

# *Introduction au langage C (PC et embarqué)*

**ESIGELEC** 

*NOM :* .....

*Prénom :* .....

## Sommaire

<b>SOMMAIRE.....</b>	<b>1</b>
<b>NOTIONS D'ALGORITHMIQUE – IMPLEMENTATIONS EN LANGAGE C.....</b>	<b>6</b>
I INTERET DES ALGORITHMES .....	6
II STRUCTURE GENERALE DES ALGORITHMES .....	6
III VARIABLES .....	7
IV TESTS.....	8
V BOUCLES.....	9
5.1 Répéter tant qu'une condition n'est pas atteinte .....	9
5.2 Répéter un nombre de fois prédéfini .....	10
VI OPERATEURS .....	10
VII TABLEAUX.....	11
VIII APPEL D'UN ALGORITHME PAR UN AUTRE .....	11
IX RECAPITULATIF DES MOTS/EXPRESSIONS-CLEFS .....	12
X TRADUCTION EN LANGAGE C .....	12
10.1 Opération arithmétique sur une variable et affichage du résultat .....	12
10.2 Test .....	13
10.3 Boucle "tant que ... faire ..."	14
10.4 Boucle "faire ... tant que ..."	14
10.5 Boucle "pour ... variant de ... à ..."	15
<b>INTRODUCTION AU LANGAGE DE PROGRAMMATION C .....</b>	<b>16</b>
I   PRESENTATION GENERALE DU LANGAGE .....	16
1.1   Un langage compilé.....	16
1.2   Avantages et inconvénients du langage C .....	18
II   ASPECT GENERAL D'UN PROGRAMME.....	18
2.1   Les fonctions.....	18
2.2   Les fichiers sources en C.....	20
2.3   stdio.h .....	20
<b>LES ELEMENTS DE BASE DU LANGAGE C .....</b>	<b>21</b>
I   LES MOTS CLES .....	21
II   LES IDENTIFICATEURS.....	21
III   LES COMMENTAIRES .....	21
IV   LES CONSTANTES .....	22
4.1   Les nombres entiers.....	22
4.2   Les nombres réels.....	22
4.3   Les caractères.....	22
4.4   Les chaînes de caractères.....	22
4.5   Les expressions constantes .....	22
V   LES OPERATEURS .....	23
5.1   Les opérateurs unaires .....	23
5.2   Les opérateurs binaires .....	23
5.3   Les opérateurs ternaires.....	23
VI   LES DELIMITEURS .....	23
VII   LES SEPARATEURS .....	23
<b>TYPES DE BASE ET OPERATEURS ASSOCIES.....</b>	<b>24</b>
I   ROLE DES VARIABLES .....	24
II   LES VARIABLES DE TYPE ENTIER.....	24
III   LES VARIABLES DE TYPE REEL .....	25
3.1   La variable réelle simple précision .....	25
3.2   La variable réelle double précision.....	25

IV	LES VARIABLES DE TYPE CARACTERE .....	25
	SYNTHESE.....	26
V	LES OPERATEURS ASSOCIES .....	27
5.1	Les opérateurs arithmétiques .....	27
5.2	L'opérateur d'affectation simple .....	27
5.3	Les opérateurs d'incrément et de décrémentation .....	28
5.4	Les opérateurs d'affectation étendue .....	28
5.5	Les opérateurs relationnels .....	29
VI	LES CONVERSIONS FORCÉES PAR UNE AFFECTATION .....	30
VII	L'OPERATEUR DE CAST.....	30
	<b>LES ENTREES/SORTIES CONVERSATIONNELLES .....</b>	<b>32</b>
I	L'AFFICHAGE A L'ECRAN A L'AIDE DE LA FONCTION PRINTF .....	32
II	LA LECTURE D'INFORMATIONS A L'AIDE SCANF .....	34
2.1	Son emploi .....	34
2.2	Les précautions pour l'utilisation de scanf.....	34
	<b>LES INSTRUCTIONS.....</b>	<b>36</b>
I	L'INSTRUCTION SIMPLE - L'INSTRUCTION COMPOSEE .....	36
II	L'INSTRUCTION D'EXECUTION CONDITIONNELLE IF .....	36
2.1	Son rôle.....	36
2.2	Sa syntaxe .....	36
2.3	Instructions if imbriquées .....	37
2.4	Particularités d'emploi de l'expression en C .....	37
III	L'INSTRUCTION DE SELECTION SWITCH .....	37
3.1	Son rôle.....	37
3.2	Sa syntaxe .....	38
IV	L'OPERATEUR CONDITIONNEL .....	38
4.1	Son rôle.....	38
4.2	Sa syntaxe .....	39
	<b>LES REPETITIONS.....</b>	<b>40</b>
I	L'INSTRUCTION FOR .....	40
1.1	Son rôle.....	40
1.2	Sa syntaxe .....	40
II	L'INSTRUCTION WHILE .....	41
2.1	Son rôle.....	41
2.2	Sa syntaxe .....	42
III	L'INSTRUCTION DO WHILE.....	42
3.1	Son rôle.....	42
3.2	Sa syntaxe .....	43
	<b>LES BRANCHEMENTS INCONDITIONNELS.....</b>	<b>44</b>
I	L'INSTRUCTION BREAK.....	44
II	L'INSTRUCTION CONTINUE .....	44
III	L'INSTRUCTION NULLE : ; .....	45
	<b>LE PRE PROCESSEUR ET FONCTIONNALITES AVANCEES DU LANGAGE C .....</b>	<b>46</b>
I	LA DIRECTIVE #INCLUDE.....	46
II	LA DIRECTIVE #DEFINE .....	46
2.1	Les constantes.....	47
2.2	Les macro-instructions .....	47
III	LES ENUMERATIONS.....	47
IV	LES PARAMETRES DE LA LIGNE DE COMMANDE .....	48
V	LA COMPILATION CONDITIONNELLE.....	49
5.1	Existence liée à l'incorporation des symboles .....	50
5.2	Incorporation liée à la valeur d'une expression .....	50
	<b>LES TABLEAUX.....</b>	<b>52</b>
I	LE TABLEAU A UNE DIMENSION .....	52

1.1	Rôle du tableau.....	52
1.2	Déclaration de tableau.....	52
1.3	Initialisation de tableau.....	52
1.4	Accès aux éléments du tableau.....	52
1.5	Le nom du tableau.....	53
II	TABLEAU A PLUSIEURS DIMENSIONS.....	53
2.1	Déclaration.....	53
2.2	Initialisation.....	53
<b>LES TYPES DE VARIABLES OU CLASSES D'ALLOCATIONS.....</b>		<b>55</b>
I	LES VARIABLES INTERNES.....	55
1.1	Les variables automatiques.....	55
1.2	Les variables statiques.....	55
1.3	Les variables volatiles.....	56
II	LES VARIABLES EXTERNES.....	57
2.1	Les variables globales.....	57
2.2	Les statiques externes.....	57
III	RESUME.....	58
<b>LES FONCTIONS.....</b>		<b>59</b>
I	NOTION DE FONCTION EN C.....	59
II	DEFINITION D'UNE FONCTION.....	59
III	LES VARIABLES GLOBALES ET VARIABLES LOCALES.....	60
3.1	Les variables locales.....	60
3.2	Les variables globales.....	60
IV	FONCTIONS NE FOURNISSANT PAS UN RESULTAT (SOUVENT APPELEES PROCEDURES).....	61
V	FONCTIONS FOURNISSANT UN RESULTAT.....	61
5.1	Son rôle.....	61
5.2	Sa syntaxe.....	62
5.3	Le type de la fonction.....	63
VI	LE PASSAGE DE PARAMETRES PAR VALEUR.....	63
6.1	Les paramètres.....	63
6.2	Le passage par valeur.....	64
VII	PROTOTYPE D'UNE FONCTION.....	65
VIII	LA RECURSIVITE.....	65
8.1	Notion de récursivité.....	65
8.2	Son fonctionnement.....	66
8.3	intérêt de la récursivité.....	66
<b>LES CHAINES DE CARACTERES.....</b>		<b>67</b>
I	DEFINITION ET INITIALISATION D'UNE CHAINE.....	67
II	ECRITURE D'UNE CHAINE.....	67
III	LECTURE D'UNE CHAINE.....	68
IV	QUELQUES FONCTIONS DE TRAITEMENT DE CHAINES DE CARACTERES.....	68
4.1	Longueur d'une chaîne.....	68
4.2	Copie d'une chaîne.....	69
4.3	Concaténation de deux chaînes.....	69
4.4	Comparaison de deux chaînes.....	69
V	TABLEAUX DE CHAINES DE CARACTERES.....	70
VI	LECTURE DEPUIS UNE CHAINE DE CARACTERES.....	70
3	Ecriture formatée vers une chaîne de caractères.....	70
<b>LES POINTEURS.....</b>		<b>71</b>
I	NOTION DE POINTEUR.....	71
II	MANIPULATION DE POINTEUR.....	72
2.1	Exemple.....	72
2.2	Quelques pièges d'écriture.....	73
III	ARITHMETIQUE DES POINTEURS.....	73
3.1	Incrémentation et décrémentation de pointeurs.....	73
3.2	Addition.....	74

3.3	Soustraction.....	74
3.4	Comparaison .....	74
3.5	Multipliation et division.....	74
IV	APPLICATION AUX PASSAGES DE PARAMETRES .....	74
V	PASSAGE DE FONCTIONS COMME ARGUMENTS D'AUTRES FONCTIONS .....	75
VI	LA GESTION DYNAMIQUE DE LA MEMOIRE .....	76
6.1	Principe de l'allocation dynamique .....	76
6.2	Demander de la place en mémoire .....	76
6.3	Restituer de la place mémoire allouée.....	78
6.4	Allocation dynamique des tableaux.....	78
<b>LES STRUCTURES .....</b>		<b>82</b>
I	DECLARATION ET INITIALISATION.....	82
1.1	Notion de structure .....	82
1.2	Modèle de structure .....	82
1.3	Déclaration de variable de type structuré .....	82
1.4	Structures imbriquées .....	83
1.5	Initialisation de variables structurées .....	84
II	UTILISATION D'UNE STRUCTURE .....	84
2.1	Accès global à la structure .....	84
2.2	Accès aux champs d'une structure .....	85
III	LA STRUCTURE EN TANT QUE PARAMETRE .....	85
3.1	Portée du nom de modèle de la structure .....	85
3.2	Passage d'informations structurées entre fonctions .....	86
IV	TYPES DE DONNEES PERSONNALISES ( TYPEDEF ) .....	87
V	STRUCTURE AUTOREFERENTIELLES (LISTES CHAINEES) .....	87
5.1	Qu'est-ce qu'une liste chaînée ?.....	88
5.2	Ajout du premier élément .....	88
5.3	Ajout d'un élément en fin de liste .....	89
5.4	Liste chaînée double .....	90
<b>LES UNIONS .....</b>		<b>91</b>
I	DECLARATION D'UNE UNION .....	91
1.1	Rôle d'une union .....	91
1.2	Déclaration.....	91
II	UTILISATION DE L'UNION .....	91
2.1	Accès global.....	91
2.2	Accès aux champs de l'union .....	92
III	L'UNION EN TANT QUE PARAMETRE .....	92
3.1	Portée de l'union.....	92
3.2	Passage de paramètre .....	92
IV	EXEMPLE .....	92
<b>LES FICHIERS.....</b>		<b>93</b>
I	NOTION DE FICHIER.....	93
II	MANIPULATION GLOBALE DE FICHIER.....	93
2.1	Déclaration du fichier .....	93
2.2	Ouverture du fichier .....	93
2.3	Fermeture du fichier.....	94
III	ENTREES/SORTIES DANS LES FICHIERS .....	94
3.1	Ecriture dans un fichier.....	94
3.2	Lecture dans un fichier.....	95
IV	LES FONCTIONS DE POSITIONNEMENT DANS LES FICHIERS .....	96
V	LES ENTREES-SORTIES FORMATEES.....	96
<b>SPECIFICITES LIEES A L'EMBARQUE.....</b>		<b>98</b>
I	CODAGE BINAIRE DES NOMBRES ENTIERS .....	98
1.1	Codage des entiers positifs ("non signés", en langage C).....	98
1.2	Codage des entiers positifs ou négatifs ("signés", en langage C) .....	98
II	OPERATIONS BIT-A-BIT .....	100

2.1 Positionnement d'un bit d'une variable .....	100
2.2 Test d'un bit d'une variable .....	101
2.3 Inversion des bits d'une variable .....	103
2.4 Décalage logique.....	104
<b>ANNEXES .....</b>	<b>1</b>
1. SEQUENCES D'ECHAPPEMENT.....	1
2. LES OPERATEURS .....	1
CODES ASCII .....	2
3. PRINCIPALES FONCTIONS DES BIBLIOTHEQUES STANDARD.....	3
4. NORME D'ECRITURE DES PROGRAMMES EN C .....	9

## Notions d'algorithmique – Implémentations en Langage C

### I Intérêt des algorithmes

Pour faciliter le développement des programmes, il est important de séparer les difficultés liées au problème à résoudre des difficultés liées au langage utilisé.

Un **algorithme** est une séquence d'instructions permettant de **formaliser** un raisonnement. L'algorithme est indépendant du langage de programmation utilisé. Il constitue une étape intermédiaire entre un raisonnement portant sur la résolution d'un problème et le programme permettant de réaliser cette résolution :

raisonnement      →      algorithme      →      programme

Un algorithme doit utiliser un **formalisme** (=règles de syntaxe) précis. Il existe un très grand nombre de formalismes différents. L'important est d'en choisir un et de s'y tenir.

Ce formalisme constitue un **pseudo-code**, c'est à dire qu'il est défini précisément comme un langage de programmation mais n'est pas compréhensible par une machine. Il peut être ensuite traduit dans différents langages (Java, langage C, etc).

Dans ce qui suit, les mots-clefs du pseudo-langage défini seront mis en gras, ainsi que les noms d'algorithmes pré-définis (voir plus loin).

On s'intéresse ici à la programmation **procédurale**. Il s'agit d'un type de programmation dans lequel des morceaux de programmes sont réunis sous forme de **fonctions** (encore appelées procédures, ou routines, selon le langage de programmation). Dans ce qui suit, chaque algorithme pourra être considéré comme correspondant à une fonction.

### II Structure générale des algorithmes

Un algorithme peut être appelé par un autre.

On supposera que l'on peut transmettre des variables à un algorithme. Ces variables seront appelées **paramètres**. On aura donc :

- des paramètres d'**entrée** : utilisés par l'algorithme sans être modifiés
- des paramètres de **sortie** : modifiés par l'algorithme, ces modifications restant effectives dans la suite du déroulement du programme ayant fait appel à cet algorithme

On aura donc la structure générale suivante :

```

algorithme NomAlgorithme (NomParamètre1 : type, NomParamètre2 : type...)
variables :
    NomVariable1 : type
    NomVariable2 : type
    ...
début
    ...
fin
```

Le type pourra être "entier", "réel" ou "caractère".

*Rm* : au niveau de l'appel à cet algorithme, les variables passées comme paramètres seront appelées "arguments".

### Commentaires

Il est toujours utile (et même fortement recommandé) d'ajouter des commentaires dans un programme ; c'est également valable pour un algorithme.

On choisira comme symbole la double division (en début de chaque ligne de commentaires) : //

### Indentations

Les indentations dans un programme consistent à décaler les blocs d'instructions pour faciliter la lecture du code. Pour développer efficacement un code et permettre son test et sa maintenance, il est indispensable de les utiliser.

Exemple :

```

Algorithme PositionMinimum(t : tableau d'entiers ; taille, imin : entier)
variables :
    i, imin : entier
début
    imin = 0
    pour i = 1 à taille
        si t[i] ≤ t[imin]
            imin = i
fin

```

Dans cet exemple, ce sont les indentations qui permettent de repérer la fin de chaque bloc d'instructions.

*Rm* : pour la compréhension de cet algorithme, voir plus loin.

## III Variables

Une variable est créée par l'utilisateur. Elle permet de mémoriser provisoirement des informations, sous forme de valeurs numériques ou de texte.

Son nom peut-être quelconque, mais différent des mots-clefs pseudo-langage utilisé. Ces derniers sont appelés "**mots-réservés**".

On choisira comme symbole d'affectation le signe égal.

Exemple :

```

...
variables :
    i : entier
Début
    i = 1      //affectation de la valeur 1 à la variable i
...

```

### Lecture/écriture d'une variable

On supposera qu'il existe une fonction de lecture "lire(...)" d'une variable au clavier, ainsi qu'une fonction "écrire(...)" permettant d'afficher cette variable à l'écran.

Exemple :



```
...
lire(x)           //lecture de la variable x au clavier
écrire(x)         //écriture de la variable x à l'écran
...
```

*Rm* : on considèrera que la fonction écrire() peut également être utilisée avec un caractère ou une chaîne de caractères, définis directement en argument de la fonction.

Exemple :

```
...
écrire("bonjour")
écrire("a")
...
```

## IV Tests

On définit les mots-clefs suivants :

si – sinon

Pour simplifier la structure des algorithmes, on choisit ici de repérer la fin du test par la fin des indentations.

```
...
    si condition
        instructions 1
    sinon
        instructions 2
...                               //fin du test
```

Exemple : algorithme affichant la valeur absolue d'une variable

```
algorithme AfficherValeurAbsolue(x : réel)
début
    si x≥0
        écrire(x)
    sinon
        écrire(-x)
fin
```

```
algorithme RetournerValeurAbsolue(x : réel)
début
    si variable<0
        x=-x
fin
```

Le test peut porter sur plusieurs conditions, combinées par un ou plusieurs opérateurs logiques ET et OU. Exemple :

```
...
    si condition 1 ET condition 2
        instructions 1
...
```

```

sinon
    instructions 2
...

```

Lorsque le nombre il devient intéressant de définir un autre type de test, plus compact, pour lequel on définit l'expression-clef suivante :

selon la valeur de ...

```

selon la valeur de x
    valeur = ...
        liste d'instructions 1
    valeur = ...
        liste d'instructions 2
    ...
    autrement
        liste d'instructions n

```

## V Boucles

Une boucle est une répétition d'un ensemble d'instructions, tant qu'une condition n'est pas atteinte ou pour un nombre de fois prédéfini.

Un passage dans la boucle est appelé une **itération**.

### 5.1 Répéter tant qu'une condition n'est pas atteinte

#### 5.1.1 Avec test avant la première itération

On définit l'expression-clef suivante :

tant que ... faire

```

...
tant que condition faire
    instructions
...

```

Exemple : afficher les 10 premiers entiers naturels

```

...
    i = 0
    tant que i < 10 faire
        écrire(i)
        i = i + 1
...

```

#### 5.1.2 Avec test après la première itération

On définit l'expression-clef suivante :

faire ... tant que ...

```
...
faire
    instructions
tant que condition
...
```

Dans cette version, les instructions situées dans la boucle sont exécutées au moins une fois.  
Exemple : afficher les 10 premiers entiers naturels

```
...
i = 0
faire
    écrire(i)
    i = i + 1
tant que i < 10
...
```

## 5.2 Répéter un nombre de fois prédéfini

On définit l'expression-clef suivante :

pour ... variant de ... à ... par pas de ... faire

Ce type de boucle est adapté dans le cas où le nombre d'itérations est connu à l'avance.

```
...
pour variable variant de ValeurInitiale à ValeurFinale par pas de 1 faire
    instructions
...
```

*Rm* : pour simplifier un peu l'écriture, on pourra omettre le pas d'incrémement quand il est égal à 1 (c'est à dire dans la plupart des cas).

```
...
pour variable variant de ValeurInitiale à ValeurFinale faire
...
```

Exemple : afficher les 10 premiers entiers naturels

```
...
pour i variant de 0 à 9 faire
    écrire(i)
fin
...
```

## VI Opérateurs

Pour que le pseudo-langage soit complet, on doit définir un certain nombre d'opérateurs : logiques (portant sur les condition des tests), de comparaison (utilisés dans les condition des tests), et arithmétiques (portant sur les variables).

*Opérateurs de comparaison*

=	égal (*)
≠	différent
>	strictement supérieur
≥	supérieur ou égal
<	strictement inférieur
≤	inférieur ou égal

*Opérateurs arithmétiques*

+	addition
-	soustraction
×	multiplication
/	division
mod	modulo (reste de la division entière)

*Opérateurs logiques*

ET	ET logique
OU	OU logique

(\*) identique à l'opérateur d'affectation défini précédemment

**VII Tableaux**

Les tableaux sont des groupes de variables.

Pour accéder aux différents éléments du tableau, on utilisera le symbole [i] où i est l'indice de l'élément. On choisit de faire commencer les indices à 1.

On définit également les nouveaux types :

"tableau d'entiers", "tableau de réels", "tableau de caractères"

Exemple : algorithme affichant tous les éléments d'un tableau

```

Algorithme AfficheTableau (t : tableau d'entiers, taille : entier)
variable i : entier
début
  pour i variant de 1 à taille faire
    Afficher(t[i])
fin
```

*Rm* On ne précise pas ici comment seront affichés les éléments du tableau, en ligne ou en colonne. Cela dépendra du format d'affichage utilisé pour la fonction du langage utilisé.

**VIII Appel d'un algorithme par un autre**

On a vu précédemment que dans la programmation procédurale, un algorithme peut également être appelé une fonction. Une fonction peut en appeler une autre.

Exemple :

```
//détection du maximum de 2 valeurs
Algorithme Max2Val(val1 : entier, val2 : entier, max : entier)
début
    si val1 ≥ val2
        max = val1
    sinon
        max = val2
fin
```

```
Algorithme Comparer2ValeursLuesAuClavier()
Variables :
    A, B, max : entier
début
    Lire(A)
    Lire(B)
    Max2Val(A,B,max)
    Ecrire(max)
fin
```

## IX Récapitulatif des mots/expressions-clefs

### Général

algorithme  
variable  
début  
fin

### Types

entier  
réel  
caractère  
tableau d'entiers  
tableau de réels  
tableau de caractères

### Boucles

**pour** ... **variant** de ... **à** ... [**par pas** de ...]  
**répéter** ... **tant que** ...  
**tant que** ...

### Tests

**si** ...  
**sinon**  
**selon** la valeur de ...

## X Traduction en langage C

### 10.1 Opération arithmétique sur une variable et affichage du résultat

```
algorithme MonPremierAlgo()
variables :
    x : entier
```

```

début
    x = 15 ;
    x = x + 1 ;
    écrire(x) ;
fin

```

Programme C correspondant :

```

#include <stdio.h>
main()
{
    int x;

    x=15;
    x=x+1;
    printf("x=%d\n", x);
}

```

## 10.2 Test

Exemple : affichage de la valeur absolue d'un nombre réel

```

algorithme AfficherValeurAbsolue(x : réel)
début
si x ≥ 0
    Afficher(x)
sinon
    Afficher(-x)
fin

```

```

void AfficherValeurAbsolue(x : réel)
{
    if(x >= 0)
        printf("x=%f\n", x) ;
    else
        printf("x=%f\n", -x) ;
}

```

Exemple : test à 4 possibilités

```

algorithme Afficher4ValeursPossibles(x : réel)
début
selon la valeur de x
    valeur = 1
        Ecrire("x égale 1")
    valeur = 2
        Ecrire("x égale 2")
    valeur = 3
        Ecrire("x égale 3")
    autrement
        Ecrire("x est inférieur à zéro ou supérieur à trois")
fin

```

```

...
switch (x)
{
  case 1 : printf("x égale un\n") ;
           break ;
  case 2 : printf("x égale deux\n") ;
           break ;
  case 3 : printf("x égale trois\n") ;
           break ;
  default : printf("x est inférieur à zéro ou supérieur à trois\n") ;
            break ;
}
...

```

*Rm* : le "break" sert à éviter d'effectuer les tests suivants une fois qu'un test est positif.

### 10.3 Boucle "tant que ... faire ..."

Exemple : afficher les 10 premiers entiers naturels

```

...
variables :
  i : entier
début
  i = 0
  tant que i < 10 faire
    écrire(i)
    i = i + 1
...

```

Programme C correspondant :

```

...
int i=0;
while(i<10)
{
  printf("%d ", i);
  i=i+1;                      //ou i++ (post-incrémentation)
}
...

