chapitre 1 :

Introduction générale au langage C

Objectif :

- **Décrire** les concepts relatifs à l'écriture d'un programme en langage C.

Eléments de contenu :

I- Historique :

Dans les dernières années, aucun langage de programmation n'a pu se vanter d'une croissance en popularité comparable à celle du **langage C** et de son jeune frère **C++**. L'étonnant dans ce fait est que le langage C n'est pas un nouveau-né dans le monde informatique, mais qu'il trouve ses sources en 1972 dans les **'Bell Laboratories'**: Pour développer une version portable du système d'exploitation **UNIX**, Dennis M. Ritchie a conçu ce langage de programmation structuré, mais très 'près' de la machine.

Le succès des années qui suivaient et le développement de compilateurs C par d'autres maisons ont rendu nécessaire la définition d'un standard actualisé et plus précis. En 1983, le **'American National Standards Institute'** (**ANSI**) chargeait une commission de mettre au point 'une définition explicite et indépendante de la machine pour le langage C', qui devrait quand même conserver l'esprit du langage. Le résultat était le **standard ANSI-C**.

En 1983 un groupe de développeurs de **AT&T** sous la direction de Bjarne Stroustrup a créé le langage **C++**. Le but était de développer un langage qui garderait les avantages de **ANSI-C** (portabilité, efficience) et qui permettrait en plus la programmation orientée objet. Depuis 1990 il existe une ébauche pour un **Standard ANSI-C++**. Entre-temps, **AT&T** a développé deux compilateurs **C++** qui respectent les nouvelles déterminations de **ANSI** et qui sont considérés comme des quasi-standards.

II- Avantages :

Le grand succès du langage C s'explique par les avantages suivants; C est un langage :

1- Universel :

Le langage C n'est pas orienté vers un domaine d'applications spéciales, comme par exemple FORTRAN (applications scientifiques et techniques) ou COBOL (applications commerciales ou traitant de grandes quantités de données).

2- Compact :

Le langage C est basé sur un noyau de fonctions et d'opérateurs limité, qui permet la formulation d'expressions simples, mais efficaces.

3- Moderne :

Le langage C est un langage structuré, déclaratif et récursif; il offre des structures de contrôle et de déclaration comparables à celles des autres grands langages de ce temps (FORTRAN, ALGOL68, PASCAL).

4- Près de la machine :

Comme C a été développé en premier lieu pour programmer le système d'exploitation UNIX, il offre des opérateurs qui sont très proches de ceux du langage machine et des fonctions qui permettent un accès simple et direct aux fonctions internes de l'ordinateur (par .exp: la gestion de la mémoire).

6- Portable :

En respectant le standard **ANSI-C**, il est possible d'utiliser le même programme sur tout autre système (autre hardware, autre système d'exploitation), simplement en le recompilant.

7- Extensible :

Le langage C ne se compose pas seulement des fonctions standards ; il est enrichi par des bibliothèques de fonctions privées ou livrées par de nombreuses maisons de développement.

III- Désavantages :

Evidemment, rien n'est parfait. Jetons un petit coup d'oeil sur le revers de la médaille:

1- Efficience et compréhensibilité :

En C, nous avons la possibilité d'utiliser des expressions compactes et efficientes. D'autre part, nos programmes doivent rester compréhensibles pour nous-mêmes et pour d'autres. Comme nous allons le constater sur les exemples suivants, ces deux exigences peuvent se contredire réciproquement.

Exemple :

Les deux lignes suivantes impriment les N premiers éléments d'un tableau A[], en insérant un espace entre les éléments et en commençant une nouvelle ligne après chaque dixième chiffre:

```
for (i=0; i<n; i++)
        printf("%6d%c", a[i], (i%10==9)?'\n':' ');
```

Cette notation est très pratique, mais plutôt intimidante pour un débutant. L'autre variante, plus près de la notation en Pascal, est plus lisible, mais elle ne profite pas des avantages du langage C:

```
for (i=0; i<n; i=i+1)
    {
      printf("%6d", A[i]);
      if ((i%10) == 9)
        printf("\n");
      else
        printf(" ");
     }
```

2- Limites de la portabilité :

La portabilité est l'un des avantages les plus importants de C: en écrivant des programmes qui respectent le standard ANSI-C, nous pouvons les utiliser sur n'importe quelle machine possédant un compilateur ANSI-C. D'autre part, le répertoire des fonctions ANSI-C est assez limité. Si un programmeur désire faire appel à une fonction spécifique de la machine (par .exp: utiliser une carte graphique spéciale), il est assisté par une foule de fonctions 'préfabriquées', mais il doit être conscient qu'il risque de perdre la portabilité. Ainsi, il devient évident que les avantages d'un programme portable doivent être payés par la restriction des moyens de programmation.

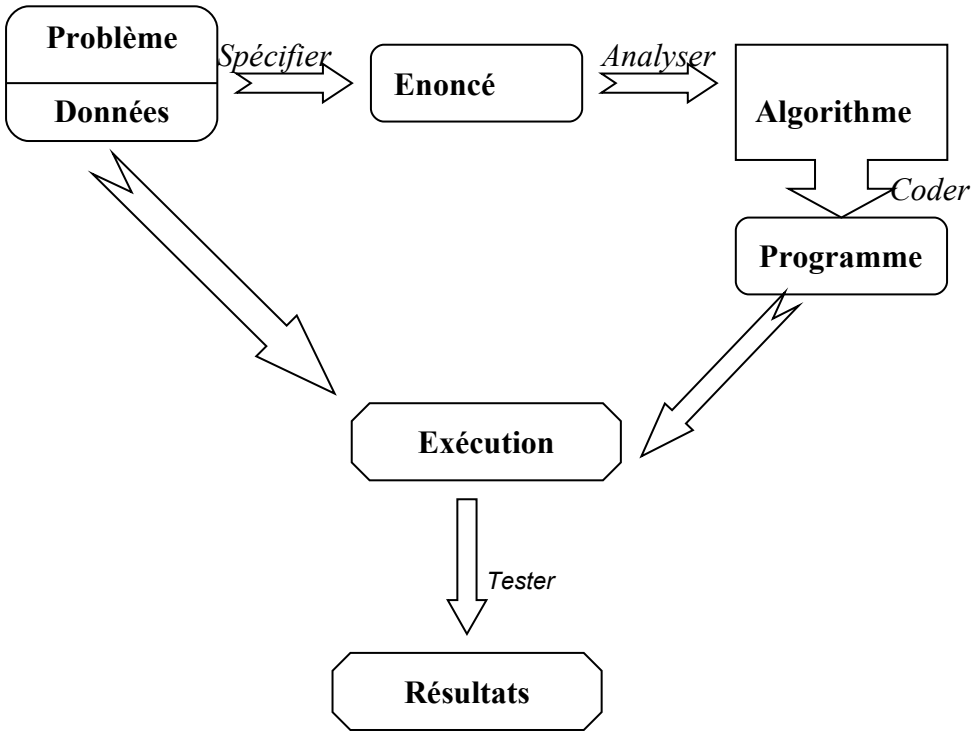
3- Discipline de programmation : Les dangers de C :

Nous voici arrivés à un point crucial: C est un langage proche de la machine, donc dangereux et bien que C soit un langage de programmation structurée, il ne nous force pas à adopter un certain style de programmation. Dans un certain sens, tout est permis et la tentation de programmer du 'code spaghetti' est grande. (même la commande 'goto', si redoutée par les puristes ne manque pas en C). Le programmeur a donc beaucoup de libertés, mais aussi des responsabilités: il doit veiller lui-même à adopter un style de programmation propre, solide et compréhensible.

