problèmes faciles à résoudre. Pour chacun des sous-problèmes, on écrit un module appelé *sous-programme*.
La résolution du problème sera composée d'un programme principal et d'un certain nombre de sous-programmes. Le programme principal a pour but d'organiser l'enchaînement des sous-programmes.
L'intérêt de l'analyse modulaire est:

- Répartir les difficultés entre les différents sous problèmes.
- Faciliter la résolution d'un problème complexe.
- Améliorer la qualité d'écriture du programme principal.
- Minimiser l'écriture du code source dans la mesure où on utilise la technique de la réutilisation.

5.2 Définition

Un sous-programme est une unité fonctionnelle formée d'un bloc d'instructions et éventuellement paramétré, que l'on déclare afin de pouvoir l'appeler par son nom en affectant des valeurs à ses paramètres (s'ils existent). Les données fournies au sous-programme et les résultats produits par ce dernier sont appelés **des arguments** ou **des paramètres**.

Un sous-programme peut être ***une procédure*** ou ***une fonction*** :
- **Une procédure** est un sous-programme ayant un nombre de paramètres, contenant un certain nombre d'instructions et sans valeur de retour.
- **Une fonction** est un sous-programme ayant un nombre de paramètres, contenant un certain nombre d'instructions et admettant au maximum un résultat unique.

Une fonction correspond à un bloc de code source portant un nom, et se compose des parties suivantes:

- **Une *en-tête*** qui comprend:
  - Le type de retour;
  - L'identificateur (Nom de fonction/procédure);
  - Les paramètres.
- **Le *corps***, qui décrit le traitement effectué par la fonction, et qui est constitué d'un *bloc* de code source.

Syntaxe :
- La syntaxe de définition d'une fonction est:

```
Type_Retour Nom_Fonction (Types Paramètres)
{
        Déclaration des variables ;
        Instructions ;
        return resultat ;
}
```

La primitive **return** permet la terminaison d'une fonction à n'importe quel endroit et de renvoyer la valeur d'une expression en tant que résultat de la fonction.

- La syntaxe de définition d'une procédure est:

```
void Nom_Procédure (Types Paramètres)
{
        Déclaration des variables ;
        Instructions ;
}
```

5.3 Déclaration d'une fonction

En C, il faut déclarer chaque fonction avant de pouvoir l'utiliser. La déclaration informe le compilateur du type des paramètres et du résultat de la fonction.
La déclaration d'une fonction se fait par un ***prototype*** de la fonction qui indique uniquement le type des données transmises et reçues par la fonction.

Syntaxe :

```
Type_Résultat NomFonction (TypePar1, TypePar2, ...);
```

5.4 Appel de sous-programmes

Pour utiliser une fonction (procédure), on doit effectuer un ***appel*** en lui passant des *expressions* correspondant aux paramètres dont elle a besoin.
L'appel d'une fonction se fait comme suit:

Variable=Nom_Fonction(Parametres_Effectifs)

L'appel d'une procédure se fait comme suit:

Nom_Procedure(Parametres_Effectifs)

Lorsqu'une fonction est appelée, un mécanisme est mis en action afin de sauvegarder plusieurs informations nécessaires (l'état courant du pg.) au déroulement du programme.
Généralement, c'est le programme principal qui appelle ou invoque des sous-programmes. Alors, il est nommé ***l'appelant*** et le sous-programme est nommé ***l'appelé***. Cependant, un sous-programme peut appeler un autre sous-programme.
Lors de l'appel d'un sous-programme, deux formes de paramètres entrent en jeu : les ***paramètres formels*** et ***les paramètres effectifs***.
Les paramètres spécifiés dans la définition de la fonction sont qualifiés de **paramètres formels**, par contre les paramètres qui seront transmis à la fonction lors de l·appel sont appelés des **paramètres effectifs**.