```

### 10.4 Boucle "faire ... tant que ..."

Exemple : afficher les 10 premiers entiers naturels

```

...
  i = 0
  faire
    écrire(i)
    i = i + 1
  tant que i < 10
...

```

Programme C correspondant :

```
...  
int i=0;  
do  
{  
    printf("%d ", i);  
    i=i+1;  
} while(i<10);  
...
```

### 10.5 Boucle "pour ... variant de ... à ..."

Exemple : afficher les 10 premiers entiers naturels

```
...  
    pour i variant de 0 à 9 faire  
        écrire(i)  
...
```

Programme C correspondant :

```
...  
for(i=0 ; i<10 ; i++)  
{  
    printf("%d ", i);  
}  
...
```



## Introduction au langage de programmation C

### I Présentation générale du langage

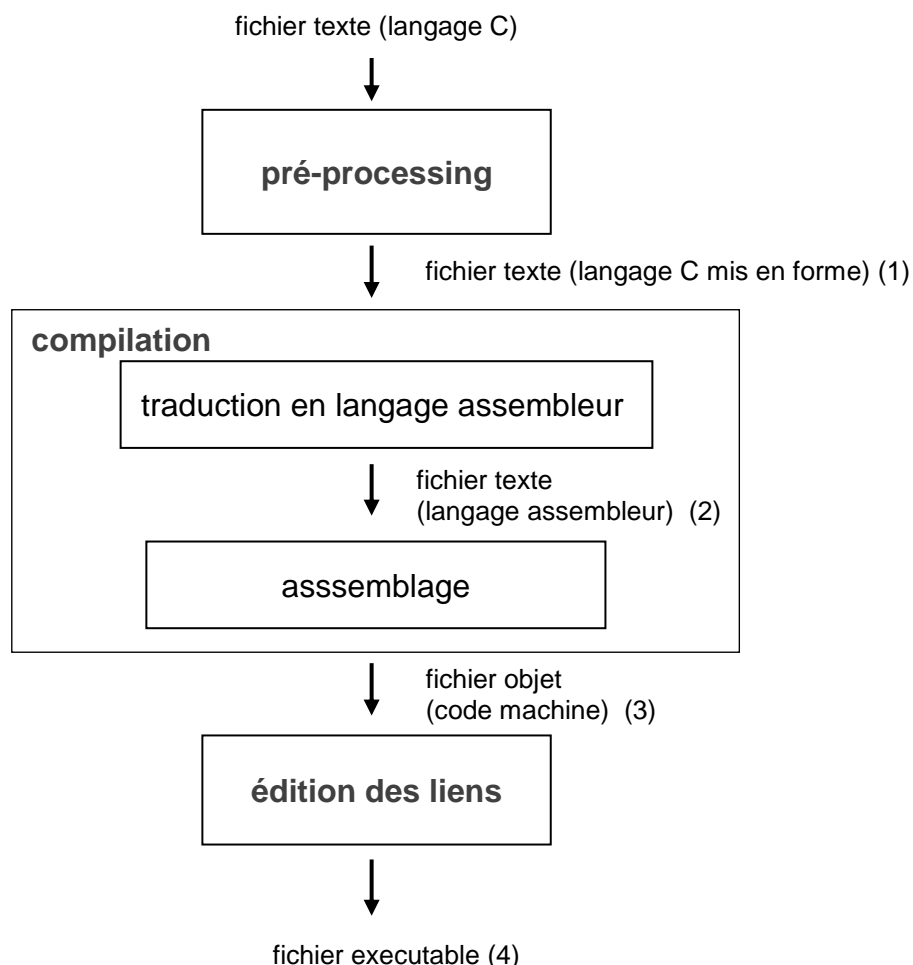
Le C est né dans les laboratoires de Bell, dans les années 1970. Il est adapté au développement de logiciels de base et des applications de haut niveau tels que des applications scientifiques, des logiciels de traitement de texte, des bases de données et l'écriture de systèmes d'exploitation (cf. le noyau d'Unix).

#### 1.1 Un langage compilé

##### 1.1.1 Notion de compilation

Par opposition à un langage interprété comme le Basic ou le Java, un programme écrit en C doit être compilé avant de pouvoir être exécuté. La compilation aboutit à la génération d'un fichier exécutable.

La compilation est un processus permettant de générer un fichier exécutable (contenant du langage machine) à partir d'un fichier texte (contenant le programme en C). Elle est composée d'un certain nombre d'étapes, représentées sur le schéma ci-dessous :



(1) L'étape de pre-processing comporte notamment :

- l'inclusion des fichiers d'entête (fichiers d'extension .h, inclus par la directive `#include`) dans le fichier .c
- la gestion des autres directives de compilation (commençant par le symbole `#` ; se reporter au cours)
- la suppression des commentaires,
- la vérification de la syntaxe du langage C
- la vérification de la cohérence du type des variables utilisées
- ...

Les erreurs détectées sont de 2 types :

- avertissements (Warning) ; il existe plusieurs niveaux d'affichage de ces avertissements (paramétrable par une option de compilation)
- erreurs bloquantes (commentaire non terminé, bloc ouvert ("`{`") et non fermé, etc)

*Remarque* : ce fichier est temporaire ; il est possible de le visualiser avec l'option E (ex. : `gcc mon_prog.c -E > mon_prog.i`)

- (2) Le langage assembleur est spécifique au processeur pour lequel la compilation est destinée

*Remarque* : ce fichier est temporaire ; il porte l'extension .s ; il est possible de le sauvegarder avec l'option S (ex. : `gcc -S mon_prog.i`).

- (3) Le code machine est une succession de valeurs numériques qui n'est plus compréhensible (directement) par le programmeur.

Ce code est contenu dans un fichier dit "fichier objet" (d'extension .o).

- (4) Ce fichier porte le nom "a.out" s'il n'y a pas d'autre nom spécifié par l'utilisateur.

Le fichier objet généré à l'étape précédente ne contient pas encore le code-machine des fonctions utilisées dans le programme C. Ce code se trouve dans un autre fichier objet, de la librairie standard s'il s'agit d'une fonction prédéfinie du C (comme `printf`), ou dans un autre fichier .o s'il s'agit d'une fonction définie par le programmeur dans un autre fichier C. L'éditeur de lien (permet de faire fonctionner ensemble ces différents fichiers objet).

### 1.1.2 Mise en pratique

Concrètement, la compilation est réalisée à l'aide d'un outil logiciel appelé compilateur. Il en existe de nombreux exemplaires disponibles. Dans notre cas on utilisera gcc (GNU).

La compilation peut être effectuée en mode "ligne de commande" (ex. : utilitaire "Invite de commandes" sous Windows ou terminal texte sous Linux), en par utilisation d'un environnement de développement intégré (IDE : Integrated Development Environment).

Dans notre cas, on utilisera l'IDE Code::Blocks, gratuit, open-source et multi-plateformes.

### 1.1.3 Compilation native vs cross-compilation

Quand un programme C est destiné à être exécuté sur la même machine (et donc avec le même processeur) que celle sur laquelle il est compilé, on parle de **compilation native**.

Au contraire, quand la compilation et l'exécution sont effectués avec deux processeurs différents, on parle de **cross-compilation**. Exemple : la compilation d'un programme destiné à être exécuté sur un micro-contrôleur MSP430 peut être effectuée à l'aide de l'IDE CCS (Code Composer Studio) de Texas Instrument, qui inclue un cross-compilateur.

## 1.2 Avantages et inconvénients du langage C

### 1.2.1 Avantages

Le C dispose de fonctions explicitement appelables (alloc, free, open, close, write, read, getc, putc, printf, etc...)

De plus sous Unix, on a des fonctions de synchronisation et de gestion de processus parallèles (fork, exec, wait, pipe, signal, etc ...)

D'autres avantages tels que la récursivité des fonctions, l'utilisation des fonctions du système d'exploitation, un jeu riche d'opérateurs et la permissivité sur la manipulation des types sont également disponibles.

☞ Il existe des compilateurs C pour pratiquement toutes les architectures système, ce qui rend le langage très portable. Associé au fait que les données soient très proches de l'architecture des machines et plus particulièrement pour les mots, les registres, les octets et les adresses, le langage C est particulièrement exploité dans le cadre de développements pour l'embarqué.

### 1.2.2 Inconvénients

Le principal inconvénient est le fait que les données sont trop proches de l'architecture des machines et plus particulièrement pour les mots, les registres, les octets et les adresses.

De plus, le C ne permet pas de manipuler directement les chaînes de caractères, les tableaux et les listes comme un tout.

## II Aspect général d'un programme

La structure d'un programme en C est faite d'un ensemble de fonctions définies dans un ou plusieurs fichiers compilables.

### 2.1 Les fonctions

Une fonction est un sous programme auquel on transmet une liste d'arguments (parfois vide) et qui retourne une valeur.

La structure générale d'une fonction est la suivante:

```
Nom_de_la_fonction (déclaration et liste des arguments)
{
    Déclaration des variables locales;
    Suite d'instructions;
}
```

**Remarques:**

- ↳ L'accolade ouvrante, fait office de début de programme, tandis que l'accolade fermante fait office de fin de programme. A l'intérieur se trouve le corps du programme.
- ↳ Une fonction particulière **main** joue le rôle de fonction principale qui a le contrôle au niveau de l'exécution du programme.
- ↳ A la fin de chaque ligne se trouve un point virgule (;). Ce délimiteur termine obligatoirement chaque déclaration de variable et chaque instruction simple.

**Exemple (version PC):**

```
#include <stdio.h>

main()
{
    int i, z, x, y;
    x=2;
    y=10;
    z=1;
    i=1;
    while (i <= y)
    {
        z = z * x;
        i++;
    }
    printf ("\n2 puissance 10 = %d\n", z);
}
```

**ou (version embarqué):**

```
int main()
{
    int i, z, x, y;
    x=2;
    y=10;
    z=1;
    i=1;
    while (i <= y)
    {
        z = z * x;
        i++;
    }
    return (z);
}
```

## 2.2 Les fichiers sources en C

Les fonctions physiquement peuvent être regroupées dans un même fichier ou dans des fichiers sources différents. Une directive d'inclusion des fichiers sources **#include** permet d'inclure à tout endroit dans le fichier principal des fichiers sources secondaires.

## 2.3 `stdio.h`

Les premières instructions utilisées dans un programme C sont en général des instructions d'entrées/sorties, qui permettent de dialoguer avec le terminal. Ces instructions ont leur définition dans un fichier appelé `stdio.h`.

Ainsi, dans la plupart des programmes développés, il faudra inclure ce fichier afin de pouvoir disposer de ces fonctions à l'aide de la directive: **#include** `<stdio.h>`.

On prendra pour habitude d'utiliser la notation `<stdio.h>` pour les fichiers prédéfinis et `"prog.h"` pour les fichiers utilisateur.

La plupart des fonctions C sont disponibles dans d'autres bibliothèques. C'est pourquoi dès que l'on utilise une fonction pré-définie, il ne faut pas oublier d'inclure le fichier qui en contient la définition.



*La bibliothèque **`stdio.h`** n'est pas utilisée pour l'embarqué (cf. exemple précédent)*

## Les éléments de base du langage C

### I Les mots clés

Comme tous les langages, le C possède des mots réservés, ayant une signification précise. Ces mots clés sont nécessaires à la sémantique du langage. Ils ne peuvent pas être redéfinis. Le C est un langage particulièrement simple de par le peu de mots qu'il définit.


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

### II Les identificateurs

Un identificateur est un nom permettant de définir un objet manipulé dans le programme. En particulier, un identificateur est associé à chaque constante, à chaque variable, à chaque type de variable défini par l'utilisateur, à chaque nom de fonction.

Tout identificateur dans un programme C doit être déclaré avant son utilisation. Le compilateur teste, en effet, la validité des grandeurs utilisées.

Un identificateur est formé exclusivement d'une suite de lettres (a...z,A...Z), de chiffres (0...9) et du caractère de soulignement ( \_ ). Il commence obligatoirement par une lettre et ne comporte aucun espace car celui-ci est considéré comme un séparateur (à sa rencontre, le compilateur considère qu'un autre identificateur commence).

 En C les minuscules et les majuscules sont autorisées mais ne sont pas *équivalentes*. Ainsi les identificateurs *TOTO*, *tOTO* et *toto* désignent des objets différents.

### III Les commentaires

Comme tout langage évolué, le langage C autorise la présence de commentaires dans les programmes sources. Les commentaires sont indispensables pour la clarté et la lisibilité d'un programme ainsi que pour sa maintenance. Ils sont formés de caractères quelconques placés entre `/*` et `*/`.

```
/* Ceci est un commentaire */
/* Ceci est un autre
   commentaire écrit
   sur plusieurs lignes */
/* Ceci /* est */ incorrect */
```

Il existe aussi la possibilité d'utiliser le commentaire monoligne du C++ :

```
// Ceci est un commentaire
```

## IV Les constantes

Le C offre plusieurs types de constantes.

### 4.1 Les nombres entiers

Ces constantes entières peuvent être écrites en décimal, en octal (016), en hexadécimal (0x ou 0X).

### 4.2 Les nombres réels

Les nombres réels se composent d'une partie entière, d'un point et d'une partie fractionnaire.

**Exemple:**     18.14 E17

### 4.3 Les caractères

'A' ... 'Z'. Ils sont entre quote simple.

### 4.4 Les chaînes de caractères

Ces constantes sont des suites de caractères. En fait, il s'agit d'un tableau de caractères. Elles sont entre double quote.

**Exemple:**     **char** chaîne[] = "Hello World";

**Remarque :** En C les chaînes de caractères se terminent automatiquement par \0 qui compte pour un caractère.

H	e	l	l	o		W	o	r	l	d	\0								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

### 4.5 Les expressions constantes

C'est une expression composée de constantes reliées entre elles par des opérateurs.

**Exemple:**     4 + 8  
                  1 < 8

## V Les opérateurs

Il existe trois types d'opérateurs:

### 5.1 Les opérateurs unaires

Les opérateurs unaires portent sur un terme seulement.

**Exemples :**     $-n$   
                   $-x + y$

### 5.2 Les opérateurs binaires

Ces opérateurs portent sur deux termes. Ce sont entre autres les opérateurs classiques tels que: l'addition, la soustraction, la multiplication et la division.

### 5.3 Les opérateurs ternaires

Ils portent sur trois expressions.

## VI Les délimiteurs

Ce sont des caractères spéciaux qui permettent au compilateur de reconnaître les différentes unités syntaxiques du langage.

On rencontre:


- ↪ le point virgule ( ; ): il termine une déclaration de variables ou une instruction.
- ↪ la virgule ( , ): elle sépare des éléments dans une liste.
- ↪ les parenthèses ( ): elles encadrent une liste d'arguments ou de paramètres.
- ↪ les crochets [ ]: ils représentent la dimension ou un indice d'un tableau.
- ↪ les accolades { }: elles encadrent un bloc d'instructions ou une liste.

## VII Les séparateurs

Les séparateurs sont: l'espace, le caractère de tabulation (`\t`), le caractère de passage à la ligne (`\n`).

**Remarque :** le C autorise une "mise en page" parfaitement libre. En particulier une instruction peut s'étendre sur un nombre quelconque de lignes et une même ligne peut comporter autant d'instructions que voulu.

Bien entendu, cette liberté de mise en page possède des contreparties. Notamment le risque existe, si l'on ne prend pas garde, d'aboutir à des programmes peu lisibles.

 Une proposition de norme d'écriture est fournie (document sur Moodle : **Normes de programmation en C**)



## Types de base et opérateurs associés

### I Rôle des variables

Toute variable utilisée dans un programme C doit avoir été définie au préalable. Cette déclaration précise différents éléments:

- ✎ Le nom de la variable, c'est-à-dire son *identificateur*.
- ✎ Son *type*, celui-ci indique l'ensemble des valeurs que peut prendre la variable. De plus, il précise aussi la taille occupée en mémoire par la variable.
- ✎ Une *valeur initiale*, implicite ou explicite, éventuellement accordée à cette variable.

Les types de base sont des types scalaires: nombres entiers, nombres réels et caractères. Les autres types dérivent de ces trois types de base.

### II Les variables de type entier

Ces variables occupent 2 ou 4 octets en mémoire selon l'architecture 16 ou 32 bits.

☞ cf. tableau page 14.

Elles vont (en 16 bits)      de -32768 à 32767 en décimal  
                                      de 8000 à 7FFF en hexadécimal  
                                      de 1000 0000 0000 0000 à 0111 1111 1111 1111 en binaire.

Une variable de type entier se définit à l'aide du mot clé **int**. Cette déclaration se termine par un point virgule.

**Exemple :**      `int i;`

Il est possible de définir plusieurs variables dans une seule déclaration.

**Exemple:**      `int k, l, m;`

La déclaration **int** n'est pas unique et se reporte autant de fois que nécessaire.

**Exemple:**      `int i;`  
                      `int k, l, m;`

Les variables peuvent être initialisées lors de leur déclaration.

**Exemple:**      `int NombreMin = 2, NombreMax = 10;`

**Remarque:**

On peut ajouter à la définition **int** des "attributs" qui agissent sur la taille de l'emplacement mémoire (mot clés **short** ou **long** ) ou sur le mode de représentation, c'est à dire signé ou non signé (mots clés **unsigned** ou par défaut **signed**).

**Exemple :** déclaration et initialisation de variables entières :

```
int somme;
unsigned int largeur, hauteur = 20;
long int nb = 55000000;
```

**III Les variables de type réel**

Le nombre réel s'écrit sous forme décimale ou sous forme scientifique avec exposant (lettre e ou E) et mantisse.

**3.1 La variable réelle simple précision**

Elle est déclarée **float** et occupe 4 octets en mémoire et est représentée sur 32 bits.

31	30	23	22	0
signe	exposant sur 8 bits		mantisse sur 23 bits	

La taille de la mantisse garantit une précision de six chiffres après la virgule

**Exemple :** **float** Abscisse = 123.36, Ordonnee = 4.2E-15;

**3.2 La variable réelle double précision**

Elle est déclarée **double**. Elle occupe 8 octets en mémoire. Sa représentation binaire est organisée comme celle de la représentation du réel simple précision. Seules les tailles de la mantisse et d'exposant sont augmentées.

63	62	52	51	0
signe	exposant sur 11 bits		mantisse sur 52 bits	

La taille de la mantisse garantit une précision de 15 chiffres décimaux après la virgule.

**Exemple :** déclaration et initialisation de variables réelles :

```
float produit;
double reel_1, reel_2 = 9.1E-31;
```

**IV Les variables de type caractère**

Une variable de type caractère se définit à l'aide du mot clé **char** et occupe toujours un seul octet en mémoire. Sa représentation binaire est sur 8 bits et est la même que le caractère soit déclaré signé ou non signé.

Ces variables prennent n'importe quelle valeur du code ASCII. Il y a donc 256 valeurs de 0 à 0xFF. Les valeurs peuvent être imposées à l'initialisation ou dynamiquement par affectation.

La façon la plus simple et la plus lisible de donner une valeur à une variable de type **char** consiste à placer le caractère voulu entre apostrophe ( ' ).

**Exemple :** définition et initialisation de variables caractères :

```
char Lettre1 = 'e';
char ponctuation = '?';
char ligne_suivante = '\x00A';    /* 00A=code ASCII hexadécimal du saut de
                                   ligne, identique à '\n' */
```

Le type **char** est par défaut **signed** ou **unsigned** selon le compilateur utilisé.

Certains caractères très courants ont une représentation conventionnelle, pour le C, à l'aide du symbole \ suivi d'une lettre. Ainsi, la fin de ligne se représente par \n.

**Exemple :**

\b	effacement arrière
\f	saut de page
\n	saut de ligne
\r	retour au début de la ligne
\t	avance d'une tabulation horizontale
\'	introduction du caractère apostrophe
\0	caractère nul, marque la fin des chaînes de caractères.

**Remarque:** Il ne faut pas confondre l'apostrophe simple ' avec le guillemet " qui s'utilise pour les chaînes de caractères.

## Synthèse

Les déclarations de type entier:

Déclaration	Valeurs possibles	Place occupée
char	0 à 255 (cf. codification ASCII)	1 octet
short int	-32768 à 32767	2 octets
unsigned short int	0 à 65535	2 octets
int	-32768 à 32767	2 octets
	-2147483647 à 2147483647	4 octets
unsigned int	0 à 65535	2 octets
	0 à 4294967295	4 octets
long int	-2147483648 à 2147483647	4 octets
unsigned long int	0 à 4294967295	4 octets
float	1.4E-45 à 3.4028235E38	4 octets
double	4.9E-324 à 1.8E308	8 octets

## V Les opérateurs associés

### 5.1 Les opérateurs arithmétiques

#### 5.1.1 Les opérateurs arithmétiques unaires

Les opérateurs unaires portent sur un terme seulement. Il s'agit du - inversant la valeur.

**Exemple :** -a;

#### 5.1.2 Les opérateurs arithmétiques binaires

Ce sont les opérateurs classiques: + - \* / % (modulo=reste de la division entière).

**Exemples :** 4+8    8-4    8\*4    8\*(-4)    8/4    8/(-4)    8%3    8%-3    -8%3

**Remarques:** ➤ faire attention aux règles de priorités des opérateurs algébriques (cf annexes).

➤ les opérateurs binaires ne sont définis que pour des opérandes de même type (ce qui implique des conversions automatiques, voir VI)) et ils fournissent un résultat du type de l'opérande occupant le plus de place en mémoire.

Par exemple 5.0/2 est le quotient d'une valeur de type **double** et d'une valeur de type **int**. L'entier est converti en double et l'opérateur quotient / fournit le résultat de type **double** 2.5. Par contre, 5/2 est le quotient de deux valeurs de type **int** et le résultat est 2.

#### 5.1.3 Les priorités relatives des opérateurs

Lorsque plusieurs opérateurs apparaissent dans une même expression, il est nécessaire de savoir dans quel ordre ils sont mis en jeu. En C, comme dans les autres langages, les règles rejoignent celles de l'algèbre traditionnelle.

Les opérateurs unaires + et - ont la priorité la plus élevée. On trouve ensuite, à un même niveau, les opérateurs \*, / et %. Enfin, sur un dernier niveau, apparaissent les opérateurs binaires + et -.

En cas de priorités identiques, les calculs s'effectuent de "gauche à droite".

Enfin des parenthèses permettent d'outrepasser ces règles de priorité, en forçant le calcul préalable de l'expression qu'elles contiennent.

### 5.2 L'opérateur d'affectation simple

L'opérateur d'affectation simple est très souvent utilisé.

Compteur = 5; a pour but de mettre la valeur 5 dans la variable compteur. La valeur se trouve toujours à droite du signe égal (=) et peut être le résultat d'une expression ou d'une fonction.

**Exemple :** c = b + 3;

L'expression b + 3 est évaluée et donne à c la valeur de l'évaluation.

La faible priorité de cet opérateur = (elle est inférieure à celle de tous les opérateurs arithmétiques et de comparaison) fait qu'il y a d'abord évaluation de l'expression  $b+3$ .

**Exemple :**

```
max = 5 * 6 + 3/4;
puiss = puissance(2, 10);
a = b = c = 0;
//l'associativité se fait de la droite vers la gauche
```

**Exemple :**

```
#include <stdio.h>
main()
{
    int i, k, t, Pos1 = 50, Pos2 = 100, Pos3 = 7;
    i = -10;
    t = Pos1 * Pos2; //t = 5000
    k = t * Pos3;    //k = -30536 au lieu de 35000
}
```

**Question:** Pourquoi obtient-on une erreur dans cet exemple ?

### 5.3 Les opérateurs d'incrément et de décrémentation

Souvent dans un programme, nous avons à faire :  $i = i + 1$ ; ou  $j = j - 1$ ;

Le langage C propose d'écrire ces expressions de façon simple. Ainsi,

$i = i + 1$ ; peut s'écrire en C:  $i++$ ;  
 $j = j - 1$ ; peut s'écrire en C:  $j--$ ;

Ces opérateurs d'incrément et de décrémentation peuvent être placés avant ou après la variable.