IV- Les étapes de réalisation d'un programme C :

1- Activité de programmation :

L'activité de programmation pet se résumer selon le schéma suivant :



2- Les étapes de réalisation d'un programme C :

L'environnement de **Turbo C** appelé « **environnement intégré** » permet le développement complet de programmes, de leur **saisie** (écriture), à leur **exécution** en passant par leur **compilation** et leur **édition de liens**. On distingue alors quatre étapes :

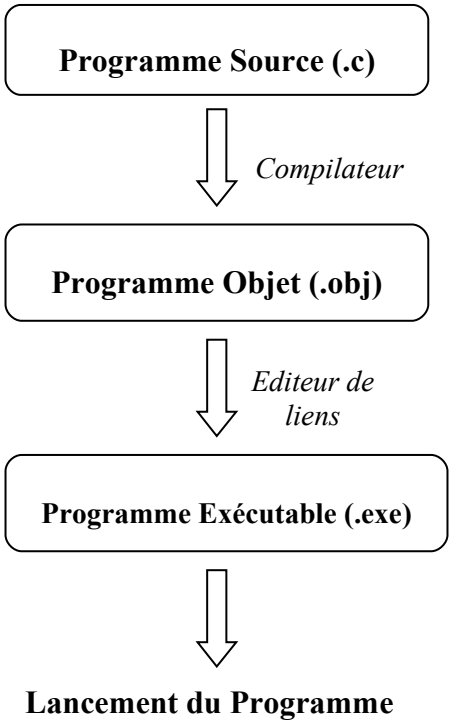
❶ Ecriture de programme ou édition : Il s'agit de la phase de **saisie** du texte du programme, puis sa sauvegarde sous un nom d'extension « **.c** ». Ce fichier sera appelé **code source** ou **fichier source** (Exp : test.c).

❷ Compilation : Cette étape consiste à **analyser syntaxiquement** le programme source et à le traduire en **langage machine** (par le compilateur). Le fichier produit comme résultat est appelé **code objet** ayant pour extension « **.obj** » (Exp : test.obj).

❸ Edition de liens : Le code objet créé au cours de la phase de compilation, bien que constitué d'instructions en langage machine, n'est pas directement exécutable. Il lui manque pour cela un ensemble de modules objets correspondant aux fonctions appelées par le programme déjà écrit (fonctions déjà **prédéfinies** au niveau de la **bibliothèque du langage**) : C'est le rôle de l'**éditeur de liens** qui effectuera l'incorporation de ces modules au niveau du programme. Le résultat de l'édition de liens est le **programme exécutable** ayant pour extension « **.exe** » (Exp : test.exe).

❹ Exécution : C'est la phase d'**appel du programme** (version exécutable) par son nom pour avoir les **résultats**.

En résumé, on a :



Remarque : Il est conseillé d'écrire des programmes où il y a des commentaires pour les rendre plus lisibles et plus compréhensibles.

3- Structure d'un programme C :

Un programme C est **une suite de fonctions**, chacune commence par « **{**« et se termine par « **}** » et dont la fonction représentant le programme principal s'appelle « **main ( )** ». La structure d'un programme C se présente comme suit :

```
Void main()
  {
      …      // Déclarations
      …      // Instructions
      …
  }

fonction1(…)
  {
      …
      …
  }
```

V- Règles générales d'écriture d'un programme C :

1- Les identificateurs :

Ils servent à désigner les différents objets manipulés par le programme (variables, fonctions,…). C'est une suite de caractères alphanumériques commençant obligatoirement par une lettre.

Remarques :

- ✘ Le caractère « _ » est considéré comme une lettre.
- ✘ Le langage C fait la différence entre les lettres majuscules et les lettres minuscules.
- ✘ La longueur d'un identificateur en langage C est fixée à 32 caractères (on peut la modifier avec Option/Compiler/Source/Identifier/Length).

2- Les mots-clés :

Ils représentent l'ensemble des mots de déclaration de types de variables et de types de fonctions et bien d'autres mots (void, char, int, if, for,…).

3- Les séparateurs :

Ils représentent l'ensemble des caractères qu'on utilise pour séparer les identificateurs, les différentes instructions,…(, ; …).

4- Les commentaries :

Le langage C autorise l'utilisation des commentaires notés entre « /* » et « */ » ou précédé par « // » s'il s'agit d'un commentaire sur une seule ligne, mais il n'accepte pas les commentaires imbriqués.

chapitre 2 :

Types de base en langage C

Objectif :

Expliquer les concepts relatifs aux types de base et aux déclaration des variables en langage C.

Eléments de contenu :

I- Introduction :

En C il existe quatre types de base :

- Les entiers (1, 2, -5, 157, etc.) : type **int**.
- Les nombres en points flottants (325, 45.14, 3.14, -47.1245, etc.) : type **float**.
- Les textes : caractères ou chaînes (‘A’, « Bonjour », etc.) type **char**.
- Les pointeurs représentent une adresse mémoire contenant de l’information.

II- Le type Entier :

Ce type représente les données entières. La déclaration d’une variable entière doit être de la forme :

int nom_variable ;

C permet aussi de préciser si les entiers doivent être de types longs (**Long int**) ou de type court (**Short int**). Ces extensions permettent de choisir les valeurs maximales et minimales que peuvent contenir les variables. La différence provient du nombre d’octets utilisés pour représenter la variable en mémoire :

- Deux octets pour **int** et **short int**.
- Quatre octets pour **long int**.

C permet la précision du signe de la variable. Si la variable ne peut contenir que des valeurs positives ou nulles on utilise **unsigned int**.

Le tableau suivant résume les différentes valeurs pour chaque type :

Type	Valeur minimale	Valeur maximale
int	-32 768	32 767
long int	-2 147 483 648	2 147 483 647
unsigned int	0	65 535
unsigned long int	0	4 294 967 295

Remarque :

Il est possible de préciser qu’une constante entière doit être de type long. Il suffit de faire suivre cette constante de la lettre **L**.

Exemple :

```
long i ;
…
i=587L ;
```

III- Le type float :

Ce type représente les données numériques réelles. La déclaration d'un tel type de variables sera de la forme :

float nom_variable ;

Il y a également une extension du type **float**, il s'agit du type **double**.

Le tableau suivant résume les différentes valeurs pour chaque type :

Type	Valeur minimale	Valeur maximale
float	3.4^{E}-38	3.4^{E}-38
double	1.7^{E}-308	1.7E+308

NB : 3.4^{E}-38 signifie 3.4 * 10 exposant –38.

Remarques :

- L'ensemble entier est un sous type du type réel.
- Les calculs sur les nombres réels prennent plus de temps, il est conseillé de n'utiliser ces nombres que si c'est vraiment nécessaire.

IV- Le type char :

C permet d'utiliser des variables caractères et des textes. La déclaration d'une variable caractère doit être de la forme :

char nom_variable ;

Les valeurs possibles pour ce type de variables correspondent aux caractères ASCII. Un caractère est représenté en mémoire par un seul octets.

Exemple :

```
char c ;
c='A' ;
```

C offre également la possibilité de définir et de manipuler des variables textes. Cela peut se faire de deux façon : soit en précisant la longueur lors de la déclaration, soit en ne précisant pas cette longueur.

Pour le premier cas, la déclaration prend la forme suivante :

Char nom_du_texte[longueur du texte]

Exemple :

Char ch1[10] ;

La variable ch1 peut contenir un texte de 9 caractères. Le dixième étant réservé par l'ordinateur pour signaler la fin du texte.

Pour assigner une valeur texte à une variable de ce type, il faut utiliser une fonction spéciale. Il s'agit de la fonction prédéfinie **strcpy** dont la syntaxe est la suivante :

strcpy (nom_variable, valeur_texte) ;

nom_variable : c'est une variable déclarée comme une chaîne de caractères.

Valeur_texte : correspond à la valeur que l'on veut assigner à la variable.

Exemple :

char texte[30] ;

strcpy(texte, « ceci est un exemple ») ;

Pour le second cas, il s'agit de déclarer un pointeur vers un ensemble de caractères. La déclaration prend la forme suivante :

char *nom_du_texte ;

Exemple :

char *ch1 ;

ch1= « exemple » ;

Remarques :

- Dans le premier cas, la place nécessaires pour la représentation de la variable dans la mémoire de l'ordinateur, est réservée définitivement lors de la déclaration.
- Dans le second cas, la place n'est réservée qu'au moment où l'on affecte une valeur à la variable.
- Le deuxième type est donc le plus souple à utiliser, car il ne nécessite pas de connaître la longueur maximale des valeurs que l'on va assigner à la variable.

chapitre 3 :

Opérateurs et expressions en langage C

Objectif :

- **Distinguer** les différents opérateurs et expressions relatifs au langage C.

