

Programmation en C - Sommaire

PROGRAMMATION EN C - SOMMAIRE	1
CHAPITRE 0 : INTRODUCTION :	5
I) NOTATIONS ET SYMBOLES :	5
II) C, LA NAISSANCE D'UN LANGAGE DE PROGRAMMATION PORTABLE ... :	6
1) <i>Historique</i> :	6
2) <i>K&R-C</i> :	6
3) <i>ANSI-C</i> :	6
4) <i>C++</i> :	6
5) <i>Avantages</i> :	6
6) <i>Désavantages</i> :	7
CHAPITRE 1 : L'ENVIRONNEMENT BORLAND C :	10
I) DESCRIPTION DE L'ENVIRONNEMENT :	10
1) <i>Les menus</i> :	10
2) <i>Les nouveautés</i> :	10
II) SELECTIONNER LE COMPILATEUR ANSI-C :	11
III) LES BIBLIOTHEQUES DE FONCTIONS PREDEFINIES :	11
1) <i>Utilisation des bibliothèques de fonctions</i> :	11
CHAPITRE 2 : NOTIONS DE BASE :	13
I) HELLO C ! :	13
II) LES COMPOSANTES D'UN PROGRAMME EN C :	13
1) <i>Les fonctions</i> :	13
2) <i>La fonction main</i> :	14
3) <i>Les variables</i> :	16
4) <i>Les identificateurs</i> :	16
5) <i>Les commentaires</i> :	16
III) DISCUSSION DE L'EXEMPLE 'HELLO_WORLD' :	17
IV) EXERCICES D'APPLICATION :	18
V) SOLUTIONS DES EXERCICES DU CHAPITRE 2 : NOTIONS DE BASE :	20
CHAPITRE 3 : TYPES DE BASE, OPÉRATEURS ET EXPRESSIONS :	23
I) LES TYPES SIMPLES :	23
1) <i>Les types entiers</i> :	24
2) <i>Les types rationnels</i> :	25
II) LA DECLARATION DES VARIABLES SIMPLES :	25
1) <i>Les constantes numériques</i> :	26
2) <i>Initialisation des variables</i> :	28
III) LES OPERATEURS STANDARD :	29
1) <i>Exemples d'affectations</i> :	29
2) <i>Les opérateurs connus</i> :	29
3) <i>Les opérateurs particuliers de C</i> :	30
IV) LES EXPRESSIONS ET LES INSTRUCTIONS :	31
1) <i>Expressions</i> :	31
2) <i>Instructions</i> :	32
3) <i>Évaluation et résultats</i> :	32
V) LES PRIORITES DES OPERATEURS :	32
1) <i>Priorité d'un opérateur</i> :	32
2) <i>Classes de priorités</i> :	32
3) <i>Evaluation d'opérateurs de la même classe</i> :	33
VI) LES FONCTIONS ARITHMETIQUES STANDARD :	34
1) <i>Type des données</i> :	34
VII) LES CONVERSIONS DE TYPE :	34
1) <i>Les conversions de type automatiques</i> :	34
2) <i>Les conversions de type forcées (casting)</i> :	36
VIII) EXERCICES D'APPLICATION :	37
IX) SOLUTIONS DES EXERCICES DU CHAPITRE 3 : TYPES DE BASE, OPÉRATEURS ET EXPRESSIONS :	40

CHAPITRE 4 : LIRE ET ÉCRIRE DES DONNÉES :	44
I) ÉCRITURE FORMATEE DE DONNEES :	44
1) <i>printf()</i> :	44
II) LECTURE FORMATEE DE DONNEES :	46
1) <i>scanf()</i> :	46
III) ÉCRITURE D'UN CARACTERE :	48
1) <i>Type de l'argument</i> :	49
IV) LECTURE D'UN CARACTERE :	49
1) <i>Type du résultat</i> :	49
V) EXERCICES D'APPLICATION :	50
VI) SOLUTIONS DES EXERCICES DU CHAPITRE 4 : LIRE ET ÉCRIRE DES DONNÉES :	53
CHAPITRE 5 : LA STRUCTURE ALTERNATIVE :	59
I) IF – ELSE :	59
II) IF SANS ELSE :	60
III) IF - ELSE IF - ... – ELSE :	61
IV) LES OPERATEURS CONDITIONNELS :	62
V) EXERCICES D'APPLICATION :	64
VI) SOLUTIONS DES EXERCICES DU CHAPITRE 5 : LA STRUCTURE ALTERNATIVE :	66
CHAPITRE 6 : LA STRUCTURE REPETITIVE :	71
I) WHILE :	71
II) DO – WHILE :	72
III) FOR :	73
IV) CHOIX DE LA STRUCTURE REPETITIVE :	75
V) EXERCICES D'APPLICATION :	76
VI) SOLUTIONS DES EXERCICES DU CHAPITRE 6 : LA STRUCTURE REPETITIVE :	79
CHAPITRE 7 :LES TABLEAUX	88
I) LES TABLEAUX A UNE DIMENSION :	88
1) <i>Déclaration et mémorisation</i> :	88
2) <i>Initialisation et réservation automatique</i> :	89
3) <i>Accès aux composantes</i> :	90
4) <i>Affichage et affectation</i> :	90
II) LES TABLEAUX A DEUX DIMENSIONS :	92
1) <i>Déclaration et mémorisation</i> :	93
2) <i>Initialisation et réservation automatique</i> :	94
3) <i>Accès aux composantes</i> :	95
4) <i>Affichage et affectation</i> :	96
III) EXERCICES D'APPLICATION :	97
1) <i>Tableaux à une dimension – Vecteurs</i> :	98
2) <i>Tableaux à deux dimensions – Matrices</i> :	102
IV) SOLUTIONS DES EXERCICES DU CHAPITRE 7 : LES TABLEAUX :	105
1) <i>Tableaux à une dimension – Vecteurs</i> :	110
2) <i>Tableaux à deux dimensions – Matrices</i> :	120
CHAPITRE 8 : LES CHAÎNES DE CARACTÈRES :	132
I) DECLARATION ET MEMORISATION :	132
1) <i>Déclaration</i> :	132
2) <i>Mémorisation</i> :	132
II) LES CHAINES DE CARACTERES CONSTANTES :	132
III) INITIALISATION DE CHAINES DE CARACTERES :	133
IV) ACCES AUX ELEMENTS D'UNE CHAINE :	134
V) PRECEDENCE ALPHABETIQUE ET LEXICOGRAPHIQUE :	134
1) <i>Précédence alphabétique des caractères</i> :	134
2) <i>Relation de précédence</i> :	134
3) <i>Précédence lexicographique des chaînes de caractères</i> :	134
4) <i>Conversions et tests</i> :	135
VI) TRAVAILLER AVEC DES CHAINES DE CARACTERES :	135
1) <i>Les fonctions de <stdio.h></i> :	135
2) <i>Les fonctions de <string></i> :	137
3) <i>Les fonctions de <stdlib></i> :	138
4) <i>Les fonctions de <ctype></i> :	139

VII) TABLEAUX DE CHAINES DE CARACTERES :	139
1) Déclaration, initialisation et mémorisation :	140
2) Accès aux différentes composantes :	140
VIII) EXERCICES D'APPLICATION :	142
IX) SOLUTIONS DES EXERCICES DU CHAPITRE 8 : LES CHAÎNES DE CARACTÈRES	148
CHAPITRE 9 : LES POINTEURS :	164
I) ADRESSAGE DE VARIABLES :	164
1) Adressage direct :	164
2) Adressage indirect :	164
II) LES POINTEURS :	165
1) Les opérateurs de base :	165
2) Les opérations élémentaires sur pointeurs :	167
III) POINTEURS ET TABLEAUX :	168
1) Adressage des composantes d'un tableau :	168
2) Arithmétique des pointeurs :	171
3) Pointeurs et chaînes de caractères :	173
4) Pointeurs et tableaux à deux dimensions :	175
IV) TABLEAUX DE POINTEURS :	177
1) Déclaration :	178
2) Initialisation :	178
V) ALLOCATION DYNAMIQUE DE MEMOIRE :	179
1) Déclaration statique de données :	179
2) Allocation dynamique :	180
3) La fonction malloc et l'opérateur sizeof :	180
4) La fonction free :	182
VI) EXERCICES D'APPLICATION :	183
VII) SOLUTIONS DES EXERCICES DU CHAPITRE 9 : LES POINTEURS :	188
CHAPITRE 10 : LES FONCTIONS :	207
I) MODULARISATION DE PROGRAMMES :	207
1) La modularité et ses avantages :	207
2) Exemples de modularisation en C :	208
II) LA NOTION DE BLOCS ET LA PORTEE DES IDENTIFICATEURS :	212
1) Variables locales :	213
2) Variables globales :	214
III) DECLARATION ET DEFINITION DE FONCTIONS :	215
1) Définition d'une fonction :	216
2) Déclaration d'une fonction :	217
3) Discussion d'un exemple :	218
IV) RENVoyer UN RESULTAT :	221
1) La commande return :	221
2) void :	222
3) main :	222
4) exit :	223
5) Ignorer le résultat :	223
V) PARAMETRES D'UNE FONCTION :	223
1) Généralités :	223
2) Passage des paramètres par valeur :	224
3) Passage de l'adresse d'une variable :	225
4) Passage de l'adresse d'un tableau à une dimension :	226
5) Passage de l'adresse d'un tableau à deux dimensions :	227
6) Les modules en lang. algorithmique, en Pascal et en C :	229
VI) DISCUSSION DE DEUX PROBLEMES :	232
1) "fonction" ou "procédure" ? :	232
2) Pointeur ou indice numérique ? :	234
VII) EXERCICES D'APPLICATION :	236
1) Passage des paramètres :	236
2) Types simples :	238
3) Tableaux à une dimension :	239
4) Tris de tableaux :	240
5) Chaînes de caractères :	242
6) Tableaux à deux dimensions :	243
VIII) SOLUTIONS DES EXERCICES DU CHAPITRE 10 : LES FONCTIONS :	245

1) Passage des paramètres :	246
2) Types simples :	249
3) Tableaux à une dimension :	254
4) Tris de tableaux :	256
5) Chaînes de caractères :	265
6) Tableaux à deux dimensions :	271
CHAPITRE 11 : LES FICHIERS SEQUENTIELS :	278
I) DEFINITIONS ET PROPRIETES :	278
1) Fichier :	278
2) Fichier séquentiel :	278
3) Fichiers standards :	278
II) LA MEMOIRE TAMPON :	279
III) ACCES AUX FICHIERS SEQUENTIELS :	279
1) Le type FILE* :	279
2) Exemple : Créer et afficher un fichier séquentiel :	280
IV) OUVRIR ET FERMER DES FICHIERS SEQUENTIELS :	281
1) Ouvrir un fichier séquentiel :	282
2) Fermer un fichier séquentiel :	283
3) Exemples : Ouvrir et fermer des fichiers en pratique :	283
V) LIRE ET ECRIRE DANS DES FICHIERS SEQUENTIELS :	285
1) Traitement par enregistrements :	285
2) Traitement par caractères :	287
3) Détection de la fin d'un fichier séquentiel :	287
VI) RESUME SUR LES FICHIERS :	288
VII) MISE A JOUR D'UN FICHIER SEQUENTIEL EN C :	288
1) Ajouter un enregistrement à un fichier :	289
2) Supprimer un enregistrement dans un fichier :	291
3) Modifier un enregistrement dans un fichier :	292
VIII) EXERCICES D'APPLICATION :	294
IX) SOLUTIONS DES EXERCICES DU CHAPITRE 11 : LES FICHIERS SEQUENTIELS :	297
CHAPITRE 12 : ANNEXES :	320
ANNEXE A : LES SEQUENCES D'ECHAPPEMENT :	321
ANNEXE B : LES PRIORITES DES OPERATEURS :	322
ANNEXE C : LES PROTOTYPES DES FONCTIONS TRAITEES :	323
I) ENTREE ET SORTIE DE DONNEES : <STDIO.H>	323
II) LIRE UN CARACTERE : <CONIO.H>	325
III) TRAITEMENT DE CHAINES DE CARACTERES : <STRING.H>	325
IV) FONCTIONS D'AIDE GENERALES : <STDLIB.H>	325
1) Conversion de chaînes de caractères en nombres :	325
2) Gestion de mémoire :	326
3) Abandon d'un programme :	326
4) Conversion de nombres en chaînes de caractères :	326
V) CLASSIFICATION ET CONVERSION DE CARACTERES : <CTYPE.H>	327
1) Fonctions de classification et de conversion de caractères :	327
VI) FONCTIONS ARITHMETIQUES : <MATH.H>	327
1) Fonctions arithmétiques :	327
ANNEXE D : BIBLIOGRAPHIE :	329

Chapitre 0 : INTRODUCTION :

I) NOTATIONS ET SYMBOLES :

Dans les textes explicatifs de ce manuel, les parties de programmes en langage algorithmique sont mis en évidence par la police d'écriture :

`Courier` (mots réservés soulignés)

Les parties de programmes en C sont écrites à l'aide de la police d'écriture :

Courier (gras)

Les modules (fonctions, procédures, programmes principaux) sont marqués par un trait vertical dans la marge gauche.

Explication des symboles utilisés dans le texte :



Conseil !



Mauvaise solution !



Bonne solution !



Attention ! Piège dangereux !



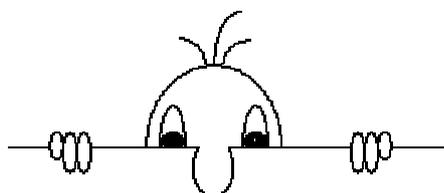
Attention ! Piège fréquent !
ou Remarque importante !



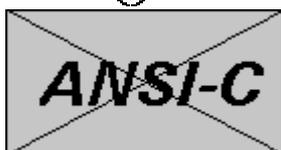
Solution incorrecte !
Peut mener à des erreurs.



Solution correcte !



Remarque avancée.
Pour ceux qui veulent
en savoir plus.



Fonction ou opération non portable
selon le standard ANSI-C.

II) C, la naissance d'un langage de programmation portable ... :

1) Historique :

Dans les dernières années, aucun langage de programmation n'a pu se vanter d'une croissance en popularité comparable à celle de 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.

2) K&R-C :

En 1978, le duo Brian W. Kernighan / Dennis M. Ritchie a publié la définition classique du langage C (connue sous le nom de *standard K&R-C*) dans un livre intitulé 'The C Programming Language'.

3) ANSI-C :

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*. La seconde édition du livre 'The C Programming Language', parue en 1988, respecte tout à fait le standard ANSI-C et elle est devenue par la suite, la 'bible' des programmeurs en C.

4) 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, efficacité) 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 (AT&T-C++ Version 2.1 [1990] et AT&T-C++ Version 3.0 [1992]).

5) Avantages :

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

a) Universel :

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).

b) Compact :

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

c) Moderne :

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).

d) 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 (p.ex : la gestion de la mémoire).

e) Rapide :

comme C permet d'utiliser des expressions et des opérateurs qui sont très proches du langage machine, il est possible de développer des programmes efficaces et rapides.

f) Indépendant de la machine :

bien que C soit un langage près de la machine, il peut être utilisé sur n'importe quel système en possession d'un compilateur C. Au début C était surtout le langage des systèmes travaillant sous UNIX, aujourd'hui C est devenu le langage de programmation standard dans le domaine des micro-ordinateurs.

g) 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.

h) Extensible :

C ne se compose pas seulement des fonctions standard; le langage est animé par des bibliothèques de fonctions privées ou livrées par de nombreuses maisons de développement.

6) Désavantages :

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

a) Efficience et compréhensibilité :

En C, nous avons la possibilité d'utiliser des expressions compactes et efficaces. 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 1 :

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(" ");
}
```

Exemple 2 :

La fonction **copietab()** copie les éléments d'une chaîne de caractères T[] dans une autre chaîne de caractères S[]. Voici d'abord la version 'simili-Pascal' :

```
void copietab(char S[], char T[])
{
    int I;
    I=0;
```

```

while (T[I] != '\0')
{
    S[I] = T[I];
    I = I+1;
}
}

```

Cette définition de la fonction est valable en C, mais en pratique elle ne serait jamais programmée ainsi. En utilisant les possibilités de C, un programmeur expérimenté préfère la solution suivante :

```

void copietab(char *S, char *T)
{
    while (*S++ = *T++);
}

```

La deuxième formulation de cette fonction est élégante, compacte, efficace et la traduction en langage machine fournit un code très rapide...; mais bien que cette manière de résoudre les problèmes soit le cas normal en C, il n'est pas si évident de suivre le raisonnement.

Conclusions :

Bien entendu, dans les deux exemples ci-dessus, les formulations 'courtes' représentent le bon style dans C et sont de loin préférables aux deux autres. Nous constatons donc que :

- la programmation efficace en C nécessite beaucoup d'expérience et n'est pas facilement accessible à des débutants.
- sans commentaires ou explications, les programmes peuvent devenir incompréhensibles, donc inutilisables.

b) Portabilité et bibliothèques de fonctions :

Les 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 (p.ex : 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.

c) Discipline de programmation :

Les dangers de C :

Nous voici arrivés à un point crucial : C est un langage près de la machine, donc dangereux et bien que C soit un langage de programmation structuré, il ne nous force pas à adopter un certain style de programmation (comme p.ex. Pascal). 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.

Remarque :

Au fil de l'introduction du langage C, ce manuel contiendra quelques recommandations au sujet de l'utilisation des différents moyens de programmation. Il est impossible de donner des règles universelles à ce sujet, mais les deux conseils suivants sont valables pour tous les langages de programmation :

Si, après avoir lu uniquement les commentaires d'un programme, vous n'en comprenez pas le fonctionnement, alors jetez le tout ! (règle non écrite dans la maison IBM)

Pratiquez la programmation défensive ...

Chapitre 1 : L'ENVIRONNEMENT BORLAND C++ :

Borland C++ Version 3.1

Pour le travail pratique en C, nous utiliserons l'environnement Borland C++ (Version 3.1). Ce programme nous offre une surface de programmation confortable et rapide. L'environnement Borland C++ ressemble en beaucoup de points à l'environnement Turbo Pascal qui vous est déjà familier.

Borland C++ est une implémentation complète du standard C++ AT&T (Version 2.1). Le compilateur Borland C++ est capable de produire du code C 'pur' selon la définition Kernighan & Ritchie ou selon le standard ANSI-C. D'autre part, Borland C++ nous offre toute une série de bibliothèques, qui (parfois aux dépens de la portabilité) nous permettent d'exploiter les capacités du PC.

I) Description de l'environnement :

L'environnement Turbo Pascal vous étant déjà connu, ce chapitre se limite à une brève description des menus et des nouveautés de Borland C++.

1) Les menus :

- FILE gestion des fichiers, retour au DOS.
- EDIT commandes d'édition du texte.
- SEARCH recherche d'un texte, d'une déclaration de fonction, de la position d'une erreur dans le programme.
- RUN exécution d'un programme en entier ou par parties.
- COMPILE traduction et/ou enchaînement (link) des programmes.
- DEBUG détection d'erreurs en inspectant les données, en insérant des points d'observation (watch) ou en insérant des points d'arrêt (breakpoints).
- PROJECT gestion de projets.
- OPTIONS changement et sauvetage des réglages par défaut :
 - * des menus
 - * des options du compilateur
 - * de l'environnement
- WINDOW visualisation et disposition des différentes fenêtres (Message, Output, Watch, User, Project) sur l'écran
- HELP système d'aide.

2) Les nouveautés :

a) L'interface utilisateur :

L'environnement Borland C++ nous permet l'utilisation confortable de plusieurs fenêtres sur l'écran. L'interaction des données des différentes fenêtres (*Message, Output, Watch, User Screen*) a rendu la recherche d'erreurs très efficace.

b) La gestion de projets multi-fichiers :

En pratique, les programmes sont souvent subdivisés en plusieurs sous-programmes ou modules, qui peuvent être répartis dans plusieurs fichiers. L'environnement Borland C++ nous offre un gestionnaire '*Project Manager*' qui détecte automatiquement les fichiers qui doivent être recompilés et enchaînés après une modification. Dans nos applications, nous allons uniquement utiliser des fonctions définies dans le même fichier que notre programme principal ou prédéfinies dans une bibliothèque standard ; ainsi, nous n'aurons pas besoin de l'option 'Project'.

c) Le système d'aide :

Borland C++ est accompagné d'un programme d'aide puissant qui est appelé directement à partir de l'éditeur. Le système d'aide peut donner des informations ponctuelles sur un sujet sélectionné ou nous laisser choisir un sujet à partir d'une liste alphabétique :

F1 Aide sur l'environnement Borland C++

Shift-F1 Index alphabétique des mots-clefs

Ctrl-F1 Aide sur le mot actif (à la position du curseur)

II) Sélectionner le compilateur ANSI-C :

Comme nous utilisons l'environnement Borland C++ pour compiler des programmes correspondant au standard ANSI-C, nous devons d'abord changer le mode de compilation par défaut :

- Choisir le menu 'compilateur' par : Alt-Options|Compiler

- Choisir le modèle ANSI-C par : Source|ANSI|OK

III) Les bibliothèques de fonctions prédéfinies :

1) Utilisation des bibliothèques de fonctions :

La pratique en C exige l'utilisation de bibliothèques de fonctions. Ces bibliothèques sont disponibles dans leur forme précompilée (extension : **.LIB**). Pour pouvoir les utiliser, il faut inclure des fichiers en-tête (*header files* - extension **.H**) dans nos programmes. Ces fichiers contiennent des '*prototypes*' des fonctions définies dans les bibliothèques et créent un lien entre les fonctions précompilées et nos programmes.

a) #include :

L'instruction **#include** insère les fichiers en-tête indiqués comme arguments dans le texte du programme au moment de la compilation.

b) Identification des fichiers :

Lors de la programmation en Borland C, nous travaillons donc avec différents types de fichiers qui sont identifiés par leurs extensions :

*.*C* fichiers source

*.*OBJ* fichiers compilés (versions objet)

*.*EXE* fichiers compilés et liés (versions exécutables)

*.*LIB* bibliothèques de fonctions précompilées

*.*H* fichiers en-tête (*header files*)

Exemple :

Nous avons écrit un programme qui fait appel à des fonctions mathématiques et des fonctions graphiques prédéfinies. Pour pouvoir utiliser ces fonctions, le programme a besoin des bibliothèques :

```
MATHS.LIB
```

```
GRAPHICS.LIB
```

Nous devons donc inclure les fichiers en-tête correspondants dans le code source de notre programme à l'aide des instructions :

```
#include <math.h>
```

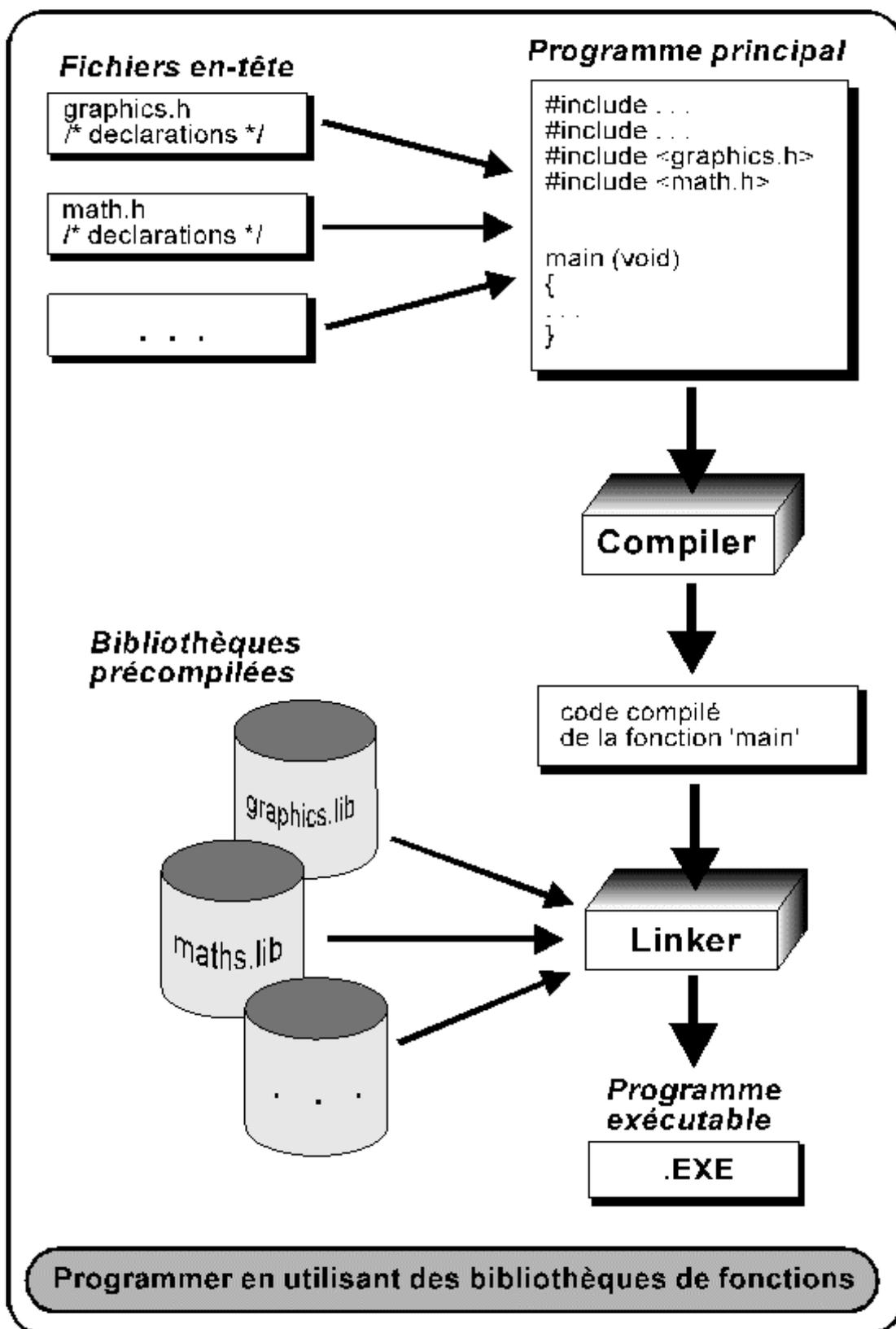
```
#include <graphics.h>
```

Après la compilation, les fonctions précompilées des bibliothèques seront ajoutées à notre programme pour former une version exécutable du programme (voir schéma).

Remarque :

La bibliothèque de fonctions **graphics.h** est spécifique aux fonctionnalités du PC et n'est pas incluse dans le standard ANSI-C.

Schéma : Bibliothèques de fonctions et compilation :



Chapitre 2 : NOTIONS DE BASE :

Avant de pouvoir comprendre ou même écrire des programmes, il faut connaître la composition des programmes dans le langage de programmation. Dans ce chapitre, nous allons discuter un petit programme en mettant en évidence les structures fondamentales d'un programme en C.

I) Hello C ! :

Suivons la tradition et commençons la découverte de C par l'inévitable programme 'hello world'. Ce programme ne fait rien d'autre qu'imprimer les mots suivants sur l'écran :



Comparons d'abord la définition du programme en C avec celle en langage algorithmique.

HELLO WORLD en langage algorithmique :

```
programme HELLO_WORLD
|   (* Notre premier programme en C *)
|   écrire "hello, world"
fprogramme
```

HELLO WORLD en C :

```
#include <stdio.h>
main()
/* Notre premier programme en C */
{
    printf("hello, world\n");
    return (0);
}
```

Dans la suite du chapitre, nous allons discuter les détails de cette implémentation.

II) Les composantes d'un programme en C :

Programmes, fonctions et variables :

Les programmes en C sont composés essentiellement de fonctions et de variables. Pour la pratique, il est donc indispensable de se familiariser avec les caractéristiques fondamentales de ces éléments.

1) Les fonctions :

En C, le programme principal et les sous-programmes sont définis comme fonctions. Il n'existe pas de structures spéciales pour le programme principal ni les procédures (comme en Pascal ou en langage algorithmique).

Le programme principal étant aussi une 'fonction', nous devons nous intéresser dès le début à la définition et aux caractéristiques des fonctions en C. Commençons par comparer la syntaxe de la définition d'une fonction en C avec celle d'une fonction en langage algorithmique :

Définition d'une fonction en langage algorithmique :

```
fonction <NomFonct> (<NomPar1>, <NomPar2>, ...):<TypeRés>
|   <déclarations des paramètres>
|   <déclarations locales>
|   <instructions>
ffonction
```

Définition d'une fonction en C :

```
<TypeRés> <NomFonct> (<TypePar1> <NomPar1>,  
                        <TypePar2> <NomPar2>, ... )  
{  
    <déclarations locales>  
    <instructions>  
}
```

En C, une fonction est définie par :

- une ligne déclarative qui contient :
 - <TypeRés> - le type du résultat de la fonction
 - <NomFonct> - le nom de la fonction
 - <TypePar1> <NomPar1>, <TypePar2> <NomPar2>, ...
les types et les noms des paramètres de la fonction
- un bloc d'instructions délimité par des accolades { }, contenant :
 - <déclarations locales> - les déclarations des données locales (c'est-à-dire : des données qui sont uniquement connues à l'intérieur de la fonction)
 - <instructions> - la liste des instructions qui définit l'action qui doit être exécutée

Résultat d'une fonction :

Par définition, *toute fonction en C fournit un résultat* dont le type doit être défini. Si aucun type n'est défini explicitement, C suppose par défaut que le type du résultat est **int** (integer).

Le retour du résultat se fait en général à la fin de la fonction par l'instruction **return**.

Le type d'une fonction qui ne fournit pas de résultat (comme les procédures en langage algorithmique ou en Pascal), est déclaré comme **void** (vide).

Paramètres d'une fonction :

La définition des paramètres (arguments) d'une fonction est placée entre parenthèses () derrière le nom de la fonction. Si une fonction n'a pas besoin de paramètres, les parenthèses restent vides ou contiennent le mot **void**. La fonction minimale qui ne fait rien et qui ne fournit aucun résultat est alors :

```
void dummy() {}
```

Instructions :

En C, *toute* instruction simple est terminée par un point virgule ; (même si elle se trouve en dernière position dans un bloc d'instructions). Par exemple :

```
printf("hello, world\n");
```

2) La fonction main :

La fonction **main** est la fonction principale des programmes en C : Elle se trouve obligatoirement dans tous les programmes. L'exécution d'un programme entraîne automatiquement l'appel de la fonction **main**.

Dans les premiers chapitres, nous allons simplement 'traduire' la structure programme du langage algorithmique par une définition équivalente de la fonction **main** :

Définition du programme principal en langage algorithmique :

```
programme <NomProgramme>  
|   <déclarations>  
|   <instructions>  
fprogramme
```

Définition de la fonction main en C :

```
main()
{
    <déclarations>
    <instructions>
    return (0);
}
```

Résultat de main :

En principe tout programme devrait retourner une valeur comme code d'erreur à son environnement. Par conséquent, le type résultat de **main** est toujours **int**. En général, le type de **main** n'est pas déclaré explicitement, puisque c'est le type par défaut. Nous allons terminer nos programmes par l'instruction :

```
return (0);
```

qui indique à l'environnement que le programme s'est terminé avec succès, sans anomalies ou erreurs fatales.

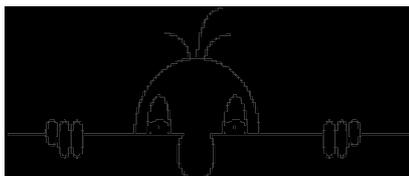
Paramètres de main :

- Si la liste des paramètres de la fonction **main** est vide, il est d'usage de la déclarer par **()**.
- Si nous utilisons des fonctions prédéfinies (par exemple : **printf**), il faut faire précéder la définition de **main** par les instructions **#include** correspondantes.

Remarque avancée :

Il est possible de faire passer des arguments de la ligne de commande à un programme. Dans ce cas, la liste des paramètres doit contenir les déclarations correspondantes. Dans notre cours, nous n'allons pas utiliser des arguments de la ligne de commande. Ainsi la liste des paramètres de la fonction **main** sera vide (**void**) dans tous nos exemples et nous pourrons employer la déclaration suivante qui fait usage des valeurs par défaut :

```
main() { ... }
```



Voici [l'exemple d'un programme utilisant des arguments de la ligne de commande](#),
Avec la forme :

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

on peut passer des paramètres sur la ligne d'exécution Ainsi si on a le programme : **MyProg.c**

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

```
int main(int argc, char *argv[])
{
    int idx;
    for (idx = 0; idx < argc; idx++)
    {
        printf("parameter %d value is %s\n", argc, argv[idx]);
    }
    return (0);
}
```

Si j'exécute MyProg de la façon suivante :

```
MyProg param1 param2 param3
```

J'aurai comme résultat :

```
Parameter 0 value is MyProg
Parameter 1 value is param1
```

```
Parameter 2 value is param2
Parameter 3 value is param3
```

Exemple publié avec l'aimable permission de **Francois Donato**

- [Exercice 2.1](#)

3) Les variables :

Les variables contiennent les valeurs qui sont utilisées pendant l'exécution du programme. Les noms des variables sont des identificateurs quelconques (voir 2.2.4). Les différents types de variables simples et les opérateurs admissibles sont discutés au chapitre 3.

4) Les identificateurs :

Les noms des fonctions et des variables en C sont composés d'une suite de lettres et de chiffres. Le premier caractère doit être une lettre. Le symbole '_' est aussi considéré comme une lettre.

* L'ensemble des symboles utilisables est donc :

{0,1,2,...,9,A,B,...,Z,_,a,b,...,z}

* Le premier caractère doit être une lettre (ou le symbole '_')

* C distingue les majuscules et les minuscules, ainsi :

'**Nom_de_variable**' est différent de '**nom_de_variable**'

* La longueur des identificateurs n'est pas limitée, mais C distingue 'seulement' les 31 premiers caractères.

Remarques :

- Il est déconseillé d'utiliser le symbole '_' comme premier caractère pour un identificateur, car il est souvent employé pour définir les variables globales de l'environnement C.
- Le standard dit que la validité de noms externes (par exemple noms de fonctions ou var. globales) peut être limité à 6 caractères (même sans tenir compte des majuscules et minuscules) par l'implémentation du compilateur, mais tous les compilateurs modernes distinguent au moins 31 caractères de façon à ce que nous pouvons généraliser qu'en pratique les règles ci-dessus s'appliquent à tous les identificateurs.

Exemples :

<i>Identificateurs corrects :</i>	<i>Identificateurs incorrects :</i>
nom1	1nom
nom_2	nom.2
_nom_3	-nom-3
Nom_de_variable	Nom de variable
deuxieme_choix	deuxième_choix
mot_francais	mot français

- [Exercice 2.2](#)

5) Les commentaires :

Un commentaire commence toujours par les deux symboles '/*' et se termine par les symboles '*/'. Il est interdit d'utiliser des commentaires imbriqués.

Exemples :

```
/* Ceci est un commentaire correct */
/* Ceci est /* évidemment */ défendu */
```

III) Discussion de l'exemple 'Hello World' :

Reprenons le programme 'Hello_World' et retrouvons les particularités d'un programme en C.

HELLO WORLD en C :

```
#include <stdio.h>
main()
/* Notre premier programme en C */
{
    printf("hello, world\n");
    return (0);
}
```

Discussion :

- La fonction **main** ne reçoit pas de données, donc la liste des paramètres est vide.
- La fonction **main** fournit un code d'erreur numérique à l'environnement, donc le type du résultat est **int** et n'a pas besoin d'être déclaré explicitement.
- Le programme ne contient pas de variables, donc le bloc de déclarations est vide.
- La fonction **main** contient deux instructions :
 - * l'appel de la fonction **printf** avec l'argument "hello, world\n";
Effet : Afficher la chaîne de caractères "hello world\n".
 - * la commande **return** avec l'argument 0 ;
Effet : Retourner la valeur 0 comme code d'erreur à l'environnement.
- L'argument de la fonction **printf** est une chaîne de caractères indiquée entre les guillemets. Une telle suite de caractères est appelée *chaîne de caractères constante (string constant)*.
- La suite de symboles '\n' à la fin de la chaîne de caractères "hello, world\n" est la notation C pour '*passage à la ligne*' (angl : new line). En C, il existe plusieurs couples de symboles qui contrôlent l'affichage ou l'impression de texte. Ces *séquences d'échappement* sont toujours précédées par le caractère d'échappement '\'. (voir exercice 2.4).

printf et la bibliothèque <stdio> :

La fonction **printf** fait partie de la bibliothèque de fonctions standard <stdio> qui gère les entrées et les sorties de données. La première ligne du programme :

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

instruit le compilateur d'inclure le fichier en-tête '**stdio.h**' dans le texte du programme.

Le fichier '**stdio.h**' contient les informations nécessaires pour pouvoir utiliser les fonctions de la bibliothèque standard <stdio> (voir chapitre 1.3).

-
- [Exercice 2.3](#)
 - [Exercice 2.4](#)
 - [Exercice 2.5](#)
-

IV) Exercices d'application :

Exercice 2.1 :

Comparez la syntaxe de la définition d'une fonction en C avec celle des fonctions et des procédures dans Pascal. (Basez-vous sur les observations mentionnées dans le cours.)

Exercice 2.2 :

Lesquels des identificateurs suivants sont acceptés par C ?

fonction-1	_MOYENNE_du_MOIS_	3e_jour
limite_inf.	lim_supérieure	_A_
_	a	3

Exercice 2.3 :

Modifiez le programme 'hello world' de façon à obtenir le même résultat sur l'écran en utilisant plusieurs fois la fonction **printf**.

Exercice 2.4 :

Expérimentez avec les séquences d'échappement que vous trouvez dans le tableau ci-dessous et complétez les colonnes vides.

<i>séquence d'échappement</i>	<i>Description anglaise</i>	<i>description française</i>
\n	new line	passage à la ligne
\t		
\b		
\r		
\"		
\\		
\0		
\a		

Exercice 2.5 :

Ci-dessous, vous trouvez un simple programme en C. Essayez de distinguer et de classer autant que possible les éléments qui composent ce programme (commentaires, variables, déclarations, instructions, etc.)

```
#include <stdio.h>
/* Ce programme calcule la somme de 4 nombres entiers
   introduits au clavier.
*/
main()
{
    int NOMBRE, SOMME, COMPTEUR;

    /* Initialisation des variables */
    SOMME = 0;
    COMPTEUR = 0;
    /* Lecture des données */
    while (COMPTEUR < 4)
```

```
{
  /* Lire la valeur du nombre suivant */
  printf("Entrez un nombre entier :");
  scanf("%i", &NOMBRE);
  /* Ajouter le nombre au résultat */
  SOMME += NOMBRE;
  /* Incrémenter le compteur */
  COMPTEUR++;
}
/* Impression du résultat */
printf("La somme est: %i \n", SOMME);
return (0);
}
```

V) Solutions des exercices du Chapitre 2 : NOTIONS DE BASE :

Exercice 2.1 :

En Pascal,	En C,
il existe des fonctions et des procédures	il existe seulement des fonctions. (Les fonctions fournissant le résultat void correspondent aux procédures de Pascal.)
il existe une structure spéciale program pour la définition du programme principal	la fonction principale main est exécutée automatiquement lors de l'appel du programme
les blocs d'instructions sont délimités par begin ... end	les blocs d'instructions sont délimités par { ... }
les commentaires sont délimités par (* ... *) ou { ... }	les commentaires sont délimités par /* ... */
les points-virgules séparent les instructions simples à l'intérieur d'un bloc d'instructions	les points-virgules marquent la fin de toutes les instructions simples

Exercice 2.2 :

fonction-1	Pas accepté (contient le caractère spécial '-')
_MOYENNE_du_MOIS_	Accepté
3e_jour	Pas accepté (chiffre au début de l'identificateur)
limite_inf.	Pas accepté (contient le caractère spécial '.')
lim_supérieure	Pas accepté (contient le caractère spécial 'é')
A	Accepté
_	Accepté
A	Accepté
3	Pas accepté (chiffre au début de l'identificateur)

Exercice 2.3 :

Voici une des solutions possibles :

```
#include <stdio.h>
/* Notre premier programme en C */
main()
{
    printf("hello, ");
    printf("world");
    printf("\n");
    return (0);
}
```

Exercice 2.4 :

Séquence d'échappement	Description Anglaise	Description Française
\n	New line	Passage à la ligne
\t	Tabulator	Tabulation
\b	Back	Curseur arrière

\r	Return	Retour au début de la ligne
\"	Quotation marks	Guillemets
\\	Back-slash	Trait oblique
\0	NUL	Fin de chaîne
\a	Attention (bell)	Signal acoustique

Exercice 2.5 :

```
#include <stdio.h>
/* Ce programme calcule la somme de 4 nombres entiers
   introduits au clavier.
*/
main()
{
    int NOMBRE, SOMME, COMPTEUR;
    /* Initialisation des variables */
    SOMME = 0;
    COMPTEUR = 0;
    /* Lecture des données */
    while (COMPTEUR < 4)
    {
        /* Lire la valeur du nombre suivant */
        printf("Entrez un nombre entier :");
        scanf("%i", &NOMBRE);
        /* Ajouter le nombre au résultat */
        SOMME += NOMBRE;
        /* Incrémenter le compteur */
        COMPTEUR++;
    }
    /* Impression du résultat */
    printf("La somme est : %i \n", SOMME);
    return (0);
}
```

Solution :

- Commande au compilateur : **#include<stdio.h>** pour pouvoir utiliser les fonctions **printf** et **scanf**.
- Fonction **main** n'a pas de paramètres (la liste des paramètres est vide) et fournit par défaut un résultat du type **int** (à l'environnement).
- Commentaires (mis en italique).
- Variables utilisées : NOMBRE, SOMME, COMPTEUR déclarées comme entiers (type **int**).
- Fonctions utilisées : **printf**, **scanf** de la bibliothèque *<stdio>*.
- Opérateurs :
 - += opérateur arithmétique d'affectation
 - ++ opérateur arithmétique
 - < opérateur de comparaison
 - = opérateur d'affectation
- Structure de contrôle : **while(<condition>) { ... }** répète le bloc d'instructions aussi longtemps que la <condition> est remplie.

- L'instruction **return 0** ;
retourne la valeur zéro comme code d'erreur à l'environnement après l'exécution du programme.

Ajoutes :

- la fonction **scanf** est appelée avec deux paramètres :
le format de saisie de la donnée (ici : "%i" pour lire un entier du type **int**) ,
l'adresse de la variable destination (ici : l'adresse de NOMBRE).
 - la fonction **printf** est appelée avec un respectivement avec deux paramètres :
le premier paramètre est une chaîne de caractères, qui peut contenir une information pour le format d'affichage des variables indiquées dans la suite (ici : "%i" pour afficher la valeur du type **int** contenue dans SOMME).
les paramètres qui suivent la chaîne de caractères indiquent les noms des variables à afficher.
(ici : SOMME)
-

Chapitre 3 : TYPES DE BASE, OPÉRATEURS ET EXPRESSIONS :

Récapitulatif du vocabulaire :

Les *variables* et les *constantes* sont les données principales qui peuvent être manipulées par un programme. Les *déclarations* introduisent les variables qui sont utilisées, fixent leur type et parfois aussi leur valeur de départ. Les *opérateurs* contrôlent les actions que subissent les valeurs des données. Pour produire de nouvelles valeurs, les variables et les constantes peuvent être combinées à l'aide des opérateurs dans des *expressions*. Le *type* d'une donnée détermine l'ensemble des valeurs admissibles, le nombre d'octets à réserver en mémoire et l'ensemble des opérateurs qui peuvent y être appliqués.

Motivation :

La grande flexibilité de C nous permet d'utiliser des opérandes de différents types dans un même calcul. Cet avantage peut se transformer dans un terrible piège si nous ne prévoyons pas correctement les effets secondaires d'une telle opération (conversions de type automatiques, arrondissements, etc.). Une étude minutieuse de ce chapitre peut donc aider à éviter des phénomènes parfois 'inexplicables' ...

I) Les types simples :

Ensembles de nombres et leur représentation :

En mathématiques, nous distinguons divers ensembles de nombres :

- * l'ensemble des entiers naturels **IN**,
- * l'ensemble des entiers relatifs **ZZ**,
- * l'ensemble des rationnels **Q**,
- * l'ensemble des réels **IR**.

En mathématiques l'ordre de grandeur des nombres est illimité et les rationnels peuvent être exprimés sans perte de précision.

Un ordinateur ne peut traiter aisément que des nombres entiers d'une taille limitée. Il utilise le système binaire pour calculer et sauvegarder ces nombres. Ce n'est que par des astuces de calcul et de représentation que l'ordinateur obtient des valeurs correctement approchées des entiers très grands, des réels ou des rationnels à partie décimale infinie.

Les charges du programmeur :

Même un programmeur utilisant C ne doit pas connaître tous les détails des méthodes de codage et de calcul, il doit quand même être capable de :

- *choisir un type numérique approprié à un problème donné; c'est-à-dire : trouver un optimum de précision, de temps de calcul et d'espace à réserver en mémoire.*
- *choisir un type approprié pour la représentation sur l'écran.*
- *prévoir le type résultant d'une opération entre différents types numériques ; c'est-à-dire : connaître les transformations automatiques de type que C accomplit lors des calculs (voir 3.7.1).*
- *prévoir et optimiser la précision des résultats intermédiaires au cours d'un calcul complexe; c'est-à-dire : changer si nécessaire l'ordre des opérations ou forcer l'ordinateur à utiliser un type numérique mieux adapté (casting : voir 3.7.2).*

Exemple :

Supposons que la mantisse du type choisi ne comprenne que 6 positions décimales (ce qui est très réaliste pour le type **float**) :

$$\begin{aligned} & \underbrace{(1.00001 \cdot 10^8 + 850)} - 1 \cdot 10^8 \\ & = 1.00001 \cdot 10^8 - 1 \cdot 10^8 = 1000 \quad \text{💣} \\ & \underbrace{(1.00001 \cdot 10^8 - 1 \cdot 10^8)} + 850 \\ & = 1000 + 850 = 1850 \quad \checkmark \end{aligned}$$

1) Les types entiers :

Avant de pouvoir utiliser une variable, nous devons nous intéresser à deux caractéristiques de son type numérique :

- le domaine des valeurs admissibles,
- le nombre d'octets qui est réservé pour une variable.

Le tableau suivant résume les caractéristiques des types numériques entiers de C :

définition	description	domaine min	Domaine max	nombre d'octets
char	caractère	-128	127	1
short	entier court	-32768	32767	2
int	Entier standard	-32768	32767	2
long	entier long	-2147483648	2147483647	4

- **char** : caractère

Une variable du type **char** peut contenir une valeur entre -128 et 127 et elle peut subir les mêmes opérations que les variables du type **short**, **int** ou **long**.

- **int** : entier standard

Sur chaque machine, le type **int** est le type de base pour les calculs avec les entiers. Le codage des variables du type **int** est donc dépendant de la machine. Sur les IBM-PC sous MS-DOS, une variable du type **int** est codée dans deux octets.

- **short** : entier court

Le type **short** est en général codé dans 2 octets. Comme une variable **int** occupe aussi 2 octets sur notre système, le type **short** devient seulement nécessaire, si on veut utiliser le même programme sur d'autres machines, sur lesquelles le type standard des entiers n'est pas forcément 2 octets.

- Les modificateurs **signed/unsigned** :

Si on ajoute le préfixe **unsigned** à la définition d'un type de variables entières, les domaines des variables sont déplacés comme suit :

définition	description	Domaine min	domaine max	nombre d'octets
unsigned char	caractère	0	255	1
unsigned short	entier court	0	65535	2
unsigned int	entier standard	0	65535	2
unsigned long	entier long	0	4294967295	4

Remarques :

1. Le calcul avec des entiers définis comme **unsigned** correspond à l'arithmétique *modulo 2ⁿ*. Ainsi, en utilisant une variable X du type **unsigned short**, nous obtenons le résultat suivant :

$$\begin{aligned} \text{Affectation : } & X = 65500 + 100 \\ \text{Résultat : } & X = 64 \quad /* [+2^{16}] */ \end{aligned}$$

2. Par défaut, les types entiers **short**, **int**, **long** sont munis d'un signe. Le type par défaut de **char** est dépendant du compilateur et peut être **signed** ou **unsigned**. Ainsi, l'attribut **signed** a seulement un sens en liaison avec **char** et peut forcer la machine à utiliser la représentation des caractères avec signe (qui n'est cependant pas très usuelle).

3. Les valeurs limites des différents types sont indiquées dans le fichier header **<limits.h>**.

4. (Ajoute de **Francois Donato**) En principe, on peut dire que :

sizeof(short) <= sizeof(int) <= sizeof(long)

Ainsi sur certaine architecture on peut avoir

short = 2 octets, int = 2 octets, long = 4 octets
et sur d'autre

short = 2 octets, int = 4 octets, long = 4 octets
(Il sera intéressant de voir l'implémentation dans un environnement 64 bits!)

2) Les types rationnels :

En informatique, les rationnels sont souvent appelés des 'flottants'. Ce terme vient de '*en virgule flottante*' et trouve sa racine dans la notation traditionnelle des rationnels :

	<+ -> <mantisse> * 10^{<exposant>}
<+ ->	<i>est le signe positif ou négatif du nombre</i>
<mantisse>	<i>est un décimal positif avec un seul chiffre devant la virgule.</i>
<exposant>	<i>est un entier relatif</i>

Exemples :

3.14159*100 1.25003*10⁻¹²
4.3001*10321 -1.5*103

En C, nous avons le choix entre trois types de rationnels : **float**, **double** et **long double**. Dans le tableau ci-dessous, vous trouverez leurs caractéristiques :

<i>min et max</i>	<i>représentent les valeurs minimales et maximales positives. Les valeurs négatives peuvent varier dans les mêmes domaines.</i>
<i>Mantisse</i>	<i>indique le nombre de chiffres significatifs de la mantisse.</i>

Définition	Précision	Mantisse	Domaine min	Domaine max	Nombre d'octets
Float	<i>simple</i>	6	$3.4 * 10^{-38}$	$3.4 * 10^{38}$	4
Double	<i>double</i>	15	$1.7 * 10^{-308}$	$1.7 * 10^{308}$	8
Long double	<i>suppl.</i>	19	$3.4 * 10^{-4932}$	$1.1 * 10^{4932}$	10

Remarque avancée :

Les détails de l'implémentation sont indiqués dans le fichier header **<float.h>**.

- [Exercice 3.1](#)

II) La déclaration des variables simples :

Maintenant que nous connaissons les principaux types de variables, il nous faut encore la syntaxe pour leur déclaration :

Déclaration de variables en langage algorithmique :

<Type> <NomVar1>, <NomVar2>, ..., <NomVarN>

Déclaration de variables en C :

<Type> <NomVar1>, <NomVar2>, ..., <NomVarN>;

Prenons quelques déclarations du langage descriptif,

```
entier    COMPTEUR, X, Y
réel     HAUTEUR, LARGEUR, MASSE_ATOMIQUE
caractère TOUCHE
booléen  T_PRESSEE
```

et traduisons-les en des déclarations du langage C :

```
int    compteur, X, Y;
float  hauteur, largeur;
double masse_atomique;
char   touche;
int    t_pressee;
```

Langage algorithmique --> C :

En général, nous avons le choix entre plusieurs types et nous devons trouver celui qui correspond le mieux au domaine et aux valeurs à traiter. Voici quelques règles générales qui concernent la traduction des déclarations de variables numériques du langage algorithmique en C :

- La **syntaxe** des déclarations en C ressemble à celle du langage algorithmique. Remarquez quand même les points virgules à la fin des déclarations en C.

Entier : Nous avons le choix entre tous les types entiers (inclusivement **char**) dans leurs formes **signed** ou **unsigned**. Si les nombres deviennent trop grands pour **unsigned long**, il faut utiliser un type rationnel (p.ex : **double**)

Réel : Nous pouvons choisir entre les trois types rationnels en observant non seulement la grandeur maximale de l'exposant, mais plus encore le nombre de chiffres significatifs de la mantisse.

Caractère : Toute variable du type **char** peut contenir un (seul) caractère. En C, il faut toujours être conscient que ce 'caractère' n'est autre chose qu'un nombre correspondant à un code (ici : code ASCII). Ce nombre peut être intégré dans toute sorte d'opérations algébriques ou logiques ...

Chaîne : En C il n'existe pas de type spécial pour chaînes de caractères. Les moyens de traiter les chaînes de caractères seront décrits au chapitre 8.

Booléen : En C il n'existe pas de type spécial pour variables booléennes. Tous les types de variables numériques peuvent être utilisés pour exprimer des opérations logiques :

valeur logique <u>faux</u>	<=>	valeur numérique zéro
valeur logique <u>vrai</u>	<=>	toute valeur différente de zéro

Si l'utilisation d'une variable booléenne est indispensable, le plus naturel sera d'utiliser une variable du type **int**.

Les opérations logiques en C retournent toujours des résultats du type **int** :

0 pour faux
1 pour vrai

- [Exercice 3.2](#)

1) Les constantes numériques :

En pratique, nous utilisons souvent des valeurs constantes pour calculer, pour initialiser des variables, pour les comparer aux variables, etc. Dans ces cas, l'ordinateur doit attribuer un type numérique aux valeurs constantes. Pour pouvoir prévoir le résultat et le type exact des calculs, il est

important pour le programmeur de connaître les règles selon lesquelles l'ordinateur choisit les types pour les constantes.

a) - Les constantes entières :

Type automatique :

Lors de l'attribution d'un type à une constante entière, C choisit en général la solution la plus économique :

Le type des constantes entières :

- * Si possible, les constantes entières obtiennent le type **int**.
- * Si le nombre est trop grand pour **int** (par ex : -40000 ou +40000) il aura automatiquement le type **long**.
- * Si le nombre est trop grand pour **long**, il aura le type **unsigned long**.
- * Si le nombre est trop grand pour **unsigned long**, la réaction du programme est imprévisible.

Type forcé :

Si nous voulons forcer l'ordinateur à utiliser un type de notre choix, nous pouvons employer les suffixes suivants :

<i>suffixe</i>	<i>type</i>	<i>Exemple</i>
u ou U	unsigned (int ou long)	550u
l ou L	long	123456789L
ul ou UL	unsigned long	12092UL

Exemples :

12345	type int
52000	type long
-2	type int
0	type int
1u	type unsigned int
52000u	type unsigned long
22lu	Erreur !

Base octale et hexadécimale :

Il est possible de déclarer des constantes entières en utilisant la base *octale* ou *hexadécimale* :

- * Si une constante entière commence par **0** (zéro), alors elle est interprétée en base octale.
- * Si une constante entière commence par **0x** ou **0X** , alors elle est interprétée en base hexadécimale.

Exemples :

<i>base décimale</i>	<i>base octale</i>	<i>base hexadécimale</i>	<i>représentation binaire</i>
100	0144	0X64	1100100
255	0377	0xff	11111111
65536	0200000	0X10000	100000000000000000
12	014	0XC	1100
4040	07710	0xFC8	111111001000

b) Les constantes rationnelles :

Les constantes rationnelles peuvent être indiquées :

- * *en notation décimale*, c'est-à-dire à l'aide d'un point décimal :

Exemples :

123.4 -0.001 1.0

- * *en notation exponentielle*, c'est-à-dire à l'aide d'un exposant séparé du nombre décimal par les caractères 'e' ou 'E' :

Exemples :

1234e-1 -1E-3 0.01E2

L'ordinateur reconnaît les constantes rationnelles au point décimal ou au séparateur de l'exposant ('e' ou 'E'). Par défaut, les constantes rationnelles sont du type **double**.

Le type des constantes rationnelles :

- * Sans suffixe, les constantes rationnelles sont du type **double**.
- * Le suffixe **f** ou **F** force l'utilisation du type **float**.
- * Le suffixe **l** ou **L** force l'utilisation du type **long double**.

c) Les caractères constants :

Les constantes qui désignent un (seul) caractère sont toujours indiquées entre des apostrophes : par exemple 'x'. La valeur d'un caractère constant est le code interne de ce caractère. Ce code (ici : le code ASCII) est dépendant de la machine.

Les caractères constants peuvent apparaître dans des opérations arithmétiques ou logiques, mais en général ils sont utilisés pour être comparés à des variables.

d) Séquences d'échappement :

Comme nous l'avons vu au chapitre 2, l'impression et l'affichage de texte peut être contrôlé à l'aide de *séquences d'échappement*. Une séquence d'échappement est un couple de symboles dont le premier est le *signe d'échappement* '\'. Au moment de la compilation, chaque séquence d'échappement est traduite en un caractère de contrôle dans le code de la machine. Comme les séquences d'échappement sont identiques sur toutes les machines, elles nous permettent d'écrire des programmes portables, c'est-à-dire : des programmes qui ont le même effet sur toutes les machines, indépendamment du code de caractères utilisé.

Séquences d'échappement :

\a	<i>sonnerie</i>	\\	<i>trait oblique</i>
\b	<i>curseur arrière</i>	\?	<i>point d'interrogation</i>
\t	<i>tabulation</i>	\'	<i>apostrophe</i>
\n	<i>nouvelle ligne</i>	\"	<i>guillemets</i>
\r	<i>retour au début de ligne</i>	\f	<i>saut de page (imprimante)</i>
\0	<i>NUL</i>	\v	<i>tabulateur vertical</i>

Le caractère NUL :

La constante '\0' qui a la valeur numérique zéro (ne pas à confondre avec le caractère '0' !!) a une signification spéciale dans le traitement et la mémorisation des chaînes de caractères : En C le caractère '\0' définit **la fin d'une chaîne** de caractères.

-
- [Exercice 3.3](#)
 - [Exercice 3.4](#)
-

2) Initialisation des variables :

Initialisation :

En C, il est possible d'initialiser les variables lors de leur déclaration :

```
int    MAX = 1023;
char   TAB = '\t';
float  X   = 1.05e-4;
```

const :

En utilisant l'attribut **const**, nous pouvons indiquer que la valeur d'une variable ne change pas au cours d'un programme :

```
const int    MAX = 767;
```

```
const double e = 2.71828182845905;
const char NEWLINE = '\n';
```

III) Les opérateurs standard :

Affectation en langage algorithmique :

en <NomVariable> ranger <Expression>

Affectation en C :

<NomVariable> = <Expression>;

1) Exemples d'affectations :

Reprenons les exemples du cours d'algorithmique pour illustrer différents types d'affectations :

a) - L'affectation avec des valeurs constantes :

<i>Langage algorithmique</i>	<i>C</i>	<i>Type de la constante</i>
<u>en</u> LONG <u>ranger</u> 141	LONG = 141;	(const. entière)
<u>en</u> PI <u>ranger</u> 3.1415926	PI = 3.1415926;	(const. réelle)
<u>en</u> NATION <u>ranger</u> "L"	NATION = 'L';	(caractère const.)

b) - L'affectation avec des valeurs de variables :

<i>Langage algorithmique</i>	<i>C</i>
<u>en</u> VALEUR <u>ranger</u> X1A	VALEUR = X1A;
<u>en</u> LETTRE <u>ranger</u> COURRIER	LETTRE = COURRIER;

c) - L'affectation avec des valeurs d'expressions :

<i>Langage algorithmique</i>	<i>C</i>
<u>en</u> AIRE <u>ranger</u> PI*R ²	AIRE = PI*pow(R,2);
<u>en</u> MOYENNE <u>ranger</u> (A+B)/2	MOYENNE = (A+B)/2;
<u>en</u> UN <u>ranger</u> sin ² (X)+cos ² (X)	UN=pow(sin(X),2)+pow(cos(X),2);
<u>en</u> RES <u>ranger</u> 45+5*X	RES = 45+5*X;
<u>en</u> PLUSGRAND <u>ranger</u> (X>Y)	PLUSGRAND = (X>Y);
<u>en</u> CORRECT <u>ranger</u> ('a'='a')	CORRECT = ('a' == 'a');

d) Observations :

Les opérateurs et les fonctions arithmétiques utilisées dans les expressions seront introduites dans la suite du chapitre. Observons déjà que :

* il n'existe pas de fonction standard en C pour calculer le carré d'une valeur; on peut se référer à la fonction plus générale **pow(x,y)** qui calcule x^y (voir 3.5.).

* le test d'égalité en C se formule avec *deux* signes d'égalité == , l'affectation avec *un seul* = .

2) Les opérateurs connus :

Avant de nous lancer dans les 'spécialités' du langage C, retrouvons d'abord les opérateurs correspondant à ceux que nous connaissons déjà en langage descriptif et en Pascal.

a) Opérateurs arithmétiques :

+	addition
-	soustraction
*	multiplication
/	division (entière et rationnelle!)
%	modulo (reste d'une div. entière)

b) Opérateurs logiques :

&&	et logique (and)
	ou logique (or)
!	négation logique (not)

c) Opérateurs de comparaison

==	égal à
!=	différent de
<, <=, >, >=	plus petit que, ...

d) Opérations logiques :

Les résultats des opérations de comparaison et des opérateurs logiques sont du type **int** :

- - la valeur 1 correspond à la valeur booléenne vrai
- - la valeur 0 correspond à la valeur booléenne faux

Les opérateurs logiques considèrent toute valeur différente de zéro comme vrai et zéro comme faux :

!	32 && 2.3	1
!	65.34	0
!	0 (32 > 12)	0

3) Les opérateurs particuliers de C :

a) Les opérateurs d'affectation :

Opération d'affectation :

En pratique, nous retrouvons souvent des affectations comme :

`i = i + 2`

En C, nous utiliserons plutôt la formulation plus compacte :

`i += 2`

L'opérateur += est un *opérateur d'affectation*.

Pour la plupart des expressions de la forme :

expr1 = (expr1) op (expr2)

il existe une formulation équivalente qui utilise un opérateur d'affectation :

expr1 op= expr2

Opérateurs d'affectation :

+=	ajouter à
-=	diminuer de
*=	multiplier par
/=	diviser par
%=	modulo

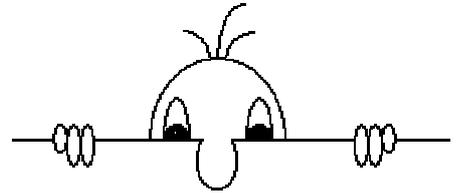
Avantages :

La formulation avec un opérateur d'affectation est souvent plus près de la logique humaine :
Un homme dirait <<Ajoute 2 à I>> plutôt que <<Ajoute 2 à I et écris le résultat dans I>>

Les opérateurs d'affectation peuvent aider le compilateur à générer un code plus efficace parce que *expr1* n'est évalué qu'une seule fois.

Les opérateurs d'affectation deviennent le plus intéressants si *expr1* est une expression complexe. Ceci peut être le cas si nous calculons avec des tableaux. L'expression :

$\text{Element}[n*i+j] = \text{Element}[n*i+j] * x[j]$
peut être formulée de manière plus efficace et plus claire :
 $\text{Element}[n*i+j] *= x[j]$



b) Opérateurs d'incrément et de décrémentation :

Les affectations les plus fréquentes sont du type :

$I = I + 1$ et $I = I - 1$

En C, nous disposons de deux opérateurs inhabituels pour ces affectations :

I++ ou ++I	pour l'incrément	(augmentation d'une unité)
I-- ou --I	pour la décrémentation	(diminution d'une unité)

Les opérateurs ++ et -- sont employés dans les cas suivants :

incrémenter/décrémenter une variable (par ex : dans une boucle). Dans ce cas il n'y a pas de différence entre la notation *préfixe* (**++I --I**) et la notation *postfixe* (**I++ I--**).

incrémenter/décrémenter une variable et en même temps affecter sa valeur à une autre variable. Dans ce cas, nous devons choisir entre la notation préfixe et postfixe :



X = I++ passe d'abord la valeur de I à X et *incrémente après*
X = I-- passe d'abord la valeur de I à X et *décrémente après*
X = ++I *incrémente d'abord* et passe la valeur incrémentée à X
X = --I *décrémente d'abord* et passe la valeur décrémentée à X

Exemple :

Supposons que la valeur de N est égal à 5 :

Incrém. postfixe : **X = N++;** Résultat : N=6 et X=5
Incrém. préfixe : **X = ++N;** Résultat : N=6 et X=6

IV) Les expressions et les instructions :

1) Expressions :

La formation des expressions est définie par récurrence :

Les *constantes* et les *variables* sont des expressions.

Les expressions peuvent être combinées entre elles par des *opérateurs* et former ainsi des expressions plus complexes.

Les expressions peuvent contenir des appels de fonctions et elles peuvent apparaître comme paramètres dans des appels de *fonctions*.

Exemples :