**Exemples:**

```
#include <stdio.h>
main()
{
    int x = 15;
    y = x++;           //est équivalent à y = x; et x++;
    printf("x=%d, y=%d", x, y); //x vaut 16 et y vaut 15
}

#include <stdio.h>
main()
{
    int x = 15;
    y = ++x;           //est équivalent à x++; et y = x;
    printf("x=%d, y=%d", x, y); // x vaut 16 et y vaut 16
}
```

### 5.4 Les opérateurs d'affectation étendue

Le langage C propose des combinaisons puissantes d'opérateurs, dans le cas où l'opérande à gauche du signe = se trouve aussi à droite.

Par exemple,  $i = i + k;$  peut s'écrire  $i += k;$

Cette écriture est effectivement plus courte pour le programmeur, mais elle est moins lisible. Toutefois, elle est utile aux programmes nécessitant une optimisation poussée car elle est beaucoup plus rapide.

Tableau des opérateurs d'affectation étendue:

$x* = y$	$\iff$	$x = x*y$
$x/ = y$	$\iff$	$x = x/y$
$x+ = y$	$\iff$	$x = x+y$
$x- = y$	$\iff$	$x = x-y$
$x\% = y$	$\iff$	$x = x\%y$

## 5.5 Les opérateurs relationnels

### 5.5.1 Les opérateurs de comparaison

Ils interviennent essentiellement au niveau des boucles de contrôle tel que le "if", pour comparer des expressions. Contrairement à d'autres langages, le résultat n'est pas une valeur booléenne (vrai ou faux) mais un entier valant :

- 0 si le résultat de la comparaison est faux;
- 1 si le résultat de la comparaison est vrai;

On retrouve:  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $==$  (est égal),  $!=$  (est non égal).

**Remarque :** ne pas confondre l'opérateur d'égalité  $==$  avec l'opérateur d'affectation  $=$ .

### 5.5.2 Les opérateurs logiques

**!** représente le « non » (not) booléen  
**Exemple :**  $!(a < b)$   
 prend la valeur 1 (vrai) si la condition  $a < b$  est fausse et la valeur 0 (faux) dans la cas contraire. Cette expression est équivalente à :  $a \geq b$ .

**&&** représente le « et » (and) booléen  
**Exemple :**  $(a < b) \&\& (c < d)$   
 prend la valeur 1 (vrai) si les deux expressions  $a < b$  et  $c < d$  sont toutes deux vraies, la valeur 0 (faux) dans la cas contraire.

**||** représente le « ou » (or) booléen  
**Exemple :**  $(a < b) || (c < d)$   
 prend la valeur 1 (vrai) si l'une au moins des deux conditions  $a < b$  et  $c < d$  est vraie, la valeur 0 (faux) dans la cas contraire.

**^** représente le « ou exclusif » (xor) booléen (idem bit à bit)  
**Exemple :**  $(a < b) ^ (c < d)$   
 prend la valeur 1 (vrai) si seulement une des deux expressions  $a < b$  et  $c < d$  est vraie, la valeur 0 (faux) dans la cas contraire.

### 5.5.3 Les opérateurs logiques bit à bit

<b>&amp;</b>	représente le « <i>et</i> » (and) bit à bit
<b>Exemple :</b>	10 & 9 donne la valeur décimale 8 car en binaire 1010 & 1001 donne 1000.
<b> </b>	représente le « <i>ou</i> » (or) bit à bit
<b>Exemple :</b>	10   9 donne la valeur décimale 11 car en binaire 1010   1001 donne 1011.
<b>^</b>	représente le « <i>ou exclusif</i> » (xor) bit à bit (idem booléen)
<b>Exemple :</b>	10 ^ 9 donne la valeur décimale 3 car en binaire 1010 ^ 1001 donne 0011.

## VI Les conversions forcées par une affectation

Le langage C est très tolérant en ce qui concerne les mélanges de types dans une expression. C'est au programmeur de vérifier que les conversions implicites réalisées par le compilateur ont le sens désiré. En ce qui concerne les conversions implicites elles sont effectuées par le compilateur.

Une expression de type  $n = x + 5.3$  entraînera tout d'abord l'évaluation de l'expression située à droite, ce qui fournira une valeur de type *float*, cette dernière sera ensuite convertie en *int* pour pouvoir être affectée à  $n$  ( $n$  ayant été déclaré comme entier).

Par exemple, la conversion *float* --> *int* (telle que celle qui est mise en jeu dans l'instruction précédente) ne fournira un résultat acceptable que si la partie entière de la valeur flottante est représentable dans le type *int*. Si une telle condition n'est pas réalisée, non seulement le résultat obtenu pourra être différent d'un environnement à un autre mais, de surcroît on pourra aboutir dans certains cas, à une erreur d'exécution.

D'une manière générale, lors d'une affectation, toutes les conversions sont acceptées par le compilateur mais le résultat en est plus ou moins satisfaisant. Le type de la variable de destination n'intervient pas pendant le calcul de l'expression situé à droite de l'opérateur =. Ce n'est qu'après le calcul de celle-ci que la valeur résultante est éventuellement ajustée pour s'exprimer selon le type de la variable de destination.

Avant chaque opération, le résultat intermédiaire est converti suivant le type de la variable occupant le plus de place en mémoire; cette conversion permet de fournir deux opérandes de même type à l'opérateur qui va être appliqué.

La hiérarchie des types pour les conversions est la suivante :

char → short int → int → long int → float → double → long double

## VII L'opérateur de cast

S'il le souhaite, le programmeur peut forcer la conversion d'une expression quelconque dans un type de son choix, à l'aide d'un opérateur un peu particulier nommé en anglais "**cast**"

Si par exemple,  $n$  et  $p$  sont des variables entières, l'expression :

**(double)** ( $n/p$ ) aura comme valeur celle de l'expression entière  $n/p$  convertie en **double**.

La notation **(double)** correspond en fait à un opérateur unaire dont le rôle est d'effectuer la conversion dans le type double de l'expression sur laquelle il porte. Cet opérateur force la conversion du résultat de l'expression et non celle des différentes valeurs qui concourent à son évaluation. Autrement dit, il y a d'abord calcul dans le type **int**, du quotient de  $n$  par  $p$  (division entière) ; c'est seulement ensuite que le résultat sera converti en **double**.



## Les Entrées/Sorties conversationnelles

Le langage C ne possède pas d'instructions spécifiques d'entrées/sorties. En contrepartie, les dialogues sont réalisés à l'aide d'un grand nombre de fonctions (fournies dans la librairie standard `stdio.h`).

### I L'affichage à l'écran à l'aide de la fonction `printf`

La fonction **`printf`** est une des fonctions les plus utilisées pour réaliser l'affichage sur le fichier standard (`stdout`) qui est en général l'écran selon un format défini par le programmeur.

Sa syntaxe est: **`printf("format",arg1,arg2,...,argn);`**

où format représente une chaîne de caractères contenant:

↳ des caractères à afficher tels quels. C'est donc un texte qui est transmis sans aucune interprétation de la part du C.

**Exemples:** `printf("Bonjour");`  
`printf("Bonjour\n");`

↳ des spécifications de format qui ont une signification très précise.

**Exemple:** `int prix = 50;`  
`printf("Le Prix est: %d", prix);`

Le code format est précédé du symbole %, ce qui le rend aisément reconnaissable. La lettre suivant le symbole % précise de quel type il s'agit.

Après le format, se trouve un ou plusieurs arguments. L'argument donne la valeur à écrire à la place du code format.

**Exemple:** `int n = 12;`  
`printf("Le double de %d est %d", n, n * 2);`

L'exécution de cette instruction donnera : Le double de 12 est 24

Lorsque le compilateur C trouve un code (repérable par la présence du symbole %), il cherche dans la liste d'arguments celui qui correspond en position, prend sa valeur et l'affiche selon le code format spécifié. L'argument peut même être une expression, comme ici dans le cas du deuxième argument.

Chaque lettre, placée après le symbole %, désigne un format différent.

%c	caractère
%d	entier signé
%u	entier non signé
%x	entier en hexadécimal
%ld	entier long signé
%lu	entier long non signé
%lx	entier long en hexadécimal
%f	réel simple précision
%e	réel simple précision, avec exposant e ou E
%lf	réel long double
%le	réel long double, avec exposant e ou E
%s	chaîne de caractères
%p	pointeur (affiché en hexadécimal)

Entre le symbole % et la lettre code, les codes format des options spécifient de façon plus précise la présentation.

☞ Un nombre :

**Exemple:**

```
int n = 12;
float x = 15.4;
printf("%3d", n);
printf("%10f", x);
```

Ce nombre précise le nombre de positions sur lequel doit s'écrire la valeur de la variable, *cadrée à droite*. Ainsi, l'exécution des instructions précédentes affiche:

^12 et ^^^^^^15.4 (le symbole ^ représente un espace).

☞ Le signe - placé immédiatement après le symbole % demande de "cadrer" l'affichage à gauche au lieu de le cadrer (par défaut) à droite.

**Exemple :**

```
int n = 12;
printf("%-3d", n);
```

le cadrage se fait à *gauche* 12^

☞ Deux nombres séparés par un point:

**Exemple :**

```
float x = 15.4;
printf("%10.3f", x);
```

Le premier nombre signifie le nombre total de positions occupées (ici 10) tandis que le deuxième précise le nombre de chiffres à écrire après la virgule (ici 3). Attention, le point occupe une position. On obtiendra donc dans notre exemple: `^^^^15.400`

## II La lecture d'informations à l'aide `scanf`

### 2.1 Son emploi

Le rôle de la fonction **`scanf`** est de lire des informations en provenance du fichier standard d'entrée (stdin), en principal le clavier.

Les informations lues sont automatiquement converties en caractères, entiers, réels selon le format attendu.

Cependant, il y a une différence fondamentale avec la fonction **`printf`**. En effet, dans le cas de **`printf`**, ce qui importait, c'était l'affichage de la *valeur* de la variable, et non l'*emplacement* de la variable en mémoire. Par contre, dans le cas de **`scanf`**, la valeur est inconnue au départ: elle est entrée au clavier. Cette valeur est ensuite rangée en mémoire, dans un *emplacement* localisé par le nom de la variable. C'est donc l'adresse de cet emplacement qu'il faut donner, pour que la fonction **`scanf`** puisse ranger la valeur lue en mémoire.

La syntaxe de la fonction est la suivante: **`scanf("format",arg1,...,argn);`**

`arg1,...,argn` représentent les **adresses** des variables. L'adresse d'une variable se distingue de la variable elle même en faisant précéder le nom de la variable de l'opérateur **`&`**.

Attention, le format est représenté par une suite de codes. Il ne contient pas de chaînes de caractères à afficher comme pour **`printf`**.

Pour bien comprendre le fonctionnement de **`scanf`**, assez déroutant pour beaucoup de ses effets, il faut tenir compte de la présence de deux tampons entre le clavier (c'est à dire le système d'exploitation) et le programme: le tampon système et le tampon du C. Les caractères entrés par l'utilisateur sont d'abord placés dans le tampon système; ils ne sont copiés dans le tampon du C que lorsque l'utilisateur tape RETURN. **`scanf`** va lire les informations qui lui sont nécessaires dans le tampon du C: les caractères sont retirés du tampon au fur et à mesure des conversions réussies vers les types de variables de destination.

Une conversion non réussie provoque un mauvais fonctionnement des lectures suivantes. De même un excès de données en entrée perturbe les lectures suivantes.

**Exemple :**

```
#include <stdio.h>
main()
{
    int i,j;
    printf("Entrez le nombre i :\n");
    scanf("%d", &i);
    printf("Entrez le nombre j :\n");
    scanf("%d", &j);
    printf("i vaut %d et j vaut %d\n", i, j);
}
```

### 2.2 Les précautions pour l'utilisation de `scanf`

Il existe deux règles pouvant diminuer les risques de mauvais fonctionnement.

- ✎ Dans la mesure du possible, ne saisir qu'une variable par fonction **scanf** et contrôler le nombre de variables saisies avec succès.

En tant que fonction, **scanf** retourne une valeur, ce qui permet de tester son bon déroulement. La valeur retournée est le nombre de valeurs lues.

**Exemple:**     **int** i, j, k, n;  
                  n = **scanf**("%d %d %d", &i, &j, &k);

Saisie	Résultats
10 20 30	i=10, j=20, k=30 et n=3
10 a 20	i=10, j et k inchangés et n=1

- ✎ A la fin de la dernière conversion effectuée par une instruction **scanf**, il reste au moins le dernier retour chariot dans le tampon du C. Ce reliquat du tampon sera présenté lors de la prochaine interrogation **scanf**. Si **scanf** cherche à lire une valeur numérique, le retour chariot précédent sera considéré comme un séparateur que **scanf** sautera **naturellement**; par contre, si la première donnée que cherche à lire **scanf** est un caractère, celui-ci sera automatiquement chargé avec '\n'. Il faudra prendre garde à la gestion du tampon lors de la lecture des caractères.

Par conséquent on s'assurera que le tampon du C est vide avant d'utiliser **scanf**. Il existe une fonction **fflush** qui permet de vider le tampon. Elle nécessite l'inclusion du fichier d'en-tête **<stdio.h>** et sa syntaxe est:

```
fflush(stdin); // vide le tampon avant une lecture
```

**ATTENTION !** : Sur linux on appellera la fonction **\_\_fpurge**(stdin); à la place de **fflush**(stdin);

## Les Instructions

### I L'instruction simple - L'instruction composée

Une instruction simple est unique et se termine par un point virgule.

**Exemples :**

```
nombre1 = 8;
printf("%c\n", moncaractere);
nombre2++;
```

Une instruction composée est constituée d'une séquence d'instructions simples. Elle est aussi appelée un *bloc*. L'instruction composée est entourée par des accolades { }. Elle comporte des instructions simples de toutes sortes (entrées, sorties, conditionnelles ...) et même d'autres instructions composées imbriquées.

### II L'instruction d'exécution conditionnelle if

#### 2.1 Son rôle

L'instruction permet d'effectuer une certaine action si et seulement si, une condition est satisfaite.

**Exemple1 :**

```
if (A > B) // instruction conditionnée
    printf("%d est supérieur à %d", A, B);
```

**Exemple2 :**

```
if (A > B) // alternative
    max = A;
else
    max = B;
```

#### 2.2 Sa syntaxe

La syntaxe de cette instruction est:

```
if (expression) instruction_1 [else instruction_2]
```

A remarquer l'absence de point virgule sur les lignes **if** et **else**. Les instructions entre crochets sont facultatives car l'instruction if peut être utilisée seule.

**Exemple :**

```
if (A > B)
{
    printf ("%d est supérieur a %d", A, B);
}
```

## 2.3 Instructions if imbriquées

Après l'expression ou après le **else** se trouve n'importe quelle instruction, simple ou composée, en particulier une autre instruction **if**.

Une certaine ambiguïté survient quand il y a plusieurs instructions **if** imbriquées. Pour lever cette ambiguïté, le langage C associe toujours le **else** au **if** sans **else** le plus proche.

**Exemple :**

```

if ( ... )
    if ( ... )
        instruction;
    else
        instruction;
```

**Exemple :**

```

if ( ... )
{
    if ( ... )
        instruction;
}
else
    instruction;
```

## 2.4 Particularités d'emploi de l'expression en C

↳ **if** ( $N==0$ ) peut s'écrire **if** ( $!N$ )

↳ **if** ( $((I=J) != 0)$ ) veut dire que: - I reçoit la valeur de J  
 - cette valeur est ensuite donnée à l'expression  
 - cette expression est ensuite comparée à 0.

↳ **if** ( $++I < \text{max}$ ) veut que l'on incrémente I puis on le compare avec max.

## III L'instruction de sélection switch

### 3.1 Son rôle

Il est très utile de pouvoir choisir une action particulière parmi plusieurs, en fonction de la valeur d'une variable. On peut par exemple, considérer le choix d'un menu.

Il existe en C, une instruction de choix multiple réalisant un aiguillage direct vers l'action voulue. Il s'agit de l'instruction **switch**.

**Exemple :**

```

#include <stdio.h>
main()
{
    int Choix;
    printf("\n\t Visualisation    <1>"
           "\n\t Modification    <2>"
           "\n\t Ajout          <3>"
           "\n\t Suppression    <4>") ;
    printf(" Entrez votre choix : ") ;
    scanf("%d", &Choix);
```

```

switch (Choix)
{
    case 1: printf("Appel de l'affichage");
            break;
    case 2: printf("Appel de la modification");
            break;
    case 3: printf("Appel de l'ajout");
            break;
    case 4: printf("Appel de la suppression");
            break;
    default: printf("\t Choix non valide");
}
}

```

L'instruction **switch** débute par l'évaluation de la valeur de la variable Choix. Ensuite, elle cherche s'il existe une étiquette **case** dont la valeur soit égale à celle de Choix. Si c'est le cas, le branchement est effectué à cet endroit et les instructions suivantes sont exécutées jusqu'à la rencontre d'une instruction **break** qui provoque la sortie du **switch**.

L'instruction **break** est très importante: si elle est omise, toutes les instructions suivantes sont exécutées pour attribuer plusieurs valeurs de **case** à la même instruction.

Si aucune égalité n'est trouvée, le branchement est réalisé sur **default**.

### 3.2 Sa syntaxe

```

switch (expression)
{
    case constante 1: [suite d'instructions 1]
    case constante 2: [suite d'instructions 2]
    ...
    case constante n: [suite d'instructions n]
    [default      : [suite d'instructions]]
}

```

L'ordre des **case** est quelconque, et **default** peut être placé en n'importe quelle position. Mais les valeurs doivent être toutes différentes.

## IV L'opérateur conditionnel

### 4.1 Son rôle

```

Des tests du style:  if (a > b)
                      max = a;
                      else
                      max = b;

```

sont très souvent effectués dans un programme C. Cette écriture peut se simplifier grâce à la présence d'un opérateur particulier : l'opérateur conditionnel, qui est ternaire, c'est-à-dire qu'il met en rapport trois termes.

Ainsi, l'instruction précédente s'écrit: `max = (a > b) ? a : b;`

L'opérateur comporte deux symboles **?** pour *alors* et **:** pour *sinon* . L'exécution est la suivante:

- ⇒ l'expression `a > b` est testée
- ⇒ si elle différente de 0, la deuxième expression est évaluée (ici `a`) et donne sa valeur à l'expression globale.
- ⇒ sinon la troisième expression est évaluée (ici `b`) et donne sa valeur à l'expression globale.

## 4.2 Sa syntaxe

`expression 1 ? expression 2 : expression 3;`

**Exemple :** utilisation de l'opérateur conditionnel

```
#include <stdio.h>
main()
{
    int i, max = 10;
    printf("Entrez i: ");
    scanf("%d", &i);
    (i > max) ? i = 0 : i++;
    printf ("\nNouveau i : %d", i);
}
```



## Les répétitions

### I L'instruction for

#### 1.1 Son rôle

Elle permet de répéter une action (c'est à dire une suite d'instructions) un nombre de fois défini à l'avance. Elle fait intervenir en général l'initialisation d'un compteur, son incrément et un test de fin de comptage. Cette instruction est très utilisée dans le cas des tableaux.

**Exemple :**

```
#include <stdio.h>
main()
{
    int i; //déclaration externe de i
    for (i = 5; i <= 15; i++)
        printf("%3d",i); // i est affiché de 5 à 15 inclus
    printf("%3d",i);      // i vaut 16
}
```

ou

```
#include <stdio.h>
main()
{
    for (int i = 5; i <= 15; i++) //déclaration interne de i
        printf("%3d",i); // i est affiché de 5 à 15 inclus
    printf("%3d",i);      // i vaut 16
}
```

#### 1.2 Sa syntaxe

**for** ([expression 1]; [expression 2]; [expression 3])

- ↳ Expression 1 (examinée une seule fois) représente la condition initiale. Elle est effectuée avant l'itération.
- ↳ Expression 2 représente le test de fin de boucle. Tant que cette expression est vraie, l'instruction est exécutée. Elle est évaluée à chaque début d'itération. Tant que l'expression est vraie, les instructions sont exécutées.
- ↳ Expression 3 agit sur la valeur de la condition. Il faut remarquer que s'il s'agit d'une simple incrémentation, la valeur de la sortie (incrémentée une fois de trop) prend la première valeur ayant *dépassé* la condition. Elle est évaluée à chaque fin d'itération (après les instructions et avant le test de continuation).

**Remarque:**

↳ Ces trois expressions sont optionnelles; elles peuvent même être absentes toutes les trois !!!

```
for(;;) instruction;
```

Dans ce cas, il faut obligatoirement que l'instruction provoque un débranchement, pour éviter la boucle infinie. Cependant, ce type de programmation n'est pas à recommander.

↳ Plusieurs **for** peuvent être imbriqués.

**Exemple :**

```
#include <stdio.h>
main()
{
    int i, j;
    for(i = 0; i <= 10; i++)
    {
        printf("\\n i = %d", i);
        for(j = 0; j <= 10; j++)
            printf("\\n \\t j = %3d", j);
    }
}
```

## II L'instruction while

### 2.1 Son rôle

L'instruction **while** permet de répéter une suite d'actions tant qu'une condition mentionnée est vraie, sans préjuger du nombre de répétitions.

**Exemple :**

```
#include <stdio.h>
main()
{
    int n;
    double fact;
    char choix = 'O';
    while(choix == 'o' || choix == 'O')
    {
        printf("La factorielle de : ");
        scanf("%d", &n);
        fact = 1;
        while(n)
            fact *= n--;
        printf(" est %10.3Le ", fact);
        printf("Voulez vous continuer <o/n> ? ");
        scanf("%c",&choix);
    }
}
```

**Remarques:**

↳ L'expression est testée avant l'exécution de la boucle: c'est une boucle pré-testée. Si l'expression a un résultat faux dès le départ, le corps de la boucle n'est jamais exécuté. Il y a donc toujours un test de plus que d'exécution de la boucle. Ce test est très précieux dès lors qu'il est important de pouvoir empêcher une action dès le départ.

↳ A l'intérieur de la boucle, la condition doit être modifiée, sous peine de ne jamais en sortir! **ATTENTION** : en embarqué, on peut avoir besoin d'écrire une boucle infinie du type : `while(1)`

↳ Il faut bien sûr ne pas oublier d'initialiser la valeur de l'expression avant d'entrer dans la boucle.

**2.2 Sa syntaxe**

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

**Remarques :** - La condition est obligatoirement entre parenthèses.  
 - L'instruction, se terminant par un point virgule, est simple ou composée.  
 - Les instructions **while** peuvent être imbriquées.

**III L'instruction do while****3.1 Son rôle**

L'instruction **do while** permet de répéter une suite d'actions tant que la condition mentionnée est vraie. Mais cette fois, la condition est testée en fin d'itération.

**Exemple :**

```
#include <stdio.h>
main()
{
    char lettre;
    do
    {
        fflush(stdin);
        scanf("%c", &lettre);
    }
    while((lettre != 'n') && (lettre != 'N'));
}
```

**Remarques:**

↳ Les instructions sont exécutées une première fois, quelle que soit la valeur de la condition, contrairement à l'instruction **while** qui exige de respecter la condition avant toute exécution, même la première.

↳ La variable de test n'a pas besoin d'être initialisée avant l'instruction **do**. Il suffit qu'elle le soit au plus tard à la fin de la boucle avant **while**.

### 3.2 Sa syntaxe

```
do
{
    instructions;
}
while (condition);
```

**Remarques :**

- La condition est obligatoirement entre parenthèses.
- L'instruction, se terminant par un point virgule, est simple ou composée.
- Même si l'expression à répéter se limite à une seule instruction simple, il ne faut pas omettre le point virgule qui la termine.

Ainsi :

```
do c = getchar() while (c != 'x');
```

est incorrect !

Il faut absolument écrire (avec le respect de la norme) :

```
do
{
    c = getchar(); //point-virgule obligatoire
}
while(c != 'x');
```

## Les Branchements Inconditionnels

En principe, la sortie d'une boucle se fait de façon normale, par le biais de l'expression testée. Cependant, dans certains cas, il est nécessaire de quitter la boucle de façon plus autoritaire. C'est ce que permettent les instructions de branchement inconditionnel.

### I L'instruction **break**

La principale utilisation de l'instruction **break** concerne le branchement de type **switch**.