Eléments de contenu :

I- Introduction :

Qu'est ce qu'un opérateur ?

Le langage C dispose d'une multitude d'opérateurs. Cette richesse se manifeste tout d'abord au niveau des opérateurs classiques (arithmétiques, relationnels, logiques) ou moins classiques (manipulation de bits). C dispose aussi d'un important éventail d'opérateurs originaux d'affectation et d'incrémentation.

Les opérateurs sont des symboles qui permettent de manipuler des variables, c'est-à-dire effectuer des opérations, les évaluer, On distingue plusieurs types d'opérateurs:

- les opérateurs de calcul.
- les opérateurs d'assignation.
- les opérateurs d'incrémentation.
- les opérateurs de comparaison.
- les opérateurs logiques.
- les opérateurs bit à bit.

II- Les opérateurs arithmétiques :

Les opérateurs arithmétiques ou de calcul permettent de modifier mathématiquement la valeur d'une variable. Ces opérateurs sont représentés au niveau du tableau suivant :

Opérateur	Désignation	Effet	Exemple	Résultat (avec x valant 7)
+	Opérateur d'addition	Additionne deux valeurs	x+3	10
-	Opérateur de soustraction	Soustrait deux valeurs	x-3	4
*	Opérateur de multiplication	Multiplie deux valeurs	x*3	21
/	Opérateur de division	Divise deux valeurs	x/3	2.3333333
%	Opérateur modulo	Donne le reste de la division entière	x%3	1
-	Moins unaire	Donne le signe moins au nombre devant lequel il est mis	-x	-7

En utilisant ces opérateurs, on peut écrire des expressions arithmétiques. Pour les évaluer, il faut respecter les règles de priorité suivantes :

- Si l'expression contient des parenthèses, les expressions se trouvant entre ces dernières seront évaluées en premier lieu.
- Si l'expression ne contient pas de parenthèses, l'évaluation se fait de gauche à droite en respectant l'ordre de priorité suivant :
    - Le moins unaire.
    - La multiplication (*), la division (/), le modulo (%).
    - L'addition (+), la soustraction (-).

III- Les opérateurs de comparaison :

Le langage C permet de comparer des expressions à l'aide d'opérateurs classiques de comparaison (ou relationnels). Le résultat de la comparaison n'est pas une valeur **booléenne**, mais un entier qui vaut :

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

Opérateur	Désignation	Effet	Exemple	Résultat (avec x valant 7)
= =	Opérateur d'égalité	Compare deux valeurs et vérifie leur égalité	x= =3	Retourne 1 si x est égal à 3, sinon 0. Ici, retourne 0.
<	Opérateur d'infériorité stricte	Vérifie qu'une variable est strictement inférieure à une valeur	x<3	Retourne 1 si x est inférieur à 3, sinon 0. Ici retourne 0.
<=	Opérateur d'infériorité	Vérifie qu'une variable est inférieure ou égale à une valeur	x<=3	Retourne 1 si x est inférieur à 3, sinon 0. Ici retourne 0.
>	Opérateur de supériorité stricte	Vérifie qu'une variable est strictement supérieure à une valeur	x>3	Retourne 1 si x est supérieur à 3, sinon 0. Ici retourne 1.
>=	Opérateur de supériorité	Vérifie qu'une variable est supérieure ou égale à une valeur	x>=3	Retourne 1 si x est supérieur ou égal à 3, sinon 0. Ici retourne 1.
!=	Opérateur de différence	Vérifie qu'une variable est différente d'une valeur	x!=3	Retourne 1 si x est différent de 3, sinon 0. Ici retourne 1.

Les opérateurs de comparaison sont moins prioritaires que les opérateurs arithmétiques.

Remarque : Le langage C ne possède pas de type booléen.

IV- Les opérateurs logiques (Booléens) :

Le langage C dispose de trois opérateurs logiques :

Opérateur	Dénomination	Effet	Syntaxe
||	OU logique	Vérifie qu'une des conditions est réalisée	((condition1)||(condition2))
&&	ET logique	Vérifie que toutes les conditions sont réalisées	((condition1)&&(condition2))
!	NON logique	Inverse l'état d'une variable booléenne (retourne la valeur 1 si la variable vaut 0, 0 si elle vaut 1)	(!condition)

Remarque : Toute valeur non nulle correspond à vrai.

L'opérateur || est moins prioritaire que l'opérateur **&&**, alors que l'opérateur **!** est le plus prioritaire.

Les opérateurs logiques sont moins prioritaires que les opérateurs de comparaison.

V- Les opérateurs d'affectation :

L'opération d'affectation consiste à mettre une valeur dans un variable déjà déclarée. C'est-à-dire préciser la donnée qui va être stockée à l'emplacement mémoire qui a été réservé lors de la déclaration.

Pour cela on utilise l'opérateur d'affectation "=" : **Nom_de_la_variable = valeur**;

Pour stocker le caractère B dans la variable que l'on a appelée *Caractere*, il faudra écrire: **Caractere = 'B'**. Il est bien évident qu'il faut avoir préalablement déclaré la variable en lui affectant le type **char**: **char Caractere**;

Remarque : L'opérateur d'affectation possède une associativité de droite à gauche.

Exemple : i= j = 5, c'est comme si on avait fait : j = 5 puis i = 5.

VI- Les opérateurs d'incrémentation :

Ce type d'opérateur permet de facilement augmenter ou diminuer d'une unité la valeur d'une variable. Ces opérateurs sont très utiles pour des structures telles que des boucles, qui ont besoin d'un compteur (variable qui augmente de un en un).

Un opérateur de type *x*++ permet de remplacer des notations lourdes telles que **x=x+1**.

Opérateur	Dénomination	Effet	Syntaxe	Résultat (avec x valant 7)
++	Incrémentation	Augmente d'une unité la variable	x++	8
--	Décrémentation	Diminue d'une unité la variable	x--	6

VII- Les opérateurs d'affectation élargie :

Ces opérateurs permettent de simplifier des opérations telles que **ajouter une valeur à une variable et stocker le résultat dans la même variable**. Une telle opérations s'écrirait habituellement de la façon suivante par exemple: **x=x+2**. Avec les opérateurs d'affectation élargie, il est possible d'écrire cette opération sous la forme suivante: **x+=2**. Ainsi, si la valeur de x est 7 avant opération, elle sera de 9 après...

Les autres opérateurs du même type sont les suivants:

Opérateur	Effet
+=	additionne deux valeurs et stocke le résultat dans la variable (à gauche)
-=	soustrait deux valeurs et stocke le résultat dans la variable
*=	multiplie deux valeurs et stocke le résultat dans la variable
/=	divise deux valeurs et stocke le résultat dans la variable

chapitre 4 :

Les entrées / sorties en langage C

Objectif :

- **Décrire** les différentes fonctions d'E/S utiles en langage C.

Eléments de contenu :

I- Introduction :

Le but général d'un programme est de traiter un certain nombre de données et de produire des résultats. Les données sont fournies au programme via des instructions d'entrées et les résultats sont répercutés vers l'extérieur via des instructions de sorties. Ces instructions d'entrées / sorties ou I/O (Input/Output) transmettent des informations entre la mémoire de l'ordinateur et un support extérieur. Ce support peut être dans le cas d'une instruction d'entrée :

- Le clavier
- Un fichier disque
- Une souris
- Un instrument de mesure quelconque, …

Et dans le cas d'une instruction de sortie :

- L'écran
- Un fichier disque
- Une imprimante

Dans le langage C, les entrées / sorties se font par l'intermédiaire de fonctions standards. Dans la suite nous présenterons des fonctions permettant d'introduire les données à partir du clavier et d'afficher le résultat sur l'écran.

II- La fonction scanf :

Cette fonction lit une liste de variables selon un format donné. Sa syntaxe générale est la suivante :

scanf("<format>",<AdrVar1>,<AdrVar2>, ...)

Avec:

- "<format>" : format de lecture des données.
- <AdrVar1>,... : adresses des variables auxquelles les données seront attribuées

L'adresse d'une variable est indiquée par le nom de la variable précédé du signe **&**.