5.5 Passage des paramètres

En langage C, la transmission des arguments se fait par deux manières différentes: **par valeur** ou **par adresse.**

5.5.1 Passage par valeur

Le passage par valeur consiste à créer une variable locale à la fonction, appelée paramètre formel, et à l'initialiser en utilisant la valeur passée lors de l'appel de la fonction, qui est appelée paramètre effectif.

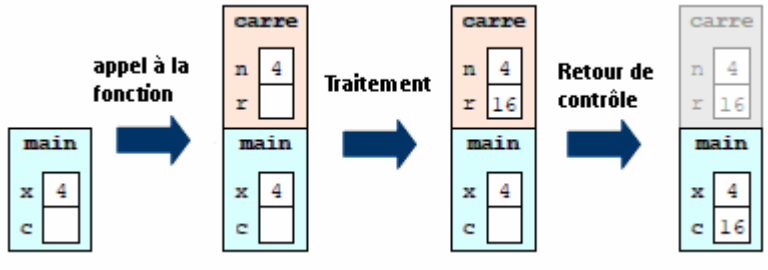
Exemple :

Soit la fonction carre, qui calcule le carré d'un entier et renvoie le résultat :

```
int carre (int n)
{ int r = n * n;
return r;
}
```

Supposons qu'on appelle cette fonction à partir de la fonction main, en utilisant une variable x, et que le résultat de la fonction carre est stocké dans une variable c pour être utilisé plus tard :

```
int main ( )
{ int x=4,c;
c = carre(x);
...
}
```



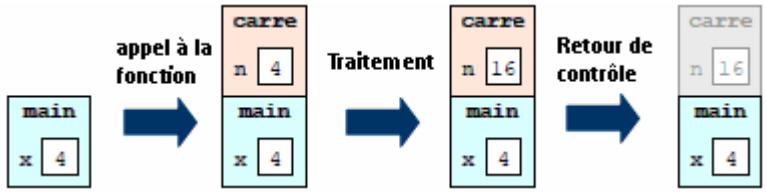
Lors d'un appel de fonction, les paramètres formels sont initialisés grâce aux valeurs correspondant aux paramètres effectifs.

En général, toute modification d'un paramètre formel est locale à la fonction. Autrement dit, quand on change la valeur d'un paramètre, cette modification n'est pas répercutée dans la fonction appelante.

Exemple : tentons de modifier les fonctions précédentes de manière à ce que *carre* modifie directement le paramètre qui lui est passé :

```
void carre(int n)
{ n = n * n;
}
int main( )
{ int x = 4;
carre(x);
...
}
```

Alors la fonction ***carre*** ne va *pas* modifier x, mais seulement la copie de x à laquelle elle a accès, c'est-à-dire la variable n :

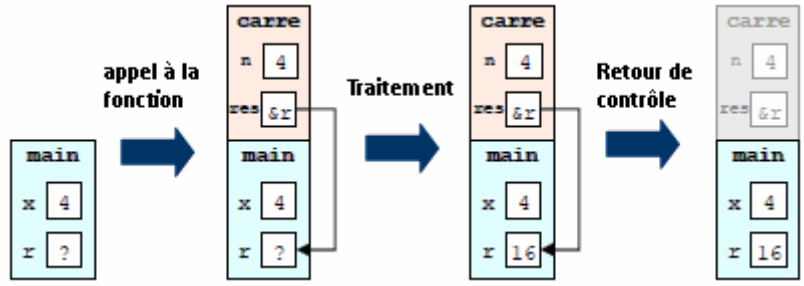


5.5.2 Passage par adresse

Pour remédier à ce problème, on doit transmettre l'adresse de la variable x et non pas sa valeur.

```
void carre(int n, int* r)
{ *r = n * n;
}
int main( )
{ int x = 4;
int r;
carre(x,&r);
}
```

Lorsqu'on appelle *carre*, le paramètre formel n est une copie de x, tandis que le paramètre res est une copie de l'adresse de r. Lorsqu'on modifie *res dans carre, on modifie en réalité r dans main. Puis la fonction carre se termine, les variables n et res sont libérés, mais la valeur 16 qui a été stockée dans r n'est pas affectée.