Il est possible de forcer son emploi dans le cas des boucles (**for**, **while**, et **do while**). Si des boucles sont imbriquées, **break** ne permet de sortir que de la boucle qui le contient. L'effet de **break** est donc limité à un seul niveau d'imbrication.

*Exemple:*

```
#include<stdio.h>
main()
{
    int i;
    for(i = 1; i <= 10; i++)
    {
        printf("debut tour %d\n", i);
        printf("bonjour\n");
        if(i == 3)
            break;
        printf("fin tour %d\n", i);
    }
    printf("après la boucle\n");
}
```

### II L'instruction **continue**

L'instruction **continue** donne le contrôle à la fin de la boucle, sans en sortir. Elle permet donc de re-exécuter cette boucle avec la valeur suivante.

En cas de boucles imbriquées, l'instruction **continue** se rapporte au niveau d'imbrication en cours. L'emploi de cette instruction entraîne la destruction rapide du programme. Il y a toujours une solution pour éviter son emploi. En particulier, puisque l'instruction **continue** est généralement placée dans un test, un simple **else** fait souvent l'affaire.

*Exemple :*

```
#include <stdio.h>
main()
{
    int i;
    for(i = 1; i <= 5; i++)
    {
```

```
        printf("debut tour %d\n", i);  
        if(i < 4)  
            continue;  
        printf("bonjour\n");  
    }  
}
```

### III L'instruction nulle ;

Le caractère « ; » délimiteur d'instruction peut être considéré comme une instruction vide s'il est isolé.

Cette instruction permet de terminer une boucle **while** sans corps de boucle.

**Exemple :** cas d'un test répétitif sur la condition **while** qui permet d'ignorer les caractères séparateurs successifs.

```
while((c = getchar()) == ' ' || c == '\t');
```

👉 la fonction **getchar()** permet de lire un caractère (au clavier par défaut)

## Le Pré Processeur et fonctionnalités avancées du langage C

### I La directive **#include**

Cette directive permet d'incorporer, avant compilation, le texte figurant dans un fichier quelconque.

Comme nous l'avons vu précédemment, le langage C est constitué de fonctions. Toutes les fonctions du langage ont leur définition dans des fichiers externes ayant une extension ".h".

Ces fonctions ont été répertoriées en diverses librairies. Les plus utilisées sont les suivantes:

- **stdio.h**: contient toutes les fonctions relatives aux entrées/sorties.
- **math.h**: contient toutes les fonctions mathématiques de base.
- **ctype.h**: contient tous les types de conversions possibles.
- **string.h**: contient toutes les fonctions de manipulation des chaînes de caractères.

Lorsque l'on veut utiliser une des fonctions de ces librairies, il faut impérativement au début du programme **inclure** la librairie à l'aide de la directive **include**.

Ainsi, nous écrirons: **#include** <stdio.h>  
ou **#include** <math.h>

Cette directive a pour effet de rechercher le fichier mentionné dans un emplacement (chemin, répertoire) défini dans l'implémentation. Les symboles inférieurs et supérieurs sont réservés aux librairies prédéfinies du langage C.

Le langage C offre la possibilité à l'utilisateur de définir ses propres fonctions et de les inclure ensuite dans son programme. Ceci est très utile, pour définir ses propres modules qui seront utilisés fréquemment dans les programmes.

Les fonctions définies par l'utilisateur se trouvent en général dans des fichiers sources portant l'extension ".h". Ces fichiers sont inclus à l'aide de la directive **#include**. Les fonctions définies par l'utilisateur sont cette fois-ci entre guillemets.

**Exemple :** **#include** "efface.h"

Cette directive a pour effet de rechercher le fichier mentionné dans le même emplacement (chemin, répertoire) que celui où se trouve le programme source.

**Remarque:** Lorsqu'on utilise la librairie **math.h**, à la compilation il faut rajouter l'option **-lm**.

### II La directive **#define**

La directive **#define** a un double emploi. Elle permet de définir des constantes, ainsi que des macro-instructions.

## 2.1 Les constantes

Le langage C offre la possibilité de définir de nouveaux symboles ou constantes. Celles-ci se définissent de la façon suivante:

**#define <nom de la constante> <valeur de la constante>**

**Exemple:**    **#define** VRAI (1) //protection : usage de parenthèses  
                 **#define** FAUX (0)

Les constantes substituent un texte par un autre. Ainsi, dans l'exemple précédent, on demande au pré processeur de remplacer littéralement le symbole VRAI par le texte 1, chaque fois que ce symbole apparaît dans le fichier source.

Cette directive est précieuse, elle permet en particulier de paramétrer un programme. Ainsi, elle peut servir à définir la taille d'un tableau. Toute modification ultérieure de sa taille se fait automatiquement à l'aide de la seule modification de la directive.

## 2.2 Les macro-instructions

La directive **#define** peut cependant faire beaucoup plus que définir de simples constantes symboliques. Elle permet notamment de définir des **macros**, c'est-à-dire des identificateurs pouvant désigner des expressions, des instructions ou des groupes d'instructions. En un sens, les macros présentent une analogie avec les fonctions. Elles sont cependant définies de façon tout à fait différentes et traitées de façon distinctes par le compilateur.

**Exemple :**    **#include** <stdio.h>  
                 **#define** surface (longueur \* largeur)  
                 **#define** lire fflush(stdin); scanf  
                 **main**()  
                 {  
                     **int** longueur, largeur;  
                     **printf**("longueur = ");  
                     **lire**("%d", &longueur);  
                     **printf**("largeur = ");  
                     **lire**("%d", &largeur);  
                     **printf**("surface = %d", surface);  
                 }

## III Les énumérations

Une énumération est un type de données dont les membres sont des constantes désignées par leurs identificateurs auxquelles sont associées des valeurs entières signées. Ces constantes représentent des valeurs qu'il est possible d'affecter à un ensemble équivalent de variables de type énumération.

La syntaxe générale d'une définition d'énumération est la suivante:

**enum** *etiquette* { *membre1*, *membre2*, ..., *membrek* };

où **enum** est un mot clé obligatoire, **etiquette** un identificateur désignant ladite énumération, et **membre1**, **membre2**, ..., **membrek** les différents identificateurs pouvant se voir affecter des variables du type énumération ainsi élaboré.



Les noms des membres doivent être impérativement distincts, et différents des noms de toutes les variables de même portée.

**Exemple :**

```
enum couleurs {noir, blanc, bleu, rouge, vert};
couleurs texte, fond;

enum couleurs {noir, blanc, bleu, rouge, vert} texte, fond;
enum {noir, blanc, bleu, rouge, vert} texte, fond;
```

Les constantes d'énumération ainsi définies représentent les valeurs suivantes:

```
noir    0
blanc   1
bleu    2
rouge   3
vert    4
```

**Exemple :** `enum {noir = -1, blanc, bleu, rouge, vert, turquoise} texte, fond;`

Les constantes d'énumération ainsi définies représentent les valeurs suivantes:

```
noir      -1
blanc      0
bleu       1
rouge      2
vert       3
turquoise  4
```

#### IV Les Paramètres de la ligne de commande

En C, nous avons vu que tout est fonction et il en est de même pour la fonction particulière **main**. Les parenthèses de **main** sont destinées à recevoir des arguments d'un type particulier, passés à la fonction **main** depuis le système d'exploitation. **ATTENTION**, ceci n'est pas valable dans le monde de l'embarqué sans un système d'exploitation !)

La plupart des compilateurs C permettent l'usage de deux arguments, usuellement appelés `argc`, `argv`, respectivement de types entier et pointeur sur un tableau de caractères (autrement dit, ce tableau est un tableau de chaînes de caractères). La valeur de `argc` indique le nombre d'arguments passés à **main**, et les différentes chaînes contenues dans le tableau pointé par `argv` chacun des arguments passés.

**Exemple :**

```
main(int argc, char *argv[ ])
{
    ...
}
```

L'exécution d'un programme se fait généralement en spécifiant son nom au niveau des commandes du système d'exploitation. Le nom du programme exécutable est alors interprété comme une commande du système d'exploitation. C'est pourquoi la ligne dans laquelle il figure est appelée *ligne de commandes*.

Le passage de paramètres à un programme dont on lance l'exécution se fait en les indiquant à la suite du nom de programme:

*nomprog parametre1, parametre2 ... parametreK*

Les différents éléments figurant sur la ligne de commandes doivent être séparés par un ou plusieurs espaces, ou par des tabulations.

Le nom du programme est alors rangé dans le premier élément de `argv`, suivi des éventuels paramètres. Ainsi, lorsque l'appel du programme est fait avec  $n$  paramètres, le tableau `argv` comprend  $(n + 1)$  éléments identifiés par `argv[0]`, `argv[1]`, ..., `argv[n]`. Par ailleurs, `argc` prend automatiquement la valeur  $(n + 1)$ . Cette valeur n'est donc pas explicitement citée dans l'appel du programme.

**Exemple :**

```
#include <stdio.h>
main(int argc, char *argv[])
{
    int compteur;
    printf("argc = %d\n", argc)
    for(compteur = 0; compteur < argc; compteur++)
        printf("argv[%d] = %s\n", compteur, argv[compteur]);
}
```

Ce programme permet de lancer son exécution avec un nombre quelconque de paramètres sur la ligne de commandes. Lorsqu'il est exécuté, la valeur courante de `argc` et les éléments de `argv` s'affichent à l'écran.

Supposons par exemple que ce programme ait été appelé `exemple`, et que l'on saisisse la ligne de commande suivante:

`exemple bleu blanc rouge`

Le programme génère l'affichage suivant:

```
argc = 4
argv[0] = exemple
argv[1] = bleu
argv[2] = blanc
argv[3] = rouge
```

## V La compilation conditionnelle

Un certain nombre de directives permettent d'incorporer ou d'exclure des portions du fichier source dans le texte qui sera généré par le pré processeur. Ces directives se classent en deux catégories en fonction de la condition qui régit l'incorporation :

- existence ou inexistence des symboles
- valeur d'une expression

## 5.1 Existence liée à l'incorporation des symboles

```
#ifdef symbole
```

```
.....
```

```
#else
```

```
.....
```

```
#endif
```

demande d'incorporer le texte figurant entre les deux lignes **#ifdef** et **#else** si le symbole indiqué est effectivement défini au moment où l'on rencontre **#ifdef**. Dans le cas contraire, c'est le texte figurant entre **#else** et **#endif** qui sera incorporé. La directive **#else** peut, naturellement être absente.

De façon comparable :

```
#ifndef symbole
```

```
.....
```

```
#else
```

```
.....
```

```
#endif
```

demande d'incorporer le texte figurant entre les deux lignes **#ifndef** et **#else** si le symbole indiqué n'est pas défini. Dans le cas contraire, c'est le texte figurant entre **#else** et **#endif** qui sera incorporé.

Pour qu'un tel symbole soit défini pour le préprocesseur, il doit faire l'objet d'une directive **#define**.

**Exemple :** **#define** miseaupoint

```
.....
```

```
#ifdef miseaupoint
```

```
    instructions1
```

```
#else
```

```
    instructions2
```

```
#endif
```

Ici instructions1 seront incorporées par le préprocesseur, tandis que instructions2 ne le seront pas.

## 5.2 Incorporation liée à la valeur d'une expression

```
#if condition
```

```
.....
```

```
#else
```

```
.....
```

```
#endif
```

permet d'incorporer l'une des deux parties du texte, suivant la valeur de la condition indiquée.

**Exemple :**    **#define** code 1  
                  .....  
          **#if** code == 1  
                  instructions 1  
          **#endif**  
          **#if** code == 2  
                  instructions 2  
          **#endif**

**Exemple :**    **#define** code 1  
                  .....  
          **#if** code == 1  
                  instructions 1  
          **#elif** code == 2  
                  instructions 2  
          **#endif**

Ici ce sont les instructions 1 qui seront remplacées par le préprocesseur.

D'une manière générale, la condition mentionnée dans les directives **#if** ou **#elif** peut faire intervenir n'importe quels symboles définis par le préprocesseur et des opérateurs relationnels, arithmétiques ou logiques.

## Les Tableaux

### I Le tableau à une dimension

#### 1.1 Rôle du tableau

Un tableau représente un type structuré. Les types vus jusqu'à présent étaient tous des types simples. Le tableau permet de définir une structure de variable constituée d'un nombre déterminé de composants, tous du même type. Le tableau est indispensable quand on désire manipuler un ensemble de variables de même type dont le nombre exclut la possibilité de les définir et les traiter individuellement.

#### 1.2 Déclaration de tableau

```
type_tableau nom_tableau[nombre_elements];
```

**Exemple :** `int tab[5];`

Cette ligne déclare un tableau appelé `tab`, formé de cinq éléments de type entier. Cette déclaration réserve en mémoire un espace de cinq entiers consécutifs, soit  $5 \times 4 = 20$  octets.

Il est possible de définir des tableaux dont les éléments sont de n'importe quel type, à condition que tous les éléments **aient le même type**.

**Exemple :** `float tableau[10];`  
`char ligne[80];`

#### 1.3 Initialisation de tableau

Comme pour toute variable, le tableau est initialisable lors de sa déclaration.

On peut initialiser un tableau de la façon suivante:

```
int tab[5] = {0, 1, 2, 3, 4};
```

Il n'est pas nécessaire d'initialiser tout le tableau. En commençant obligatoirement au début du tableau, une partie seulement peut être affectée d'une valeur initiale.

**Exemple :** `int tab[5] = {0, 1, 2};`

Seuls les trois premiers éléments sont initialisés avec une valeur définie explicitement.

Dans le cas d'une initialisation complète du tableau, il est possible de ne pas déclarer sa taille. Le compilateur la détermine en calculant le nombre de valeurs énumérées.

**Exemple :** `int tab[] = {0, 1, 2, 3, 4};`

#### 1.4 Accès aux éléments du tableau

Chaque élément est repéré par sa position dans le tableau. Le premier élément d'un tableau a le **rang 0**, le deuxième a le rang 1 et le dernier élément a le rang *nombre d'éléments - 1*.

**Exemples :**    `tab[0]` désigne le premier élément du tableau.  
                   `tab[i]` désigne le (i+1)ème élément du tableau

Grâce à cette indexation, chaque élément est atteint et manipulé exactement comme s'il s'agissait d'une variable simple dont le type est le type du tableau.

**Exemple :**    `tab[2] = 10;`  
                   `tab[4] = tab[2] * 10;`  
                   `tab[j] = tab[i + 4];`  
                   `tab[1] = tab[10 - i * j];`

**Remarque:**    Aucun contrôle de débordement de tableau n'est effectué.  
                   Attention, aux indices qui peuvent devenir négatifs.

## 1.5 Le nom du tableau

Un tableau représente une succession de variables de même type. Ainsi, `tab[i]` représente une variable; `tab` lui, est une constante ou identificateur qui identifie le tableau.

**Il n'y a donc pas de travail possible sur l'ensemble du tableau.** Il ne peut être question de faire lire ou écrire le tableau en une seule instruction. Tout travail sur le tableau se fait à l'aide des instructions répétitives.

**Exemple :**    `for(i = 0; i < 5; i++) //initialisation d'un tableau`  
                   `tab[i] = 0;                //de 5 éléments`

## II Tableau à plusieurs dimensions

### 2.1 Déclaration

L'opérateur `[]` peut être juxtaposé à un autre opérateur `[]`, pour définir les tableaux à plusieurs dimensions.

**Exemple :**    `char ecran[24][80];`  
                   `int matrice[100][100];`  
                   `float courbe[10][10][10];`

`matrice` représente 100 tableaux constitués chacun de 100 entiers.

### 2.2 Initialisation

Le tableau à plusieurs dimensions peut lui aussi être initialisé lors de sa déclaration.

**Exemple :**    `int tab_2d[2][5] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};`

Pour bien comprendre cette écriture, il faut connaître le rangement en mémoire des différents éléments. En C, les éléments du tableau sont rangés dans l'ordre obtenu en faisant varier d'abord le *dernier indice*

**Exemple :**

<code>float tab[3][3][3];</code>	
<code>tab[0][0][0]</code>	position 0
<code>tab[0][0][1]</code>	position 1
<code>tab[0][0][2]</code>	position 2
<code>tab[0][1][0]</code>	position 3
<code>tab[0][1][1]</code>	position 4
<code>tab[0][1][2]</code>	position 5
<code>tab[0][2][0]</code>	position 6
<code>tab[0][2][1]</code>	position 7
<code>tab[0][2][2]</code>	position 8
<code>tab[1][0][0]</code>	position 9
.....	
<code>tab[2][2][1]</code>	position 25
<code>tab[2][2][2]</code>	position 26

**Exemple :** saisie et affichage des éléments d'une matrice 3x3

```
#include <stdio.h>
#define nblig (3)
#define nbcol (3)
main()
{
    int i ,j;
    int mat [nblig][nbcol];
    /*saisie de la matrice*/
    for(i = 0; i < nblig; i++)
    {
        for(j = 0; j < nbcol; j++)
        {
            printf("\n entrez l'élément (%d, %d) :", i+1, j+1);
            scanf("%d", &mat[i][j]);
        }
    }
    /*affichage de la matrice */
    printf("\n \n les éléments de la matrice sont :\n");
    for(i = 0; i < nblig; i++)
    {
        for(j = 0; j < nbcol; j++)
        {
            printf ("\t%d", mat[i][j]);
        }
        printf("\n");
    }
}
```

## Les Types de variables ou classes d'allocations

Dès que la taille d'un programme augmente, il peut être utile de le décomposer en plusieurs morceaux. Le fait de placer ces morceaux dans des fichiers distincts permet une manipulation plus aisée.

Cette méthode de travail nécessite une structuration rigoureuse des fichiers qui composent un programme, ainsi que l'utilisation de types de variables nouveaux (*statiques*, *externes* ...).

On peut distinguer deux types de variables: les variables internes et les variables externes. Il faut comprendre interne comme déclaré à l'intérieur d'une fonction, et externe comme déclaré à l'extérieur de toute fonction. Ce qui différencie les différents types de variables, c'est leur durée de vie et leur visibilité.

Il existe en langage C quatre spécifications pour la classe de mémorisation : *automatique* (**auto**), *externe* (**extern**), *statique* (**static**), *registre* (**register**).

### I Les variables internes

#### 1.1 Les variables automatiques

Les variables que nous avons utilisé jusqu'à présent sont dites automatiques, c'est-à-dire qu'elles "naissent" lors de leur déclaration et "mourraient" à la sortie du bloc où elles étaient nées.

Il s'agit de variables locales, définies à l'intérieur d'un bloc pour lequel elles jouent le rôle de variables locales. Le mot clé **auto**, qui les spécifie, est facultatif. L'espace mémoire qu'elles occupent n'est réservé que lors de sa création et est libéré à la sortie du bloc.

Par défaut, elles ne sont pas initialisées, il faut donc penser à leur donner une valeur avant la première utilisation.

On appelle visibilité d'une variable la zone programme où elle peut être utilisée. La visibilité d'une variable automatique est le bloc où elle est déclarée, ainsi que tous les blocs internes à ce bloc. Des variables automatiques définies dans une autre fonction sont donc totalement indépendantes, bien qu'elles puissent porter le même nom.

**Exemples :**    `int i;            // i est une variable entière automatique`  
                  `auto int j; // idem`

#### 1.2 Les variables statiques

Une variable **statique** interne est, comme une variable automatique déclarée dans un bloc. Sa visibilité est identique à celle d'une variable automatique. En revanche, sa durée de vie est la durée d'exécution du programme.

Bien que n'étant utilisable que dans le bloc où elle est définie, elle ne meurt pas à la sortie de ce bloc. A chaque nouvelle entrée dans le bloc, elle retrouve la valeur qu'elle avait lors de la dernière sortie de ce bloc. Par conséquent, lorsqu'une fonction est appelée après avoir été appelée puis quittée auparavant, les variables statiques qu'elles contiennent ont *conservé leur valeur d'alors*.



Une variable statique est initialisée par défaut à zéro. Son initialisation se fait lors du "chargement" du programme, et non pas lors de la rencontre de la déclaration de la variable. Le mot clé **static** permet de déclarer une telle variable.

**Exemples :**

```
static int i;           //i variable statique entière initialisée
                        //à 0
static float j = 3.5;  //j variable statique réelle initialisée
                        //à 3.5
```

**Exemple :**

```
#include <stdio.h>

void fonction(int premier)
{
    static int i;
    if(premier == 1)
        i = 2;
    printf("i = %d", i);
    i++;
}

main()
{
    fonction(1);
    fonction(2);
}

résultats:      2 et 3
```

### 1.3 Les variables volatiles

Le mot clé volatile est un modificateur de type permettant d'indiquer au compilateur que la variable déclarée est susceptible d'être modifiée à plusieurs endroits dans le code, par exemple :

- variable globale d'un programme comportant du multithreading,
- variable correspondant à un registre matériel,
- variable globale susceptible d'être modifiée par une routine d'interruption (le 1er cas est général et les 2 autres sont plus spécifiques à l'embarqué)

Le compilateur doit donc éviter toute optimisation portant sur cette variable sous peine de ne pas permettre d'obtenir le comportement désiré du programme.

**Exemple :**

```
static int var1;
void bar(void)
{
    var1 = 0;
    while (var1 != 255)
        continue;
}
```

Dans cette fonction, le compilateur va considérer que la condition du `while` est toujours atteinte (ce qui n'est pas forcément le cas dans les exemples cités ci-dessus), et exécuter un comportement correspondant à la fonction suivante :

```
void bar(void)
{
    var1 = 0;
    while (1)
        continue;
}
```

En ajoutant le mot-clef `volatile`, cette optimisation est désactivée :

```
static volatile int var1;
```

## II Les variables externes

Les variables externes sont définies en dehors de toute fonction. Elles sont par défaut visibles à l'extérieur du fichier où elles sont définies (globale); cette visibilité peut être restreinte au fichier de définition en utilisant l'attribut statique.

De ce fait elles ont généralement un emploi dans deux fonctions ou plus, et souvent dans le programme en entier.

Ces variables sont initialisées par défaut à zéro. Il est possible d'initialiser aussi bien les variables de types simples que celles de types composés (tableaux).

### 2.1 Les variables globales

Une variable globale existe pendant toute l'exécution du programme. Elle est visible dans tous les fichiers où elle est déclarée, à partir de sa déclaration.

Il est important de distinguer la déclaration d'une variable globale, de la définition de celle-ci. La déclaration le décrit (donne son type) alors que sa définition provoque de plus la réservation d'espace mémoire correspondant.

Une variable utilisée dans plusieurs fichiers, doit être définie dans un seul de ces fichiers, et déclarée dans tous les autres à l'aide du mot clé **extern**. Pour ces fichiers, cette variable est dite de classe importée.

**Exemples :**

```
// dans un fichier où la variable est définie
float x;

// dans un fichier où elle est utilisée mais définie ailleurs
extern float x;
```

### 2.2 Les statiques externes

Les variables globales permettent d'utiliser des variables dans plusieurs fonctions sans avoir besoin de les passer en paramètres. En revanche, cela ne permet pas l'indépendance entre les modules d'un programme.

Si dans un des fichiers composant un programme, une variable globale s'appelle "a", aucune autre variable globale ne pourra s'appeler "a" dans un autre fichier.

Cela peut être trop contraignant. Une variable externe peut être déclarée avec l'attribut **static**, dans ce cas elle devient "locale" au fichier où elle est déclarée. Sa durée de vie restant celle du programme.

### III Résumé

<b>variables internes:</b>
<b>auto</b>
<b>static</b>
<b>register</b>
<b>volatile</b>
<b>variables externes:</b>
<b>extern</b>
<b>static</b>

## Les Fonctions

### I Notion de fonction en C

Plutôt que de réécrire sans arrêt les instructions constituant une action souvent répétée, il est préférable de les regrouper sous la forme d'un module.

Cette forme modulaire permet d'optimiser le texte source d'un programme ainsi que la taille du code exécutable.

De plus, cette façon de programmer permet une lisibilité et une maintenance beaucoup plus facile d'un programme.

Enfin, cela permet de réunir dans des fichiers extérieurs tous les modules déjà écrits, déjà testés et pouvant servir à d'autres programmes. Ceci est la base de futures bibliothèques.

En C, ces modules ou sous programmes se traduisent sous la forme de fonctions. L'ordre des fonctions n'a pas d'importance. Toutefois, on veillera à placer la fonction **main** à la fin afin de respecter une certaine ergonomie du programme.