Spécificateurs de format pour scanf :

SYMBOLE	LECTURE D'UN(E)	TYPE
`%d ou %i`	entier relatif	`int*`
`%u`	entier naturel (unsigned)	`int*`
`%o`	entier exprimé en octal	`int*`
`%b`	entier exprimé en hexadécimal	`int*`
`%c`	caractère	`char*`
`%s`	chaîne de caractères	`char*`
`%f ou %e`	rationnel en notation décimale ou exponentielle (scientifique)	`float*`

Le symbole ***** indique que l'argument n'est pas une variable, mais l'*adresse* d'une variable de ce type.

On peut placer la longueur de k=la variable entre le signe % et la lettre. Ainsi « %3d » indique que l'on va lire un entier de 3 chiffres.

Un blanc mis entre deux spécifications de format signifie que l'on peut mettre autant de blanc ou de lignes que l'on veut entre les valeurs à lire. Si l'on sépare les spécifications par une virgule, alors les valeurs à lire seront séparées par des virgules. Lorsque la liste des données entrées est complète, il faut, pour signaler la fin de cette liste, taper la touche *Enter*.

Exemple :

```
int i ;
float r ;
char c ;

scanf ("%d %f  %c », &i, &r, &c) ;
scanf ("%d,%f ,%c », &i, &r, &c) ;
```

- Dans le premier cas les valeurs à lire pourraient être : 5487 2.1456 d
- Dans le deuxième cas les valeurs à lire pourraient être : 54,3.14,f

III- La fonction gets :

Cette fonction est utilisée pour la lecture de chaîne de caractères. Sa syntaxe générale est la suivante :

gets(nom de chaîne)

Lorsqu'on lit une chaîne de caractère avec l'instruction scanf(), la lecture s'arrête dès que l'on rencontre un blanc. Avec la fonction gets(), la lecture s'effectue jusqu'à ce que la touche *Enter* soit enfoncée.

Remarque :

Pour lire une variable de type pointeur vers un ensemble de caractère (*char), il faut affecter à cette variable l'adresse renvoyée par gets().

Exemples :

```
char c[20] ;
gets (c) ;
```

ou encore

```
char*c, *aux ;
c=gets(aux) ;
```

IV- La fonction getch() :

Cette fonction lit un caractère unique. La valeur renvoyée par la fonction correspond au caractère lu. Sa syntaxe générale est la suivante :

getch(nom du caractère)

Exemple :

```
char c ;
c=getch() ;
```

V- La fonction printf :

Cette fonction écrit des informations selon un format donné. Sa syntaxe générale est la suivante :

printf("<format>",<Expr1>,<Expr2>, ...)

Avec :

- "<format>" : format de représentation
- <Expr1>,... : variables et expressions dont les valeurs sont à représenter

La partie "***<format>"*** est en fait une chaîne de caractères qui peut contenir:

- *du texte*
- *des séquences d'échappement*
- *des spécificateurs de format*

Les ***spécificateurs de format*** indiquent la manière dont les valeurs des expressions <Expr1..N> sont imprimées.

La partie "<format>" contient *exactement un* spécificateur de format pour chaque expression <Expr1..N>.

Les spécificateurs de format commencent toujours par le symbole **%** et se terminent par un ou deux caractères qui indiquent le format d'impression.

Spécificateurs de format pour printf :

SYMBOLE	TYPE	IMPRESSION COMME
`%d ou %i`	`int`	entier relatif
`%u`	`int`	entier naturel (unsigned)
`%o`	`int`	entier exprimé en octal
`%x`	`int`	entier exprimé en hexadécimal
`%c`	`int`	caractère
`%f`	`float`	rationnel en notation décimale
`%e`	`float`	rationnel en notation scientifique
`%s`	`char*`	chaîne de caractères

On peut insérer des séquences spéciales appelées « séquences espaces », qui permettent de contrôler l'affichage des données.

Les principales espaces sont :

- \n : Passage à la ligne suivante.
- \r : Retour du curseur en début de ligne.
- \b : Déplacement du curseur d'un caractère vers la droite.
- \f : Déplacement à la page suivante.
- \t : Déplacement du curseur d'une tabulation vers la droite.
- \a : Déclenchement d'un signal sonore.

Remarque :

Si on veut afficher le caractère \, on double ce caractère dans la chaîne à afficher, c'est à dire qu'on met \\.

Exemple :

```
int i,
printf(« Entrer une valeurs :\n ») ;
scanf(« %d », &i) ;
i=i*2;
printf("valeur *2=%d", i);
```

VI- La fonction puts :

Cette fonction affiche une chaîne de caractères. Sa syntaxe générale est la suivante :

puts(nom_de_chaîne)

La fonction puts affiche « nom_de_chaîne » et passe automatiquement à la ligne suivante.

Exemple :

```
puts(« ceci est un programme C ») ;
```

VII- La fonction putchar :

Cette fonction affiche une caractère. Sa syntaxe générale est la suivante :

putchar(élément)

Où élément est une variable de type caractère ou une valeur caractère.

Exemple :

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

chapitre 5 :

Les structures de contrôle

Objectif :

Ecrire des programmes en utilisant les différentes structures de contrôle en langage C.

Eléments de contenu :

I- Introduction :

A priori, dans un programme, les instructions sont exécutées séquentiellement, c'est-à-dire dans l'ordre où elles apparaissent. Or, la puissance d'un programme provient essentiellement :

- De la possibilité d'effectuer des **choix**, de se comporter différemment suivant les circonstances.
- De la possibilité d'effectuer des **boucles**, c'est-à-dire de répéter plusieurs fois un ensemble donné d'instructions.

Tous les langages disposent d'instructions, nommées instructions de contrôle. Le langage C dispose d'**instructions structurées** permettant de réaliser :

- Des choix : instructions **if .. else** et **switch**.
- Des boucles : instructions **do .. while**, **while** et **for**.

II- Les structures conditionnelles :

1- L'instruction if .. else

La structure alternative en langage algorithmique	La structure alternative en C
Si (<expression logique>) **Alors** <bloc d'instructions 1> **Sinon** <bloc d'instructions 2> **FinSi**	**if (<expression>)** **<bloc d'instructions 1>** **else** **<bloc d'instructions 2>**
Interprétation	**Interprétation**
Si l'<expression logique> a la valeur logique vrai, alors le <bloc d'instructions 1> est exécuté. Si l'<expression logique> a la valeur logique faux, alors le <bloc d'instructions 2> est exécuté	Si l'<expression> fournit une valeur différente de zéro, alors le <bloc d'instructions 1> est exécuté. Si l'<expression> fournit la valeur zéro, alors le <bloc d'instructions 2> est exécuté.

La partie **<expression>** peut désigner une variable d'un type numérique, une expression fournissant un résultat numérique. La partie **<bloc d'instructions>** peut désigner un (vrai) bloc d'instructions compris entre accolades { }, ou une seule instruction terminée par un point virgule.

Exemple 1 :

```
if (a > b)
   max = a;
else
   max = b;
```

La partie **else** est facultative. On peut donc utiliser **if** de la façon suivante :

```
if ( <expression> )
    <bloc d'instructions>
```

Remarques :

- La condition doit être entre des parenthèses.
- Il est possible de définir plusieurs conditions à remplir avec les opérateurs **ET** et **OU** (&& et ||) par exemple : **if ((condition1)&&(condition2))** teste si les deux conditions sont vraies **if ((condition1)||(condition2))** exécutera les instructions si l'une ou l'autre des deux conditions est vraie.
- S'il n'y a qu'une instruction, les accolades ne sont pas nécessaires.
- S'il n'y a qu'une instruction, les accolades ne sont pas nécessaires...
- Les instructions situées dans le bloc qui suit **else** sont les instructions qui seront exécutées si la ou les conditions ne sont pas remplies.
- Comme la partie **else** est optionnelle, les expressions contenant plusieurs structures **if** et **if - else** peuvent mener à des confusions. Alors il est convenu en C qu'une partie **else** est toujours liée au dernier **if** qui ne possède pas de partie **else**.