5.6 Tableaux et les fonctions

5.6.1 Tableau à 1D

Le passage par valeur consiste à créer une variable locale à la fonction, appelée paramètre formel, et à l'initialiser en utilisant la valeur passée lors de l'appel de la fonction, qui est appelée paramètre effectif. Si cette méthode ne pose pas de problème pour une variable de type simple, en revanche elle peut être critiquée quand il s'agit d'un tableau. En effet, il faut alors créer un nouveau tableau, aussi grand que le tableau existant, ce qui occupe de la mémoire, puis copier chacun des éléments du tableau original dans le tableau copie, ce qui peut prendre du temps.

Pour éviter cela, en langage C les tableaux sont systématiquement passés par adresse.

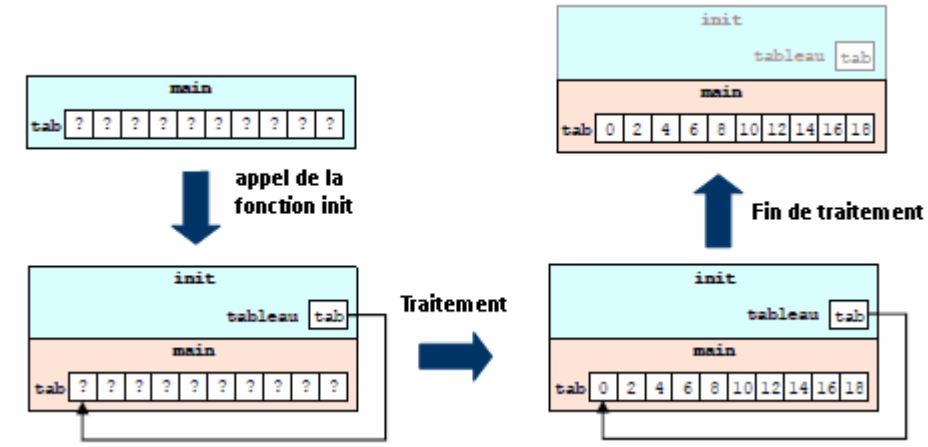
Par exemple : supposons que l'on veut écrire une fonction *init* qui initialise un tableau de taille N. La fonction peut prendre la forme suivante :

```
void init (int tableau [])
{
int  i, nb=0;
for(i=0;i<N;i++)
tableau[i] = nb;
nb=nb+2;
}
```

L'appel de la fonction « init » se fait de la façon suivante :

```
#define N=10
int main ( )
{ int tab[N];
init (tab);
}
```

La fonction ***`init`*** accède ensuite directement aux valeurs contenues dans le tableau `tab` qui existe dans la fonction `main`. La figure ci-dessous illustre le traitement réalisé en mémoire.

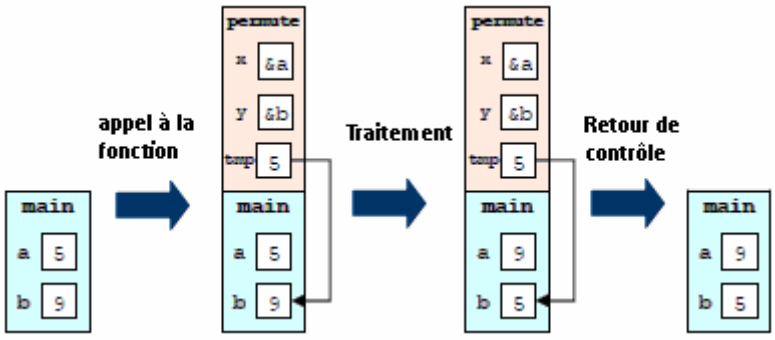


Exercice

Écrivez une fonction permute qui permute les valeurs de deux variables (i.e. qui les échange), et la fonction main qui l'appelle. Donnez la représentation graphique de l'évolution de la mémoire au cours de l'appel pour 5 et 9.