### II Définition d'une fonction

La définition d'une fonction est la description de la fonction (en-tête et corps).

```
type_de_retour nom_de_la_fonction(type param1, type param2,...) /* en-tête */
{
    corps de la fonction;
    [ return (valeur_a_renvoyer);]
}
```

**Exemple :**

```
int divisibilite (int a, int b)
{
    if ((b == 0) || ((a % b) == 0))
        return 0;
    else
        return 1;
}
```

Le type de la fonction précisé dans l'en-tête est le type du résultat que la fonction retourne à la fonction appelante.

Dans l'en-tête figure aussi le type et la liste des paramètres formels, c'est à dire les informations qui sont utiles à la fonction.

L'instruction **return** permet de préciser quelle valeur doit être retournée à la fonction appelante.

### III Les variables globales et variables locales

#### 3.1 Les variables locales

En C, toute variable déclarée dans une fonction est **locale** à cette fonction. Cela signifie qu'elle ne peut être utilisée que dans cette fonction et qu'elle n'a de valeur que pour elle.

Les variables locales n'ont aucun lien avec des variables globales de même nom ou avec d'autres variables locales à d'autres fonctions.

**Exemple :**

```
#include <stdio.h>
void Lit()
{
    int numero = 12;
    printf("Valeur d'enregistrement = %d\n", numero);
}

main()
{
    int numero = 15;
    printf("Vous en êtes à %d essais\n", numero);
    Lit();
    printf("Vous en êtes à %d essais\n", numero);
}
```

Dans cet exemple, la variable *numero* est locale à la fonction *Lit()*. On constate, qu'il y a une autre déclaration de la variable *numero* dans la fonction **main**. En fait, il s'agit de deux variables différentes, occupant deux emplacements mémoires différents et ayant donc des valeurs distinctes.

#### 3.2 Les variables globales

L'utilisation de variables locales est très pratique, mais ne permet pas d'utiliser une variable commune à plusieurs fonctions. En fait, comment avoir une variable globale?

On peut se rendre compte que toutes variables déclarées à l'intérieur des accolades sont locales. Par conséquent, pour obtenir la globalité, il suffit de les sortir des accolades.

**Exemple:**

```
#include <stdio.h>
int num_enr; // déclaration de la variable en global

void Lit()
{
    printf("Numéro %d\n", num_enr);
    num_enr++;
}

main()
{
    num_enr = 12;
    printf("Vous en êtes à %d essais\n", num_enr);
    Lit();
    printf("Vous en êtes à %d essais\n", num_enr);
}
```

Dans cet exemple, la variable `num_enr` est connue dans la fonction `main()` et dans la fonction `Lit()`.

Les variables globales sont souvent pratiques dans les programmes. Cependant, elles représentent aussi une source d'erreur. De plus, elles suppriment la possibilité de créer des bibliothèques.

#### IV Fonctions ne fournissant pas un résultat (souvent appelées **procédures**)

Quand une fonction ne renvoie pas un résultat, on le précise dans l'en-tête à l'aide du mot-clé **void**.

**Exemple :** en-tête d'une fonction recevant un argument de type **int** et ne fournissant aucune valeur :

```
void sansval (int n )
```

#### V Fonctions fournissant un résultat

##### 5.1 Son rôle

Dans le cas courant, en mathématique par exemple, une fonction retourne un résultat. De la même façon, en C comme pour tous les langages informatiques, la fonction est destinée à fournir **un** résultat.

Le fait qu'une fonction retourne un résultat est représenté par un mot clé **return**. Entre les parenthèses de cette instruction se trouve la valeur à retourner à la fonction appelante. Cette instruction peut figurer n'importe où dans le corps de la fonction.

```
Exemple :  #include <stdio.h>
            int moyenne (int a, int b)
            {
                return ((a + b) / 2);
            }

            main()
            {
                int a, b, moy;
                printf("entrer a et b \n");
                scanf("%d %d", &a, &b);
                moy = moyenne(a, b);
                printf("la moyenne de a et b est %d", moy);
            }
```

Cette instruction a un double rôle:

- ↳ définir la valeur du résultat,
- ↳ terminer la fonction et redonner le contrôle à la fonction appelante.

**Remarque:** Une fonction peut contenir plusieurs instructions **return**, mais cela n'est guère souhaitable pour une bonne structuration de la fonction si celle-ci est complexe.

**Exemple :**

```
#include <stdio.h>
int Max(int A, int B)
{
    if (A > B)
        return(A);
    else
        return(B);
}

main()
{
    int A,B;
    printf("Entrez deux valeurs: ");
    while(scanf("%d %d", &A, &B) != 2);
    printf("La plus grande valeur est %d", Max(A, B));
}
```

## 5.2 Sa syntaxe

L'instruction **return** s'écrit de trois façons possibles:

↳ **return;**

La valeur retournée est indéfinie. Le compilateur ajoute automatiquement un retour de ce type quand il arrive à la fin d'une fonction et qu'il ne trouve pas cette instruction.

↳ **return** constante;

La valeur retournée est une constante entière (**return** 4 par exemple). Cet usage est intéressant lorsque l'information retournée est pauvre, comme par exemple **return** 0; pour indiquer que l'exécution s'est déroulée sans erreur.

↳ **return** (expression);

L'expression est évaluée et sa valeur est retournée après conversion de type si le résultat de l'expression est d'un type différent de celui de la fonction.

**Exemple :**

```
float fexple(float x, int b, int c)
{
    return(x * x + b * x + c);
}
```

**Remarques :**

- l'instruction **return** définit la valeur du résultat, mais en même temps elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée.

- une fonction peut ne fournir aucune valeur : aucune instruction **return** ne figurera alors dans sa définition. Dans ce cas, le retour est mis en place automatiquement par le compilateur à la "fin" de la fonction.

### 5.3 Le type de la fonction

Si la fonction retourne une valeur, celle-ci peut être de différents types. Cela suppose donc que la fonction ait un type. Pour que le compilateur puisse le connaître, celui-ci est précisé dans l'en-tête de la définition de la fonction.

**Exemples :**    **int** max()  
                  **float** moyenne()

L'instruction permet de retourner une valeur: caractère, entier ou réel dans les différentes longueurs.

Sauf indication contraire, par défaut le type de la fonction est **int**. Aussi, le compilateur suppose qu'une fonction sans type retourne un entier. Ce type par défaut s'applique aussi aux fonctions retournant un **char**. Ce caractère est placé dans la partie basse de la valeur retournée (bits de poids faible).

## VI Le passage de paramètres par valeur

### 6.1 Les paramètres

Pour qu'une fonction soit parfaitement utilisable en toutes circonstances, il est nécessaire qu'elle puisse, non seulement, renvoyer des informations mais qu'elle puisse aussi en recevoir. C'est le rôle des paramètres. La fonction appelante passe des arguments à la fonction appelée.

Les noms des arguments figurant dans l'en-tête de la fonction se nomment des "arguments muets ou formels". Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.

Les arguments fournis lors de l'appel de la fonction se nomment des "arguments effectifs". On peut utiliser n'importe quelle expression comme argument effectif, c'est la valeur de cette expression qui sera transmise à la fonction lors de son appel.

**Exemple :**

```
#include <stdio.h>
int Max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

main()
{
    int x = 3, y = 9;
    int val_max;
    val_max = Max(x, y);
    printf("La plus grande valeur est %d", val_max);
}
```



Dans cet exemple, lors de l'appel, les paramètres effectifs donnent leur valeur aux paramètres formels.

**Remarque:** S'il n'y a pas de concordance de type entre les paramètres deux à deux, le C effectue les conversions nécessaires, comme pour l'affectation.

## 6.2 Le passage par valeur

Le passage des arguments effectifs à la fonction se fait uniquement par valeur.

**Exemple :**

```
void Permute(int x, int y)
{
    int tampon;
    printf("\nx = %d, y = %d avant permutation", x, y);
    tampon = x;
    x = y;
    y = tampon;
    printf("\nx = %d, y = %d apres permutation", x, y);
}

main()
{
    int i = 10, j = 20;
    printf("\ni=%d et j=%d avant appel de la fonction", i, j);
    Permute(i, j);
    printf("\ni=%d et j=%d au retour de la fonction", i, j);
}
```

L'exécution affiche les résultats suivants:

```
i = 10 et j = 20 avant l'appel de la fonction
x = 10 et y = 20 avant la permutation dans la fonction
x = 20 et y = 10 après la permutation dans la fonction
i = 10 et j = 20 au retour de la fonction
```

On constate que finalement les valeurs de *i* et *j* n'ont pas été modifiées. C'est la conséquence du passage par valeurs. En effet, ce sont les contenus des variables qui sont transmis et non leurs adresses.

Chaque contenu est recopié dans un nouvel emplacement mémoire, propre à la fonction. Ce nouvel emplacement prend le nom de paramètre formel. Il est situé dans la pile et disparaît à la fin de la fonction.

Cette copie évite d'altérer les variables de la fonction appelante. Ce type de passage permet une bonne sécurité car la fonction appelée est indépendante et ne peut modifier la variable effective, tout en travaillant avec la valeur de cette variable.

Il est de nombreux cas où l'on désire que cette modification s'effectue. On utilise alors un autre type de passage qui est le passage par adresse. En fait, ce type consiste à passer comme argument non pas la valeur de la variable, mais son adresse. Nous aborderons ce problème lors du chapitre sur les pointeurs.

## VII Prototype d'une fonction

L'utilisation d'une fonction peut avoir lieu alors que la définition de la fonction n'est pas encore connue du compilateur : soit parce que la définition a lieu plus bas dans le même fichier source, soit parce que la définition est effectuée dans un autre fichier.

Pour que le compilateur puisse effectuer son travail, il est nécessaire de lui préciser toutes les fonctions qui sont utilisées avant d'être définies: il suffit de recopier l'en-tête des fonctions, terminé par un point virgule, au début du fichier. C'est ce que l'on appelle le prototype d'une fonction.

Si la fonction est définie dans un autre fichier, on fera précéder le prototype du mot clé **extern**.

Le prototype d'une fonction précise au compilateur le nom de la fonction, son type, la liste et le type des arguments; le compilateur peut ainsi vérifier la concordance des types, forcer les conversions éventuelles et réserver la place mémoire nécessaire.

**Exemple :**

```
#include <stdio.h>
int moyenne (int a, int b); //prototypage de la fonction

main()
{
    int a, b, moy;
    printf("entrer a et b \n");
    scanf("%d %d", &a, &b);
    moy = moyenne(a, b);
    printf("la moyenne de a et b est %d", moy);
}

int moyenne(int a, int b)
{
    return ((a + b) / 2);
}
```

## VIII La Récursivité

### 8.1 Notion de récursivité

Le langage C autorise la réentrance et la récursivité des fonctions.

La réentrance est la propriété qui qualifie le fait qu'une fonction peut être interrompue à n'importe quel moment, ré exécutée elle-même avec d'autres données, puis reprise dans l'état où elle était au moment de l'interruption.

La récursivité est la propriété d'un objet défini à partir de lui même. Elle peut être directe: la fonction s'appelle elle-même ou croisée: une fonction appelle une autre fonction qui appelle à son tour la fonction initiale.

## 8.2 Son fonctionnement

La notion importante qu'il faut saisir, est le fait que lors de chaque appel à la fonction récursive, les variables locales, les arguments et les résultats de la fonction sont stockés dans la pile.

Lorsqu'il n'y a plus d'appel à la fonction récursive, c'est la fin du traitement récursif, ce qui provoque la lecture de la pile avec le calcul du résultat.

## 8.3 intérêt de la récursivité

Les techniques récursives sont séduisantes. Elles permettent l'écriture de programmes courts et de trouver aisément des solutions.

Cependant, elles ne conduisent pas toujours aux solutions les plus efficaces car elles sont chères en temps (gestion de la pile) et en place mémoire. Il est donc en général préférable d'utiliser plutôt des méthodes itératives.

**Exemple :**

```
#include <stdio.h>
long int factorielle(int n);

main()
{
    int n;
    printf("n = ");
    scanf("%d", &n);
    printf("n! = %ld\n", factorielle(n));
}

long int factorielle(int n)
{
    if (n <= 1)
        return(1);
    else
        return(n * factorielle(n - 1));
}
```

## Les chaînes de caractères

Une chaîne de caractères est une suite de caractères alphanumériques terminée par le caractère '\0'. En C il n'existe pas de type chaîne, les caractères sont mémorisés dans un tableau de caractères à une dimension.

### I Définition et initialisation d'une chaîne

Les définitions suivantes : `char texte[8] = "bonjour";`  
ou  
`char texte[] = "bonjour";`

permettent de réserver un tableau de 8 éléments dans lequel sont stockés les 7 caractères du mot `bonjour` suivis par le caractère nul '\0' (c'est à dire dont le code ASCII est 0).

L'élément `texte[0]` contient le caractère 'b' et `texte[7]` contient le caractère '\0' ajouté par le compilateur.

La chaîne "bonjour" qui est dans les deux définitions précédentes est une notation qui permet de simplifier l'initialisation classique d'un tableau:

```
char texte[] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
```

**Remarques :** ⚡ La chaîne de caractères se présente entre guillemets " ", à ne pas confondre avec les apostrophes ' ' qui entourent un caractère.

⚡ Toute chaîne de caractères est terminée par le caractère '\0'. Celui-ci est placé automatiquement par le compilateur.

⚡ Le compilateur peut aussi lui-même calculer la taille de la chaîne ( nombre de caractères +1) si celle-ci n'est pas précisée entre crochets.

### II Ecriture d'une chaîne

En considérant la chaîne comme un tableau de caractères, on peut réaliser l'affichage caractère par caractère jusqu'au moment où le caractère nul est atteint.

**Exemple :**

```
int i=0;
char phrase[] = "il fait beau";
while(phrase[i])
    printf("%c", phrase[i++]);
```

Mais la solution la plus simple consiste à utiliser **printf** avec le format %s réservé aux chaînes de caractères.

**Exemple :**    `char phrase[] = "il fait beau";  
printf("%s", phrase);`

On peut aussi utiliser la fonction **puts** qui n'affiche qu'une chaîne de caractères à la fois et se termine par une fin de ligne.

**Exemple :**    `char phrase[] = "il fait beau";  
puts(phrase);`

### III      Lecture d'une chaîne

Une solution pour lire une chaîne de caractères consiste à utiliser la fonction **scanf** avec le format `%s`, mais la lecture s'effectue jusqu'à la rencontre d'un séparateur (espace, tabulation ou retour chariot). La chaîne est complétée par le caractère `\0`.

**Exemple :**    `char nom[20];  
printf("entrez le nom :");  
scanf("%s", nom);  
printf("le nom est %s", nom);`

Si on désire saisir des chaînes avec des espaces ou des tabulations, il faut utiliser la fonction **fgets**.

**Exemple :**    `char nom [20];  
printf("entrez le nom :");  
fgets(nom, 20, stdin);  
printf("le nom est %s", nom);`

### IV      Quelques fonctions de traitement de chaînes de caractères

Le langage C fournit un grand nombre de fonctions de chaînes (copie, concaténation, recherche d'occurrence, conversion, initialisation..). Ces fonctions sont déclarées dans le fichier en-tête **string.h**.

#### 4.1      Longueur d'une chaîne

La fonction **strlen** fournit la longueur d'une chaîne de caractères. Le caractère nul de fin de chaîne n'est pas compté.

`int strlen (const char *chaîne);`

**Exemple :**    `char nom[20];  
int longueur;  
printf("entrez le nom : ");  
gets(nom);  
longueur = strlen(nom);  
printf("la longueur de la chaîne est %d", longueur);`

## 4.2 Copie d'une chaîne

La fonction **strcpy** permet de copier une chaîne source dans une chaîne destination (y compris le caractère nul de fin de chaîne). La fonction renvoie l'adresse de la chaîne destination.

```
char *strcpy (char *destination, char *source);
```

*Exemple :*

```
char nom[20], name[20];  
printf("entrez le nom :");  
fgets(nom, 20, stdin);  
strcpy(name, nom);  
printf("le nom est : %s ", name);
```

La fonction **strncpy** copie au plus max caractères de la chaîne source dans la chaîne destination.

```
char *strncpy (char *destination, char *source, int max);
```

## 4.3 Concaténation de deux chaînes

La chaîne source peut être ajoutée à la fin de la chaîne destination avec la fonction **strcat**. La fonction renvoie l'adresse de la chaîne destination.

```
char *strcat (char *destination, char *source).
```

La fonction **strncat** ajoute au plus max caractères de la chaîne source à la fin de la chaîne destination.

```
char *strncat (char *destination, char *source, int max ).
```

## 4.4 Comparaison de deux chaînes

La fonction **strcmp** compare une chaîne *chaine1* avec une chaîne *chaine2*. La comparaison s'arrête quand deux caractères sont différents ou quand une chaîne est terminée. La fonction **strcmp** renvoie 0 si les deux chaînes sont identiques.

```
int strcmp (char *chaine1, char *chaine2);
```

La fonction **strncmp** compare deux chaînes en se limitant aux max premiers caractères.

```
int strncmp (char *chaine1, char *chaine2, int max);
```

*Exemple :*

```
#include <stdio.h>  
#include <string.h>  
main()  
{  
    char mot1[10] = "bonjour";  
    char mot2[10] = " à tous";  
    char phrase[20];
```

```
    strcpy(phrase, mot1);
    strcat(phrase, mot2);
    printf("%s", strcat(phrase, " !"));
    printf("la phrase a %d caractères", strlen(phrase)) ;
}
```

## V Tableaux de chaînes de caractères

Pour traiter un ensemble de chaînes de caractères, on peut définir un tableau de chaînes de caractères, c'est à dire un tableau à deux dimensions de caractères.

*Exemple :*

```
char jours[7][9] = {"lundi", "mardi", "mercredi", "jeudi",
                    "vendredi", "samedi", "dimanche"};
```

## VI Lecture depuis une chaîne de caractères

**Exemple :**

```
#include<stdio.h>
main()
{
    int a;
    char b;
    char c;
    char ch[] = "3YZ";
    sscanf(ch, "%1d%c%c", &a, &b, &c);
    printf("a=%d, b=%c, c=%c", a, b, c);
}
```

Le programme affiche : a=3, b=Y, c=Z

### 3 : Ecriture formatée vers une chaîne de caractères

```
#include<stdio.h>

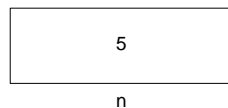
main()
{
    char ch[30];
    int t = 25;
    sprintf(ch, "Il fait %d degres", t);
    puts(ch);
}
```

Le programme affiche : Il fait 25 degres

## Les Pointeurs

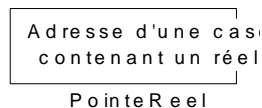
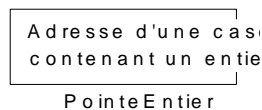
### I Notion de pointeur

L'écriture `int n = 5;` représente une variable `n` de type entier initialisée à 5. Plus précisément, `n` indique au compilateur une *adresse* en mémoire dont le *contenu* est un entier de valeur 5.



Le contenu d'une variable peut être un caractère, un entier, ou un réel mais il peut aussi contenir une adresse:

```
int *PointeEntier;
float *PointeReel;
```



`PointeEntier` est une variable dont le *contenu* est l'*adresse* d'une variable entière et `PointeReel` contient l'adresse d'une variable de type réel. `PointeEntier` est un pointeur sur un entier et `PointeReel` un pointeur sur un réel. Il est possible d'avoir un pointeur sur n'importe quel type de variables.

Le symbole `*` est un opérateur unaire, à ne pas confondre avec le symbole `*` de la multiplication!

Ces déclarations réservent en mémoire la place nécessaire pour y écrire deux adresses (une dans `PointeEntier` et l'autre dans `PointeReel`) mais elles **ne déclarent pas les variables correspondantes**. De plus, `PointeEntier` et `PointeReel`, comme toute variable, ne sont pas initialisées a priori. On n'a donc pas le droit de les utiliser avant qu'elles n'aient été initialisées, soit avec l'adresse d'une autre variable, soit à l'aide d'une allocation dynamique (voir plus loin).



## II Manipulation de pointeur

### 2.1 Exemple

```
#include <stdio.h>
main()
{
    /* Declaration des variables */
    int i, j, *PointeEntier;

    /* Emplacement des variables en mémoire */
    printf("\n\n L'adresse de i est %ld\n L'adresse de j est %ld\n L'adresse
        de PointeEntier est %ld", &i, &j, &PointeEntier);

    /* Travail avec les variables */
    i = 10;
    printf("\n L'emplacement i à l'adresse %ld contient %d", &i, i);

    PointeEntier = &i;    /* PointeEntier reçoit l'adresse de i */
    printf("\n\n L'emplacement PointeEntier situé à l'adresse %ld \n contient %ld
        (qui est l'adresse de i) %d ", &PointeEntier, PointeEntier, *PointeEntier);

    j = *PointeEntier;
    printf("\n\n L'emplacement j à l'adresse %ld contient %d qui est aussi la valeur
        de i", &j, j);

    *PointeEntier = j * 2;
    printf("\n\n L'emplacement %ld (adresse de i) contient maintenant %d",
        PointeEntier, *PointeEntier);
}
```

#### Résultats:

L'adresse de i est 12  
 L'adresse de j est 16  
 L'adresse de PointeEntier est 20  
 L'emplacement i à l'adresse 12 contient 10  
 L'emplacement PointeEntier situé à l'adresse 20 contient 12 (qui est l'adresse de i), 10  
 L'emplacement j à l'adresse 16 contient 10 (qui est aussi la valeur de i)  
 L'emplacement 12 (adresse de i) contient maintenant 20

#### Signification:

↪ `PointeEntier = &i` signifie que l'on met l'adresse de i dans le pointeur `PointeEntier` (et non le contenu de i). Lorsque l'on écrit cette instruction, on dit que "PointeEntier pointe vers i".

↪ `j = *PointeEntier` demande de mettre dans j le *contenu* de la *variable pointée* par `PointeEntier`. Cela signifie que le programme va à l'adresse de `PointeEntier`, lit le

contenu de cette variable, se déplace à cette nouvelle adresse, prend son contenu et l'affecte enfin à la variable `j`.

↳ `*PointeEntier = j*2` met le contenu de `j` multiplié par 2 dans la variable pointée par `PointeEntier`. Donc `i` vaudra 20.

## 2.2 Quelques pièges d'écriture

**Exemple :**

```
main()
{
    int *Pointe1, i, j, *Pointe2;
    Pointe1 = &i;
    Pointe2 = Pointe1;
}
```

↳ `Pointe2 = Pointe1` traduit une affectation entre pointeurs, à l'issue de laquelle `Pointe1` et `Pointe2` contiennent la même adresse.

↳ `*Pointe2 = *Pointe1` signifie que le contenu de l'emplacement mémoire pointé par `Pointe1` est transféré dans celui pointé par `Pointe2`. Les deux pointeurs ne contiennent pas les mêmes adresses.

↳ `*Pointe1 = 0` signifie que le contenu de l'emplacement mémoire pointé par `Pointe1` est mis à 0.

↳ `Pointe1 = 0` signifie par convention que `Pointe1` ne pointe sur rien. En fait, on utilise plutôt dans ce cas la constante **NULL**. Cette notation sert dans le cas des variables chaînées pour caractériser l'extrémité de la chaîne.

## III Arithmétique des pointeurs

### 3.1 Incrémentation et décrémentation de pointeurs

Les opérations sur les pointeurs sont à employer avec beaucoup de précautions. Une erreur peut entraîner des écritures n'importe où en mémoire.

**Exemple :**

```
main()
{
    int i;
    float tab[20], *ptr;

    ptr = tab;          /* On se positionne à l'adresse de la 1ère case de tab */
    for (i=0; i<20; i++)
    {
        *ptr = 0;        /* On initialise toutes les cases de tab à 0 */
        ptr++;           /* Incrémentation d'une case, i.e. de 4 octets */
    }

    /* Attention: en sortie de boucle, ptr pointe en dehors de tab. On n'a donc pas le
    droit de l'utiliser directement. */
    ptr--;               /* ptr pointe maintenant sur la dernière case de tab */
}
```

```
printf("%f", *ptr); /* Affichage du contenu de la dernière case */
}
```

### 3.2 Addition

Elle n'est possible qu'avec des entiers. Mais, tout comme l'incrémentation, l'unité de compte n'est pas l'octet mais la taille de l'élément.