2- L'instruction switch :

L'instruction **switch** est une instruction de **choix multiple** uniquement sur les entiers (**int**) ou les caractères (**char**). Elle permet de faire plusieurs tests de valeurs sur le contenu d'une même variable. Ce branchement conditionnel simplifie beaucoup le test de plusieurs valeurs d'une variable, car cette opération aurait été compliquée (mais possible) avec des **if** imbriqués. Sa syntaxe est la suivante:

```
switch (Variable)
  {
     case Valeur1 : Liste d'instructions break;
     case Valeur2 : Liste d'instructions break;
     case Valeurs : Liste d'instructions break;
     default      : Liste d'instructions break;
  }
```

Les parenthèses qui suivent le mot clé **switch** indiquent une expression dont la valeur est testée successivement par chacun des **case**. Lorsque l'expression testée est égale à une des valeurs suivant un **case**, la liste d'instruction qui suit celui-ci est exécutée. Le mot clé **break** indique la sortie de la structure conditionnelle. Le mot clé **default** précède la liste d'instructions qui sera exécutée si l'expression n'est jamais égale à une des valeurs.

Remarques :

- Le bloc **default** n'est pas obligatoire.
- L'instruction **switch** correspond à une cascade d'instructions **if .. else**.

Exemple : Cette instruction est commode pour fabriquer des "menus":

```
#include <stdio.h>
void main ( )
{
    char choix;
    printf("LISTE PAR GROUPE TAPER 1\n");
    printf("LISTE ALPHABETIQUE TAPER 2\n");
    printf("POUR SORTIR TAPER S\n");
    printf("\nVOTRE CHOIX: ");
    choix = getchar();
    switch(choix)
      {
        case '1':   .......;
                    .......;
                    break;
        case '2':   ......;
                    ......;
                    break;
        case 'S':   printf("\nFIN DU PROGRAMME ....");
                    break;
        default :   printf("\nERREUR DE CHOIX");
                    /* pas de break ici */
      }
```

III- Les structures itératives (les boucles) :

Les boucles sont des structures qui permettent d'exécuter plusieurs fois la même série d'instructions jusqu'à ce qu'une condition ne soit plus réalisée... On appelle parfois ces structures **instructions répétitives** ou bien **itérations**. La façon la plus commune de faire une boucle, est de créer un compteur (une variable qui s'incrémente, c'est-à-dire qui augmente de 1 à chaque tour de boucle) et de faire arrêter la boucle lorsque le compteur dépasse une certaine valeur.

1- L'instruction for :

L'instruction **for** permet d'exécuter plusieurs fois la même série d'instructions: c'est une boucle!
Dans sa syntaxe, il suffit de préciser le nom de la variable qui sert de compteur (et éventuellement sa valeur de départ, la condition sur la variable pour laquelle la boucle s'arrête (une condition qui teste si la valeur du compteur dépasse une limite) et enfin une instruction qui incrémente (ou décrémente) le compteur.

La syntaxe de cette expression est la suivante:

```
for (compteur; condition; modification du compteur)
   {
      liste d'instructions
   }
```

Exemple :

```
for (i=1; i<6; i++)
   {
        printf("%d", i);
   }
```

Cette boucle affiche 5 fois la valeur de **i** à partir de la valeur initiale 1, c'est-à-dire 1,2,3,4,5.
Elle commence à i=1, vérifie que i est bien inférieur à 6, etc... jusqu'à atteindre la valeur i=6, pour laquelle la condition ne sera plus réalisée, la boucle s'interrompra et le programme continuera son cours à l'instruction suivante à la boucle.

Remarques :

- Il faudra toujours vérifier que la boucle a bien une condition de sortie (le compteur s'incrémente correctement)
- Une instruction **printf( )***;* dans votre boucle est un bon moyen pour vérifier la valeur du compteur pas à pas en l'affichant!

- Il faut bien compter le nombre de fois pour lesquelles on veut exécuter la boucle:
  - **for(i=0;i<10;i++)** exécute 10 fois la boucle (i de 0 à 9).
  - **for(i=0;i<=10;i++)** exécute 11 fois la boucle (i de 0 à 10).
  - **for(i=1;i<10;i++)** exécute 9 fois la boucle (i de 1 à 9).
  - **for(i=1;i<=10;i++)** exécute 10 fois la boucle (i de 1 à 10).

2- L'instruction while :

La structure **while** correspond tout à fait à la structure **tant que** en algorithmique. Cette instruction représente un autre moyen d'exécuter plusieurs fois la même série d'instructions.

Sa syntaxe est la suivante :

La structure *tant que* en langage algorithmique	**La structure while en C**
Tant que (<expression logique>) **Faire** <bloc d'instructions> **Fin tant que**	**while** (<expression>) <bloc d'instructions>
Interprétation	**Interprétation**
Tant que l'<expression logique> fournit la valeur **vrai**, le <bloc d'instructions> est exécuté. Si l'<expression logique> fournit la valeur **faux**, l'exécution continue avec l'instruction qui suit **fin tant que**. Le <bloc d'instructions> est exécuté zéro ou plusieurs fois.	Tant que l'<expression> fournit une valeur différente de zéro, le <bloc d'instructions> est exécuté. Si l'<expression> fournit la valeur zéro, l'exécution continue avec l'instruction qui suit le bloc d'instructions. Le <bloc d'instructions> est exécuté zéro ou plusieurs fois.

La partie <expression> peut désigner : une variable d'un type numérique, une expression fournissant un résultat numérique. La partie <bloc d'instructions> peut désigner : un (vrai) bloc d'instructions compris entre accolades, une seule instruction terminée par un point virgule.

Exemple :

```
# include <stdio.h>
void main ( )
/* Afficher les nombres de 0 à 9 */
   {
      int I = 0;
      while (I<10)
        {
          printf("%i \n", I);
          I++;
        }
   }
```

La condition de sortie pouvant être n'importe quelle structure conditionnelle, ceci risque d'avoir des boucles infinies (boucle dont la condition est toujours vraie), c'est-à-dire qu'elle risque de provoquer un blocage du programme.

3- L'instruction do .. while

La structure **do .. while** est semblable à la structure **while**. La différence entre ces deux boucles c'est que **while** évalue la condition **avant** d'exécuter le bloc d'instructions, par contre **do .. while** évalue la condition **après** avoir exécuté le bloc d'instructions. Ainsi le bloc d'instructions est exécuté au moins une fois. La syntaxe de la structure est la suivante :

```
do
   <bloc d'instructions>
while ( <expression> );
```

Le **<bloc d'instructions>** est exécuté au moins une fois et aussi longtemps que l'**<expression>** fournit une valeur différente de zéro.

En pratique, la structure **do .. while** n'est pas si fréquente que **while**; mais dans certains cas, elle fournit une solution plus élégante. Une application typique de **do .. while** est la saisie de données qui doivent remplir une certaine condition.

Exemple :

```
void main ( )
 {
  float N;
  do
     {
       printf("Introduisez un nombre entre 1 et 10 :");
       scanf("%f", &N);
     } while (N<1 || N>10);
 }
```

chapitre 6 :

Les tableaux

Objectif :

- **Manipuler** les tableaux en langage C.

Eléments de contenu :

I- Introduction :

Les tableaux sont certainement les variables structurées les plus populaires. Ils sont disponibles dans tous les langages de programmation et servent à résoudre une multitude de problèmes. Dans une première approche, le traitement des tableaux en C ne diffère pas de celui des autres langages de programmation.

Ce sont des variables dimensionnées composées d'une suite de variables de même type. Chaque élément d'un tableau est désigné comme suit :

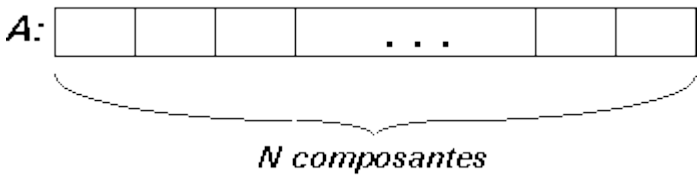
Nom_tableau[indice_de_l'élément]

L'indice de l'élément représente sa position dans le tableau.