Solution :

```
void echange(int *x, int *y)
{       int temp ;
        temp = *x;
        *x = *y;
        *y = temp;
}
int main( )
{ int a=5, b=9;
printf ("Avant : a=%d et b=%d\n",a,b);
echange (&a,&b);
printf ("Après : a=%d et b=%d\n",a,b);
}
```

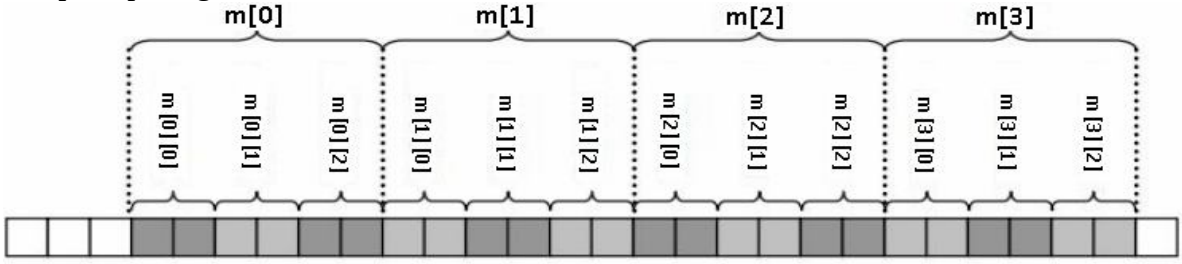


5.6.2 Tableau à 2D

Le principe est le même pour les tableaux multidimensionnels passés en paramètres : on utilise là-aussi le passage paramètre. Cependant, si la taille de la première dimension du tableau reste optionnelle, il est par contre nécessaire de spécifier les tailles des dimensions suivantes.

En effet, le compilateur a *besoin* de connaître ces dimensions pour pouvoir faire correspondre un élément du tableau à une adresse en mémoire.

Exemple : passage d'un tableau d'entiers de dimensions 4 × 3.




```
void fonction (int tab [][])
{
        ...
}
```

ou

```
void fonction (int *t)
{
        ...
}
```

5.7 Fonctions récursives

Il existe deux types de solution pour résoudre des problèmes nécessitant plusieurs traitements similaires et répétitifs: les solutions **itératives** et **récursives**.

Il s'agit d'un mécanisme permettant d'exécuter plusieurs fois la même séquence d'instruction.

Ce mécanisme se traduisent par :

- des boucles de contrôle pour les solutions itératives.
- des fonctions qui appellent-elles même pour les solutions récursives.

La récursivité est un mécanisme qui permet de définir une fonction qui fait appel à elle-même

Exemple 1: ensemble des valeurs de la fonction "factorielle" sur les entiers peut être donné par la fonction récursive suivante:

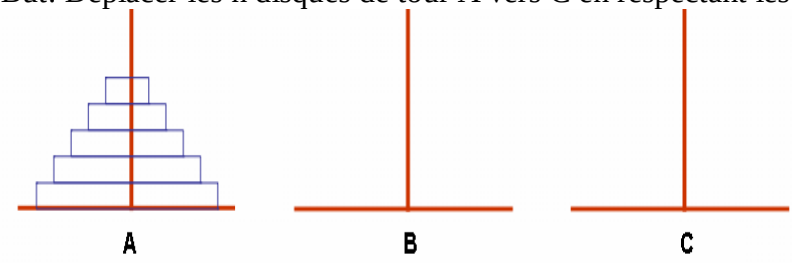
$$0! = 1$$
$$n! = n*(n-1)!$$

```
int fact(int n) {
if (n = = 0)
return 1;
else
return n * fact(n−1);
}
```

Exécution: n=4

```
Appel à fact(4)
.   4*fact(3) = ?
.   Appel à fact(3)
.   .   3*fact(2) = ?
.   .   Appel à fact(2)
.   .   .   2*fact(1) = ?
.   .   .   Appel à fact(1)
.   .   .   .   1*fact(0) = ?
.   .   .   .   Appel à fact(0)
.   .   .   .   Retour de la valeur 1
.   .   .   .   1*1
.   .   .   Retour de la valeur 1
.   .   .   2*1
.   .   Retour de la valeur 2
.   .   3*2
.   Retour de la valeur 6
.   4*6
Retour de la valeur 24
```

Exemple 2.
On considère trois axes et n disques concentriques de tailles différentes.
But: Déplacer les n disques de tour A vers C en respectant les règles suivantes:



Règles:
- Un seul disque peut être déplacé à la fois.
- Un disque ne peut jamais être déposé sur un disque plus petit.