**Exemples :**    **long** \*PointeL;  
                  **float** \*PointeR;

```
PointeL += 2; /* PointeL est augmenté de 2 entiers longs, c a d de 8 octets */
PointeR += 2; /* PointeR est augmenté de 2 réels, c a d de 8 octets */
```

**Remarque :** L'addition de deux pointeurs n'a pas de sens.

### 3.3 Soustraction

Elle est possible avec des entiers et fonctionne comme l'addition en prenant la taille de l'élément pointé comme unité de compte.

### 3.4 Comparaison

Il s'agit en fait de réaliser la soustraction de deux pointeurs à comparer et de conclure en fonction du signe ou de la valeur nulle du résultat. Les règles appliquées à cette opération sont donc les mêmes pour la comparaison et la soustraction.

### 3.5 Multiplication et division

Ces opérations n'ont pas de sens et sont rejetées par le compilateur.

## IV Application aux passages de paramètres

Le passage de paramètres est, sans conteste, une des sources principales d'utilisation des pointeurs. En effet, le passage de paramètres par valeur empêche de modifier les variables de départ. Il est maintenant possible d'apporter une solution grâce aux pointeurs.

**Exemple :**    **#include** <stdio.h>  
                  **void** Permute(**int** \*PtrX, **int** \*PtrY)  
                  {  
                  **int** tampon;  
                  **printf**("\\n\\n x = %d et y = %d avant la permutation dans la  
                          fonction", \*PtrX, \*PtrY);  
                  tampon = \*PtrX;  
                  \*PtrX = \*PtrY;  
                  \*PtrY = tampon;  
                  **printf**("\\n\\n x = %d et y = %d apres la permutation dans la  
                          fonction", \*PtrX, \*PtrY);  
                  }  
                  **main**()  
                  {

```

int i = 10, j = 20;
printf("\\n\\n i=%d et j=%d avant l'appel de la fonction", i, j);
Permute(&i, &j);
printf("\\n\\n i=%d et j=%d au retour de la fonction", i, j);
}

```

L'appel de la fonction `Permute` a été modifié. Les deux paramètres effectifs passés sont les adresses des deux variables `i` et `j`.

Les paramètres formels de la fonction `Permute` sont devenus des pointeurs, qui reçoivent les adresses de `i` et `j`. Par le biais de ces pointeurs, il est possible maintenant pour la fonction `Permute` de travailler directement sur le contenu des adresses pointées à leur emplacement dans le programme appelant.

## V Passage de fonctions comme arguments d'autres fonctions

Lorsqu'une déclaration de fonction apparaît à l'intérieur d'une autre fonction, le nom de la fonction déclarée devient un pointeur vers cette fonction.

On peut utiliser de tels pointeurs comme arguments d'appel d'une autre fonction. Ceci revient à passer à la fonction appelée un paramètre qui est une fonction, de la même façon que l'on passe une variable. La fonction ainsi transmise est accessible à l'intérieur de la fonction appelée. Des appels successifs peuvent passer des pointeurs différents (en fait des fonctions différentes).

Lorsqu'une fonction accepte en argument le nom d'une autre fonction, la déclaration des arguments formels doit être modifiée en conséquence. Dans sa forme la plus simple, la déclaration d'un argument de type fonction s'écrit:

`Type_donnée ( *nom_fonction ) ( )`  
 dans laquelle `Type_donnée` définit le type de donnée que renvoie la fonction. Cette fonction peut alors être référencée grâce à l'opérateur d'indirection. Dans ce but, l'opérateur d'indirection doit précéder le nom de la fonction et doit être inclus, avec le nom de la fonction, entre parenthèses:

`type_donnée( *nom_fonction ) ( type arg1, type arg2, ..., type arg n )`

**Exemple:** `int traiter(int (*pf)(int x, int x))`

```

{
int a, b, c;
...
c = (*pf)( a, b);
...
return c;
}

```

```

int fonc1(int a, int b )
{
int c;
...
c = ...;
return c;
}

```

```

    }

    int fonc2(int x, int y )
    {
        int z;
        ...
        z = ...;
        return z;
    }

    main()
    {
        int i , j;
        i = traiter( fonc1 );
        i = traiter( fonc2 );
    }

```

## VI La gestion dynamique de la mémoire

### 6.1 Principe de l'allocation dynamique

Jusqu'à présent, une variable était déclarée dans le programme et rendue accessible ensuite grâce à son identificateur. Elle se voyait allouer une place en mémoire dès le début de l'exécution du programme pour les variables statiques ou dès l'appel de la fonction pour les variables locales.

Mais une variable peut aussi être dynamique. Elle est dite dynamique quand elle n'apparaît pas dans une déclaration de variable explicite et ne peut être référencée directement. Sa place n'est allouée que par une demande explicite du programme.

Elle est alors repérée par un pointeur dont la valeur n'est générée qu'au moment de l'exécution. Ce pointeur contient l'adresse de l'emplacement en mémoire où cette variable est stockée.

### 6.2 Demander de la place en mémoire

#### 6.2.1 malloc

**Exemple:**

```

#include <stdio.h>
main()
{
    char *adr;
    long *ptr;
    adr = malloc(10);
    ptr = malloc(8);
}

```

Le rôle de la fonction d'allocation **malloc** est de retourner un pointeur sur une zone mémoire de la taille demandée. Dans cet exemple, la fonction **malloc** réserve successivement deux emplacements: un de 10 octets et l'autre de 8.

Le seul argument de **malloc** est la taille demandée, comptée en octets. Cet argument est de type unsigned **int**;

Parfois, le calcul à la main, de la taille d'une variable (ex: tableaux) est assez hasardeux. Il est préférable de demander au compilateur de faire ce travail à l'aide de l'opérateur **sizeof**.

```
sizeof(short int)    -> 2
sizeof(double)       -> 8
sizeof(char)         -> 1
```

La valeur de retour est :

- un pointeur sur le premier octet en cas de succès
- NULL s'il n'y a pas assez de place mémoire

Il serait mieux d'écrire l'exemple précédent sous la forme :

```
#include <stdio.h>
main()
{
    char *adr;
    long *ptr;
    adr = (char *) malloc( sizeof (char)*10);
    ptr = (long *) malloc( sizeof (long) );
}
```

On utilise l'opérateur **sizeof** pour obtenir la taille de l'objet et on convertit le type du pointeur retourné en type désiré.

### 6.2.2 realloc

Augmentation de la taille d'une zone précédemment allouée.

```
void *realloc(descripteur,size)
    void *descripteur;
    int size; //correspond à la nouvelle taille à allouer en octet
```

**realloc** alloue un nouveau tampon et y transfère le contenu du précédent tampon. La valeur de retour est :

- nouvelle adresse du tampon en cas de succès
- NULL en cas d'échec

### 6.2.3 calloc

Allocation d'une zone mémoire initialisée avec des caractères nuls.

```
void *calloc(n_elem,elem_size)
unsigned n_elem,elem_size;
```

la fonction alloue un bloc de taille « `n_elem` » fois « `elem_size` » octets, et initialise ce bloc à 0. On peut dire que c'est un tableau dont le nombre d'éléments est « `n_elem` » et la taille d'un élément « `elem_size` », et qui est initialisé à zéro.

La valeur de retour est :

- un pointeur sur la zone allouée
- NULL en cas d'échec

### 6.3 Restituer de la place mémoire allouée

Toute zone mémoire précédemment réservée par **malloc** ou **calloc** qui n'est plus utilisée doit être libérée à l'aide de la fonction **free**

**void free(void \*ptr)** (ptr doit être l'adresse du début de cette zone)

### 6.4 Allocation dynamique des tableaux

#### 6.3.1 Tableau à un seul indice

Dans le cas simple d'un tableau mono indicé il y a équivalence formelle entre l'identificateur du tableau et le pointeur.

En langage C, l'identificateur d'un tableau, lorsqu'il est employé seul (sans indices à sa suite) est considéré comme un pointeur (constant) sur le début du tableau. Supposons par exemple que l'on effectue la déclaration suivante :

**int t[10]** la notation `t` est alors totalement équivalente à `&t[0]`

L'identificateur `t` est considéré comme étant de type pointeur sur le type correspondant aux éléments du tableau, c'est à dire ici, **int\***. Ainsi voici quelques exemples de notations équivalentes.

<code>t + 1</code>	$\Leftrightarrow$	<code>&amp;t[1]</code>
<code>t + i</code>	$\Leftrightarrow$	<code>&amp;t[i]</code>
<code>t[i]</code>	$\Leftrightarrow$	<code>*(t + i)</code>

Le pointeur obtenu par allocation dynamique peut donc s'utiliser directement après une simple conversion de type comme un identificateur de tableau. Par exemple pour allouer dynamiquement un tableau `t` de `n` entiers on fera :

```
int *t;  
.....  
t = (int *) malloc (n * sizeof (int));
```

et le pointeur `t` pourra s'utiliser de façon indexée comme un tableau ordinaire soit par exemple

**Exemples :**

```
main()  
{  
  int i, tab[10];  
  int *ptrtab;
```

```

/* initialisation avec une syntaxe de type tableau */
for (i=0; i< 10; i++)
    tab[i]=0;
/* initialisation avec une syntaxe de type pointeur */
for (i=0; i< 10; i++)
    *(tab + i)=0;

/* initialisation à l'aide d'un pointeur */
for (ptrtab= tab; ptrtab < tab+10; ptrtab++)
    *ptrtab=0;
}

```

Pour l'initialisation avec une syntaxe de type pointeur, `tab` représente une constante de type pointeur, on peut l'utiliser comme tout pointeur pour accéder aux éléments du tableau. `*(tab + i) = 0` permet d'écrire la valeur 0 dans l'élément pointé par `(tab + i)`. L'adresse contenue dans `tab` est alors cherchée, à laquelle est ajoutée `i * sizeof (tab[0])` et la valeur 0 est alors écrite à cette adresse.

Pour l'initialisation à l'aide d'un pointeur, `ptrtab= tab` permet d'écrire dans le pointeur `ptrtab` l'adresse du début du tableau `tab`. Au début de la boucle, `ptrtab` pointe sur l'élément 0 du tableau. La valeur de `ptrtab` n'est pas gardée constante, elle est incrémentée à chaque exécution de la boucle pour permettre à `ptrtab` de pointer l'élément de tableau suivant. `ptrtab` désigne donc successivement tous les éléments du tableau.

La différence entre tableau et pointeur est qu'un tableau représente une adresse constante (allouée par le compilateur) qui ne peut pas être modifiée. Des instructions comme `tab++` ou `tab=ptr` sont donc interdites si `tab` est un nom de tableau, alors que l'on peut incrémenter ou affecter une valeur à un pointeur.

### 6.3.2 Tableaux à double indice

Pour un tableau à double indice, le nombre d'éléments d'une ligne est nécessaire à l'indexation. Deux cas peuvent alors se présenter suivant qu'il s'agit d'une constante ou d'une variable.

↳ S'il s'agit d'une constante :

```
int (*t) [100];      /* t est un pointeur sur un tableau de 100 entiers*/
```

On peut donc allouer un tableau de `n` lignes de 100 éléments:

```
t= (int (*)[100] ) malloc (n *sizeof ( int [100]));
```

On notera les parenthèses indispensables et le type « pointeur sur un tableau de 100 entiers » utilisé dans l'opérateur « cast ». Après cette allocation le pointeur `t` peut maintenant s'employer comme un tableau à double indice.

↳ Si la taille de la ligne est variable, on peut linéariser le tableau ou mieux créer en plus du tableau de pointeurs un autre tableau de pointeurs sur les lignes. Prenons l'exemple



de l'allocation dynamique d'une matrice d'entiers de dimension 5\*2. On alloue les 5\*2 éléments de la matrice

**Exemple :** `#include <stdio.h>`

```

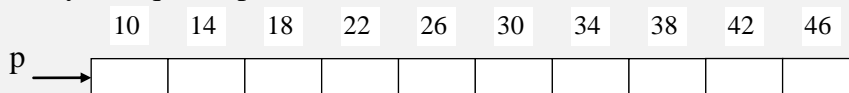
main()
{
  int *p,i,j,**t,*x;
  p= (int *) malloc (5*2* sizeof(int));           /*on alloue les 5*2 éléments de la
                                                    matrice*/
  t= (int * *) malloc (5 * sizeof(int *));        /*on alloue un tableau de 5
                                                    pointeurs sur les lignes*/
  x=p;                                              /*pointe sur le début de la matrice */
  for (i=0;i<5;i++)
  {
    t[i]=x;
    x+=2;                                          /*pointe sur la ligne suivante*/
  }

  for (i=0;i<5;i++)
    for (j=0;j<2;j++)
      scanf ("%d",&t[i][j]);

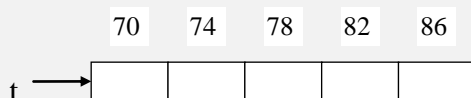
  for (i=0;i<5;i++)
    for (j=0;j<2;j++)
      printf ("%d",t[i][j]);
}

```

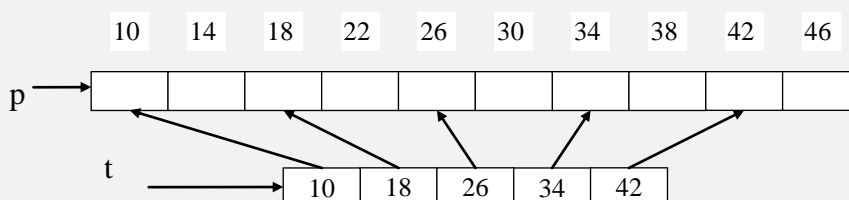
allocation dynamique de p:



allocation dynamique de t :



résultat:



autre méthode de déclaration dynamique d'un tableau de 5\*2 éléments:

**Exemple :**

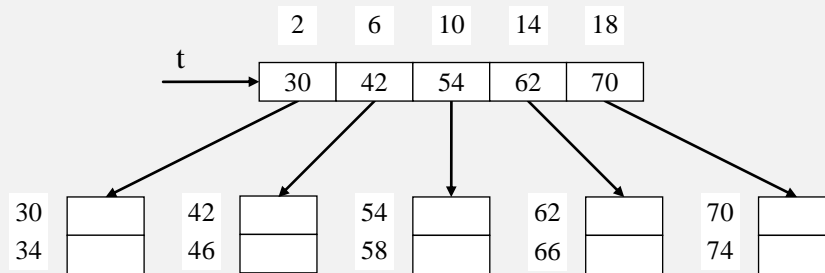
```

#include <stdio.h>
main()
{
    int i,j,**t;
    t= (int **) malloc (5 * sizeof(int *));
    for (i=0;i<5;i++)
    {
        t[i]= (int *) malloc (2*sizeof(int));
    }

    for (i=0;i<5;i++)
        for (j=0;j<2;j++)
            scanf ("%d",&t[i][j]);
    for (i=0;i<5;i++)
        for (j=0;j<2;j++)
            printf ("%d",t[i][j]);
}

```

**Représentation:** allocation dynamique de t (cf ci-dessus)  
 résultat:



## Les Structures

Nous avons vu comment le tableau permettait de désigner sous un seul nom un ensemble de valeurs de même type, chacune d'elles étant repérée par un indice. La structure, quant à elle, va nous permettre de désigner sous un seul nom un ensemble de valeurs pouvant être de types différents.

### I Déclaration et initialisation

#### 1.1 Notion de structure

Une personne peut être caractérisée par son nom, son prénom, son âge ... Il est donc très intéressant de manipuler à l'aide d'une seule entité ces informations de types différents, mais qualifiant le même être.

**PERSONNE:**

Nom	Prénom	Age	...
-----	--------	-----	-----

Une structure définit un nouveau type qui, de même qu'un entier ou un réel, est une entité pouvant être traitée comme un élément autonome.

#### 1.2 Modèle de structure

Les structures ne sont pas des types préexistants en C. Il est nécessaire de les définir avant toute utilisation. Cette définition consiste principalement à nommer les différents éléments, avec leur taille, qui recevront ensuite les informations.

```
struct Personne
{
    char Nom[20];
    char Prenom[20];
    int Age;
};
```

Cette déclaration n'est pas une déclaration de variable. Elle se contente de décrire la structure en indiquant les champs la composant et en lui donnant un nom. Ainsi, dans l'exemple précédent, le mot `Personne` joue le même rôle que le mot clé `int` ou `float`. Ce modèle qui s'appelle `Personne` précise le nom et le type de chacun des "champs" constituant la structure (`Nom`, `Prenom`, `Age`).

#### 1.3 Déclaration de variable de type structuré

Une fois la définition de la structure, il faut déclarer des variables de ce nouveau type. Il existe deux façons habituelles de définir des variables de type structuré.

```

struct Personne
{
    char Nom[20];
    char Prenom[20];
    int Age;
} p1, p2;

```

Cette première façon déclare le modèle de structure Personne en même temps que les deux types variables p1 et p2. Dans ce cas, l'identificateur de la structure peut même être omis s'il n'est plus nécessaire ultérieurement.

```

struct
{
    char Nom[20];
    char Prenom[20];
    int Age;
} p1, p2;

```

La deuxième façon sépare la définition du modèle et la déclaration de la variable.

```

struct Personne
{
    char Nom[20];
    char Prenom[20];
    int Age;
};

struct Personne p1, p2;

```

Le modèle de structure est défini dans un premier temps. Puis des variables de ce type sont déclarées autant de fois qu'il est nécessaire. Cette deuxième forme est plus pratique et plus claire pour la maintenance des programmes.

Les variables de type structuré prennent différentes formes. Ainsi, à l'aide du modèle de Personne, il est possible de déclarer les variables suivantes:

```

struct Personne p, *ptr, tab[100];

```

Cette ligne déclare:

- ↪ Une variable p de type Personne qui occupe 44 octets en mémoire.
- ↪ Un pointeur ptr vers cette structure, qui lui occupe 4 octets et ne réserve en aucun cas de la place d'une variable de type structuré.
- ↪ Un tableau de 100 éléments de type structuré.

## 1.4 Structures imbriquées

Les champs dans une structure sont de tout type, y compris des structures elles-mêmes. Ainsi, nous pouvons avoir:

```

struct date
{
    int jour;
    int mois;
    int an;
};

struct Personne
{
    char Nom[20];
    char Prenom[20];
    int Age;
    struct date Naissance;
    struct date Embauche;
};

struct Personne Employe, Salarie[50];

```

### 1.5 Initialisation de variables structurées

L'initialisation des variables de type structuré se fait de la même façon que pour les tableaux.

```

struct Personne Employe = {"Dupont", "Jean", 20, 13, 8, 69, 1, 7, 88};

```

L'initialisation peut être incomplète mais elle doit être contiguë à partir du début.

## II Utilisation d'une structure

### 2.1 Accès global à la structure

L'affectation globale entre deux variables définies à partir du même modèle (de nom identique) est possible.

```

struct Personne
{
    char Nom[20];
    char Prenom[20];
    int Age;
};

struct Personne p1, p2;
p1 = p2;

```

**Remarque:** Tout autre opérateur est interdit sur la variable structurée prise dans son ensemble. Il est impossible de comparer des variables structurées. Ainsi **if** ( $p1 == p2$ ) provoque une faute de compilation.

De même, la lecture ou l'écriture globale de la variable structurée sur l'écran est rejetée. Il faut le faire champ par champ.

## 2.2 Accès aux champs d'une structure

Les champs d'une structure peuvent être manipulés comme n'importe quelle variable du type correspondant. La désignation d'un champ se note en faisant suivre le nom de la variable structure de l'opérateur "." (point) qui a la priorité la plus élevée (exemple: p1.Age).

**Exemples:**     `printf("%s",p1.Nom);`  
                   `scanf("%d",&p1.Age);`  
                   `p1.Prenom[0] = 'D';`  
                   `p1.Age++;`

Dans le cas de structures imbriquées, le principe reste le même:  
       `Employe.Embauche.an = 1989;`

Toutefois, les affectations globales du champ structuré `Embauche` sont valides:  
       `Employe.Embauche = Salarie[10].Embauche;`

## III La structure en tant que paramètre

### 3.1 Portée du nom de modèle de la structure

A l'image de ce qui se produit pour les identificateurs de variables, la "portée" d'un modèle de structure dépend de l'emplacement de sa déclaration :

Si elle se situe au sein d'une fonction (y compris la fonction **main**), elle n'est accessible que depuis cette fonction.

Si elle se situe en dehors d'une fonction, elle est accessible de toute la partie du fichier source qui suit sa déclaration; elle peut ainsi être utilisée par plusieurs fonctions.

```
fonction()
{
/* Le modèle de la structure Personne est inconnu */
/* Une variable de type Personne ne peut être déclarée */
... }

main()
{
struct Personne    {...};
/* Utilisation de la structure Personne possible */
}
```

Un moyen de palier à ce problème est de rendre la structure globale :

```
struct Personne { ... };

fonction()
{ /* Utilisation de la structure Personne possible */
}
```

```

main()
{ /* Utilisation de la structure Personne possible */
}

```

### 3.2 Passage d'informations structurées entre fonctions

Des fonctions peuvent avoir besoin de se transmettre des variables structurées. Il est dans ce cas conseillé de passer l'adresse de la variable structurée. On évite ainsi de dupliquer une zone mémoire importante, et de plus la fonction appelée peut modifier la variable structurée.

**Exemple:**

```

struct Personne {
    char Nom[20];
    char Prenom[20];
    int Age;
};

void vutab(char tab[20])
{
    int i;
    for (i=0;(i<20) && (tab[i] != '\0');i++)
        printf("\n\t %c",tab[i]);
}

void affiche (struct personne *ptr )
{
    printf("\n\n\t\t%s",ptr->Nom);      /* simplifie l'écriture et augmente la visibilité*/
    printf("\n\n\t\t%s",(*ptr).Prenom); /* l'écriture équivalente est -> Attention:
                                         l'oubli des parenthèses provoque une erreur car
                                         c'est le CONTENU du champ qui est pris
                                         comme adresse */

    printf("\n\n\t\t%d", ptr->Age);
}

main()
{
    struct Personne p1={"SCHMIDT","Henri",35},p2;
    affiche (&p1);
    vutab(p1.Nom);
}

```

Dans cet exemple, la fonction affiche récupère un pointeur sur une structure de type personne.

L'écriture (\*ptr).Prenom permet d'atteindre le champ Prenom de la variable p1, pointée par ptr. Les parenthèses sont obligatoires car l'opérateur '.' a la priorité la plus haute. Les omettre reviendrait à demander au langage de prendre le contenu du champ Prenom comme adresse! Cet oubli déclenche normalement un message d'erreur à la compilation sauf si le champ est lui même un pointeur.

Cette écriture est lourde. C'est pourquoi, on dispose d'un autre opérateur qui simplifie l'écriture :ptr -> Nom. L'opérateur -> relie le pointeur d'une structure au champ désiré. Le langage C voit qu'il s'agit d'un pointeur. Il lit l'adresse contenue dans le pointeur et s'y

déplace. Là, il ajoute le décalage indiqué par le champ `Nom` et se trouve alors positionné sur la bonne variable.

#### IV Types de Données personnalisés ( `typedef` )

La directive **`typedef`** permet à un utilisateur de créer des types de données personnels en renommant des types existants. Une fois défini ce nouveau type, il peut être utilisé pour les objets à créer ensuite (variables, tableaux, structures, etc).

La syntaxe générale de création d'un type personnalisé est la suivante:

***`typedef type nouveau_type`***

où *type* fait référence à un ***type*** de donnée existant (lui même type standard ou type personnalisé antérieurement défini), et ***nouveau\_type*** désigne le type personnalisé créé.

**Exemple:**

```
typedef struct
{
    int NumCompte;
    char Nom[20];
} enregistrement;

enregistrement anc_client, nouv_client;
```

*Pour aller plus loin ...*

#### V Structure autoréférentielles (Listes chaînées)

Une *structure autoréférentielle* correspond à une structure dont au moins un des champs contient un pointeur vers une structure de même type. De cette façon on crée des éléments (appelés parfois *noeuds* ou *liens*) contenant des données, mais, contrairement à un tableau, celles-ci peuvent être éparpillées en mémoire et reliées entre-elles par des liens logiques (des pointeurs), c'est-à-dire un ou plusieurs champs dans chaque structure contenant l'adresse d'une ou plusieurs structures de même type.