```
i = 0
i++
X=pow(A, 4)
printf(" Bonjour !\n")
a=(5*x+10*y)*2
(a+b)>=100
position!=limite
```

2) Instructions :

Une expression comme `I=0` ou `I++` ou `printf(...)` devient une *instruction*, si elle est suivie d'un point virgule.

Exemples :

```
i=0;
i++;
X=pow(A,4);
printf(" Bonjour !\n");
a=(5*x+10*y)*2;
```

3) Évaluation et résultats :

En C toutes les expressions sont évaluées et retournent une valeur comme résultat :

```
(3+2==5)    retourne la valeur 1 (vrai)
A=5+3       retourne la valeur 8
```

Comme les affectations sont aussi interprétées comme des expressions, il est possible de profiter de la valeur rendue par l'affectation :

```
((A=sin(X)) == 0.5)
```

V) Les priorités des opérateurs :

L'ordre de l'évaluation des différentes parties d'une expression correspond en principe à celle que nous connaissons des mathématiques.

Exemple :

Supposons pour l'instruction suivante : `A=5, B=10, C=1`

```
X = 2*A+3*B+4*C;
```

L'ordinateur évalue d'abord les multiplications :

```
2*A ==> 10 , 3*B ==> 30 , 4*C ==> 4
```

Ensuite, il fait l'addition des trois résultats obtenus :

```
10+30+4 ==> 44
```

A la fin, il affecte le résultat général à la variable :

```
X = 44
```

1) Priorité d'un opérateur :

On dit alors que la multiplication *a la priorité* sur l'addition et que la multiplication et l'addition ont la priorité sur l'affectation.

Si nous voulons forcer l'ordinateur à commencer par un opérateur avec une priorité plus faible, nous devons (comme en mathématiques) entourer le terme en question par des *parenthèses*.

Exemple :

Dans l'instruction :

```
X = 2*(A+3)*B+4*C;
```

l'ordinateur évalue d'abord l'expression entre parenthèses, ensuite les multiplications, ensuite l'addition et enfin l'affectation. (En reprenant les valeurs de l'exemple ci-dessus, le résultat sera 164).

Entre les opérateurs que nous connaissons jusqu'ici, nous pouvons distinguer les classes de priorités suivantes :

2) Classes de priorités :

Priorité 1 (la plus forte) :	()
Priorité 2 :	! ++ --

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

3) Evaluation d'opérateurs de la même classe :

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

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

Exemples :

L'expression 10+20+30-40+50-60 sera évaluée comme suit :

$$\begin{array}{l}
 \mathbf{10+20} \implies 30 \\
 \mathbf{30+30} \implies 60 \\
 \mathbf{60-40} \implies 20 \\
 \mathbf{20+50} \implies 70 \\
 \mathbf{70-60} \implies 10
 \end{array}$$

Pour A=3 et B=4, l'expression A *= B += 5 sera évaluée comme suit :

$$\begin{array}{c}
 \underbrace{\mathbf{B += 5}} \\
 \mathbf{B = 9} \\
 \underbrace{\mathbf{A *= 9}} \\
 \mathbf{A = 27}
 \end{array}$$

Pour A=1 et B=4, l'expression !--A==++!B sera évaluée comme suit :

$$\begin{array}{c}
 \underbrace{\mathbf{--A}} \\
 \mathbf{!0} \\
 \mathbf{1} \\
 \downarrow \\
 \mathbf{1}
 \end{array}
 \quad
 \begin{array}{c}
 \underbrace{\mathbf{!B}} \\
 \mathbf{++0} \\
 \mathbf{1}
 \end{array}$$

$$\underbrace{\mathbf{1 == 1}} \\
 \mathbf{1}$$

Les parenthèses

Les parenthèses sont seulement nécessaires si nous devons forcer la priorité, mais elles sont aussi permises si elles ne changent rien à la priorité. En cas de parenthèses imbriquées, l'évaluation se fait de l'intérieur vers l'extérieur.

Exemple :

En supposant à nouveau que A=5, B=10, C=1 l'expression suivante s'évaluera à 134 :

$$\mathbf{X = ((2*A+3)*B+4)*C}$$

Observez la priorité des opérateurs d'affectation :



X *= Y + 1
X *= Y + 1

X = X * (Y + 1)
X = X * Y + 1

- [Exercice 3.5](#)
- [Exercice 3.6](#)

VI) Les fonctions arithmétiques standard :

Les fonctions suivantes sont prédéfinies dans la bibliothèque standard `<math>`. Pour pouvoir les utiliser, le programme doit contenir la ligne :

```
#include <math.h>
```

1) Type des données :

Les arguments et les résultats des fonctions arithmétiques sont du type **double**.

Fonctions arithmétiques :

COMMANDE C	EXPLICATION	LANG. ALGORITHMIQUE
exp(X)	Fonction exponentielle	e^X
log(X)	Logarithme naturel	$\ln(X)$, $X>0$
log10(X)	Logarithme à base 10	$\log_{10}(X)$, $X>0$
pow(X,Y)	X exposant Y	X^Y
sqrt(X)	Racine carrée de X	pour $X>0$
fabs(X)	Valeur absolue de X	$ X $
floor(X)	Arrondir en moins	int(X)
ceil(X)	Arrondir en plus	
fmod(X,Y)	Reste rationnel de X/Y (même signe que X)	pour X différent de 0

sin(X), cos(X), tan(X)	sinus, cosinus, tangente de X
asin(X), acos(X), atan(X)	arcsin(X), arccos(X), arctan(X)
sinh(X), cosh(X), tanh(X)	sinus, cosinus, tangente hyperboliques de X

Remarque avancée :

La liste des fonctions ne cite que les fonctions les plus courantes. Pour la liste complète et les constantes prédéfinies voir `<math.h>`.

- [Exercice 3.7](#)

VII) Les conversions de type :

La grande souplesse du langage C permet de mélanger des données de différents types dans une expression. Avant de pouvoir calculer, les données doivent être converties dans un même type. La plupart de ces conversions se passent automatiquement, sans l'intervention du programmeur, qui doit quand même prévoir leur effet. Parfois il est nécessaire de convertir une donnée dans un type différent de celui que choisirait la conversion automatique; dans ce cas, nous devons forcer la conversion à l'aide d'un opérateur spécial ("cast").

1) Les conversions de type automatiques :

a) Calculs et affectations :

Si un opérateur a des opérandes de différents types, les valeurs des opérandes sont converties automatiquement dans un type commun.

Ces manipulations implicites convertissent en général des types plus 'petits' en des types plus 'larges'; de cette façon on ne perd pas en précision.



Lors d'une affectation, la donnée à droite du signe d'égalité est convertie dans le type à gauche du signe d'égalité. Dans ce cas, il peut y avoir perte de précision si le type de la destination est plus faible que celui de la source.

Exemple :

Considérons le calcul suivant :

```
int I = 8;
float X = 12.5;
double Y;
Y = I * X;
```

Pour pouvoir être multiplié avec *X*, la valeur de *I* est convertie en **float** (le type le plus large des deux). Le résultat de la multiplication est du type **float**, mais avant d'être affecté à *Y*, il est converti en **double**. Nous obtenons comme résultat :

```
Y = 100.00
```

b) Appels de fonctions :

Lors de l'appel d'une fonction, les paramètres sont automatiquement convertis dans les types déclarés dans la définition de la fonction.

Exemple :

Au cours des expressions suivantes, nous assistons à trois conversions automatiques :

```
int A = 200;
int RES;
RES = pow(A, 2);
```

A l'appel de la fonction **pow**, la valeur de *A* et la constante 2 sont converties en **double**, parce que **pow** est définie pour des données de ce type. Le résultat (type **double**) retourné par **pow** doit être converti en **int** avant d'être affecté à *RES*.

c) Règles de conversion automatique :

Conversions automatiques lors d'une opération avec,

- (1) deux entiers :
D'abord, les types **char** et **short** sont convertis en **int**. Ensuite, l'ordinateur choisit le plus large des deux types dans l'échelle suivante :

```
int, unsigned int, long, unsigned long
```

- (2) un entier et un rationnel :
Le type entier est converti dans le type du rationnel.
- (3) deux rationnels :
L'ordinateur choisit le plus large des deux types selon l'échelle suivante :

```
float, double, long double
```

- (4) affectations et opérateurs d'affectation :
Lors d'une affectation, le résultat est toujours converti dans le type de la destination. Si ce type est plus faible, il peut y avoir une perte de précision.

Exemple :

Observons les conversions nécessaires lors d'une simple division :

```
int X;
float A=12.48;
char B=4;
X=A/B;
```

B est converti en **float** (règle 2). Le résultat de la division est du type **float** (valeur 3.12) et sera converti en **int** avant d'être affecté à X (règle 4), ce qui conduit au résultat $X=3$.

d) Phénomènes imprévus ... :

Le mélange de différents types numériques dans un calcul peut inciter à ne pas tenir compte des phénomènes de conversion et conduit parfois à des résultats imprévus ...

Exemple :

Dans cet exemple, nous divisons 3 par 4 à trois reprises et nous observons que le résultat ne dépend pas seulement du type de la destination, mais aussi du type des opérandes.

```
char A=3;
int B=4;
float C=4;
float D,E;
char F;
D = A/C;
E = A/B;
F = A/C;
```

- * Pour le calcul de D , A est converti en **float** (règle 2) et divisé par C . Le résultat (0.75) est affecté à D qui est aussi du type **float**. On obtient donc : $D=0.75$.
- * Pour le calcul de E , A est converti en **int** (règle 1) et divisé par B . Le résultat de la division (type **int**, valeur 0) est converti en **float** (règle 4). On obtient donc : $E=0.000$
- * Pour le calcul de F , A est converti en **float** (règle 2) et divisé par C . Le résultat (0.75) est retraduit en **char** (règle 4). On obtient donc : $F=0$

e) Perte de précision :

Lorsque nous convertissons une valeur en un type qui n'est pas assez précis ou pas assez grand, la valeur est coupée sans arrondir et sans nous avertir ...

Exemple :

```
unsigned int A = 70000;
/* la valeur de A sera : 70000 mod 65536 = 4464 */
```

-
- [Exercice 3.8](#)
-

2) Les conversions de type forcées (casting) :

Il est possible de convertir explicitement une valeur en un type quelconque en forçant la transformation à l'aide de la syntaxe :

Casting (conversion de type forcée) :

```
(<Type>) <Expression>
```

Exemple :

Nous divisons deux variables du type entier. Pour avoir plus de précision, nous voulons avoir un résultat de type rationnel. Pour ce faire, nous convertissons l'une des deux opérandes en **float**. Automatiquement C convertira l'autre opérande en **float** et effectuera une division rationnelle :

```
char A=3;
int B=4;
float C;
C = (float)A/B;
```

La valeur de A est explicitement convertie en **float**. La valeur de B est automatiquement convertie en **float** (règle 2). Le résultat de la division (type rationnel, valeur 0.75) est affecté à C .

Résultat : $C=0.75$

Attention ! :

Les contenus de A et de B restent inchangés; seulement les valeurs utilisées dans les calculs sont converties !

VIII) Exercices d'application :

Exercice 3.1 :

Quel(s) type(s) numérique(s) pouvez-vous utiliser pour les groupes de nombres suivants? Dressez un tableau et marquez le choix le plus économique :

(1)	1	12	4	0	-125
(2)	1	12	-4	0	250
(3)	1	12	4	0	250
(4)	1	12	-4	0.5	125
(5)	-220	32000	0		
(6)	-3000005.000000001				
(7)	410	50000	2		
(8)	410	50000	-2		
(9)	3.14159265	1015			
(10)	2*10 ⁷	10000001			
(11)	2*10 ⁻⁷	10000001			
(12)	-1.05*1050	0.0001			
(13)	305.122212	0	-12		

Exercice 3.2 :

Traduisez les déclarations suivantes en C, sachant que vous travaillerez dans les ensembles de nombres indiqués. Choisissez les types les plus économiques, sans perdre en précision.

- | | | | |
|------|----------------|----------|---|
| (1) | <u>entier</u> | COMPTEUR | {0 ,..., 300} |
| (2) | <u>entier</u> | X,Y | {-120 ,..., 100} |
| (3) | <u>entier</u> | MESURE | {-10 ,..., 10 ⁴ } |
| (4) | <u>réel</u> | SURFACE1 | {0.5 ,..., 150075} |
| (5) | <u>réel</u> | SURFACE2 | {-12 ,..., 1500750.5} |
| (6) | <u>entier</u> | N1 | {0 ,..., 2 ¹⁰ } |
| (7) | <u>entier</u> | N2 | {-4 ⁷ ,..., 4 ⁷ } |
| (8) | <u>entier</u> | N3 | {0 ,..., 32 ⁶ } |
| (9) | <u>entier</u> | N4 | {-128 ⁰ ,..., 128 ⁵ } |
| (10) | <u>booléen</u> | TROUVE | { <u>vrai</u> , <u>faux</u> } |

Exercice 3.3 :

Complétez le tableau suivant :

<i>base décimale</i>	<i>base octale</i>	<i>base hexadécimale</i>	<i>représ. binaire</i>
	01770		
8100			
		0XAAAA	
			1001001001
			1100101011111110
10000			
	0234		

Exercice 3.4 :

Pour les constantes correctement définies, trouvez les types et les valeurs numériques décimales :

12332	23.4	345lu	34.5L	-1.0
0xeba	0123l	'\n'	1.23ul	-1.0e-1
0FE0	40000	40000u	70000u	1e1f
'0'	o	'\0'	0	'O'
67e0	\r	01001	0.0l	0XEUL

Exercice 3.5 :

Evaluer les expressions suivantes en supposant

a=20 b=5 c=-10 d=2 x=12 y=15

Notez chaque fois la valeur rendue comme résultat de l'expression et les valeurs des variables dont le contenu a changé.

- (1) `(5*X)+2*((3*B)+4)`
- (2) `(5*(X+2)*3)*(B+4)`
- (3) `A == (B=5)`
- (4) `A += (X+5)`
- (5) `A != (C *= (-D))`
- (6) `A *= C+(X-D)`
- (7) `A %= D++`
- (8) `A %= ++D`
- (9) `(X++)*(A+C)`
- (10) `A = X*(B<C)+Y*(B<C)`
- (11) `!(X-D+C) || D`
- (12) `A&&B || !0&&C&&!D`
- (13) `((A&&B) || (!0&&C)) &&!D`
- (14) `((A&&B) || !0) &&(C&&(!D))`

Exercice 3.6 :

Éliminer les parenthèses superflues dans les expressions de l'exercice 3.5.

Exercice 3.7 :

Essayez le programme suivant et modifiez-le de façon à ce qu'il affiche :

- * A^B ,
- * l'hypoténuse d'un triangle rectangle de côtés A et B,
- * la tangente de A en n'utilisant que les fonctions **sin** et **cos**,
- * la valeur arrondie (en moins) de A/B,
- * la valeur arrondie (en moins) à trois positions derrière la virgule de A/B.

```
#include <stdio.h>
main()
{
    double A;
    double B;
    double RES;
    /* Saisie de A et B */
    printf("Introduire la valeur pour A : ");
    scanf("%lf", &A);
    printf("Introduire la valeur pour B : ");
    scanf("%lf", &B);
    /* Calcul */
    RES = A*A;
    /* Affichage du résultat */
```

```
printf("Le carré de A est %f \n", RES);
/* Calcul */
RES = B*B;
/* Affichage du résultat */
printf("Le carré de B est %f \n", RES);
return (0);
}
```

Exercice 3.8 :

Soient les déclarations :

```
long  A = 15;
char  B = 'A'; /* code ASCII : 65 */
short C = 10;
```

Quels sont le type et la valeur de chacune des expressions :

- (1) $C + 3$
 - (2) $B + 1$
 - (3) $C + B$
 - (4) $3 * C + 2 * B$
 - (5) $2 * B + (A + 10) / C$
 - (6) $2 * B + (A + 10.0) / C$
-

IX) Solutions des exercices du Chapitre 3 : TYPES DE BASE, OPÉRATEURS ET EXPRESSIONS :

Exercice 3.1 :

	signed				unsigned							
No :	char	short	int	long	char	short	int	Long	float	double	long double	
1	!X!	X	X	X					X	X	X	
2		!X!	!X!	X					X	X	X	
3		X	X	X	!X!	X	X	X	X	X	X	
4									!X!	X	X	
5		!X!	!X!	X					X	X	X	
6											!X!	
7				X		!X!	!X!	X	X	X	X	
8				!X!					X	X	X	
9										!X!	X	
10				!X!				X		X	X	
11										!X!	X	
12										!X!	X	
13										!X!	X	

X : choix possible

!X! : meilleur choix

Exercice 3.2 :

Traduisez les déclarations suivantes en C, sachant que vous travaillerez dans les ensembles de nombres indiqués. Choisissez les types les plus économiques, sans perdre en précision.

Solution :

(1)	entier COMPTEUR	{0 ,..., 300}
	int COMPTEUR ;	
(2)	entier X,Y	{-120 ,..., 100}
	char X,Y ;	
(3)	entier MESURE	{-10 ,..., 10 ⁴ }
	int MESURE ;	
(4)	réel SURFACE1	{0.5 ,..., 150075}
	float SURFACE1 ;	6 positions significatives
(5)	réel SURFACE2	{-12 ,..., 1500750.5}
	double SURFACE2 ;	8 positions significatives
(6)	entier N1	{0 ,..., 2 ¹⁰ } = {0 ,..., 1024}
	int N1 ;	
(7)	entier N2	{-4 ⁷ ,..., 4 ⁷ } = {-16384 ,..., 16384}
	int N2 ;	
(8)	entier N3	{0 ,..., 32 ⁶ } = {0 ,..., 1 073 741 824}
	long N3 ;	
(9)	entier N4	{-128 ⁰ ,..., 128 ⁵ } = {-1 ,..., 3.4*10 ¹⁰ }
	double N4 ;	11 positions significatives

(10)	booléen TROUVE	{vrai, faux}
	int TROUVE ;	par convention

Exercice 3.3 :

base décimale	base octale	base hexadécimale	représentation binaire
1016	01770	0X3F8	111111000
8100	017644	0X1FA4	1111110100100
43690	0125252	0XAAAA	1010101010101010
585	01111	0X249	1001001001
51966	0145376	0XCAFE	1100101011111110
10000	023420	0X2710	10011100010000
156	0234	0X9C	10011100

Exercice 3.4 :

Donnée	Type (si correct)	Valeur / Erreur
12332	Int	12332
23.4	Double	23.4
345lu	-	déclaration correcte : 345ul
34.5L	long double	34.5
-1.0	Double	-1.0
0xeba	Int	$10+11*16+14*256 = 3770$
0123l	Long	$3+2*8+1*64 = 83$
'\n'	Char	val. dépend de la machine (ASCII : 10)
1.23ul	-	'unsigned' seulement pour des entiers
-1.0e-1	Double	-0.1
0FE0	-	0 => octal; 'FE' seulement pour hexa.
40000	Long	40000
40000u	unsigned int	40000
70000u	unsigned long	70000
1e1f	Float	10
'0'	Char	val. dépend de la machine (ASCII : 48)
O	-	apostrophes manquent
'\0'	Char	valeur en général zéro
0	Int	valeur zéro
'O'	Char	val. dépend de la machine (ASCII : 79)
67e0	Double	67
\r	-	apostrophes manquent
01001	Int	$1+0*8+0*64+1*512 = 513$
0.0l	Long double	0
0XEL	unsigned long	14

Exercice 3.5 :

Evaluer les expressions suivantes en supposant :
A=20, B=5, C=-10, D=2, X=12, Y=15

(1)	$(5*X)+2*((3*B)+4)$	-> 98	/
(2)	$(5*(X+2)*3)*(B+4)$	-> 1890	/
(3)	A == (B=5)	-> 0	B=5
(4)	A += (X+5)	-> 37	A=37
(5)	A != (C *= (-D))	-> 0	C=20
(6)	A *= C+(X-D)	-> 0	A=0
(7)	A %= D++	-> 0	D=3 A=0
(8)	A %= ++D	-> 2	D=3 A=2
(9)	(X++)*(A+C)	-> 120	X=13
(10)	A = X*(B<C)+Y*(B<C)	-> 0+15 = 15	A=15
(11)	!(X-D+C) D	-> !0 1 = 1	/
(12)	A&&B !0&&C&&!D	-> 1 1&&1&&0 = 1	/
(13)	((A&&B) (!0&&C))&&!D	-> (1 1)&&0 = 0	/
(14)	((A&&B) !0)&&(C&&!D)	-> (1 1)&&(1&&0) = 0	/

Exercice 3.6 :

(1)	$(5*X) + 2*((3*B)+4)$	$\Leftrightarrow 5*X + 2*(3*B+4)$
(2)	$(5*(X+2)*3)*(B+4)$	$\Leftrightarrow 5*(X+2)*3*(B+4)$
(3)	A == (B=5)	/
(4)	A += (X+5)	$\Leftrightarrow A += X+5$
(5)	A != (C *= (-D))	$\Leftrightarrow A != (C *= -D)$
(6)	A *= C + (X-D)	$\Leftrightarrow A *= C + X-D$
(7)	A %= D++	/
(8)	A %= ++D	/
(9)	(X++) * (A+C)	$\Leftrightarrow X++ * (A+C)$
(10)	A = X*(B<C) + Y*(B<C)	/
(11)	!(X-D+C) D	/
(12)	A&&B !0&&C&&!D	/
(13)	((A&&B) (!0&&C))&&!D	$\Leftrightarrow (A&&B !0&&C)&&!D$
(14)	((A&&B) !0)&&(C&&!D)	$\Leftrightarrow (A&&B !0)&&C&&!D$

Exercice 3.7 :

```
#include <stdio.h>
#include <math.h>
main()
{
    double A;
    double B;
    double RES;
    /* Saisie de A et B */
    printf("Introduire la valeur pour A : ");
    scanf("%lf", &A);
    printf("Introduire la valeur pour B : ");
    scanf("%lf", &B);
    /* Calcul */
    RES = pow(A,B);
    /* Affichage du résultat */
}
```

```

printf("A exposant B est %f \n", RES);
/* Calcul */
RES = sqrt(pow(A,2)+pow(B,2));
/* Affichage du résultat */
printf("L'hypoténuse du triangle rectangle est %f \n", RES);
/* Calcul */
RES = sin(A)/cos(A);
/* Affichage du résultat */
printf("La tangente de A est %f \n", RES);
/* Calcul */
RES = floor(A/B);
/* Affichage du résultat */
printf("La valeur arrondie en moins de A/B est %f \n", RES);
/* Calcul */
RES = floor(1000*(A/B))/1000;
/* Affichage du résultat */
printf("La valeur A/B arrondie à trois décimales : %f \n",
RES);
return (0);
}

```

Exercice 3.8 :

Soient les déclarations :

```

long A = 15;
char B = 'A';          /* code ASCII : 65 */
short C = 10;

```

	<u>Expression</u>	<u>Type</u>	<u>Valeur</u>
(1)	C + 3	int	13
(2)	B + 1	int	66
(3)	C + C	int	75
(4)	3 * C + 2 * B	int	160
(5)	2 * B + (A + 10) / C	long	132
(6)	2 * B + (A + 10.0) / C	double	132.5

Chapitre 4 : LIRE ET ÉCRIRE DES DONNÉES :

La bibliothèque standard `<stdio>` contient un ensemble de fonctions qui assurent la communication de la machine avec le monde extérieur. Dans ce chapitre, nous allons en discuter les plus importantes :

<code>printf()</code>	écriture formatée de données
<code>scanf()</code>	lecture formatée de données
<code>putchar()</code>	écriture d'un caractère
<code>getchar()</code>	lecture d'un caractère

I) Écriture formatée de données :

1) printf() :

La fonction `printf` est utilisée pour transférer du texte, des valeurs de variables ou des résultats d'expressions vers le fichier de sortie standard `stdout` (par défaut l'écran).

Écriture formatée en langage algorithmique :

`écrire` `<Expression1>`, `<Expression2>`, ...

Écriture formatée en C :

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

"**<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,
- * Les spécificateurs de format impliquent une conversion d'un nombre en chaîne de caractères. Ils sont encore appelés symboles de conversion.

Exemple 1 :

La suite d'instructions :

```
int A = 1234;  
int B = 567;  
printf("%i fois %i est %li\n", A, B, (long)A*B);
```

va afficher sur l'écran :

```
1234 fois 567 est 699678
```

Les arguments de **print :f** sont

- la partie format **"%i fois %i est %li"**
- la variable **A**
- la variable **B**
- l'expression **(long)A*B**

Le *1^{er}* Spécificateur (**%i**) indique que la valeur de A

Le 2^e Sera imprimée comme entier relatif ==> 1234
 Spécificateur (%i) indique que la valeur de B
 Sera imprimée comme entier relatif ==> 567
 Spécificateur (%li) indique que la valeur de
 Le 3^e (long)A*B sera imprimée comme entier relatif ==> 699678
 long

Exemple 2 :

La suite d'instructions :

```
char B = 'A';
printf("Le caractère %c a le code %i !\n", B, B);
```

va afficher sur l'écran :

```
Le caractère A a le code 65 !
```

La valeur de B est donc affichée sous deux formats différents :

```
% c comme caractère : A
% i comme entier relatif : 65
```

Spécificateurs de format pour printf :

<i>SYMBOLE</i>	<i>TYPE</i>	<i>IMPRESSION COMME</i>
% 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	Double	rationnel en notation décimale
% e	Double	rationnel en notation scientifique
% s	Char*	chaîne de caractères

a) Arguments du type long :

Les spécificateurs % d, % i, % u, % o, % x peuvent seulement représenter des valeurs du type **int** ou **unsigned int**. Une valeur trop grande pour être codée dans deux octets est coupée sans avertissement si nous utilisons % d.

Pour pouvoir traiter correctement les arguments du type **long**, il faut utiliser les spécificateurs % ld, % li, % lu, % lo, % lx.

Exemple :

```
long N = 1500000;
printf("%ld, %lx", N, N);    ==> 1500000, 16e360
printf("%x, %x", N);       ==> e360, 16
printf("%d, %d", N);       ==> -7328, 22
```

b) Arguments rationnels :

Les spécificateurs % f et % e peuvent être utilisés pour représenter des arguments du type **float** ou **double**. La mantisse des nombres représentés par % e contient exactement un chiffre (non nul) devant le point décimal. Cette représentation s'appelle la *notation scientifique* des rationnels.

Pour pouvoir traiter correctement les arguments du type **long double**, il faut utiliser les spécificateurs % Lf et % Le.

Exemple :

```
float N = 12.1234;
double M = 12.123456789;
long double P = 15.5;
```

<code>printf ("%f", N);</code>	<code>==> 12.123400</code>
<code>printf ("%f", M);</code>	<code>==> 12.123457</code>
<code>printf ("%e", N);</code>	<code>==> 1.212340e+01</code>
<code>printf ("%e", M);</code>	<code>==> 1.212346e+01</code>
<code>printf ("%Le", P);</code>	<code>==> 1.550000e+01</code>

c) Largeur minimale pour les entiers :

Pour les entiers, nous pouvons indiquer la *largeur minimale* de la valeur à afficher. Dans le champ ainsi réservé, les nombres sont justifiés à droite.

Exemples :

(_ <=> position libre)

<code>printf ("%4d", 123);</code>	<code>==> _ 123</code>
<code>printf ("%4d", 1234);</code>	<code>==> 1234</code>
<code>printf ("%4d", 12345);</code>	<code>==> 12345</code>
<code>printf ("%4u", 0);</code>	<code>==> ___ 0</code>
<code>printf ("%4X", 123);</code>	<code>==> ___ 7B</code>
<code>printf ("%4x", 123);</code>	<code>==> ___ 7b</code>

d) Largeur minimale et précision pour les rationnels :

Pour les rationnels, nous pouvons indiquer la *largeur minimale* de la valeur à afficher et la *précision* du nombre à afficher. La précision par défaut est fixée à six décimales. Les positions décimales sont arrondies à la valeur la plus proche.

Exemples :

<code>printf ("%f", 100.123);</code>	<code>==> 100.123000</code>
<code>printf ("%12f", 100.123);</code>	<code>==> ___100.123000</code>
<code>printf ("%2f", 100.123);</code>	<code>==> 100.12</code>
<code>printf ("%5.0f", 100.123);</code>	<code>==> ___100</code>
<code>printf ("%10.3f", 100.123);</code>	<code>==> ___100.123</code>
<code>printf ("%4f", 1.23456);</code>	<code>==> 1.2346</code>

-
- [Exercice 4.1](#)
-

II) Lecture formatée de données :

1) scanf() :

La fonction `scanf` est la fonction symétrique à `printf`; elle nous offre pratiquement les mêmes conversions que `printf`, mais en sens inverse.

Lecture formatée en langage algorithmique :

`lire <NomVariable1>, <NomVariable2>, ...`

Lecture formatée en C :

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

"<format>" : *format de lecture des données*

<AdrVar1>, ... : *adresses des variables auxquelles les données seront attribuées*

- * La fonction `scanf` reçoit ses données à partir du fichier d'entrée standard *stdin* (par défaut le clavier).
- * La chaîne de format détermine comment les données reçues doivent être interprétées.
- * Les données reçues correctement sont mémorisées successivement aux *adresses* indiquées par <AdrVar1>,

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

Exemple :

La suite d'instructions :

```
int JOUR, MOIS, ANNEE;  
scanf("%i %i %i", &JOUR, &MOIS, &ANNEE);
```

lit trois entiers relatifs, séparés par des espaces, tabulations ou interlignes. Les valeurs sont attribuées respectivement aux trois variables **JOUR**, **MOIS** et **ANNEE**.

* **scanf** retourne comme résultat le nombre de données correctement reçues (type **int**).

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 (c'est-à-dire : un *pointeur sur une variable* - voir chapitre 9 'Les pointeurs').

a) Le type long :

Si nous voulons lire une donnée du type **long**, nous devons utiliser les spécificateurs **%ld**, **%li**, **%lu**, **%lo**, **%lx**. (Sinon, le nombre est simplement coupé à la taille de **int**).

b) Le type double :

Si nous voulons lire une donnée du type **double**, nous devons utiliser les spécificateurs **%le** ou **%lf**.

c) Le type long double :

Si nous voulons lire une donnée du type **long double**, nous devons utiliser les spécificateurs **%Le** ou **%Lf**.



d) Indication de la largeur maximale :

Pour tous les spécificateurs, nous pouvons indiquer la *largeur maximale* du champ à évaluer pour une donnée. Les chiffres qui passent au-delà du champ défini sont attribués à la prochaine variable qui sera lue !

Exemple :

Soient les instructions :

```
int A,B;  
scanf("%4d %2d", &A, &B);
```

Si nous entrons le nombre **1234567**, nous obtiendrons les affectations suivantes :

```
A=1234  
B=56
```

le chiffre 7 sera gardé pour la prochaine instruction de lecture.

e) Les signes d'espacement :

Lors de l'entrée des données, une suite de signes d'espacement (espaces, tabulateurs, interlignes) est évaluée comme un seul espace. Dans la chaîne de format, les symboles **\t**, **\n**, **\r** ont le même effet qu'un simple espace.

Exemple :

Pour la suite d'instructions

```
int JOUR, MOIS, ANNEE;
scanf("%i %i %i", &JOUR, &MOIS, &ANNEE);
```

les entrées suivantes sont correctes et équivalentes :

```
12 4 1980
ou
12 004 1980
ou
12
4
1980
```

f) Formats 'spéciaux' :

Si la chaîne de format contient aussi d'autres caractères que des signes d'espacement, alors ces symboles doivent être introduits exactement dans l'ordre indiqué.

Exemple :

La suite d'instructions :

```
int JOUR, MOIS, ANNEE;
scanf("%i/%i/%i", &JOUR, &MOIS, &ANNEE);
```

Accepte les entrées :	rejette les entrées :
12/4/1980	12 4 1980
12/04/01980	12 /4 /1980

g) Nombre de valeurs lues :

Lors de l'évaluation des données, **scanf** s'arrête si la chaîne de format a été travaillée jusqu'à la fin ou si une donnée ne correspond pas au format indiqué. **scanf** retourne comme résultat le nombre d'arguments correctement reçus et affectés.

Exemple

La suite d'instructions

```
int JOUR, MOIS, ANNEE, RECU;
RECU = scanf("%i %i %i", &JOUR, &MOIS, &ANNEE);
```

réagit de la façon suivante (- valeur indéfinie) :

Introduit :		RECU	JOUR	MOIS	ANNEE
12 4 1980	==>	3	12	4	1980
12/4/1980	==>	1	12	-	-
12.4 1980	==>	1	12	-	-
12 4 19.80	==>	3	12	4	19

- [Exercice 4.2](#)

III) Écriture d'un caractère :

La commande

```
putchar('a');
```

transfère le caractère *a* vers le fichier standard de sortie *stdout*. Les arguments de la fonction **putchar** sont ou bien des caractères (c'est-à-dire des nombres entiers entre 0 et 255) ou bien le symbole **EOF** (*End Of File*).

EOF est une constante définie dans `<stdio>` qui marque la fin d'un fichier. La commande **putchar(EOF)**; est utilisée dans le cas où *stdout* est dévié vers un fichier.

1) Type de l'argument :

Pour ne pas être confondue avec un caractère, la constante **EOF** doit nécessairement avoir une valeur qui sort du domaine des caractères (en général **EOF** a la valeur -1). Ainsi, les arguments de **putchar** sont par définition du type **int** et toutes les valeurs traitées par **putchar** (même celles du type **char**) sont d'abord converties en **int**.

Exemples :

```
char A = 225;
char B = '\a';
int C = '\a';
putchar('x'); /* afficher la lettre x */
putchar('?'); /* afficher le symbole ? */
putchar('\n'); /* retour à la ligne */
putchar(65); /* afficher le symbole avec
             /* le code 65 (ASCII : 'A') */
putchar(A); /* afficher la lettre avec
            /* le code 225 (ASCII : '\a') */
putchar(B); /* beep sonore */
putchar(C); /* beep sonore */
putchar(EOF); /* marquer la fin du fichier */
```

IV) Lecture d'un caractère :

Une fonction plus souvent utilisée que **putchar** est la fonction **getchar**, qui lit le prochain caractère du fichier d'entrée standard *stdin*.

1) Type du résultat :

Les valeurs retournées par **getchar** sont ou bien des caractères (0 - 255) ou bien le symbole **EOF**. Comme la valeur du symbole **EOF** sort du domaine des caractères, le type résultat de **getchar** est **int**. En général, **getchar** est utilisé dans une affectation :

```
int C;
C = getchar();
```

getchar lit les données de la zone tampon de *stdin* et fournit les données seulement après confirmation par 'Enter'. La bibliothèque `<conio>` contient une fonction du nom **getch** qui fournit immédiatement le prochain caractère entré au clavier.

La fonction **getch** n'est pas compatible avec ANSI-C et elle peut seulement être utilisée sous MS-DOS.

-
- [Exercice 4.3](#)
-

V) Exercices d'application :

Exercice 4.1 :

```
#include <stdio.h>
main()
{
    int N=10, P=5, Q=10, R;
    char C='S';

    N = 5; P = 2;
    Q = N++ > P || P++ != 3;
    printf ("C : N=%d P=%d Q=%d\n", N, P, Q);

    N = 5; P = 2;
    Q = N++ < P || P++ != 3;
    printf ("D : N=%d P=%d Q=%d\n", N, P, Q);

    N = 5; P = 2;
    Q = ++N == 3 && ++P == 3;
    printf ("E : N=%d P=%d Q=%d\n", N, P, Q);

    N=5; P=2;
    Q = ++N == 6 && ++P == 3;
    printf ("F : N=%d P=%d Q=%d\n", N, P, Q);

    N=C;
    printf ("G : %c %c\n", C, N);
    printf ("H : %d %d\n", C, N);
    printf ("I : %x %x\n", C, N);
    return (0);
}
```

- Sans utiliser l'ordinateur, trouvez et notez les résultats du programme ci-dessus.
 - Vérifiez vos résultats à l'aide de l'ordinateur.
-

Exercice 4.2 :

En vous référant aux exemples du chapitre 4.2, écrivez un programme qui lit la date du clavier et écrit les données ainsi que le nombre de données correctement reçues sur l'écran.

Exemple :

```
Introduisez la date (jour mois année) : 11 11 1991

données reçues : 3
jour   : 11
mois   : 11
année  : 1991
```

* Testez les réactions du programme à vos entrées. Essayez d'introduire des nombres de différents formats et différentes grandeurs.

* Changez la partie format du programme de façon à séparer les différentes données par le symbole '-' .

Exercice 4.3 :

Ecrire un programme qui lit un caractère au clavier et affiche le caractère ainsi que son code numérique :

a) en employant **getchar** et **printf**,

b) en employant **getch** et **printf**.

Exercice 4.4 :

Ecrire un programme qui permute et affiche les valeurs de trois variables A, B, C de type entier qui sont entrées au clavier :

$$A \Rightarrow B, B \Rightarrow C, C \Rightarrow A$$

Exercice 4.5 :

Ecrire un programme qui affiche le quotient et le reste de la division entière de deux nombres entiers entrés au clavier ainsi que le quotient rationnel de ces nombres.

Exercice 4.6 :

Ecrire un programme qui affiche la résistance équivalente à trois résistances R1, R2, R3 (type **double**),

- si les résistances sont branchées en série :

$$R_{\text{sér}} = R1 + R2 + R3$$

- si les résistances sont branchées en parallèle :

$$R_{\text{par}} = \frac{R1 \cdot R2 \cdot R3}{R1 \cdot R2 + R1 \cdot R3 + R2 \cdot R3}$$

Exercice 4.7 :

Ecrire un programme qui calcule et affiche l'aire d'un triangle dont il faut entrer les longueurs des trois côtés. Utilisez la formule :

$$S^2 = P(P-A)(P-B)(P-C)$$

où A, B, C sont les longueurs des trois côtés (type **int**) et P le demi-périmètre du triangle.

Exercice 4.8 :

Ecrire un programme qui calcule la somme de quatre nombres du type **int** entrés au clavier,

a) en se servant de 5 variables (mémoire des valeurs entrées),

b) en se servant de 2 variables (perte des valeurs entrées).

Exercice 4.9 :

a) Ecrire un programme qui calcule le prix brut (type **double**) d'un article à partir du prix net (type **int**) et du pourcentage de TVA (type **int**) à ajouter. Utilisez la formule suivante en faisant attention aux priorités et aux conversions automatiques de type :

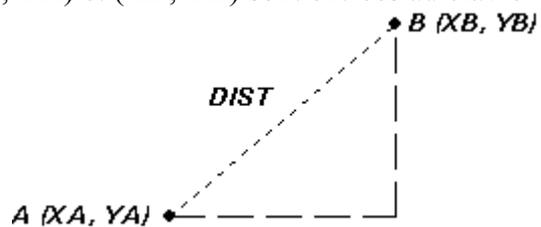
$$PBRUT = PNET + PNET \cdot \frac{TVA}{100}$$

b) Ecrire un programme qui calcule le prix net d'un article (type **double**) à partir du prix brut (type **double**) et du pourcentage de TVA (type **int**) qui a été ajoutée.

(Déduisez la formule du calcul de celle indiquée ci-dessus)

Exercice 4.10 :

Ecrire un programme qui calcule et affiche la distance *DIST* (type **double**) entre deux points A et B du plan dont les coordonnées (XA, YA) et (XB, YB) sont entrées au clavier comme entiers.



VI) Solutions des exercices du Chapitre 4 : LIRE ET ÉCRIRE DES DONNÉES :

Exercice 4.1 :

Voici les résultats du programme :

```
C : n=6 p=2 q=1
D : n=6 p=3 q=1
E : n=6 p=2 q=0
F : n=6 p=3 q=1
G : S S
H : 83 83
I : 53 53
```

Exercice 4.2 :

```
#include <stdio.h>
main()
{
    int JOUR, MOIS, ANNEE, RECU;
    printf("Introduisez la date (JOUR, MOIS, ANNÉE) : ");
    RECU=scanf("%i %i %i", &JOUR, &MOIS, &ANNEE);
    printf("\ndonnées reçues : %i\njour   : %i\nmois   :
           %i\nannee  : %i\n", RECU, JOUR, MOIS, ANNEE);
    return (0);
}
```

Changez la partie format du programme de façon à séparer les différentes données par le symbole '-'

Solution :

```
. . .
RECU=scanf("%i-%i-%i", &JOUR, &MOIS, &ANNEE);
. . .
```

Exercice 4.3 :

Ecrire un programme qui lit un caractère au clavier et affiche le caractère ainsi que son code numérique :

a) en employant **getchar** et **printf**,

```
#include <stdio.h>
main()
{
    int C;
    printf("Introduire un caractère suivi de 'Enter'\n");
    C = getchar();
    printf("Le caractère %c a le code ASCII %d\n", C, C);
    return (0);
}
```

b) en employant **getch** et **printf**.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int C;
```

```

printf("Introduire un caractère \n");
C = getch();
printf("Le caractère %c a le code ASCII %d\n", C, C);
return (0);
}

```

Exercice 4.4 :

Solution :

```

#include <stdio.h>
main()
{
    int A, B, C, AIDE;
    printf("Introduisez trois nombres (A, B, C) : ");
    scanf("%i %i %i", &A, &B, &C);
    /* Affichage à l'aide de tabulations */
    printf("A = %i\tB = %i\tC = %i\n", A, B, C);
    AIDE=A;
    A=C;
    C=B;
    B=AIDE;
    printf("A = %i\tB = %i\tC = %i\n", A, B, C);
    return (0);
}

```

Exercice 4.5 :

```

#include <stdio.h>
main()
{
    int A, B;
    printf("Introduisez deux nombres entiers : ");
    scanf("%i %i", &A, &B);
    printf("Division entiere      : %i\n", A/B);
    printf("Reste                  : %i\n", A%B);
    printf("Quotient rationnel    : %f\n", (float)A/B);
    return (0);
}

```

Remarque :

Conversion des types :

Pour obtenir un résultat rationnel, il faut qu'au moins l'une des deux opérands soit d'un type rationnel. Si nous convertissons la valeur de l'opérande A en **float** à l'aide de l'opérateur de conversion forcée, alors la valeur de la deuxième opérande B est convertie automatiquement en **float**. Le résultat de la division est du type **float** et doit être représenté par le spécificateur de format **%f** (ou **%e**).

Remarquez bien que la conversion forcée concerne uniquement la valeur de la variable A ! La valeur de B est convertie automatiquement.

D'autre part, si nous écrivions :

```
(float) (A/B)
```

nous n'obtiendrions pas le résultat désiré, parce que la conversion se ferait seulement *après* la division !

Exercice 4.6 :

```
#include <stdio.h>
main()
{
    double R1, R2, R3, RRES;
    printf("Introduisez les valeurs pour R1, R2 et R3 : ");
    scanf("%lf %lf %lf", &R1, &R2, &R3);
    RRES=R1+R2+R3;
    printf("Résistance résultante sérielle      : %f\n", RRES);
    RRES=(R1*R2*R3)/(R1*R2+R1*R3+R2*R3);
    printf("Résistance résultante parallèle    : %f\n", RRES);
    return (0);
}
```

En affichant immédiatement le résultat du calcul, nous n'avons pas besoin de la variable d'aide RRES :

```
#include <stdio.h>
main()
{
    double R1, R2, R3;
    printf("Introduisez les valeurs pour R1, R2 et R3 : ");
    scanf("%lf %lf %lf", &R1, &R2, &R3);
    printf("Résistance résultante sérielle      : %f\n",
           R1+R2+R3);
    printf("Résistance résultante parallèle    : %f\n",
           (R1*R2*R3)/(R1*R2+R1*R3+R2*R3));
    return (0);
}
```

Exercice 4.7 :

```
#include <stdio.h>
#include <math.h>
main()
{
    /* Pour ne pas perdre de précision lors de la division, */
    /* déclarons P comme rationnel. */
    int A, B, C;
    double P; /* ou bien : float P; */
    printf("Introduisez les valeurs pour A, B et C : ");
    scanf("%i %i %i", &A, &B, &C);
    /* En forçant la conversion de A, les autres opérandes */
    /* sont converties automatiquement. */
    P=((double)A+B+C)/2;
    printf("Surface du triangle S = %f\n",
           sqrt(P*(P-A)*(P-B)*(P-C)));
    return (0);
}
```

Exercice 4.8 :

a) en se servant de 5 variables (mémorisation des valeurs entrées)

```
#include <stdio.h>
main()
{
```

```

/* Pour être sûrs de ne pas dépasser le domaine de la */
/* variable, nous choisissons le type long pour la somme. */
int A, B, C, D;
long SOM;
printf("Entrez le premier nombre : ");
scanf("%d", &A);
printf("Entrez le deuxième nombre : ");
scanf("%d", &B);
printf("Entrez le troisième nombre : ");
scanf("%d", &C);
printf("Entrez le quatrième nombre : ");
scanf("%d", &D);
SOM = (long)A+B+C+D;
printf(" %d + %d + %d + %d = %ld\n", A, B, C, D, SOM);
return (0);
}

```

b) en se servant de 2 variables (perte des valeurs entrées)

```

main()
{
/* Pour être sûrs de ne pas dépasser le domaine de la */
/* variable, nous choisissons le type long pour la somme. */
int A;
long SOM;
SOM = 0;
printf("Entrez le premier nombre : ");
scanf("%d", &A);
SOM+=A;
printf("Entrez le deuxième nombre : ");
scanf("%d", &A);
SOM+=A;
printf("Entrez le troisième nombre : ");
scanf("%d", &A);
SOM+=A;
printf("Entrez le quatrième nombre : ");
scanf("%d", &A);
SOM+=A;
printf("La somme des nombres entrés est %ld\n", SOM);
return (0);
}

```

Exercice 4.9 :

a)

```

#include <stdio.h>
main()
{
int PNET, TVA;
double PBRUT;
printf("Entrez le prix net de l'article : ");
scanf("%d", &PNET);
printf("Entrez le taux de la TVA (en %) : ");
scanf("%d", &TVA);
PBRUT = PNET+(double)PNET*TVA/100;
printf("Le prix brut est %.21f Francs\n", PBRUT);
}

```

```

    return (0);
}

```

Remarque :

Conversion des types et priorités :

Lors du calcul de PBRUT (type **double**), nous divisons le produit du PNET (type **int**) et de la TVA (type **int**) par 100. Pour ne pas perdre de la précision lors de la division et pour éviter un débordement du domaine lors du calcul du produit, il faut forcer l'utilisation du type **double**.

En utilisant l'opérateur de la conversion forcée, il faut respecter la suite de l'évaluation de l'expression. Il ne suffirait donc pas d'écrire :

$$\text{PBRUT} = (\text{double})\text{PNET} + \text{PNET} * \text{TVA} / 100 ;$$

Parce que la multiplication et la division ont la priorité sur l'addition; la conversion se ferait trop tard, c'est-à-dire : à la fin du calcul.

Il existe plusieurs possibilités de résoudre ce problème :

- utiliser la conversion (**double**) lors des premières opérations effectuées :

$$\text{PBRUT} = \text{PNET} + (\text{double})\text{PNET} * \text{TVA} / 100 ;$$

- utiliser une constante du type **double** lors de la division :

$$\text{PBRUT} = \text{PNET} + \text{PNET} * \text{TVA} / 100.0 ;$$

- déclarer PNET et/ou TVA comme **double** :

$$\text{double PNET, TVA, PBRUT} ;$$

suite ...

b)

```

#include <stdio.h>
main()
{
    int  TVA;
    double PNET, PBRUT; /* donnée et résultat du type double */
    printf("Entrez le prix brut de l'article : ");
    scanf("%lf", &PBRUT);
    printf("Entrez le taux de la TVA (en %) : ");
    scanf("%d", &TVA);
    /* Calcul: Ici, pas de forçage de type nécessaire */
    PNET = PBRUT*100/(100+TVA);
    printf("Le prix net est %.2lf Francs\n", PNET);
    return (0);
}

```

Remarque :

Conversion de types et priorités :

Ici, PNET et PBRUT sont du type **double**. Comme l'évaluation des opérateurs binaires * et / se fait en passant de gauche à droite, la valeur 100 est d'abord convertie en **double** (valeur 100.0) pour être multipliée avec PBRUT. La première opérande de la division est donc du type **double** et le résultat de l'addition est automatiquement converti en **double** avant la division.

Un simple remède aux confusions serait ici l'utilisation de la constante 100.0 à la place de 100 :

$$\text{PNET} = \text{PBRUT} * 100.0 / (100.0 + \text{TVA}) ;$$

Exercice 4.10 :

```

#include <stdio.h>
#include <math.h>
main()
{
    int  XA, YA, XB, YB;
    double DIST;

```

```
/* Attention : La chaîne de format que nous utilisons */
/* s'attend à ce que les données soient séparées par */
/* une virgule lors de l'entrée. */
printf("Entrez les coordonnées du point A :  XA,YA  ");
scanf("%d,%d", &XA, &YA);
printf("Entrez les coordonnées du point B :  XB,YB  ");
scanf("%d,%d", &XB, &YB);
DIST=sqrt(pow(XA-XB,2)+pow(YA-YB,2));
printf("La distance entre A(%d,% d) et B(%d, %d) est
%.2f\n",XA, YA, XB, YB, DIST);
return (0);
}
```

Chapitre 5 : LA STRUCTURE ALTERNATIVE :

Les structures de contrôle définissent la suite dans laquelle les instructions sont effectuées. Dans ce chapitre, nous allons voir comment les instructions de sélection connues fonctionnent en C et nous allons faire connaissance d'un couple d'opérateurs spécial qui nous permet de choisir entre deux valeurs à l'intérieur d'une expression.

Constatons déjà que la particularité la plus importante des instructions de contrôle en C est le fait que les 'conditions' en C peuvent être des *expressions quelconques qui fournissent un résultat numérique*. La valeur zéro correspond à la valeur logique faux et toute valeur différente de zéro est considérée comme vrai.

I) if – else :

La structure alternative en langage algorithmique :

```
si (<expression logique>)  
  alors  
    <bloc d'instructions 1>  
  sinon  
    <bloc d'instructions 2>  
  fsi
```

- * 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

La structure alternative en C :

```
if ( <expression> )  
  <bloc d'instructions 1>  
else  
  <bloc d'instructions 2>
```

- * 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, une seule instruction terminée par un point virgule.

Exemple 1 :

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

Exemple 2 :

```
if (EGAL)  
  printf("A est égal à B\n");  
else  
  printf("A est différent de B\n");
```

Exemple 3 :

```

if (A-B) printf("A est différent de B\n");
else printf("A est égal à B\n");

```

Exemple 4 :

```

if (A > B)
{
    AIDE = A;
    A = C;
    C = AIDE;
}
else
{
    AIDE = B;
    B = C;
    C = AIDE;
}

```

Remarque avancée :

Commentaire de **Francois Donato** au sujet de [l'usage d'accolades](#).

Je crois qu'il est de bonne pratique de forcer les accolades "{ }" dans un if (while, do ...while, for ...) même pour une seule instruction.

Exemple :

Je n'écris jamais (qu'il y est ou pas de else est sans importance) :

```

if (condition)
    printf("Message");

```

mais toujours :

```

if (condition)
{
    printf("Message");
}

```

La raison est que lors du développement de gros projet (où plusieurs personnes peuvent être impliquées) il se peut que le code soit modifié par une autre personne plus tard dans le cadre du projet (ou d'une nouvelle version, ou d'un bug...). Il suffit d'un moment d'inattention et une erreur est introduite.

Ainsi si vous avez :

```

if (condition)
    a = b; /* une expression quelconque */

```

Si vous voulez une nouvelle fonctionnalité, une erreur fréquente qui peut arriver (et qui arrive!) est :

```

if (condition)
    a = b; /* une expression quelconque */
    a += fct() /* l'ajout d'une nouvelle valeur mais dans
tout les cas !!! */

```

Ainsi si les accolades sont forcées il n'y a pas d'erreur potentielle.

(Le seul cas acceptable est celui-ci : if (condition) printf(...); car sur une seule ligne.)

Ceci n'est qu'une opinion, mais si on fait de la programmation défensive forcer les accolades est de bonne pratique.

II) if sans else :

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

if sans else :

```

if ( <expression> )

```

<bloc d'instructions>

Attention ! :

Comme la partie **else** est optionnelle, les expressions contenant plusieurs structures **if** et **if - else** peuvent mener à des confusions.

Exemple :

L'expression suivante peut être interprétée de deux façons :

```
if (N>0)
  if (A>B)
    MAX=A;
  else
    MAX=B;
if (N>0)
  if (A>B)
    MAX=A;
else
  MAX=B;
```



Pour N=0, A=1 et B=2,

* dans la première interprétation, MAX reste inchangé,

* dans la deuxième interprétation, MAX obtiendrait la valeur de B.

Sans règle supplémentaire, le résultat de cette expression serait donc imprévisible.

Convention :

En C une partie **else** est toujours liée au dernier **if** qui ne possède pas de partie **else**.

Dans notre exemple, C utiliserait donc la première interprétation.

Solution :

Pour éviter des confusions et pour forcer une certaine interprétation d'une expression, il est recommandé d'utiliser des accolades { } .

Exemple :

Pour forcer la deuxième interprétation de l'expression ci-dessus, nous pouvons écrire :

```
if (N>0)
{
  if (A>B)
    MAX=A;
}
else
  MAX=B;
```

-
- [Exercice 5.1](#)
-

III) if - else if - ... - else :

En combinant plusieurs structures **if - else** en une expression nous obtenons une structure qui est très courante pour prendre des décisions entre plusieurs alternatives :

if - else - ... - else :

```
if ( <expr1> )
  <bloc1>
else if (<expr2>)
  <bloc2>
```

```

else if (<expr3>)
    <bloc3>
else if (<exprN>)
    <blocN>
else <blocN+1>

```

Les expressions <expr1> ... <exprN> sont évaluées du haut vers le bas jusqu'à ce que l'une d'elles soit différente de zéro. Le bloc d'instructions qui lui est lié est alors exécuté et le traitement de la commande est terminé.

Exemple :

```

#include <stdio.h>
main()
{
    int A,B;
    printf("Entrez deux nombres entiers :");
    scanf("%i %i", &A, &B);
    if (A > B)
        printf("%i est plus grand que %i\n", A, B);
    else if (A < B)
        printf("%i est plus petit que %i\n", A, B);
    else
        printf("%i est égal à %i\n", A, B);
    return (0);
}

```

La dernière partie **else** traite le cas où aucune des conditions n'a été remplie. Elle est optionnelle, mais elle peut être utilisée très confortablement pour détecter des erreurs.

Exemple :

```

...
printf("Continuer (O)ui / (N)on ?");
getchar(C);
if (C=='O')
    {
        ...
    }
else if (C=='N')
    printf("Au revoir ...\n");
else
    printf("\aErreur d'entrée !\n");
...

```

-
- [Exercice 5.2](#)
-

IV) Les opérateurs conditionnels :

Le langage C possède une paire d'opérateurs un peu exotiques qui peut être utilisée comme alternative à **if - else** et qui a l'avantage de pouvoir être intégrée dans une expression :

Les opérateurs conditionnels :

```
<expr1> ? <expr2> : <expr3>
```

* Si <expr1> fournit une valeur différente de zéro, alors la valeur de <expr2> est fournie comme résultat,

* Si <expr1> fournit la valeur zéro, alors la valeur de <expr3> est fournie comme résultat.

Exemple :

La suite d'instructions

```
if (A>B)
    MAX=A;
else
    MAX=B;
```

peut être remplacée par :

```
MAX = (A > B) ? A : B;
```



Employés de façon irréfléchis, les opérateurs conditionnels peuvent nuire à la lisibilité d'un programme, mais si on les utilise avec précaution, ils fournissent des solutions très élégantes :

Exemple :

```
printf("Vous avez %i carte%c \n", N, (N==1) ? ' ' : 's');
```

! Les *règles de conversion de types* s'appliquent aussi aux opérateurs conditionnels ? : Ainsi, pour un entier N du type **int** et un rationnel F du type **float**, l'expression :

```
(N>0) ? N : F
```

va *toujours* fournir un résultat du type **float**, que N soit plus grand ou plus petit que zéro !

V) Exercices d'application :

Exercice 5.1 :

Considérez la séquence d'instructions suivante :

```
if (A>B) printf ("premier choix \n"); else
if (A>10) printf ("deuxième choix \n");
if (B<10) printf ("troisième choix \n");
else printf ("quatrième choix \n");
```

- a) Copiez la séquence d'instructions en utilisant des tabulateurs pour marquer les blocs **if - else** appartenant ensemble.
 - b) Déterminez les réponses du programme pour chacun des couples de nombres suivants et vérifiez à l'aide de l'ordinateur.
 - A=10 et B=5
 - A=5 et B=5
 - A=5 et B=10
 - A=10 et B=10
 - A=20 et B=10
 - A=20 et B=20
-

Exercice 5.2 :

Considérez la séquence d'instructions suivante :

```
if (A>B)
if (A>10)
printf ("premier choix \n"); else if (B<10)
printf ("deuxième choix \n"); else
if (A==B) printf ("troisième choix \n");
else printf ("quatrième choix \n");
```

- a) Copiez la séquence d'instructions en utilisant des tabulateurs pour marquer les blocs **if - else** appartenant ensemble.
 - b) Pour quelles valeurs de A et B obtient-on les résultats : *premier choix*, *deuxième choix*, ... sur l'écran ?
 - c) Pour quelles valeurs de A et B n'obtient-on pas de réponse sur l'écran ?
 - d) Notez vos réponses et choisissez vous-mêmes des valeurs pour A et B pour les vérifier l'aide de l'ordinateur.
-

Exercice 5.3 :

Ecrivez un programme qui lit trois valeurs entières (A, B et C) au clavier et qui affiche la plus grande des trois valeurs, en utilisant :

- a) **if - else** et une variable d'aide MAX
 - b) **if - else if - ... - else** sans variable d'aide
 - c) les opérateurs conditionnels et une variable d'aide MAX
 - d) les opérateurs conditionnels sans variable d'aide
-

Exercice 5.4 :

Ecrivez un programme qui lit trois valeurs entières (A, B et C) au clavier. Triez les valeurs A, B et C par échanges successifs de manière à obtenir :

```
val(A) val(B) val(C)
Affichez les trois valeurs.
```

Exercice 5.5 :

Ecrivez un programme qui lit deux valeurs entières (A et B) au clavier et qui affiche le signe du produit de A et B sans faire la multiplication.