Exercice : Recherche dichotomique (l*e tableau initial doit être trié).*
La recherche dans un tableau de taille N conduit à découper le problème en deux sous problèmes de même nature et à rechercher dans un sous tableau de taille N/2.
A chaque étape : Comparer le nombre recherché à la valeur au milieu du tableau,
- s'il y a égalité ou si le tableau est épuisé ➔ arrêter le traitement.
- si la valeur recherchée précède la valeur actuelle du tableau, continuer la recherche dans le demi-tableau à gauche de la position actuelle.
- si la valeur recherchée suit la valeur actuelle du tableau, continuer la recherche dans le demi-tableau à droite de la position actuelle.

Définir une fonction itérative et une autre récursive permettant de rechercher une valeur dans un tableau ordonné.

Chapitre 6
Les Types Structurés

→ Les types composés
- Introduction
- Structures
- Unions
- Enumérations
- Nommage des types avec typdef

6.1 Introduction

Le langage C permet au programmeur de définir ses propres types afin de pouvoir manipuler des valeurs adaptées au problème à résoudre. Ceci concerne aussi bien les types complexes que les types simples.

Type structuré : type défini par le programmeur, par opposition aux types *par défaut*, qui font partie du standard du langage C (int, char, etc.).

Dans cette section, nous aborderons :

- Les *structures*, qui sont des types complexes hétérogènes ;
- Les unions, qui sont des types simples hétérogènes ;
- Les énumérations, qui sont des types simples homogènes.

6.2 Les Structures

6.2.1 Déclaration

La déclaration d'une structure se fait grâce au mot-clé **struct**, et en précisant :

- Son ***nom*** ;
- La liste de ses ***champs***, avec pour chacun son nom et son type.

```
struct Nouv_type
{
Type1 champ1 ;
 Type2 champ2 ;
…
 Type_n champ_n ;
};
```

Exemple : on veut représenter des personnes par leurs noms, prénoms et âge.

```
struct personne
{  char nom[10];
  char prenom[10];
  short age;
};
```

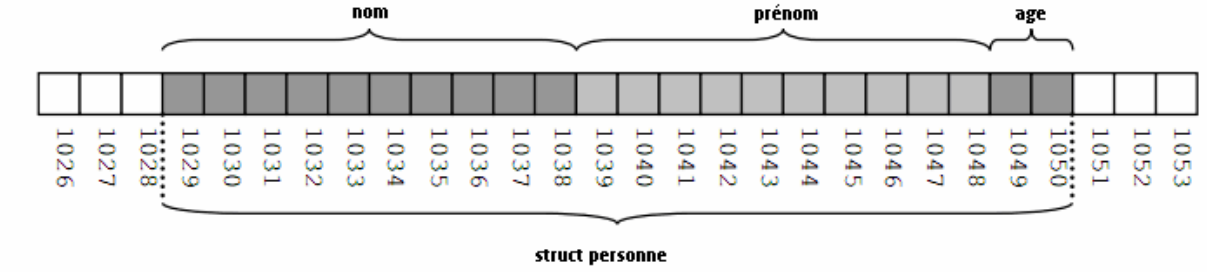
La déclaration d'une variable structure se fait de façon classique, i.e. en précisant son type puis son identificateur :

```
struct personne x;
```

On déclare une variable appelée x et dont le type est la structure personne définie précédemment.