↳ Lorsque la structure contient des données et un pointeur vers la structure suivante on parle de liste chaînée.

↳ Lorsque la structure contient des données, un pointeur vers la structure suivante, et un pointeur vers la structure précédente on parle de liste chaînée double.

↳ Lorsque la structure contient des données, un pointeur vers une première structure suivante, et un pointeur vers une seconde, on parle d'arbre binaire.

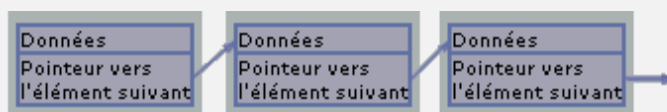


## 5.1 Qu'est-ce qu'une liste chaînée ?

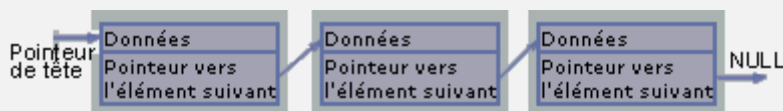
Une liste chaînée est une structure comportant des champs contenant des données et un pointeur vers une structure de même type. Ainsi, la structure correspondant à une liste chaînée contenant une chaîne de 15 caractères et un entier sera définie comme ceci:

```
struct Nom_de_la_liste {
    char Chaîne[16];
    int Entier;
    struct Nom_de_la_liste *pSuivant;
};
```

Une liste chaînée se représente de la façon suivante:



Il est nécessaire de conserver une "trace" du premier enregistrement afin de pouvoir accéder aux autres, c'est pourquoi un pointeur vers le premier élément de la liste est **indispensable**. Ce pointeur est appelé *pointeur de tête*. D'autre part, étant donné que le dernier enregistrement ne pointe vers rien, il est nécessaire de donner à son pointeur la valeur NULL.



Pour créer une liste chaînée en langage C, il s'agit dans un premier temps de définir la structure de données.

```
struct Liste {
    char Chaîne[16];
    int Entier ;
    struct Liste *pSuivant;
};
```

## 5.2 Ajout du premier élément

Une fois la structure définie, il est possible d'ajouter un premier maillon à la liste chaînée, puis de l'affecter au pointeur *Tete*. Pour cela il est nécessaire:

↳ de définir les pointeurs

```
struct Liste *Nouveau;
struct Liste *Tete;
```

↳ d'allouer la mémoire nécessaire au nouveau maillon grâce à la fonction *malloc*, selon la syntaxe suivante:

```
Nouveau = (struct Liste*)malloc(sizeof(struct Liste));
```

↳ d'assigner au champ "pointeur" du nouveau maillon, la valeur du pointeur NULL

```
Nouveau->pSuivant = NULL;
```

↳ de définir le nouveau maillon comme maillon de tête

```
Tete = Nouveau;
```

### 5.3 Ajout d'un élément en fin de liste

Pour ajouter un élément à la fin de la liste chaînée il faut définir un pointeur (appelé généralement *pointeur courant*) afin de parcourir la liste jusqu'à atteindre le dernier maillon (celui dont le pointeur possède la valeur NULL). Les étapes à suivre sont:

↳ la définition d'un pointeur *courant* et du pointeur du nouvel élément:

```
struct Liste *pCourant, *Nouveau;
```

↳ le parcours de la liste chaînée jusqu'au dernier noeud:

```
while (pCourant->pSuivant != NULL)
    pCourant = pCourant->pSuivant;
```

↳ l'allocation de mémoire pour le nouvel élément:

```
Nouveau = (struct Liste*)malloc(sizeof(struct Liste));
```

↳ faire pointer le champ « pointeur » de courant vers le nouveau noeud, et le champ « pointeur » du nouveau noeud vers NULL:

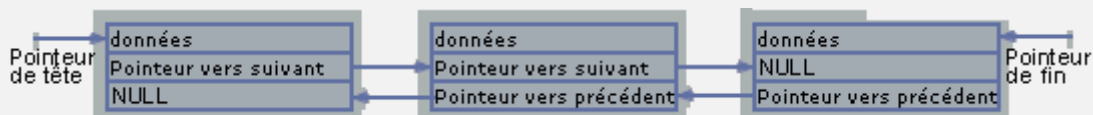
```
Courant->pSuivant = Nouveau;
Nouveau->pSuivant = NULL;
```

## 5.4 Liste chaînée double

Une liste chaînée double est basée sur le même principe que la liste chaînée simple, à la différence qu'elle contient non seulement un pointeur vers le maillon suivant mais aussi un pointeur vers le maillon précédent, permettant de cette façon le parcours de la liste dans les deux sens. Ainsi, la structure utilisée précédemment devient :

```
struct Liste {  
char Chaine[16];  
int Entier;  
struct Liste * pSuivant;  
struct Liste * pPrecedent  
};
```

Une liste chaînée double se représente donc de la façon suivante:



*Pour aller plus loin ...***Les Unions****I Déclaration d'une union****1.1 Rôle d'une union**

Une union est une structure de données qui permet de faire partager une même place mémoire à des variables de type différents. Ceci est particulièrement utile pour utiliser de plusieurs façons différentes une même information. Les unions permettent donc d'économiser la mémoire utilisée, et présentent un intérêt pour les applications mettant en oeuvre des membres dont les valeurs n'ont pas à être affectées au même moment.

Dans une union, la gestion de la mémoire nécessaire à chaque membre est automatiquement prise en charge par le compilateur. Le programmeur doit cependant disposer d'une trace à tout moment du type de l'information manipulée. Toute tentative d'accès à une donnée erronée produit des résultats invalides.

**1.2 Déclaration**

La déclaration d'une union est similaire dans sa forme à celle d'une structure et présente les mêmes caractéristiques.

```
union ecran {  
    unsigned mot;  
    unsigned char octet[8];  
};
```

Cette déclaration est suivie de la réservation en mémoire d'une variable.

```
union ecran cellule;
```

Dans le cas de la structure, la place occupée par la variable est la somme de tous les champs tandis que dans le cas de l'union, la place occupée est la taille du champ le plus grand. Les champs peuvent avoir, en effet, des tailles différentes. Une autre façon de présenter l'union est de dire qu'elle est une structure dont tous les champs sont alignés à la même adresse.

**II Utilisation de l'union****2.1 Accès global**

Tout comme une structure, l'accès global est valide. L'affectation directe de deux variables structurées à l'aide de la même union est très pratique. Il est bien sûr toujours possible de travailler également avec un pointeur chargé par l'adresse de l'union à l'aide de l'opérateur &.

## 2.2 Accès aux champs de l'union

Les champs de l'union sont atteints, comme pour la structure, à l'aide de l'opérateur "."

**Exemple :**    ecran.mot;  
                ecran.octet[0];

L'opérateur -> est aussi utilisé et simplifie l'écriture dans le cas de pointeur sur des unions (ptr->mot au lieu de (\*ptr).mot).

## III L'union en tant que paramètre

### 3.1 Portée de l'union

La déclaration du modèle de l'union n'est connue que dans la fonction où elle est écrite. Si ce modèle est utilisé dans plusieurs fonctions, il est préférable de l'écrire en variable globale au début du source, en dehors de toute fonction, comme pour les structures.

### 3.2 Passage de paramètre

Le passage comme paramètre d'une variable de type union se fait classiquement à l'aide d'un pointeur. La fonction appelante passe l'adresse de la variable de type union. La fonction appelée reçoit un pointeur sur une variable de type union.

## IV Exemple

```
#include <stdio.h>
main( )
{
    union ref {
        char couleur;
        int taille;
    };
    struct {
        char fabricant[20];
        float prix;
        union ref description;
    } chemise, blouson;

    printf("%d\n", sizeof(union ref));
    chemise.description.couleur = 'b';
    printf("%c %d\n", chemise.description.couleur, chemise.description.taille );
    chemise.description.taille = 42;
    printf("%c %d\n", chemise.description.couleur, chemise.description.taille );
}
```

L'exécution du programme donne le résultat suivant :

```
4
b -24713
@ 42
```

*Pour aller plus loin ...***Les Fichiers****I Notion de fichier**

Les fichiers permettent de stocker sur un support permanent (disque ou disquette) un ensemble de données produites par le programme. Nous verrons toutefois qu'en langage C, tous les périphériques, qu'ils soient d'archivage (disque, disquette) ou de communication (clavier, écran) peuvent être considérés comme des fichiers. Ainsi les entrées-sorties conversationnelles, c'est à dire les fonctions permettant d'échanger des informations entre le programme et l'utilisateur apparaîtront comme un cas particulier de la gestion de fichiers.

Les fonctions dites "de niveau 1" ou bas niveau sont très proches du système d'exploitation. Elles considèrent le fichier comme une suite ininterrompue d'octets, sans considération d'organisation logique.

Les fonctions dites "de niveau 2" considèrent le fichier comme une suite d'enregistrements organisés logiquement (mais il est toujours possible de lire n'importe quel nombre d'octets). L'accès aux informations se fait à travers un tampon géré par le langage C. De ce fait, les informations peuvent être formatées, comme une entrée/sortie conversationnelle.

**Remarque :** Ce cours ne considère que les fonctions de haut niveau.

**II Manipulation globale de fichier****2.1 Déclaration du fichier**

On déclare un fichier de la façon suivant: `FILE *Dico;`

Cette déclaration déclare une variable dont l'identificateur est Dico.

La variable Dico est un pointeur sur un objet de FILE. Ce nom écrit en majuscule, désigne un modèle de structure défini dans **stdio.h**.

**2.2 Ouverture du fichier**

Déclarer un fichier n'est pas suffisant pour pouvoir l'utiliser. Il faut en plus:

- ☞ Indiquer de quel fichier physique il s'agit. Il faut donner le nom du fichier, tel qu'il apparaît sur le support.
- ☞ Il faut préciser le type d'opération à réaliser sur le fichier.

Cela se fait à l'aide de la fonction **fopen** :

```
Dico = fopen("individu.dat", "r");
```

Syntaxe : `FILE *fopen(char *nom_fichier, char *mode_ouverture);`

Les différents modes d'ouverture possibles sont :

"r" ou "rb" :	Lecture seule. Lors de l'ouverture, le curseur est positionné sur le premier octet du fichier. Le fichier doit déjà exister.
"w" ou "wb" :	Ecriture seule. Le curseur est positionné sur le premier octet du fichier. Celui-ci est créé s'il n'existe pas, son ancien contenu est écrasé s'il existe.
"a" ou "ab" :	Ce mode autorise l'ajout. Le curseur est positionné en fin de fichier. Celui-ci est créé s'il n'existe pas.
"r+" ou "rb+" :	Lecture et écriture sur un fichier existant. Utile pour effectuer des modifications. Le fichier doit déjà exister.
"w+" ou "wb+" :	Ecriture et lecture. Le fichier est créé s'il n'existe pas, son ancien contenu est écrasé s'il existe.
"a+" ou "ab+" :	Ecriture et lecture. Le curseur est positionné à la fin du fichier. Celui-ci est créé s'il n'existe pas.

**Remarque :** le caractère b signifie que le fichier est un fichier en mode binaire (à l'opposé du mode texte).

Si la fonction a pu être exécutée avec succès, elle fournit un pointeur qui est stocké, dans l'exemple, dans la variable Dico. En cas de problème, ce pointeur a une valeur nulle.

**Exemple :**

```
FILE *Dico ;
Dico = fopen("individu.dat", "a"); /*ouverture du fichier en mode ajout*/
if (Dico==NULL)
    puts("problème à l'ouverture du fichier");
```

## 2.3 Fermeture du fichier

Le rôle principal de la fermeture du fichier est de vider le tampon associé au fichier. En effet l'écriture dans le fichier ne se fait pas variable après variable. Les différentes écritures transitent dans un canal appelé buffer. Une fois que ce buffer est plein, il y a transfert dans le fichier.

La fermeture se fait à l'aide de la fonction **fclose** :

```
fclose(Dico);
```

## III Entrées/Sorties dans les fichiers

### 3.1 Ecriture dans un fichier

On utilise la fonction **fwrite**. Cette fonction déplace elle-même le curseur après chaque écriture. Elle nécessite quatre paramètres:

- ↗ L'adresse d'un bloc d'informations.
- ↗ La taille d'un bloc en octets à écrire.
- ↗ Le nombre de blocs de cette taille que l'on souhaite écrire dans le fichier.
- ↗ L'adresse de la structure décrivant le fichier

La fonction **fwrite** fournit en retour le nombre de blocs effectivement écrits.

**Exemple:** Retour = **fwrite**(&Nombre, sizeof(Nombre), 1, Dico);

**if** (Retour != 1) ...

### *Exemple d'écriture d'un élément d'une structure dans un fichier*

```
#include <stdio.h>
#define max 20

/*structure dédiée au fichier*/
struct personne
{
    char nom[max] ;
    int age ;
};

typedef struct personne pers ;

/*fonction saisie
entrée : nomFic nom du fichier a créer ou mettre à jour
fonction qui permet la saisie d'une personne et le stockage
dans le fichier nomFic */

void saisie (char *nomFic)
{
    pers p ;
    FILE *fic ;

    fic = fopen (nomFic , "ab"); /*ouverture du fichier en mode ajout (binaire)*/
    if (fic==NULL)
        puts ("problème a l'ouverture du fichier");
    else
    {
        /*saisie des données*/
        puts (« entrer le nom de la personne ») ;
        fflush (stdin) ;
        gets (p.nom) ;
        puts (« entrer l'age de la personne ») ;
        scanf( « %d »,&p.age) ;

        fwrite (&p, sizeof(pers),1,fic) ; /*écriture du contenu de la structure dans le fichier*/
        fclose(fic); /*fermeture du fichier*/
    }
}

/*fonction principale*/
main()
{
    char nomFichier[max] ;
    puts( « entrer le nom du fichier ») ;
    gets (nomFichier) ;
    saisie(nomFichier) ;
}
```

## 3.2 Lecture dans un fichier



La lecture se fait toujours à partir de la position du curseur. Celui-ci est déplacé du nombre d'octets occupés par la variable lue. Après chaque lecture, le curseur est donc placé sur le premier octet non encore lu, jusqu'à la fin du fichier. Il est préférable de tester la fin du fichier à l'aide **feof**.

La lecture se fait avec la fonction **fread** dont les arguments sont identiques à ceux de **fwrite**. La seule différence est que le premier paramètre indique l'adresse de la zone où les informations lues seront stockées.

La fonction **fread** retourne le nombre de blocs effectivement lus, ce qui permet de tester sa bonne exécution. La prise en compte des erreurs peut s'écrire en utilisant une boucle **while**:

```
while (fread(&Nombre,sizeof(Nombre),1, Dico) && !feof(Dico)) ....
```

#### IV Les fonctions de positionnement dans les fichiers

**int fseek(FILE \*stream, long offset, int origin)**

**fseek** positionne le pointeur de fichier pour le flot stream; une lecture ou une écriture ultérieure accédera aux données commençant à la nouvelle position. Pour un fichier binaire, la position est fixée à offset caractères de origin, qui peut valoir **SEEK\_SET** (début de fichier), **SEEK\_CUR** (position courante) ou **SEEK\_END** (fin de fichier). Ces différentes valeurs sont définies dans le fichier **stdio.h**

**int ftell (FILE \*stream)**

**ftell** retourne la position courante du pointeur de fichier par rapport au début du fichier.

**void rewind (FILE \*stream)**

**rewind** repositionne le pointeur de fichier stream sur le début du fichier.

**fseek** et **rewind** retournent 0 après une correction sans erreur du pointeur de fichier. Elles retournent une valeur différente de 0 s'il y a erreur.

#### V Les entrées-sorties formatées

Nous venons de voir que les fonctions **fread** et **fwrite** réalisent un transfert d'informations (entre mémoire et fichier) qui se fait sans aucune transformation de l'information.

Mais en langage C, il est possible d'accompagner ces transferts d'information d'opérations de formatage analogues à celles que réalisent **printf** et **scanf**. Les fichiers concernés par ces opérations de formatage sont alors ce que l'on a coutume d'appeler des "fichiers de type texte" que l'on peut manipuler avec un éditeur quelconque ou lister par les commandes appropriées du système d'exploitation (ex : **cat** ou **more** pour **LINUX**).

Dans de tels fichiers, chaque octet représente un caractère. Généralement on y trouve des caractères de fin de ligne (**/n**), de sorte qu'ils apparaissent comme une suite de lignes.

Les fonctions permettant de travailler avec des fichiers de texte ne sont rien d'autre qu'une généralisation aux fichiers déjà rencontrés pour les entrées/sorties conversationnelles.

Ce sont :

```
fscanf ( FILE *fp, char *format, liste-d'adresses )
fprintf( FILE *fp, char *format, liste-d'expressions )
fgetc ( FILE *fp ) /*lecture d'un caractère*/
fputc ( FILE *fp ) /*écriture d'un caractère*/
fgets ( char chaine, int taille_max, FILE *fp ) /*lecture d'une chaîne*/
fputs ( char chaine, FILE *fp ) /*écriture d'une chaîne*/
```

La signification de leurs arguments est la même que pour les fonctions conversationnelles correspondantes. Seule **fgets** comporte un argument entier (taille\_max) de contrôle de longueur. Il précise le nombre maximal de caractères (y compris \0 de fin) qui seront placés dans la chaîne.

### *Exemple d'écriture dans un fichier texte avec fprintf*

```
#include <stdio.h>
main()
{
  FILE *fic ;
  int i;
  int tab[5]={ 1, 2, 3, 4, 5};
  fic = fopen ("resultats.dat", "wt"); /*mode écriture texte */
  if (fic==NULL)
  {
    printf ("problème ouverture du fichier\n");
  }
  else
  {
    fprintf (fic, « nombre d'éléments %d », 5) ;
    for (i=0, i<5 ;i++)
      fprintf (fic, "\n element %d: %d", i, tab[i]);
    fclose (fic);
  }
}
```

## Spécificités liées à l'embarqué

### I Codage binaire des nombres entiers

#### 1.1 Codage des entiers positifs ("non signés", en langage C)

Pour coder des nombres entiers positifs, on utilise le codage binaire naturel. Par exemple sur 8 bits :

binaire	décimal
0 0 0 0 0 0 0 0	0
0 0 0 0 0 0 0 1	1
0 0 0 0 0 0 1 0	2
...	
0 1 1 1 1 1 1 1	127
1 0 0 0 0 0 0 0	128
...	
1 1 1 1 1 1 0 1	253
1 1 1 1 1 1 1 0	254
1 1 1 1 1 1 1 1	255

Du point de vue général, sur  $n$  bits,  $2^n$  valeurs positives différentes peuvent être codées (de 0 à  $2^n - 1$ ).

#### 1.2 Codage des entiers positifs ou négatifs ("signés", en langage C)

Dans un système de calcul numérique (comme l'ordinateur), il est nécessaire de pouvoir travailler avec des nombres négatifs.

Pour leur codage, le principe est de répartir la plage des valeurs possibles en 2 parts égales : la première pour les valeurs positives et la 2<sup>e</sup> pour les valeurs négatives. En observant le tableau ci-dessus, on constate qu'à la moitié du tableau, le bit le plus à gauche change de signe ; d'où l'idée d'utiliser ce bit comme **bit de signe**.

binaire	décimal	
0 0 0 0 0 0 0 0	0	valeurs positives
0 0 0 0 0 0 0 1	1	
0 0 0 0 0 0 1 0	2	
...		
0 1 1 1 1 1 1 1	127	valeurs négatives
1 0 0 0 0 0 0 0		
...		
1 1 1 1 1 1 0 1	?	
1 1 1 1 1 1 1 0		
1 1 1 1 1 1 1 1		

Se pose alors le problème du codage de ces valeurs.

La manière la plus naturelle serait de considérer que les bits autres que le bit de signe correspondent à la valeur absolue du nombre codé. Mais ce codage comporte 2 problèmes :

- 0 possède 2 codages possibles (00000000 et 10000000)
- l'addition binaire, utilisée pour les nombres positifs, ne fonctionne pas ici. Exemple :

$$(0010)_2 + (1001)_2 = (1011)_2$$

soit

$$(2)_{10} + (-1)_{10} = (-3)_{10} \quad (\text{faux})$$

Une autre idée consiste à inverser les bits codant la valeur ("**complément à 1**"). Mais les 2 problèmes soulevés ci-dessus sont toujours présents :

- 0 possède 2 codages possibles (00000000 et 11111111)
- l'addition binaire ne fonctionne toujours pas. Exemple :

$$(0010)_2 + (1110)_2 = (0000)_2$$

soit

$$(2)_{10} + (-1)_{10} = (0)_{10} \quad (\text{faux})$$

Autre exemple :

$$(1101)_2 + (1110)_2 = (1011)_2$$

soit

$$(-2)_{10} + (-1)_{10} = (-4)_{10} \quad (\text{faux})$$

Le résultat est faux, mais on peut remarquer qu'à chaque fois, il est inférieur de 1 au résultat recherché. D'où l'idée d'ajouter 1 aux opérandes négatives ; on obtient alors le bon résultat. Reprenons les exemples précédents :

$$(0010)_2 + (1111)_2 = (0001)_2$$

soit

$$(2)_{10} + (-1)_{10} = (1)_{10} \quad (\text{correct})$$

Et pour l'autre exemple :

$$(1110)_2 + (1111)_2 = (1101)_2$$

soit

$$(-2)_{10} + (-1)_{10} = (-3)_{10} \quad (\text{correct})$$

Ce codage est appelé "**complément à 2**". Son intérêt est de ne comporter qu'un seul codage du 0, et de permettre d'utiliser les opérations arithmétiques binaires (+, -, \*, /) aussi bien avec les nombres positifs qu'avec les nombres négatifs. Dans le cas de 8 bits, on aura donc :

binaire	décimal
0 0 0 0 0 0 0 0	0
0 0 0 0 0 0 0 1	1
0 0 0 0 0 0 1 0	2
...	
0 1 1 1 1 1 1 1	127
1 0 0 0 0 0 0 0	-128
...	
1 1 1 1 1 1 0 1	-3
1 1 1 1 1 1 1 0	-2
1 1 1 1 1 1 1 1	-1

En langage C, le cas décrit ci-dessus correspond au type "char". Le principe reste le même pour 16 bits (entiers courts, type "short int") et 32 bits (entiers long, type "long int").

Ce codage est utilisé sur la plupart des systèmes numériques, et notamment les microprocesseurs (donc les ordinateurs) et les microcontrôleurs.

## II Opérations bit-à-bit

Dans le domaine de l'embarqué et plus précisément dans le développement de programmes pour microcontrôleurs, une grande partie de la programmation consiste à écrire des valeurs dans des registres et à lire la valeur d'autres registres. Un registre est un dont chaque bit a un rôle bien déterminé dans le fonctionnement du microcontrôleur (par exemple, un registre d'entrée-sortie dont chaque bit permet, en mode entrée, d'acquérir un signal binaire du monde extérieur, ou bien de commander, en mode sortie, un élément extérieur).

Dans le cas de la programmation en langage C, un registre est constituée par une variable dont on va lire ou modifier les bits indépendamment les uns des autres.

Dans ce qui suit sont décrites les différentes opérations logiques "bit-à-bit" permettant de modifier et lire des variables. On rappelle les symboles des opérateurs :

&	ET
	OU
^	OU-exclusif
~	inversion
<< (>>)	décalage à gauche (resp. droite)

### 2.1 Positionnement d'un bit d'une variable

#### 2.1.1 Positionnement à 1

Considérons la table de vérité du OU logique :

A	B	A OU B
0	0	0
0	1	1
1	0	1
1	1	1

On remarque que, quel que soit l'état d'une variable binaire 1 bit (en raisonnant sur A : voir valeur entourées dans la table ci-dessus), un OU logique entre cette variable et un 1 (valeurs correspondantes de B) impose un 1 en sortie. Le OU logique permet donc de forcer un bit à 1. Dans le cas de la programmation en langage C, les variables comportent au moins 8 bits (type char) ; on utilise alors ce qu'on appelle un masque, c'est à dire une autre variable ou une constante, composée d'un 1 à l'indice du bit que l'on souhaite positionner, et de 0 partout ailleurs.

```
int a=0b00010101;
int b=0b00011010;
int c;
c = a | b;
printf("c=%d\n", c);          // résultat : c=31  (00010101 | 00011010 = 00011111)
printf("c=%x\n", c);          // affichage en hexadécimal : c=1f
```

Dans cet exemple, on vérifie bien que tous les bits à 1 dans le masque ont imposé un 1 dans le résultat, et que les autres bits n'ont pas été modifiés.