Exercice 5.6 :

Ecrivez un programme qui lit deux valeurs entières (A et B) au clavier et qui affiche le signe de la somme de A et B sans faire l'addition. Utilisez la fonction **fabs** de la bibliothèque `<math>`.

Exercice 5.7 :

Ecrivez un programme qui calcule les solutions réelles d'une équation du second degré $ax^2+bx+c = 0$ en discutant la formule :

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Utilisez une variable d'aide **D** pour la valeur du discriminant b^2-4ac et décidez à l'aide de **D**, si l'équation a une, deux ou aucune solution réelle. Utilisez des variables du type **int** pour A, B et C.

Considérez aussi les cas où l'utilisateur entre des valeurs nulles pour A; pour A et B; pour A, B et C. Affichez les résultats et les messages nécessaires sur l'écran.

VI) Solutions des exercices du Chapitre 5 : LA STRUCTURE ALTERNATIVE

Exercice 5.1 :

```
if (A>B) printf ("premier choix \n"); else
if (A>10) printf ("deuxième choix \n");
if (B<10) printf ("troisième choix \n");
else printf ("quatrième choix \n");
```

a) Copiez la séquence d'instructions en utilisant des tabulateurs pour marquer les blocs **if - else** appartenant ensemble.

```
if (A>B)
    printf ("premier choix \n");
else
    if (A>10)
        printf ("deuxième choix \n");
if (B<10)
    printf ("troisième choix \n");
else
    printf ("quatrième choix \n");
```

b) Déterminez les réponses du programme pour chacun des couples de nombres suivants et vérifiez à l'aide de l'ordinateur.

A=10 et B=5 :	premier choix
	troisième choix
A=5 et B=5 :	troisième choix
A=5 et B=10 :	quatrième choix
A=10 et B=10 :	quatrième choix
A=20 et B=10 :	premier choix
	quatrième choix
A=20 et B=20 :	deuxième choix
	quatrième choix

Exercice 5.2 :

Considérez la séquence d'instructions suivante :

```
if (A>B)
if (A>10)
printf ("premier choix \n"); else if (B<10)
printf ("deuxième choix \n"); else
if (A==B) printf ("troisième choix \n");
else printf ("quatrième choix \n");
```

a) Copiez la séquence d'instructions en utilisant des tabulateurs pour marquer les blocs **if - else** appartenant ensemble.

```
if (A>B)
    if (A>10)
        printf ("premier choix \n");
    else if (B<10)
        printf ("deuxième choix \n");
    else if (A==B)
        printf ("troisième choix \n");
    else
        printf ("quatrième choix \n");
```

b) Le résultat :

"premier choix"	apparaît pour (A>B) et (A>10)	
"deuxième choix"	apparaît pour (10A>B)	
"troisième choix"	apparaît pour (10A>B10) et (A=B) 10>10 impossible A>B et A=B impossible	=> "troisième choix" n'apparaît jamais
"quatrième choix"	apparaît pour (10A>B10) et (AB) 10>10 impossible	=> "quatrième choix" n'apparaît jamais

c)

On n'obtient pas de réponses pour (AB). Si (A>B) alors la construction **if - else if - ... - else** garantit que toutes les combinaisons sont traitées et fournissent un résultat.

Exercice 5.3 :

a) **if - else** et une variable d'aide MAX

```
#include <stdio.h>
main()
{
    int A, B, C;
    int MAX;
    printf("Introduisez trois nombres entiers :");
    scanf("%i %i %i", &A, &B, &C);
    if (A>B)
        MAX=A;
    else
        MAX=B;
    if (C>MAX)
        MAX=C;
    printf("La valeur maximale est %i\n", MAX);
    return (0);
}
```

b) **if - else if - ... - else** sans variable d'aide

```
int A, B, C;
printf("Introduisez trois nombres entiers :");
scanf("%i %i %i", &A, &B, &C);
printf("La valeur maximale est ");
if (A>B && A>C)
    printf("%i\n",A);
else if (B>C)
    printf("%i\n",B);
else
    printf("%i\n",C);
```

c) opérateurs conditionnels et une variable d'aide MAX

```
int A, B, C;
int MAX;
printf("Introduisez trois nombres entiers :");
scanf("%i %i %i", &A, &B, &C);
MAX = (A>B) ? A : B;
```

```
MAX = (MAX>C) ? MAX : C;
printf("La valeur maximale est %i\n", MAX);
```

d) opérateurs conditionnels sans variable d'aide

```
int A, B, C;
printf("Introduisez trois nombres entiers :");
scanf("%i %i %i", &A, &B, &C);
printf("La valeur maximale est %i\n",
      (A>((B>C)?B :C)) ? A : ((B>C)?B :C));
```

Exercice 5.4 :

```
#include <stdio.h>
main()
{
/* Tri par ordre décroissant de trois entiers
   en échangeant les valeurs
*/
int A, B, C, AIDE;
printf("Introduisez trois nombres entiers :");
scanf("%i %i %i", &A, &B, &C);
printf("Avant le tri : \tA = %i\tB = %i\tC = %i\n", A, B, C);
/* Valeur maximale -> A */
if (A<B)
{
    AIDE = A;
    A = B;
    B = AIDE;
}
if (A<C)
{
    AIDE = A;
    A = C;
    C = AIDE;
}
/* trier B et C */
if (B<C)
{
    AIDE = B;
    B = C;
    C = AIDE;
}
printf("Après le tri : \tA = %i\tB = %i\tC = %i\n", A, B, C);
return (0);
}
```

Exercice 5.5 :

```
#include <stdio.h>
main()
{
/* Afficher le signe du produit de deux entiers sans
   faire la multiplication
*/
int A, B;
```

```

printf("Introduisez deux nombres entiers :");
scanf("%i %i", &A, &B);
if ((A>0 && B>0) || (A<0 && B<0))
    printf("Le signe du produit %i * %i est positif\n", A, B);
else if ((A<0 && B>0) || (A>0 && B<0))
    printf("Le signe du produit %i * %i est négatif\n", A, B);
else
    printf("Le produit %i * %i est zéro\n", A, B);
return (0);
}

```

Exercice 5.6 :

```

#include <stdio.h>
#include <math.h>
main()
{
/* Afficher le signe de la somme de deux entiers sans
   faire l'addition
*/
int A, B;
printf("Introduisez deux nombres entiers :");
scanf("%i %i", &A, &B);
if ((A>0 && B>0) || (A<0 && B>0 && fabs(A)<fabs(B))
    || (A>0 && B<0 && fabs(A)>fabs(B)))
    printf("Le signe de la somme %i + %i est positif\n",A,B);
else if ((A<0 && B<0) || (A<0 && B>0 && fabs(A)>fabs(B))
    || (A>0 && B<0 && fabs(A)<fabs(B)))
    printf("Le signe de la somme %i + %i est négatif\n",A,B);
else
    printf("La somme %i + %i est zéro\n", A, B);
return (0);
}

```

Exercice 5.7 :

```

#include <stdio.h>
#include <math.h>
main()
{
/* Calcul des solutions réelles d'une équation du second
degré */
int A, B, C;
double D; /* Discriminant */
printf("Calcul des solutions réelles d'une équation du second
\n");
printf("degré de la forme  ax^2 + bx + c = 0 \n\n");
printf("Introduisez les valeurs pour a, b, et c : ");
scanf("%i %i %i", &A, &B, &C);
/* Calcul du discriminant b^2-4ac */
D = pow(B,2) - 4.0*A*C;

/* Distinction des différents cas */

```

```

if (A==0 && B==0 && C==0) /* 0x = 0 */
    printf("Tout réel est une solution de cette
équation.\n");
else if (A==0 && B==0) /* Contradiction : c # 0 et c = 0 */
    printf("Cette équation ne possède pas de solutions.\n");
else if (A==0) /* bx + c = 0 */
    {
        printf("La solution de cette équation du premier degré
est :\n");
        printf(" x = %.4f\n", (double)C/B);
    }
else if (D<0) /* b^2-4ac < 0 */
    printf("Cette équation n'a pas de solutions réelles.\n");
else if (D==0) /* b^2-4ac = 0 */
    {
        printf("Cette équation a une seule solution réelle :\n");
        printf(" x = %.4f\n", (double)-B/(2*A));
    }
else /* b^2-4ac > 0 */
    {
        printf("Les solutions réelles de cette équation sont
:\n");
        printf(" x1 = %.4f\n", (-B+sqrt(D))/(2*A));
        printf(" x2 = %.4f\n", (-B-sqrt(D))/(2*A));
    }
return (0);
}

```

Chapitre 6 : LA STRUCTURE REPETITIVE :

En C, nous disposons de trois structures qui nous permettent la définition de boucles conditionnelles :

- 1) la structure : **while**
- 2) la structure : **do - while**
- 3) la structure : **for**

Théoriquement, ces structures sont interchangeables, c'est-à-dire il serait possible de programmer toutes sortes de boucles conditionnelles en n'utilisant qu'une seule des trois structures. Comme en Pascal, il est quand même absolument recommandé de choisir toujours la structure la mieux adaptée au cas actuel (voir chapitre [6.4.](#)).

I) while :

La structure **while** correspond tout à fait à la structure tant que du langage algorithmique. (Si on néglige le fait qu'en C les conditions peuvent être formulées à l'aide d'expressions numériques.)

La structure tant que en langage algorithmique :

```
tant que (<expression logique>) faire  
    <bloc d'instructions>  
ftant
```

- * 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 ftant.
- * Le <bloc d'instructions> est exécuté zéro ou plusieurs fois.

La structure while en C :

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

- * 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 1 :

```
/* Afficher les nombres de 0 à 9 */  
int I = 0;  
while (I<10)  
{  
    printf("%i \n", I);  
    I++;  
}
```

Exemple 2 :

```
int I;  
/* Afficher les nombres de 0 à 9 */
```

```

I = 0;
while (I<10)
    printf("%i \n", I++);
/* Afficher les nombres de 1 à 10 */
I = 0;
while (I<10)
    printf("%i \n", ++I);

```

Exemple 3 :

```

/* Afficher les nombres de 10 à 1 */
int I=10;
while (I)
    printf("%i \n", I--);

```

Remarque :

Parfois nous voulons seulement attendre un certain événement, sans avoir besoin d'un traitement de données. Dans ce cas, la partie <bloc d'instructions> peut être vide (notation : ; ou {}). La ligne suivante ignore tous les espaces entrés au clavier et peut être utilisée avant de lire le premier caractère significatif :

```

while (getch() == ' ')
    ;

```

II) do – while :

La structure **do - while** est semblable à la structure **while**, avec la différence suivante :

- * **while** évalue la condition *avant* d'exécuter le bloc d'instructions,
- * **do - while** évalue la condition *après* avoir exécuté le bloc d'instructions. Ainsi le bloc d'instructions est exécuté au moins une fois.

La structure do - while en C :

```

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 1 :

```

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

```

Exemple 2 :

```

int n, div;
printf("Entrez le nombre à diviser : ");
scanf("%i", &n);
do
{

```

```

        printf("Entrez le diviseur ( 0 ) : ");
        scanf("%i", &div);
    }
    while (!div);
    printf("%i / %i = %f\n", n, div, (float)n/div);

```

! **do - while** est comparable à la structure répéter du langage algorithmique (**repeat until** en Pascal) *si la condition finale est inversée logiquement.*

Exemple 3 :

Le programme de calcul de la racine carrée :

```

programme RACINE_CARREE
    réel N
    répéter
        écrire "Entrer un nombre (>=0) : "
        lire N
        jusqu'à (N >= 0)
        écrire "La racine carrée de ",N ,"vaut ", N
fprogramme (* fin RACINE_CARRE *)

```

se traduit en C par :

```

#include <stdio.h>
#include <math.h>
main()
{
    float N;
    do
    {
        printf("Entrer un nombre (>= 0) : ");
        scanf("%f", &N)
    }
    while (N < 0);
    printf("La racine carrée de %.2f est %.2f\n", N, sqrt(N));
    return (0);
}

```

III) for :

La structure **for** en Pascal et la structure pour en langage algorithmique sont utilisées pour faciliter la programmation de boucles de comptage. La structure **for** en C est plus générale et beaucoup plus puissante.

La structure for en C :

```

for ( <expr1> ; <expr2> ; <expr3> )
    <bloc d'instructions>

```

est équivalent à :

```

<expr1>;
while ( <expr2> )
{
    <bloc d'instructions>
    <expr3>;
}

```

<expr1> est évaluée une fois avant le passage de la boucle. Elle est utilisée pour initialiser les données de la boucle.

<expr2> est évaluée avant chaque passage de la boucle. Elle est utilisée pour décider si la boucle est répétée ou non.

<expr3> est évaluée à la fin de chaque passage de la boucle. Elle est utilisée pour réinitialiser les données de la boucle.

Le plus souvent, **for** est utilisé comme boucle de comptage :

```
for ( <init.> ; <cond. répétition> ; <compteur> )
    <bloc d'instructions>
```

Exemple :

```
int I;
for (I=0 ; I<=20 ; I++)
    printf("Le carré de %d est %d \n", I, I*I);
```

En pratique, les parties <expr1> et <expr2> contiennent souvent plusieurs initialisations ou réinitialisations, *séparées par des virgules*.

Exemple :

```
int n, tot;
for (tot=0, n=1 ; n<101 ; n++)
    tot+=n;
printf("La somme des nombres de 1 à 100 est %d\n", tot);
```

Exemple :

Cet exemple nous présente différentes variations pour réaliser le même traitement et nous montre la puissance de la structure **for**. Les expressions suivantes lisent un caractère au clavier et affichent son code numérique en notation binaire :

```
/* a */
/* notation utilisant la structure while */
int C, I;
C=getchar();
I=128;
while (I>=1)
{
    printf("%i ", C/I);
    C%=I;
    I/=2;
}

/* b */
/* notation utilisant for - très lisible - */
/* préférée par les débutants en C */
int C, I;
C=getchar();
for (I=128 ; I>=1 ; I/=2)
{
    printf("%i ", C/I);
    C%=I;
}

/* c */
/* notation utilisant for - plus compacte - */
/* préférée par les experts en C */
int C, I;
C=getchar();
```

```

for (I=128 ; I>=1 ; C%=I, I/=2)
    printf("%i ", C/I);

/* d */
/* notation utilisant for - à déconseiller - */
/* surcharge et mauvais emploi de la structure */
int C, I;
for(C=getchar(),I=128; I>=1 ;printf("%i ",C/I),C%=i,i/=2);

```

IV) Choix de la structure répétitive :

Dans ce chapitre, nous avons vu trois façons différentes de programmer des boucles (**while**, **do - while**, **for**). Utilisez la structure qui reflète le mieux l'idée du programme que vous voulez réaliser, en respectant toutefois les directives suivantes :

- * Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez **while** ou **for**.
- * Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez **do - while**.
- * Si le nombre d'exécutions du bloc d'instructions dépend d'une ou de plusieurs variables qui sont modifiées à la fin de chaque répétition, alors utilisez **for**.
- * Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie (p.ex aussi longtemps qu'il y a des données dans le fichier d'entrée), alors utilisez **while**.

Le choix entre **for** et **while** n'est souvent qu'une question de préférence ou d'habitudes :

- * **for** nous permet de réunir avantageusement les instructions qui influencent le nombre de répétitions au début de la structure.
- * **while** a l'avantage de correspondre plus exactement aux structures d'autres langages de programmation (**while**, **tant que**).
- * **for** a le désavantage de favoriser la programmation de structures surchargées et par la suite illisibles.
- * **while** a le désavantage de mener parfois à de longues structures, dans lesquelles il faut chercher pour trouver les instructions qui influencent la condition de répétition.

V) Exercices d'application :

Exercice 6.1 :

Ecrivez un programme qui lit N nombres entiers au clavier et qui affiche leur somme, leur produit et leur moyenne. Choisissez un type approprié pour les valeurs à afficher. Le nombre N est à entrer au clavier. Résolvez ce problème,

- en utilisant **while**,
 - en utilisant **do - while**,
 - en utilisant **for**.
 - Laquelle des trois variantes est la plus naturelle pour ce problème?
-

Exercice 6.2 :

Complétez la 'meilleure' des trois versions de l'exercice 6.1 :
Répétez l'introduction du nombre N jusqu'à ce que N ait une valeur entre 1 et 15.
Quelle structure répétitive utilisez-vous? Pourquoi?

Exercice 6.3 :

Calculez par des soustractions successives le quotient entier et le reste de la division entière de deux entiers entrés au clavier.

Exercice 6.4 :

Calculez la factorielle $N! = 123...(N-1)N$ d'un entier naturel N en respectant que $0!=1$.
a) Utilisez **while**,
b) Utilisez **for**.

Exercice 6.5 :

Calculez par multiplications successives X^N de deux entiers naturels X et N entrés au clavier.

Exercice 6.6 :

Calculez la somme des N premiers termes de la série harmonique :
 $1 + 1/2 + 1/3 + \dots + 1/N$

Exercice 6.7 :

Calculez la somme, le produit et la moyenne d'une suite de chiffres non nuls entrés au clavier, sachant que la suite est terminée par zéro. Retenez seulement les chiffres (0, 1 ... 9) lors de l'entrée des données et effectuez un signal sonore si les données sortent de ce domaine.

Exercice 6.8 :

Calculez le nombre lu à rebours d'un nombre positif entré au clavier en supposant que le fichier d'entrée standard contient une suite de chiffres non nuls, terminée par zéro (Contrôlez s'il s'agit vraiment de chiffres). **Exemple :** Entrée : 1 2 3 4 0 Affichage : 4321

Exercice 6.9 :

Calculez le nombre lu à rebours d'un nombre positif entré au clavier en supposant que le fichier d'entrée standard contient le nombre à inverser. **Exemple** : Entrée : 1234 Affichage : 4321

Exercice 6.10 :

Calculez pour une valeur X donnée du type **float** la valeur numérique d'un polynôme de degré n :

$$P(X) = A_n X^n + A_{n-1} X^{n-1} + \dots + A_1 X + A_0$$

Les valeurs de n, des coefficients A_n, \dots, A_0 et de X seront entrées au clavier.

Utilisez le *schéma de Horner* qui évite les opérations d'exponentiation lors du calcul :

$$\begin{array}{c} \underbrace{A_n} \\ \underbrace{* X + A_{n-1}} \\ \underbrace{* X + A_{n-2}} \\ \dots \\ \underbrace{* X + A_0} \end{array}$$

Exercice 6.11 :

Calculez le P.G.C.D. de deux entiers naturels entrés au clavier en utilisant l'algorithme d'EUCLIDE.

Exercice 6.12 :

Calculez le N-ième terme U_N de la suite de FIBONACCI qui est donnée par la relation de récurrence :

$$U_1=1 \quad U_2=1 \quad U_N=U_{N-1} + U_{N-2} \text{ (pour } N>2)$$

Déterminez le rang N et la valeur U_N du terme maximal que l'on peut calculer si on utilise pour U_N :

- le type **int**
 - le type **long**
 - le type **double**
 - le type **long double**
-

Exercice 6.13 :

- a) Calculez la racine carrée X d'un nombre réel positif A par approximations successives en utilisant la relation de récurrence suivante :

$$X_{J+1} = (X_J + A/X_J) / 2 \quad X_1 = A$$

La précision du calcul J est à entrer par l'utilisateur.

- b) Assurez-vous lors de l'introduction des données que la valeur pour A est un réel positif et que J est un entier naturel positif, plus petit que 50.
- c) Affichez lors du calcul toutes les approximations calculées :

La 1ère approximation de la racine carrée de ... est ...
La 2e approximation de la racine carrée de ... est ...
La 3e approximation de la racine carrée de ... est ...
...

Exercice 6.14 :

Affichez un triangle isocèle formé d'étoiles de N lignes (N est fourni au clavier) :
Nombre de lignes : 8

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

Exercice 6.15 :

Affiche la table des produits pour N variant de 1 à 10 :

X*Y	I	0	1	2	3	4	5	6	7	8	9	10
0	I	0	0	0	0	0	0	0	0	0	0	0
1	I	0	1	2	3	4	5	6	7	8	9	10
2	I	0	2	4	6	8	10	12	14	16	18	20
3	I	0	3	6	9	12	15	18	21	24	27	30
4	I	0	4	8	12	16	20	24	28	32	36	40
5	I	0	5	10	15	20	25	30	35	40	45	50
6	I	0	6	12	18	24	30	36	42	48	54	60
7	I	0	7	14	21	28	35	42	49	56	63	70
8	I	0	8	16	24	32	40	48	56	64	72	80
9	I	0	9	18	27	36	45	54	63	72	81	90
10	I	0	10	20	30	40	50	60	70	80	90	100

VI) Solutions des exercices du Chapitre 6 : LA STRUCTURE REPETITIVE :

Exercice 6.1 :

a) en utilisant **while**,

```
#include <stdio.h>
main()
{
    int N;          /* nombre de données */
    int NOMB;       /* nombre courant */
    int I;          /* compteur */
    long SOM;       /* la somme des nombres entrés */
    double PROD;   /* le produit des nombres entrés */

    printf("Nombre de données : ");
    scanf("%d", &N);

    SOM=0;
    PROD=1;
    I=1;
    while (I<=N)
    {
        printf("%d. nombre : ", I);
        scanf("%d", &NOMB);
        SOM += NOMB;
        PROD *= NOMB;
        I++;
    }

    printf("La somme des %d nombres est %ld \n", N, SOM);
    printf("Le produit des %d nombres est %.0f\n", N, PROD);
    printf("La moyenne des %d nombres est %.4f\n", N,
(float)SOM/N);
    return (0);
}
```

b) en utilisant **do - while**,

Remplacez le bloc de traitement (en gras) de (a) par :

```
SOM=0;
PROD=1;
I=1;
do
{
    printf("%d. nombre : ", I);
    scanf("%d", &NOMB);
    SOM += NOMB;
    PROD *= NOMB;
    I++;
}
while (I<=N);
```

c) en utilisant **for**.

Remplacez le bloc de traitement (en gras) de (a) par :

```
for (SOM=0, PROD=1, I=1 ; I<=N ; I++)
{
```

```

printf("%d. nombre : ", I);
scanf("%d", &NOMB);
SOM += NOMB;
PROD *= NOMB;
}

```

d) Laquelle des trois variantes est la plus naturelle pour ce problème ?

La structure **for** est la plus compacte et celle qui exprime le mieux l'idée de l'algorithme. D'autre part, elle permet d'intégrer très confortablement l'initialisation et l'incrémentation des variables dans la structure.

Exercice 6.2 :

Remplacer les lignes

```

printf("Nombre de données : ");
scanf("%d", &N);

```

par

```

do
{
printf("Nombre de données : ");
scanf("%d", &N);
}
while(N<1 || N>15);

```

Quelle structure répétitive utilisez-vous ? Pourquoi ?

Comme il n'y pas d'initialisation ni de réinitialisation de variables, nous avons le choix entre **while** et **do - while**. Comme l'introduction du nombre de données doit toujours être exécuté (au moins une fois), le plus naturel sera l'utilisation de la structure **do - while**.

Exercice 6.3 :

```

#include <stdio.h>
main()
{
int NUM; /* numérateur de la division entière */
int DEN; /* dénominateur de la division entière */
int DIV; /* résultat de la division entière */
int RES; /* reste de la division entière */

printf("Introduisez le numérateur : ");
scanf("%d", &NUM);
printf("Introduisez le dénominateur : ");
scanf("%d", &DEN);

RES=NUM;
DIV=0;
while (RES>=DEN)
{
RES-=DEN;
DIV++;
}

/* ou mieux encore : */
/*
for (RES=NUM, DIV=0 ; RES>=DEN ; DIV++)
RES-=DEN;
*/

```

```

*/

printf(" %d divisé par %d est %d reste %d\n", NUM, DEN, DIV,
RES);
return (0);
}

```

Exercice 6.4 :

Solution combinée :

(Essayez l'une ou l'autre des solutions en déplaçant les marques des commentaires !)

```

#include <stdio.h>
main()
{
  int N;      /* La donnée */
  int I;      /* Le compteur */
  double FACT; /* La factorielle N! - Type double à */
              /* cause de la grandeur du résultat. */

  do
  {
    printf("Entrez un entier naturel : ");
    scanf("%d", &N);
  }
  while (N<0);

  /* a */
  /* Pour N=0, le résultat sera automatiquement 0!=1 */
  I=1;
  FACT=1;
  while (I<=N)
  {
    FACT*=I;
    I++;
  }

  /* b */
  /* Pour N=0, le résultat sera automatiquement 0!=1 */
  /*
  for (FACT=1.0, I=1 ; I<=N ; I++)
    FACT*=I;
  */

  printf ("%d! = %.0f\n", N, FACT);
  return (0);
}

```

Exercice 6.5 :

```

#include <stdio.h>
main()
{
  int X, N; /* Les données */

```

```

int I;          /* Le compteur */
double RESU;   /* Type double à cause de la */
               /* grandeur du résultat.    */

do
{
    printf("Entrez l'entier naturel X : ");
    scanf("%d", &X);
}
while (X<0);
do
{
    printf("Entrez l'exposant          N : ");
    scanf("%d", &N);
}
while (N<0);

/* Pour N=0, le résultat sera automatiquement X^0=1 */
for (RESU=1.0, I=1 ; I<=N ; I++)
    RESU*=X;

/* Attention : Pour X=0 et N=0 , 0^0 n'est pas défini */
if (N==0 && X==0)
    printf("zéro exposant zéro n'est pas défini !\n");
else
    printf("Résultat : %d ^ %d = %.0f\n", X, N, RESU);
return (0);
}

```

Exercice 6.6 :

```

#include <stdio.h>
main()
{
    int N;          /* nombre de termes à calculer */
    int I;          /* compteur pour la boucle */
    float SOM;     /* Type float à cause de la précision du
résultat.    */
    do
    {
        printf ("Nombre de termes : ");
        scanf ("%d", &N);
    }
    while (N<1);
    for (SOM=0.0, I=1 ; I<=N ; I++)
        SOM += (float)1/I;
    printf("La somme des %d premiers termes est %f \n", N,
SOM);
    return (0);
}

```

Exercice 6.7 :

Solution (une de plusieurs solutions possibles) :

```

#include <stdio.h>
main()
{
    int X;          /* Le chiffre courant      */
    int N=0;        /* Le compteur des données */
    int SOM=0;      /* La somme actuelle       */
    long PROD=1;   /* Le produit actuel - Type long à */
                  /* cause de la grandeur du résultat. */
                  /*
do
    {
        /* Saisie des données (pour perfectionnistes) */
        printf("Entrez le %d%s chiffre : ", (N+1), (N)?"e"
:"er");
        scanf("%d", &X);

        if (X<0||X>9)
            printf("\a");
        else if (X)
            {
                N++;
                SOM+=X;
                PROD*=X;
            }
        else if (!X && N>0)
            /* Seulement si au moins un chiffre a été accepté
*/
            printf("La somme des chiffres est %d \n", SOM);
            printf("Le produit des chiffres est %ld\n", PROD);
            printf("La moyenne des chiffres est %f \n",
(float)SOM/N);
        }
    }
    while (X);
    return (0);
}

```

Exercice 6.8 :

```

#include <stdio.h>
main()
{
    int X;          /* Le chiffre courant      */
    int N=0;        /* Le compteur des décimales */
    long VALD=1;    /* Valeur de la position décimale courante */
    long NOMB=0;    /* Le nombre résultat      */
do
    {
        printf("Entrez le %d%s chiffre : ", (N+1),
(N)?"e":"er");
        scanf("%d", &X);

        if (X<0||X>9)

```

```

        printf("\a");
    else if (X)
        {
            NOMB += VALD*X;
            N++;
            VALD *= 10;
        }
    else
        printf("La valeur du nombre renversé est %ld\n",
NOMB);
    }
    while (X);
    return (0);
}

```

Remarque :

En remplaçant la ligne

NOMB += VALD*X;

par

NOMB += pow(10, N)*X;

on n'a plus besoin de la variable VALD. Il faut cependant inclure les fonctions de la bibliothèque *<math>*. D'autre part, le calcul de 10^N serait alors répété à chaque exécution de la boucle.

Finalement, cette variante est plus lente et plus volumineuse que la première.

Exercice 6.10 :

```

#include <stdio.h>
main()
{
    int N;      /* degré du polynôme */
    float X;   /* argument          */
    float A;   /* coefficients successifs du polynôme */
    float P;   /* coefficient courant du terme Horner */

    printf("Entrer le degré N du polynôme : ");
    scanf("%d", &N);
    printf("Entrer la valeur X de l'argument : ");
    scanf("%f", &X);

    for(P=0.0 ; N>=0 ; N--)
        {
            printf("Entrer le coefficient A%d : ", N);
            scanf("%f", &A);
            P = P*X + A;
        }

    printf("Valeur du polynôme pour X = %.2f : %.2f\n", X, P);
    return (0);
}

```

Exercice 6.11 :

```

#include <stdio.h>

```

```

main()
{
    int A, B;          /* données */
    int X, Y, RESTE; /* var. d'aide pour l'algorithme d'Euclide
*/

    do
    {
        printf("Entrer l'entier A (non nul) : ");
        scanf("%d", &A);
    }
    while(!A);
    do
    {
        printf("Entrer l'entier B (non nul) : ");
        scanf("%d", &B);
    }
    while(!B);

    for (RESTE=A, X=A, Y=B ; RESTE ; X=Y, Y=RESTE)
        RESTE = X%Y;

    printf("Le PGCD de %d et de %d est %d\n", A, B, X);
    return (0);
}

```

Exercice 6.12 :

```

#include <stdio.h>
main()
{
    int U1, U2, UN; /* pour parcourir la suite */
    int N;          /* rang du terme demandé */
    int I;          /* compteur pour la boucle */
    do
    {
        printf("Rang du terme demandé : ");
        scanf("%d", &N);
    }
    while (N<1);

    U1=U2=1; /* Initialisation des deux premiers termes */
    if (N==1)
        UN=U1;
    else if (N==2)
        UN=U2;
    else
    {
        for (I=3 ; I<=N ; I++)
        {
            UN = U1+U2;
            U1 = U2;
            U2 = UN;
        }
    }
}

```

```

    }
    printf("Valeur du terme de rang %d : %d\n", N, UN);
    return (0);
}

```

Rang et terme maximal calculable en utilisant les déclarations :

```

int U1, U2, UN;          (spéc. de format : %d)   U23 = 28657
long U1, U2, UN;       (spéc. de format : %ld)  U46 = 1836311903
double U1, U2, UN;    (spéc. de format : %e)   U1476 = 1.306989e308
long double U1, U2, UN; (spéc. de format : %Le) U23601 = 9.285655e4931

```

Exercice 6.13 :

```

#include <stdio.h>
main()
{
    double A;    /* donnée */
    double X;    /* approximation de la racine carrée de A */
    int N;       /* degré/précision de l'approximation */
    int J;       /* degré de l'approximation courante */

    do
    {
        printf("Entrer le réel positif A : ");
        scanf("%lf", &A);
    }
    while(A<0);
    do
    {
        printf("Entrer le degré de l'approximation : ");
        scanf("%d", &N);
    }
    while(N<=0 || N>=50);

    for(X=A, J=1 ; J<=N ; J++)
    {
        X = (X + A/X) / 2;
        printf("La %2d%s approximation de la racine carrée"
               " de %.2f est %.2f\n", J, (J==1)?"ère":"e", A,
X);
    }
    return (0);
}

```

Exercice 6.14 :

```

#include <stdio.h>
main()
{
    int LIG;    /* nombre de lignes */
    int L;      /* compteur des lignes */
    int ESP;    /* nombre d'espaces */
    int I;      /* compteur des caractères */

    do

```

```

    {
        printf("Nombres de lignes : ");
        scanf("%d", &LIG);
    }
while (LIG<1 || LIG>20);

for (L=0 ; L<LIG ; L++)
    {
        ESP = LIG-L-1;
        for (I=0 ; I<ESP ; I++)
            putchar(' ');
        for (I=0 ; I<2*L+1 ; I++)
            putchar('*');
        putchar('\n');
    }
return (0);
}

```

Exercice 6.15 :

```

#include <stdio.h>
main()
{
    const int MAX = 10; /* nombre de lignes et de colonnes */
    int I;               /* compteur des lignes */
    int J;               /* compteur des colonnes */

    /* Affichage de l'en-tête */
    printf(" X*Y I");
    for (J=0 ; J<=MAX ; J++)
        printf("%4d", J);
    printf("\n");
    printf("-----");
    for (J=0 ; J<=MAX ; J++)
        printf("----");
    printf("\n");

    /* Affichage du tableau */
    for (I=0 ; I<=MAX ; I++)
        {
            printf("%3d I", I);
            for (J=0 ; J<=MAX ; J++)
                printf("%4d", I*J);
            printf("\n");
        }
    return (0);
}

```

Chapitre 7 :LES TABLEAUX

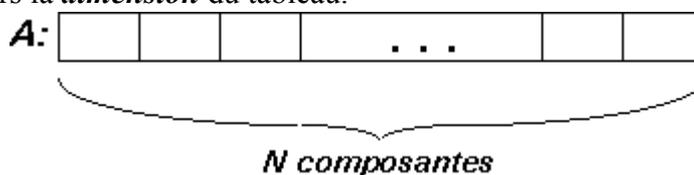
Les tableaux sont certainement les variables structurées les plus populaires. Ils sont disponibles dans tous les langages de programmation et servent à résoudre une multitude de problèmes. Dans une première approche, le traitement des tableaux en C ne diffère pas de celui des autres langages de programmation. Nous allons cependant voir plus loin (Chapitre 9. Les Pointeurs), que le langage C permet un accès encore plus direct et plus rapide aux données d'un tableau.

Les chaînes de caractères sont déclarées en C comme tableaux de caractères et permettent l'utilisation d'un certain nombre de notations et de fonctions spéciales. Les particularités des tableaux de caractères seront traitées séparément au chapitre 8.

I) Les tableaux à une dimension :

Définitions :

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 "

Exemple :

La déclaration

```
int JOURS [12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

définit un tableau du type `int` de dimension 12. Les 12 composantes sont initialisées par les valeurs respectives 31, 28, 31, ..., 31.

On peut accéder à la première composante du tableau par `JOURS[0]`, à la deuxième composante par `JOURS[1]`, ..., à la dernière composante par `JOURS[11]`.

1) Déclaration et mémorisation :

a) Déclaration

Déclaration de tableaux en langage algorithmique :

```
<TypeSimple> tableau <NomTableau> [<Dimension>]
```

Déclaration de tableaux en C :

```
<TypeSimple> <NomTableau> [<Dimension>];
```

Les noms des tableaux sont des *identificateurs* qui doivent correspondre aux restrictions définies au chapitre 2.2.4.

Exemples :

Les déclarations suivantes en langage algorithmique,

```
entier tableau A[25]
réel tableau B[100]
booléen tableau C[10]
caractère tableau D[30]
```

se laissent traduire en C par :

```
int A[25];    ou bien long A[25];    ou bien ...
```

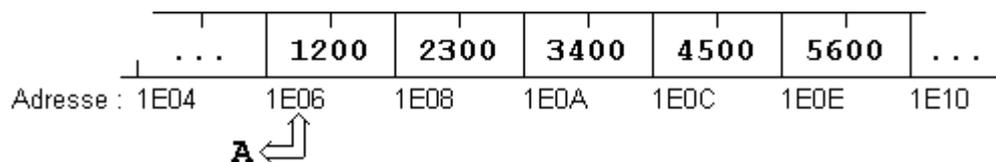
```
float B[100]; ou bien double B[100]; ou bien ...
int C[10];
char D[30];
```

b) Mémorisation :

En C, le nom d'un tableau est le représentant de *l'adresse du premier élément* du tableau. Les adresses des autres composantes sont calculées (automatiquement) relativement à cette adresse.

Exemple :

```
short A[5] = {1200, 2300, 3400, 4500, 5600};
```



Si un tableau est formé de N composantes et si une composante a besoin de M octets en mémoire, alors le tableau occupera de N*M octets.

Exemple :

En supposant qu'une variable du type **long** occupe 4 octets (c.-à-d : **sizeof(long)=4**), pour le tableau T déclaré par : **long T[15];**

C réservera $N*M = 15*4 = 60$ octets en mémoire.

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.

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};
```

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 !

3) Accès aux composantes :

En déclarant un tableau par :

```
int A[5];
```

nous avons défini un tableau A avec cinq composantes, auxquelles on peut accéder par :

```
A[0], A[1], ... , A[4]
```

Exemple :

```
short A[5] = {1200, 2300, 3400, 4500, 5600};
```

A:	1200	2300	3400	4500	5600
	A[0]	A[1]	A[2]	A[3]	A[4]

Exemples :

```
MAX = (A[0] > A[1]) ? A[0] : A[1];
```

```
A[4] *= 2;
```

Attention ! :

Considérons un tableau T de dimension N :

En C,

- l'accès au premier élément du tableau se fait par T[0]
- l'accès au dernier élément du tableau se fait par T[N-1]

En langage algorithmique,

- l'accès au premier élément du tableau se fait par T[1]
- l'accès au dernier élément du tableau se fait par T[N]



4) Affichage et affectation :

La structure **for** se prête particulièrement bien au travail avec les tableaux. La plupart des applications se laissent implémenter par simple modification des exemples types de l'affichage et de l'affectation.

a) - Affichage du contenu d'un tableau :

Traduisons le programme AFFICHER du langage algorithmique en C :

```
programme AFFICHER
|   entier tableau A[5]
|   entier I  (* Compteur *)
```

```

|   pour I variant de 1 à 5 faire
|       écrire A[I]
|   fpour
fprogramme

```

```

main()
{
    int A[5];
    int I; /* Compteur */
    for (I=0; I<5; I++)
        printf("%d ", A[I]);
    return (0);
    printf("\n");
}

```

Remarques :

- * Avant de pouvoir afficher les composantes d'un tableau, il faut évidemment leur affecter des valeurs.
- * Rappelez-vous que la deuxième condition dans la structure **for** n'est pas une condition d'arrêt, mais une condition de répétition! Ainsi la commande d'affichage sera répétée *aussi longtemps* que **I** est inférieur à 5. La boucle sera donc bien exécutée pour les indices 0,1,2,3 et 4 !
- * Par opposition à la commande simplifiée écrire A[I] du langage algorithmique, la commande **printf** doit être informée du type exact des données à afficher. (Ici : **%d** ou **%i** pour des valeurs du type **int**)
- * Pour être sûr que les valeurs sont bien séparées lors de l'affichage, il faut inclure au moins un espace dans la chaîne de format. Autres possibilités :

```

    printf("%d\t", A[I]); /* tabulateur */
    printf("%7d", A[I]); /* format d'affichage */

```

b) Affectation :

- Affectation avec des valeurs provenant de l'extérieur :

Traduisons le programme REMPLIR du langage algorithmique en C :

```

programme REMPLIR
|   entier tableau A[5]
|   entier I (* Compteur *)
|   pour I variant de 1 à 5 faire
|       lire A[I]
|   fpour
fprogramme

```

```

main()
{
    int A[5];
    int I; /* Compteur */
    for (I=0; I<5; I++)
        scanf("%d", &A[I]);
    return (0);
}

```

Remarques :

- * Comme **scanf** a besoin des adresses des différentes composantes du tableau, il faut faire précéder le terme A[I] par l'opérateur adresse **'&'**.

- La commande de lecture **scanf** doit être informée du type exact des données à lire. (Ici : **%d** ou **%i** pour lire des valeurs du type **int**).

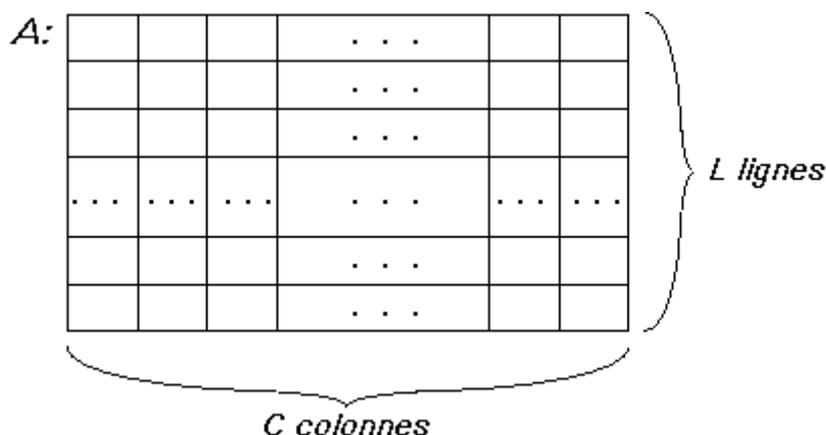
- [Exercice 7.1](#)
- [Exercice 7.2](#)
- [Exercice 7.3](#)
- [Exercice 7.4](#)

II) Les tableaux à deux dimensions :

Définitions :

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

On appelle L le **nombre de lignes** du tableau et C le **nombre de colonnes** du tableau. L et C sont alors les deux **dimensions** du tableau. Un tableau à deux dimensions contient donc **L*C composantes**.



On dit qu'un tableau à deux dimensions est **carré**, si L est égal à C.

En faisant le rapprochement avec les mathématiques, on peut dire que "A est un vecteur de L vecteurs de dimension C", ou mieux :

"A est une **matrice** de dimensions L et C".

Exemple

Considérons un tableau NOTES à une dimension pour mémoriser les notes de 20 élèves d'une classe dans un devoir :

```
int NOTE[20] = {45, 34, ... , 50, 48};
```

NOTE:	45	34	...	50	48
	A[0]	A[1]		A[18]	A[19]

Pour mémoriser les notes des élèves dans les 10 devoirs d'un trimestre, nous pouvons rassembler plusieurs de ces tableaux unidimensionnels dans un tableau NOTES à deux dimensions :

```
int NOTE[10][20] = {{45, 34, ... , 50, 48},
                    {39, 24, ... , 49, 45},
                    ...
                    {40, 40, ... , 54, 44}};
```

NOTE :

45	34	...	50	48
39	24	...	49	45
...
40	40	...	54	44

} 10 lignes

20 colonnes

Dans une ligne nous retrouvons les notes de tous les élèves dans un devoir. Dans une colonne, nous retrouvons toutes les notes d'un élève.

1) Déclaration et mémorisation :

a) Déclarations :

Déclaration de tableaux à deux dimensions en lang. algorithmique

<TypeSimple> tableau <NomTabl> [<DimLigne>, <DimCol>]

Déclaration de tableaux à deux dimensions en C

<TypeSimple> <NomTabl> [<DimLigne>] [<DimCol>];

Exemples

Les déclarations suivantes en langage algorithmique,

entier tableau A[10,10]

réel tableau B[2,20]

booléen tableau C[3,3]

caractère tableau D[15,40]

se laissent traduire en C par :

int A[10][10]; ou bien long A[10][10]; ou bien ...

float B[2][20]; ou bien double B[2][20]; ou bien ...

int C[3][3];

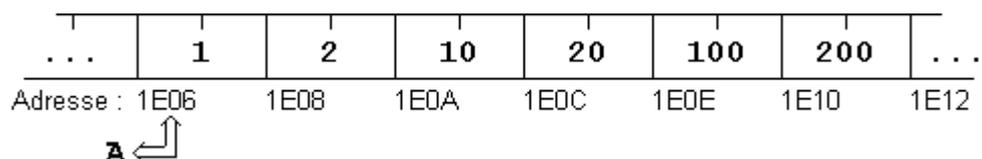
char D[15][40];

b) Mémorisation :

Comme pour les tableaux à une dimension, le nom d'un tableau est le représentant de *l'adresse du premier élément* du tableau (c'est-à-dire l'adresse de la première *ligne* du tableau). Les composantes d'un tableau à deux dimensions sont stockées ligne par ligne dans la mémoire.

Exemple : Mémorisation d'un tableau à deux dimensions

```
short A[3][2] = { {1, 2 },
                  {10, 20 },
                  {100, 200} };
```



Un tableau de dimensions L et C, formé de composantes dont chacune a besoin de M octets, occupera L*C*M octets en mémoire.

Exemple

En supposant qu'une variable du type **double** occupe 8 octets (c.-à-d : **sizeof(double)=8**), pour le tableau T déclaré par : **double T[10][15];**

C réservera $L * C * M = 10 * 15 * 8 = 1200$ octets en mémoire.

2) Initialisation et réservation automatique :

a) Initialisation

Lors de la déclaration d'un tableau, on peut initialiser les composantes du tableau, en indiquant la liste des valeurs respectives entre accolades. A l'intérieur de la liste, les composantes de chaque ligne du tableau sont encore une fois comprises entre accolades. Pour améliorer la lisibilité des programmes, on peut indiquer les composantes dans plusieurs lignes.

Exemples :

```
int A[3][10] = {{ 0,10,20,30,40,50,60,70,80,90},
                {10,11,12,13,14,15,16,17,18,19},
                { 1,12,23,34,45,56,67,78,89,90}};
```

```
float B[3][2] = {{-1.05, -1.10 },
                 {86e-5, 87e-5 },
                 {-12.5E4, -12.3E4}};
```

Lors de l'initialisation, les valeurs sont affectées ligne par ligne en passant de gauche à droite. Nous ne devons pas nécessairement indiquer toutes les valeurs : Les valeurs manquantes seront initialisées par zéro. Il est cependant défendu d'indiquer trop de valeurs pour un tableau.

Exemples :

```
int C[4][4] = {{1, 0, 0, 0}
               {1, 1, 0, 0}
               {1, 1, 1, 0}
               {1, 1, 1, 1}};
```

C:

1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1

```
int C[4][4] = {{1, 1, 1, 1}};
```

C:

1	1	1	1
0	0	0	0
0	0	0	0
0	0	0	0

```
int C[4][4] = {{1}, {1}, {1}, {1}};
```

C:

1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0

```
int C[4][4] = {{1, 1, 1, 1, 1}};
```

↘ ERREUR !

Réservation automatique

Si le nombre de **lignes L** n'est pas indiqué explicitement lors de l'initialisation, l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

```
int A[][10] = { { 0,10,20,30,40,50,60,70,80,90 },
               { 10,11,12,13,14,15,16,17,18,19 },
               { 1,12,23,34,45,56,67,78,89,90 } };
```

réserve de $3 \times 10 \times 2 = 60$ octets

```
float B[][2] = { { -1.05, -1.10 },
                 { 86e-5, 87e-5 },
                 { -12.5E4, -12.3E4 } };
```

réserve de $3 \times 2 \times 4 = 24$ octets

Exemple :

```
int C[][4] = { { 1, 0, 0, 0 }
               { 1, 1, 0, 0 }
               { 1, 1, 1, 0 }
               { 1, 1, 1, 1 } };
```



C:

1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1

⇒ réserve de $4 \times 4 \times 2 = 32$ octets

3) Accès aux composantes :

Accès à un tableau à deux dimensions en lang. algorithmique

<NomTableau> [<Ligne>, <Colonne>]

Accès à un tableau à deux dimensions en C

<NomTableau> [<Ligne>] [<Colonne>]

Les éléments d'un tableau de dimensions L et C se présentent de la façon suivante :

/	A[0][0]	A[0][1]	A[0][2]	. . .	A[0][C-1]	\
	A[1][0]	A[1][1]	A[1][2]	. . .	A[1][C-1]	
	A[2][0]	A[2][1]	A[2][2]	. . .	A[2][C-1]	
	
	A[L-1][0]	A[L-1][1]	A[L-1][2]	. . .	A[L-1][C-1]	
\						/

Attention !

Considérons un tableau A de dimensions L et C.

En C,

- les indices du tableau varient de **0** à **L-1**, respectivement de **0** à **C-1**.
- la composante de la N^{ième} ligne et M^{ième} colonne est notée :

A[N-1][M-1]

En langage algorithmique,

- les indices du tableau varient de **1** à **L**, respectivement de **1** à **C**.
- la composante de la N^{ième} ligne et M^{ième} colonne est notée :

A[N,M]



4) Affichage et affectation :

Lors du travail avec les tableaux à deux dimensions, nous utiliserons deux indices (p.ex : I et J), et la structure **for**, souvent imbriquée, pour parcourir les lignes et les colonnes des tableaux.

a) - Affichage du contenu d'un tableau à deux dimensions :

Traduisons le programme AFFICHER du langage algorithmique en C :

```
programme AFFICHER
|   entier tableau A[5,10]
|   entier I,J
|   (* Pour chaque ligne ... *)
|   pour I variant de 1 à 5 faire
|       (* ... considérer chaque composante *)
|       pour J variant de 1 à 10 faire
|           écrire A[I,J]
|       fpour
|       (* Retour à la ligne *)
|       écrire
|   fpour
fprogramme

main()
{
  int A[5][10];
  int I,J;
  /* Pour chaque ligne ... */
  for (I=0; I<5; I++)
  {
    /* ... considérer chaque composante */
    for (J=0; J<10; J++)
      printf("%7d", A[I][J]);
    /* Retour à la ligne */
    printf("\n");
  }
  return (0);
}
```

Remarques

- * Avant de pouvoir afficher les composantes d'un tableau, il faut leur affecter des valeurs.
- * Pour obtenir des colonnes bien alignées lors de l'affichage, il est pratique d'indiquer la largeur minimale de l'affichage dans la chaîne de format. Pour afficher des matrices du type **int** (valeur la plus 'longue' : -32768), nous pouvons utiliser la chaîne de format "%7d" :
`printf("%7d", A[I][J]);`

b) - Affectation avec des valeurs provenant de l'extérieur

Traduisons le programme REMPLIR du langage algorithmique en C :

```
programme REMPLIR
|   entier tableau A[5,10]
|   entier I,J
|   (* Pour chaque ligne ... *)
|   pour I variant de 1 à 5 faire
|       (* ... considérer chaque composante *)
|       pour J variant de 1 à 10 faire
|           lire A[I,J]
|       fpour
```

```

|   fpour
fprogramme

main()
{
  int A[5][10];
  int I,J;
  /* Pour chaque ligne ... */
  for (I=0; I<5; I++)
    /* ... considérer chaque composante */
    for (J=0; J<10; J++)
      scanf("%d", &A[I][J]);
  return (0);
}

```

-
- [Exercice 7.5](#)
 - [Exercice 7.6](#)
 - [Exercice 7.7](#)
-

III) Exercices d'application :

Exercice 7.1 :

Ecrire un programme qui lit la dimension N d'un tableau T du type **int** (dimension maximale : 50 composantes), remplit le tableau par des valeurs entrées au clavier et affiche le tableau.

Calculer et afficher ensuite la somme des éléments du tableau.

Exercice 7.2 :

Ecrire un programme qui lit la dimension N d'un tableau T du type **int** (dimension maximale : 50 composantes), remplit le tableau par des valeurs entrées au clavier et affiche le tableau.

Effacer ensuite toutes les occurrences de la valeur 0 dans le tableau T et tasser les éléments restants. Afficher le tableau résultant.

Exercice 7.3 :

Ecrire un programme qui lit la dimension N d'un tableau T du type **int** (dimension maximale : 50 composantes), remplit le tableau par des valeurs entrées au clavier et affiche le tableau.

Ranger ensuite les éléments du tableau T dans l'ordre inverse sans utiliser de tableau d'aide. Afficher le tableau résultant.

Idée : Echanger les éléments du tableau à l'aide de deux indices qui parcourent le tableau en commençant respectivement au début et à la fin du tableau et qui se rencontrent en son milieu.

Exercice 7.4 :

Ecrire un programme qui lit la dimension N d'un tableau T du type **int** (dimension maximale : 50 composantes), remplit le tableau par des valeurs entrées au clavier et affiche le tableau.

Copiez ensuite toutes les composantes strictement positives dans un deuxième tableau TPOS et toutes les valeurs strictement négatives dans un troisième tableau TNEG. Afficher les tableaux TPOS et TNEG.

Exercice 7.5 :

Ecrire un programme qui lit les dimensions L et C d'un tableau T à deux dimensions du type **int** (dimensions maximales : 50 lignes et 50 colonnes). Remplir le tableau par des valeurs entrées au clavier et afficher le tableau ainsi que la somme de tous ses éléments.

Exercice 7.6 :

Ecrire un programme qui lit les dimensions L et C d'un tableau T à deux dimensions du type **int** (dimensions maximales : 50 lignes et 50 colonnes). Remplir le tableau par des valeurs entrées au clavier et afficher le tableau ainsi que la somme de chaque ligne et de chaque colonne en n'utilisant qu'une variable d'aide pour la somme.

Exercice 7.7 :

Ecrire un programme qui transfère un tableau M à deux dimensions L et C (dimensions maximales : 10 lignes et 10 colonnes) dans un tableau V à une dimension L*C.

Exemple :

$$\begin{pmatrix} / & & & \backslash \\ | & a & b & c & d & | \\ | & e & f & g & h & | \\ | & i & j & k & l & | \\ \backslash & & & / & & \end{pmatrix} ==> \begin{pmatrix} / & & & & & & & & & & \backslash \\ | & a & b & c & d & e & f & g & h & i & j & k & l & | \\ \backslash & & & & & & & & & & / & & \end{pmatrix}$$

Remarque :

Les exercices sur les tableaux sont partagés en deux séries :

- [1\) Tableaux à une dimension - Vecteurs](#)
- [2\) Tableaux à deux dimensions - Matrices](#)

Schéma :

Les exercices des deux séries sont à traiter d'après le même schéma :

- Déclaration des variables
- Lecture des dimensions des tableaux
- Lecture des données des tableaux
- Traitements
- Affichage des résultats

Choisissez les dimensions maximales appropriées pour les tableaux.

1) Tableaux à une dimension – Vecteurs :

Exercice 7.8 :

Produit scalaire de deux vecteurs :

Ecrire un programme qui calcule le produit scalaire de deux vecteurs d'entiers U et V (de même dimension).

Exemple :

$$\begin{pmatrix} / & & & \backslash \\ | & 3 & 2 & -4 & | \\ \backslash & & & / & \end{pmatrix} * \begin{pmatrix} / & & & \backslash \\ | & 2 & -3 & 5 & | \\ \backslash & & & / & \end{pmatrix} = 3*2+2*(-3)+(-4)*5 = -20$$

Exercice 7.9 :

Calcul d'un polynôme de degré N :

Calculer pour une valeur X donnée du type **float** la valeur numérique d'un polynôme de degré n :

$$P(X) = A_n X^n + A_{n-1} X^{n-1} + \dots + A_1 X + A_0$$

Les valeurs des coefficients A_n, \dots, A_0 seront entrées au clavier et mémorisées dans un tableau A de type **float** et de dimension n+1.

- Utilisez la fonction **pow()** pour le calcul.
- Utilisez le *schéma de Horner* qui évite les opérations d'exponentiation :

$$\begin{array}{c} \underbrace{A_n} \\ * X + A_{n-1} \\ \underbrace{\quad} \\ * X + A_{n-2} \\ \underbrace{\quad} \\ \dots \\ * X + A_0 \end{array}$$

Exercice 7.10 :

Maximum et minimum des valeurs d'un tableau :

Ecrire un programme qui détermine la plus grande et la plus petite valeur dans un tableau d'entiers A. Afficher ensuite la valeur et la position du maximum et du minimum. Si le tableau contient plusieurs maxima ou minima, le programme retiendra la position du premier maximum ou minimum rencontré.

Exercice 7.11 :

Insérer une valeur dans un tableau trié :

Un tableau A de dimension N+1 contient N valeurs entières triées par ordre croissant; la (N+1)^{ième} valeur est indéfinie. Insérer une valeur VAL donnée au clavier dans le tableau A de manière à obtenir un tableau de N+1 valeurs triées.

Exercice 7.12 :

Recherche d'une valeur dans un tableau :

Problème :

Rechercher dans un tableau d'entiers A une valeur VAL entrée au clavier. Afficher la position de VAL si elle se trouve dans le tableau, sinon afficher un message correspondant. La valeur POS qui est utilisée pour mémoriser la position de la valeur dans le tableau, aura la valeur -1 aussi longtemps que VAL n'a pas été trouvée.

Implémenter deux versions :

- La recherche séquentielle
Comparer successivement les valeurs du tableau avec la valeur donnée.
- La recherche dichotomique ('recherche binaire', 'binary search')
Condition : Le tableau A doit être trié
Comparer le nombre recherché à la valeur au milieu du tableau,
 - s'il y a égalité ou si le tableau est épuisé, arrêter le traitement avec un message correspondant.
 - si la valeur recherchée précède la valeur actuelle du tableau, continuer la recherche dans le demi tableau à gauche de la position actuelle.

- si la valeur recherchée suit la valeur actuelle du tableau, continuer la recherche dans le demi tableau à droite de la position actuelle.

Ecrire le programme pour le cas où le tableau A est trié par ordre croissant.

Question :

Quel est l'avantage de la recherche dichotomique? Expliquer brièvement.

Exercice 7.13 :

Fusion de deux tableaux triés :

Problème :

On dispose de deux tableaux A et B (de dimensions respectives N et M), triés par ordre croissant. Fusionner les éléments de A et B dans un troisième tableau FUS trié par ordre croissant.

Méthode :

Utiliser trois indices IA, IB et IFUS. Comparer A[IA] et B[IB]; remplacer FUS[IFUS] par le plus petit des deux éléments; avancer dans le tableau FUS et dans le tableau qui a contribué son élément. Lorsque l'un des deux tableaux A ou B est épuisé, il suffit de recopier les éléments restants de l'autre tableau dans le tableau FUS.

Exercice 7.14 :

Tri par sélection du maximum :

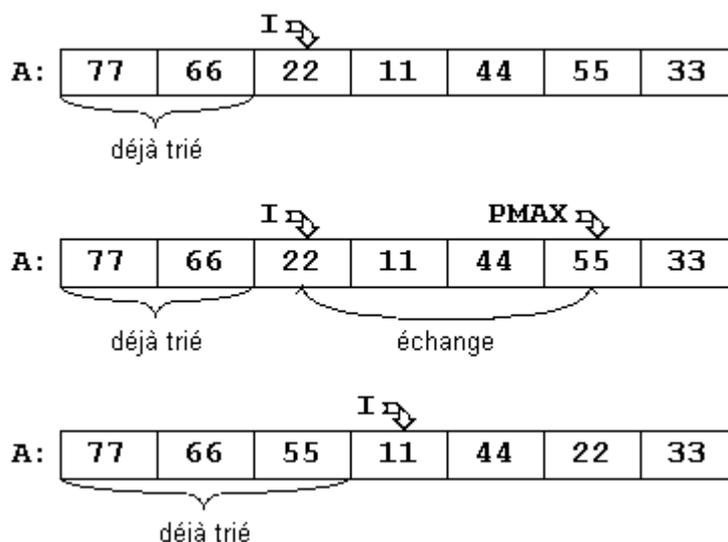
Problème :

Classer les éléments d'un tableau A par ordre décroissant.

Méthode :

Parcourir le tableau de gauche à droite à l'aide de l'indice I. Pour chaque élément A[I] du tableau, déterminer la position PMAX du (premier) maximum à droite de A[I] et échanger A[I] et A[PMAX].

Exemple :



Exercice 7.15 :

Tri par propagation (bubble sort) :

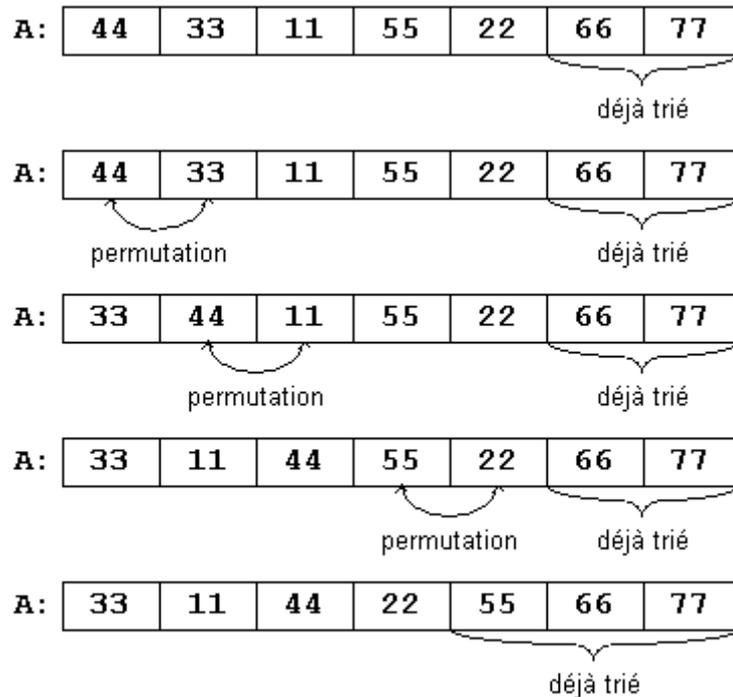
Problème :

Classer les éléments d'un tableau A par ordre croissant.

Méthode :

En recommençant chaque fois au début du tableau, on effectue à plusieurs reprises le traitement suivant : On propage, par permutations successives, le plus grand élément du tableau vers la fin du tableau (comme une bulle qui remonte à la surface d'un liquide).

Exemple :



Implémenter l'algorithme en considérant que :

- * La partie du tableau (à droite) où il n'y a pas eu de permutations est triée.
- * Si aucune permutation n'a eu lieu, le tableau est trié.

Exercice 7.16 :

Statistique des notes :

Ecrire un programme qui lit les points de N élèves d'une classe dans un devoir et les mémorise dans un tableau POINTS de dimension N.

* Rechercher et afficher :

- la note maximale,
- la note minimale,
- la moyenne des notes.

* A partir des POINTS des élèves, établir un tableau NOTES de dimension 7 qui est composé de la façon suivante :

- NOTES [6] contient le nombre de notes 60
- NOTES [5] contient le nombre de notes de 50 à 59
- NOTES [4] contient le nombre de notes de 40 à 49
- ...
- NOTES [0] contient le nombre de notes de 0 à 9

Etablir un graphique de barreaux représentant le tableau NOTES. Utilisez les symboles ##### pour la représentation des barreaux et affichez le domaine des notes en dessous du graphique.

Idée :

Déterminer la valeur maximale MAXN dans le tableau NOTES et afficher autant de lignes sur l'écran. (Dans l'exemple ci-dessous, MAXN = 6).

Exemple :

La note maximale est 58
 La note minimale est 13
 La moyenne des notes est 37.250000