6.2.2 Représentation en mémoire

En mémoire, les structures présentent également des similitudes avec les tableaux. Les champs qui composent une structure sont en effet placés de façon contiguë en mémoire, exactement comme les éléments formant un tableau.
Exemple : pour la structure de l'exemple précédent, on a :



La différence avec les tableaux est qu'ici, les éléments sont hétérogènes, dans le sens où ils peuvent avoir des types différents.

6.2.3 Manipulation

L'accès au contenu d'une variable structurée se fait un élément à la fois en utilisant l'opérateur. (un simple point) et le nom du champ.

Syntaxe :

variable.champ;

Cette expression se manipule comme une variable classique.
Exemple : modifier l'âge d'une personne p :

p.age = 25;

Au niveau des types : l'expression prend le type du champ concerné. Donc ici, la variable x est de type struct personne, mais x.age est de type short.

L'initialisation d'une structure peut être effectuée à la déclaration, comme pour les tableaux :

struct personne z = {"vincent", "labatut", 37};

Les valeurs des champs doivent être indiquées dans l'ordre de leur déclaration dans le type structuré.

6.2.4 Tableaux structurés

En général, un tableau pouvait être défini sur n'importe quel type, qu'il soit simple ou complexe. Par conséquent, il est possible de manipuler des tableaux de structures. La déclaration se fait comme pour un tableau classique, à l'exception du fait qu'on précise un type structuré.

Exemple : on déclare un tableau pouvant contenir 10 valeurs de type struct personne :

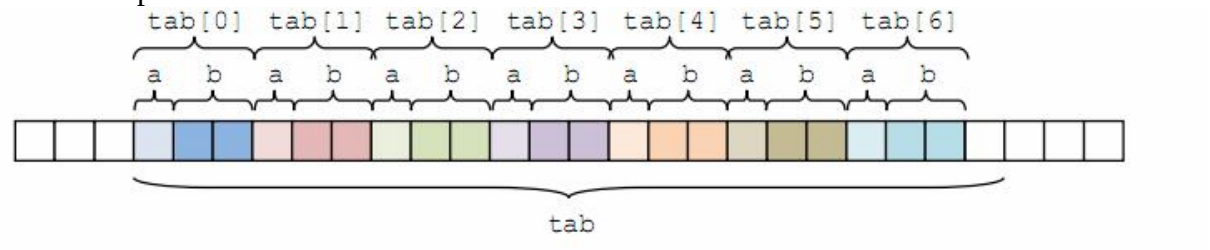
struct personne tab[10];

Pour accéder à l'élément k du tableau on utilise l'expression **tab[k]**, dont le type est struct personne. Pour accéder au champ age de l'élément k du tableau, on utilise l'expression **tab[k].age**, qui est de type short. Pour accéder à la première lettre du champ prenom de l'élément k du tableau, on utilise l'expression tab[k].prenom[0], qui est de type char.

Exemple : considérons la structure suivante :

```
struct ma_struct
{  char a;
  short b;
};
```

Soit tab un tableau, qui contient 7 valeurs de ce type :
Alors sa représentation en mémoire est :



6.2.5 Passage des Paramètres

Une variable de type structure peut bien entendu être utilisée comme paramètre d'une fonction. Le passage se déroule alors comme pour une variable de type simple : on doit choisir entre un passage par valeur ou par adresse.
Exemple :
Supposons qu'on a une fonction recevant en paramètre l'adresse d'une variable de type struct personne :

void ma_fonction(struct personne* p)

Alors la modification d'un de ses champs passe par l'utilisation de l'opérateur *.

(*p).age = 29;

Cependant, un opérateur spécial peut être utilisé pour accéder à un champ à partir de l'adresse d'une structure. Il s'agit de l'opérateur ->, qu'on utilise de la façon suivante :

p→age = 29;

6.3 Unions

6.3.1 Déclaration

Une union est une autre forme de type complexe hétérogène. À la différence d'une structure, une union ne peut contenir qu'une seule valeur simultanément. Cependant, cette valeur peut être de plusieurs types différents, c'est pourquoi une union est un type hétérogène. De plus, les types de cette valeur peuvent être simples ou complexes, c'est pourquoi l'union est qualifiée de type complexe.