II- Les tableaux à une dimension :

1- Définition :

Un tableau (uni-dimensionnel) A est une variable structurée formée d'un nombre entier N de variables simples du même type, qui sont appelées les ***composantes*** du tableau. Le nombre de composantes N est alors la ***dimension*** du tableau.



En faisant le rapprochement avec les mathématiques, on dit encore que *"A est un **vecteur** de dimension N"*

Syntaxe :

Type_des_éléments nom_du_tableau[taille] ;

Exemple :

int JOURS[12] ;

Réserve l'emplacement de 12 éléments de type entier. En C, la première position porte le n° 0. Pour cet exemple les indices vont de 0 à 11.

La première composante du tableau sera désigné par JOURS[0], la deuxième composante par JOURS[1], . . . , la dernière composante par JOURS[11].

Remarque : L'indice peut prendre la forme de n'importe quelle expression arithmétique de type entier (ou caractère).

Exemples :

T[n-3] , T[3*p+8]

Char c1, c2 ;

T[c1+3], T[c2-c1]

} ces notations sont correctes

2- Initialisation et réservation automatique :

- **Initialisation** :

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

Syntaxe :

Type_des_éléments nom_du_tableau[taille] = {liste_des_valeurs} ;

Exemples :

```
int A[5]   = {10, 20, 30, 40, 50};
float B[4] = {-1.05, 3.33, 87e-5, -12.3E4};
int C[10]  = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};
```

Remarque : Il faut évidemment veiller à ce que le nombre de valeurs dans la liste corresponde à la dimension du tableau. Si la liste ne contient pas assez de valeurs pour toutes les composantes, les composantes restantes sont initialisées par zéro.

- **Réservation automatique :**

Si la dimension n'est pas indiquée explicitement lors de l'initialisation, alors l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

Exemples :

int A[] = {10, 20, 30, 40, 50};

==> Réservation de **5*sizeof(int)** octets (dans notre cas: 10 octets)

float B[] = {-1.05, 3.33, 87e-5, -12.3E4};

==> Réservation de **4*sizeof(float)** octets (dans notre cas: 16 octets)

int C[] = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};

==> Réservation de **10*sizeof(int)** octets (dans notre cas: 20 octets)

Exemples :

```
short A[] = {1200, 2300, 3400, 4500, 5600};
```

A:	1200	2300	3400	4500	5600

```
short A[5] = {1200, 2300, 3400};
```

A:	1200	2300	3400	0	0

```
short A[3] = {1200, 2300, 3400, 4500, 5600};
```

A:	1200	2300	3400	☠	☠

ERREUR !

III- Tableaux à plusieurs dimensions :

1- Syntaxe :

Type_des_éléments nom_du_tableau[taille1] [taille2] … [taillen] ;

Exemple :

Float [10] [20] ;

Représente une matrice de nombres réels de 10 lignes et 20 colonnes.

2- Initialisation :

Comme pour les tableaux à une dimension, les tableaux multi dimensionnés peuvent être initialisés lors de leur déclaration.

Syntaxe :

Type_des_éléments nom_du_tableau[taille1] [taille2] … [taillen] =
{{ liste_des_valeurs1}, { liste_des_valeurs2},…, { liste_des_valeursn}} ;

Exemples :

Int M[2][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}} ;

Ou

Int M[2][4] = {1, 2, 3, 4, 5, 6, 7, 8}} ;

chapitre 7 :

Les chaînes de caractères

Objectif :

- **Manipuler** les chaînes de caractères en langage C.

Eléments de contenu :

I- Définition et déclaration :

1- Définition :

Une chaîne de caractères (appelée **string** en anglais) est une suite de caractères, c'est-à-dire un ensemble de symboles faisant partie du jeu de caractères, défini par le code ASCII.

En langage C, il n'existe pas de véritable type chaîne mais une chaîne de caractères n'est autre qu'un tableau, comportant plusieurs données de type **char**, dont le dernier élément est le caractère nul '**\0**', c'est-à-dire le premier caractère du code ASCII (dont la valeur est 0). Ce caractère est un caractère de contrôle (donc non affichable) qui permet d'indiquer une fin de chaîne de caractères. Ainsi une chaîne composée de **n** éléments sera en fait un tableau de **(n+1)** éléments de type **char**. On peut par exemple représenter la chaîne "Bonjour" de la manière suivante:

B	o	N	j	u	u	r	**\0**

2- Déclaration :

Pour définir une chaîne de caractères en langage C, il suffit de définir un tableau de caractères. Le nombre maximum de caractères que comportera la chaîne sera égal au nombre d'éléments du tableau moins un (réservé au caractère de fin de chaîne. Soit : **char Nom_du_tableau[Nombre_d_elements]**.

Le nombre d'éléments que comporte le tableau définit la taille maximale de la chaîne, on peut toutefois utiliser partiellement cet espace en insérant le caractère de fin de chaîne à l'emplacement désiré dans le tableau.

Exemples :

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

Le nom d'une chaîne est le représentant de **l'adresse du premier caractère** de la chaîne.

II- Chaîne de caractères constante :

1- Généralités :

- Les chaînes de caractères constantes **(string literals)** sont indiquées entre guillemets. La chaîne de caractères vide est alors: **""**
- Dans les chaînes de caractères, nous pouvons utiliser toutes les séquences d'échappement définies comme caractères constants.

Exemple : "**Ce \ntexte \nsera réparti sur 3 lignes.**"

Ceci donne : **Ce**
texte
sera réparti en 3 lignes.

- Le symbole **"** peut être représenté à l'intérieur d'une chaîne par la séquence d'échappement **\"**:

Exemple : "**Affichage de \"guillemets" \n**"

Ceci donne : Affichage de "guillemets"

- Le symbole **'** peut être représenté à l'intérieur d'une liste de caractères par la séquence d'échappement **\'** :

Exemple : **{'L','\'','a','s','t','u','c','e','\0'}** représente la chaîne **"L'astuce"**

- Plusieurs chaînes de caractères constantes qui sont séparées par des signes d'espacement (espaces, tabulateurs ou interlignes) dans le texte du programme seront réunies en une seule chaîne constante lors de la compilation:

Exemple : La chaîne : **"un " "deux" " trois"**
sera évalué à : **"un deux trois"**

Ainsi, il est possible de définir de très longues chaînes de caractères constantes en utilisant plusieurs lignes dans le texte du programme.

2- Initialisation d'un chaîne de caractères :

En général, les tableaux sont initialisés par l'indication de la liste des éléments du tableau entre accolades:

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

Pour le cas spécial des tableaux de caractères, nous pouvons utiliser une initialisation plus confortable en indiquant simplement une chaîne de caractère constante:

char CHAINE[ ] = "Hello";

Lors de l'initialisation par **[ ]**, l'ordinateur réserve automatiquement le nombre d'octets nécessaires pour la chaîne, c.-à-d.: le nombre de caractères + **1** (ici: 6 octets). Nous pouvons aussi indiquer explicitement le nombre d'octets à réserver, si celui-ci est supérieur ou égal à la longueur de la chaîne d'initialisation.

III- Les fonctions des entrées-sorties des chaînes de caractères :

Le langage C offre plusieurs possibilités de lecture ou d'écriture des chaînes de caractères:

- L'utilisation du format **%s** dans les fonctions **printf** et **scanf**.
- Les fonctions spécifiques de lecture (**gets**) ou d'affichage (**puts**) d'une chaîne.

1- Les fonctions printf et scanf :

Exemple :

```
Void main( )
 {
   char nom[20], prenom[20], ville[25];
   printf ("Donnez votre nom:");
   scanf ("%s",nom);
   printf ("Donnez votre prénom:");
   scanf ("%s",prenom);
   printf ("Donnez votre ville:");
   scanf ("%s",ville);
   printf ("Bonjour cher %s %s qui habitez à %s",prenom,nom,ville);
 }
```

Les fonctions **printf** et **scanf** permettent de lire ou d'afficher simultanément plusieurs informations de type quelconque.

Les délimiteurs pour la saisie des chaînes de caractères avec la fonction **scanf** sont : l'espace, la tabulation ou la fin de ligne.