```

6 > #####
5 > #####
4 > #####
3 > #####
2 > #####
1 > #####
+-----+-----+-----+-----+-----+-----+
| 0 - 9 | 10-19 | 20-29 | 30-39 | 40-49 | 50-59 | 60 |

```

2) Tableaux à deux dimensions – Matrices :

Exercice 7.17 :

Mise à zéro de la diagonale principale d'une matrice

Ecrire un programme qui met à zéro les éléments de la diagonale principale d'une matrice *carrée* A donnée.

Exercice 7.18 :

Matrice unitaire

Ecrire un programme qui construit et affiche une matrice *carrée unitaire* U de dimension N. Une matrice unitaire est une matrice, telle que :

$$u_{ij} = \begin{cases} 1 & \text{si } i=j \\ 0 & \text{si } ij \end{cases}$$

Exercice 7.19 :

Transposition d'une matrice

Ecrire un programme qui effectue la transposition t_A d'une matrice A de dimensions N et M en une matrice de dimensions M et N.

- a) La matrice transposée sera mémorisée dans une deuxième matrice B qui sera ensuite affichée.
- b) La matrice A sera transposée par permutation des éléments.

Rappel :

$$tA = \begin{pmatrix} a & e & i \\ b & f & j \\ c & g & k \\ d & h & l \end{pmatrix}$$

Exercice 7.20 :

Multiplication d'une matrice par un réel

Ecrire un programme qui réalise la multiplication d'une matrice A par un réel X.

Rappel :

$$X * \begin{pmatrix} a & b & c & d \\ e & f & g & h \end{pmatrix} = \begin{pmatrix} X*a & X*b & X*c & X*d \\ X*e & X*f & X*g & X*h \end{pmatrix}$$

$$\begin{pmatrix} i & j & k & l \end{pmatrix} \begin{pmatrix} x*i & x*j & x*k & x*l \end{pmatrix}$$

- a) Le résultat de la multiplication sera mémorisé dans une deuxième matrice B qui sera ensuite affichée.
- b) Les éléments de la matrice A seront multipliés par X.

Exercice 7.21 :

Addition de deux matrices :

Ecrire un programme qui réalise l'addition de deux matrices A et B de mêmes dimensions N et M.

Rappel :

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{pmatrix} + \begin{pmatrix} a' & b' & c' & d' \\ e' & f' & g' & h' \\ i' & j' & k' & l' \end{pmatrix} = \begin{pmatrix} a+a' & b+b' & c+c' & d+d' \\ e+e' & f+f' & g+g' & h+h' \\ i+i' & j+j' & k+k' & l+l' \end{pmatrix}$$

- a) Le résultat de l'addition sera mémorisé dans une troisième matrice C qui sera ensuite affichée.
- b) La matrice B est ajoutée à A.

Exercice 7.22 :

Multiplication de deux matrices :

En multipliant une matrice A de dimensions N et M avec une matrice B de dimensions M et P on obtient une matrice C de dimensions N et P :

$$A(N,M) * B(M,P) = C(N,P)$$

La multiplication de deux matrices se fait en multipliant les composantes des deux matrices lignes par colonnes :

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

Rappel :

$$\begin{pmatrix} a & b & c \\ e & f & g \\ h & i & j \\ k & l & m \end{pmatrix} * \begin{pmatrix} p & q \\ r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a*p + b*r + c*t & a*q + b*s + c*u \\ e*p + f*r + g*t & e*q + f*s + g*u \\ h*p + i*r + j*t & h*q + i*s + j*u \\ k*p + l*r + m*t & k*q + l*s + m*u \end{pmatrix}$$

Ecrire un programme qui effectue la multiplication de deux matrices A et B. Le résultat de la multiplication sera mémorisé dans une troisième matrice C qui sera ensuite affichée.

Exercice 7.23 :

Triangle de Pascal :

Ecrire un programme qui construit le triangle de PASCAL de degré N et le mémorise dans une matrice carrée P de dimension N+1.

Exemple :

Triangle de Pascal de degré 6 :

$$\begin{matrix} n=0 & 1 \\ n=1 & 1 & 1 \\ n=2 & 1 & 2 & 1 \end{matrix}$$

n=3	1	3	3	1			
n=4	1	4	6	4	1		
n=5	1	5	10	10	5	1	
n=6	1	6	15	20	15	5	1

Méthode :

Calculer et afficher seulement les valeurs jusqu'à la diagonale principale (incluse). Limiter le degré à entrer par l'utilisateur à 13.

Construire le triangle ligne par ligne :

- Initialiser le premier élément et l'élément de la diagonale à 1.
- Calculer les valeurs entre les éléments initialisés de gauche à droite en utilisant la relation :

$$P_{i,j} = P_{i-1,j} + P_{i-1,j-1}$$

Exercice 7.24 :

Recherche de 'points-cols'

Rechercher dans une matrice donnée A les éléments qui sont à la fois un maximum sur leur ligne et un minimum sur leur colonne. Ces éléments sont appelés des *points-cols*. Afficher les positions et les valeurs de *tous* les points-cols trouvés.

Exemples :

Les éléments soulignés sont des points-cols :

/	1 8 3 4 0	\	/	4 5 8 9	\	/	3 5 6 7 7	\	/	1 2 3	\
6 7 2 7 0	3 8 9 3	4 2 2 8 9	3 4 9 3	6 3 2 9 7	4 5 6	7 8 9					
\	/	\	/	\	/	\					

Méthode :

Etablir deux matrices d'aide MAX et MIN de même dimensions que A, telles que :

$$\text{MAX}[i,j] = \begin{cases} 1 & \text{si } A[i,j] \text{ est un maximum} \\ & \text{sur la ligne } i \\ 0 & \text{sinon} \end{cases}$$

$$\text{MIN}[i,j] = \begin{cases} 1 & \text{si } A[i,j] \text{ est un minimum} \\ & \text{sur la colonne } j \\ 0 & \text{sinon} \end{cases}$$

IV) Solutions des exercices du Chapitre 7 : LES TABLEAUX :

Exercice 7.1 :

```
#include <stdio.h>
main()
{
    /* Déclarations */
    int T[50]; /* tableau donné */
    int N;     /* dimension      */
    int I;     /* indice courant */
    long SOM; /* somme des éléments - type long à cause */
              /* de la grandeur prévisible du résultat. */

    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (I=0; I<N; I++)
    {
        printf("Elément %d : ", I);
        scanf("%d", &T[I]);
    }

    /* Affichage du tableau */
    printf("Tableau donné :\n");
    for (I=0; I<N; I++)
        printf("%d ", T[I]);
    printf("\n");

    /* Calcul de la somme */
    for (SOM=0, I=0; I<N; I++)
        SOM += T[I];

    /* Edition du résultat */
    printf("Somme des éléments : %ld\n", SOM);
    return (0);
}
```

Exercice 7.2 :

```
#include <stdio.h>
main()
{
    /* Déclarations */
    int T[50]; /* tableau donné      */
    int N;     /* dimension      */
    int I,J;   /* indices courants */

    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (I=0; I<N; I++)
    {
        printf("Elément %d : ", I);
        scanf("%d", &T[I]);
    }
}
```

```

/* Affichage du tableau */
printf("Tableau donné : \n");
for (I=0; I<N; I++)
    printf("%d ", T[I]);
printf("\n");
/* Effacer les zéros et comprimer :          */
/* Copier tous les éléments de I vers J et */
/* augmenter J pour les éléments non nuls. */
for (I=0, J=0 ; I<N ; I++)
    {
        T[J] = T[I];
        if (T[I]) J++;
    }
/* Nouvelle dimension du tableau ! */
N = J;
/* Edition des résultats */
printf("Tableau résultat : \n");
for (I=0; I<N; I++)
    printf("%d ", T[I]);
printf("\n");
return (0);
}

```

Exercice 7.3 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int T[50]; /* tableau donné */
    int N;     /* dimension      */
    int I,J;   /* indices courants */
    int AIDE;  /* pour l'échange   */

    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (I=0; I<N; I++)
        {
            printf("Elément %d : ", I);
            scanf("%d", &T[I]);
        }
    /* Affichage du tableau */
    printf("Tableau donné : \n");
    for (I=0; I<N; I++)
        printf("%d ", T[I]);
    printf("\n");
    /* Inverser le tableau */
    for (I=0, J=N-1 ; I<J ; I++,J--)
        /* Echange de T[I] et T[J] */
        {
            AIDE = T[I];
            T[I] = T[J];
            T[J] = AIDE;
        }
}

```

```

    }
    /* Edition des résultats */
    printf("Tableau résultat :\n");
    for (I=0; I<N; I++)
        printf("%d ", T[I]);
    printf("\n");
    return (0);
}

```

Exercice 7.4 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    /* Les tableaux et leurs dimensions */
    int T[50], TPOS[50], TNEG[50];
    int N,      NPOS,      NNEG;
    int I; /* indice courant */

    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (I=0; I<N; I++)
    {
        printf("Elément %d : ", I);
        scanf("%d", &T[I]);
    }
    /* Affichage du tableau */
    printf("Tableau donné :\n");
    for (I=0; I<N; I++)
        printf("%d ", T[I]);
    printf("\n");
    /* Initialisation des dimensions de TPOS et TNEG */
    NPOS=0;
    NNEG=0;
    /* Transfer des données */
    for (I=0; I<N; I++)
        { if (T[I]>0) {
                TPOS [NPOS]=T[I];
                NPOS++;
            }
          if (T[I]<0) {
                TNEG [NNEG]=T[I];
                NNEG++;
            }
        }
    /* Edition du résultat */
    printf("Tableau TPOS :\n");
    for (I=0; I<NPOS; I++)
        printf("%d ", TPOS[I]);
    printf("\n");
    printf("Tableau TNEG :\n");
    for (I=0; I<NNEG; I++)

```

```

        printf("%d ", TNEG[I]);
    printf("\n");
    return (0);
}

```

Exercice 7.5 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int T[50][50]; /* tableau donné */
    int L, C; /* dimensions */
    int I, J; /* indices courants */
    long SOM; /* somme des éléments - type long à cause */
              /* de la grandeur prévisible du résultat. */

    /* Saisie des données */
    printf("Nombre de lignes (max.50) : ");
    scanf("%d", &L );
    printf("Nombre de colonnes (max.50) : ");
    scanf("%d", &C );
    for (I=0; I<L; I++)
        for (J=0; J<C; J++)
            {
                printf("Elément [%d] [%d] : ", I, J);
                scanf("%d", &T[I][J]);
            }
    /* Affichage du tableau */
    printf("Tableau donné :\n");
    for (I=0; I<L; I++)
        {
            for (J=0; J<C; J++)
                printf("%7d", T[I][J]);
            printf("\n");
        }
    /* Calcul de la somme */
    for (SOM=0, I=0; I<L; I++)
        for (J=0; J<C; J++)
            SOM += T[I][J];
    /* Edition du résultat */
    printf("Somme des éléments : %ld\n", SOM);
    return (0);
}

```

Exercice 7.6 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int T[50][50]; /* tableau donné */
    int L, C; /* dimensions */
    int I, J; /* indices courants */

```

```

long SOM; /* somme des éléments - type long à cause */
          /* de la grandeur prévisible des résultats. */

/* Saisie des données */
printf("Nombre de lignes (max.50) : ");
scanf("%d", &L );
printf("Nombre de colonnes (max.50) : ");
scanf("%d", &C );
for (I=0; I<L; I++)
    for (J=0; J<C; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &T[I] [J]);
        }
/* Affichage du tableau */
printf("Tableau donné :\n");
for (I=0; I<L; I++)
    {
        for (J=0; J<C; J++)
            printf("%7d", T[I] [J]);
        printf("\n");
    }
/* Calcul et affichage de la somme des lignes */
for (I=0; I<L; I++)
    {
        for (SOM=0, J=0; J<C; J++)
            SOM += T[I] [J];
        printf("Somme - ligne %d : %ld\n", I, SOM);
    }
/* Calcul et affichage de la somme des colonnes */
for (J=0; J<C; J++)
    {
        for (SOM=0, I=0; I<L; I++)
            SOM += T[I] [J];
        printf("Somme - colonne %d : %ld\n", J, SOM);
    }
return (0);
}

```

Exercice 7.7 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int M[10][10]; /* tableau à 2 dimensions */
    int V[100];    /* tableau à 1 dimension */
    int L, C;     /* dimensions */
    int I, J;     /* indices courants */

    /* Saisie des données */
    printf("Nombre de lignes (max.10) : ");
    scanf("%d", &L );
    printf("Nombre de colonnes (max.10) : ");

```

```

scanf("%d", &C );
for (I=0; I<L; I++)
    for (J=0; J<C; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &M[I] [J]);
        }
/* Affichage du tableau 2-dim */
printf("Tableau donné :\n");
for (I=0; I<L; I++)
    {
        for (J=0; J<C; J++)
            printf("%7d", M[I] [J]);
        printf("\n");
    }
/* Transfer des éléments ligne par ligne */
for (I=0; I<L; I++)
    for (J=0; J<C; J++)
        V[I*C+J] = M[I] [J];
/* Affichage du tableau 1-dim */
printf("Tableau résultat : ");
for (I=0; I<L*C; I++)
    printf("%d ", V[I]);
printf("\n");
return (0);
}

```

1) Tableaux à une dimension – Vecteurs :

Exercice 7.8 :

Produit scalaire de deux vecteurs

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int U[50], V[50]; /* tableaux donnés */
    int N;           /* dimension          */
    int I;           /* indice courant      */
    long PS;        /* produit scalaire    */
    /* Saisie des données */
    printf("Dimension des tableaux (max.50) : ");
    scanf("%d", &N );
    printf("*** Premier tableau **\n");
    for (I=0; I<N; I++)
        {
            printf("Elément %d : ", I);
            scanf("%d", &U[I]);
        }
    printf("*** Deuxième tableau **\n");
    for (I=0; I<N; I++)
        {
            printf("Elément %d : ", I);

```

```

        scanf("%d", &V[I]);
    }
    /* Calcul du produit scalaire */
    for (PS=0, I=0; I<N; I++)
        PS += (long)U[I]*V[I];
    /* Edition du résultat */
    printf("Produit scalaire : %ld\n", PS);
    return (0);
}

```

Exercice 7.9 :

Calcul d'un polynôme de degré N

Solution combinée :

```

#include <stdio.h>
#include <math.h>
main()
{
    float A[20]; /* tableau des coefficients de P */
    int I;      /* indice courant */
    int N;      /* degré du polynôme */
    float X;    /* argument */
    float P;    /* résultat */

    /* Saisie du degré N et de l'argument X */
    printf("Entrer le degré N du polynôme (max.20) : ");
    scanf("%d", &N);
    printf("Entrer la valeur X de l'argument : ");
    scanf("%f", &X);
    /* Saisie des coefficients */
    for (I=0 ; I<N ; I++)
    {
        printf("Entrer le coefficient A%d : ", I);
        scanf("%f", &A[I]);
    }

    /* a) Calcul à l'aide de pow
    for (P=0.0, I=0 ; I<N+1 ; I++)
        P += A[I]*pow(X,I); */

    /* b) Calcul de Horner */
    for (P=0.0, I=0 ; I<N+1 ; I++)
        P = P*X + A[I];

    /* Edition du résultat */
    printf("Valeur du polynôme pour X = %.2f : %.2f\n", X, P);
    return (0);
}

```

Exercice 7.10 :

Maximum et minimum des valeurs d'un tableau

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

```

main()
{
    /* Déclarations */
    int A[50]; /* tableau donné */
    int N;     /* dimension     */
    int I;     /* indice courant */
    int MIN;   /* position du minimum */
    int MAX;   /* position du maximum */
    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (I=0; I<N; I++)
        {
            printf("Elément %d : ", I);
            scanf("%d", &A[I]);
        }
    /* Affichage du tableau */
    printf("Tableau donné :\n");
    for (I=0; I<N; I++)
        printf("%d ", A[I]);
    printf("\n");
    /* Recherche du maximum et du minimum */
    MIN=0;
    MAX=0;
    for (I=0; I<N; I++)
        {
            if (A[I]>A[MAX]) MAX=I;
            if (A[I]<A[MIN]) MIN=I;
        }
    /* Edition du résultat */
    printf("Position du minimum : %d\n", MIN);
    printf("Position du maximum : %d\n", MAX);
    printf("Valeur du minimum : %d\n", A[MIN]);
    printf("Valeur du maximum : %d\n", A[MAX]);
    return (0);
}

```

Exercice 7.11 :

Insérer une valeur dans un tableau trié

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50]; /* tableau donné */
    int VAL;   /* valeur à insérer */
    int N;     /* dimension     */
    int I;     /* indice courant */

    /* Saisie des données */
    printf("Dimension N du tableau initial (max.50) : ");
    scanf("%d", &N );
    for (I=0; I<N; I++)
        {

```

```

        printf("Elément %d : ", I);
        scanf("%d", &A[I]);
    }
    printf("Elément à insérer : ");
    scanf("%d", &VAL );
    /* Affichage du tableau */
    printf("Tableau donné : \n");
    for (I=0; I<N; I++)
        printf("%d ", A[I]);
    printf("\n");
    /* Déplacer les éléments plus grands que */
    /* VAL d'une position vers l'arrière.      */
    for (I=N ; (I>0)&&(A[I-1]>VAL) ; I--)
        A[I]=A[I-1];
    /* VAL est copié à la position du dernier */
    /* élément déplacé. */
    A[I]=VAL;
    /* Nouvelle dimension du tableau ! */
    N++;
    /* Edition des résultats */
    printf("Tableau résultat : \n");
    for (I=0; I<N; I++)
        printf("%d ", A[I]);
    printf("\n");
    return (0);
}

```

Exercice 7.12 :

Recherche d'une valeur dans un tableau

a) La recherche séquentielle

Comparer successivement les valeurs du tableau avec la valeur donnée.

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50]; /* tableau donné */
    int VAL;   /* valeur à rechercher */
    int POS;   /* position de la valeur */
    int N;     /* dimension */
    int I;     /* indice courant */

    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (I=0; I<N; I++)
    {
        printf("Elément %d : ", I);
        scanf("%d", &A[I]);
    }
    printf("Elément à rechercher : ");
    scanf("%d", &VAL );
    /* Affichage du tableau */
    printf("Tableau donné : \n");

```

```

for (I=0; I<N; I++)
    printf("%d ", A[I]);
printf("\n");
/* Recherche de la position de la valeur */
POS = -1;
for (I=0 ; (I<N)&&(POS== -1) ; I++)
    if (A[I]==VAL)
        POS=I;
/* Edition du résultat */
if (POS== -1)
    printf("La valeur recherchée ne se trouve pas "
           "dans le tableau.\n");
else
    printf("La valeur %d se trouve à la position %d. \n",
           VAL, POS);
return (0);
}

```

b) La recherche dichotomique ('recherche binaire', 'binary search')

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50]; /* tableau donné */
    int VAL;   /* valeur à rechercher */
    int POS;   /* position de la valeur */
    int N;     /* dimension */
    int I;     /* indice courant */
    int INF, MIL, SUP; /* limites du champ de recherche */

    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (I=0; I<N; I++)
    {
        printf("Elément %d : ", I);
        scanf("%d", &A[I]);
    }
    printf("Elément à rechercher : ");
    scanf("%d", &VAL );
    /* Affichage du tableau */
    printf("Tableau donné : \n");
    for (I=0; I<N; I++)
        printf("%d ", A[I]);
    printf("\n");
    /* Initialisation des limites du domaine de recherche */
    INF=0;
    SUP=N-1;
    /* Recherche de la position de la valeur */
    POS=-1;
    while ((INF<=SUP) && (POS== -1))
    {
        MIL=(SUP+INF)/2;
        if (VAL < A[MIL])
            SUP=MIL-1;
    }
}

```

```

        else if (VAL > A[MIL])
            INF=MIL+1;
        else
            POS=MIL;
    }

    /* Edition du résultat */
    if (POS== -1)
        printf("La valeur recherchée ne se trouve pas "
            "dans le tableau.\n");
    else
        printf("La valeur %d se trouve à la position %d. \n",
            VAL, POS);
    return (0);
}

```

Question :

Quel est l'avantage de la recherche dichotomique?

Dans le pire des cas d'une recherche séquentielle, il faut traverser tout le tableau avant de trouver la valeur ou avant d'être sûr qu'une valeur ne se trouve pas dans le tableau.

Lors de la recherche dichotomique, on élimine la moitié des éléments du tableau à chaque exécution de la boucle. Ainsi, la recherche se termine beaucoup plus rapidement.

La recherche dichotomique devient extrêmement avantageuse pour la recherche dans de grands tableaux (triés) : L'avantage de la recherche dichotomique par rapport à la recherche séquentielle monte alors exponentiellement avec la grandeur du tableau à trier.

Exemple :

Lors de la recherche dans un tableau de 1024 éléments :

- le pire des cas pour la recherche séquentielle peut entraîner 1024 exécutions de la boucle.
- le pire des cas pour la recherche dichotomique peut entraîner 10 exécutions de la boucle.

Lors de la recherche dans un tableau de 1 048 576 éléments :

- le pire des cas pour la recherche séquentielle peut entraîner 1 048 576 exécutions de la boucle.
- le pire des cas pour la recherche dichotomique peut entraîner 20 exécutions de la boucle.

Exercice 7.13 :

Fusion de deux tableaux triés

```

#include <stdio.h>
main()
{
    /* Déclarations */
    /* Les tableaux et leurs dimensions */
    int A[50], B[50], FUS[100];
    int N, M;
    int IA, IB, IFUS; /* indices courants */

    /* Saisie des données */
    printf("Dimension du tableau A (max.50) : ");
    scanf("%d", &N );
    printf("Entrer les éléments de A dans l'ordre croissant
:\n");
    for (IA=0; IA<N; IA++)
        {

```

```

        printf("Elément A[%d] : ", IA);
        scanf("%d", &A[IA]);
    }
    printf("Dimension du tableau B (max.50) : ");
    scanf("%d", &M );
    printf("Entrer les éléments de B dans l'ordre croissant
:\n");
    for (IB=0; IB<M; IB++)
    {
        printf("Elément B[%d] : ", IB);
        scanf("%d", &B[IB]);
    }
    /* Affichage des tableaux A et B */
    printf("Tableau A :\n");
    for (IA=0; IA<N; IA++)
        printf("%d ", A[IA]);
    printf("\n");
    printf("Tableau B :\n");
    for (IB=0; IB<M; IB++)
        printf("%d ", B[IB]);
    printf("\n");

/* Fusion des éléments de A et B dans FUS */
/* de façon à ce que FUS soit aussi trié. */
IA=0; IB=0; IFUS=0;
while ((IA<N) && (IB<M))
    if (A[IA]<B[IB])
    {
        FUS[IFUS]=A[IA];
        IFUS++;
        IA++;
    }
    else
    {
        FUS[IFUS]=B[IB];
        IFUS++;
        IB++;
    }
/* Si IA ou IB sont arrivés à la fin de leur tableau, */
/* alors copier le reste de l'autre tableau. */
while (IA<N)
    {
        FUS[IFUS]=A[IA];
        IFUS++;
        IA++;
    }
while (IB<M)
    {
        FUS[IFUS]=B[IB];
        IFUS++;
        IB++;
    }

/* Edition du résultat */

```

```

printf("Tableau FUS :\n");
for (IFUS=0; IFUS<N+M; IFUS++)
    printf("%d ", FUS[IFUS]);
printf("\n");
return (0);
}

```

Exercice 7.14 :

Tri par sélection du maximum

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50]; /* tableau donné */
    int N;     /* dimension      */
    int I;     /* rang à partir duquel A n'est pas trié */
    int J;     /* indice courant      */
    int AIDE;  /* pour la permutation */
    int PMAX;  /* indique la position de l'élément */
                /* maximal à droite de A[I]      */

    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (J=0; J<N; J++)
    {
        printf("Elément %d : ", J);
        scanf("%d", &A[J]);
    }
    /* Affichage du tableau */
    printf("Tableau donné :\n");
    for (J=0; J<N; J++)
        printf("%d ", A[J]);
    printf("\n");

    /* Tri du tableau par sélection directe du maximum. */
    for (I=0; I<N-1; I++)
    {
        /* Recherche du maximum à droite de A[I] */
        PMAX=I;
        for (J=I+1; J<N; J++)
            if (A[J]>A[PMAX]) PMAX=J;
        /* Echange de A[I] avec le maximum */
        AIDE=A[I];
        A[I]=A[PMAX];
        A[PMAX]=AIDE;
    }

    /* Edition du résultat */
    printf("Tableau trié :\n");
    for (J=0; J<N; J++)
        printf("%d ", A[J]);
    printf("\n");
}

```

```

    return (0);
}

```

Exercice 7.15 :

Tri par propagation (bubble sort)

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50]; /* tableau donné */
    int N;     /* dimension      */
    int I;     /* rang à partir duquel A est trié */
    int J;     /* indice courant      */
    int AIDE;  /* pour la permutation */
    int FIN;   /* position où la dernière permutation a eu lieu.
*/
                /* permet de ne pas trier un sous-ensemble déjà trié.
*/

    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (J=0; J<N; J++)
    {
        printf("Élément %d : ", J);
        scanf("%d", &A[J]);
    }
    /* Affichage du tableau */
    printf("Tableau donné :\n");
    for (J=0; J<N; J++)
        printf("%d ", A[J]);
    printf("\n");

    /* Tri du tableau par propagation de l'élément maximal. */
    for (I=N-1 ; I>0 ; I=FIN)
    {
        FIN=0;
        for (J=0; J<I; J++)
            if (A[J]>A[J+1])
            {
                FIN=J;
                AIDE=A[J];
                A[J]=A[J+1];
                A[J+1]=AIDE;
            }
    }

    /* Edition du résultat */
    printf("Tableau trié :\n");
    for (J=0; J<N; J++)
        printf("%d ", A[J]);
    printf("\n");
    return (0);
}

```

```
}
```

Exercice 7.16 :

Statistique des notes

```
#include <stdio.h>
main()
{
    int POINTS[50]; /* tableau des points */
    int NOTES[7]; /* tableau des notes */
    int N; /* nombre d'élèves */
    int I, IN; /* compteurs d'aide */
    int SOM; /* somme des points */
    int MAX, MIN; /* maximum, minimum de points */
    int MAXN; /* nombre de lignes du graphique */

    /* Saisie des données */
    printf("Entrez le nombre d'élèves (max.50) : ");
    scanf("%d", &N);
    printf("Entrez les points des élèves :\n");
    for (I=0; I<N; I++)
        {printf("Elève %d :", I+1);
         scanf("%d", &POINTS[I]);
        }
    printf("\n");

    /* Calcul et affichage du maximum et du minimum des points */
    for (MAX=0, MIN=60, I=0; I<N; I++)
        {if (POINTS[I] > MAX) MAX=POINTS[I];
         if (POINTS[I] < MIN) MIN=POINTS[I];
        }
    printf("La note maximale est %d \n", MAX);
    printf("La note minimale est %d \n", MIN);
    /* Calcul et affichage de la moyenne des points */
    for (SOM=0, I=0 ; I<N ; I++)
        SOM += POINTS[I];
    printf("La moyenne des notes est %f \n", (float)SOM/N);

    /* Etablissement du tableau NOTES */
    for (IN=0 ; IN<7 ; IN++)
        NOTES[IN] = 0;
    for (I=0; I<N; I++)
        NOTES[POINTS[I]/10]++;

    /* Recherche du maximum MAXN dans NOTES */
    for (MAXN=0, IN=0 ; IN<7 ; IN++)
        if (NOTES[IN] > MAXN)
            MAXN = NOTES[IN];

    /* Affichage du graphique de barreaux */
    /* Représentation de MAXN lignes */
    for (I=MAXN; I>0; I--)
        {
```

```

printf("\n %2d >", I);
for (IN=0; IN<7; IN++)
{
    if (NOTES[IN]>=I)
        printf(" #####");
    else
        printf("          ");
}
}

/* Affichage du domaine des notes */
printf("\n          +");
for (IN=0; IN<7; IN++)
    printf("-----+");
printf("\n          I 0 - 9 I 10-19 I 20-29 "
        "I 30-39 I 40-49 I 50-59 I 60   I\n");
return (0);
}

```

2) Tableaux à deux dimensions – Matrices :

Exercice 7.17 :

Mise à zéro de la diagonale principale d'une matrice

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50][50]; /* matrice carrée */
    int N;         /* dimension de la matrice carrée */
    int I, J;      /* indices courants */

    /* Saisie des données */
    printf("Dimension de la matrice carrée (max.50) : ");
    scanf("%d", &N);
    for (I=0; I<N; I++)
        for (J=0; J<N; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &A[I][J]);
        }

    /* Affichage de la matrice */
    printf("Matrice donnée :\n");
    for (I=0; I<N; I++)
    {
        for (J=0; J<N; J++)
            printf("%7d", A[I][J]);
        printf("\n");
    }

    /* Mise à zéro de la diagonale principale */
    for (I=0; I<N; I++)
        A[I][I]=0;
}

```

```

/* Edition du résultat */
printf("Matrice résultat :\n");
for (I=0; I<N; I++)
{
    for (J=0; J<N; J++)
        printf("%7d", A[I][J]);
    printf("\n");
}
return (0);
}

```

Exercice 7.18 :

Matrice unitaire

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int U[50][50]; /* matrice unitaire */
    int N;          /* dimension de la matrice unitaire */
    int I, J;       /* indices courants */

    /* Saisie des données */
    printf("Dimension de la matrice carrée (max.50) : ");
    scanf("%d", &N);

    /* Construction de la matrice carrée unitaire */
    for (I=0; I<N; I++)
        for (J=0; J<N; J++)
            if (I==J)
                U[I][J]=1;
            else
                U[I][J]=0;

    /* Edition du résultat */
    printf("Matrice unitaire de dimension %d :\n", N);
    for (I=0; I<N; I++)
    {
        for (J=0; J<N; J++)
            printf("%7d", U[I][J]);
        printf("\n");
    }
    return (0);
}

```

Remarque :

L'opération :

```

if (I==J)
    U[I][J]=1;
else
    U[I][J]=0;

```

peut être simplifiée par

```

U[I][J] = (I==J);

```

Exercice 7.19 :

Transposition d'une matrice

- a) La matrice transposée sera mémorisée dans une deuxième matrice B qui sera ensuite affichée.

```
#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50][50]; /* matrice initiale */
    int B[50][50]; /* matrice résultat */
    int N, M;      /* dimensions des matrices */
    int I, J;      /* indices courants */

    /* Saisie des données */
    printf("Nombre de lignes (max.50) : ");
    scanf("%d", &N );
    printf("Nombre de colonnes (max.50) : ");
    scanf("%d", &M );
    for (I=0; I<N; I++)
        for (J=0; J<M; J++)
            {
                printf("Elément [%d] [%d] : ", I, J);
                scanf("%d", &A[I][J]);
            }

    /* Affichage de la matrice */
    printf("Matrice donnée :\n");
    for (I=0; I<N; I++)
        {
            for (J=0; J<M; J++)
                printf("%7d", A[I][J]);
            printf("\n");
        }

    /* Affectation de la matrice transposée à B */
    for (I=0; I<N; I++)
        for (J=0; J<M; J++)
            B[J][I]=A[I][J];

    /* Edition du résultat */
    /* Attention : maintenant le rôle de N et M est inversé. */
    printf("Matrice résultat :\n");
    for (I=0; I<M; I++)
        {
            for (J=0; J<N; J++)
                printf("%7d", B[I][J]);
            printf("\n");
        }
    return (0);
}
```

- b) La matrice A sera transposée par permutation des éléments.

```
#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50][50]; /* matrice donnée */
```

```

int N, M;          /* dimensions de la matrice */
int I, J;          /* indices courants      */
int AIDE;         /* pour la permutation   */
int DMAX;         /* la plus grande des deux dimensions */

/* Saisie des données */
printf("Nombre de lignes (max.50) : ");
scanf("%d", &N );
printf("Nombre de colonnes (max.50) : ");
scanf("%d", &M );
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &A[I] [J]);
        }
/* Affichage de la matrice */
printf("Matrice donnée :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", A[I] [J]);
        printf("\n");
    }
/* Transposition de la matrice A par permutation des */
/* éléments [I] [J] à gauche de la diagonale principale */
/* avec les éléments [J] [I] à droite de la diagonale. */
DMAX = (N>M) ? N : M;
for (I=0; I<DMAX; I++)
    for (J=0; J<I; J++)
        {
            AIDE = A[I] [J];
            A[I] [J] = A[J] [I];
            A[J] [I] = AIDE;
        }
/* Edition du résultat */
/* Attention : maintenant le rôle de N et M est inversé. */
printf("Matrice résultat :\n");
for (I=0; I<M; I++)
    {
        for (J=0; J<N; J++)
            printf("%7d", A[I] [J]);
        printf("\n");
    }
return (0);
}

```

Exercice 7.20 :

Multiplication d'une matrice par un réel

a) Le résultat de la multiplication sera mémorisé dans une deuxième matrice A qui sera ensuite affichée.

```

#include <stdio.h>
main()

```

```

{
  /* Déclarations */
  float A[50][50]; /* matrice donnée */
  float B[50][50]; /* matrice résultat */
  int N, M;        /* dimensions des matrices */
  int I, J;        /* indices courants */
  float X;         /* multiplicateur */
  /* Saisie des données */
  printf("Nombre de lignes (max.50) : ");
  scanf("%d", &N );
  printf("Nombre de colonnes (max.50) : ");
  scanf("%d", &M );
  for (I=0; I<N; I++)
    for (J=0; J<M; J++)
      {
        printf("Elément [%d] [%d] : ", I, J);
        scanf("%f", &A[I][J]);
      }
  printf("Multiplicateur X : ");
  scanf("%f", &X );
  /* Affichage de la matrice */
  printf("Matrice donnée :\n");
  for (I=0; I<N; I++)
    {
      for (J=0; J<M; J++)
        printf("%10.2f", A[I][J]);
      printf("\n");
    }
  /* Affectation du résultat de la multiplication à B */
  for (I=0; I<N; I++)
    for (J=0; J<M; J++)
      B[I][J] = X*A[I][J];
  /* Edition du résultat */
  printf("Matrice résultat :\n");
  for (I=0; I<N; I++)
    {
      for (J=0; J<M; J++)
        printf("%10.2f", B[I][J]);
      printf("\n");
    }
  return (0);
}

```

b) Les éléments de la matrice A seront multipliés par X.

```

#include <stdio.h>
main()
{
  /* Déclarations */
  float A[50][50]; /* matrice donnée */
  int N, M;        /* dimensions de la matrice */
  int I, J;        /* indices courants */
  float X;         /* multiplicateur */

  /* Saisie des données */
  printf("Nombre de lignes (max.50) : ");

```

```

scanf("%d", &N );
printf("Nombre de colonnes (max.50) : ");
scanf("%d", &M );
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%f", &A[I][J]);
        }
printf("Multiplicateur X : ");
scanf("%f", &X);
/* Affichage de la matrice */
printf("Matrice donnée :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%10.2f", A[I][J]);
        printf("\n");
    }
/* Multiplication des éléments de A par X */
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        A[I][J] *= X;
/* Edition du résultat */
printf("Matrice résultat :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%10.2f", A[I][J]);
        printf("\n");
    }
return (0);
}

```

Exercice 7.21 :

Addition de deux matrices

- a) Le résultat de l'addition sera mémorisé dans une troisième matrice C qui sera ensuite affichée.

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50][50]; /* matrice donnée */
    int B[50][50]; /* matrice donnée */
    int C[50][50]; /* matrice résultat */
    int N, M; /* dimensions des matrices */
    int I, J; /* indices courants */

    /* Saisie des données */
    printf("Nombre de lignes (max.50) : ");
    scanf("%d", &N );
    printf("Nombre de colonnes (max.50) : ");
    scanf("%d", &M );
    printf("*** Matrice A ***\n");

```

```

for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &A[I][J]);
        }
printf("*** Matrice B ***\n");
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &B[I][J]);
        }
/* Affichage des matrices */
printf("Matrice donnée A :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", A[I][J]);
        printf("\n");
    }
printf("Matrice donnée B :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", B[I][J]);
        printf("\n");
    }

/* Affectation du résultat de l'addition à C */
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        C[I][J] = A[I][J]+B[I][J];
/* Edition du résultat */
printf("Matrice résultat C :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", C[I][J]);
        printf("\n");
    }
return (0);
}

```

b) La matrice B est ajoutée à A.

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50][50]; /* matrice donnée et résultat */
    int B[50][50]; /* matrice donnée */
    int N, M;      /* dimensions des matrices */
    int I, J;      /* indices courants */

    /* Saisie des données */

```

```

printf("Nombre de lignes (max.50) : ");
scanf("%d", &N );
printf("Nombre de colonnes (max.50) : ");
scanf("%d", &M );
printf("*** Matrice A ***\n");
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &A[I] [J]);
        }
printf("*** Matrice B ***\n");
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &B[I] [J]);
        }
/* Affichage des matrices */
printf("Matrice donnée A :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", A[I] [J]);
        printf("\n");
    }

printf("Matrice donnée B :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", B[I] [J]);
        printf("\n");
    }
/* Addition de B à A */
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        A[I] [J] += B[I] [J];
/* Edition du résultat */
printf("Matrice résultat A :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", A[I] [J]);
        printf("\n");
    }
return (0);
}

```

Exercice 7.22 :

Multiplication de deux matrices

```

#include <stdio.h>
main()

```

```

{
/* Déclarations */
int A[50][50]; /* matrice donnée */
int B[50][50]; /* matrice donnée */
int C[50][50]; /* matrice résultat */
int N, M, P; /* dimensions des matrices */
int I, J, K; /* indices courants */

/* Saisie des données */
printf("*** Matrice A ***\n");
printf("Nombre de lignes de A (max.50) : ");
scanf("%d", &N );
printf("Nombre de colonnes de A (max.50) : ");
scanf("%d", &M );
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &A[I][J]);
        }
printf("*** Matrice B ***\n");
printf("Nombre de lignes de B : %d\n", M);
printf("Nombre de colonnes de B (max.50) : ");
scanf("%d", &P );
for (I=0; I<M; I++)
    for (J=0; J<P; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &B[I][J]);
        }
/* Affichage des matrices */
printf("Matrice donnée A :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", A[I][J]);
        printf("\n");
    }
printf("Matrice donnée B :\n");
for (I=0; I<M; I++)
    {
        for (J=0; J<P; J++)
            printf("%7d", B[I][J]);
        printf("\n");
    }
/* Affectation du résultat de la multiplication à C */
for (I=0; I<N; I++)
    for (J=0; J<P; J++)
        {
            C[I][J]=0;
            for (K=0; K<M; K++)
                C[I][J] += A[I][K]*B[K][J];
        }
/* Edition du résultat */

```

```

printf("Matrice résultat C :\n");
for (I=0; I<N; I++)
{
    for (J=0; J<P; J++)
        printf("%7d", C[I][J]);
    printf("\n");
}
return (0);
}

```

Exercice 7.23 :

Triangle de Pascal

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int P[14][14]; /* matrice résultat */
    int N;          /* degré du triangle */
    int I, J;       /* indices courants */
    /* Saisie des données */
    do {
        printf("Entrez le degré N du triangle (max.13) : ");
        scanf("%d", &N);
    } while (N>13 || N<0);
    /* Construction des lignes 0 à N du triangle : */
    /* Calcul des composantes du triangle jusqu'à */
    /* la diagonale principale. */
    for (I=0; I<=N; I++)
    {
        P[I][I]=1;
        P[I][0]=1;
        for (J=1; J<I; J++)
            P[I][J] = P[I-1][J] + P[I-1][J-1];
    }
    /* Edition du résultat */
    printf("Triangle de Pascal de degré %d :\n", N);
    for (I=0; I<=N; I++)
    {
        printf(" N=%2d", I);
        for (J=0; J<=I; J++)
            if (P[I][J])
                printf("%5d", P[I][J]);
        printf("\n");
    }
    return (0);
}

```

Exercice 7.24 :

Recherche de 'points-cols'

```

#include <stdio.h>
main()
{

```

```

/* Déclarations */
int A[50][50]; /* matrice donnée */
int MAX[50][50]; /* matrice indiquant les maxima des lignes
*/
int MIN[50][50]; /* matrice indiquant les minima des colonnes
*/
int N, M; /* dimensions des matrices */
int I, J; /* indices courants */
int AIDE; /* pour la détection des extréma */
int C; /* compteur des points-cols */

/* Saisie des données */
printf("Nombre de lignes (max.50) : ");
scanf("%d", &N );
printf("Nombre de colonnes (max.50) : ");
scanf("%d", &M );
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", &A[I][J]);
        }
/* Affichage de la matrice */
printf("Matrice donnée :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", A[I][J]);
        printf("\n");
    }

/* Construction de la matrice d'aide MAX */
/* qui indique les positions de tous les */
/* maxima sur une ligne. */
for (I=0; I<N; I++)
    {
        /* Recherche du maximum sur la ligne I */
        AIDE=A[I][0];
        for (J=1; J<M; J++)
            if (A[I][J]>AIDE) AIDE=A[I][J];
        /* Marquage de tous les maxima sur la ligne */
        for (J=0; J<M; J++)
            if (A[I][J]==AIDE) /* ou bien :
*/
                MAX[I][J]=1; /* MAX[I][J] = (A[I][J]==AIDE)
*/
            else
                MAX[I][J]=0;
        }
/* Construction de la matrice d'aide MIN */
/* qui indique les positions de tous les */
/* minima sur une colonne. */
for (J=0; J<M; J++)
    {

```

```

/* Recherche du minimum sur la colonne J */
AIDE=A[0][J];
for (I=1; I<N; I++)
    if (A[I][J]<AIDE) AIDE=A[I][J];
/* Marquage de tous les minima sur la colonne J */
for (I=0; I<N; I++)
    if (A[I][J]==AIDE) /* ou bien : */
        MIN[I][J]=1; /* MIN[I][J] = (A[I][J]==AIDE) */
    else
        MIN[I][J]=0;
}

/* Edition du résultat */
/* Les composantes qui sont marquées comme extréma */
/* dans MAX et dans MIN sont des points-cols. */
printf("Points - cols :\n");
for (C=0, I=0; I<N; I++)
    for (J=0; J<M; J++)
        if (MAX[I][J]&&MIN[I][J])
            {
                C++;
                printf("L'élément %d\test un maximum "
                    "sur la ligne %d\n"
                    "          \t et un minimum "
                    "sur la colonne %d\n", A[I][J], I, J);
            }
if (C==0)
    printf("Le tableau ne contient pas de points-cols.\n");
return (0);
}

```

Chapitre 8 : LES CHAÎNES DE CARACTÈRES :

Il n'existe pas de type spécial chaîne ou *string* en C. Une chaîne de caractères est traitée comme un *tableau à une dimension de caractères* (vecteur de caractères). Il existe quand même des notations particulières et une bonne quantité de fonctions spéciales pour le traitement de tableaux de caractères.

I) Déclaration et mémorisation :

1) Déclaration :

Déclaration de chaînes de caractères en langage algorithmique :

```
chaîne <NomVariable>
```

Déclaration de chaînes de caractères en C :

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

Exemples :

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

Espace à réserver :

Lors de la déclaration, nous devons indiquer l'espace à réserver en mémoire pour le stockage de la chaîne.

La représentation interne d'une chaîne de caractères est terminée par le symbole '\0' (NUL). Ainsi, pour un texte de **n** caractères, nous devons prévoir **n+1** octets.

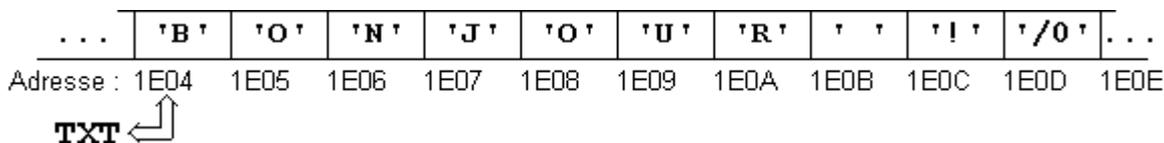
Malheureusement, le compilateur C ne contrôle pas si nous avons réservé un octet pour le symbole de fin de chaîne; l'erreur se fera seulement remarquer lors de l'exécution du programme ...

2) Mémorisation :

Le nom d'une chaîne est le représentant de *l'adresse du premier caractère* de la chaîne. Pour mémoriser une variable qui doit être capable de contenir un texte de N caractères, nous avons besoin de N+1 octets en mémoire :

Exemple : Mémorisation d'un tableau :

```
char TXT[10] = "BONJOUR !";
```



Perspectives

Pour l'avenir, la ISO (*International Organization for Standardization*) prévoit un code de caractères multi-octets extensible (*UNICODE*) qui contiendra tous les caractères et symboles spéciaux utilisés sur toute la terre. Lors de l'introduction de ce code, les méthodes de mémorisation de la plupart des langages de programmation devront être révisées.

II) Les chaînes de caractères constantes :

- * 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 :

"Ce \ntexte \nsera réparti sur 3 lignes."

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

"Affichage de \"guillemets\" \n"

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

{'l', '\'', 'a', 's', 't', 'u', 'c', 'e', '\0'}

* 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 :

"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.

Observation :

Pour la mémorisation de la chaîne de caractères "Hello", C a besoin de **six (!)** octets.

'x' est un *caractère constant*, qui a une valeur numérique :

P.ex : 'x' a la valeur 120 dans le code ASCII.

"x" est un *tableau de caractères* qui contient deux caractères :

la lettre 'x' et le caractère NUL : '\0'

'x' est codé dans un octet

"x" est codé dans deux octets

III) Initialisation de chaînes 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'est-à-dire : 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.

Exemples :

```
char TXT[] = "Hello";
```

TXT:	'H'	'e'	'l'	'l'	'o'	'\0'
------	-----	-----	-----	-----	-----	------



```
char TXT[6] = "Hello";
```

TXT:	'H'	'e'	'l'	'l'	'o'	'\0'
------	-----	-----	-----	-----	-----	------



```
char TXT[8] = "Hello";
```

TXT:	'H'	'e'	'l'	'l'	'o'	'\0'	0	0
------	-----	-----	-----	-----	-----	------	---	---



```
char TXT[5] = "Hello";
```

```
TXT: [ 'H' | 'e' | 'l' | 'l' | 'o' | ]
```

ERREUR pendant l'exécution



```
char TXT[4] = "Hello";
```

ERREUR pendant la compilation



- [Exercice 8.1](#)

IV) Accès aux éléments d'une chaîne :

L'accès à un élément d'une chaîne de caractères peut se faire de la même façon que l'accès à un élément d'un tableau. En déclarant une chaîne par :

```
char A[6];
```

nous avons défini un tableau A avec six éléments, auxquels on peut accéder par :

```
A[0], A[1], ... , A[5]
```

Exemple :

```
char A[6] = "Hello";
```

```
A: [ 'H' | 'e' | 'l' | 'l' | 'o' | '\0' ]  
   A[0] A[1] A[2] A[3] A[4] A[5]
```

V) Précédence alphabétique et lexicographique :

1) Précédence alphabétique des caractères :

La précédence des caractères dans l'alphabet d'une machine est dépendante du code de caractères utilisé. Pour le code ASCII, nous pouvons constater l'ordre suivant :

... ,0,1,2, ... ,9, ... ,A,B,C, ... ,Z, ... ,a,b,c, ... ,z, ...

Les symboles spéciaux (' , + , - , / , { , } , ...) et les lettres accentuées (é , è , à , û , ...) se trouvent répartis autour des trois grands groupes de caractères (chiffres, majuscules, minuscules). Leur précédence ne correspond à aucune règle d'ordre spécifique.

2) Relation de précédence :

De la précédence alphabétique des caractères, on peut déduire une *relation de précédence 'est inférieur à'* sur l'ensemble des caractères. Ainsi, on peut dire que

'0' est inférieur à 'Z'

et noter

'0' < 'Z'

car dans l'alphabet de la machine, le code du caractère '0' (ASCII : 48) est inférieur au code du caractère 'Z' (ASCII : 90).

3) Précédence lexicographique des chaînes de caractères :

En nous basant sur cette relation de *précédence alphabétique des caractères*, nous pouvons définir une *précédence lexicographique pour les chaînes de caractères*. Cette relation de précédence suit l'«ordre du dictionnaire» et est définie de façon récurrente :

- a) La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
- b) La chaîne A = "a₁a₂a ... a_p" (p caractères) précède lexicographiquement la chaîne B = "b₁b₂ ... b_m" (m caractères) si l'une des deux conditions suivantes est remplie :
 - 1) 'a₁' < 'b₁'
 - 2) 'a₁' = 'b₁' et "a₂a₃ ... a_p" précède lexicographiquement "b₂b₃ ... b_m"

Exemples :

"ABC" précède "BCD" car 'A' < 'B'
 "ABC" précède "B" car 'A' < 'B'
 "Abc" précède "abc" car 'A' < 'a'
 "ab" précède "abcd" car "" précède "cd"
 " ab" précède "ab" car ' ' < 'a'
 (le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97)

Remarque :

Malheureusement, il existe différents codes de caractères (p.ex. ASCII, EBCDIC, ISO) et l'ordre lexicographique est dépendant de la machine. Même la fonction **strcmp** qui indique la précedence lexicographique de deux chaînes de caractères (voir 8.6.2.) dépend du code de caractères utilisé.

4) Conversions et tests :

En tenant compte de l'ordre alphabétique des caractères, on peut contrôler le type du caractère (chiffre, majuscule, minuscule).

Exemples :

```
if (C>='0' && C<='9') printf("Chiffre\n", C);
if (C>='A' && C<='Z') printf("Majuscule\n", C);
if (C>='a' && C<='z') printf("Minuscule\n", C);
```

Il est facile, de convertir des lettres majuscules dans des minuscules :

```
if (C>='A' && C<='Z') C = C - 'A' + 'a';
```

ou vice-versa :

```
if (C>='a' && C<='z') C = C - 'a' + 'A';
```

Remarque :

Le code EBCDIC est organisé en zones, de façon que les caractères des trois grands groupes ne sont pas codés consécutivement. (P.ex. : les codes des caractères 'i' et 'j' diffèrent de 8 unités). Les méthodes de conversion discutées ci-dessus ne fonctionnent donc pas correctement dans le code EBCDIC et un programme portable doit être écrit à l'aide des fonctions (**isalpha**, **islower**, **toupper**, ...) de la bibliothèque <ctype> qui sont indépendantes du code de caractères (voir 8.6.4.).

VI) Travailler avec des chaînes de caractères :

Les bibliothèques de fonctions de C contiennent une série de fonctions spéciales pour le traitement de chaînes de caractères. Sauf indication contraire, les fonctions décrites dans ce chapitre sont portables conformément au standard ANSI-C.

1) Les fonctions de <stdio.h> :

Comme nous l'avons déjà vu au chapitre 4, la bibliothèque <stdio> nous offre des fonctions qui effectuent l'entrée et la sortie des données. A côté des fonctions **printf** et **scanf** que nous connaissons déjà, nous y trouvons les deux fonctions **puts** et **gets**, spécialement conçues pour l'écriture et la lecture de chaînes de caractères.

a) - Affichage de chaînes de caractères

printf avec le spécificateur de format **%s** permet d'intégrer une chaîne de caractères dans une phrase.

En plus, le spécificateur **%s** permet l'indication de la largeur *minimale* du champ d'affichage. Dans ce champ, les données sont justifiées à droite. Si on indique une largeur minimale négative, la chaîne sera justifiée à gauche. Un nombre suivant un point indique la largeur *maximale* pour l'affichage.

Exemples :

```
char NOM[] = "hello, world";
printf(":%s:", NOM);           -> :hello, world:
printf(":%5s:", NOM);         -> :hello, world:
printf(":%15s:", NOM);        -> : hello, world:
printf(":%-15s:", NOM);       -> :hello, world :
printf(":%.5s:", NOM);        -> :hello:
```

puts est idéale pour écrire une chaîne constante ou le contenu d'une variable dans une ligne isolée.

Syntaxe : **puts(<Chaîne>)**

Effet : **puts** écrit la chaîne de caractères désignée par <Chaîne> sur *stdout* et provoque un retour à la ligne. En pratique, **puts(TXT);** est équivalent à **printf("%s\n",TXT);**

Exemples :

```
char TEXTE[] = "Voici une première ligne.";
puts(TEXTE);
puts("Voici une deuxième ligne.");
```

b) - Lecture de chaînes de caractères :

scanf avec le spécificateur **%s** permet de lire un mot isolé à l'intérieur d'une suite de données du même ou d'un autre type.

Effet : **scanf** avec le spécificateur **%s** lit un *mot* du fichier d'entrée standard *stdin* et le mémorise à l'adresse qui est associée à **%s**.

Exemple :

```
char LIEU[25];
int JOUR, MOIS, ANNEE;
printf("Entrez lieu et date de naissance : \n");
scanf("%s %d %d %d", LIEU, &JOUR, &MOIS, &ANNEE);
```

Remarques importantes

- La fonction **scanf** a besoin des *adresses de ses arguments* :
 - * Les noms des variables numériques (**int**, **char**, **long**, **float**, ...) doivent être marqués par le symbole **'&'** (voir chap 4.4.).
 - * *Comme le nom d'une chaîne de caractères est le représentant de l'adresse du premier caractère de la chaîne, il ne doit pas être précédé de l'opérateur adresse '&' !*
- La fonction **scanf** avec plusieurs arguments présuppose que l'utilisateur connaisse exactement le nombre et l'ordre des données à introduire! Ainsi, l'utilisation de **scanf** pour la lecture de chaînes de caractères est seulement conseillée si on est forcé de lire un nombre fixé de mots en une fois.

c) Lecture de lignes de caractères :

gets est idéal pour lire une ou plusieurs lignes de texte (p.ex. des phrases) terminées par un retour à la ligne.

Syntaxe : **gets(<Chaîne>)**

Effet : **gets** lit une *ligne* de caractères de *stdin* et la copie à l'adresse indiquée par <Chaîne>.

Le retour à la ligne final est remplacé par le symbole de fin de chaîne '\0'.

Exemple :

```
int MAXI = 1000;
char LIGNE[MAXI];
gets(LIGNE);
```

-
- [Exercice 8.2](#)
 - [Exercice 8.3](#)
 - [Exercice 8.4](#)
-

2) Les fonctions de <string> :

La bibliothèque <string> fournit une multitude de fonctions pratiques pour le traitement de chaînes de caractères. Voici une brève description des fonctions les plus fréquemment utilisées.

Dans le tableau suivant, <n> représente un nombre du type **int**. Les symboles <s> et <t> peuvent être remplacés par :

- * une chaîne de caractères constante
- * le nom d'une variable déclarée comme tableau de **char**
- * un pointeur sur **char** (voir chapitre 9)

Fonctions pour le traitement de chaînes de caractères :

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

Remarques

- Comme le nom d'une chaîne de caractères représente une adresse fixe en mémoire, on ne peut pas 'affecter' une autre chaîne au nom d'un tableau :



~~A = "Hello";~~

Il faut bien copier la chaîne caractère par caractère ou utiliser la fonction **strcpy** respectivement **strncpy** :

strcpy(A, "Hello");

- La concaténation de chaînes de caractères en C ne se fait pas par le symbole '+' comme en langage algorithmique ou en Pascal. Il faut ou bien copier la deuxième chaîne caractère par caractère ou bien utiliser la fonction **strcat** ou **strncat**.

- La fonction **strcmp** est dépendante du code de caractères et peut fournir différents résultats sur différentes machines (voir § 8.5.).

-
- [Exercice 8.5](#)
 - [Exercice 8.6](#)
 - [Exercice 8.7](#)
 - [Exercice 8.8](#)

3) Les fonctions de <stdlib> :

La bibliothèque <stdlib> contient des déclarations de fonctions pour la conversion de nombres en chaînes de caractères et vice-versa.

a) Chaîne --> Nombre :

Les trois fonctions définies ci-dessous correspondent au standard ANSI-C et sont portables. Le symbole <s> peut être remplacé par :

- une chaîne de caractères constante
- le nom d'une variable déclarée comme tableau de **char**
- un pointeur sur **char** (voir chapitre 9)

Conversion de chaînes de caractères en nombres :

atoi(<s>) retourne la valeur numérique représentée par <s> comme **int**
atol(<s>) retourne la valeur numérique représentée par <s> comme **long**
atof(<s>) retourne la valeur numérique représentée par <s> comme **double** (!)

Règles générales pour la conversion :

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

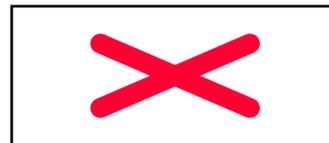
• [Exercice 8.9](#)

b) Nombre --> Chaîne :

Le standard ANSI-C ne contient pas de fonctions pour convertir des nombres en chaînes de caractères. Si on se limite aux systèmes fonctionnant sous DOS, on peut quand même utiliser les fonctions **itoa**, **ltoa** et **ultoa** qui convertissent des entiers en chaînes de caractères.

Conversion de nombres en chaînes de caractères :

itoa (<n_int>, <s>,)
ltoa (<n_long>, <s>,)
ultoa (<n_uns_long>, <s>,)



Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à <s>. La conversion se fait dans la base .

<n_int> est un nombre du type **int**
<n_long> est un nombre du type **long**
<n_uns_long> est un nombre du type **unsigned long**
<s> est une chaîne de caractères
Longueur maximale de la chaîne : 17 resp. 33 byte
 est la base pour la conversion (2 ... 36)

Remarque avancée :

En ANSI-C il existe la possibilité d'employer la fonction **sprintf** pour copier des données formatées *dans une variable* de la même façon que **printf** les imprime à l'écran.

Syntaxe :

```
sprintf( <chaîne cible>, <chaîne de formatage>, <expr1>,  
<expr2>, . . . )
```

- [Exercice 8.10](#)

4) Les fonctions de <ctype> :

Les fonctions de <ctype> servent à classifier et à convertir des caractères. Les symboles nationaux (é, è, ä, ü, ß, ç, ...) ne sont pas considérés. Les fonctions de <ctype> sont indépendantes du code de caractères de la machine et favorisent la portabilité des programmes. Dans la suite, <c> représente une valeur du type **int** qui peut être représentée comme caractère.

a) Fonctions de classification et de conversion :

Les fonctions de *classification* suivantes fournissent un résultat du type **int** différent de zéro, si la condition respective est remplie, sinon zéro.

La fonction : retourne une valeur différente de zéro.
isupper(<c>) si <c> est une majuscule ('A'...'Z')
islower(<c>) si <c> est une minuscule ('a'...'z')
isdigit(<c>) si <c> est un chiffre décimal ('0'...'9')
isalpha(<c>) si **islower(<c>)** ou **isupper(<c>)**
isalnum(<c>) si **isalpha(<c>)** ou **isdigit(<c>)**
isxdigit(<c>) si <c> est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
isspace(<c>) si <c> est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de *conversion* suivantes fournissent une valeur du type **int** qui peut être représentée comme caractère; la valeur originale de <c> reste inchangée :

tolower(<c>) retourne <c> converti en minuscule si <c> est une majuscule
toupper(<c>) retourne <c> converti en majuscule si <c> est une minuscule

VII) Tableaux de chaînes de caractères :

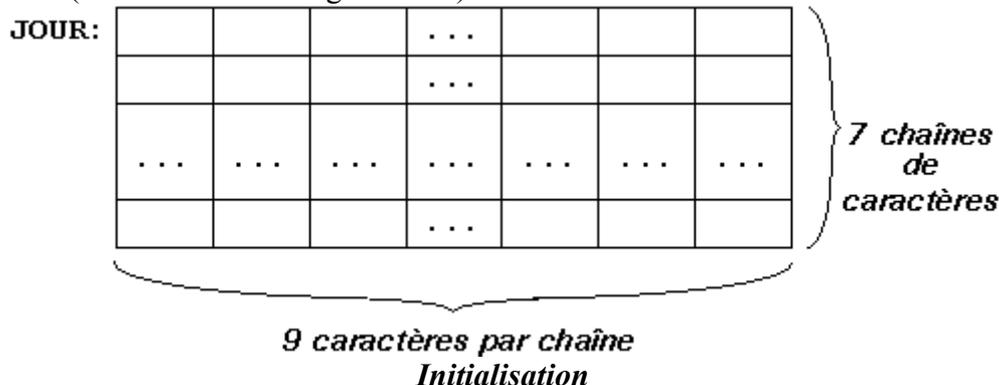
Souvent, il est nécessaire de mémoriser une suite de mots ou de phrases dans des variables. Il est alors pratique de créer un tableau de chaînes de caractères, ce qui allégera les déclarations des variables et simplifiera l'accès aux différents mots (ou phrases).

1) Déclaration, initialisation et mémorisation :

Un tableau de chaînes de caractères correspond à un tableau à deux dimensions du type **char**, où *chaque ligne contient une chaîne de caractères*.

a) Déclaration

La déclaration **char JOUR[7][9]** ; réserve l'espace en mémoire pour 7 mots contenant 9 caractères (dont 8 caractères significatifs).



Lors de la déclaration il est possible d'initialiser toutes les composantes du tableau par des chaînes de caractères constantes :

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

JOUR:

'l'	'u'	'n'	'd'	'i'	'\0'			
'm'	'a'	'r'	'd'	'i'	'\0'			
'm'	'e'	'r'	'c'	'r'	'e'	'd'	'i'	'\0'
...
'd'	'i'	'm'	'a'	'n'	'c'	'h'	'e'	'\0'

Mémorisation

Les tableaux de chaînes sont mémorisés ligne par ligne. La variable JOUR aura donc besoin de $7*9*1 = 63$ octets en mémoire.

2) Accès aux différentes composantes :

a) Accès aux chaînes

Il est possible d'accéder aux différentes *chaînes de caractères* d'un tableau, en indiquant simplement la ligne correspondante.

Exemple :

L'exécution des trois instructions suivantes :

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