La syntaxe utilisée pour déclarer un type union est très similaire à celle des structures, à la différence qu'on utilise le mot-clé union au lieu de struct.

```
Union Nouv_type
{
Type1 champ1 ;
 Type2 champ2 ;
…
 Type_n champ_n ;
};
```

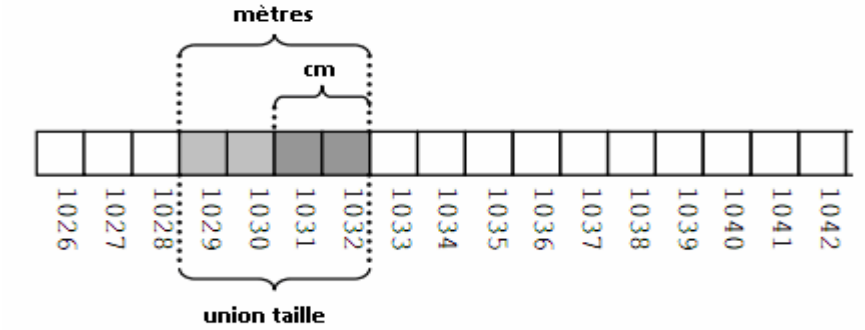
Exemple : on veut définir une union dont la valeur est une taille, qui peut être représentée soit sous forme d'un nombre de centimètres représenté par un entier, soit d'un nombre de mètres représenté par un réel :

```
union taille
{  int centimetres;
   float metres;
};
```

6.3.2 Représentation en mémoire

La place occupée en mémoire par une variable de type union correspond au champ occupant le plus d'espace dans ce type.

Exemple : dans l'union taille, il s'agit du champ mètres, qui est un float et prend donc 4 octets) alors que centimètres est un entier int, et ne prend donc que 2 octets:



6.3.3 Manipulation

Chaque champ déclaré dans une union représente une modalité d'accès à l'information stockée.

Par exemple, pour l'union taille, on peut accéder à la donnée contenue dans les 4 octets de la variable soit comme un entier int, soit comme un réel float.

Si on veut accéder à une taille exprimée en nombre de cm, on utilisera le champ centimetres :

```
t.centimetres = 172;
printf("taille en cm : %d\n",t.centimetres);
```

Si on veut accéder à une taille exprimée en nombre de m, on utilisera le champ metres :

```
t.metres = 1.72;
printf("taille en m : %f\n",t.metres);
```

Remarque

Pour les tableaux et les paramètres de fonction, les unions se manipulent de la même façon que les structures.

6.4 Énumérations

6.4.1 Déclaration

Un type énuméré est un type simple correspondant à une liste de valeurs symboliques. On définit une liste de symboles (i.e. des noms) et une variable de ce type ne peut prendre sa valeur que parmi ces symboles.

On définit un type énuméré grâce au mot-clé **enum**, suivi de **la liste de symboles** séparés par des virgules :

```
enum nom_du_type
{  valeur_1,
   valeur_2,
   ...
   valeur_n
};
```

Par la suite, quand on déclare une variable de type énuméré, celle-ci peut prendre pour valeurs uniquement l'un des symboles valeur_1, valeur_2, …, valeur_n définis dans le type.

Exemple :

```
enum booleen {faux, vrai} ;
enum couleur {rouge, vert, blue} ;

enum booleen trouve ;
 trouve=faux ;
```

6.5 Nommage de types

Il est possible de définir un nouvel identificateur pour un type, en utilisant l'instruction **typedef**, avec la syntaxe suivante :

```
typedef type nom_du_type;
```

Si le type est anonyme, alors il le fait de le nommer le rend plus facile à manipuler. S'il a déjà un nom, alors ce nouveau nom ne supprime pas l'ancien.

Remarque

La principale utilisation de typdef concerne les types structurés, unions et énumérés, car elle évite de devoir répéter les mots clés struct, union ou enum en permanence.

FIN