### 2.1.2 Positionnement à 0

Considérons la table de vérité du ET logique :

A	B	A ET B
0	0	0
0	1	0
1	0	0
1	1	1

On remarque que, quel que soit l'état d'une variable binaire 1 bit (en raisonnant sur A : voir valeur entourées dans la table ci-dessus), un ET logique entre cette variable et un 0 (valeurs de B correspondantes) impose un 0 en sortie. Le ET logique permet donc de forcer un bit à 0. Exemple :

```
int a=0b00111101;      // la variable à tester
int b=0b00100010;      // le masque
int c;                  // le résultat
...
c = a & b;
printf("c=%d\n", c);    // résultat : c=32 (00111101 & 00100010= 00100000)
printf("c=%x\n", c);    // affichage en hexadécimal : c=20
```

Dans cet exemple, on vérifie bien que tous les bits à 0 dans le masque ont imposé un 0 dans le résultat, et que les autres bits n'ont pas été modifiés.

## 2.2 Test d'un bit d'une variable

### 2.2.1 Test à 1

Considérons à nouveau la table de vérité du ET logique :

A	B	A ET B
0	0	0
0	1	0
1	0	0
1	1	1

On peut remarquer que le résultat d'un ET logique entre un bit variable (supposons qu'il s'agisse de A dans la table ci-dessus) et un 1 permet de savoir si la valeur du premier est 1. D'où le principe de test, en supposant que A est une variable à 1 bit (sous forme algorithmique, et en utilisant le symbole de l'opérateur ET bit-à-bit du langage C) :

si `A & 1 == 1`

*action associée au fait que A soit est égal à 1*

Exemple d'une variable de type `int` :

```

int n=0b00110011; //tester différentes valeurs
if((n & 0b0000010) == 0b0000010)
    printf("le bit 1 est égal à 1\n"); //l'indice des bits commence à 0
else
    printf("le bit 1 est égal à 0\n");

```

Ce même test peut bien sûr être effectué avec la valeur hexadécimale correspondante :

```

...
if((n & 0x02) == 0x02)
    printf("le bit 1 est égal à 1\n");
else
    printf("le bit 1 est égal à 0\n");

```

ou la valeur décimale :

```

...
if((n & 2) == 2)
    printf("le bit 1 est égal à 1\n");
else
    printf("le bit 1 est égal à 0\n");

```

De la même manière, on peut tester plusieurs variables simultanément. Exemple :

```

...
if((n & 0b0001010) == 0b0001010)
    printf("les bit 1 et 3 sont à 1\n");
else
    printf("au moins un des bits 1 et 3 est à 0\n");

```

*Remarque* : dans le cas du test d'un seul bit, on peut remarquer que le premier exemple ci-dessus peut s'écrire également :

```

...
if((n & 0b0000010) != 0)
    printf("le bit 1 est égal à 1\n");
else
    printf("le bit 1 est égal à 0\n");

```

ce qui, en langage C, peut s'écrire encore plus simplement :

```

...
if(n & 0b0000010)
    printf("le bit 1 est égal à 1\n");
else
    printf("le bit 1 est égal à 0\n");

```

### 2.2.2 Test à 0

Considérons à nouveau la table de vérité du OU logique :

A	B	A OU B
0	0	0
0	1	1
1	0	1
1	1	1

On peut remarquer que le résultat d'un OU logique entre un bit variable (supposons qu'il s'agisse de A dans la table ci-dessus) et un 0 permet de savoir si la valeur du premier est 0. D'où le principe de test, en supposant que A est une variable à 1 bit (sous forme algorithmique, et en utilisant le symbole de l'opérateur OU bit-à-bit du langage C) :

si  $A \mid 0 == 0$

*action associée au fait que A soit est égal à 0*

Dans le cas de variables composées de plusieurs bits (c'est à dire le cas de toutes les variables en langage C), on utilise ce qu'on appelle un masque, c'est à dire une constante, composée d'un 0 à l'indice du bit que l'on souhaite tester, et de 1 partout ailleurs. Exemple :

```
...
if((n | 0b11111101) == 0b11111101)
    printf("le bit 1 est égal à 0\n");
else
    printf("le bit 1 est égal à 1\n");
```

*Remarque* : si l'on souhaite utiliser un masque composé de 1 aux indices des bits à tester (et non pas de 0 comme dans l'exemple ci-dessus), il suffit d'inverser le masque lors de son utilisation (à l'aide de l'opérateur d'inversion  $\sim$ ) :

```
...
if((n | ~0b00000010) == ~0b00000010)
    printf("le bit 1 est égal à 0\n");
else
    printf("le bit 1 est égal à 1\n");
```

*Remarque* : le test à 0 peut bien sûr également être basé sur le test à 1, en inversant l'ordre du test ; par exemple :

```
n = 0b10101010;
if(n & 0b00000010 != 0b00000010)
    printf("le bit 1 est égal à 0\n");
else
    printf("le bit 1 est égal à 1\n");
```

## 2.3 Inversion des bits d'une variable

### 2.3.1 Inversion de tous les bits

Exemple :

```
int a=0b11111010, b=0b00001111, c;
c = ~a & b;
printf("c=%d\n", c);          // résultat : b=5 (00000101 & 00001111 = 00000101)
```

### 2.3.2 Inversion d'une sélection de bits

Considérons une nouvelle fois la table de vérité du OU logique :



A	B	A OU B
0	0	0
0	1	1
1	0	1
1	1	0

On peut remarquer que, quand B=1, le résultat de l'opération est une inversion de A (voir valeur entourées). Le OU-exclusif avec un 1 peut donc être utilisé pour inverser les bits d'une variable. On réalise alors un masque, c'est à dire une variable ou une constante composée de 1 aux indices des bits que l'on veut inverser. Dans l'exemple suivant, on peut considérer par exemple que b est la variable à inverser et a le masque :

```
int a=0b00010101, b=0b00101010 ;
b ^= a;           // équivalent à b = b ^ a
printf("b=%x\n", b); // résultat : b=3f (00101010 ^ 00010101 = 00111111)
...
```

## 2.4 Décalage logique

La syntaxe de l'opération de décalage logique est la suivante :

- $x \ll n$  : décalage logique des bits de x, de n positions vers la gauche
- $x \gg n$  : décalage logique des bits de x, de n positions vers la droite

Exemple :

```
int n;
n=6<<1;
printf("6<<1=%d\n", n); //résultat : 12
n=6>>1;
printf("6>>1=%x\n", n); //résultat : 3
```

*Remarque* : on aurait pu écrire le même exemple différemment :

```
int n=6;
n <<= 1;
printf("6<<1=%d\n", n); //résultat : 12
n >>= 1;
printf("6>>1=%x\n", n); //résultat : 3
```

## Annexes

### 1. Séquences d'échappement

Certains caractères particuliers ne peuvent pas être imprimés directement dans un appel à **printf**. On insère à leur place dans la chaîne spécifiant le format d'affichage des séquences d'échappement.

### 2. Les opérateurs

Opérateur	Pluralité	Description
++, --	Unaire	Incrémentation et décrémentation
+, -	Unaire	Plus et moins unaires
(type)	Unaire	Conversion ("cast")
*, /	Binaire	Multiplication, division
%	Binaire	Modulo (reste)
+, -	Binaire	Addition, soustraction
<, <=, >, >=, ==, !=	Binaire	Comparaison
~	Unaire	Complément à un
<<	Binaire	Décalage à gauche
>>	Binaire	Décalage à droite
&	Binaire	ET bit à bit
^	Binaire	OU exclusif bit à bit
	Binaire	OU inclusif bit à bit
!	Unaire	Complément logique
&&	Binaire	ET conditionnel
	Binaire	OU conditionnel
^^	Binaire	OU exclusif conditionnel
? :	Ternaire	Opérateur ternaire de comparaison
=	Binaire	Affectation
*=, /=, %=, >>=, +=, -=,  =, &=, ^=, <<=	Binaire	Affectation avec opérateur associé

## Codes ASCII

Code ASCII	Caractère	Code ASCII	Caractère	Code ASCII	Caractère	Code ASCII	Caractère
000	NUL	032	espace	064	@	096	`
001	SOH	033	!	065	A	097	a
002	STX	034	"	066	B	098	b
003	ETX	035	#	067	C	099	c
004	EOT	036	\$	068	D	100	d
005	ENQ	037	%	069	E	101	e
006	ACK	038	&	070	F	102	f
007	BEL	039	'	071	G	103	g
008	BS	040	(	072	H	104	h
009	HT	041	)	073	I	105	i
010	LF	042	*	074	J	106	j
011	VT	043	+	075	K	107	k
012	FF	044	,	076	L	108	l
013	CR	045	-	077	M	109	m
014	SO	046	.	078	N	110	n
015	SI	047	/	079	O	111	o
016	DLE	048	0	080	P	112	p
017	DC1	049	1	081	Q	113	q
018	DC2	050	2	082	R	114	r
019	DC3	051	3	083	S	115	s
020	DC4	052	4	084	T	116	t
021	NAK	053	5	085	U	117	u
022	SYN	054	6	086	V	118	v
023	ETB	055	7	087	W	119	w
024	CAN	056	8	088	X	120	x
025	EM	057	9	089	Y	121	y
026	SUB	058	:	090	Z	122	z
027	ESC	059	;	091	[	123	{
028	FS	060	<	092	\	124	
029	GS	061	=	093	]	125	}
030	RS	062	>	094	^	126	~
031	US	063	?	095	_	127	DEL

*Note* : Les 32 premiers caractères ainsi que le dernier sont des caractères non imprimables.

### 3. Principales fonctions des bibliothèques standard

Fonction	Type	Rôle	Fichier d'en-tête
abs(i)	int	Renvoie la valeur absolue de i	stdlib.h
acos(d)	double	Renvoie l'arc-cosinus de d	math.h
asin(d)	double	Renvoie l'arc-sinus de d	math.h
atan(d)	double	Renvoie l'arc-tangente de d	math.h
atan2(d1,d2)	double	Renvoie l'arc-tangente de d1 / d2	math.h
atof(s)	double	Convertit la chaîne s en nombre en double précision	stdlib.h
atoi(s)	int	Convertit la chaîne s en nombre entier	stdlib.h
atol(s)	int	Convertit la chaîne s en nombre entier long	stdlib.h
calloc(u1,u2)	void*	Alloue la mémoire nécessaire pour un ensemble de u1 blocs de u2 octets. Renvoie un pointeur sur le début de la zone mémoire allouée.	malloc.h stdlib.h
ceil(d)	double	Renvoie la valeur spécifiée arrondie à l'entier immédiatement supérieur	math.h
cos(d)	double	Renvoie le cosinus de d	math.h
cosh(d)	double	Renvoie le cosinus hyperbolique de d	math.h
difftime(l1,l2)	double	Renvoie la différence l1 - l2 où l1 et l2 sont des valeurs de temps écoulé depuis une date de référence (voir la fonction time)	time.h
exit(u)	void	Ferme tous les flots et buffers et termine le programme. La valeur de u est affectée par la fonction et indique le code retour du programme	stdlib.h
exp(d)	double	Elève e à la puissance d (e=2.7182818... est la base des logarithmes népériens)	math.h
fabs(d)	double	Renvoie la valeur absolue de d	math.h
fclose(f)	int	Ferme le flot f et renvoie 0 en cas de fermeture normale	stdio.h
feof(f)	int	Détermine si une fin de flot est atteinte. Renvoie dans ce cas une valeur non nulle et 0 dans le cas contraire	stdio.h

fgetc(f)	int	Lit un caractère unique dans le flot f	stdio.h
fgets(s,i,f)	char*	Lit une chaîne s formée de i caractères dans le flot f	stdio.h
floor(d)	double	Renvoie la valeur spécifiée arrondie à l'entier immédiatement inférieur	math.h
fmod(d1,d2)	double	Renvoie le reste de la division de d1 par d2 (avec le signe de d1)	math.h
fopen(s1,s2)	file*	Ouvre le fichier de nom s1 dans le mode s2 (r, w, a, ...)	stdio.h
fprintf(f,...)	int	Ecrit des données sur le flot f	stdio.h
fputc(c,f)	int	Ecrit un caractère simple c sur le flot f	stdio.h
fputs(s,f)	int	Ecrit une chaîne s sur le flot f	stdio.h
fread(s,i1,i2,f)	int	Lit i2 données de longueur i1 (en octets) depuis le fichier f dans s (le type de s dépend des données lues)	stdio.h
free(p)	void	Libère la zone de mémoire débutant à l'adresse indiquée par p (la taille libérée dépend du type pointé par p)	malloc.h ou stdlib.h
fscanf(f,...)	int	Lit des données dans le flot f	stdlib.h
fseek(f,l,i)	int	Décale de l octets à partir de i le pointeur sur le flot f (i correspondant à n'importe quelle adresse entre le début et la fin du flot f)	stdio.h
ftell(f)	long int	Renvoie la position courante du pointeur dans le flot f	stdio.h
fwrite(s,il,i2,f)	int	Ecrit i2 données de longueur i1 (en octets) sur le flot f depuis s (le type de s dépend des données écrites)	stdio.h
getc(f)	int	Lit un caractère simple c à partir du flot f	stdio.h
getchar(void)	int	Lit un caractère simple sur le flot d'entrée standard	stdio.h
gets(s)	char*	Lit le chaîne s sur le flot d'entrée standard	stdio.h
isalnum(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un caractère alphanumérique. Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h

isalpha(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un caractère alphabétique (A-Z, a-z). Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
isascii(c)	int	Détermine si l'entier c donné correspond à un code ASCII. Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
isctrl(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un caractère de contrôle. Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
isdigit(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un chiffre (0 à 9). Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
isgraph(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un caractère graphique (041 à 126). Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
islower(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un caractère minuscule (a-z). Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
isodigit(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un chiffre octal (0 à 7). Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
isprint(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un caractère imprimable (040 à 126). Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
ispunct(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un signe de ponctuation. Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
isspace(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un caractère d'espacement. Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
isupper(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un caractère majuscule (A-Z). Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
isxdigit(c)	int	Détermine si l'entier c donné correspond au code ASCII d'un chiffre hexadécimal (0-9, A-F). Renvoie une valeur différente de zéro si c'est le cas, zéro sinon	ctype.h
labs(l)	long int	Renvoie la valeur absolue de l	math.h
log(d)	double	Renvoie le logarithme népérien de d	math.h
log10(d)	double	Renvoie le logarithme décimal de d	math.h

malloc(u)	void*	Alloue u octets de mémoire et renvoie l'adresse du début de la zone allouée	stdlib.h
pow(d1,d2)	double	Renvoie la valeur de d1 élevé à la puissance d2	math.h
printf(...)	int	Ecrit des données sur le flot de sortie standard	stdio.h
putc(c,f)	int	Ecrit un caractère sur le flot f	stdio.h
putchar(c)	int	Ecrit un caractère sur le flot de sortie standard	stdio.h
puts(s)	int	Ecrit une chaîne s sur le flot de sortie standard	stdio.h
rand(void)	int	Renvoie un nombre entier positif aléatoire (voir srand)	stdlib.h
realloc(p,u)	void*	Modifie la taille de la zone allouée à un pointeur p avec u octets. Le contenu pointé n'est pas modifié sauf si la taille de la zone est diminuée	stdlib.h
rewind(f)	void	Repositionne le pointeur du flot f au début de ce dernier	stdio.h
scanf(...)	int	Lit des données sur le flot d'entrée standard	stdio.h
sin(d)	double	Renvoie le sinus de d	math.h
sinh(d)	double	Renvoie le sinus hyperbolique de d	math.h
sscanf(s,...)	int	Lit des données à partir de la chaîne s	stdio.h
sprintf(s,...)	int	Ecrit des données dans la chaîne s	stdio.h
sqrt(d)	double	Renvoie la racine carrée de d	math.h
srand(u)	void	Initialise le générateur de nombre aléatoire	stdlib.h
strcat(s1,s2)	char*	Concaténation de s1 et de s2 dans s1	string.h
strcmp(s1,s2)	int	Comparaison lexicographique des chaînes s1 et s2. renvoie une valeur négative si s1 < s2, zéro si s1 = s2 et positive si s1 > s2	string.h
strcasecmp(s1,s2)	int	Idem strcmp mais sans distinction des majuscules et minuscules	string.h

strcpy(s1,s2)	char*	Copie de s2 dans s1	string.h
strlen(s)	int	Renvoie la longueur de la chaîne s	string.h
strset(s,c)	char*	Remplace tous les caractères de la chaîne s par le caractère c (sauf la fin de chaîne '\0')	string.h
system(s)	int	Transmet au système d'exploitation la commande contenue dans la chaîne s. Renvoie zéro en cas d'exécution correcte, une valeur < 0 sinon (souvent -1)	stdlib.h
tan(d)	double	Renvoie la tangente de d	math.h
tanh(d)	double	Renvoie la tangente hyperbolique de d	math.h
time(p)	long int	Renvoie le nombre secondes écoulées depuis une date spécifiée	time.h
toascii(c)	int	Renvoie la conversion le code ASCII de c	ctype.h
tolower(c)	int	Renvoie la conversion en minuscule de c	ctype.h
toupper(c)	int	Renvoie la conversion en majuscule de c	ctype.h

Notes: Le type indiqué est celui de la valeur retournée par la fonction. La mention d'un astérisque (\*) indique un pointeur.

c	indique un argument de type caractère
d	indique un argument de type réel en double précision
f	indique un argument de type fichier
i	indique un argument de type entier
l	indique un argument de type entier long
p	indique un argument de type pointeur
s	indique un argument de type chaîne
u	indique un argument de type entier non signé

La plupart des compilateurs C sont accompagnés de nombreuses bibliothèques de fonctions. Le lecteur se reportera au manuel de référence du compilateur qu'il emploie pour des précisions sur l'ensemble des fonctions dont il dispose.



**Exemple d'utilisation : génération de nombres aléatoires**

```
#include<stdlib.h>
#include<time.h>
main()
{
    int a,b,c;
    //initialisation du générateur de nombres aléatoires
    srand(time(NULL));
    a=rand();          //a contient un nombre aléatoire compris entre 0 et  $2^{32}-1$ 
    b=rand()%10;       //b contient un nombre aléatoire compris entre 0 et 9
    c=rand()%5+2;      //c contient un nombre aléatoire compris entre 2 et 6
    printf("a=%d, b=%d, c=%d",a,b,c);
}
```

## 4. Norme d'écriture des programmes en C

### a. Commentaires

#### En-tête :

Chaque source de programme doit obligatoirement commencer par une *cartouche d'en-tête* en commentaire contenant les informations suivantes : nom et prénom du ou des élèves l'ayant développé, nom du binôme éventuel, nom du source, n° du TP ou du DS concerné, date de création, description sommaire et commentaires, type et rôle des arguments éventuels. Exemple :

```

/*****
/* Développé par : NOM1 Prénom1, NOM2 Prénom2 - alg0b0 */
/* Source : exemple.c - TP n°1 - le 01/10/03 */
/* Description : programme d'exemple pour les normes */
/* Commentaires : code développé au point b. */
/* Arguments : néant */
*****/

```

#### Instructions pré-processeur :

Ces instructions doivent toutes être commentées.

```

/* inclusion des librairies */
#include <stdio.h> /* librairie standard de gestion des E/S */

/* définition des constantes */
#define ffl __fpurge(stdin) /* raccourci purge du flot d'entree */
#define MAX 10 /* nombre d'éléments du tableau d'entiers */
...

```

#### Déclaration de fonctions :

Toute fonction développée dans le programme doit d'abord être prototypée avant la fonction principale main() et son implémentation après cette dernière doit être précédée d'un commentaire expliquant son rôle et précisant les particularités éventuelles. Exemple :

```

/* prototypes des fonctions */
int calcul_somme(int , int);

main(void)
{
    ...
}

/*****
/* Fonction calcul_somme : calcule la somme de 2 entiers */
/*                               passes en argument */
*****/
int calcul_somme(int dp_nb1, int dp_nb2)
{
    return(dp_nb1 + dp_nb2);
}

```

### Déclaration de variables :

Toute variable déclarée doit obligatoirement être suivie d'un commentaire expliquant son rôle ou son utilisation dans le programme, ainsi que sa portée (ou visibilité). Exemple :

```
char cg_conf = 'n'; /* saisie de confirmation (o/n), global */  
int dl_compt = 0; /* compteur de boucle, local */
```

#### *b. Lisibilité*

### Indentation :

L'indentation correspond au nombre d'espaces (ou blancs) insérés devant le premier caractère imprimable d'une ligne de code. Ce nombre d'espaces dépend du niveau d'imbrication (profondeur) du bloc de programme auquel appartient la ligne en question. Exemple au paragraphe suivant.

### Sauts de ligne :

Il est souvent utile pour la lisibilité du code de laisser une ligne blanche pour séparer certains ensembles d'instructions (exemple : entre la dernière déclaration de variable et la première instruction suivante).

### Espacement, parenthèses, accolades, virgule et séparateur d'instructions ( ;) :

- deux "mots" consécutifs (mots réservés, noms de variables et de fonctions, opérateurs,...) sont séparés par un espace sauf pour les opérateurs unaires (++ et --) et l'opérande auquel ils s'appliquent ;
- une parenthèse ouvrante n'est jamais suivie d'un espace, ni d'un "fin de ligne" ;
- une parenthèse fermante n'est jamais précédée d'un espace, ni d'un "fin de ligne" ;
- une accolade ouvrante est toujours suivie d'un "fin de ligne" et est située sous le premier caractère imprimable de la ligne précédente ;
- une accolade fermante est toujours suivie d'un "fin de ligne" et est alignée avec l'accolade ouvrante précédente ;
- contrairement à l'accolade, une parenthèse ne figure jamais toute seule sur une ligne ;
- un bloc de code formé d'une seule ligne n'est pas délimité par des accolades ;
- la virgule est toujours suivie d'un espace ;
- le séparateur d'instructions suit toujours le dernier caractère imprimable de l'instruction précédente ou le séparateur précédent (cas du "for" par exemple) ;

**Exemple :**

```

/*****
/* Développé par : NOM1 Prénom1, NOM2 Prénom2 - groupe N */
/* Source : exemple.c - TP n°1 - le 02/09/13 */
/* Description : programme d'exemple pour les normes */
/* Commentaires : néant */
/* Arguments : néant */
*****/
/* inclusion des librairies */
#include<stdio.h>

/* definition des constantes */
#define ffl __purge(stdin)

/* fonction main : fonction principale du programme */
main()
{
    /* variables locales */
    float fl_x1,fl_y1; /* Coordonnées du point P1 */
    float fl_x2,fl_y2; /* Coordonnées du point P2 */
    float fl_xmoy, fl_ymoy; /* Coordonnées du milieu de P1 et P2 */

    /* instructions */
    system("clear");

    printf("\nEntrez la coordonnee x1 de P1 : ");
    ffl;
    scanf("%f", &fl_x1);
    printf("\nEntrez la coordonnee y1 de P1 : ");
    ffl;
    scanf("%f", &fl_y1);
    printf("\nEntrez la coordonnee x2 de P2 : ");
    ffl;
    scanf("%f", &fl_x2);
    printf("\nEntrez la coordonnée y2 de P2 : ");
    ffl;
    scanf("%f", &fl_y2);

    if((fl_x1 == fl_x2) && (fl_y1 == fl_y2))
        printf("\n Les points sont confondus !!! Recommencez.");
    else
    {
        fl_xmoy = (fl_x1 + fl_x2) / 2;
        fl_ymoy = (fl_y1 + fl_y2) / 2;
        printf("\nLe point milieu de P1 et P2 a pour");
        printf("\ncoordonnees (X,Y): (%f,%f)", fl_xmoy, fl_ymoy);
    }
}

```