```
int I = 2;
```

```
printf("Aujourd'hui, c'est %s !\n", JOUR[I]);
```

affichera la phrase :

```
Aujourd'hui, c'est mercredi !
```

Affectation :

Des expressions comme JOUR[I] représentent *l'adresse* du premier élément d'une chaîne de caractères. N'essayez donc pas de 'modifier' une telle adresse par une affectation directe !

~~JOUR[4] = "Friday";~~



L'attribution d'une chaîne de caractères à une composante d'un tableau de chaînes se fait en général à l'aide de la fonction strcpy :

Exemple :

La commande

```
strcpy(JOUR[4], "Friday");
```

changera le contenu de la 5^e composante du tableau JOUR de "vendredi" en "Friday".

Accès aux caractères :

Evidemment, il existe toujours la possibilité d'accéder directement aux différents *caractères* qui composent les mots du tableau.

Exemple :

L'instruction

```
for(I=0; I<7; I++)
```

```
printf("%c ", JOUR[I][0]);
```

va afficher les premières lettres des jours de la semaine :

```
l m m j v s d
```

- [Exercice 8.11](#)
 - [Exercice 8.12](#)
 - [Exercice 8.13](#)
 - [Exercice 8.14](#)
-

VIII) Exercices d'application :

Exercice 8.1. :

Lesquelles des chaînes suivantes sont initialisées correctement ? Corrigez les déclarations fausses et indiquez pour chaque chaîne de caractères le nombre d'octets qui sera réservé en mémoire.

- a) `char a[] = "un\ndeux\ntrois\n";`
 - b) `char b[12] = "un deux trois";`
 - c) `char c[] = 'abcdefg';`
 - d) `char d[10] = 'x';`
 - e) `char e[5] = "cinq";`
 - f) `char f[] = "Cette " "phrase" "est coupée";`
 - g) `char g[2] = {'a', '\0'};`
 - h) `char h[4] = {'a', 'b', 'c'};`
 - i) `char i[4] = "'o'";`
-

Exercice 8.2 :

Ecrire un programme qui lit 5 mots, séparés par des espaces et qui les affiche ensuite dans une ligne, mais dans l'ordre inverse. Les mots sont mémorisés dans 5 variables M1, ... ,M5.

Exemple :

```
voici une petite phrase !
! phrase petite une voici
```

Exercice 8.3 :

Ecrire un programme qui lit une ligne de texte (ne dépassant pas 200 caractères) la mémorise dans une variable TXT et affiche ensuite :

- a) la longueur L de la chaîne.
- b) le nombre de 'e' contenus dans le texte.
- c) toute la phrase à rebours, sans changer le contenu de la variable TXT.
- d) toute la phrase à rebours, après avoir inversé l'ordre des caractères dans TXT :

```
voici une petite phrase !
! esarhp etitep enu iciov
```

Exercice 8.4 :

Ecrire un programme qui lit un texte TXT (de moins de 200 caractères) et qui enlève toutes les apparitions du caractère 'e' en tassant les éléments restants. Les modifications se feront dans la même variable TXT.

Exemple :

```
Cette ligne contient quelques lettres e.
Ctt lign contint qulqus ltrrs .
```

Exercice 8.5 :

Ecrire un programme qui demande l'introduction du nom et du prénom de l'utilisateur et qui affiche alors la longueur totale du nom sans compter les espaces. Employer la fonction **strlen**.

Exemple :

Introduisez votre nom et votre prénom :
Mickey Mouse

Bonjour Mickey Mouse !
Votre nom est composé de 11 lettres.

Exercice 8.6 :

Ecrire un programme qui lit deux chaînes de caractères CH1 et CH2, les compare lexicographiquement et affiche le résultat :

Exemple :

Introduisez la première chaîne : ABC
Introduisez la deuxième chaîne : abc
"ABC" précède "abc"

Exercice 8.7 :

Ecrire un programme qui lit deux chaînes de caractères CH1 et CH2 et qui copie la première moitié de CH1 et la première moitié de CH2 dans une troisième chaîne CH3. Afficher le résultat.

- a) Utiliser les fonctions spéciales de <string>.
 - b) Utiliser uniquement les fonctions **gets** et **puts**.
-

Exercice 8.8 :

Ecrire un programme qui lit un verbe régulier en "er" au clavier et qui en affiche la conjugaison au présent de l'indicatif de ce verbe. Contrôlez s'il s'agit bien d'un verbe en "er" avant de conjuguer. Utiliser les fonctions **gets**, **puts**, **strcat** et **strlen**.

Exemple :

Verbe : fêter
je fête
tu fêtes
il fête
nous fêtons
vous fêtez
ils fêtent

Exercice 8.9 :

Soient les instructions :

```
char STR[200];  
puts("Entrez un nombre :");  
gets(STR);  
printf("Entrée = %s \n", STR);  
printf("integer = %d \n", atoi(STR));  
printf("long = %ld \n", atol(STR));  
printf("double = %f \n", atof(STR));
```

Quelles sont les valeurs affichées si on entre les chaînes de caractères suivantes :

- a) 123
- b) -123
- c) - 123
- d) 123.45
- e) 12E3
- f) 1234f5

- g) -1234567
h) 123e-02
i) -0,1234
-

Exercice 8.10 :

Ecrivez un petit programme utilisant la fonction **ltoa** qui sert à contrôler les résultats de l'exercice 3.3.

Exercice 8.11 :

Ecrire un programme qui lit 10 mots et les mémorise dans un tableau de chaînes de caractères. Trier les 10 mots lexicographiquement en utilisant les fonctions **strcmp** et **strcpy**. Afficher le tableau trié. Utilisez la méthode de tri par sélection directe (voir Exercice 7.14).

Exercice 8.12 :

Ecrire un programme qui lit un nombre entre 1 et 7 et qui affiche le nom du jour de la semaine correspondant :

"lundi"	pour 1
"mardi"	pour 2
...	...
"dimanche"	pour 7

Utiliser le premier élément du tableau pour mémoriser un petit message d'erreur.

Exercice 8.13 :

Ecrire un programme qui lit 5 mots, séparés par des espaces et qui les affiche ensuite dans une ligne, mais dans l'ordre inverse. Les mots sont mémorisés dans un tableau de chaînes de caractères.

Exemple :

```
voici une petite phrase !  
! phrase petite une voici
```

Exercice 8.14 :

Refaire l'exercice 8.8 (Conjugaison des verbes réguliers en "er") en utilisant deux tableaux de chaînes de caractères :

SUJ	pour les sujets
TERM	pour les terminaisons

Employez les fonctions **printf**, **scanf**, **strlen**.

Remarque :



Sauf indication contraire, les exercices suivants sont à résoudre *sans utiliser les fonctions spéciales des bibliothèques <string>, <stdlib> ou <ctype>*.

Ils servent à comprendre et à suivre le raisonnement de ces fonctions.

Exercice 8.15 :

Ecrire un programme qui lit deux chaînes de caractères, et qui indique leur précedence lexicographique dans le code de caractères de la machine (ici : code ASCII). Testez votre programme à l'aide des exemples du chapitre 8.5.

Exercice 8.16 :

Ecrire un programme qui lit une chaîne de caractères CH et qui convertit toutes les majuscules dans des minuscules et vice-versa.

Le résultat sera mémorisé dans la même variable CH et affiché après la conversion.

Exercice 8.17 :

Ecrire une procédure qui lit une chaîne de caractères et l'interprète comme un entier positif dans la base *décimale*. Pour la conversion, utiliser les fonctions de *<ctype>* et la précedence alphabétique des caractères de '0' à '9'. Mémoriser le résultat dans une variable du type **long**. La conversion s'arrête à la rencontre du premier caractère qui ne représente pas de chiffre décimal. Utiliser un indicateur logique OK qui précise si la chaîne représente correctement une valeur entière et positive.

Exercice 8.18 :

Ecrire une procédure qui lit une chaîne de caractères et l'interprète comme un entier positif dans la base *hexadécimale*. Pour la conversion, utiliser les fonctions de *<ctype>* et la précedence alphabétique des caractères. La conversion ignore les caractères qui ne représentent pas de chiffre hexadécimal et s'arrête à la fin de la chaîne de caractères. Le résultat sera mémorisé dans une variable du type **long** et affiché dans les bases hexadécimale **et** décimale.

Exercice 8.19 :

En se basant sur l'exercice 8.17, écrire un programme qui lit une chaîne de caractères et l'interprète comme un nombre rationnel positif ou négatif introduit en *notation décimale*. Mémoriser le résultat dans une variable du type **double**. Si le nombre a été introduit correctement, la valeur du résultat sera affichée, sinon le programme affichera un message d'erreur.

Méthode :

Utiliser une variable SIG pour mémoriser le signe de la valeur. Convertir tous les caractères numériques (avant et derrière le point décimal) en une valeur entière N. Compter les décimales (c'est-à-dire : les positions derrière le point décimal) à l'aide d'une variable DEC et calculer la valeur rationnelle comme suit :

$$N = N * SIG / \text{pow}(10, DEC)$$

Exemples :

-1234.234	-1234.23400
-123 45	Erreur!
123.23.	Erreur!
+00123.0123	123.012300

Exercice 8.20 :

En se basant sur l'exercice 8.19, écrire un programme qui lit une chaîne de caractères et l'interprète comme un nombre rationnel positif ou négatif introduit en *notation exponentielle*. Mémoriser le

résultat dans une variable du type **double**. Si le nombre a été introduit correctement, la valeur du résultat sera affichée, sinon le programme affichera un message d'erreur.

Méthode :

Utiliser une variable SIGE pour mémoriser le signe de l'exposant. Utiliser une variable EXP pour la valeur de l'exposant. Calculer la valeur de l'exposant à l'aide de SIGE, DEC et EXP. Calculer ensuite la valeur exacte de N à l'aide d'une formule analogue à celle de l'exercice ci-dessus.

Exemples :

-1234.234	-1234.234000
-1234. 234	Erreur!
123E+02	123400.000000
123E-02	1.230000
123.4e	123.400000
-12.1234e02	-1212.340000
123.4e3.4	Erreur!
12.12E1	121.200000
12.12 E1	Erreur!

Exercice 8.21 :

Ecrire un programme qui supprime la première occurrence d'une chaîne de caractères OBJ dans une chaîne de caractères SUJ.

Exemples :

PHON	ALPHONSE	ALSE
EI	PIERRE	PIERRE
T	TOTALEMENT	OTALEMENT
	HELLO	HELLO

Exercice 8.22 :

Ecrire un programme qui remplace la première occurrence d'une chaîne de caractères CH1 par la chaîne CH2 dans une chaîne de caractères SUJ. Utiliser une chaîne de sauvegarde FIN pendant le remplacement.

Exemples :

PHON	OY	ALPHONSE	ALOYSE
IE	EI	PIERRE	PEIRRE
IE	ARTE	PIERRE	PARTERRE
EI	IE	PIERRE	PIERRE
TOT	FIN	TOTALEMENT	FINALEMENT
	TTT	HELLO	HELLO

Exercice 8.23 :

Ecrire un programme qui remplace toutes les occurrences d'une chaîne de caractères CH1 par la chaîne CH2 dans une chaîne de caractères SUJ. Utiliser une chaîne de sauvegarde FIN pendant le remplacement.

Exemples :

PHON	OY	ALPHONSE	ALOYSE
AN	ONT	BANANE	BONTONTE
T	Y	TOTALEMENT	YOYALEMENY

L TTT HELLO HELLO
 HELLO HEO

IX) Solutions des exercices du Chapitre 8 : LES CHÂÎNES DE CARACTÈRES

Exercice 8.1 :

a) `char a[] = "un\ndeux\ntrois\n";`

Déclaration correcte

Espace : 15 octets

b) `char b[12] = "un deux trois";`

Déclaration incorrecte : la chaîne d'initialisation dépasse le bloc de mémoire réservé.

Correction : `char b[14] = "un deux trois";`

ou mieux : `char b[] = "un deux trois";`

Espace : 14 octets

c) `char c[] = 'abcdefg';`

Déclaration incorrecte : Les symboles " encadrent des caractères; pour initialiser avec une chaîne de caractères, il faut utiliser les guillemets (ou indiquer une liste de caractères).

Correction : `char c[] = "abcdefg";`

Espace : 8 octets

d) `char d[10] = 'x';`

Déclaration incorrecte : Il faut utiliser une liste de caractères ou une chaîne pour l'initialisation

Correction : `char d[10] = {'x', '\0'}`

ou mieux : `char d[10] = "x";`

Espace : 2 octets

e) `char e[5] = "cinq";`

Déclaration correcte

Espace : 5 octets

f) `char f[] = "Cette " "phrase" "est coupée";`

Déclaration correcte

Espace : 24 octets

g) `char g[2] = {'a', '\0'};`

Déclaration correcte

Espace : 2 octets

h) `char h[4] = {'a', 'b', 'c'};`

Déclaration incorrecte : Dans une liste de caractères, il faut aussi indiquer le symbole de fin de chaîne.

Correction : `char h[4] = {'a', 'b', 'c', '\0'};`

Espace : 4 octets

i) `char i[4] = "'o'";`

Déclaration correcte, mais d'une chaîne contenant les caractères '\', 'o', \' et '\0'.

Espace : 4 octets

Exercice 8.2 :

```
#include <stdio.h>
main()
{
    char M1[30], M2[30], M3[30], M4[30], M5[30];
    printf("Entrez 5 mots, séparés par des espaces :\n");
    scanf ("%s %s %s %s %s", M1, M2, M3, M4, M5);
    printf("%s %s %s %s %s\n", M5, M4, M3, M2, M1);
    return (0);
}
```

Exercice 8.3 :

```
#include <stdio.h>
main()
{
    /* Déclarations */
    char TXT[201]; /* chaîne donnée */
    int I,J; /* indices courants */
    int L; /* longueur de la chaîne */
    int C; /* compteur des lettres 'e' */
    int AIDE; /* pour l'échange des caractères */

    /* Saisie des données */
    printf("Entrez une ligne de texte (max.200 caractères)
:\n");
    gets(TXT); /* L'utilisation de scanf est impossible pour */
    /* lire une phrase contenant un nombre variable de mots. */

    /* a) Compter les caractères */
    /* La marque de fin de chaîne '\0' est */
    /* utilisée comme condition d'arrêt. */
    for (L=0; TXT[L]; L++)
        ;
    printf("Le texte est composé de %d caractères.\n",L);

    /* b) Compter les lettres 'e' dans le texte */
    C=0;
    for (I=0; TXT[I]; I++)
        if (TXT[I]=='e') C++;
    printf("Le texte contient %d lettres 'e'.\n",C);

    /* c) Afficher la phrase à l'envers */
    for (I=L-1; I>=0; I--)
        putchar(TXT[I]); /* ou printf("%c",TXT[I]); */
    putchar('\n'); /* ou printf("\n"); */

    /* d) Inverser l'ordre des caractères */
    for (I=0,J=L-1 ; I<J ; I++,J--)
    {
        AIDE=TXT[I];
        TXT[I]=TXT[J];
        TXT[J]=AIDE;
    }
    puts(TXT); /* ou printf("%s\n",TXT); */
    return (0);
}
```

Exercice 8.4 :

```
#include <stdio.h>
main()
{
    /* Déclarations */
```

```

char TXT[201]; /* chaîne donnée */
int I,J;      /* indices courants */

/* Saisie des données */
printf("Entrez une ligne de texte (max.200 caractères)
:\n");
gets(TXT);
/* Eliminer les lettres 'e' et comprimer : */
/* Copier les caractères de I vers J et incrémenter J */
/* seulement pour les caractères différents de 'e'. */
for (J=0,I=0 ; TXT[I] ; I++)
    {
        TXT[J] = TXT[I];
        if (TXT[I] != 'e') J++;
    }
/* Terminer la chaîne !! */
TXT[J]='\0';
/* Edition du résultat */
puts(TXT);
return (0);
}

```

Exercice 8.5 :

```

#include <stdio.h>
#include <string.h>
main()
{
    char NOM[40], PRENOM[40];
    printf("Introduisez votre nom et votre prénom : \n");
    scanf("%s %s", NOM, PRENOM);
    printf("\nBonjour %s %s !\n", NOM, PRENOM);
    printf("Votre nom est composé de %d lettres.\n",
           strlen(NOM) + strlen(PRENOM));

    /* ou bien
    printf("Votre nom est composé de %d lettres.\n",

strlen(strcat(NOM, PRENOM)) );
    */
    return (0);
}

```

Exercice 8.6 :

```

#include <stdlib.h>
#include <string.h>
main()
{
    /* Déclarations */
    char CH1[200], CH2[200]; /* chaînes entrées */
    int RES; /* résultat de la fonction strcmp */

    printf("Introduisez la première chaîne de caractères : ");
    gets(CH1);
}

```

```

printf("Introduisez la deuxième chaîne de caractères : ");
gets(CH2);

/* Comparaison et affichage du résultat */
RES = strcmp(CH1,CH2);
if (RES<0)
    printf("\"%s\" précède \"%s\"\\n",CH1 ,CH2);
else if (RES>0)
    printf("\"%s\" précède \"%s\"\\n",CH2 ,CH1);
else
    printf("\"%s\" est égal à \"%s\"\\n",CH1, CH2);
return (0);
}

```

Exercice 8.7 :

- a) Utiliser les fonctions spéciales de *<string>*.

```

#include <stdio.h>
#include <string.h>
main()
{
/* Déclarations */
char CH1[100], CH2[100]; /* chaînes données */
char CH3[100]="";      /* chaîne résultat */

/* Saisie des données */
printf("Introduisez la première chaîne de caractères : ");
gets(CH1);
printf("Introduisez la deuxième chaîne de caractères : ");
gets(CH2);

/* Traitements */
strncpy(CH3, CH1, strlen(CH1)/2);
strncat(CH3, CH2, strlen(CH2)/2);
/* Affichage du résultat */
printf("Un demi \"%s\" plus un demi \"%s\" donne \"%s\"\\n",
      CH1,          CH2,
CH3);
return (0);
}

```

- b) Utiliser uniquement les fonctions **gets** et **puts**.

```

#include <stdio.h>
main()
{
/* Déclarations */
char CH1[100], CH2[100]; /* chaînes données */
char CH3[100]="";      /* chaîne résultat */
int L1,L2; /* longueurs de CH1 et CH2 */
int I; /* indice courant dans CH1 et CH2 */
int J; /* indice courant dans CH3 */

/* Saisie des données */
puts("Introduisez la première chaîne de caractères : ");

```

```

gets(CH1);
puts("Introduisez la deuxième chaîne de caractères : ");
gets(CH2);

/* Détermination les longueurs de CH1 et CH2 */
for (L1=0; CH1[L1]; L1++) ;
for (L2=0; CH2[L2]; L2++) ;
/* Copier la première moitié de CH1 vers CH3 */
for (I=0 ; I<(L1/2) ; I++)
    CH3[I]=CH1[I];
/* Copier la première moitié de CH2 vers CH3 */
J=I;
for (I=0 ; I<(L2/2) ; I++)
    {
        CH3[J]=CH2[I];
        J++;
    }
/* Terminer la chaîne CH3 */
CH3[J]='\0';

/* Affichage du résultat */
puts("Chaîne résultat : ");
puts(CH3);
return (0);
}

```

Exercice 8.8 :

```

#include <stdio.h>
#include <string.h>
main()
{
    /* Déclarations */
    char VERB[20]; /* chaîne contenant le verbe */
    char AFFI[30]; /* chaîne pour l'affichage */
    int L;          /* longueur de la chaîne */

    /* Saisie des données */
    printf("Verbe : ");
    gets(VERB);

    /* Contrôler s'il s'agit d'un verbe en 'er' */
    L=strlen(VERB);
    if ((VERB[L-2]!='e') || (VERB[L-1]!='r'))
        puts("\aCe n'est pas un verbe du premier groupe!");
    else
        {
            /* Couper la terminaison 'er'. */
            VERB[L-2]='\0';
            /* Conjuguer ... */
            AFFI[0]='\0';
            strcat(AFFI, "je ");
            strcat(AFFI, VERB);
            strcat(AFFI, "e");
        }
}

```

```

    puts (AFFI) ;

    . . .

    AFFI[0]='\0';
    strcat(AFFI, "ils ");
    strcat(AFFI, VERB);
    strcat(AFFI, "ent");
    puts (AFFI) ;
}
return (0);
}

```

Exercice 8.9 :

	Entrée :	integer	long	double
a)	123	123	123	123.000000
b)	-123	-123	-123	-123.000000
c)	- 123	0	0	-0.000000
d)	123.45	123	123	123.45
e)	12E3	12	12	12000.000000
f)	1234f5	1234	1234	1234.000000
g)	-1234567	dépassement	-1234567	-1234567.000000
h)	123e-02	123	123	1.230000
i)	-0.1234	0	0	-0.123400

Exercice 8.10 :

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    long N;
    char STR[200];
    do
    {
        puts("Entrez un nombre :");
        scanf("%ld", &N);
        printf("Entrée   = %ld\n", N);
        printf("base 2   = %s\n", ltoa(N, STR, 2));
        printf("base 8   = %s\n", ltoa(N, STR, 8));
        printf("base 16  = %s\n", ltoa(N, STR, 16));
    }
    while(N);
    return (0);
}

```

Exercice 8.11 :

```

#include <stdio.h>
#include <string.h>
main()

```

```

{
/* Déclarations */
char MOT[10][50]; /* tableau de 10 mots à trier */
char AIDE[50]; /* chaîne d'aide pour la permutation */
int I; /* rang à partir duquel MOT n'est pas trié */
int J; /* indice courant */
int PMOT; /* indique la position du prochain mot */
/* dans la suite lexicographique. */

/* Saisie des données */
for (J=0; J<10; J++)
{
printf("Mot %d : ", J);
gets(MOT[J]); /* ou : scanf ("%s\n", MOT[J]); */
}

/* Tri du tableau par sélection directe du
/* prochain mot dans la suite lexicographique. */
for (I=0; I<9; I++)
{
/* Recherche du prochain mot à droite de A[I] */
PMOT=I;
for (J=I+1; J<10; J++)
if (strcmp(MOT[J], MOT[PMOT]) < 0)
PMOT=J;
/* Echange des mots à l'aide de strcpy */
strcpy(AIDE, MOT[I]);
strcpy(MOT[I], MOT[PMOT]);
strcpy(MOT[PMOT], AIDE);
}

/* Edition du résultat */
printf("Tableau trié lexicographiquement :\n");
for (J=0; J<10; J++)
puts(MOT[J]); /* ou : printf("%s\n",MOT[J]); */
printf("\n");
return (0);
}

```

Exercice 8.12 :

```

#include <stdio.h>
main()
{
/* Déclarations */
int N; /* nombre entré */
char JOUR[8][9] = {"\aErreur!", "lundi", "mardi",
"mercredi", "jeudi", "vendredi", "samedi", "dimanche"};
/* Saisie du nombre */
printf("Entrez un nombre entre 1 et 7 : ");
scanf("%d", &N);
/* Affichage du résultat - pour perfectionnistes */
if (N>0 && N<8)
printf("Le %de%c jour de la semaine est %s.\n",

```

```

N, (N==1)?'r':' ', JOUR[N]);
else
    puts(JOUR[0]);
return (0);
}

```

Exercice 8.13 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    char MOT[5][50]; /* tableau pour les 5 mots */
    int I;           /* indice courant */
    /* Saisie des mots */
    printf("Entrez 5 mots, séparés par des espaces :\n");
    /* Après le retour à la ligne, scanf lit */
    /* les 5 données à la fois. */
    for (I=0; I<5; I++)
        scanf("%s", MOT[I]);
    /* Affichage du résultat */
    for (I=4; I>=0; I--)
        printf("%s ", MOT[I]);
    printf("\n");
    return (0);
}

```

Exercice 8.14 :

```

#include <stdio.h>
#include <string.h>
main()
{
    /* Déclarations */
    /* Sujets et terminaisons */
    char SUJ[6][5] = {"je", "tu", "il", "nous", "vous", "ils"};
    char TERM[6][4] = {"e", "es", "e", "ons", "ez", "ent"};
    char VERB[20]; /* chaîne contenant le verbe */
    int L;        /* longueur de la chaîne */
    int I;        /* indice courant */

    /* Saisie des données */
    printf("Verbe : ");
    scanf("%s", VERB);

    /* Contrôler s'il s'agit d'un verbe en 'er' */
    L=strlen(VERB);
    if ((VERB[L-2] != 'e') || (VERB[L-1] != 'r'))
        printf("\n%s\n n'est pas un verbe du premier
groupe.\n", VERB);
    else
    {
        /* Couper la terminaison 'er'. */
        VERB[L-2]='\0';
    }
}

```

```

        /* Conjuguer ... */
        for (I=0; I<6; I++)
            printf("%s %s%s\n",SUJ[I], VERB, TERM[I]);
    }
    return (0);
}

```

Sauf indication contraire, les exercices suivants sont à résoudre *sans utiliser les fonctions spéciales des bibliothèques <string>, <stdlib> ou <ctype>*. Ils servent à comprendre et à suivre le raisonnement de ces fonctions.

Exercice 8.15 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    char CH1[50], CH2[50]; /* chaînes à comparer */
    int I;                 /* indice courant */

    /* Saisie des données */
    printf("Entrez la première chaîne à comparer : ");
    gets(CH1);
    printf("Entrez la deuxième chaîne à comparer : ");
    gets(CH2);

    /* Chercher la première position où */
    /* CH1 et CH2 se distinguent. */
    for (I=0; (CH1[I]==CH2[I]) && CH1[I] && CH2[I]; I++)
        ;
    /* Comparer le premier élément qui */
    /* distingue CH1 et CH2. */
    if (CH1[I]==CH2[I])
        printf("\'%s\' est égal à \''%s\'\\n", CH1, CH2);
    else if (CH1[I]<CH2[I])
        printf("\'%s\' précède \''%s\'\\n", CH1, CH2);
    else
        printf("\'%s\' précède \''%s\'\\n", CH2, CH1);
    return (0);
}

```

Exercice 8.16 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    char CH[100]; /* chaîne à convertir */
    int I;        /* indice courant */

    /* Saisie de la chaîne */
    printf("Entrez la chaîne à convertir : ");
    gets(CH);
    /* Conversion de la chaîne */
    for (I=0; CH[I]; I++)

```

```

    {
        if (CH[I]>='A' && CH[I]<='Z')
            CH[I] = CH[I] - 'A' + 'a';
        else if (CH[I]>='a' && CH[I]<='z')
            CH[I] = CH[I] - 'a' + 'A';
    }
    /* Affichage de la chaîne convertie */
    printf("Chaîne convertie : %s\n", CH);
    return (0);
}

```

Exercice 8.17 :

```

#include <stdio.h>
#include <ctype.h>
main()
{
    /* Déclarations */
    char CH[100]; /* chaîne numérique à convertir */
    long N; /* résultat numérique */
    int I; /* indice courant */
    int OK; /* indicateur logique précisant si la */
            /* chaîne a été convertie avec succès */

    /* Saisie de la chaîne */
    printf("Entrez un nombre entier et positif : ");
    gets(CH);
    /* Conversion de la chaîne */
    OK=1;
    N=0;
    for (I=0; OK && CH[I]; I++)
        if (isdigit(CH[I]))
            N = N*10 + (CH[I] - '0');
        else
            OK=0;

    /* Affichage de la chaîne convertie */
    if (OK)
        printf("Valeur numérique : %ld\n", N);
    else
        printf("\a\"%s\" ne représente pas correctement "
            "un entier et positif.\n", CH);
    return (0);
}

```

Exercice 8.18 :

```

#include <stdio.h>
#include <ctype.h>
main()
{
    /* Déclarations */
    char CH[100]; /* chaîne numérique à convertir */
    long N; /* résultat numérique */

```

```

int I; /* indice courant */
int OK; /* indicateur logique précisant si la */
        /* chaîne a été convertie avec succès */

/* Saisie de la chaîne */
printf("Entrez un nombre hexadécimal entier et positif : ");
gets(CH);
/* Conversion de la chaîne */
OK=1;
N=0;
for (I=0; OK && CH[I]; I++)
    if (isxdigit(CH[I]))
        {
            CH[I] = toupper(CH[I]);
            if (isdigit(CH[I]))
                N = N*16 + (CH[I]-'0');
            else
                N = N*16 + 10 + (CH[I]-'A');
        }
    else
        OK=0;

/* Affichage de la chaîne convertie */
if (OK)
    {
        printf("Valeur numérique hexadécimale : %lx\n", N);
        printf("Valeur numérique décimale      : %ld\n", N);
    }
else
    printf("\a\"%s\" n'est pas une valeur "
           "hexadécimale correcte.\n", CH);
return (0);
}

```

Exercice 8.19 :

```

#include <stdio.h>
#include <math.h>
#include <ctype.h>
main()
{
    /* Déclarations */
    char CH[100]; /* chaîne numérique à convertir */
    double N; /* résultat numérique */
    int I; /* indice courant */
    int SIG; /* signe de la valeur rationnelle */
    int DEC; /* nombre de décimales */
    int OK; /* indicateur logique précisant si la */
            /* chaîne a été convertie avec succès */

    /* Saisie de la chaîne */
    printf("Entrez un nombre rationnel : ");
    gets(CH);
}

```

```

/* Conversion de la chaîne : */
/* Initialisation des variables */
OK=1;
N=0.0;
I=0;
SIG=1;
/* Traitement du signe */
if (CH[I]=='-') SIG=-1;
if (CH[I]=='-' || CH[I]=='+')
    I++;
/* Positions devant le point décimal */
for ( ; isdigit(CH[I]); I++)
    N = N*10.0 + (CH[I]-'0');
/* Traitement du point décimal */
if (CH[I]=='.')
    I++;
else if (CH[I])
    OK=0;

/* Traitement et comptage des décimales */
for (DEC=0; isdigit(CH[I]); I++, DEC++)
    N = N*10.0 + (CH[I]-'0');
if (CH[I]) OK=0;
/* Calcul de la valeur à partir du signe et */
/* du nombre de décimales. */
N = SIG*N/pow(10,DEC);
/* Affichage de la chaîne convertie */
if (OK)
    printf("Valeur numérique : %f\n", N);
else
    printf("\a\"%s\" n'est pas une valeur "
           "rationnelle correcte.\n", CH);
return (0);
}

```

Exercice 8.20 :

```

#include <stdio.h>
#include <math.h>
#include <ctype.h>
main()
{ /* Déclarations */
char CH[100]; /* chaîne numérique à convertir */
double N; /* résultat numérique */
int I; /* indice courant */
int SIG; /* signe de la valeur rationnelle */
int DEC; /* nombre de décimales */
int SIGE; /* signe de l'exposant */
int EXP; /* valeur de l'exposant */
int OK; /* indicateur logique précisant si la */
/* chaîne a été convertie avec succès */

/* Saisie de la chaîne */
printf("Entrez un nombre rationnel : ");

```

```

gets(CH);

/* Conversion de la chaîne */
/* Initialisation des variables */
OK=1;
N=0.0;
I=0;
SIG=1;
SIGE=1;
/* Traitement du signe */
if (CH[I]=='-') SIG=-1;
if (CH[I]=='-' || CH[I]=='+') I++;
/* Positions devant le point décimal */
for (; isdigit(CH[I]); I++)
    N = N*10.0 + (CH[I]-'0');
/* Traitement du point décimal */
if (CH[I]=='.')
    I++;
/* Traitement et comptage des décimales */
for (DEC=0; isdigit(CH[I]); I++, DEC++)
    N = N*10.0 + (CH[I]-'0');
/* Traitement de la marque exponentielle */
if (CH[I]=='e' || CH[I]=='E')
    I++;
else if (CH[I])
    OK=0;
/* Traitement du signe de l'exposant */
if (CH[I]=='-') SIGE=-1;
if (CH[I]=='-' || CH[I]=='+') I++;
/* Traitement de la valeur de l'exposant */
for (EXP=0; isdigit(CH[I]); I++)
    EXP = EXP*10 + (CH[I]-'0');
if (CH[I]) OK=0;
/* Calcul de l'exposant à partir du signe */
/* SIGE, de la valeur de l'exposant EXP et */
/* du nombre de positions rationnelles DEC */
EXP = SIGE*EXP - DEC;
/* Calcul de la valeur à partir du signe et */
/* de l'exposant. */
N = SIG*N*pow(10,EXP);

/* Affichage de la chaîne convertie */
if (OK)
    printf("Valeur numérique : %f\n", N);
else
    printf("\a\"%s\" n'est pas une valeur "
          "rationnelle correcte.\n", CH);
return (0);
}

```

Exercice 8.21 :

```

#include <stdio.h>
main()

```

```

{
/* Déclarations */
char SUJ[100]; /* chaîne à transformer */
char OBJ[100]; /* chaîne à supprimer dans SUJ */
int I; /* indice courant dans SUJ */
int J; /* indice courant dans OBJ */
int TROUVE; /* indicateur logique qui précise */
/* si la chaîne OBJ a été trouvée */

/* Saisie des données */
printf("Introduisez la chaîne à supprimer : ");
gets(OBJ);
printf("Introduisez la chaîne à transformer : ");
gets(SUJ);
/* Recherche de OBJ dans SUJ */
TROUVE=0;
for (I=0; SUJ[I] && !TROUVE; I++)
/* Si la première lettre est identique, */
if (SUJ[I]==OBJ[0])
{
/* alors comparer le reste de la chaîne */
for (J=1; OBJ[J] && (OBJ[J]==SUJ[I+J]); J++)
;
if (OBJ[J]=='\0') TROUVE=1;
}
/* Si la position de départ de OBJ dans SUJ a été trouvée */
/* alors déplacer le reste de SUJ à cette position. */
if (TROUVE)
{
I--;
/* Maintenant I indique la position de OBJ */
/* dans SUJ et J indique la longueur de OBJ */
for (; SUJ[I+J]; I++)
SUJ[I]=SUJ[I+J];
SUJ[I]='\0';
}
/* Affichage du résultat */
printf("Chaîne résultat : \"%s\"\n", SUJ);
return (0);
}

```

Exercice 8.22 :

```

#include <stdio.h>
main()
{
/* Déclarations */
char SUJ[100]; /* chaîne à transformer */
char CH1[100]; /* chaîne à rechercher */
char CH2[100]; /* chaîne de remplacement */
char FIN[100]; /* chaîne de sauvegarde pour */
/* la fin de SUJ. */
int I; /* indice courant dans SUJ */
int J; /* indice courant dans CH1 et CH2 */

```

```

int K;          /* indice d'aide pour les copies */
int TROUVE;    /* indicateur logique qui précise */
               /* si la chaîne OBJ a été trouvée */

/* Saisie des données */
printf("Introduisez la chaîne à rechercher CH1 : ");
gets(CH1);
printf("Introduisez la chaîne à remplacer CH2 : ");
gets(CH2);
printf("Introduisez la chaîne à transformer SUJ : ");
gets(SUJ);

/* Recherche de CH1 dans SUJ */
TROUVE=0;
for (I=0; SUJ[I] && !TROUVE; I++)
    if (SUJ[I]==CH1[0])
        {
            for (J=1; CH1[J] && (CH1[J]==SUJ[I+J]); J++)
                ;
            if (CH1[J]=='\0') TROUVE=1;
        }

/* Si CH1 a été trouvée dans SUJ alors sauvegarder la fin */
/* de SUJ dans FIN, copier ensuite CH2 et FIN dans SUJ. */
if (TROUVE)
    {
        I--;
        /* Maintenant I indique la position de CH1 */
        /* dans SUJ et J indique la longueur de CH1 */
        /* Sauvegarder la fin de SUJ dans FIN */
        for (K=0; SUJ[K+I+J]; K++)
            FIN[K]=SUJ[K+I+J];
        FIN[K]='\0';
        /* Copier CH2 dans SUJ */
        for (K=0; CH2[K]; K++,I++)
            SUJ[I]=CH2[K];
        /* Recopier FIN dans SUJ */
        for (K=0; FIN[K]; K++,I++)
            SUJ[I]=FIN[K];
        /* Terminer la chaîne SUJ */
        SUJ[I]='\0';
    }

/* Affichage du résultat */
printf("Chaîne résultat : \"%s\"\n", SUJ);
return (0);
}

```

Exercice 8.23 :

```

#include <stdio.h>
main()
{

```

```

/* Déclarations */
char SUJ[100]; /* chaîne à transformer */
char CH1[100]; /* chaîne à rechercher */
char CH2[100]; /* chaîne de remplacement */
char FIN[100]; /* chaîne de sauvegarde pour */
                /* la fin de SUJ. */
int I;         /* indice courant dans SUJ */
int J;         /* indice courant dans CH1 et CH2 */
int K;         /* indice d'aide pour les copies */

/* Saisie des données */
printf("Introduisez la chaîne à rechercher CH1 : ");
gets(CH1);
printf("Introduisez la chaîne à remplacer CH2 : ");
gets(CH2);
printf("Introduisez la chaîne à transformer SUJ : ");
gets(SUJ);

/* Recherche de CH1 dans SUJ */
for (I=0; SUJ[I]; I++)
    if (SUJ[I]==CH1[0])
    {
        for (J=1; CH1[J] && (CH1[J]==SUJ[I+J]); J++)
            ;
        if (CH1[J]=='\0') /* TROUVE ! */
        {
            /* Maintenant I indique la position de CH1 */
            /* dans SUJ et J indique la longueur de CH1 */
            /* Sauvegarder la fin de SUJ dans FIN */
            for (K=0; SUJ[K+I+J]; K++)
                FIN[K]=SUJ[K+I+J];
            FIN[K]='\0';
            /* Copier CH2 dans SUJ et déplacer */
            /* I derrière la copie de CH2. */
            for (K=0; CH2[K]; K++,I++)
                SUJ[I]=CH2[K];
            /* Recopier FIN dans SUJ */
            for (K=0; FIN[K]; K++)
                SUJ[I+K]=FIN[K];
            /* Terminer la chaîne SUJ */
            SUJ[I+K]='\0';
            I--; /* réajustement de l'indice I */
        }
    }

/* Affichage du résultat */
printf("Chaîne résultat : \"%s\"\n", SUJ);
return (0);
}

```

Chapitre 9 : LES POINTEURS :

L'importance des pointeurs en C

La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de **pointeurs**, c'est-à-dire à l'aide de variables auxquelles on peut attribuer les **adresses d'autres variables**.

En C, les pointeurs jouent un rôle primordial dans la définition de fonctions : comme le passage des paramètres en C se fait toujours par la valeur, les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions. Ainsi le traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs (voir Chapitre 10).

En outre, les pointeurs nous permettent d'écrire des programmes plus compacts et plus efficaces et fournissent souvent la seule solution raisonnable à un problème. Ainsi, la majorité des applications écrites en C profitent excessivement des pointeurs.

Le revers de la médaille est très bien formulé par Kernighan & Ritchie dans leur livre 'Programming in C' :

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

Cette constatation a ensuite motivé les créateurs du standard ANSI-C à prescrire des règles explicites pour la manipulation des pointeurs.

I) Adressage de variables :

Avant de parler de pointeurs, il est indiqué de passer brièvement en revue les deux modes d'adressage principaux, qui vont d'ailleurs nous accompagner tout au long des chapitres suivants.

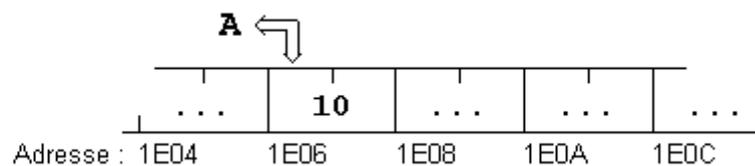
1) Adressage direct :

Dans la programmation, nous utilisons des variables pour stocker des informations. La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéder *directement* à cette valeur.

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

Exemple :

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



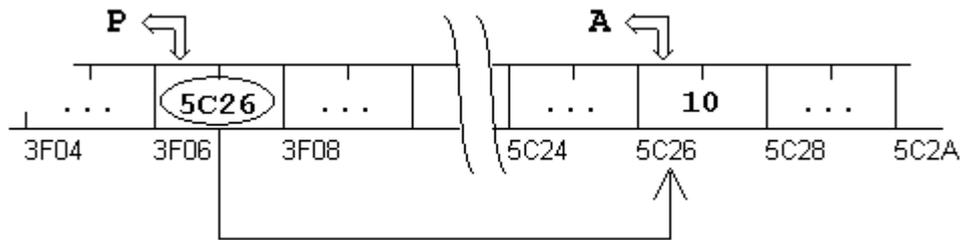
2) Adressage indirect :

Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable *A*, nous pouvons copier l'adresse de cette variable dans une variable spéciale *P*, appelée **pointeur**. Ensuite, nous pouvons retrouver l'information de la variable *A* en passant par le pointeur *P*.

Adressage indirect : Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

Exemple :

Soit *A* une variable contenant la valeur 10 et *P* un pointeur qui contient l'adresse de *A*. En mémoire, *A* et *P* peuvent se présenter comme suit :



II) Les pointeurs :

Définition : Pointeur

Un **pointeur** est une variable spéciale qui peut contenir l'**adresse** d'une autre variable.

En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type.

Si un pointeur P contient l'adresse d'une variable A, on dit que

'P pointe sur A'.

Remarque :

- X
- Les pointeurs et les noms de variables ont le même rôle : Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :
- * Un **pointeur** est une variable qui peut 'pointer' sur différentes adresses.
 - * Le **nom d'une variable** reste toujours lié à la même adresse.

1) Les opérateurs de base :

Lors du travail avec des pointeurs, nous avons besoin

- d'un opérateur 'adresse de' : **&** pour obtenir l'adresse d'une variable.
- d'un opérateur 'contenu de' : ***** pour accéder au contenu d'une adresse.
- d'une syntaxe de déclaration pour pouvoir déclarer un pointeur.

a) L'opérateur 'adresse de' : &

<NomVariable>

fournit l'adresse de la variable <NomVariable>

L'opérateur **&** nous est déjà familier par la fonction **scanf**, qui a besoin de l'adresse de ses arguments pour pouvoir leur attribuer de nouvelles valeurs.

Exemple :

```
int N;
printf("Entrez un nombre entier : ");
scanf("%d", &N);
```

Attention !

L'opérateur **&** peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c'est-à-dire à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

Représentation schématique :

Soit P un pointeur non initialisé

P : ●

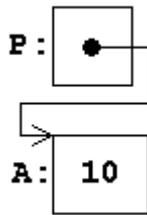
et A une variable (du même type) contenant la valeur 10 :

A : 10

Alors l'instruction

P = &A;

affecte l'adresse de la variable A à la variable P. En mémoire, A et P se présentent comme dans le graphique à la fin du chapitre 9.1.2. Dans notre représentation schématique, nous pouvons illustrer le fait que 'P pointe sur A' par une flèche :



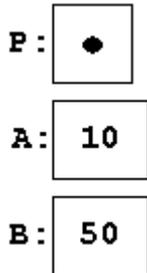
b) L'opérateur 'contenu de' : *

*<NomPointeur>

désigne le contenu de l'adresse référencée par le pointeur <NomPointeur>

Exemple :

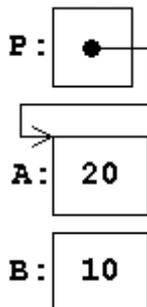
Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur non initialisé :



Après les instructions,

P = &A;
B = *P;
***P = 20;**

- P pointe sur A,
- le contenu de A (référéncé par *P) est affecté à B, et
- le contenu de A (référéncé par *P) est mis à 20.



c) Déclaration d'un pointeur :

<Type> *<NomPointeur>

déclare un pointeur <NomPointeur> qui peut recevoir des adresses de variables du type <Type>

Une déclaration comme

int *PNUM;

peut être interprétée comme suit :

*"*PNUM est du type int"*

ou

"PNUM est un pointeur sur **int**"

ou

"PNUM peut contenir l'adresse d'une variable du type **int**"

Exemple :

Le programme complet effectuant les transformations de l'exemple ci-dessus peut se présenter comme suit :

main()	ou bien	main()
{		{
/* déclarations */		/* déclarations */
short A = 10;		short A, B, *P;
short B = 50;		/* traitement */
short *P;		A = 10;
/* traitement */		B = 50;
P = &A;		P = &A;
B = *P;		B = *P;
*P = 20;		*P = 20;
return (0);		return (0);
}		}

Remarque :

Lors de la déclaration d'un pointeur en C, ce pointeur est lié explicitement à un type de données. Ainsi, la variable PNUM déclarée comme pointeur sur **int** ne peut pas recevoir l'adresse d'une variable d'un autre type que **int**.

Nous allons voir que la limitation d'un pointeur à un type de variables n'élimine pas seulement un grand nombre de sources d'erreurs très désagréables, mais permet une série d'opérations très pratiques sur les pointeurs (voir 9.3.2.).

2) Les opérations élémentaires sur pointeurs :

En travaillant avec des pointeurs, nous devons observer les règles suivantes :

a) Priorité de :

- * et &
- Les opérateurs * et & ont la même priorité que les autres opérateurs unaires (la négation !, l'incréméntation ++, la décréméntation --). Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.
- Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

Exemple :

Après l'instruction

P = &X;

les expressions suivantes, sont équivalentes :

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

Dans le dernier cas, les parenthèses sont nécessaires :

Comme les opérateurs unaires * et ++ sont évalués *de droite à gauche*, sans les parenthèses le *pointeur* P serait incrémenté, *non pas l'objet* sur lequel P pointe.

On peut uniquement affecter des adresses à un pointeur.



b) Le pointeur NUL :

Seule exception : La valeur numérique 0 (zéro) est utilisée pour indiquer qu'un pointeur ne pointe 'nulle part'.



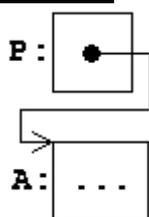
```
int *P;  
P = 0;
```

Finalement, les pointeurs sont aussi des variables et peuvent être utilisés comme telles. Soit P1 et P2 deux pointeurs sur **int**, alors l'affectation

```
P1 = P2;
```

copie le contenu de P2 vers P1. P1 pointe alors sur le même objet que P2.

c) Résumons :



Après les instructions :

```
int A;  
int *P;  
P = &A;
```

A désigne le contenu de A

&A désigne l'adresse de A

P désigne l'adresse de A

***P** désigne le contenu de A

En outre :

&P désigne l'adresse du pointeur P

***A** est illégal (puisque A n'est pas un pointeur)

-
- [Exercice 9.1](#)
-

III) Pointeurs et tableaux :

En C, il existe une relation très étroite entre tableaux et pointeurs. Ainsi, chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs. En général, les versions formulées avec des pointeurs sont plus compactes et plus efficaces, surtout à l'intérieur de fonctions. Mais, du moins pour des débutants, le 'formalisme pointeur' est un peu inhabituel.

1) Adressage des composantes d'un tableau :

Comme nous l'avons déjà constaté au chapitre 7, le nom d'un tableau représente l'adresse de son premier élément. En d'autres termes :

&tableau[0] et **tableau**

sont une seule et même adresse.

En simplifiant, nous pouvons retenir que le nom d'un tableau est un **pointeur constant** sur le premier élément du tableau.

Exemple :

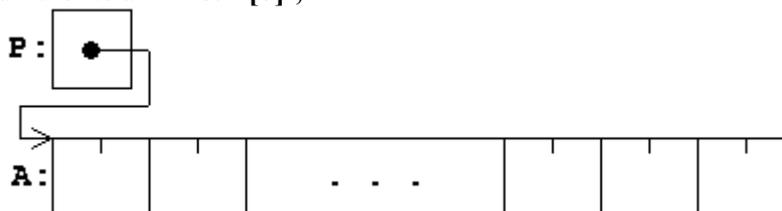
En déclarant un tableau A de type **int** et un pointeur P sur **int**,

```
int A[10];
```

```
int *P;
```

l'instruction :

P = A; est équivalente à **P = &A[0];**



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

P+i pointe sur la i-ième composante derrière P et

P-i pointe sur la i-ième composante devant P.

Ainsi, après l'instruction,

```
P = A;
```

le pointeur P pointe sur A[0], et

***(P+1)** désigne le contenu de A[1]

***(P+2)** désigne le contenu de A[2]

...

***(P+i)** désigne le contenu de A[i]

Remarques :

Au premier coup d'œil, il est bien surprenant que P+i n'adresse pas le i-ième *octet* derrière P, mais la i-ième *composante* derrière P ...

Ceci s'explique par la stratégie de programmation 'défensive' des créateurs du langage C :

Si on travaille avec des pointeurs, les erreurs les plus perfides sont causées par des pointeurs mal placés et des adresses mal calculées. En C, le compilateur peut calculer automatiquement l'adresse de l'élément P+i en ajoutant à P la grandeur d'une composante multipliée par i. Ceci est possible, parce que :

- chaque pointeur est limité à un seul type de données, et
- le compilateur connaît le nombre d'octets des différents types.

Exemple :

Soit A un tableau contenant des éléments du type **float** et P un pointeur sur **float** :

```
float A[20], X;
```

```
float *P;
```

Après les instructions,

```
P = A;
```

```
X = *(P+9);
```

X contient la valeur du 10-ième élément de A, (c'est-à-dire celle de A[9]). Une donnée du type **float** ayant besoin de 4 octets, le compilateur obtient l'adresse P+9 en ajoutant $9 * 4 = 36$ octets à l'adresse dans P.

Rassemblons les constatations ci dessus :

Comme A représente l'adresse de A[0],

***(A+1)** Désigne le contenu de A[1]

***(A+2)** Désigne le contenu de A[2]

...

***(A+i)** Désigne le contenu de A[i]

Attention !

Il existe toujours une différence essentielle entre un pointeur et le nom d'un tableau :



- Un *pointeur* est une variable, donc des opérations comme $P = A$ ou $P++$ sont permises.
- Le *nom d'un tableau* est une constante, donc des opérations comme $A = P$ ou $A++$ sont impossibles.

Ceci nous permet de jeter un petit coup d'œil derrière les rideaux :

Lors de la première phase de la compilation, toutes les expressions de la forme $A[i]$ sont traduites en $*(A+i)$. En multipliant l'indice i par la grandeur d'une composante, on obtient un indice en octets :

$$\langle \text{indice en octets} \rangle = \langle \text{indice élément} \rangle * \langle \text{grandeur élément} \rangle$$

Cet indice est ajouté à l'adresse du premier élément du tableau pour obtenir l'adresse de la composante i du tableau. Pour le calcul d'une adresse donnée par une adresse plus un indice en octets, on utilise un mode d'adressage spécial connu sous le nom '*adressage indexé*' :

$$\langle \text{adresse indexée} \rangle = \langle \text{adresse} \rangle + \langle \text{indice en octets} \rangle$$

Presque tous les processeurs disposent de plusieurs registres spéciaux (*registres index*) à l'aide desquels on peut effectuer l'adressage indexé de façon très efficace.

Résumons :

Soit un tableau A d'un type quelconque et i un indice pour les composantes de A , alors

A	désigne l'adresse de	$A[0]$
$A+i$	désigne l'adresse de	$A[i]$
$*(A+i)$	désigne le contenu de	$A[i]$

Si $P = A$, alors

P	pointe sur l'élément	$A[0]$
$P+i$	pointe sur l'élément	$A[i]$
$*(P+i)$	désigne le contenu de	$A[i]$

Formalisme tableau et formalisme pointeur :

A l'aide de ce bagage, il nous est facile de 'traduire' un programme écrit à l'aide du '*formalisme tableau*' dans un programme employant le '*formalisme pointeur*'.

Exemple :

Les deux programmes suivants copient les éléments positifs d'un tableau T dans un deuxième tableau POS .

Formalisme tableau :

```
main()
{
  int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
  int POS[10];
  int I,J; /* indices courants dans T et POS */
  for (J=0,I=0 ; I<10 ; I++)
    if (T[I]>0)
      {
        POS[J] = T[I];
        J++;
      }
  return (0);
}
```

Nous pouvons remplacer systématiquement la notation **tableau[I]** par ***(tableau + I)**, ce qui conduit à ce programme :

Formalisme pointeur :

```
main()
```

```

{
  int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
  int POS[10];
  int I,J; /* indices courants dans T et POS */
  for (J=0,I=0 ; I<10 ; I++)
    if (*(T+I)>0)
      {
        *(POS+J) = *(T+I);
        J++;
      }
  return (0);
}

```

Sources d'erreurs :

Un bon nombre d'erreurs lors de l'utilisation de C provient de la confusion entre soit contenu et adresse, soit pointeur et variable. Revoyons donc les trois types de déclarations que nous connaissons jusqu'ici et résumons les possibilités d'accès aux données qui se présentent.

Les *variables et leur utilisation* **int A** ; déclare une *variable simple* du type **int**

A désigne le contenu de A

&A désigne l'adresse de A

int B[] ; déclare un *tableau* d'éléments du type **int**

B désigne l'adresse de la première composante de B.
(Cette adresse est toujours constante)

B[i] désigne le contenu de la composante i du tableau

&B[i] désigne l'adresse de la composante i du tableau

en utilisant le formalisme pointeur :

B+i désigne l'adresse de la composante i du tableau

***(B+i)** désigne le contenu de la composante i du tableau

int *P ; déclare un *pointeur* sur des éléments du type **int**.

P peut pointer sur des variables simples du type **int** ou
sur les composantes d'un tableau du type **int**.

P désigne l'adresse contenue dans P
(Cette adresse est variable)

***P** désigne le contenu de l'adresse dans P

Si P pointe dans un tableau, alors

P désigne l'adresse de la première composante

P+i désigne l'adresse de la i-ème composante derrière P

***(P+i)** désigne le contenu de la i-ème composante derrière P

- [Exercice 9.2](#)

2) Arithmétique des pointeurs :

Comme les pointeurs jouent un rôle si important, le langage C soutient une série d'opérations arithmétiques sur les pointeurs que l'on ne rencontre en général que dans les langages machines. Le confort de ces opérations en C est basé sur le principe suivant :

Toutes les opérations avec les pointeurs tiennent compte automatiquement du type et de la grandeur des objets pointés.

a) - Affectation par un pointeur sur le même type :

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

P1 = P2;

fait pointer P1 sur le même objet que P2

b) - Addition et soustraction d'un nombre entier :

Si P pointe sur l'élément A[i] d'un tableau, alors

P+n pointe sur A[i+n]

P-n pointe sur A[i-n]

c) - Incrémentation et décrémentation d'un pointeur :

Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction

P++; P pointe sur A[i+1]

P+=n; P pointe sur A[i+n]

P--; P pointe sur A[i-1]

P-=n; P pointe sur A[i-n]

d) Domaine des opérations :

L'addition, la soustraction, l'incrément et la décrémentation sur les pointeurs sont seulement définies à l'intérieur d'un tableau. Si l'adresse formée par le pointeur et l'indice sort du domaine du tableau, alors le résultat n'est pas défini.

Seule exception : Il est permis de 'pointer' sur le premier octet derrière un tableau (à condition que cet octet se trouve dans le même segment de mémoire que le tableau). Cette règle, introduite avec le standard ANSI-C, légalise la définition de boucles qui incrémentent le pointeur *avant* l'évaluation de la condition d'arrêt.

Exemples :

```
int A[10];
```

```
int *P;
```

```
P = A+9; /* dernier élément -> légal */
```

```
P = A+10; /* dernier élément + 1 -> légal */
```

```
P = A+11; /* dernier élément + 2 -> illégal */
```

```
P = A-1; /* premier élément - 1 -> illégal */
```

e) - Soustraction de deux pointeurs :

Soient P1 et P2 deux pointeurs qui pointent *dans le même tableau :*

P1-P2 fournit le nombre de composantes comprises entre P1 et P2.

Le résultat de la soustraction **P1-P2** est

- négatif, si P1 précède P2

- zéro, si P1 = P2

- positif, si P2 précède P1

- indéfini, si P1 et P2 ne pointent pas dans le même tableau

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

f) - Comparaison de deux pointeurs :

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

La comparaison de deux pointeurs qui pointent *dans le même tableau* est équivalente à la comparaison des indices correspondants. (Si les pointeurs ne pointent pas dans le même tableau, alors le résultat est donné par leurs positions relatives dans la mémoire).

-
- [Exercice 9.3](#)
 - [Exercice 9.4](#)
 - [Exercice 9.5](#)
 - [Exercice 9.6](#)

3) Pointeurs et chaînes de caractères :

De la même façon qu'un pointeur sur **int** peut contenir l'adresse d'un nombre isolé ou d'une composante d'un tableau, un pointeur sur **char** peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères. Un pointeur sur **char** peut en plus contenir l'adresse d'une chaîne de caractères constante et il peut même être *initialisé* avec une telle adresse.

A la fin de ce chapitre, nous allons anticiper avec un exemple et montrer que les pointeurs sont les éléments indispensables mais effectifs des fonctions en C.

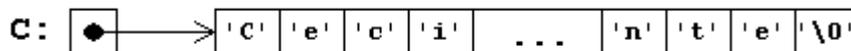
a) - Pointeurs sur char et chaînes de caractères constantes :

i) Affectation :

On peut attribuer l'adresse d'une chaîne de caractères constante à un pointeur sur **char** :

Exemple :

```
char *C;  
C = "Ceci est une chaîne de caractères constante";
```



Nous pouvons lire cette chaîne constante (par exemple : pour l'afficher), mais il n'est pas recommandé de la modifier, parce que le résultat d'un programme qui essaie de modifier une chaîne de caractères constante n'est pas prévisible en ANSI-C.



ii) Initialisation :

Un pointeur sur **char** peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante :

```
char *B = "Bonjour !";
```

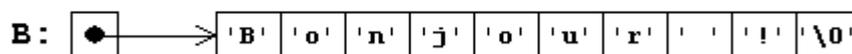
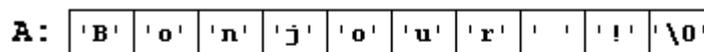
Attention !

Il existe une différence importante entre les deux déclarations :

```
char A[] = "Bonjour !"; /* un tableau */  
char *B = "Bonjour !"; /* un pointeur */
```

A est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '\0'. Les caractères de la chaîne peuvent être changés, mais le nom A va toujours pointer sur la même adresse en mémoire.

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.



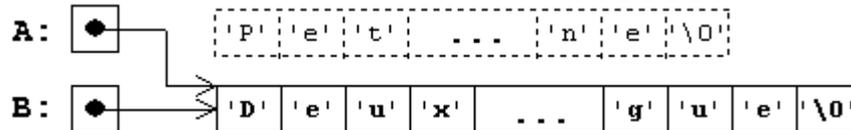
iii) Modification :

Si nous affectons une nouvelle valeur à un pointeur sur une chaîne de caractères constante, nous risquons de perdre la chaîne constante. D'autre part, un pointeur sur **char** a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur :

Exemple :

```
char *A = "Petite chaîne";  
char *B = "Deuxième chaîne un peu plus longue";  
A = B;
```

Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue :



Attention !

Les affectations discutées ci-dessus ne peuvent pas être effectuées avec des tableaux de caractères :

Exemple :



```
char A[45] = "Petite chaîne";
char B[45] = "Deuxième chaîne un peu plus longue";
char C[30];
A = B;          /* IMPOSSIBLE -> ERREUR !!! */
C = "Bonjour !"; /* IMPOSSIBLE -> ERREUR !!! */
```

```
A: ['P' 'e' 't' ... 'n' 'e' '\0']
```

```
B: ['D' 'e' 'u' 'x' ... 'g' 'u' 'e' '\0']
```

Dans cet exemple, nous essayons de copier l'adresse de B dans A, respectivement l'adresse de la chaîne constante dans C. Ces opérations sont impossibles et illégales parce que *l'adresse représentée par le nom d'un tableau reste toujours constante*.

Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre (par exemple dans une boucle) ou déléguer cette charge à une fonction de `<stdio>` ou `<string>`.

Conclusions :

- Utilisons des *tableaux de caractères* pour déclarer les chaînes de caractères que nous voulons modifier.
- Utilisons des *pointeurs sur char* pour manipuler des chaînes de caractères constantes (dont le contenu ne change pas).
- Utilisons de préférence des *pointeurs* pour effectuer les manipulations à l'intérieur des tableaux de caractères. (voir aussi les remarques ci-dessous).

Perspectives et motivation :

Avantages des pointeurs sur char

Comme la fin des chaînes de caractères est marquée par un symbole spécial, nous n'avons pas besoin de connaître la longueur des chaînes de caractères; nous pouvons même laisser de côté les indices d'aide et parcourir les chaînes à l'aide de pointeurs.

Cette façon de procéder est indispensable pour traiter de chaînes de caractères dans des fonctions. En anticipant sur la matière du chapitre 10, nous pouvons ouvrir une petite parenthèse pour illustrer les avantages des pointeurs dans la définition de fonctions traitant des chaînes de caractères :

Pour fournir un tableau comme paramètre à une fonction, il faut passer *l'adresse du tableau* à la fonction. Or, les *paramètres des fonctions sont des variables locales*, que nous pouvons utiliser comme variables d'aide. Bref, une fonction obtenant une chaîne de caractères comme paramètre, dispose d'une *copie locale de l'adresse de la chaîne*. Cette copie peut remplacer les indices ou les variables d'aide du formalisme tableau.

Discussion d'un exemple :

Reprenons l'exemple de la fonction `strcpy`, qui copie la chaîne CH2 vers CH1. Les deux chaînes sont les arguments de la fonction et elles sont déclarées comme *pointeurs sur char*. La première version de `strcpy` est écrite entièrement à l'aide du formalisme tableau :

```
void strcpy(char *CH1, char *CH2)
{
```

```

int I;
I=0;
while ((CH1[I]=CH2[I]) != '\0')
    I++;
}

```

Dans une première approche, nous pourrions remplacer simplement la notation **tableau[I]** par ***(tableau + I)**, ce qui conduirait au programme :

```

void strcpy(char *CH1, char *CH2)
{
    int I;
    I=0;
    while ((*CH1+I)=*(CH2+I)) != '\0')
        I++;
}

```

Cette transformation ne nous avance guère, nous avons tout au plus gagné quelques millièmes de secondes lors de la compilation. Un 'véritable' avantage se laisse gagner en calculant directement avec les pointeurs CH1 et CH2 :

```

void strcpy(char *CH1, char *CH2)
{
    while ((*CH1=*CH2) != '\0')
    {
        CH1++;
        CH2++;
    }
}

```

Comme nous l'avons déjà constaté dans l'introduction de ce manuel, un vrai professionnel en C escaladerait les 'simplifications' jusqu'à obtenir :

```

void strcpy(char *CH1, char *CH2)
{
    while (*CH1++ = *CH2++)
        ;
}

```

Assez 'optimisé' - fermons la parenthèse et familiarisons-nous avec les notations et les manipulations du 'formalisme pointeur' ...

- [Exercice 9.7](#)
- [Exercice 9.8](#)
- [Exercice 9.9](#)
- [Exercice 9.10](#)
- [Exercice 9.11](#)
- [Exercice 9.12](#)
- [Exercice 9.13](#)
- [Exercice 9.14](#)

4) Pointeurs et tableaux à deux dimensions :

L'arithmétique des pointeurs se laisse élargir avec *toutes* ses conséquences sur les tableaux à deux dimensions. Voyons cela sur un exemple :

Exemple :

Le tableau M à deux dimensions est défini comme suit :

```

int M[4][10] = { { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
                 { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 },

```

```
{20,21,22,23,24,25,26,27,28,29},
{30,31,32,33,34,35,36,37,38,39}};
```

Le nom du tableau M représente l'adresse du premier élément du tableau et pointe (oh, surprise...) sur le *tableau* M[0] qui a la valeur :

```
{0,1,2,3,4,5,6,7,8,9}.
```

L'expression (M+1) est l'adresse du deuxième élément du tableau et pointe sur M[1] qui a la valeur :

```
{10,11,12,13,14,15,16,17,18,19}.
```

Explication :

Au sens strict du terme, un tableau à deux dimensions est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, le premier élément de la matrice M est le *vecteur* {0,1,2,3,4,5,6,7,8,9}, le deuxième élément est {10,11,12,13,14,15,16,17,18,19} et ainsi de suite.

L'arithmétique des pointeurs qui respecte automatiquement les dimensions des éléments conclut logiquement que :

M+I désigne l'adresse du tableau M[I]

Problème :

Comment pouvons-nous accéder à l'aide de pointeurs aux éléments de chaque composante du tableau, c.à-d. : aux éléments M[0][0], M[0][1], ... , M[3][9] ?

Discussion :

Une solution consiste à convertir la valeur de M (qui est un pointeur sur *un tableau du type int*) en un pointeur sur *int*. On pourrait se contenter de procéder ainsi :

```
 int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
                 {10,11,12,13,14,15,16,17,18,19},
                 {20,21,22,23,24,25,26,27,28,29},
                 {30,31,32,33,34,35,36,37,38,39}};

int *P;
P = M; /* conversion automatique */
```

Cette dernière affectation entraîne une conversion automatique de l'adresse &M[0] dans l'adresse &M[0][0]. (Remarquez bien que l'adresse transmise reste la même, seule la nature du pointeur a changé).

Cette solution n'est pas satisfaisante à cent pour-cent : Généralement, on gagne en lisibilité en explicitant la conversion mise en œuvre par l'opérateur de conversion forcée ("cast"), qui évite en plus des messages d'avertissement de la part du compilateur.

Solution :

Voici finalement la version que nous utiliserons :

```
 int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
                 {10,11,12,13,14,15,16,17,18,19},
                 {20,21,22,23,24,25,26,27,28,29},
                 {30,31,32,33,34,35,36,37,38,39}};

int *P;
P = (int *)M; /* conversion forcée */
```

Dû à la mémorisation ligne par ligne des tableaux à deux dimensions, il nous est maintenant possible traiter M à l'aide du pointeur P comme un tableau unidimensionnel de dimension 4*10.

Exemple :

Les instructions suivantes calculent la somme de tous les éléments du tableau M :

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
```

```

                                {30,31,32,33,34,35,36,37,38,39}};
int *P;
int I, SOM;
P = (int*)M;
SOM = 0;
for (I=0; I<40; I++)
    SOM += *(P+I);

```

Attention !