Remarque :

Dans les appels de la fonction **scanf**, les identificateurs des tableaux de caractères (nom, prenom, ville), ne doivent pas être précédés de l'opérateur **&**, puisqu'ils représentent déjà des adresses.

2- Les fonctions gets et puts :

Exemple :

```
Void main( )
  {
    char nom[20], prenom[20], ville[25];
    printf ("Donnez votre nom:");
    gets(nom);
    printf ("Donnez votre prénom:");
    gets(prenom);
    printf ("Donnez votre ville:");
    gets(ville);
    printf ("Bonjour cher");
    puts (nom);
    puts (prenom);
    printf ("qui habitez à ");
    puts (ville);
  }
```

Les fonctions **gets** et **puts** ne traitent qu'une seule chaîne à la fois. Avec la fonction **gets**, seule la fin de ligne sert de délimiteur.

IV- Fonctions de manipulation des chaînes de caractères :

Les fonction de manipulation des chaînes de caractères se trouvent dans la bibliothèque <**string.h>**.

1- Les fonctions de concaténation de chaînes :

La concaténation n'est rien d'autre que la juxtaposition de deux chaînes afin d'en former une seule. Elle est réalisée en C à l'aide de l'une des deux fonctions suivantes : **strcat** et **strncat**.

- **La fonction strcat :**

Syntaxe : strcat (but, source)

Cette fonction recopie la seconde chaîne (source) à la suite de la première chaîne (but), après avoir effacé le caractère de fin.

Exemple :

```
Void main ( )
   {
      char ch1[20] = "bonjour";
      char ch[50] = " monsieur";
      printf ("avant : %s \n", ch1);
      strcat (ch1, ch2);
      printf ("après : %s \n", ch1);
   }
```

Ce programme donne : avant : bonjour
Après : bonjour monsieur

- **La fonction strncat :**

Syntaxe : strncat (but, source, lgmax)

Cette fonction travaille de façon semblable à **strcat** en offrant en outre un contrôle sur le nombre de caractères qui seront concaténés à la chaîne d'arrivée (but).

Exemple :

```
Void main ( )
   {
      char ch1[20] = "bonjour";
      char ch[50] = " monsieur";
      printf ("avant : %s \n", ch1);
      strncat (ch1, ch2, 6);
      printf ("après : %s \n", ch1);
   }
```

Ce programme donne : avant : bonjour
Après : bonjour monsi

Remarque : On peut savoir la longueur d'une chaîne de caractères en utilisant la fonction : **strlen (ch)**.

2- Les fonctions de comparaison de chaînes :

Il est possible de comparer deux chaînes de caractères en utilisant l'ordre des caractères défini par le code ASCII.

- **La fonction strcmp :**

Syntaxe : strcmp (ch1, ch2)

Cette fonction compare deux chaînes et fournit une valeur entière définie comme étant :

  - Positive si ch1 > ch2 (**Exemple :** strcmp ("bonjour", "monsieur")).
  - Nulle si ch1 = ch2
  - Négative si ch1 < ch2 (**Exemple :** strcmp ("salle2", "salle10")).

- **La fonction strncmp :**

Syntaxe : strncmp (ch1, ch2, lgmax)

Cette fonction travaille comme **strcmp** mais elle limite la comparaison au nombre maximum de caractères indiqués par l'entier **lgmax**.

- **Les fonctions stricmp et strnicmp :**

Syntaxe : stricmp (ch1, ch2)
strnicmp (ch1, ch2, lgmax)

Ces deux fonctions travaillent respectivement comme **strcmp** et **strncmp**, mais sans tenir compte de la différence ente majuscules et minuscules (pour les caractères alphabétiques).

3- Les fonctions de copie de chaînes :

- **La fonction strcpy :**

Syntaxe : strcpy (destin, source)

Cette fonction recopie la chaîne **source** dans la chaîne **destin**. Il est nécessaire que la chaîne **destin** soit de taille suffisante pour accueillir la chaîne **source**.

- **La fonction strncpy :**

Syntaxe : strncpy (destin, source, lgmax)

Cette fonction procède de la même manière que **strcpy** en limitant la recopie au nombre de caractères précisé par **lgmax**.

chapitre 8 :

Les fonctions

Objectif :

- **Expliquer** les concepts relatifs aux fonctions en langage C.

Eléments de contenu :

I- Notion de fonction :

Comme tous les langages de programmation, C permet de découper un programme en plusieurs parties nommées souvent **modules**. Cette programmation dite **modulaire** se justifie pour plusieurs raisons :

- Un programme écrit en un seul morceau devient difficile à comprendre dès qu'il dépasse deux pages de texte. Une écriture modulaire permet de le scinder en plusieurs parties et de regrouper dans le **programme principal** les différents enchaînements entre les modules. Chacun de ces module, peut à son tour être décomposé.
- La programmation modulaire permet d'éviter les répétitions dans un programme.

Dans beaucoup de langages de programmation, on trouve deux sortes de modules :

- Les **fonctions**, assez proches de la notion mathématique correspondante. Il est à noter qu'une fonction dispose d'un ensemble d'**arguments** qui correspondent à des informations qui lui sont transmises et elle fournit un **résultat unique**, désigné par le nom même de la fonction.
- Les **procédures** qui élargissent le notion de fonction. Une procédure possède un ensemble d'**arguments**, parmi lesquels se trouvent les résultats qui seront donnés et les informations qui sont transmises. Une procédure peut alors retourner zéro, un ou plusieurs résultats.

En langage C, il 'existe qu'une seule sorte de module : **la fonction**. On l'utilise à la manière d'une fonction mathématique en faisant suivre son nom d'une liste d'**arguments**, le tout possède **une valeur** et peut apparaître dans une **expression** quelconque.

Plusieurs fonctions peuvent partager des informations, autrement que par passage d'arguments (ou paramètres), on parle alors de **variables globales**.

Exemple de fonction :

```
Void main ( )
   {
      Temps ( );
   }

void Temps ( )
   {
      printf ("Il fait beau");
   }
```

L'exécution de ce programme donne l'affichage suivant : **Il fait beau**.
La fonction **Temps** est sans paramètre et ne retourne pas de **valeur** c'est pourquoi on utilise le type **void**.

II- Déclaration et appel :

1- Déclaration :

Avant d'être utilisée, une fonction doit être définie car pour l'appeler dans le corps du programme il faut que le compilateur la connaisse, c'est-à-dire qu'il connaît son nom, ses arguments et les instructions qu'elle contient. La définition d'une fonction s'appelle "**déclaration**". La déclaration d'une fonction se fait selon la syntaxe suivante:

```
type_de_donnee Nom_De_La_Fonction (type1 argument1, type2 argument2, ...)
   {
      liste d'instructions
   }
```

Remarques :

- **Type_de_donnee** représente le type de valeur que la fonction est sensée retourner (char, int, float,...)

- Si la fonction ne renvoie aucune valeur, on la fait alors précéder du mot-clé **void**.
- Si aucun type de donnée n'est précisé, le type **int** est pris par défaut.
- Le nom de la fonction suit les mêmes règles que les noms de variables :
    - Le nom doit commencer par une lettre.
    - Un nom de fonction peut comporter des lettres, des chiffres et les caractères _ et & (les espaces ne sont pas autorisés!).
    - Le nom de la fonction, comme celui des variables est sensible à la casse (différenciation entre les minuscules et majuscules).
- Les arguments sont facultatifs, mais s'il n'y a pas d'arguments, les parenthèses doivent rester présentes.
- Il ne faut pas oublier de refermer les accolades.
- Le nombre d'accolades ouvertes (fonction, boucles et autres structures) doit être égal au nombre d'accolades fermées!

Une fois cette étape franchie, votre fonction ne s'exécutera pas tant que l'on ne fait pas appel à elle quelque part dans le programme!