Lors de l'interprétation d'un tableau à deux dimensions comme tableau unidimensionnel *il faut calculer avec le nombre de colonnes indiqué dans la déclaration* du tableau.

Exemple :

Pour la matrice A, nous réservons de la mémoire pour 3 lignes et 4 colonnes, mais nous utilisons seulement 2 lignes et 2 colonnes :

```

int A[3][4];
A[0][0]=1;
A[0][1]=2;
A[1][0]=10;
A[1][1]=20;

```

Dans la mémoire, ces composantes sont stockées comme suit :



L'adresse de l'élément A[I][J] se calcule alors par :

$$A + I*4 + J$$

Conclusion :

Pour pouvoir travailler à l'aide de pointeurs dans un tableau à deux dimensions, nous avons besoin de quatre données :

- a) l'adresse du premier élément du tableau converti dans le type simple des éléments du tableau
- b) la longueur d'une ligne réservée en mémoire (- voir déclaration - ici : 4 colonnes)
- c) le nombre d'éléments effectivement utilisés dans une ligne (- p.ex : lu au clavier - ici : 2 colonnes)
- d) le nombre de lignes effectivement utilisées (- p.ex : lu au clavier - ici : 2 lignes)

- [Exercice 9.15](#)
- [Exercice 9.16](#)
- [Exercice 9.17](#)

IV) Tableaux de pointeurs :

Note au lecteur :

Si la notion de pointeurs vous était nouvelle jusqu'ici, alors sautez les sections 9.4 et 9.5 de ce chapitre et les exercices correspondants. Traitez d'abord le chapitre suivant jusqu'à ce que les notions de pointeurs et d'adresses se soient bien consolidées.

Si nous avons besoin d'un ensemble de pointeurs du même type, nous pouvons les réunir dans un tableau de pointeurs.

1) Déclaration :

Déclaration d'un tableau de pointeurs :

`<Type> *<NomTableau> [<N>]`
déclare un tableau `<NomTableau>` de `<N>` pointeurs sur des données du type `<Type>`.

Exemple :

```
double *A[10];
```

déclare un tableau de 10 pointeurs sur des rationnels du type **double** dont les adresses et les valeurs ne sont pas encore définies.

Remarque :

Le plus souvent, les tableaux de pointeurs sont utilisés pour mémoriser de façon économique des chaînes de caractères de différentes longueurs. Dans la suite, nous allons surtout considérer les tableaux de pointeurs sur des chaînes de caractères.

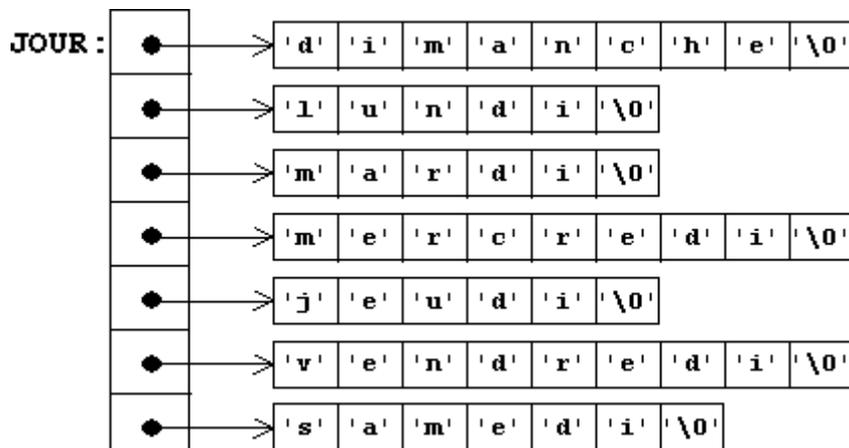
2) Initialisation :

Nous pouvons initialiser les pointeurs d'un tableau sur **char** par les adresses de chaînes de caractères constantes.

Exemple :

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

déclare un tableau **JOUR[]** de 7 pointeurs sur **char**. Chacun des pointeurs est initialisé avec l'adresse de l'une des 7 chaînes de caractères.



On peut afficher les 7 chaînes de caractères en fournissant les adresses contenues dans le tableau **JOUR** à **printf** (ou **puts**) :

```
int I;  
for (I=0; I<7; I++) printf("%s\n", JOUR[I]);
```

Comme **JOUR[I]** est un pointeur sur **char**, on peut afficher les premières lettres des jours de la semaine en utilisant l'opérateur 'contenu de' :

```
int I;  
for (I=0; I<7; I++) printf("%c\n", *JOUR[I]);
```

L'expression **JOUR[I]+J** désigne la **J**-ième lettre de la **I**-ième chaîne. On peut afficher la troisième lettre de chaque jour de la semaine par :

```
int I;  
for (I=0; I<7; I++) printf("%c\n", *(JOUR[I]+2));
```

Résumons :

Les tableaux de pointeurs `int *D[]` ; déclare un tableau de pointeurs sur des éléments du type `int`

D[i] peut pointer sur des variables simples ou sur les composantes d'un tableau.

D[i] désigne l'adresse contenue dans l'élément `i` de `D`
(Les adresses dans `D[i]` sont variables)

***D[i]** désigne le contenu de l'adresse dans `D[i]`

Si `D[i]` pointe dans un tableau,

D[i]	désigne l'adresse de la première composante
D[i]+j	désigne l'adresse de la j-ième composante
*(D[i]+j)	désigne le contenu de la j-ième composante

- [Exercice 9.18](#)
- [Exercice 9.19](#)

V) Allocation dynamique de mémoire :

Nous avons vu que l'utilisation de pointeurs nous permet de mémoriser économiquement des données de différentes grandeurs. Si nous générons ces données pendant l'exécution du programme, il nous faut des moyens pour réserver et libérer de la mémoire au fur et à mesure que nous en avons besoin. Nous parlons alors de *l'allocation dynamique* de la mémoire.

Revoyons d'abord de quelle façon la mémoire a été réservée dans les programmes que nous avons écrits jusqu'ici.

1) Déclaration statique de données :

Chaque variable dans un programme a besoin d'un certain nombre d'octets en mémoire. Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des déclarations des données. Dans tous ces cas, le nombre d'octets à réserver était déjà connu pendant la compilation. Nous parlons alors de la *déclaration statique* des variables.

Exemples :

```
float A, B, C;           /* réservation de 12 octets */
short D[10][20];       /* réservation de 400 octets */
char E[] = {"Bonjour !"};
                        /* réservation de 10 octets */
char F[][10] = {"un", "deux", "trois", "quatre"};
                        /* réservation de 40 octets */
```

a) Pointeurs :

Le nombre d'octets à réserver pour un *pointeur* dépend de la machine et du 'modèle' de mémoire choisi, mais il est déjà connu lors de la compilation. Un pointeur est donc aussi déclaré statiquement. Supposons dans la suite qu'un pointeur ait besoin de `p` octets en mémoire. (En DOS : `p=2` ou `p=4`)

Exemples :

```
double *G;             /* réservation de p octets */
char *H;               /* réservation de p octets */
float *I[10];          /* réservation de 10*p octets */
```

b) Chaînes de caractères constantes

L'espace pour les *chaînes de caractères constantes* qui sont affectées à des pointeurs ou utilisées pour initialiser des pointeurs sur **char** est aussi réservé automatiquement :

Exemples :

```
char *J = "Bonjour !";
        /* réservation de p+10 octets */
float *K[] = {"un", "deux", "trois", "quatre"};
        /* réservation de 4*p+3+5+6+7 octets */
```

2) Allocation dynamique :

Problème :

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation. Ce serait alors un gaspillage de réserver toujours l'espace maximal prévisible. Il nous faut donc un moyen de gérer la mémoire lors de l'exécution du programme.

Exemple :

Nous voulons lire 10 phrases au clavier et mémoriser les phrases en utilisant un tableau de pointeurs sur **char**. Nous déclarons ce tableau de pointeurs par :

```
char *TEXTE[10];
```

Pour les 10 pointeurs, nous avons besoin de 10*p octets. Ce nombre est connu dès le départ et les octets sont réservés automatiquement. Il nous est cependant impossible de prévoir à l'avance le nombre d'octets à réserver pour les phrases elles-mêmes qui seront introduites lors de l'exécution du programme ...

Allocation dynamique :

La réservation de la mémoire pour les 10 phrases peut donc seulement se faire *pendant l'exécution du programme*. Nous parlons dans ce cas de *allocation dynamique* de la mémoire.

3) La fonction malloc et l'opérateur sizeof :

La fonction **malloc** de la bibliothèque `<stdlib>` nous aide à localiser et à réserver de la mémoire au cours d'un programme. Elle nous donne accès au tas (*heap*); c'est-à-dire à l'espace en mémoire laissé libre une fois mis en place le DOS, les gestionnaires, les programmes résidents, le programme lui-même et la pile (*stack*).

a) La fonction malloc :

```
malloc( <N> )
fournit l'adresse d'un bloc en mémoire de <N> octets libres
ou la valeur zéro s'il n'y a pas assez de mémoire.
```

Attention !

Sur notre système, le paramètre <N> est du type **unsigned int**. A l'aide de **malloc**, nous ne pouvons donc pas réserver plus de 65535 octets à la fois!

Exemple :

Supposons que nous ayons besoin d'un bloc en mémoire pour un texte de 4000 caractères. Nous disposons d'un pointeur T sur **char** (**char *T**). Alors l'instruction :

```
T = malloc(4000);
```

fournit l'adresse d'un bloc de 4000 octets libres et l'affecte à T. S'il n'y a plus assez de mémoire, T obtient la valeur zéro.

Si nous voulons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine à l'autre, nous avons besoin de la grandeur effective d'une donnée de ce type. L'opérateur **sizeof** nous aide alors à préserver la portabilité du programme.

b) L'opérateur unaire sizeof

```
sizeof <var>
```

fournit la grandeur de la variable <var>
sizeof <const>
 fournit la grandeur de la constante <const>
sizeof (<type>)
 fournit la grandeur pour un objet du type <type>

Exemple :

Après la déclaration,

```
short A[10];
char B[5][10];
```

nous obtenons les résultats suivants sur un IBM-PC (ou compatible) :

sizeof A	s'évalue à 20
sizeof B	s'évalue à 50
sizeof 4.25	s'évalue à 8
sizeof "Bonjour !"	s'évalue à 10
sizeof(float)	s'évalue à 4
sizeof(double)	s'évalue à 8

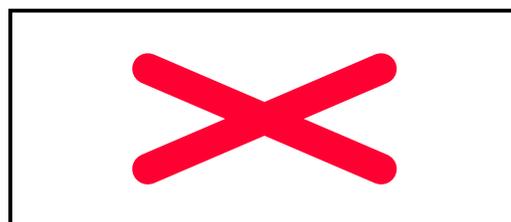
Exemple :

Nous voulons réserver de la mémoire pour X valeurs du type **int**; la valeur de X est lue au clavier :

```
int X;
int *PNum;
printf("Introduire le nombre de valeurs :");
scanf("%d", &X);
PNum = malloc(X*sizeof(int));
```

c) exit :

S'il n'y a pas assez de mémoire pour effectuer une action avec succès, il est conseillé d'interrompre l'exécution du programme à l'aide de la commande **exit** (de <stdlib>) et de renvoyer une valeur différente de zéro comme code d'erreur du programme (voir aussi chapitre 10.4).



Exemple :

Le programme à la page suivante lit 10 phrases au clavier, recherche des blocs de mémoire libres assez grands pour la mémorisation et passe les adresses aux composantes du tableau **TEXTE[]**. S'il n'y a pas assez de mémoire pour une chaîne, le programme affiche un message d'erreur et interrompt le programme avec le code d'erreur -1.

Nous devons utiliser une variable d'aide **INTRO** comme zone intermédiaire (non dynamique). Pour cette raison, la longueur maximale d'une phrase est fixée à 500 caractères.

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

main()
{
  /* Déclarations */
  char INTRO[500];
  char *TEXTE[10];
  int I;
  /* Traitement */
  for (I=0; I<10; I++)
```

```

{
  gets(INTRO);
  /* Réserve de la mémoire */
  TEXTE[I] = malloc(strlen(INTRO)+1);
  /* S'il y a assez de mémoire, ... */
  if (TEXTE[I])
    /* copier la phrase à l'adresse */
    /* fournie par malloc, ... */
    strcpy(TEXTE[I], INTRO);
  else
    {
      /* sinon quitter le programme */
      /* après un message d'erreur. */
      printf("ERREUR : Pas assez de mémoire \n");
      exit(-1);
    }
}
return (0);
}

```

-
- [Exercice 9.20](#)
-

4) La fonction free :

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de **malloc**, alors nous pouvons le libérer à l'aide de la fonction **free** de la bibliothèque `<stdlib>`.

free(<Pointeur>)

libère le bloc de mémoire désigné par le <Pointeur>; n'a pas d'effet si le pointeur a la valeur zéro.

Attention !

- * La fonction **free** peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par **malloc**.
- * La fonction **free** ne change pas le contenu du pointeur; il est conseillé d'affecter la valeur zéro au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.
- * Si nous ne libérons pas explicitement la mémoire à l'aide **free**, alors elle est libérée automatiquement à la fin du programme.

-
- [Exercice 9.21](#)
 - [Exercice 9.22](#)
 - [Exercice 9.23](#)
-

VI) Exercices d'application :

Exercice 9.1 :

```
main()
{
    int A = 1;
    int B = 2;
    int C = 3;
    int *P1, *P2;
    P1=&A;
    P2=&C;
    *P1=(*P2)++;
    P1=P2;
    P2=&B;
    *P1-=*P2;
    ++*P2;
    *P1*=*P2;
    A=++*P2**P1;
    P1=&A;
    *P2=*P1/=*P2;
    return (0);
}
```

Copiez le tableau suivant et complétez-le pour chaque instruction du programme ci-dessus.

	<u>A</u>	<u>B</u>	<u>C</u>	<u>P1</u>	<u>P2</u>
Init.	1	2	3	/	/
P1=&A	1	2	3	&A	/
P2=&C					
*P1=(*P2)++					
P1=P2					
P2=&B					
*P1-=*P2					
++*P2					
P1=*P2					
A=++*P2**P1					
P1=&A					
*P2=*P1/=*P2					

Exercice 9.2 :

Ecrire un programme qui lit deux tableaux A et B et leurs dimensions N et M au clavier et qui ajoute les éléments de B à la fin de A. Utiliser le formalisme pointeur à chaque fois que cela est possible.

Exercice 9.3 :

Pourquoi les créateurs du standard ANSI-C ont-ils décidé de légaliser les pointeurs sur le premier élément derrière un tableau? Donner un exemple.

Exercice 9.4 :

Soit P un pointeur qui 'pointe' sur un tableau A :

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;
```

Quelles valeurs ou adresses fournissent ces expressions :

- a) *P+2
- b) *(P+2)
- c) &P+1
- d) &A[4] - 3
- e) A+3
- f) &A[7] - P
- g) P+(*P-10)
- h) *(P+(P+8) - A[7])

Exercice 9.5 :

Ecrire un programme qui lit un entier X et un tableau A du type **int** au clavier et élimine toutes les occurrences de X dans A en tassant les éléments restants. Le programme utilisera les pointeurs P1 et P2 pour parcourir le tableau.

Exercice 9.6 :

Ecrire un programme qui range les éléments d'un tableau A du type **int** dans l'ordre inverse. Le programme utilisera des pointeurs P1 et P2 et une variable numérique AIDE pour la permutation des éléments.

Exercice 9.7 :

Ecrire un programme qui lit deux tableaux d'entiers A et B et leurs dimensions N et M au clavier et qui ajoute les éléments de B à la fin de A. Utiliser deux pointeurs PA et PB pour le transfert et afficher le tableau résultant A.

Exercice 9.8 :

Ecrire de deux façons différentes, un programme qui vérifie sans utiliser une fonction de *<string>*, si une chaîne CH introduite au clavier est un palindrome :

- a) en utilisant uniquement le formalisme tableau
- b) en utilisant des pointeurs au lieu des indices numériques

Rappel :

Un palindrome est un mot qui reste le même qu'on le lise de gauche à droite ou de droite à gauche :

Exemples :

```
PIERRE ==> n'est pas un palindrome
OTTO    ==> est un palindrome
23432   ==> est un palindrome
```

Exercice 9.9 :

Ecrire un programme qui lit une chaîne de caractères CH et détermine la longueur de la chaîne à l'aide d'un pointeur P. Le programme n'utilisera pas de variables numériques.

Exercice 9.10 :

Ecrire un programme qui lit une chaîne de caractères CH et détermine le nombre de mots contenus dans la chaîne. Utiliser un pointeur P, une variable logique, la fonction **isspace** et une variable numérique N qui contiendra le nombre des mots.

Exercice 9.11 :

Ecrire un programme qui lit une chaîne de caractères CH au clavier et qui compte les occurrences des lettres de l'alphabet en ne distinguant pas les majuscules et les minuscules. Utiliser un tableau ABC de dimension 26 pour mémoriser le résultat et un pointeur PCH pour parcourir la chaîne CH et un pointeur PABC pour parcourir ABC. Afficher seulement le nombre des lettres qui apparaissent au moins une fois dans le texte.

Exemple :

```
Entrez un ligne de texte (max. 100 caractères) :
Jeanne
La chaîne "Jeanne" contient :
1 fois la lettre 'A'
2 fois la lettre 'E'
1 fois la lettre 'J'
3 fois la lettre 'N'
```

Exercice 9.12 :

Ecrire un programme qui lit un caractère C et une chaîne de caractères CH au clavier. Ensuite toutes les occurrences de C dans CH seront éliminées. Le reste des caractères dans CH sera tassé à l'aide d'un pointeur et de la fonction **strcpy**.

Exercice 9.13 :

Ecrire un programme qui lit deux chaînes de caractères CH1 et CH2 au clavier et élimine toutes les lettres de CH1 qui apparaissent aussi dans CH2. Utiliser deux pointeurs P1 et P2, une variable logique TROUVE et la fonction **strcpy**.

Exemples :

```
Bonjour Bravo ==> njou
Bonjour bravo ==> Bnjou
abacab aa ==> bcab
```

Exercice 9.14 :

Ecrire un programme qui lit deux chaînes de caractères CH1 et CH2 au clavier et supprime la première occurrence de CH2 dans CH1. Utiliser uniquement des pointeurs, une variable logique TROUVE et la fonction **strcpy**.

Exemples :

```
Alphonse phon ==> Alse
totalement t ==> otatement
abacab aa ==> abacab
```

Exercice 9.15 :

Ecrire un programme qui lit une matrice A de dimensions N et M au clavier et affiche les données suivantes en utilisant le formalisme pointeur à chaque fois que cela est possible :

- a) la matrice A
- b) la transposée de A
- c) la matrice A interprétée comme tableau unidimensionnel

Exercice 9.16 :

Ecrire un programme qui lit deux matrices A et B de dimensions N et M respectivement M et P au clavier et qui effectue la multiplication des deux matrices. Le résultat de la multiplication sera affecté à la matrice C, qui sera ensuite affichée. Utiliser le formalisme pointeur à chaque fois que cela est possible.

Exercice 9.17 :

Ecrire un programme qui lit 5 mots d'une longueur maximale de 50 caractères et les mémorise dans un tableau de chaînes de caractères TABCH. Inverser l'ordre des caractères à l'intérieur des 5 mots à l'aide de deux pointeurs P1 et P2. Afficher les mots.

Exercice 9.18 :

Considérez les déclarations de **NOM1** et **NOM2** :

```
char *NOM1 [] = {"Marc", "Jean-Marie", "Paul",  
                "François-Xavier", "Claude" };
```

```
char NOM2 [] [16] = {"Marc", "Jean-Marie", "Paul",  
                    "François-Xavier", "Claude" };
```

- a) Représenter graphiquement la mémorisation des deux variables **NOM1** et **NOM2**.
- b) Imaginez que vous devez écrire un programme pour chacun des deux tableaux qui trie les chaînes selon l'ordre lexicographique. En supposant que vous utilisez le même algorithme de tri pour les deux programmes, lequel des deux programmes sera probablement le plus rapide?

Exercice 9.19 :

Ecrire un programme qui lit le jour, le mois et l'année d'une date au clavier et qui affiche la date en français et en allemand. Utiliser deux tableaux de pointeurs, **MFRAN** et **MDEUT** que vous initialisez avec les noms des mois dans les deux langues. La première composante de chaque tableau contiendra un message d'erreur qui sera affiché lors de l'introduction d'une donnée illégale.

Exemples :

```
Introduisez la date : 1 4 1993  
Luxembourg, le 1er avril 1993  
Luxemburg, den 1. April 1993
```

```
Introduisez la date : 2 4 1993  
Luxembourg, le 2 avril 1993  
Luxemburg, den 2. April 1993
```

Exercice 9.20 :

Ecrire un programme qui lit 10 phrases d'une longueur maximale de 200 caractères au clavier et qui les mémorise dans un tableau de pointeurs sur **char** en réservant dynamiquement l'emplacement en mémoire pour les chaînes. Ensuite, l'ordre des phrases est inversé en modifiant les pointeurs et le tableau résultant est affiché.

Exercice 9.21 :

Ecrire un programme qui lit 10 mots au clavier (longueur maximale : 50 caractères) et attribue leurs adresses à un tableau de pointeurs MOT. Effacer les 10 mots un à un, en suivant l'ordre lexicographique et en libérant leur espace en mémoire. Afficher à chaque fois les mots restants en attendant la confirmation de l'utilisateur (par 'Enter').

Exercice 9.22 :

Ecrire un programme qui lit 10 mots au clavier (longueur maximale : 50 caractères) et attribue leurs adresses à un tableau de pointeurs MOT. Copier les mots selon l'ordre lexicographique en une seule 'phrase' dont l'adresse est affectée à un pointeur PHRASE. Réserver l'espace nécessaire à la PHRASE avant de copier les mots. Libérer la mémoire occupée par chaque mot après l'avoir copié. Utiliser les fonctions de *<string>*.

Exercice 9.23 :

Ecrire un programme qui lit 10 phrases au clavier (longueur maximale : 50 caractères) et attribue leurs adresses à un tableau de pointeurs MOT. Réserver dynamiquement l'emplacement en mémoire pour les mots. Trier les phrases lexicographiquement en n'échangeant que les pointeurs. Utiliser la méthode de tri par propagation (méthode de la bulle - décrite dans l'exercice 7.15).

VII) Solutions des exercices du Chapitre 9 : LES POINTEURS :

Exercice 9.1 :

	<u>A</u>	<u>B</u>	<u>C</u>	<u>P1</u>	<u>P2</u>
Init.	1	2	3	/	/
P1=&A	1	2	3	&A	/
P2=&C	1	2	3	&A	&C
*P1=(*P2)++	3	2	4	&A	&C
P1=P2	3	2	4	&C	&C
P2=&B	3	2	4	&C	&B
*P1-=*P2	3	2	2	&C	&B
++*P2	3	3	2	&C	&B
P1=*P2	3	3	6	&C	&B
A=++*P2**P1	24	4	6	&C	&B
P1=&A	24	4	6	&A	&B
*P2=*P1/=*P2	6	6	6	&A	&B

Exercice 9.2 :

```
#include <stdio.h>
main()
{
    /* Déclarations */
    int A[100], B[50]; /* tableaux */
    int N, M; /* dimensions des tableaux */
    int I; /* indice courant */

    /* Saisie des données */
    printf("Dimension du tableau A (max.50) : ");
    scanf("%d", &N );
    for (I=0; I<N; I++)
    {
        printf("Elément %d : ", I);
        scanf("%d", A+I);
    }
    printf("Dimension du tableau B (max.50) : ");
    scanf("%d", &M );
    for (I=0; I<M; I++)
    {
        printf("Elément %d : ", I);
        scanf("%d", B+I);
    }

    /* Affichage des tableaux */
    printf("Tableau donné A :\n");
    for (I=0; I<N; I++)
        printf("%d ", *(A+I));
    printf("\n");
    printf("Tableau donné B :\n");
    for (I=0; I<M; I++)
        printf("%d ", *(B+I));
}
```

```

printf("\n");
/* Copie de B à la fin de A */
for (I=0; I<M; I++)
    *(A+N+I) = *(B+I);
/* Nouvelle dimension de A */
N += M;
/* Edition du résultat */
printf("Tableau résultat A :\n");
for (I=0; I<N; I++)
    printf("%d ", *(A+I));
printf("\n");
return (0);
}

```

Exercice 9.3 :

Solution :

En traitant des tableaux à l'aide de pointeurs, nous utilisons souvent des expressions de la forme :

```

for (P=A ; P<A+N ; P++) ou for (P=CH ; *P ; P++)
{
    ...
}

```

ou les versions analogues avec **while**.

Dans ces boucles, le pointeur P est incrémenté à la fin du bloc d'instruction et comparé ensuite à la condition de la boucle. Au moment où la condition est remplie, P pointe déjà à l'extérieur du tableau; plus précisément sur le premier élément derrière le tableau.

Exemple :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[10]; /* tableau */
    int *P;    /* pointeur dans A */

    /* Saisie des données */
    printf("Introduire 10 entiers : \n");
    for (P=A; P<A+10; P++)
        scanf("%d", P);
    /* Affichage du tableau */
    printf("Tableau donné A :\n");
    for (P=A; P<A+10; P++)
        printf("%d ", *P);
    printf("\n");
    return (0);
}

```

À la fin des boucles, P contient l'adresse A+10 et pointe donc sur l'élément A[10] qui ne fait plus partie du tableau.

Exercice 9.4 :

Soit P un pointeur qui 'pointe' sur un tableau A :

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;
```

Quelles valeurs ou adresses fournissent ces expressions :

- a) *P+2 => la valeur 14
- b) *(P+2) => la valeur 34
- c) &P+1 => l'adresse du pointeur derrière le pointeur P
 (rarement utilisée)
- d) &A[4]-3 => l'adresse de la composante A[1]
- e) A+3 => l'adresse de la composante A[3]
- f) &A[7]-P => la valeur (indice) 7
- g) P+(*P-10) => l'adresse de la composante A[2]
- h) *(P+(P+8)-A[7]) => la valeur 23

Exercice 9.5 :

```
#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50];      /* tableau donné          */
    int N;          /* dimension du tableau */
    int X;          /* valeur à éliminer    */
    int *P1, *P2; /* pointeurs d'aide     */

    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (P1=A; P1<A+N; P1++)
    {
        printf("Elément %d : ", P1-A);
        scanf("%d", P1);
    }
    printf("Introduire l'élément X à éliminer du tableau : ");
    scanf("%d", &X );
    /* Affichage du tableau */
    for (P1=A; P1<A+N; P1++)
        printf("%d ", *P1);
    printf("\n");
    /* Effacer toutes les occurrences de X et comprimer : */
    /* Copier tous les éléments de P1 vers P2 et augmenter */
    /* P2 pour tous les éléments différents de X.          */
    for (P1=P2=A; P1<A+N; P1++)
    {
        *P2 = *P1;
        if (*P2 != X)
            P2++;
    }
    /* Nouvelle dimension de A */
    N = P2-A;
    /* Edition du résultat */
    for (P1=A; P1<A+N; P1++)
        printf("%d ", *P1);
    printf("\n");
}
```

```

    return (0);
}

```

Exercice 9.6 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50];      /* tableau donné          */
    int N;          /* dimension du tableau */
    int AIDE;       /* pour la permutation  */
    int *P1, *P2;  /* pointeurs d'aide     */
    /* Saisie des données */
    printf("Dimension du tableau (max.50) : ");
    scanf("%d", &N );
    for (P1=A; P1<A+N; P1++)
    {
        printf("Elément %d : ", P1-A);
        scanf("%d", P1);
    }
    /* Affichage du tableau */
    for (P1=A; P1<A+N; P1++)
        printf("%d ", *P1);
    printf("\n");
    /* Inverser la tableau */
    for (P1=A, P2=A+(N-1); P1<P2; P1++, P2--)
    {
        AIDE = *P1;
        *P1 = *P2;
        *P2 = AIDE;
    }
    /* Edition du résultat */
    for (P1=A; P1<A+N; P1++)
        printf("%d ", *P1);
    printf("\n");
    return (0);
}

```

Exercice 9.7 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[100], B[50]; /* tableaux */
    int N, M;          /* dimensions des tableaux */
    int *PA, *PB;     /* pointeurs d'aide          */

    /* Saisie des données */
    printf("Dimension du tableau A (max.50) : ");
    scanf("%d", &N );
    for (PA=A; PA<A+N; PA++)

```

```

    {
        printf("Elément %d : ", PA-A);
        scanf("%d", PA);
    }
    printf("Dimension du tableau B (max.50) : ");
    scanf("%d", &M );
    for (PB=B; PB<B+M; PB++)
    {
        printf("Elément %d : ", PB-B);
        scanf("%d", PB);
    }
    /* Affichage des tableaux */
    printf("Tableau donné A :\n");
    for (PA=A; PA<A+N; PA++)
        printf("%d ", *PA);
    printf("\n");
    printf("Tableau donné B :\n");
    for (PB=B; PB<B+M; PB++)
        printf("%d ", *PB);
    printf("\n");
    /* Copier B à la fin de A */
    for (PA=A+N,PB=B ; PB<B+M ; PA++,PB++)
        *PA = *PB;
    /* Nouvelle dimension de A */
    N += M;
    /* Edition du résultat */
    printf("Tableau résultat A :\n");
    for (PA=A; PA<A+N; PA++)
        printf("%d ", *PA);
    printf("\n");
    return 0;
}

```

Exercice 9.8 :

a) en utilisant uniquement le formalisme tableau

```

#include <stdio.h>
main()
{
    /* Déclarations */
    char CH[101]; /* chaîne donnée */
    int I,J;      /* indices courants */
    int PALI;     /* indicateur logique : */
                 /* vrai si CH est un palindrome */

    /* Saisie des données */
    printf("Entrez une ligne de texte (max.100 caractères)
:\n");
    gets(CH);
    /* Placer J sur la dernière lettre de la chaîne */
    for(J=0; CH[J]; J++)
        ;
    J--;
    /* Contrôler si CH est un palindrome */

```

```

PALI=1;
for (I=0 ; PALI && I<J ; I++,J--)
    if (CH[I] != CH[J])
        PALI=0;

/* Affichage du résultat */
if (PALI)
    printf("La chaîne \"%s\" est un palindrome.\n", CH);
else
    printf("La chaîne \"%s\" n'est pas un palindrome.\n",
CH);
return (0);
}

```

b) en utilisant des pointeurs au lieu des indices numériques :

```

#include <stdio.h>
main()
{
/* Déclarations */
char CH[101]; /* chaîne donnée */
char *P1,*P2; /* pointeurs d'aide */
int PALI; /* indicateur logique : */
/* vrai si CH est un palindrome */

/* Saisie des données */
printf("Entrez une ligne de texte (max.100 caractères)
:\n");
gets(CH);
/* Placer P2 sur la dernière lettre de la chaîne */
for (P2=CH; *P2; P2++)
    ;
P2--;
/* Contrôler si CH est un palindrome */
PALI=1;
for (P1=CH ; PALI && P1<P2 ; P1++,P2--)
    if (*P1 != *P2) PALI=0;
/* Affichage du résultat */
if (PALI)
    printf("La chaîne \"%s\" est un palindrome.\n", CH);
else
    printf("La chaîne \"%s\" n'est pas un palindrome.\n",
CH);
return (0);
}

```

Exercice 9.9 :

```

#include <stdio.h>
main()
{
/* Déclarations */
char CH[101]; /* chaîne donnée */
char *P; /* pointeur d'aide */

/* Saisie des données */

```

```

printf("Entrez une ligne de texte (max.100 caractères)
:\n");
gets(CH);
/* Placer P à la fin de la chaîne */
for (P=CH; *P; P++)
    ;
/* Affichage du résultat */
printf("La chaîne \"%s\" est formée de %d caractères.\n",
                                             CH,    P-
CH);
return (0);
}

```

Exercice 9.10 :

```

#include <stdio.h>
#include <ctype.h>
main()
{
/* Déclarations */
char CH[101]; /* chaîne donnée */
char *P;      /* pointeur d'aide */
int N;        /* nombre des mots */
int DANS_MOT; /* indicateur logique : */
              /* vrai si P pointe à l'intérieur un mot */

/* Saisie des données */
printf("Entrez une ligne de texte (max.100 caractères)
:\n");
gets(CH);
/* Compter les mots */
N=0;
DANS_MOT=0;
for (P=CH; *P; P++)
    if (isspace(*P))
        DANS_MOT=0;
    else if (!DANS_MOT)
        {
            DANS_MOT=1;
            N++;
        }
/* Affichage du résultat (pour perfectionnistes) */
printf("La chaîne \"%s\" \nest formée de %d mot%c.\n",
                                             CH,    N,    (N==1)?'
': 's');
return (0);
}

```

Exercice 9.11 :

```

#include <stdio.h>
main()
{
/* Déclarations */

```

```

char CH[101]; /* chaîne donnée */
char *PCH;    /* pointeur d'aide dans CH */
int ABC[26]; /* compteurs des différents caractères */
int *PABC;    /* pointeur d'aide dans ABC */

/* Saisie des données */
printf("Entrez une ligne de texte (max.100 caractères)
:\n");
gets(CH);
/* Initialiser le tableau ABC */
for (PABC=ABC; PABC<ABC+26; PABC++)
    *PABC=0;
/* Compter les lettres */
for (PCH=CH; *PCH; PCH++)
    {
        if (*PCH>='A' && *PCH<='Z')
            (*(ABC+(*PCH-'A')))+; /* Attention aux parenthèses!
*/
        if (*PCH>='a' && *PCH<='z')
            (*(ABC+(*PCH-'a')))+;
    }
/* Affichage des résultats */
/* (PABC-ABC) est le numéro de la lettre de l'alphabet. */
printf("La chaîne \"%s\" contient :\n", CH);
for (PABC=ABC; PABC<ABC+26; PABC++)
    if (*PABC)
        printf(" %d\tfois la lettre '%c' \n",
                *PABC, 'A'+(PABC-
ABC));
return (0);
}

```

Exercice 9.12 :

```

#include <stdio.h>
#include <string.h>
main()
{
    /* Déclarations */
    char CH[101]; /* chaîne donnée */
    char C;      /* lettre à éliminer */
    char *P;     /* pointeur d'aide dans CH */

    /* Saisie des données */
    printf("Entrez une ligne de texte (max.100 caractères)
:\n");
    gets(CH);
    printf("Entrez le caractère à éliminer (suivi de Enter) :
");
    C=getchar();
    /* Comprimer la chaîne à l'aide de strcpy */
    for (P=CH; *P; P++)
        if (*P==C)
            strcpy(P, P+1);
}

```

```

    /* Affichage du résultat */
    printf("Chaîne résultat : \"%s\"\n", CH);
    return (0);
}

```

Exercice 9.13 :

```

#include <stdio.h>
#include <string.h>
main()
{
    /* Déclarations */
    char CH1[101], CH2[101]; /* chaînes données */
    char *P1, *P2; /* pointeurs d'aide dans CH1 et CH2 */
    int TROUVE; /* indicateur logique : vrai, si le caractère */
                /* actuel dans CH1 a été trouvé dans CH2. */

    /* Saisie des données */
    printf("Entrez la première chaîne de caractères"
           " (max.100 caractères) :\n");
    gets(CH1);
    printf("Entrez la deuxième chaîne de caractères"
           " (max.100 caractères) :\n");
    gets(CH2);
    /* Eliminer les lettres communes */
    /* Idée : Parcourir CH2 de gauche à droite et contrôler */
    /* pour chaque caractère s'il se trouve aussi dans CH1. */
    /* Si tel est le cas, éliminer le caractère de CH1 à */
    /* l'aide de strcpy. */
    for (P2=CH2; *P2; P2++)
    {
        TROUVE = 0;
        for (P1=CH1 ; *P1 && !TROUVE ; P1++)
            if (*P2==*P1)
            {
                TROUVE = 1;
                strcpy(P1, P1+1);
            }
    }
    /* Affichage du résultat */
    printf("Chaîne résultat : \"%s\" \n", CH1);
    return (0);
}

```

Exercice 9.14 :

```

#include <stdio.h>
#include <string.h>
main()
{
    /* Déclarations */
    char CH1[101], CH2[101]; /* chaînes données */
    char *P1, *P2; /* pointeurs d'aide dans CH1 et CH2 */
    int TROUVE; /* indicateur logique : vrai, si le caractère */

```

```

        /* actuel dans CH1 a été trouvé dans CH2.    */

/* Saisie des données */
printf("Entrez la chaîne à transformer"
       " (max.100 caractères) :\n");
gets(CH1);
printf("Entrez la chaîne à supprimer  "
       " (max.100 caractères) :\n");
gets(CH2);
/* Rechercher CH2 dans CH1 : */
/* L'expression P2-CH2 est utilisée pour déterminer l'indice
*/
/* de P2 dans CH2. On pourrait aussi résoudre le problème à
*/
/* l'aide d'un troisième pointeur P3 parcourant CH1. */
TROUVE=0;
for (P1=CH1 ; *P1 && !TROUVE ; P1++)
    {
        for (P2=CH2 ; *P2 == *(P1+(P2-CH2)) ; P2++)
            ;
        if (!*P2)
            TROUVE = 1;
    }

/* A la fin de la boucle, P1 est incrémenté, donc */
P1--;
/* Si CH2 se trouve dans CH1, alors P1 indique la position
*/
/* de la première occurrence de CH2 dans CH1 et P2 pointe à
*/
/* la fin de CH2. (P2-CH2) est alors la longueur de CH2.
*/
if (TROUVE)
    strcpy(P1, P1+(P2-CH2));

/* Affichage du résultat */
printf("Chaîne résultat : \"%s\" \n", CH1);
return (0);
}

```

Exercice 9.15 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50][50]; /* matrice */
    int N, M; /* dimensions de la matrice */
    int I, J; /* indices courants */
    /* Saisie des données */
    printf("Nombre de lignes (max.50) : ");
    scanf("%d", &N );
    printf("Nombre de colonnes (max.50) : ");
    scanf("%d", &M );
}

```

```

/* Lecture de la matrice au clavier */
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", (int *)A+I*50+J);
        }

/* a) Affichage de la matrice */
printf("Matrice donnée :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", *((int *)A+I*50+J));
        printf("\n");
    }

/* b) Affichage de la transposée de A */
printf("Matrice transposée :\n");
for (J=0; J<M; J++)
    {
        for (I=0; I<N; I++)
            printf("%7d ", *((int *)A+I*50+J));
        printf("\n");
    }

/* c) Interprétation de la matrice comme vecteur : */
/* Attention, ce serait une faute grave d'afficher */
/* 'simplement' les NxM premiers éléments de A ! */
printf("Matrice affichée comme vecteur :\n");
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        printf(" %d ", *((int *)A+I*50+J));
printf("\n");
return (0);
}

```

Exercice 9.16 :

```

#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50][50]; /* matrice donnée */
    int B[50][50]; /* matrice donnée */
    int C[50][50]; /* matrice résultat */
    int N, M, P; /* dimensions des matrices */
    int I, J, K; /* indices courants */

    /* Saisie des données */
    printf("*** Matrice A ***\n");
    printf("Nombre de lignes de A (max.50) : ");
    scanf("%d", &N );
    printf("Nombre de colonnes de A (max.50) : ");

```

```

scanf("%d", &M );
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", (int *)A+I*50+J);
        }
printf("*** Matrice B ***\n");
printf("Nombre de lignes de B : %d\n", M);
printf("Nombre de colonnes de B (max.50) : ");
scanf("%d", &P );
for (I=0; I<M; I++)
    for (J=0; J<P; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", (int *)B+I*50+J);
        }

/* Affichage des matrices */
printf("Matrice donnée A :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<M; J++)
            printf("%7d", *((int *)A+I*50+J));
        printf("\n");
    }
printf("Matrice donnée B :\n");
for (I=0; I<M; I++)
    {
        for (J=0; J<P; J++)
            printf("%7d", *((int *)B+I*50+J));
        printf("\n");
    }

/* Affectation du résultat de la multiplication à C */
for (I=0; I<N; I++)
    for (J=0; J<P; J++)
        {
            *((int *)C+I*50+J)=0;
            for (K=0; K<M; K++)
                *((int *)C+I*50+J) += *((int *)A+I*50+K) * *((int *)B+K*50+J);
        }

/* Edition du résultat */
printf("Matrice résultat C :\n");
for (I=0; I<N; I++)
    {
        for (J=0; J<P; J++)
            printf("%7d", *((int *)C+I*50+J));
        printf("\n");
    }
return (0);
}

```

Exercice 9.17 :

```
#include <stdio.h>
main()
{
    /* Déclarations */
    char TABCH[5][51]; /* tableau de chaînes de caractères */
    char AIDE;          /* pour la permutation des caractères */
    char *P1, *P2;     /* pointeurs d'aide */
    int I;              /* indice courant */

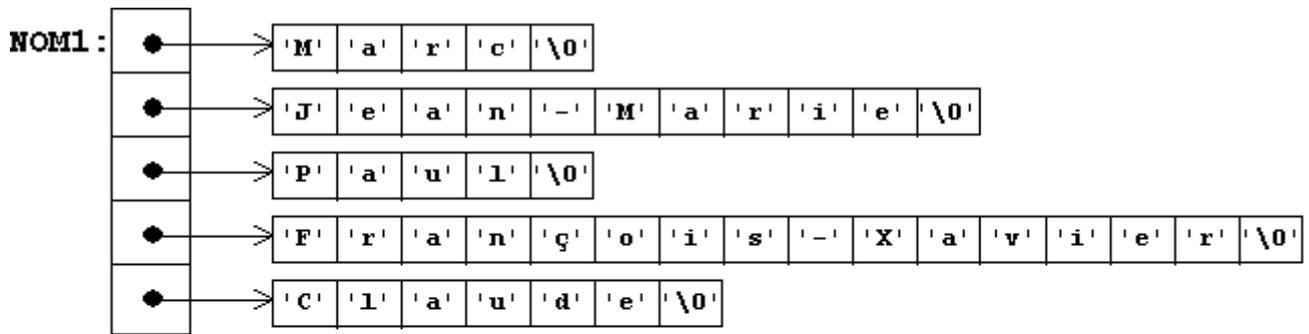
    /* TABCH+I est l'adresse de la I-ième chaîne du tableau */
    /* Il vaut mieux convertir TABCH+I en pointeur sur char */
    /* Saisie des données */
    printf("Entrez 5 mots :\n");
    for (I=0; I<5; I++)
    {
        printf("Mot %d (max.50 caractères) : ", I);
        gets((char *) (TABCH+I));
    }

    /* Inverser l'ordre des caractères à l'intérieur des mots */
    for (I=0; I<5; I++)
    {
        P1 = P2 = (char *) (TABCH+I);
        /* Placer P2 à la fin de la chaîne */
        while (*P2)
            P2++;
        P2--; /* sinon '\0' est placé au début de la chaîne */
        while (P1<P2)
        {
            AIDE = *P1;
            *P1 = *P2;
            *P2 = AIDE;
            P1++;
            P2--;
        }
    }

    /* Affichage des mots inversés */
    for (I=0; I<5; I++)
        puts((char *) (TABCH+I));
    return (0);
}
```

Exercice 9.18 :

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



NOM2 :

'M'	'a'	'r'	'c'	'\0'															
'J'	'e'	'a'	'n'	'-'	'M'	'a'	'r'	'i'	'e'	'\0'									
'P'	'a'	'u'	'l'	'\0'															
'F'	'r'	'a'	'n'	'ç'	'o'	'i'	's'	'-'	'X'	'a'	'v'	'i'	'e'	'r'	'\0'				
'C'	'l'	'a'	'u'	'd'	'e'	'\0'													

b)

Pour trier les chaînes du tableau de pointeurs, il faut uniquement changer les pointeurs. La durée d'une opération d'échange est constante, peu importe la longueur des chaînes.

Pour trier le tableau de chaînes de caractères, il faut changer tous les caractères des chaînes un à un. La durée d'une opération d'échange est dépendante de la longueur des chaînes de caractères.

Pour des chaînes de caractères d'une longueur 'normale', le tri d'un tableau de pointeurs est donc certainement plus rapide que le tri d'un tableau de chaînes de caractères.

Exercice 9.19 :

```
#include <stdio.h>
main()
{
    /* Tableaux de pointeurs sur des chaînes de caractères */
    char *MFRAN[] = {"\aErreur d'entrée !", "janvier",
                    "février",
                    "mars", "avril", "mai", "juin", "juillet",
                    "août", "septembre", "octobre", "novembre",
                    "décembre"};
    char *MDEUT[] = {"\aEingabefehler !", "Januar", "Februar",
                    "März", "April", "Mai", "Juni", "Juli",
                    "August", "September", "Oktober",
                    "November",
                    "Dezember"};
    int JOUR, MOIS, ANNEE; /* données pour la date */
    int CORRECT; /* indicateur logique : */
                    /* vrai si la date entrée est correcte */
    /* Saisie des données */
    do
    {
        printf("Introduire le jour, le mois et l'année : ");
        scanf("%d %d %d", &JOUR, &MOIS, &ANNEE);
        CORRECT=1;
        if (JOUR<0 || JOUR>31 || MOIS<0 || MOIS>12 || ANNEE<0 || ANNEE>3000)
        {
            CORRECT=0;
            puts(MFRAN[0]);
        }
    }
}
```

```

        puts(MDEUT[0]);
    }
}
while (!CORRECT);
/* Affichage des dates */
printf("Luxembourg, le %d%s %s %d \n",
    JOUR, (JOUR==1)?"er":"", MFRAN[MOIS], ANNEE);
printf("Luxembourg, den %d. %s %d \n", JOUR, MDEUT[MOIS],
ANNEE);
return (0);
}

```

Exercice 9.20 :

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main()
{
    /* Déclarations */
    char INTRO[500]; /* chaîne pour l'introduction des données
    */
    char *TEXTE[10]; /* Tableau des pointeurs sur les 10 chaînes
    */
    char *PAIDE; /* pointeur d'aide pour l'échange des pointeurs
    */
    int I,J;      /* indices courants */
    /* Saisie des données et allocation dynamique de mémoire */
    puts("Introduire 10 phrases terminées chaque fois"
        " par un retour à la ligne :");
    for (I=0; I<10; I++)
    {
        /* Lecture d'une phrase */
        printf("Phrase %d : ",I);
        gets(INTRO);
        /* Réserve de la mémoire */
        TEXTE[I] = malloc(strlen(INTRO)+1);
        /* S'il y a assez de mémoire, ... */
        if (TEXTE[I])
            /* copier la phrase à l'adresse */
            /* fournie par malloc, */
            strcpy(TEXTE[I], INTRO);
        else
        {
            /* sinon afficher un message d'erreur */
            printf("\aPas assez de mémoire \n");
            /* et quitter le programme. */
            exit(-1);
        }
    }
    /* Afficher le tableau donné */
    puts("Contenu du tableau donné :");
    for (I=0; I<10; I++) puts(TEXTE[I]);
    /* Inverser l'ordre des phrases avec le pointeur PAIDE */
}

```

```

for (I=0,J=9 ; I<J ; I++,J--)
{
    PAIDE      = TEXTE[I];
    TEXTE[I]   = TEXTE[J];
    TEXTE[J]   = PAIDE;
}
/* Afficher le tableau résultat */
puts("Contenu du tableau résultat :");
for (I=0; I<10; I++) puts(TEXTE[I]);
return (0);
}

```

Exercice 9.21 :

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main()
{
    /* Déclarations */
    char INTRO[51]; /* chaîne pour l'introduction des données */
    char *MOT[10]; /* Tableau de pointeurs sur les 10 chaînes */
    int MAX;      /* indice du prochain candidat à supprimer */
    int I,J;     /* indices courants */
    /* Saisie des données et allocation dynamique de mémoire */
    puts("Introduire 10 phrases terminées chaque fois"
         " par un retour à la ligne :");
    for (I=0; I<10; I++)
    {
        /* Lecture d'une phrase */
        printf("Phrase %d : ",I);
        gets(INTRO);
        /* Réserve de la mémoire */
        MOT[I] = malloc(strlen(INTRO)+1);
        /* S'il y a assez de mémoire, ... */
        if (MOT[I])
            /* copier la phrase à l'adresse */
            /* fournie par malloc, */
            strcpy(MOT[I], INTRO);
        else
        {
            /* sinon afficher un message d'erreur */
            printf("\aPas assez de mémoire \n");
            /* et quitter le programme. */
            exit(-1);
        }
    }

    /* Suppression des mots du tableau par ordre lexicographique
    */
    for (I=0; I<10; I++)
    {
        /* Recherche de la dernière chaîne dans l'ordre */
        /* lexicographique : Initialiser d'abord MAX avec */

```

```

/* l'indice d'une chaîne encore existante. */
/* Les conditions de la forme MOT[I] ou !MOT[I] */
/* contrôlent si la chaîne I a déjà été supprimée */
/* ou non. */
for (MAX=0 ; !MOT[MAX] ; MAX++)
    ;
for (J=MAX; J<10; J++)
    if (MOT[J] && strcmp(MOT[MAX], MOT[J])>0)
        MAX=J;
/* Suppression de la chaîne */
free(MOT[MAX]);
MOT[MAX]=0;
/* Affichage des mots restants */
printf("Passage No.%d :\n", I);
for (J=0; J<10; J++)
    if (MOT[J]) puts(MOT[J]);
printf("Poussez Enter pour continuer ... \n");
getchar();
}
return (0);
}

```

Exercice 9.22 :

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main()
{
    /* Déclarations */
    char INTRO[51]; /* chaîne pour l'introduction des données */
    char *MOT[10]; /* Tableau de pointeurs sur les 10 chaînes */
    char *PHRASE; /* Pointeur cible */
    int MAX; /* indice du prochain candidat à copier */
    int I,J; /* indices courants */
    int L; /* Longueur de la phrase */

    /* Saisie des données et allocation dynamique de mémoire */
    puts("Introduire 10 phrases terminées chaque fois"
         " par un retour à la ligne :");
    for (I=0; I<10; I++)
    {
        /* Lecture d'une phrase */
        printf("Phrase %d : ",I);
        gets(INTRO);
        /* Réserve de la mémoire */
        MOT[I] = malloc(strlen(INTRO)+1);
        if (MOT[I])
            strcpy(MOT[I], INTRO);
        else
        {
            printf("\aPas assez de mémoire \n");
            exit(-1);
        }
    }
}

```

```

    }
    /* Calcul de la longueur totale de la 'phrase' */
    L=11; /* pour les espaces et le symbole de fin de chaîne */
    for (I=0; I<10; I++)
        L += (strlen(MOT[I])+1);
    /* Réserve de la mémoire pour la 'phrase' */
    PHRASE = malloc(L);
    if (!PHRASE)
    {
        printf("\aPas assez de mémoire pour"
            " mémoriser la phrase.\n");
        exit(-1);
    }
    /* Initialisation de la PHRASE */
    PHRASE[0]='\0';
    /* Copier et supprimer les mots du tableau par */
    /* ordre lexicographique */
    for (I=0; I<10; I++)
    {
        /* Recherche de la dernière chaîne dans l'ordre */
        /* lexicographique : Initialiser d'abord MAX avec */
        /* l'indice d'une chaîne encore existante. */
        for (MAX=0 ; !MOT[MAX] ; MAX++) ;
        for (J=MAX; J<10; J++)
            if (MOT[J] && strcmp(MOT[MAX], MOT[J])>0)
                MAX=J;
        /* Copier la chaîne dans la PHRASE */
        strcat(PHRASE,MOT[MAX]);
        strcat(PHRASE," ");
        /* Suppression de la chaîne */
        free(MOT[MAX]);
        MOT[MAX]=0;
    }
    /* Affichage de la PHRASE */
    puts("Résultat :");
    puts(PHRASE);
    return (0);
}

```

Exercice 9.23 :

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main()
{
    /* Déclarations */
    char INTRO[51]; /* chaîne pour l'introduction des données */
    char *MOT[10]; /* Tableau de pointeurs sur les 10 chaînes */
    int I; /* ligne à partir de laquelle MOT est trié */
    int J; /* indice courant */
    char *AIDE; /* pour la permutation des pointeurs */

```

```

int FIN; /* ligne où la dernière permutation a eu lieu */
        /* permet de ne pas trier un sous-ensemble déjà trié */

/* Saisie des données et allocation dynamique de mémoire */
puts("Introduire 10 phrases terminées chaque fois"
     " par un retour à la ligne :");
for (I=0; I<10; I++)
{
    /* Lecture d'une phrase */
    printf("Phrase %d : ",I);
    gets(INTRO);
    /* Réserve de la mémoire */
    MOT[I] = malloc(strlen(INTRO)+1);
    if (MOT[I])
        strcpy(MOT[I], INTRO);
    else
    {
        printf("\aPas assez de mémoire \n");
        exit(-1);
    }
}

/* Tri du tableau par propagation de l'élément maximal. */
for (I=9 ; I>0 ; I=FIN)
{
    FIN=0;
    for (J=0; J<I; J++)
        if (strcmp(MOT[J],MOT[J+1])>0)
        {
            FIN=J;
            AIDE = MOT[J];
            MOT[J] = MOT[J+1];
            MOT[J+1] = AIDE;
        }
}

/* Affichage du tableau */
puts("Tableau trié :");
for (I=0; I<10; I++)
    puts(MOT[I]);
return (0);
}

```

Chapitre 10 : LES FONCTIONS :

La structuration de programmes en sous-programmes se fait en C à l'aide de *fonctions*. Les fonctions en C correspondent aux fonctions *et* procédures en Pascal et en langage algorithmique. Nous avons déjà utilisé des fonctions prédéfinies dans des bibliothèques standard (**printf** de `<stdio>`, **strlen** de `<string>`, **pow** de `<math>`, etc.). Dans ce chapitre, nous allons découvrir comment nous pouvons définir et utiliser nos propres fonctions.

I) Modularisation de programmes :

Jusqu'ici, nous avons résolu nos problèmes à l'aide de fonctions prédéfinies et d'une seule fonction nouvelle : la fonction principale **main()**. Pour des problèmes plus complexes, nous obtenons ainsi de longues listes d'instructions, peu structurées et par conséquent peu compréhensibles. En plus, il faut souvent répéter les mêmes suites de commandes dans le texte du programme, ce qui entraîne un gaspillage de mémoire interne et externe.

1) La modularité et ses avantages :

La plupart des langages de programmation nous permettent de subdiviser nos programmes en sous-programmes, fonctions ou procédures plus simples et plus compacts. A l'aide de ces structures nous pouvons *modulariser* nos programmes pour obtenir des solutions plus élégantes et plus efficaces.

a) Modules :

Dans ce contexte, un *module* désigne une entité de données et d'instructions qui fournissent une solution à une (petite) partie bien définie d'un problème plus complexe. Un module peut faire appel à d'autres modules, leur transmettre des données et recevoir des données en retour. L'ensemble des modules ainsi reliés doit alors être capable de résoudre le problème global.

Dans ce manuel, les modules (en C : les fonctions) sont mis en évidence par une bordure dans la marge gauche.

b) Avantages :

Voici quelques avantages d'un programme modulaire :

- * **Meilleure lisibilité**
- * **Diminution du risque d'erreurs**
- * **Possibilité de tests sélectifs**
- * **Dissimulation des méthodes**

Lors de l'utilisation d'un module il faut seulement connaître son effet, sans devoir s'occuper des détails de sa réalisation.

* **Réutilisation de modules déjà existants**

Il est facile d'utiliser des modules qu'on a écrits soi-même ou qui ont été développés par d'autres personnes.

* **Simplicité de l'entretien**

Un module peut être changé ou remplacé sans devoir toucher aux autres modules du programme.

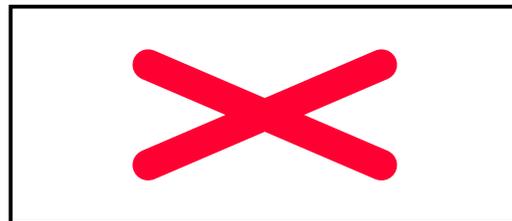
* **Favorisation du travail en équipe**

Un programme peut être développé en équipe par délégation de la programmation des modules à différentes personnes ou groupes de personnes. Une fois développés, les modules peuvent constituer une base de travail commune.

* **Hiérarchisation des modules**

Un programme peut d'abord être résolu globalement au niveau du module principal. Les détails peuvent être reportés à des modules sous-ordonnés qui peuvent eux aussi être subdivisés en sous-modules et ainsi de suite. De cette façon, nous obtenons une *hiérarchie de modules*. Les modules peuvent être développés en passant du haut vers le bas dans la hiérarchie (*'top-down-development'* - *méthode du raffinement progressif*) ou bien en passant du bas vers le haut (*'bottom-up-development'*).

En principe, la méthode du raffinement progressif est à préférer, mais en pratique, on aboutit souvent à une méthode hybride, qui construit un programme en considérant aussi bien le problème posé que les possibilités de l'ordinateur ciblé (*'méthode du jo-jo'*).



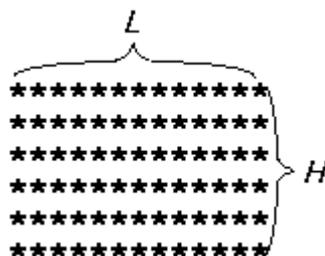
2) Exemples de modularisation en C :

Les deux programmes présentés ci-dessous vous donnent un petit aperçu sur les propriétés principales des fonctions en C. Les détails seront discutés plus loin dans ce chapitre.

a) Exemple 1 : Afficher un rectangle d'étoiles :

Commençons par un petit programme que nous vous proposons d'examiner vous-mêmes sans autres explications :

Le programme suivant permet d'afficher à l'écran un rectangle de longueur L et de hauteur H, formé d'astérisques '*' :



Implémentation en C

```

#include <stdio.h>

main()
{
    /* Prototypes des fonctions appelées par main */
    void RECTANGLE(int L, int H);
    /* Déclaration des variables locales de main */
    int L, H;
    /* Traitements */
    printf("Entrer la longueur (>= 1): ");
    scanf("%d", &L);
    printf("Entrer la hauteur (>= 1): ");
    scanf("%d", &H);
    /* Afficher un rectangle d'étoiles */
    RECTANGLE(L,H);
    return (0);
}
  
```

Pour que la fonction soit exécutable par la machine, il faut encore spécifier la fonction RECTANGLE :

```

void RECTANGLE(int L, int H)
{
    /* Prototypes des fonctions appelées */
  
```

```

void LIGNE(int L);
/* Déclaration des variables locales */
int I;
/* Traitements */
/* Afficher H lignes avec L étoiles */
for (I=0; I<H; I++)
    LIGNE(L);
}

```

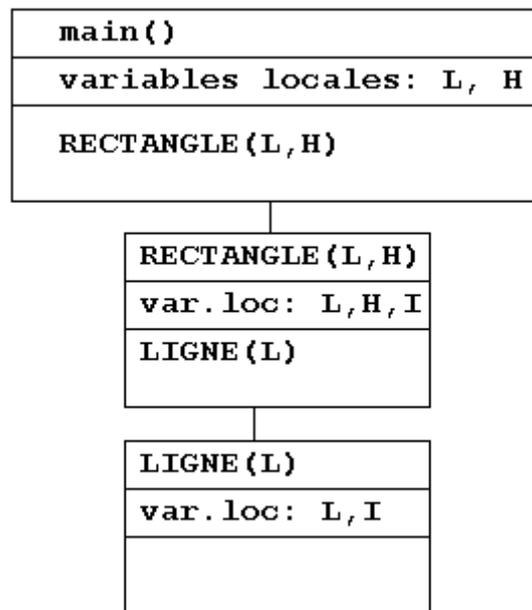
Pour que la fonction RECTANGLE soit exécutable par la machine, il faut spécifier la fonction LIGNE :

```

void LIGNE(int L)
{
/* Affiche à l'écran une ligne avec L étoiles */
/* Déclaration des variables locales */
int I;
/* Traitements */
for (I=0; I<L; I++)
    printf("*");
printf("\n");
}

```

Schématiquement, nous pouvons représenter la hiérarchie des fonctions du programme comme suit :



b) Exemple 2 : Tableau de valeurs d'une fonction :

Soit F la fonction numérique définie par $F(X) = X^3 - 2X + 1$. On désire construire un tableau de valeurs de cette fonction. Le nombre N de valeurs ainsi que les valeurs de X sont entrés au clavier par l'utilisateur.

Exemple :

Entrez un entier entre 1 et 100 : 9

Entrez 9 nombres réels :

-4 -3 -2 -1 0 1 2 3 4

X -4.0 -3.0 -2.0 -1.0 0.0 1.0 2.0 3.0 4.0

F(X) -55.0 -20.0 -3.0 2.0 1.0 0.0 5.0 22.0 57.0

En modularisant ce problème, nous obtenons un programme principal très court et bien 'lisible'. La fonction **main** joue le rôle du programme principal :

```

main()
{
    float X[100];    /* valeurs de X */
    float V[100];    /* valeurs de F(X) */
    int N;
    ACQUERIR(&N);    /* 1 <= N <= 100 */
    LIRE_VECTEUR(X, N);
    CALCULER_VALEURS(X, V, N);
    AFFICHER_TABLE(X, V, N);
    return (0);
}

```

Pour que la machine puisse exécuter ce programme, il faut encore implémenter les modules **ACQUERIR**, **LIRE_VECTEUR**, **CALCULER_VALEURS** et **AFFICHER_TABLE**. Ces spécifications se font en C sous forme de *fonctions* qui remplacent les fonctions *et* les procédures que nous connaissons en langage algorithmique et en Pascal. Une 'procédure' est réalisée en C par une fonction qui fournit le résultat **void** (vide). Les fonctions sont ajoutées dans le texte du programme au-dessus ou en-dessous de la fonction **main**.

Si dans le texte du programme une fonction est définie après la fonction appelante, il faut la **déclarer** ou bien localement à l'intérieur de la fonction appelante ou bien globalement au début du programme. La déclaration d'une fonction se fait à l'aide d'un '**prototype**' de la fonction qui correspond en général à la première ligne (la ligne déclarative) de la fonction.

Par convention, nous allons définir la fonction **main** en premier lieu. Ainsi nous obtenons le programme suivant :

Implémentation en C :

```

#include <stdio.h>

main()
{
    /* Prototypes des fonctions appelées par main */
    void ACQUERIR(int *N);
    void LIRE_VECTEUR(float T[], int N);
    void CALCULER_VALEURS(float X[], float V[], int N);
    void AFFICHER_TABLE(float X[], float V[], int N);
    /* Déclaration des variables locales de main */
    float X[100];    /* valeurs de X */
    float V[100];    /* valeurs de F(X) */
    int N;
    /* Traitements */
    ACQUERIR(&N);    /* 1 <= N <= 100 */
    LIRE_VECTEUR(X, N);
    CALCULER_VALEURS(X, V, N);
    AFFICHER_TABLE(X, V, N);
    return (0);
}

void ACQUERIR(int *N)
{
    do
    {
        printf("Entrez un entier entre 1 et 100 : ");
        scanf("%d", N);
    }
    while (*N<1 || *N>100);
}

```

```

}

void LIRE_VECTEUR(float T[], int N)
{
    /* Remplit un tableau T d'ordre N avec des nombres
       réels entrés au clavier */
    /* Déclaration des variables locales */
    int I;
    /* Remplir le tableau */
    printf("Entrez %d nombres réels :\n", N);
    for (I=0; I<N; I++)
        scanf("%f", &T[I]);
}

void CALCULER_VALEURS(float X[], float V[], int N)
{
    /* Remplit le tableau V avec les valeurs de */
    /* F(X[I]) pour les N premières composantes */
    /* X[I] du tableau X */
    /* Prototype de la fonction F */
    float F(float X);
    /* Déclaration des variables locales */
    int I;
    /* Calculer les N valeurs */
    for (I=0; I<N; I++)
        V[I] = F(X[I]);
}

float F(float X)
{
    /* Retourne la valeur numérique du polynôme défini
       par  $F(X) = X^3 - 2X + 1$  */
    return (X*X*X - 2*X + 1);
}

void AFFICHER_TABLE(float X[], float V[], int N)
{
    /* Affiche une table de N valeurs :
       X contient les valeurs données et
       V contient les valeurs calculées. */
    /* Déclaration des variables locales */
    int I;
    /* Afficher le tableau */
    printf("\n X   : ");
    for (I=0; I<N; I++)
        printf("%.1f", X[I]);
    printf("\n F(X): ");
    for (I=0; I<N; I++)
        printf("%.1f", V[I]);
    printf("\n");
}

```

Le programme est composé de six fonctions dont quatre ne fournissent pas de résultat. La fonction F retourne la valeur de F(X) comme résultat. Le résultat de F est donc du type **float**; nous disons alors que 'F est du type *float*' ou 'F a le type *float*'.

Les fonctions fournissent leurs résultats à l'aide de la commande **return**. La valeur rendue à l'aide de **return** doit correspondre au type de la fonction, sinon elle est automatiquement convertie dans ce type.

A la fin de l'exécution du programme, la fonction **main** fournit par défaut une valeur comme code d'erreur à l'environnement. Le retour de la valeur zéro veut dire que le programme s'est terminé normalement et sans erreurs fatales.

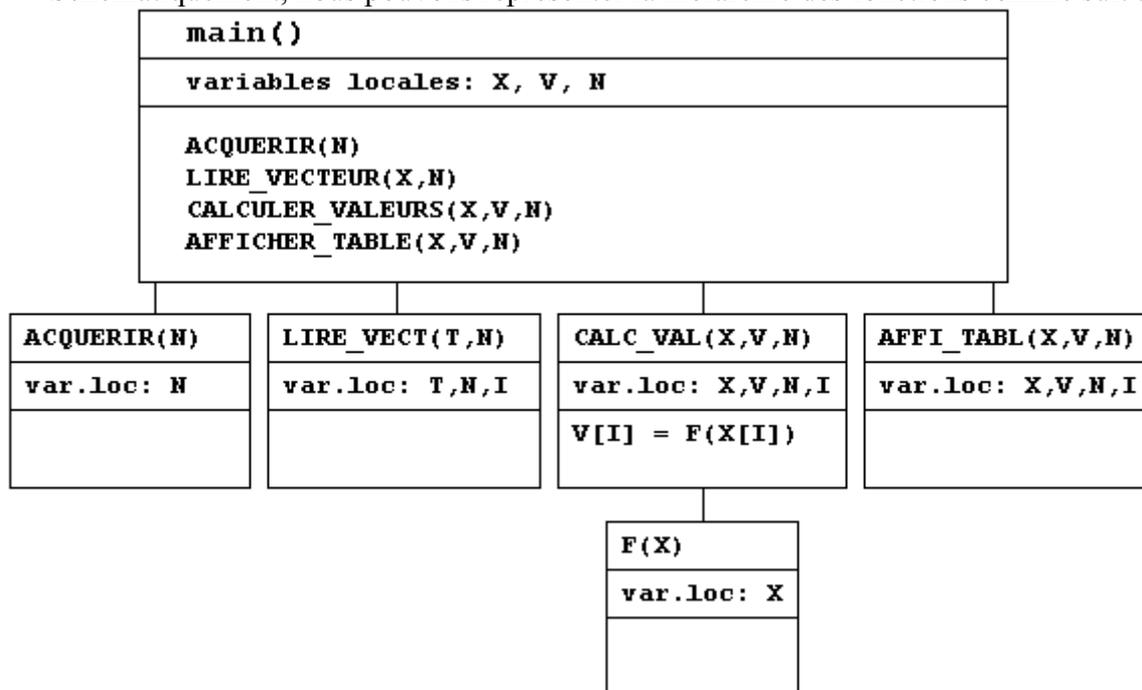
Le passage des paramètres en C se fait toujours par la valeur. Pour pouvoir modifier une variable déclarée dans la procédure appelante, la fonction appelée a besoin de l'adresse de cette variable. Le paramètre correspondant doit donc être un pointeur et lors d'un appel de la fonction, il faut veiller à envoyer l'adresse et non la valeur de la variable.

Dans notre exemple, la fonction ACQUERIR a besoin de l'adresse de la variable N pour pouvoir affecter une nouvelle valeur à N. Le paramètre N doit donc être défini comme pointeur (sur **int**). Lors de l'appel, il faut transmettre l'adresse de N par &N. A l'intérieur de la fonction il faut utiliser l'opérateur 'contenu de' pour accéder à la valeur de N. Les autres fonctions ne changent pas le contenu de N et ont seulement besoin de sa valeur. Dans les en-têtes de ces fonctions, N est simplement déclaré comme **int**.

Lorsque nous passons un tableau comme paramètre à une fonction, il ne faut pas utiliser l'opérateur adresse & lors de l'appel, parce que le nom du tableau représente déjà l'adresse du tableau.

Dans notre exemple, la fonction LIRE_VECTEUR modifie le contenu de la variable X, mais lors de l'appel, il suffit d'envoyer le nom du tableau comme paramètre.

Schématiquement, nous pouvons représenter la hiérarchie des fonctions comme suit :



II) La notion de blocs et la portée des identificateurs :

Les fonctions en C sont définies à l'aide de blocs d'instructions. Un bloc d'instructions est encadré d'accolades et composé de deux parties :

Blocs d'instructions en C :

```

{
  <déclarations locales>
  <instructions>
}
  
```

Par opposition à d'autres langages de programmation, ceci est vrai pour **tous** les blocs d'instructions, non seulement pour les blocs qui renferment une fonction. Ainsi, le bloc d'instructions d'une

commande **if**, **while** ou **for** peut théoriquement contenir des déclarations locales de variables et même de fonctions.

Exemple :

La variable d'aide I est déclarée à l'intérieur d'un bloc conditionnel. Si la condition (N>0) n'est pas remplie, I n'est pas défini. A la fin du bloc conditionnel, I disparaît.

```
if (N>0)
{
    int I;
    for (I=0; I<N; I++)
        ...
}
```

1) Variables locales :

Les variables déclarées dans un bloc d'instructions sont *uniquement visibles à l'intérieur de ce bloc*. On dit que ce sont des **variables locales** à ce bloc.

Exemple :

La variable NOM est définie localement dans le bloc extérieur de la fonction HELLO. Ainsi, aucune autre fonction n'a accès à la variable NOM :

```
void HELLO(void);
{
    char NOM[20];
    printf("Introduisez votre nom : ");
    gets(NOM);
    printf("Bonjour %s !\n", NOM);
}
```

Exemple :

La déclaration de la variable I se trouve à l'intérieur d'un bloc d'instructions conditionnel. Elle n'est pas visible à l'extérieur de ce bloc, ni même dans la fonction qui l'entoure.

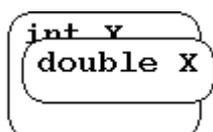
```
if (N>0)
{
    int I;
    for (I=0; I<N; I++)
        ...
}
```

Attention !

Une variable déclarée à l'intérieur d'un bloc **cache** toutes les variables du même nom des blocs qui l'entourent.

Exemple :

Dans la fonction suivante,



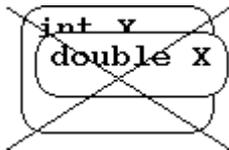
```
int FONCTION(int A);
{
    int X;
    ...
    X = 100;
```

```

...
while (A>10)
{
    double X;
    ...
    X *= A;
    ...
}
}

```

la première instruction **X=100** se rapporte à la variable du type **int** déclarée dans le bloc extérieur de la fonction; l'instruction **X*=A** agit sur la variable du type **double** déclarée dans la boucle **while**. A l'intérieur de la boucle, il est impossible d'accéder à la variable X du bloc extérieur.



Ce n'est pas du bon style d'utiliser des noms de variables qui cachent des variables déclarées dans des blocs extérieurs; ceci peut facilement mener à des malentendus et à des erreurs.

La plupart des programmes C ne profitent pas de la possibilité de déclarer des variables ou des fonctions à l'intérieur d'une boucle ou d'un bloc conditionnel. Dans la suite, nous allons faire toutes nos déclarations locales *au début des fonctions*.

2) Variables globales :

Les variables déclarées au début du fichier, à l'extérieur de toutes les fonctions *sont disponibles à toutes les fonctions du programme*. Ce sont alors des **variables globales**. En général, les variables globales sont déclarées immédiatement derrière les instructions **#include** au début du programme.

Attention !

*Les variables déclarées au début de la fonction principale **main** ne sont pas des variables globales, mais elles sont locales à **main** !*

Exemple :

La variable STATUS est déclarée globalement pour pouvoir être utilisée dans les procédures A et B.

```

#include <stdio.h>
int STATUS;

void A(...)
{
    ...
    if (STATUS>0)
        STATUS--;
    else
        ...
    ...
}

void B(...)
{
    ...
    STATUS++;
}

```

```
...
}
```

Conseils :

- * Les variables globales sont à utiliser avec précaution, puisqu'elles créent des *liens invisibles entre les fonctions*. La modularité d'un programme peut en souffrir et le programmeur risque de perdre la vue d'ensemble.
- * Il faut faire attention à ne pas cacher involontairement des variables globales par des variables locales du même nom.
- * Le codex de la programmation défensive nous conseille *d'écrire nos programmes aussi 'localement' que possible*.



L'utilisation de variables globales devient inévitable, si :

- * plusieurs fonctions qui ne s'appellent pas ont besoin des mêmes variables, ou
- * plusieurs fonctions d'un programme ont besoin du même ensemble de variables. Ce serait alors trop encombrant de passer toutes les variables comme paramètres d'une fonction à l'autre.

III) Déclaration et définition de fonctions :

En général, le nom d'une fonction apparaît à trois endroits dans un programme :

- lors de la **déclaration**
- lors de la **définition**
- lors de l'**appel**

Exemple :

Avant de parler des détails, penchons-nous sur un exemple. Dans le programme suivant, la fonction **main** utilise les deux fonctions :

- **ENTREE** qui lit un nombre entier au clavier et le fournit comme résultat. La fonction ENTREE n'a pas de paramètres.
- **MAX** qui renvoie comme résultat le maximum de deux entiers fournis comme paramètres.

```
#include <stdio.h>

main()
{
    /* Prototypes des fonctions appelées */
    int ENTREE(void);
    int MAX(int N1, int N2);
    /* Déclaration des variables */
    int A, B;
    /* Traitement avec appel des fonctions */
    A = ENTREE();
    B = ENTREE();
    printf("Le maximum est %d\n", MAX(A,B));
}

/* Définition de la fonction ENTREE */
int ENTREE(void)
{
    int NOMBRE;
    printf("Entrez un nombre entier : ");
    scanf("%d", &NOMBRE);
    return NOMBRE;
}
```

```

/* Définition de la fonction MAX */
int MAX(int N1, int N2)
{
  if (N1>N2)
    return N1;
  else
    return N2;
}

```

1) Définition d'une fonction :

Dans la définition d'une fonction, nous indiquons :

- le nom de la fonction
- le type, le nombre et les noms des paramètres de la fonction
- le type du résultat fourni par la fonction
- les données locales à la fonction
- les instructions à exécuter

Définition d'une fonction en langage algorithmique :

```

fonction <NomFonct> (<NomPar1>, <NomPar2>, ...) :<TypeRés>
|
| <déclarations des paramètres>
| <déclarations locales>
| <instructions>
ffonction

```

Définition d'une fonction en C :

```

<TypeRés> <NomFonct> (<TypePar1> <NomPar1>,
                    <TypePar2> <NomPar2>, ... )
{
  <déclarations locales>
  <instructions>
}

```

Remarquez qu'il n'y a pas de point virgule derrière la définition des paramètres de la fonction.

Les identificateurs :

Les noms des paramètres et de la fonction sont des identificateurs qui doivent correspondre aux restrictions définies dans chapitre 2.2.4. Des noms bien choisis peuvent fournir une information utile sur leur rôle. Ainsi, les identificateurs font aussi partie de la documentation d'un programme. (La définition et le rôle des différents types de paramètres dans une fonction seront discutés au chapitre 10.5. "Paramètres d'une fonction".)

Attention !

Si nous choisissons un nom de fonction qui existe déjà dans une bibliothèque, notre fonction cache la fonction prédéfinie.

Type d'une fonction :

Si une fonction F fournit un résultat du type T, on dit que 'la fonction F est du type T' ou que 'la fonction F a le type T'.

Exemple :

La fonction MAX est du type **int** et elle a besoin de deux paramètres du type **int**. Le résultat de la fonction MAX peut être intégré dans d'autres expressions.

```

int MAX(int N1, int N2)
{
    if (N1>N2)
        return N1;
    else
        return N2;
}

```

Exemple :

La fonction PI fournit un résultat rationnel du type **float**. La liste des paramètres de PI est déclarée comme **void** (vide), c'est-à-dire PI n'a pas besoin de paramètres et il faut l'appeler par : **PI()**

```

float PI(void)
{
    return 3.1415927;
}

```

Remarques :

- Une fonction peut fournir comme résultat :
 - - un type arithmétique,
 - - une structure (définie par **struct** - pas traité dans ce cours),
 - - une réunion (définie par **union** - pas traité dans ce cours),
 - - un pointeur,
 - - **void** (la fonction correspond alors à une 'procédure').

Une fonction **ne peut pas** fournir comme résultat des tableaux, des chaînes de caractères ou des fonctions. (**Attention** : Il est cependant possible de renvoyer un pointeur sur le premier élément d'un tableau ou d'une chaîne de caractères.)
- Si une fonction ne fournit pas de résultat, il faut indiquer **void** (vide) comme type du résultat.
- Si une fonction n'a pas de paramètres, on peut déclarer la liste des paramètres comme (**void**) ou simplement comme **()**.
- Le type par défaut est **int**; autrement dit : si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type **int**.
- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).
- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée ou définie avant d'être appelée (voir aussi 10.3.3.)

Rappel : main

La fonction principale **main** est du type **int**. Elle est exécutée automatiquement lors de l'appel du programme. A la place de la définition :

```
int main(void)
```

on peut écrire simplement :

```
main()
```

2) Déclaration d'une fonction :

En C, il faut déclarer chaque fonction avant de pouvoir l'utiliser. La déclaration informe le compilateur du type des paramètres et du résultat de la fonction. A l'aide de ces données, le compilateur peut contrôler si le nombre et le type des paramètres d'une fonction sont corrects. Si dans le texte du programme la fonction est définie avant son premier appel, elle n'a pas besoin d'être déclarée.

Prototype d'une fonction :

La déclaration d'une fonction se fait par un **prototype** de la fonction qui indique uniquement le type des données transmises et reçues par la fonction.

Déclaration : Prototype d'une fonction

```
<TypeRés> <NomFonct> (<TypePar1>, <TypePar2>, ...);  
ou bien  
<TypeRés> <NomFonct> (<TypePar1> <NomPar1>,  
                        <TypePar2> <NomPar2>, ... );
```

Attention !

Lors de la *déclaration*, le nombre et le type des paramètres doivent nécessairement correspondre à ceux de la *définition* de la fonction.

Noms des paramètres

On peut facultativement inclure les *noms des paramètres* dans la déclaration, mais ils ne sont pas considérés par le compilateur. Les noms fournissent pourtant une information intéressante pour le programmeur qui peut en déduire le rôle des différents paramètres.

Conseil pratique

Il est d'usage de copier (à l'aide de Edit - Copy & Paste) la première ligne de la définition de la fonction comme déclaration. (N'oubliez pas d'ajouter un point virgule à la fin de la déclaration !)

Règles pour la déclaration des fonctions :

De façon analogue aux déclarations de variables, nous pouvons déclarer une fonction localement ou globalement. La définition des fonctions joue un rôle spécial pour la déclaration. En résumé, nous allons considérer les règles suivantes :

Déclaration locale :

Une fonction peut être déclarée localement *dans la fonction qui l'appelle* (avant la déclaration des variables). Elle est alors disponible à cette fonction.

Déclaration globale :

Une fonction peut être déclarée globalement *au début du programme* (derrière les instructions **#include**). Elle est alors disponible à toutes les fonctions du programme.

Déclaration implicite par la définition :

La fonction est automatiquement disponible à toutes les fonctions qui suivent sa définition.

Déclaration multiple :

Une fonction peut être déclarée *plusieurs fois* dans le texte d'un programme, mais les indications doivent concorder.

Main :

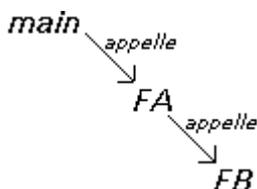
La fonction principale **main** n'a pas besoin d'être déclarée.

3) Discussion d'un exemple :

Considérons la situation suivante :

- * La fonction **main** appelle la fonction FA.
- * La fonction FA appelle la fonction FB.

Nous obtenons donc la hiérarchie suivante :



Il y a beaucoup de possibilités de déclarer et de définir ces fonctions. Nous allons retenir trois variantes qui suivent une logique conséquente :

a) Déclarations locales des fonctions et définition 'top-down' :

La définition 'top-down' suit la hiérarchie des fonctions :

Nous commençons par la définition de la fonction principale **main**, suivie des sous-programmes FA et FB. Nous devons déclarer explicitement FA et FB car leurs définitions suivent leurs appels.

```
/* Définition de main */
main()
{
    /* Déclaration locale de FA */
    int FA (int X, int Y);
    ...
    /* Appel de FA */
    I = FA(2, 3);
    ...
}

/* Définition de FA */
int FA(int X, int Y)
{
    /* Déclaration locale de FB */
    int FB (int N, int M);
    ...
    /* Appel de FB */
    J = FB(20, 30);
    ...
}

/* Définition de FB */
int FB(int N, int M)
{
    ...
}
```

Cet ordre de définition a l'avantage de refléter la hiérarchie des fonctions : Ainsi l'utilisateur qui ne s'intéresse qu'à la solution globale du problème n'a qu'à lire le début du fichier. Pour retrouver les détails d'une implémentation, il peut passer du haut vers le bas dans le fichier. Sur ce chemin, il retrouve toutes les dépendances des fonctions simplement en se référant aux déclarations locales. S'il existe beaucoup de dépendances dans un programme, le nombre des déclarations locales peut quand même s'accroître dans des dimensions insoutenables.

b) Définition 'bottom-up' sans déclarations :

La définition 'bottom-up' commence en bas de la hiérarchie :

La fonction **main** se trouve à la fin du fichier. Les fonctions qui traitent les détails du problème sont définies en premier lieu.

```
/* Définition de FB */
int FB(int N, int M)
{
    ...
}

/* Définition de FA */
int FA(int X, int Y)
```

```

{
    ...
    /* Appel de FB */
    J = FB(20, 30);
    ...
}

/* Définition de main */
main()
{
    ...
    /* Appel de FA */
    I = FA(2, 3);
    ...
}

```

Comme les fonctions sont définies avant leur appel, les déclarations peuvent être laissées de côté. Ceci allège un peu le texte du programme, mais il est beaucoup plus difficile de retrouver les dépendances entre les fonctions.

c) Déclaration globale des fonctions et définition 'top-down' :

En déclarant toutes les fonctions globalement au début du texte du programme, nous ne sommes pas forcés de nous occuper de la dépendance entre les fonctions. Cette solution est la plus simple et la plus sûre pour des programmes complexes contenant une grande quantité de dépendances. Il est quand même recommandé de définir les fonctions selon l'ordre de leur hiérarchie :

```

/* Déclaration globale de FA et FB */
int FA (int X, int Y);
int FB (int N, int M);

/* Définition de main */
main()
{
    ...
    /* Appel de FA */
    I = FA(2, 3);
    ...
}

/* Définition de FA */
int FA(int X, int Y)
{
    ...
    /* Appel de FB */
    J = FB(20, 30);
    ...
}

/* Définition de FB */
int FB(int N, int M)
{
    ...
}

```

d) Conclusions :

Dans la suite, nous allons utiliser l'ordre de définition 'top-down' qui reflète le mieux la structure d'un programme. Comme nos programmes ne contiennent pas beaucoup de dépendances, nous allons déclarer les fonctions localement dans les fonctions appelantes.

-
- [Exercice 10.1](#)
-

IV) Renvoyer un résultat :

Par définition, toutes les fonctions fournissent un résultat d'un type que nous devons déclarer. Une fonction peut renvoyer une valeur d'un type simple ou l'adresse d'une variable ou d'un tableau.

Pour fournir un résultat en quittant une fonction, nous disposons de la commande **return** :

1) La commande return :

```
return <expression>;
```

L'instruction a les effets suivants :

- *évaluation de l'<expression>*
- *conversion automatique du résultat de l'expression dans le type de la fonction*
- *renvoi du résultat*
- *terminaison de la fonction*

Exemples :

La fonction CARRE du type **double** calcule et fournit comme résultat le carré d'un réel fourni comme paramètre.

```
double CARRE(double X)  
{  
  return X*X;  
}
```

Nous pouvons définir nous-mêmes une fonction TAN qui calcule la tangente d'un réel X à l'aide des fonctions **sin** et de **cos** de la bibliothèque *<math>*. En langage algorithmique cette fonction se présente comme suit :

```
fonction TAN(X) : réel  
  | donnée : réel X  
  | si (cos(X) <> 0)  
  |   | alors en TAN ranger sin(X)/cos(X)  
  |   | sinon écrire "Erreur !"  
  | fsi  
ffonction (* fin TAN *)
```

En C, il faut d'abord inclure le fichier en-tête de la bibliothèque *<math>* pour pouvoir utiliser les fonctions prédéfinies **sin** et **cos**.

```
#include <math.h>  
  
double TAN(double X)  
{  
  if (cos(X) != 0)  
    return sin(X)/cos(X);  
  else  
    printf("Erreur !\n");  
}
```

Si nous supposons les déclarations suivantes,

```
double X, COT;
```

les appels des fonctions CARRE et TAN peuvent être intégrés dans des calculs ou des expressions :

```
printf("Le carré de %f est %f \n", X, CARRE(X));
printf("La tangente de %f est %f \n", X, TAN(X));
COT = 1/TAN(X);
```

2) void :

En C, il n'existe pas de structure spéciale pour la définition de *procédures* comme en Pascal et en langage algorithmique. Nous pouvons cependant employer une fonction du type **void** partout où nous utiliserions une procédure en langage algorithmique ou en Pascal.

Exemple :

La procédure LIGNE affiche L étoiles dans une ligne :

```
procédure LIGNE(L)
| donnée L
| (* Déclarations des variables locales *)
| entier I
| (* Traitements *)
| en I ranger 0
| tant que I<>L faire
| | écrire "*"
| | en I ranger I+1
| ftant (* I=L *)
| écrire (* passage à la ligne *)
fprocédure
```

Pour la traduction en C, nous utilisons une fonction du type **void** :

```
void LIGNE(int L)
{
    /* Déclarations des variables locales */
    int I;
    /* Traitements */
    for (I=0; I<L; I++)
        printf("*");
    printf("\n");
}
```

3) main :

Dans nos exemples, la fonction **main** n'a pas de paramètres et est toujours du type **int** (Voir aussi [Chap 2.2.2. Remarque avancée](#))

Typiquement, les programmes renvoient la valeur zéro comme code d'erreur s'ils se terminent avec succès. Des valeurs différentes de zéro indiquent un arrêt fautif ou anormal.

En MS-DOS, le code d'erreur retourné par un programme peut être contrôlé à l'aide de la commande IF ERRORLEVEL ...

Remarque avancée :

Si nous quittons une fonction (d'un type différent de **void**) sans renvoyer de résultat à l'aide de **return**, la valeur transmise à la fonction appelante est indéfinie. Le résultat d'une telle action est imprévisible. Si une erreur fatale s'est produite à l'intérieur d'une fonction, il est conseillé d'interrompre l'exécution de tout le programme et de renvoyer un code erreur différent de zéro à l'environnement pour indiquer que le programme ne s'est pas terminé normalement.

Vu sous cet angle, il est dangereux de déclarer la fonction TAN comme nous l'avons fait plus haut : Le cas d'une division par zéro, est bien intercepté et reporté par un message d'erreur, mais l'exécution du programme continue 'normalement' avec des valeurs incorrectes.

4) exit :

Pour remédier à ce dilemme, nous pouvons utiliser la fonction **exit** qui est définie dans la bibliothèque `<stdlib>`. **exit** nous permet d'interrompre l'exécution du programme en fournissant un code d'erreur à l'environnement. Pour pouvoir localiser l'erreur à l'intérieur du programme, il est avantageux d'afficher un message d'erreur qui indique la nature de l'erreur et la fonction dans laquelle elle s'est produite.

Une version plus solide de TAN se présenterait comme suit :

```
#include <math.h>

double TAN(double X)
{
    if (cos(X) != 0)
        return sin(X)/cos(X);
    else
    {
        printf("\aFonction TAN :\n"
            "Erreur : Division par zéro !\n");
        exit(-1); /* Code erreur -1 */
    }
}
```

5) Ignorer le résultat :

Lors de l'appel d'une fonction, l'utilisateur est libre d'accepter le résultat d'une fonction ou de l'ignorer.

Exemple :

La fonction **scanf** renvoie le nombre de données correctement reçues comme résultat. En général, nous avons ignoré ce fait :

```
int JOUR, MOIS, ANNEE;
printf("Entrez la date actuelle : ");
scanf("%d %d %d", &JOUR, &MOIS, &ANNEE);
```

Nous pouvons utiliser le résultat de **scanf** comme contrôle :

```
int JOUR, MOIS, ANNEE;
int RES;
do
{
    printf("Entrez la date actuelle : ");
    RES = scanf("%d %d %d", &JOUR,&MOIS,&ANNEE);
}
while (RES != 3);
```

V) Paramètres d'une fonction :

Les *paramètres* ou *arguments* sont les 'boîtes aux lettres' d'une fonction. Elles acceptent les données de l'extérieur et déterminent les actions et le résultat de la fonction. Techniquement, nous pouvons résumer le rôle des paramètres en C de la façon suivante :

*Les paramètres d'une fonction sont simplement des **variables locales** qui sont initialisées par les valeurs obtenues lors de l'appel.*

1) Généralités :

Conversion automatique :

Lors d'un appel, le nombre et l'ordre des paramètres doivent nécessairement correspondre aux indications de la déclaration de la fonction. Les paramètres sont automatiquement convertis dans les types de la déclaration avant d'être passés à la fonction.

Exemple :

Le prototype de la fonction **pow** (bibliothèque *<math></i>) est déclaré comme suit :*

```
double pow (double, double);
```

Au cours des instructions,

```
int A, B;
```

```
...
```

```
A = pow (B, 2);
```

nous assistons à trois conversions automatiques :

Avant d'être transmis à la fonction, la valeur de B est convertie en **double**; la valeur 2 est convertie en 2.0 . Comme **pow** est du type **double**, le résultat de la fonction doit être converti en **int** avant d'être affecté à A.

void :

Evidemment, il existe aussi des fonctions qui fournissent leurs résultats ou exécutent une action sans avoir besoin de données. La liste des paramètres contient alors la déclaration **void** ou elle reste vide (P.ex. : **double PI(void)** ou **int ENTREE()**).

2) Passage des paramètres par valeur :

En C, le passage des paramètres se fait toujours par la valeur, c'est-à-dire les fonctions n'obtiennent que les *valeurs* de leurs paramètres et n'ont pas d'accès aux variables elles-mêmes.

Les paramètres d'une fonction sont à considérer comme des *variables locales* qui sont initialisées automatiquement par les valeurs indiquées lors d'un appel.

A l'intérieur de la fonction, nous pouvons donc changer les valeurs des paramètres sans influencer les valeurs originales dans les fonctions appelantes.

Exemple :

La fonction ETOILES dessine une ligne de N étoiles. Le paramètre N est modifié à l'intérieur de la fonction :

```
void ETOILES(int N)
{
    while (N>0)
    {
        printf("*");
        N--;
    }
    printf("\n");
}
```

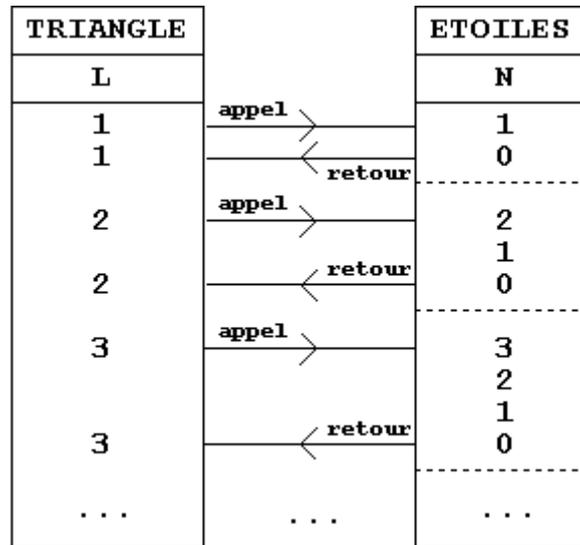
En utilisant N comme compteur, nous n'avons pas besoin de l'indice d'aide I comme dans la fonction LIGNES définie plus haut.

La fonction TRIANGLE, appelle la fonction ETOILES en utilisant la variable L comme paramètre :

```
void TRIANGLE(void)
{
    int L;
    for (L=1; L<10; L++)
        ETOILES(L);
}
```

Au moment de l'appel, la *valeur* de L est copiée dans N. La variable N peut donc être décrétementée à l'intérieur de ETOILES, sans influencer la valeur originale de L.

Schématiquement, le passage des paramètres peut être représenté dans une 'grille' des valeurs :



Avantages

Le passage par *valeur* a l'avantage que nous pouvons utiliser les paramètres comme des variables locales bien initialisées. De cette façon, nous avons besoin de moins de variables d'aide.

3) Passage de l'adresse d'une variable :

Comme nous l'avons constaté ci-dessus, une fonction n'obtient que les valeurs de ses paramètres.

Pour changer la valeur d'une variable de la fonction appelante,
nous allons procéder comme suit :

- la fonction appelante doit *fournir l'adresse de la variable* et
- la fonction appelée doit *déclarer le paramètre comme pointeur*.

On peut alors atteindre la variable à l'aide du pointeur.

Discussion d'un exemple :

Nous voulons écrire une fonction PERMUTER qui échange le contenu de deux variables du type **int**. En première approche, nous écrivons la fonction suivante :

```
void PERMUTER (int A, int B)
{
  int AIDE;
  AIDE = A;
  A = B;
  B = AIDE;
}
```

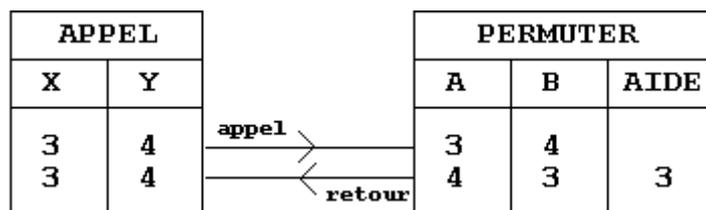


Nous appelons la fonction pour deux variables X et Y par :

```
PERMUTER (X, Y);
```

Résultat : X et Y restent inchangés !

Explication : Lors de l'appel, les *valeurs* de X et de Y sont copiées dans les paramètres A et B. PERMUTER échange bien contenu des variables *locales* A et B, mais les valeurs de X et Y restent les mêmes.



Pour pouvoir modifier le contenu de X et de Y, la fonction PERMUTER a besoin des adresses de X et Y. En utilisant des pointeurs, nous écrivons une deuxième fonction :

```
void PERMUTER (int *A, int *B)
{
  int AIDE;
```



```

AIDE = *A;
*A = *B;
*B = AIDE;
}

```

Nous appelons la fonction par :

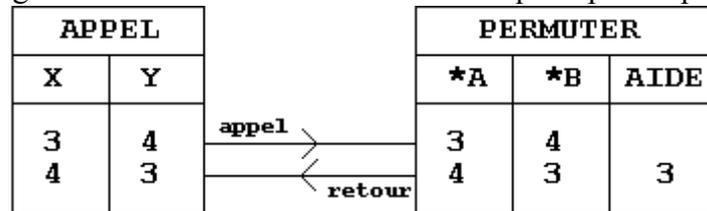
```

PERMUTER (&X, &Y);

```

Résultat : Le contenu des variables X et Y est échangé !

Explication : Lors de l'appel, les *adresses* de X et de Y sont copiées dans les *pointeurs* A et B. PERMUTER échange ensuite le contenu des adresses indiquées par les pointeurs A et B.



Remarque

Dans les grilles, nous allons marquer les paramètres acceptant une adresse comme pointeurs et indiquer le contenu de ces adresses.

4) Passage de l'adresse d'un tableau à une dimension :

a) Méthode :

Comme il est impossible de passer 'la valeur' de tout un tableau à une fonction, on fournit *l'adresse d'un élément du tableau*.

En général, on fournit l'adresse du premier élément du tableau, qui est donnée par *le nom du tableau*.

b) Déclaration :

Dans la liste des paramètres d'une fonction, on peut déclarer un tableau par le nom suivi de crochets,

```

<type> <nom> []

```

ou simplement par un pointeur sur le type des éléments du tableau :

```

<type> *<nom>

```

Exemple :

La fonction **strlen** calcule et retourne la longueur d'une chaîne de caractères fournie comme paramètre :

```

int strlen(char *S)
{
    int N;
    for (N=0; *S != '\0'; S++)
        N++;
    return N;
}

```

A la place de la déclaration de la chaîne comme

```

char *S

```

on aurait aussi pu indiquer

```

char S[]

```

comme nous l'avons fait dans l'exemple d'introduction (chapitre 10.1.2). Dans la suite, nous allons utiliser la première notation pour mettre en évidence que le paramètre est un *pointeur variable* que nous pouvons modifier à l'intérieur de la fonction.

c) Appel :

Lors d'un appel, l'adresse d'un tableau peut être donnée par le nom du tableau, par un pointeur ou par l'adresse d'un élément quelconque du tableau.

Exemple :

Après les instructions,

```
char CH[] = "Bonjour !";  
char *P;  
P = CH;
```

nous pouvons appeler la fonction **strlen** définie ci-dessus par :

```
strlen(CH)           /* résultat : 9 */  
strlen(P)           /* résultat : 9 */  
strlen(&CH[4])      /* résultat : 5 */  
strlen(P+2)         /* résultat : 7 */  
strlen(CH+2)        /* résultat : 7 */
```

! Dans les trois derniers appels, nous voyons qu'il est possible de fournir *une partie* d'un tableau à une fonction, en utilisant l'adresse d'un élément à l'intérieur de tableau comme paramètre.

Remarque pratique :

Pour qu'une fonction puisse travailler correctement avec un tableau qui n'est pas du type **char**, il faut aussi fournir la dimension du tableau ou le nombre d'éléments à traiter comme paramètre, sinon la fonction risque de sortir du domaine du tableau.

Exemple :

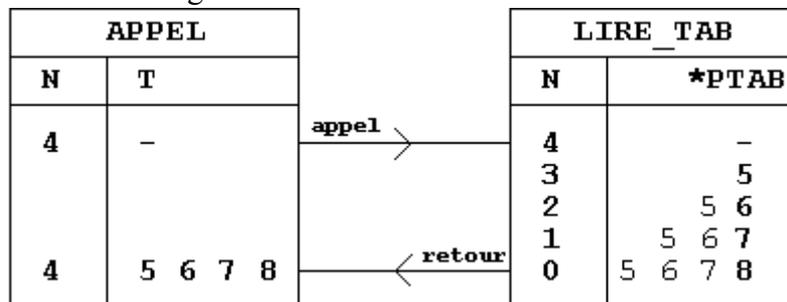
La fonction **LIRETAB** lit N données pour un tableau (unidimensionnel) du type **int** et les mémorise à partir de l'adresse indiquée par le pointeur **PTAB**. **PTAB** et **N** sont fournis comme paramètres.

```
void LIRE_TAB(int N, int *PTAB)  
{  
    printf("Entrez %d valeurs : \n", N);  
    while(N)  
    {  
        scanf("%d", PTAB++);  
        N--  
    }  
}
```

Dans l'appel de la fonction nous utilisons en général le nom du tableau :

```
LIRE_TAB(4, T);
```

Nous obtenons alors les grilles suivantes :



5) Passage de l'adresse d'un tableau à deux dimensions :

Exemple :

Imaginons que nous voulons écrire une fonction qui calcule la somme de tous les éléments d'une matrice de réels A dont nous fournissons les deux dimensions N et M comme paramètres.

Problème :

Comment pouvons-nous passer l'adresse de la matrice à la fonction ?

Par analogie avec ce que nous avons vu au chapitre précédent, nous pourrions envisager de déclarer le tableau concerné dans l'en-tête de la fonction sous la forme `A[][]`. Dans le cas d'un tableau à deux dimensions, cette méthode ne fournit pas assez de données, parce que le compilateur a besoin de la deuxième dimension du tableau pour déterminer l'adresse d'un élément `A[i][j]`.

Une solution praticable consiste à faire en sorte que la fonction reçoive un pointeur (de type `float*`) sur le début de la matrice et de parcourir tous les éléments comme s'il s'agissait d'un tableau à une dimension `N*M`.

Cela nous conduit à cette fonction :

```
float SOMME(float *A, int N, int M)
{
    int I;
    float S;
    for (I=0; I<N*M; I++)
        S += A[I];
    return S;
}
```

Lors d'un appel de cette fonction, la seule difficulté consiste à transmettre l'adresse du début du tableau sous forme d'un pointeur sur `float`. Prenons par exemple un tableau déclaré par

```
float A[3][4];
```

Le nom `A` correspond à la bonne adresse, mais cette adresse est du type "*pointeur sur un tableau de 4 éléments du type float*". Si notre fonction est correctement déclarée, le compilateur la convertira automatiquement dans une adresse du type '*pointeur sur float*'.

Toutefois, comme nous l'avons déjà remarqué au chapitre 9.3.4, on gagne en lisibilité et on évite d'éventuels messages d'avertissement si on utilise l'opérateur de conversion forcée ("*cast*").

Solution :

Voici finalement un programme faisant appel à notre fonction `SOMME` :

```
#include <stdio.h>
main()
{
    /* Prototype de la fonction SOMME */
    float SOMME(float *A, int N, int M);
    /* Déclaration de la matrice */
    float T[3][4] = {{1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9,10,11,12}};
    /* Appel de la fonction SOMME */
    printf("Somme des éléments : %f \n",
           SOMME((float*)T, 3, 4) );
    return (0);
}
```

Rappel :

Rappelons encore une fois que lors de l'interprétation d'un tableau à deux dimensions comme tableau unidimensionnel, il faut calculer les adresses des composantes à l'aide du *nombre de colonnes maximal réservé lors de la déclaration*.

Remarque de Francois Donato :

Une méthode plus propre pour éviter le cast

```
SOMME((float*)T, 3, 4) );
```

est de renvoyer explicitement l'adresse du premier élément du tableau :

```
SOMME(&T[0][0], 3, 4) );
```

6) Les modules en lang. algorithmique, en Pascal et en C :

Ce chapitre résume les différences principales entre les modules (programme principal, fonctions, procédures) dans les différents langages de programmation que nous connaissons.

Modules :

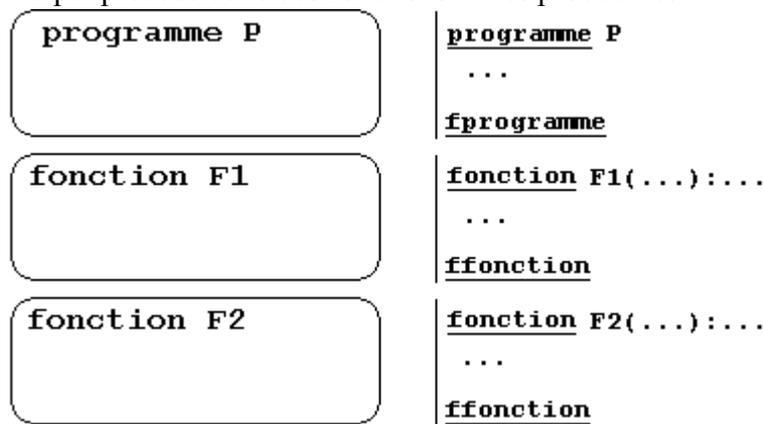
En Pascal et en langage algorithmique, nous distinguons programme principal, procédures et fonctions.

En C, il existe uniquement des fonctions. La fonction principale **main** se distingue des autres fonctions par deux qualités :

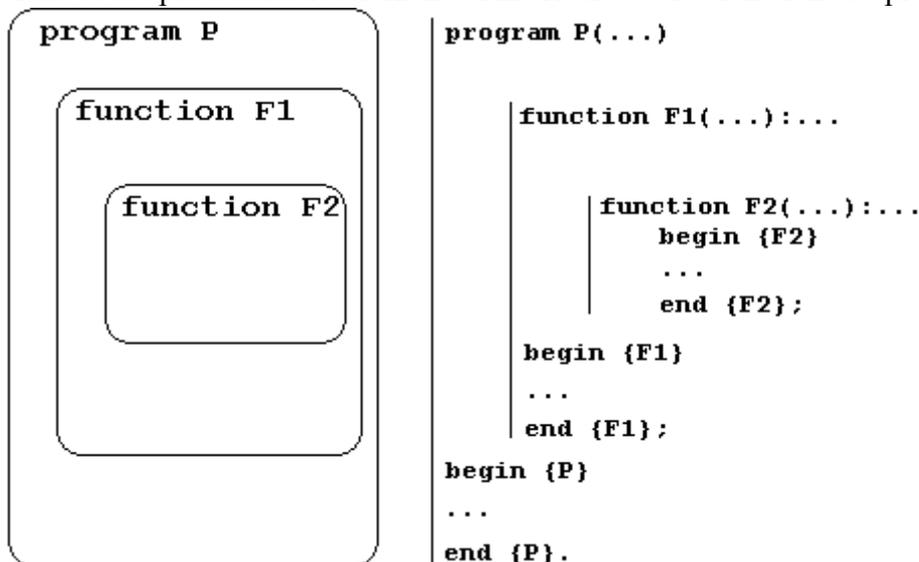
- Elle est exécutée lors de l'appel du programme.
- Les types du résultat (**int**) et des paramètres (**void**) sont fixés.

Définition des modules :

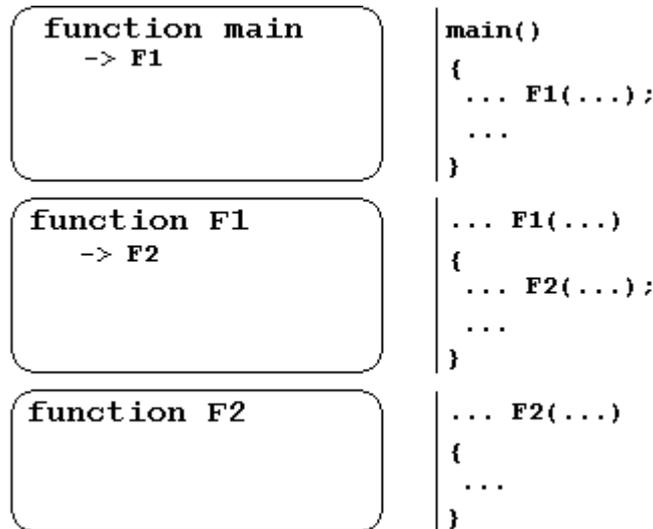
En langage algorithmique, le programme principal, les fonctions et les procédures sont déclarés dans des blocs distincts. Il est interdit d'imbriquer leurs définitions. La définition du programme principal précède celle des fonctions et des procédures.



En Pascal, les définitions des modules peuvent être imbriquées; c'est-à-dire : on peut définir des fonctions et des procédures localement à l'intérieur d'autres fonctions ou procédures.



En C, il est interdit de *définir* des fonctions à l'intérieur d'autres fonctions, mais nous pouvons *déclarer* des fonctions localement.



Variables locales :

En **Pascal et en langage algorithmique**, nous pouvons déclarer des variables locales au début des fonctions et des procédures.

En **C**, il est permis (mais déconseillé) de déclarer des variables locales au début de chaque bloc d'instructions.

Variables globales :

En **Pascal et en langage algorithmique**, les variables globales sont définies au début du programme principal.

En **C**, les variables globales sont définies au début du fichier, à l'extérieur de toutes les fonctions. (Les variables de la fonction principale **main** sont locales à **main**.)

Passage des paramètres :

En **Pascal et en langage algorithmique**, nous distinguons entre passage des paramètres par valeur et passage des paramètres par référence.

En **C**, le passage des paramètres se fait toujours par la valeur. Pour pouvoir changer le contenu d'une variable déclarée dans une autre fonction, il faut utiliser un pointeur comme paramètre de passage et transmettre l'adresse de la variable lors de l'appel.

Exemple comparatif :

La fonction DIVI divise son premier paramètre A par son deuxième paramètre B et fournit le reste de la division entière comme résultat. Le contenu du paramètre A est modifié à l'intérieur de la fonction, le paramètre B reste inchangé. Le programme principal appelle la fonction DIVI avec deux entiers lus au clavier et affiche les résultats.

- Solution du problème en langage algorithmique :

```

programme TEST_DIVI
  entier N,D,R
  écrire "Entrer numérateur et dénominateur : "
  lire N
  lire D
  en R ranger DIVI(N,D)
  écrire "Résultat : ",N," Reste : ",R
fprogramme

fonction DIVI (A, B): entier
  résultat : entier A
  donnée : entier B
  entier C

```

```

    en C ranger A modulo B
    en A ranger A divent B
    en DIVI ranger C
ffonction (* fin DIVI *)

```

- * Le paramètre A est *transféré par référence* : Il est déclaré par le mot-clef résultat au début de la fonction.
- * Le paramètre B est *transféré par valeur* : Il est déclaré par le mot-clef donnée au début de la fonction.
- * Le résultat de la fonction est *affecté au nom* de la fonction. Cette affectation doit se trouver à la fin la fonction.
- * Dans un appel, il n'y a pas de différence entre la notation des paramètres passés par référence et ceux passés par valeur.

- Solution du problème en Pascal :

```

program TEST_DIVI
var N, D, R : integer

    function DIVI (var A : integer; B : integer) :integer;
begin
    DIVI := A mod B;
    A := A div B;
end;

begin
write('Entrer numérateur et dénominateur : ');
read(N);
readln(D);
R := DIVI (N, D);
writeln('Résultat : ',N,' Reste : ',R );
end.

```

- * Le paramètre A est *transféré par référence* : Il est déclaré par le mot-clef **var** dans l'en-tête de la fonction.
- * Le paramètre B est *transféré par valeur* : Il est déclaré sans désignation spéciale dans l'en-tête de la fonction.
- * Le résultat de la fonction est *affecté au nom* de la fonction. Cette affectation peut se trouver n'importe où dans la fonction.
- * Dans un appel, il n'y a pas de différence entre la notation des paramètres passés par référence et ceux passés par valeur.

- Solution du problème en C :

```

#include <stdio.h>

main()
{
    int DIVI(int *A, int B);
    int N, D, R;
printf("Entrer numérateur et dénominateur : ");
scanf("%d %d", &N, &D);
R = DIVI (&N, D);
printf("Résultat : %d Reste : %d\n", N, R);
return (0);
}

int DIVI (int *A, int B)

```

```

{
  int C;
  C = *A % B;
  *A /= B;
  return C;
}

```

- * Le paramètre A *reçoit l'adresse d'une variable* : Il est déclaré comme pointeur sur **int**.
- * Le paramètre B *reçoit la valeur d'une variable* : Il est déclaré comme **int**.
- * Le résultat de la fonction est retourné à l'aide de la commande **return**. Comme l'exécution de la fonction s'arrête après la commande **return**, celle-ci doit se trouver à la fin de la fonction.
- * Dans un appel, le premier paramètre est une adresse. Le nom de la variable N est donc précédé par l'opérateur adresse **&**. Le deuxième paramètre est passé par valeur. Le nom de la variable est indiqué sans désignation spéciale.

Vu de plus près, les trois langages offrent les mêmes mécanismes pour le passage des paramètres, mais :

>> en C nous devons veiller nous-mêmes à opérer avec les adresses et les pointeurs respectifs si nous voulons changer le contenu d'une variable déclarée dans une autre fonction;

>> en langage algorithmique et en Pascal, les mêmes opérations se déroulent derrière les rideaux, sous l'étiquette 'passage par référence' ('call-by-reference').



VI) Discussion de deux problèmes :

1) "fonction" ou "procédure" ? :

Problème 1 :

Ecrire une fonction qui lit un nombre entier au clavier en affichant un petit texte d'invite.

Réflexion :

Avant d'attaquer le problème, nous devons nous poser la question, de quelle façon nous allons transférer la valeur lue dans la variable de la fonction appelante. Il se présente alors deux possibilités :

- Nous pouvons fournir la valeur comme résultat de la fonction.
- Nous pouvons affecter la valeur à une adresse que nous obtenons comme paramètre. Dans ce cas, le résultat de la fonction est **void**. Cette fonction est en fait une "procédure" au sens du langage Pascal et du langage algorithmique.

a) Résultat int ==> "fonction" :

Reprenons d'abord la fonction ENTREE que nous avons définie au chapitre 10.3. :

```

int ENTREE(void)
{
  int NOMBRE;
  printf("Entrez un nombre entier : ");
  scanf("%d", &NOMBRE);
  return NOMBRE;
}

```

La fonction ENTREE fournit un résultat du type **int** qui est typiquement affecté à une variable,

```

int A;
A = ENTREE();

```

ou intégré dans un calcul :

```

long SOMME;
int I;
for (I=0; I<10; I++)
    SOMME += ENTREE();

```

b) Résultat void ==> "procédure" :

Nous pouvons obtenir le même effet que ENTREE en définissant une fonction ENTRER du type **void**, qui affecte la valeur lue au clavier immédiatement à une adresse fournie comme paramètre. Pour accepter cette adresse, le paramètre de la fonction doit être déclaré comme pointeur :

```

void ENTRER(int *NOMBRE)
{
    printf("Entrez un nombre entier : ");
    scanf("%d", NOMBRE);
}

```

Remarquez : Comme le paramètre NOMBRE est un pointeur, il n'a pas besoin d'être précédé du symbole **&** dans l'instruction **scanf**.

Lors de l'appel, nous devons transférer l'adresse de la variable cible comme paramètre :

```

int A;
ENTRER(&A);

```

Jusqu'ici, la définition et l'emploi de la fonction ENTRER peuvent sembler plus simples que ceux de la fonction ENTREE. Si nous essayons d'intégrer les valeurs lues par ENTRER dans un calcul, nous allons quand même constater que ce n'est pas toujours le cas :

```

long SOMME;
int I;
int A;
for (I=0; I<10; I++)
{
    ENTRER(&A);
    SOMME += A;
}

```

Conclusions :

Dans la plupart des cas, nous pouvons remplacer une fonction qui fournit un résultat par une fonction du type **void** qui modifie le contenu d'une variable de la fonction appelante. Dans le langage Pascal, nous dirions que nous pouvons remplacer une *fonction* par une *procédure*. En général, la préférence pour l'une ou l'autre variante dépend de l'utilisation de la fonction :

Si le résultat de la fonction est typiquement intégré dans un calcul ou une expression, alors nous employons une fonction qui fournit un résultat. En fait, personne ne remplacerait une fonction comme

```
double sin(double X)
```

par une fonction

```
void sin(double *Y, double X)
```

- * Si la charge principale d'une fonction est de modifier des données ou l'état de l'environnement, sans que l'on ait besoin d'un résultat, alors il vaut mieux utiliser une fonction du type **void**.
- * Si une fonction doit fournir plusieurs valeurs comme résultat, il s'impose d'utiliser une procédure du type **void**. Ce ne serait pas une bonne solution de fournir une valeur comme résultat et de transmettre les autres valeurs comme paramètres.

Exemple :

La fonction MAXMIN fournit le maximum et le minimum des valeurs d'un tableau T de N d'entiers. Comme nous ne pouvons pas fournir les deux valeurs comme résultat, il vaut mieux utiliser deux paramètres pointeurs MAX et MIN qui obtiendront les adresses cibles pour les deux valeurs. Ainsi la fonction MAXMIN est définie avec quatre paramètres :

```

void MAXMIN(int N, int *T, int *MAX, int *MIN);
{
  int I;
  *MAX=*T;
  *MIN=*T;
  for (I=1; I<N; I++)
    {
      if (*(T+I)>*MAX) *MAX = *(T+I);
      if (*(T+I)<*MIN) *MIN = *(T+I);
    }
}

```

Lors d'un appel de MAXMIN, il ne faut pas oublier d'envoyer les *adresses* des paramètres pour MAX et MIN.

```

int TAB[8] = {2,5,-1,0,6,9,-4,6};
int N = 8;
int MAXVAL, MINVAL;
MAXMIN(N, TAB, &MAXVAL, &MINVAL);

```

2) Pointeur ou indice numérique ? :

Problème 2 :

Ecrire une fonction qui fournit la position du prochain signe d'espace dans une chaîne de caractères ou la position de la fin de la chaîne si elle ne contient pas de signe d'espace. Utiliser la fonction `isspace` de la bibliothèque `<ctype>` pour la recherche.

Réflexion :

Il y a plusieurs possibilités de résoudre ce problème : Une "indication" d'une position dans une chaîne de caractères peut être fournie par un *pointeur* ou par un *indice numérique*. Dans la suite, nous allons développer les deux variations.

Dans les deux cas, nous avons besoin de l'adresse du tableau qui est passée comme paramètre `char *CH` à la fonction.

a) Résultat int :

La fonction `CHERCHE1` fournit l'indice de l'élément recherché comme résultat du type `int`.