2- Appel de fonction :

Pour exécuter une fonction, il suffit de faire appel à elle en écrivant son nom (une fois de plus en respectant la casse) suivi d'une parenthèse ouverte (éventuellement des arguments) puis d'une parenthèse fermée: **Nom_De_La_Fonction( );**

Remarques :

- Le point virgule signifie la fin d'une instruction et permet au navigateur de distinguer les différents blocs d'instructions.
- Si jamais vous avez défini des arguments dans la déclaration de la fonction, il faudra veiller à les inclure lors de l'appel de la fonction (le même nombre d'arguments séparés par des virgules!) **Nom_De_La_Fonction(argument1, argument2);**

Exemple :

```
Void main ( )
   {
      int a = 10, b = 20;
      ecris (a);
      ecris (b);
      ecris (a + b);
   }

void ecris (int n)
   {
      printf("valeur : %d\n",n);
   }
```

La fonction **ecris** a un paramètre mais ne retourne pas de valeur.

Ce programme donne : valeur : 10
valeur : 20
valeur : 30

III- Notion de prototype:

Le prototype d'une fonction est une description d'une fonction qui est définie plus loin dans le programme. On place donc le prototype en début de programme (avant la fonction principale **main( )**).

Cette description permet au compilateur de **vérifier** la validité de la fonction à chaque fois qu'il la rencontre dans le programme, en lui indiquant:

- Le type de valeur renvoyée par la fonction.
- Le nom de la fonction.
- Les types d'arguments.

Contrairement à la **définition** de la fonction, le prototype n'est pas suivi du corps de la fonction (contenant les instructions à exécuter), et ne comprend pas le nom des paramètres (seulement leur type). Un prototype de fonction ressemble donc à ceci:

Type_de_donnee_renvoyee Nom_Fonction (type_arg1,type_arg2, ...);

Le prototype est une instruction, il est donc suivi d'un point-virgule!

Exemples :

- void Affiche_car (char);
- int Somme (int, int);

IV- Fonction retournant un résultat :

1- Exemple :

```
# include <stdio.h>
int somme (int, int);          //prototype de la fonction somme
void main ( )
   {
      int a, b, c, d, x, y;
      a = 5;
      b = 4;
      c = 3;
      d = 2;
      x = somme (a, b);
      printf ("x = %d\n", x);
      y = 3 * somme (c, d);
      printf ("y = %d\n", y);
   }
int somme (int u, int v)
   {
      int s;
      s = u + v;
      return (s);
   }
```

La fonction somme possède deux arguments et retourne un résultat qui est la somme des deux entiers qui lui sont fournis en argument. Ce retour de résultat est exprimé par l'instruction return.

2- L'instruction Return :

D'une manière générale :

- L'instruction **return** peut mentionner n'importe quelle expression. Ainsi, on aurait pu écrire la fonction précédente comme suit :

```
int somme (int u, int v)
   {
      return (u + v);
   }
```

- L'instruction **return** peut apparaître à plusieurs reprises dans une fonction. Par exemple :

```
int absolue (int a)
   {
      if (a > 0)      return (a);
      else return (-a);
   }
```

V- Transmission par valeur :

1- Exemple :

```
void main ( )
   {
      int n = 10, p = 20;
      printf ("Avant echange : %d et %d\n", n, p);
      echange (n, p)
      printf ("Après echange : %d et %d\n", n, p);
   }
```

```
void echange (int a, int b)
   {
      int c;
      printf ("Début echange : %d et %d\n", n, p);
      c = a;
      a = b;
      b = c;
      printf ("Fin echange : %d et %d\n", n, p);
   }
```

2- Interprétation :

Ce programme avec l'appel de fonction echange donnent comme affichages :

Avant echange : 10 et 20
Début echange : 10 et 20
Fin echange : 10 et 20
Après echange : 10 et 20

La fonction **echange** reçoit deux valeurs correspondant à deux paramètres **a** et **b**. Elle effectue l'échange de ces deux valeurs, mais en revenant au programme principal aucune trace de cet échange n'est retrouvée au niveau des **paramètres effectifs n** et **p**. La transmission a été effectuée par **copie de valeur locale** et ceci explique bien le résultat obtenu. Ce mode de transmission semble donc interdire a priori à une fonction de produire une ou plusieurs valeurs en retour.

Or, il ne faut pas oublier qu'en tous les modules doivent être écrits sous forme de fonction. Donc, ce simple problème d'échange des valeurs de deux variables doit pouvoir se résoudre à l'aide d'une fonction. Ce problème peut être résolu de deux manières :

- Utiliser des variables globales.
- Transmettre en argument l'adresse de la variable et non la valeur de la variable. La fonction pourra alors agir directement sur le contenu de cette adresse.

VI- Notion de variables globales :

En langage C, comme d'autres langages de programmation, plusieurs fonctions peuvent partager des variables communes qu'on qualifie de **variables globales**.

1- Exemple :

```
int i;
void main ( )
   {
      for (i = 1; i <= 5; i ++)
         Temps ( );
   }

void Temps ( )
   {
      printf ("Il fait beau %d fois", i);
   }
```

Ce programme donne les affichages suivants :

```
Il fait beau 1 fois
Il fait beau 2 fois
Il fait beau 3 fois
Il fait beau 4 fois
Il fait beau 5 fois
```

La variable **i** a été déclarée en dehors de la fonction principale **main**. Elle est alors connue de toutes les fonctions qui seront compilées par la suite au sein du même code source. Ainsi, le programme principal affecte à **i** des valeurs qui se trouvent reconnues au niveau de la fonction **Temps**.

2- Propriétés des variables globales :

Les variables globales ainsi définies ne sont connues du compilateur que dans la partie du code source suivant leur déclaration. On dit que leur **portée** (ou encore leur espace de validité) est limitée à la partie du code source qui suit leur déclaration. D'une manière générale, les variables globales existent pendant toute l'exécution du programme dans lequel elles apparaissent. Leurs emplacements en mémoire sont parfaitement définis lors de l'édition des liens.

Remarque :

Ces variables sont initialisées à zéro avant le début de l'exécution du programme, sauf si on leur attribue explicitement une valeur initiale lors de leur déclaration.

Exemple :

```
int a,b;
Void main ( )
   {
      …
   }
int n;
float x;
fct1 ( … )
   {
      …
   }
fct2 ( … )
   {
      …
   }
```

Les variables a et b sont des variables globales reconnues au niveau de la fonction principale main et des fonctions fct1 et fct2 alors que les variables n et x ne sont reconnues qu'au niveau des fonctions fct1 et fct2.

Il est possible ainsi de déclarer des variables globales à n'importe quel endroit du code source, mais pour des raisons de lisibilité du programme, on préférera regrouper les déclarations de toutes les variables globales au début du code source.

VII- Les tableaux transmis comme argument de fonction :

1- Exemple 1 : Tableau à nombre fixe d'éléments :

Soit la fonction **remplir** permettant de remplir un tableau de 10 entiers :

```
void remplir (int t[10])
  {
     int i;
     for (i = 0; i < 10; i ++)
        {
           printf ("T[%d] = ", i+1);
           scanf ("%d", &t[i]);
        }
   }
```

L'appel à cette fonction se fait de la manière suivante : int t1[10]; remplir (t1);

2- Exemple 2 : Tableau à nombre variable d'éléments :

La fonction som fait la somme es élément d'un tableau d'entiers de taille quelconque :

```
int som (int t[ ], in nb)
  {
     int s = 0, i;
     for (i = 0; i < nb; i ++)
        s = s + t[i];
     return (s);
  }
```

Voici quelques exemples d'appel de cette fonction :

```
Void main ( )
  {
     int t1[30], t2[15], t3[10];
     int s1, s2, s3;
     …
     s1= som (t1, 30);
     s2 = som (t2, 15; + som (t3,10);
     …
   }
```

3- Exemple 3 : Tableau à plusieurs dimensions :

Soit la fonction **remplir** permettant de remplir une matrice 50 élément (10x5) :

```
void remplir (int m[10][5])
  {
     int i, j;
     for (i = 0; i < 10; i ++)
        for (j = 0; j < 10; j ++)
           {
              printf ("M[%d][%d] = ", i+1, j+1);
              scanf ("%d", &m[i][j]);
           }
  }
```

Remarque :

On pourrait aussi écrire l'en-tête de cette fonction comme suit : **remplir (int m[ ][5])** mais non : **remplir (int m[ ][ ])**.