```

int CHERCHE1(char *CH)
{
  int INDEX=0;
  while (*CH && !isspace(*CH))
    {
      CH++;
      INDEX++;
    }
  return INDEX;
}

```

Cette information peut être affectée à une variable :

```

int I;
char TXT[40];
...
I = CHERCHE1(TXT);

```

ou être intégrée dans une expression en profitant même de l'arithmétique des pointeurs :

```

main()
{
  /* Prototype de la fonction appelée */
  int CHERCHE1(char *CH);
}

```

```

char TXT[40];
printf("Entrer une phrase : ");
gets(TXT);
/* Affichage de la phrase sans le premier mot */
puts(TXT + CHERCHE1(TXT));
return (0);
}

```

b) Résultat char* :

La fonction CHERCHE2 fournit un pointeur sur l'élément recherché. Remarquez la déclaration du résultat de la fonction CHERCHE2 comme pointeur sur **char** :

```

char *CHERCHE2(char *CH)
{
    while (*CH && !isspace(*CH))
        CH++;
    return CH;
}

```

Il se montre à nouveau que l'utilisation de pointeurs permet une solution très compacte et élégante. Dans cette version, nous n'avons plus besoin de variables d'aide et nous pouvons renvoyer la valeur modifiée du paramètre local CH comme résultat. L'utilisation de la fonction peut se présenter de façon aussi élégante :

```

main()
{
    /* Prototype de la fonction appelée */
    char *CHERCHE2(char *CH);
    char TXT[40];
    printf("Entrer une phrase : ");
    gets(TXT);
    /* Affichage de la phrase sans le premier mot */
    puts(CHERCHE2(TXT));
    return (0);
}

```

Conclusion :

Lors du travail avec des tableaux et surtout avec des chaînes de caractères, il est toujours avantageux d'utiliser des pointeurs et de profiter de l'arithmétique des pointeurs. Les fonctions employant des pointeurs lors du traitement de tableaux permettent souvent des solutions très naturelles, d'autant plus qu'elles disposent des adresses des tableaux dans des paramètres locaux.

VII) Exercices d'application :

Exercice 10.1 :

Soient les fonctions **main**, **PI** et **SURFACE** définies par :

```
#include <stdio.h>

main()
{
    double R;
    printf("Introduire le rayon du cercle : ");
    scanf("%lf", &R);
    printf("La surface du cercle est %f. \n", SURFACE(R));
    return (0);
}

double PI(void)
{
    return 3.14159265;
}

double SURFACE(double RAYON)
{
    return PI() *RAYON*RAYON;
}
```

- Etablir la hiérarchie des appels pour les trois fonctions.
 - Définir les fonctions d'après les trois méthodes décrites ci-dessus en ajoutant les déclarations manquantes.
 - Quels messages fournit le compilateur lorsqu'une fonction n'est pas définie ou déclarée avant son appel ? Comment peut-on expliquer ces messages ?
-

1) Passage des paramètres :

Exercice 10.2 :

Exécuter le programme suivant et construire les grilles correspondantes. Implémenter le programme ensuite en C.

```
programme PARAMETRES
|   entier A,B
|   en A ranger 0
|   en B ranger 0
|   P(A,B)
|   écrire A,B
fprogramme (* fin PARAMETRES *)
procédure P(X,Y)
|   donnée : entier X
|   résultat : entier Y
|   en X ranger X+1
|   en Y ranger Y+1
|   écrire X,Y
fprocédure (* fin P *)
```

Exercice 10.3 :

Exécuter le programme suivant et construire les grilles correspondantes. Implémenter le programme ensuite en C.

```
programme TRUC
|   entier A
|   en A ranger 2
|   écrire A
|   MACHIN(A)
|   écrire A
fprogramme (* fin TRUC *)
procédure MACHIN(X)
|   donnée : entier X
|   écrire X
|   en X ranger 1000
|   écrire X
fprocédure (* fin MACHIN *)
```

Exercice 10.4 :

Exécuter le programme suivant et construire les grilles correspondantes. Implémenter le programme ensuite en C.

```
programme CALCUL
|   entier A,B,C
|   en A ranger 3
|   en B ranger -8
|   en C ranger 12
|   écrire A,B,C
|   MODIFIER(A,B,C)
|   écrire A,B,C
fprogramme (* fin CALCUL *)

procédure MODIFIER(X,Y,Z)
|   donnée : entier X
|   résultat : entier Y,Z
|   entier T
|   en T ranger X
|   en X ranger Y
|   en Y ranger Z
|   en Z ranger T
fprocédure (* fin MODIFIER *)
```

Exercice 10.5 :

Exécuter le programme suivant et construire les grilles correspondantes. Implémenter le programme ensuite en C.

```
programme MANIPULATION
|   entier A,B,C
|   en A ranger 208
|   en B ranger 5
|   en C ranger -34
|   écrire A,B,C
|   MANIPULER(A,B,C)
|   écrire A,B,C
fprogramme (* fin MANIPULATION *)
```

```

procédure MANIPULER (X, Y, Z)
|   donnée : entier X, Y
|   résultat : entier Z
|   écrire X, Y, Z
|   en X ranger X divent 2
|   en Y ranger Y*2
|   en Z ranger X+Y
|   écrire X, Y, Z
fprocédure (* fin MANIPULER *)

```

2) Types simples :

Exercice 10.6 :

Ecrire un programme se servant d'une fonction MOYENNE du type **float** pour afficher la moyenne arithmétique de deux nombres réels entrés au clavier.

Exercice 10.7 :

Ecrire deux fonctions qui calculent la valeur X^N pour une valeur réelle X (type **double**) et une valeur entière positive N (type **int**) :

- a) EXP1 retourne la valeur X^N comme résultat.
- b) EXP2 affecte la valeur X^N à X.

Ecrire un programme qui teste les deux fonctions à l'aide de valeurs lues au clavier.

Exercice 10.8 :

Ecrire une fonction MIN et une fonction MAX qui déterminent le minimum et le maximum de deux nombres réels.

Ecrire un programme se servant des fonctions MIN et MAX pour déterminer le minimum et le maximum de quatre nombres réels entrés au clavier.

Exercice 10.9 :

Ecrire un programme se servant d'une fonction F pour afficher la table de valeurs de la fonction définie par

$$f(x) = \sin(x) + \ln(x) - \sqrt{x}$$

où x est un entier compris entre 1 et 10.

Exercice 10.10 :

Ecrire la fonction NCHIFFRES du type **int** qui obtient une valeur entière N (positive ou négative) du type **long** comme paramètre et qui fournit le nombre de chiffres de N comme résultat.

Ecrire un petit programme qui teste la fonction NCHIFFRES :

Exemple :

```

Introduire un nombre entier : 6457392
Le nombre 6457392 a 7 chiffres.

```

Exercice 10.11 :

En mathématiques, on définit la fonction factorielle de la manière suivante

$$0! = 1$$

$$n! = n*(n-1)*(n-2)* \dots * 1 \text{ (pour } n > 0)$$

Ecrire une fonction FACT du type **double** qui reçoit la valeur N (type **int**) comme paramètre et qui fournit la factorielle de N comme résultat. Ecrire un petit programme qui teste la fonction FACT.

Exercice 10.12 :

Ecrire un programme qui construit et affiche le triangle de Pascal en calculant les coefficients binomiaux :

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
. . .
```

On n'utilisera pas de tableau, c'est-à-dire il faudra calculer les coefficients d'après la formule ci-dessous, tout en définissant et utilisant les fonctions adéquates.

$$C_p^q = \frac{p!}{q! \cdot (p - q)!}$$

3) Tableaux à une dimension :

Exercice 10.13 :

La fonction LIRE_TAB à trois paramètres TAB, N et NMAX lit la dimension N et les composantes d'un tableau TAB du type **int**. La dimension N doit être inférieure à NMAX. Implémenter la fonction LIRE_TAB en choisissant bien le type des paramètres.

Exemple :

Pour un appel par

```
LIRE_TAB(T, &N, 10);
```

la fonction se comportera comme suit :

```
Dimension du tableau (max.10) : 11
Dimension du tableau (max.10) : 4
Elément [0] : 43
Elément [1] : 55
Elément [2] : 67
Elément [3] : 79
```

Exercice 10.14 :

Ecrire la fonction ECRIRE_TAB à deux paramètres TAB et N qui affiche N composantes du tableau TAB du type **int**.

Exemple :

Le tableau T lu dans l'exemple ci-dessus sera affiché par l'appel :

```
ECRIRE_TAB(T, N);
```

et sera présenté comme suit :

```
43 55 67 79
```

Exercice 10.15 :

Ecrire la fonction SOMME_TAB qui calcule la somme des N éléments d'un tableau TAB du type **int**. N et TAB sont fournis comme paramètres; la somme est retournée comme résultat du type **long**.

Exercice 10.16 :

A l'aide des fonctions des exercices précédents, écrire un programme qui lit un tableau A d'une dimension inférieure ou égale à 100 et affiche le tableau et la somme des éléments du tableau.

4) Tris de tableaux :

Exercice 10.17 Tri de Shell :

Traduire la fonction TRI_SHELL définie ci-dessous en C. Utiliser la fonction PERMUTER définie dans le cours.

Ecrire un programme profitant des fonctions définies dans les exercices précédents pour tester la fonction TRI_SHELL.

```
procédure TRI_SHELL(T,N)
  (* Trie un tableau T d'ordre N par la méthode
    de Shell en ordre croissant. *)
  résultat : entier tableau T[100]
  donnée : entier N
  entier SAUT, M, K
  booléen TERMINE
  en SAUT ranger N
  tant que (SAUT>1) faire
    en SAUT ranger SAUT divent 2
    répéter
      en TERMINE ranger vrai
      pour M variant de 1 à N-SAUT faire
        en K ranger M+SAUT
        si (T[M]>T[K]) alors
          PERMUTER(T[M],T[K])
          en TERMINE ranger faux
        fsi
      fpour
    jusqu'à TERMINE
  ftant (* SAUT <= 1 *)
fprocédure (* fin TRI_SHELL *)
```

Remarque :

L'algorithme a été développé par D.L.Shell en 1959. En comparant d'abord des éléments très éloignés, l'algorithme a tendance à éliminer rapidement les grandes perturbations dans l'ordre des éléments. La distance entre les éléments qui sont comparés est peu à peu réduite jusqu'à 1. A la fin du tri, les éléments voisins sont arrangés.

Exercice 10.18 :

Déterminer le maximum de N éléments d'un tableau TAB d'entiers de trois façons différentes :

- a) la fonction MAX1 retourne la valeur maximale
- b) la fonction MAX2 retourne l'indice de l'élément maximal
- c) la fonction MAX3 retourne l'adresse de l'élément maximal

Ecrire un programme pour tester les trois fonctions.

Exercice 10.19 : Tri par sélection :

Ecrire la fonction TRI_SELECTION qui trie un tableau de N entiers par la méthode de sélection directe du maximum (voir exercice 7.14). La fonction fera appel à la fonction PERMUTER (définie dans le cours) et à la fonction MAX3 (définie dans l'exercice précédent).

Ecrire un programme pour tester la fonction TRI_SELECTION.

Exercice 10.20 :

Ecrire la fonction INSERER qui place un élément X à l'intérieur d'un tableau qui contient N éléments triés par ordre croissant, de façon à obtenir un tableau à N+1 éléments triés par ordre croissant. La dimension du tableau est incrémentée dans la fonction INSERER.

Ecrire un programme profitant des fonctions définies plus haut pour tester la fonction INSERER.

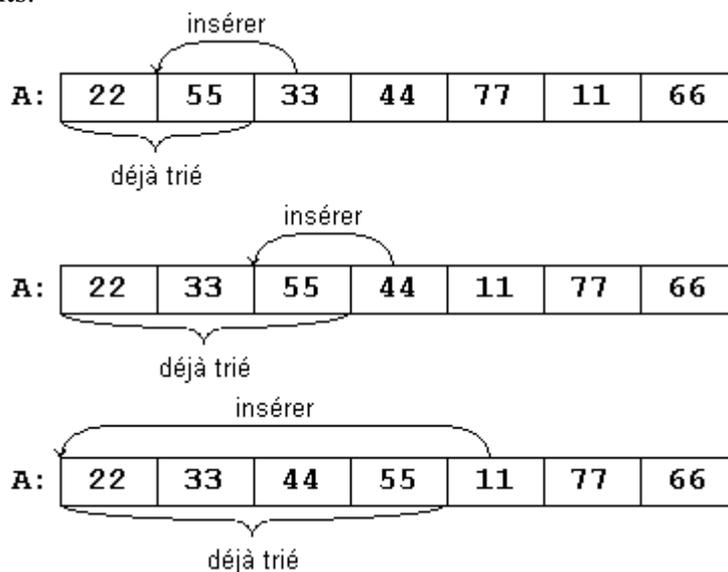
Exercice 10.21 : Tri par insertion :

Ecrire la fonction TRI_INSERTION qui utilise la fonction INSERER pour trier par ordre croissant les éléments d'un tableau à N éléments.

Ecrire un programme pour tester la fonction TRI_INSERTION.

Méthode :

Trier le tableau de gauche à droite en insérant à chaque fois l'élément I+1 dans le tableau (déjà trié) des I premiers éléments.



Exercice 10.22 :

Ecrire la fonction RANGER qui arrange le contenu de ses deux paramètres X et Y de façon à ce que le contenu de X soit plus petit que celui de Y. RANGER retourne la valeur logique 1 si un échange a eu lieu, sinon 0.

Exercice 10.23 : Tri par propagation :

Ecrire la fonction TRI_BULLE qui trie un tableau de N éléments entiers par ordre croissant en appliquant la méthode de la bulle (tri par propagation - voir exercice 7.15). Employer la fonction RANGER de l'exercice ci-dessus.

Ecrire un programme pour tester la fonction TRI_BULLE.

Exercice 10.24 : Fusion de tableaux triés :

Ecrire la fonction FUSION qui construit un tableau FUS trié par ordre croissant avec les éléments de deux tableaux A et B triés par ordre croissant. Pour deux tableaux de dimensions N et M, le tableau FUS aura la dimension N+M. (Méthode : voir exercice 7.13)

Ecrire un programme qui teste la fonction FUSION à l'aide de deux tableaux lus au clavier et triés à l'aide de TRI_BULLE.

5) Chaînes de caractères :

Exercice 10.25 :

Ecrire la fonction LONG_CH qui retourne la longueur d'une chaîne de caractères CH comme résultat. Implémentez LONG_CH sans utiliser de variable d'aide numérique.

Exercice 10.26 :

Ecrire la fonction MAJ_CH qui convertit toutes les lettres d'une chaîne en majuscules, sans utiliser de variable d'aide.

Exercice 10.27 :

Ecrire la fonction AJOUTE_CH à deux paramètres CH1 et CH2 qui copie la chaîne de caractères CH2 à la fin de la chaîne CH1 sans utiliser de variable d'aide.

Exercice 10.28 :

Ecrire la fonction INVERSER_CH qui inverse l'ordre des caractères d'une chaîne CH. Utiliser la fonction LONG_CH (définie plus haut) et définir une fonction d'aide PERMUTER_CH qui échange les valeurs de deux variables du type **char**.

Exercice 10.29 :

Ecrire la fonction NMOTS_CH qui retourne comme résultat le nombre de mots contenus dans une chaîne de caractères CH. Utiliser une variable logique, la fonction **isspace** et une variable d'aide N.

Exercice 10.30 :

Ecrire la fonction MOT_CH qui retourne un pointeur sur le N-ième mot d'une chaîne CH s'il existe, sinon un pointeur sur le symbole de fin de chaîne.

Exercice 10.31 :

Ecrire la fonction EGAL_N_CH qui retourne la valeur 1 si les N premiers caractères de CH1 et CH2 sont égaux, sinon la valeur 0. (Si N est plus grand que la longueur de CH1 ou de CH2, le résultat peut être 1 ou 0).

Exercice 10.32 :

Utiliser la fonction EGAL_N_CH et LONG_CH pour écrire la fonction CHERCHE_CH qui retourne un pointeur sur la première apparition de CH1 dans CH2, sinon un pointeur sur le symbole de fin de chaîne.

Exercice 10.33 :

Ecrire la fonction CH_ENTIER qui retourne la valeur numérique d'une chaîne de caractères représentant un entier (positif ou négatif) du type **long**. Si la chaîne ne représente pas une valeur entière correcte, la fonction arrête la conversion et fournit la valeur qu'elle a su reconnaître jusqu'à ce point.

Exercice 10.34 :

Ecrire la fonction CH_DOUBLE qui retourne la valeur numérique d'une chaîne de caractères représentant un réel en *notation décimale*. Si la chaîne ne représente pas une valeur décimale correcte, la fonction arrête la conversion et fournit la valeur qu'elle a su reconnaître jusqu'à ce point. (Méthode : voir exercice 8.18.)

Exercice 10.35 :

Ecrire la fonction ENTIER_CH qui construit une chaîne de caractères CH qui représente un nombre entier N du type **long**. N et CH sont les paramètres de la fonction ENTIER_CH. Utiliser la fonction INVERSER_CH définie plus haut.

Exercice 10.36 :

Ecrire la fonction DOUBLE_CH qui construit une chaîne de caractères CH qui représente un nombre réel N avec 4 positions derrière la virgule. N et CH sont les paramètres de la fonction DOUBLE_CH.

Idée pour la conversion : Multiplier N par 10^4 et utiliser ENTIER_CH.

Représenter schématiquement la hiérarchie des appels des fonctions utilisées.

6) Tableaux à deux dimensions :

Exercice 10.37 :

- a) Ecrire la fonction LIRE_DIM à quatre paramètres L, LMAX, C, CMAX qui lit les dimensions L et C d'une matrice à deux dimensions. Les dimensions L et C doivent être inférieures à LMAX respectivement CMAX.
- b) Ecrire la fonction LIRE_MATRICE à quatre paramètres MAT, L, C, et CMAX qui lit les composantes d'une matrice MAT du type **int** et de dimensions L et C.

Implémenter les fonctions en choisissant bien le type des paramètres et utiliser un dialogue semblable à celui de LIRE_TAB.

Exercice 10.38 :

Ecrire la fonction ECRIRE_MATRICE à quatre paramètres MAT, L, C et CMAX qui affiche les composantes de la matrice de dimensions L et C.

Exercice 10.39 :

Ecrire la fonction SOMME_MATRICE du type **long** qui calcule la somme des éléments d'une matrice MAT du type **int**. Choisir les paramètres nécessaires. Ecrire un petit programme qui teste la fonction SOMME_MATRICE.

Exercice 10.40 :

Ecrire la fonction ADDITION_MATRICE qui effectue l'addition des matrices suivante :

$$\text{MAT1} = \text{MAT1} + \text{MAT2}$$

Choisir les paramètres nécessaires et écrire un petit programme qui teste la fonction `ADDITION_MATRICE`.

Exercice 10.41 :

Écrire la fonction `MULTI_MATRICE` qui effectue la multiplication de la matrice `MAT1` par un entier `X` :

$$\text{MAT1} = X * \text{MAT1}$$

Choisir les paramètres nécessaires et écrire un petit programme qui teste la fonction `MULTI_MATRICE`.

Exercice 10.42 :

Écrire la fonction `TRANSPO_MATRICE` à cinq paramètres `MAT`, `L`, `LMAX`, `C`, `CMAX` qui effectue la transposition de la matrice `MAT` en utilisant la fonction `PERMUTER`. `TRANSPO_MATRICE` retourne une valeur logique qui indique si les dimensions de la matrice sont telles que la transposition a pu être effectuée. Écrire un petit programme qui teste la fonction `TRANSPO_MATRICE`.

Exercice 10.43 :

Écrire la fonction `MULTI_2_MATRICES` qui effectue la multiplication de deux matrices `MAT1` (dimensions `N` et `M`) et `MAT2` (dimensions `M` et `P`) en une troisième matrice `MAT3` (dimensions `N` et `P`) :

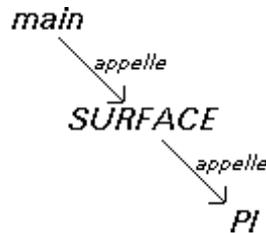
$$\text{MAT3} = \text{MAT1} * \text{MAT2}$$

Supposez que les dimensions maximales des trois matrices soient toutes égales à 30 lignes et 30 colonnes. Écrire un petit programme qui teste la fonction `MULTI_2_MATRICES`. (Méthode de calcul : voir exercice 7.22.)

VIII) Solutions des exercices du Chapitre 10 : LES FONCTIONS :

Exercice 10.1 :

a) Etablir la hiérarchie des appels pour les trois fonctions :



b) Définir les fonctions d'après les trois méthodes décrites ci-dessus en ajoutant les déclarations manquantes.

suite ...

- Déclarations locales des fonctions et définition 'top-down' :

```
#include <stdio.h>

main()
{
    /* Déclaration locale de SURFACE */
    double SURFACE(double RAYON);
    double R;
    printf("Introduire le rayon du cercle : ");
    scanf("%lf", &R);
    printf("La surface du cercle est %f. \n", SURFACE(R));
    return (0);
}

double SURFACE(double RAYON)
{
    /* Déclaration locale de PI */
    double PI(void);
    return PI()*RAYON*RAYON;
}

double PI(void)
{
    return 3.14159265;
}
```

- Définition 'bottom-up' sans déclarations :

```
#include <stdio.h>

double PI(void)
{
    return 3.14159265;
}

double SURFACE(double RAYON)
{
    return PI()*RAYON*RAYON;
}
```

```

main()
{
    double R;
    printf("Introduire le rayon du cercle : ");
    scanf("%lf", &R);
    printf("La surface du cercle est %f. \n", SURFACE(R));
    return (0);
}

```

- Déclaration globale des fonctions et définition 'top-down':

```

#include <stdio.h>
/* Déclaration globale des fonctions */
double SURFACE(double RAYON);
double PI(void);

main()
{
    double R;
    printf("Introduire le rayon du cercle : ");
    scanf("%lf", &R);
    printf("La surface du cercle est %f. \n", SURFACE(R));
    return (0);
}

double SURFACE(double RAYON)
{
    return PI()*RAYON*RAYON;
}

double PI(void)
{
    return 3.14159265;
}

```

c)

Si nous compilons le programme donné sans changements, nous obtenons les messages suivants :

```

Warning ... : Call to function 'SURFACE' with no prototype
Error ... : Type mismatch in redeclaration of 'SURFACE'

```

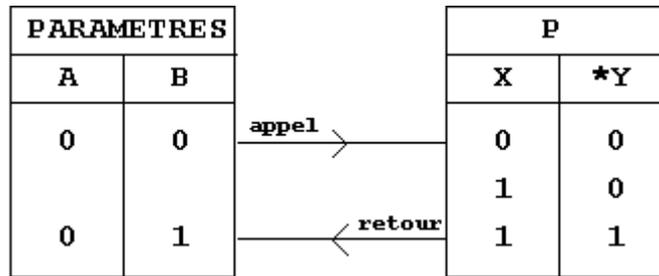
Explications :

Le premier message est un avertissement qui nous informe que la fonction SURFACE est appelée sans être déclarée auparavant. Le compilateur suppose alors par défaut que la fonction fournit *un résultat du type int* et que *les paramètres ont le type des arguments utilisés lors de l'appel*. Cet avertissement restera sans conséquences si ces suppositions sont correctes.

En rencontrant ensuite la définition de la fonction, le compilateur détecte une incompatibilité de type, car la fonction SURFACE retourne en réalité une valeur du type **double**. Cette incompatibilité de type ("**Type mismatch**") est signalée par un message d'erreur et le programme ne peut pas être compilé avec succès.

1) Passage des paramètres :

Exercice 10.2 :



Affichage : 1 1
 0 1

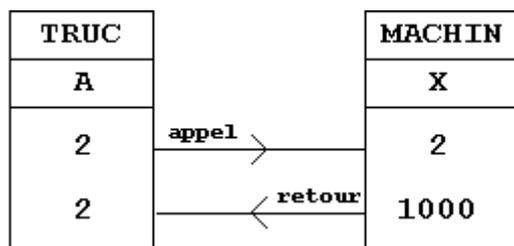
Implémentation :

```
#include <stdio.h>

main()
{
  void P(int X, int *Y); /* Prototype de la fonction appelée */
  int A,B;
  A=0;
  B=0;
  P(A, &B);
  printf("%d %d \n", A, B);
  return (0);
}

void P(int X, int *Y)
{
  X = X+1;
  *Y = *Y+1;
  printf("%d %d \n", X, *Y);
}
```

Exercice 10.3 :



Affichage :

2
2
1000
2

Implémentation :

```
#include <stdio.h>

main()
{
  void MACHIN(int X); /* Prototype de la fonction appelée */
```

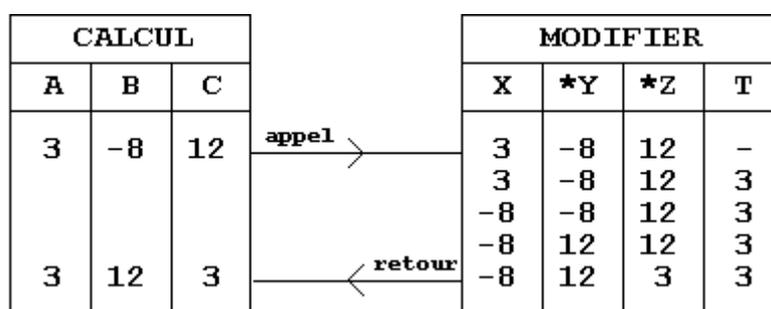
```

int A;
A=2;
printf("%d \n", A);
MACHIN(A);
printf("%d \n", A);
return (0);
}

void MACHIN(int X)
{
printf("%d \n", X);
X = 1000;
printf("%d \n", X);
}

```

Exercice 10.4 :



Affichage :

```

3 -8 12
3 12 3

```

Implémentation :

```

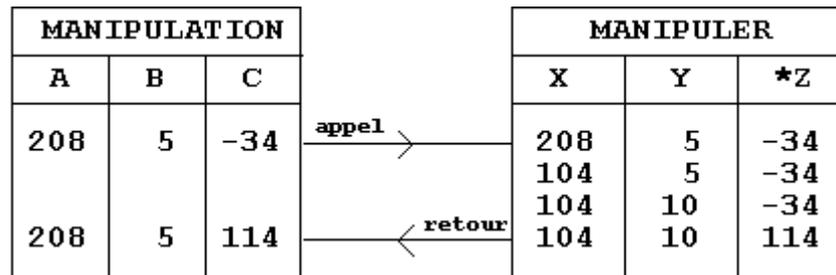
#include <stdio.h>

main()
{
void MODIFIER(int X, int *Y, int *Z); /* Prototype */
int A,B,C;
A=3;
B=-8;
C=12;
printf("%d %d %d \n", A, B, C);
MODIFIER(A,&B,&C);
printf("%d %d %d \n", A, B, C);
return (0);
}

void MODIFIER(int X, int *Y, int *Z)
{
int T;
T = X;
X = *Y;
*Y = *Z;
*Z = T;
}

```

Exercice 10.5 :



Affichage :

208	5	-34
208	5	-34
104	10	114
208	5	114

suite ...

Implémentation :

```
#include <stdio.h>

main()
{
    void MANIPULER(int X, int Y, int *Z); /* Prototype */
    int A,B,C;
    A=208;
    B=5;
    C=-34;
    printf("%d %d %d \n", A, B, C);
    MANIPULER(A,B,&C);
    printf("%d %d %d \n", A, B, C);
    return (0);
}

void MANIPULER(int X, int Y, int *Z)
{
    printf("%d %d %d \n", X, Y, *Z);
    X = X/2;
    Y = Y*2;
    *Z = X+Y;
    printf("%d %d %d \n", X, Y, *Z);
}
```

2) Types simples :

Exercice 10.6 :

```
#include <stdio.h>

main()
{
    /* Prototypes des fonctions appelées */
    float MOYENNE(float X, float Y);
    /* Variables locales */
    float A,B;
```

```

/* Traitements */
printf("Introduire deux nombres : ");
scanf("%f %f", &A, &B);
printf("La moyenne arithmétique de %.2f et %.2f est %.4f\n",
      A, B,
      MOYENNE(A,B));
return (0);
}

float MOYENNE(float X, float Y)
{
return (X+Y)/2;
}

```

Exercice 10.7 :

```

#include <stdio.h>

main()
{
/* Prototypes des fonctions appelées */
double EXP1(double X, int N);
void EXP2(double *X, int N);
/* Variables locales */
double A;
int B;
/* Traitements */
printf("Introduire un réel X : ");
scanf("%lf", &A);
printf("Introduire l'exposant positif N : ");
scanf("%d", &B);
/* a */
printf("EXP1(%.2f , %d) = %f\n", A, B, EXP1(A,B));
/* b */
/* Comme la valeur initiale de A est perdue à l'appel */
/* de EXP2, il faut partager l'affichage si on veut */
/* afficher la valeur de A avant et après l'appel ! */
printf("EXP2(%.2f , %d) = ", A, B);
EXP2(&A, B);
printf("%f\n", A);
return (0);
}

double EXP1(double X, int N)
{
double RES;
for (RES=1.0; N>0; N--)
RES *= X;
return RES;
}

void EXP2(double *X, int N)
{
double AIDE;

```

```

    for (AIDE=1.0; N>0; N--)
        AIDE *= *X;
    *X = AIDE;
}

```

Remarque :

Cette solution de EXP2 respecte automatiquement le cas où N=0.

Exercice 10.8 :

```

#include <stdio.h>

main()
{
    /* Prototypes des fonctions appelées */
    double MIN(double X, double Y);
    double MAX(double X, double Y);
    /* Variables locales */
    double A,B,C,D;
    /* Traitements */
    printf("Introduire 4 réels : ");
    scanf("%lf %lf %lf %lf", &A, &B, &C, &D);
    printf("Le minimum des 4 réels est %f \n",
           MIN( MIN(A,B), MIN(C,D) )
);
    printf("Le maximum des 4 réels est %f \n",
           MAX( MAX(A,B), MAX(C,D) )
);
    return (0);
}

double MIN(double X, double Y)
{
    if (X<Y)
        return X;
    else
        return Y;
}

double MAX(double X, double Y)
{
    if (X>Y)
        return X;
    else
        return Y;
}

/* ou bien */
/*
double MIN(double X, double Y)
{
    return (X<Y) ? X : Y;
}

```

```

double MAX(double X, double Y)
{
    return (X>Y) ? X : Y;
}
*/

```

Exercice 10.9 :

```

#include <stdio.h>
#include <math.h>
main()
{
    /* Prototypes des fonctions appelées */
    double F(int X);
    /* Variables locales */
    int I;
    /* Traitements */
    printf("\tX\tF(X)\n");
    for (I=1 ; I<=10 ; I++)
        printf("\t%d\t%f\n", I, F(I));
    return (0);
}

double F(int X)
{
    return sin(X)+log(X)-sqrt(X);
}

```

Exercice 10.10 :

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    int NCHIFFRES(long N);
    /* Variables locales */
    long A;
    /* Traitements */
    printf("Introduire un nombre entier : ");
    scanf("%ld", &A);
    printf("Le nombre %ld a %d chiffres.\n",A ,NCHIFFRES(A));
    return (0);
}

int NCHIFFRES(long N)
{
    /* Comme N est transmis par valeur, N peut être */
    /* modifié à l'intérieur de la fonction. */
    int I;
    /* Conversion du signe si N est négatif */
    if (N<0)
        N *= -1;
    /* Compter les chiffres */

```

```

    for (I=1; N>10; I++)
        N /= 10;
    return I;
}

```

Exercice 10.11 :

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    double FACT(int N);
    /* Variables locales */
    int N;
    /* Traitements */
    printf("Introduire un nombre entier N : ");
    scanf("%d", &N);
    printf("La factorielle de %d est %.0f \n",N , FACT(N));
    return (0);
}

double FACT(int N)
{
    /* Comme N est transmis par valeur, N peut être */
    /* modifié à l'intérieur de la fonction. */
    double RES;
    for (RES=1.0 ; N>0 ; N--)
        RES *= N;
    return RES;
}

```

Exercice 10.12 :

Il y a beaucoup de solutions possibles à ce problème.
Voici probablement la solution la plus modulaire.

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    void TRIANGLE(int LIGNES);
    /* Variables locales */
    int N;
    /* Traitements */
    printf("Introduire le nombre de lignes N : ");
    scanf("%d", &N);
    TRIANGLE(N);
    return (0);
}

void TRIANGLE(int LIGNES)
{
    /* Prototypes des fonctions appelées */

```

```

void LIGNEC(int P);
/* Variables locales */
int P;
/* Traitements */
for (P=0; P<LIGNES; P++)
    LIGNEC(P);
}

void LIGNEC(int P)
{
    /* Prototypes des fonctions appelées */
    double C(int P, int Q);
    /* Variables locales */
    int Q;
    /* Traitements */
    for (Q=0; Q<=P; Q++)
        printf("%6.0f", C(P,Q));
    printf("\n");
}

double C(int P, int Q)
{
    /* Prototypes des fonctions appelées */
    double FACT(int N);
    /* Traitements */
    return FACT(P)/(FACT(Q)*FACT(P-Q));
}

double FACT(int N)
{
    . . .
}

```

3) Tableaux à une dimension :

Exercice 10.13 :

```

void LIRE_TAB (int *TAB, int *N, int NMAX)
{
    /* Variables locales */
    int I;
    /* Saisie de la dimension du tableau */
    do
    {
        printf("Dimension du tableau (max.%d) : ", NMAX);
        scanf("%d", N); /* Attention : écrire N et non &N ! */
    }
    while (*N<0 || *N>NMAX);
    /* Saisie des composantes du tableau */
    for (I=0; I<*N; I++)
    {
        printf("Elément [%d] : ", I);
        scanf("%d", TAB+I);
    }
}

```

```
    }  
}
```

Exercice 10.14 :

```
void ECRIRE_TAB (int *TAB, int N)  
{  
    /* Affichage des composantes du tableau */  
    while(N)  
    {  
        printf("%d ", *TAB);  
        TAB++;  
        N--;  
    }  
    printf("\n");  
}
```

Exercice 10.15 :

```
long SOMME_TAB(int *TAB, int N)  
{  
    /* Variables locales */  
    long SOMME = 0;  
    /* Calcul de la somme */  
    while(N)  
    {  
        SOMME += *TAB;  
        TAB++;  
        N--;  
    }  
    return SOMME;  
}
```

Exercice 10.16 :

```
#include <stdio.h>  
main()  
{  
    /* Prototypes des fonctions appelées */  
    void LIRE_TAB (int *TAB, int *N, int NMAX);  
    void ECRIRE_TAB (int *TAB, int N);  
    long SOMME_TAB(int *TAB, int N);  
    /* Variables locales */  
    int T[100]; /* Tableau d'entiers */  
    int DIM;    /* Dimension du tableau */  
    /* Traitements */  
    LIRE_TAB (T, &DIM, 100);  
    printf("Tableau donné : \n");  
    ECRIRE_TAB (T, DIM);  
    printf("Somme des éléments du tableau : %ld\n",  
          SOMME_TAB(T, DIM))  
};  
return (0);
```

```

}

void LIRE_TAB (int *TAB, int *N, int NMAX)
{
. . .
}

void ECRIRE_TAB (int *TAB, int N)
{
. . .
}

long SOMME_TAB(int *TAB, int N)
{
. . .
}

```

4) Tris de tableaux :

Exercice 10.17 :

Tri de Shell :

```

#include <stdio.h>
main()
{
/* Prototypes des fonctions appelées */
void TRI_SHELL(int *T, int N);
void LIRE_TAB (int *TAB, int *N, int NMAX);
void ECRIRE_TAB (int *TAB, int N);
/* Variables locales */
int T[100]; /* Tableau d'entiers */
int DIM; /* Dimension du tableau */

/* Traitements */
LIRE_TAB (T, &DIM, 100);
printf("Tableau donné : \n");
ECRIRE_TAB (T, DIM);
TRI_SHELL(T, DIM);
printf("Tableau trié : \n");
ECRIRE_TAB (T, DIM);
return (0);
}

void TRI_SHELL(int *T, int N)
{
/* Trie un tableau T d'ordre N par la méthode de Shell */
/* Prototypes des fonctions appelées */
void PERMUTER(int *A, int *B);
/* Variables locales */
int SAUT, M, K;
int TERMINE;
/* Traitements */

```

```

SAUT = N;
while (SAUT>1)
{
    SAUT /= 2;
    do
    {
        TERMINE=1;
        for (M=0; M<N-SAUT; M++) /* Attention aux indices!
*/
            {
                K=M+SAUT;
                if (*(T+M) > *(T+K))
                {
                    PERMUTER(T+M, T+K);
                    TERMINE=0;
                }
            }
        while(!TERMINE); /* Attention : utiliser la négation de
*/
    } /* la condition employée en lang algorithmique
*/
}

void PERMUTER(int *A, int *B)
{
    int AIDE;
    AIDE = *A;
    *A = *B;
    *B = AIDE;
}

void LIRE_TAB (int *TAB, int *N, int NMAX)
{
    . . .
}

void ECRIRE_TAB (int *TAB, int N)
{
    . . .
}

```

Exercice 10.18 :

Déterminer le maximum de N éléments d'un tableau TAB d'entiers de trois façons différentes :

- a) la fonction MAX1 retourne la valeur maximale

```

int MAX1(int *TAB, int N)
{
    int MAX,I; /* variables d'aide */
    MAX=*TAB;
    for (I=1; I<N; I++)
        if (MAX < *(TAB+I))
            MAX = *(TAB+I);
    return MAX;
}

```

```

}
b) la fonction MAX2 retourne l'indice de l'élément maximal
int MAX2(int *TAB, int N)
{
    int I,MAX; /* variables d'aide */
    MAX=0;
    for (I=1; I<N; I++)
        if (*(TAB+MAX) < *(TAB+I))
            MAX = I;
    return MAX;
}
c) la fonction MAX3 retourne l'adresse de l'élément maximal
int *MAX3(int *TAB, int N)
{
    int *MAX, *P; /* pointeurs d'aide */
    MAX=TAB;
    for (P=TAB; P<TAB+N; P++)
        if (*MAX < *P)
            MAX=P;
    return MAX;
}

```

Ecrire un programme pour tester les trois fonctions :

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    int MAX1 (int *TAB, int N);
    int MAX2 (int *TAB, int N);
    int *MAX3(int *TAB, int N);
    void LIRE_TAB (int *TAB, int *N, int NMAX);
    void ECRIRE_TAB (int *TAB, int N);
    /* Variables locales */
    int T[100]; /* Tableau d'entiers */
    int DIM; /* Dimension du tableau */

    /* Traitements */
    LIRE_TAB (T, &DIM, 100);
    printf("Tableau donné : \n");
    ECRIRE_TAB (T, DIM);
    printf("MAX1 : %d \n", MAX1 (T,DIM) );
    printf("MAX2 : %d \n", T [MAX2 (T,DIM) ] );
    printf("MAX3 : %d \n", *MAX3 (T,DIM) );
    return (0);
}

int MAX1(int *TAB, int N)
{
    . . .
}

int MAX2(int *TAB, int N)
{

```

```

    . . .
}

int *MAX3(int *TAB, int N)
{
    . . .
}

void LIRE_TAB (int *TAB, int *N, int NMAX)
{
    . . .
}

void ECRIRE_TAB (int *TAB, int N)
{
    . . .
}

```

Exercice 10.19 :

Tri par sélection :

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    void TRI_SELECTION(int *T, int N);
    void LIRE_TAB (int *TAB, int *N, int NMAX);
    void ECRIRE_TAB (int *TAB, int N);
    /* Variables locales */
    int T[100]; /* Tableau d'entiers */
    int DIM;    /* Dimension du tableau */

    /* Traitements */
    LIRE_TAB (T, &DIM, 100);
    printf("Tableau donné : \n");
    ECRIRE_TAB (T, DIM);
    TRI_SELECTION(T, DIM);
    printf("Tableau trié : \n");
    ECRIRE_TAB (T, DIM);
    return (0);
}

void TRI_SELECTION(int *T, int N)
{
    /* Prototypes des fonctions appelées */
    void PERMUTER(int *A, int *B);
    int *MAX3(int *TAB, int N);
    /* Variables locales */
    int I; /* rang à partir duquel T n'est pas trié */

    /* Tri par sélection directe du maximum */
    for (I=0 ; I<N-1 ; I++)
        PERMUTER(T+I, MAX3(T+I,N-I) );
}

```

```

}

int *MAX3(int *TAB, int N)
{
    . . .
}

void PERMUTER(int *A, int *B)
{
    . . .
}

void LIRE_TAB (int *TAB, int *N, int NMAX)
{
    . . .
}

void ECRIRE_TAB (int *TAB, int N)
{
    . . .
}

```

Exercice 10.20 :

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    void INSERER(int X, int *T, int *N);
    void LIRE_TAB (int *TAB, int *N, int NMAX);
    void ECRIRE_TAB (int *TAB, int N);
    /* Variables locales */
    int T[100]; /* Tableau d'entiers */
    int DIM;    /* Dimension du tableau */
    int A;     /* Nombre à insérer */
    /* Traitements */
    LIRE_TAB (T, &DIM, 100);
    printf("Tableau donné : \n");
    ECRIRE_TAB (T, DIM);
    printf("Introduire le nombre à insérer : ");
    scanf("%d", &A);
    INSERER(A, T, &DIM);
    printf("Tableau résultat : \n");
    ECRIRE_TAB (T, DIM);
    return (0);
}

void INSERER(int X, int *T, int *N)
{
    /* Variables locales */
    int I;
    /* Insertion de X dans le tableau T supposé trié : */
    /* Déplacer les éléments plus grands que X d'une */

```

```

/* position vers l'arrière. */
for (I=*N ; I>0 && *(T+I-1)>X ; I--)
    *(T+I) = *(T+I-1);
/* X est copié à la position du dernier élément déplacé */
*(T+I)=X;
/* Nouvelle dimension du tableau : */
(*N)++; /* Attention aux parenthèses ! */
}

void LIRE_TAB (int *TAB, int *N, int NMAX)
{
    . . .
}

void ECRIRE_TAB (int *TAB, int N)
{
    . . .
}

```

Exercice 10.21 :

Tri par insertion

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    void TRI_INSERTION(int *T, int N);
    void LIRE_TAB (int *TAB, int *N, int NMAX);
    void ECRIRE_TAB (int *TAB, int N);
    /* Variables locales */
    int T[100]; /* Tableau d'entiers */
    int DIM;    /* Dimension du tableau */
    /* Traitements */
    LIRE_TAB (T, &DIM, 100);
    printf("Tableau donné : \n");
    ECRIRE_TAB (T, DIM);
    TRI_INSERTION(T, DIM);
    printf("Tableau trié : \n");
    ECRIRE_TAB (T, DIM);
    return (0);
}

void TRI_INSERTION(int *T, int N)
{
    void INSERER(int X, int *T, int *N);
    /* Variables locales */
    int I; /* indice courant */
    /* Tri de T par insertion */
    I=1;
    while (I<N)
        INSERER(*(T+I), T, &I);
}

```

```

void INSERER(int X, int *T, int *N)
{
    . . .
}

void LIRE_TAB (int *TAB, int *N, int NMAX)
{
    . . .
}

void ECRIRE_TAB (int *TAB, int N)
{
    . . .
}

```

Exercice 10.22 :

```

int RANGER(int *X, int *Y)
{
    int AIDE;
    if (*X>*Y)
        {
            AIDE = *X;
            *X = *Y;
            *Y = AIDE;
            return 1;
        }
    else
        return (0);
}

```

Exercice 10.23 :

Tri par propagation

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    void LIRE_TAB (int *TAB, int *N, int NMAX);
    void TRI_BULLE(int *T, int N);
    void ECRIRE_TAB (int *TAB, int N);
    /* Variables locales */
    int T[100]; /* Tableau d'entiers */
    int DIM;    /* Dimension du tableau */
    /* Traitements */
    LIRE_TAB (T, &DIM, 100);
    printf("Tableau donné : \n");
    ECRIRE_TAB (T, DIM);
    TRI_BULLE(T, DIM);
    printf("Tableau trié : \n");
    ECRIRE_TAB (T, DIM);
    return (0);
}

```

```

void TRI_BULLE(int *T, int N)
{
    /* Prototypes des fonctions appelées */
    int RANGER(int *X, int *Y);
    /* Variables locales */
    int I,J; /* indices courants */
    int FIN; /* position où la dernière permutation a eu lieu */
    /* permet de ne pas trier un sous-ensemble déjà trié. */
    /* Tri de T par propagation de l'élément maximal */
    for (I=N-1 ; I>0 ; I=FIN)
    {
        FIN=0;
        for (J=0; J<I; J++)
            if (RANGER(T+J, T+J+1)) FIN = J;
    }
}

int RANGER(int *X, int *Y)
{
    . . .
}
void LIRE_TAB (int *TAB, int *N, int NMAX)
{
    . . .
}
void ECRIRE_TAB (int *TAB, int N)
{
    . . .
}

```

Exercice 10.24 :

Fusion de tableaux triés

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    void FUSION(int *A, int *B, int *FUS, int N, int M);
    void TRI_BULLE(int *T, int N);
    void LIRE_TAB (int *TAB, int *N, int NMAX);
    void ECRIRE_TAB (int *TAB, int N);
    /* Variables locales */
    /* Les tableaux et leurs dimensions */
    int A[100], B[100], FUS[200];
    int N, M;
    /* Traitements */
    printf("*** Tableau A ***\n");
    LIRE_TAB (A, &N, 100);
    printf("*** Tableau B ***\n");
    LIRE_TAB (B, &M, 100);
}

```

```

TRI_BULLE(A, N);
    printf("Tableau A trié : \n");
ECRIRE_TAB (A, N);
TRI_BULLE(B, M);
    printf("Tableau B trié : \n");
ECRIRE_TAB (B, M);
FUSION(A, B, FUS, N, M);
    printf("Tableau FUS : \n");
ECRIRE_TAB (FUS, N+M);
    return (0);
}

void FUSION(int *A, int *B, int *FUS, int N, int M)
{
    /* Variables locales */
    /* Indices courants dans A, B et FUS */
    int IA, IB, IFUS;
    /* Fusion de A et B dans FUS */
    IA=0, IB=0; IFUS=0;
    while ((IA<N) && (IB<M))
        if (*(A+IA)<*(B+IB))
            {
                *(FUS+IFUS)=*(A+IA);
                IFUS++;
                IA++;
            }
        else
            {
                FUS[IFUS]=B[IB];
                IFUS++;
                IB++;
            }
    /* Si A ou B sont arrivés à la fin, alors */
    /* copier le reste de l'autre tableau. */
    while (IA<N)
        {
            *(FUS+IFUS)=*(A+IA);
            IFUS++;
            IA++;
        }
    while (IB<M)
        {
            *(FUS+IFUS)=*(B+IB);
            IFUS++;
            IB++;
        }
}

void TRI_BULLE(int *T, int N)
{
    /* Prototypes des fonctions appelées */
    int RANGER(int *X, int *Y);
    . . .
}

```

```

}
int RANGER(int *X, int *Y)
{
. . .
}
void LIRE_TAB (int *TAB, int *N, int NMAX)
{
. . .
}
void ECRIRE_TAB (int *TAB, int N)
{
. . .
}

```

5) Chaînes de caractères :

Exercice 10.25 :

```

int LONG_CH(char *CH)
{
char *P;
for (P=CH ; *P; P++) ;
return P-CH;
}

```

Exercice 10.26 :

```

void MAJ_CH(char *CH)
{
for ( ; *CH; CH++)
if (*CH>='a' && *CH<='z')
*CH = *CH-'a'+'A';
}

```

Exercice 10.27 :

```

void AJOUTE_CH(char *CH1, char *CH2)
{
while (*CH1) /* chercher la fin de CH1 */
CH1++;
while (*CH2) /* copier CH2 à la fin de CH1 */
{
*CH1 = *CH2;
CH1++;
CH2++;
}
*CH1='\0'; /* terminer la chaîne CH1 */
}

```

Solution plus compacte :

```

void AJOUTE_CH(char *CH1, char *CH2)
{
for ( ; *CH1 ; CH1++) ;

```

```

    for ( ; *CH1 = *CH2 ; CH1++, CH2++) ;
}

```

Comme la condition d'arrêt est évaluée lors de l'affectation, le symbole de fin de chaîne est aussi copié.

Exercice 10.28 :

```

void INVERSER_CH (char *CH)
{
    /* Prototypes des fonctions appelées */
    int LONG_CH(char *CH);
    void PERMUTER_CH(char *A, char *B);
    /* Variables locales */
    int I,J;
    /* Inverser la chaîne par permutations successives */
    J = LONG_CH(CH)-1;
    for (I=0 ; I<J ; I++,J--)
        PERMUTER_CH(CH+I, CH+J);
}

void PERMUTER_CH(char *A, char *B)
{
    char AIDE;
    AIDE = *A;
    *A = *B;
    *B = AIDE;
}

int LONG_CH(char *CH)
{
    . . .
}

```

Exercice 10.29 :

```

#include <ctype.h>
int NMOTS_CH(char *CH)
{
    /* Variables locales */
    int N;          /* nombre de mots */
    int DANS_MOT;  /* indicateur logique : */
                  /* vrai si CH pointe à l'intérieur d'un mot */
    DANS_MOT=0;
    for (N=0; *CH; CH++)
        if (isspace(*CH))
            DANS_MOT=0;
        else if (!DANS_MOT)
            {
                DANS_MOT=1;
                N++;
            }
    return N;
}

```

Exercice 10.30 :

```
#include <ctype.h>
char *MOT_CH(int N, char *CH)
{
    /* Variables locales */
    int DANS_MOT; /* indicateur logique : */
                    /* vrai si CH pointe à l'intérieur d'un mot */
    DANS_MOT=0;
    for ( ; N>0 && *CH ; CH++)
        if (isspace(*CH))
            DANS_MOT=0;
        else if (!DANS_MOT)
            {
                DANS_MOT=1;
                N--;
                CH--; /* Pour réajuster l'effet de l'incrémentatation
*/
            }
    return CH;
}
```

Exercice 10.31 :

```
int EGAL_N_CH(int N, char *CH1, char *CH2)
{
    while (--N && *CH1==*CH2)
        {
            CH1++;
            CH2++;
        }
    return (*CH1==*CH2);
}
```

Exercice 10.32 :

```
char *CHERCHE_CH(char *CH1, char *CH2)
{
    /* Prototypes des fonctions appelées */
    int LONG_CH(char *CH);
    int EGAL_N_CH(int N, char *CH1, char *CH2);
    /* Variables locales */
    int L;
    /* Recherche de CH1 dans CH2 */
    L=LONG_CH(CH1);
    while (*CH2 && !EGAL_N_CH(L, CH1, CH2))
        CH2++;
    return CH2;
}

int LONG_CH(char *CH)
{

```

```

    . . .
}

int EGAL_N_CH(int N, char *CH1, char *CH2)
{
    . . .
}

```

Exercice 10.33 :

```

long CH_ENTIER(char *CH)
{
    /* Variables locales */
    long N;
    int SIGNE;
    /* Traitement du signe */
    SIGNE = 1;
    if (*CH=='-') SIGNE = -1;
    if (*CH=='-' || *CH=='+') CH++;
    /* Conversion des chiffres */
    for (N=0 ; *CH>='0' && *CH<='9' ; CH++)
        N = N*10 + (*CH-'0');
    return SIGNE*N;
}

```

Exercice 10.34 :

```

#include <ctype.h>
#include <math.h>
double CH_DOUBLE(char *CH)
{
    /* Variables locales */
    double N; /* résultat numérique */
    int SIGNE; /* signe de la valeur rationnelle */
    int DEC; /* positions derrière la virgule */

    /* Initialisation des variables */
    N = 0.0;
    SIGNE = 1;
    /* Traitement du signe */
    if (*CH=='-') SIGNE = -1;
    if (*CH=='-' || *CH=='+') CH++;
    /* Positions devant le point décimal */
    for ( ; isdigit(*CH); CH++)
        N = N*10.0 + (*CH-'0');
    /* Traitement du point décimal */
    if (*CH=='.') CH++;

    /* Traitement des positions derrière le point décimal */
    for (DEC=0; isdigit(*CH); CH++)
    {
        N = N*10.0 + (*CH-'0');
        DEC++;
    }
}

```

```

    }
    /* Calcul de la valeur à partir du signe et */
    /* du nombre de décimales. */
    return SIGNE*N/pow(10,DEC);
}

```

Exercice 10.35 :

```

#include <stdio.h>
void ENTIER_CH(long N, char *CH)
{
    /* Prototypes des fonctions appelées */
    void INVERSER_CH(char *CH);
    /* Variables locales */
    int I;
    int SIGNE;
    /* Traitement du signe */
    SIGNE = (N<0) ? -1 : 1;
    if (N<0) N=-N;
    /* Conversion des chiffres (à rebours) */
    I=0;
    do
    {
        *(CH+I) = N % 10 + '0';
        I++;
    }
    while ((N/=10) > 0);
    /* Ajouter le signe à la fin de la chaîne */
    if (SIGNE<0)
    {
        *(CH+I)='-';
        I++;
    }
    /* Terminer la chaîne */
    *(CH+I)='\0';
    /* Inverser la chaîne */
    INVERSER_CH(CH);
}

void INVERSER_CH (char *CH)
{
    /* Prototypes des fonctions appelées */
    int LONG_CH(char *CH);
    void PERMUTER_CH(char *A, char *B);
    . . .
}

int LONG_CH(char *CH)
{
    . . .
}

void PERMUTER_CH(char *A, char *B)

```

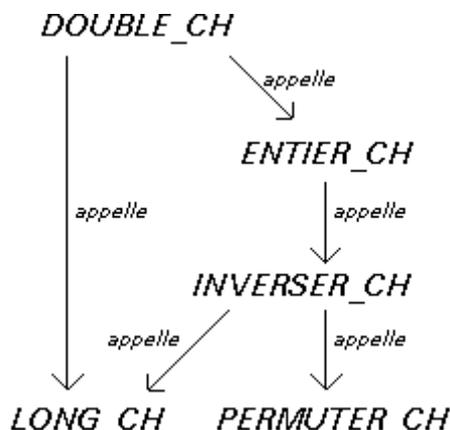
```

{
. . .
}

```

Exercice 10.36 :

Solution :



```

void DOUBLE_CH(double N, char *CH)
{
  /* Prototypes des fonctions appelées */
  int LONG_CH(char *CH);
  void ENTIER_CH(long N, char *CH);
  /* Variables locales */
  int I,L;
  /* Conversion */
  N *= 10000.0;
  ENTIER_CH((long)N, CH); /* Conversion forcée est facultative
  */
  L=LONG_CH(CH);
  for (I=L; I>=L-4; I--) /* Libérer une position pour le */
    *(CH+I+1) = *(CH+I); /* point décimal. */
  *(CH+L-4)='.';
}

void ENTIER_CH(long N, char *CH)
{
  /* Prototypes des fonctions appelées */
  void INVERSER_CH(char *CH);
  . . .
}

void INVERSER_CH (char *CH)
{
  /* Prototypes des fonctions appelées */
  int LONG_CH(char *CH);
  void PERMUTER_CH(char *A, char *B);
  . . .
}

```

```

int LONG_CH(char *CH)
{
    . . .
}

void PERMUTER_CH(char *A, char *B)
{
    . . .
}

```

6) Tableaux à deux dimensions :

Exercice 10.37 :

```

void LIRE_DIM (int *L, int LMAX, int *C, int CMAX)
{
    /* Saisie des dimensions de la matrice */
    do
    {
        printf("Nombre de lignes de la matrice      (max.%d)  :
", LMAX);
        scanf("%d", L);
    }
    while (*L<0 || *L>LMAX);
    do
    {
        printf("Nombre de colonnes de la matrice (max.%d)  :
", CMAX);
        scanf("%d", C);
    }
    while (*C<0 || *C>CMAX);
}

```

b) Ecrire la fonction `LIRE_MATRICE` à quatre paramètres `MAT`, `L`, `C`, et `CMAX` qui lit les composantes d'une matrice `MAT` du type `int` et de dimensions `L` et `C`.

```

void LIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
    /* Variables locales */
    int I,J;
    /* Saisie des composantes de la matrice */
    for (I=0; I<L; I++)
        for (J=0; J<C; J++)
        {
            printf("Elément [%d] [%d] : ", I, J);
            scanf("%d", MAT + I*CMAX + J);
        }
}

```

Exercice 10.38 :

```

void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
    /* Variables locales */
    int I,J;

```

```

/* Affichage des composantes de la matrice */
for (I=0; I<L; I++)
{
    for (J=0; J<C; J++)
        printf("%7d", *(MAT + I*CMAX + J));
    printf("\n");
}
}

```

Exercice 10.39 :

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    long SOMME_MATRICE (int *MAT, int L, int C, int CMAX);
    void LIRE_DIM      (int *L, int LMAX, int *C, int CMAX);
    void LIRE_MATRICE  (int *MAT, int L, int C, int CMAX);
    void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX);

    /* Variables locales */
    int M[30][30]; /* Matrice d'entiers */
    int L, C;      /* Dimensions de la matrice */
    /* Traitements */
    LIRE_DIM (&L, 30, &C, 30);
    LIRE_MATRICE ( (int*)M, L,C,30);
    printf("Matrice donnée : \n");
    ECRIRE_MATRICE ( (int*)M, L,C,30);
    printf("Somme des éléments de la matrice : %ld\n",
           SOMME_MATRICE( (int*)M, L,C,30));

    return (0);
}

long SOMME_MATRICE(int *MAT, int L, int C, int CMAX)
{
    /* Variables locales */
    int I,J;
    long SOMME = 0;
    /* Calcul de la somme */
    for (I=0; I<L; I++)
        for (J=0; J<C; J++)
            SOMME += *(MAT + I*CMAX + J);
    return SOMME;
}

void LIRE_DIM (int *L, int LMAX, int *C, int CMAX)
{
    . . .
}

void LIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{

```

```

    . . .
}

void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
    . . .
}

```

Exercice 10.40 :

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    void ADDITION_MATRICE (int *MAT1, int *MAT2, int L, int C,
int CMAX);
    void LIRE_DIM          (int *L, int LMAX, int *C, int CMAX);
    void LIRE_MATRICE      (int *MAT, int L, int C, int CMAX);
    void ECRIRE_MATRICE    (int *MAT, int L, int C, int CMAX);
    /* Variables locales */
    /* Les matrices et leurs dimensions */
    int M1[30][30], M2[30][30];
    int L, C;
    /* Traitements */
    LIRE_DIM (&L,30,&C,30);
    printf("*** Matrice 1 ***\n");
    LIRE_MATRICE ((int*)M1,L,C,30 );
    printf("*** Matrice 2 ***\n");
    LIRE_MATRICE ((int*)M2,L,C,30 );
    printf("Matrice donnée 1 : \n");
    ECRIRE_MATRICE ((int*)M1,L,C,30);
    printf("Matrice donnée 2 : \n");
    ECRIRE_MATRICE ((int*)M2,L,C,30);
    ADDITION_MATRICE( (int*)M1 , (int*)M2 ,L,C,30);
    printf("Matrice résultat : \n");
    ECRIRE_MATRICE ((int*)M1,L,C,30);
    return (0);
}

void ADDITION_MATRICE (int *MAT1, int *MAT2, int L, int C,
int CMAX)
{
    /* Variables locales */
    int I,J;
    /* Ajouter les éléments de MAT2 à MAT1 */
    for (I=0; I<L; I++)
        for (J=0; J<C; J++)
            *(MAT1+I*CMAX+J) += *(MAT2+I*CMAX+J);
}

void LIRE_DIM (int *L, int LMAX, int *C, int CMAX)
{
    . . .
}

```

```

}

void LIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
    . . .
}

void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
    . . .
}

```

Exercice 10.41 :

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    void MULTI_MATRICE(int X, int *MAT, int L, int C, int CMAX);
    void LIRE_DIM      (int *L, int LMAX, int *C, int CMAX);
    void LIRE_MATRICE  (int *MAT, int L, int C, int CMAX);
    void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX);
    /* Variables locales */
    int M[30][30]; /* Matrice d'entiers */
    int L, C;      /* Dimensions de la matrice */
    int X;
    /* Traitements */
    LIRE_DIM (&L,30,&C,30);
    LIRE_MATRICE ((int*)M,L,C,30 );
    printf("Introduire le multiplicateur (entier) : ");
    scanf("%d", &X);
    printf("Matrice donnée : \n");
    ECRIRE_MATRICE ((int*)M,L,C,30);
    MULTI_MATRICE (X, (int*)M,L,C,30);
    printf("Matrice résultat : \n");
    ECRIRE_MATRICE ((int*)M,L,C,30);
    return (0);
}

void MULTI_MATRICE(int X, int *MAT, int L, int C, int CMAX)
{
    /* Variables locales */
    int I,J;
    /* Multiplication des éléments */
    for (I=0; I<L; I++)
        for (J=0; J<C; J++)
            *(MAT+I*CMAX+J) *= X;
}

void LIRE_DIM (int *L, int LMAX, int *C, int CMAX)
{
    . . .
}

```

```

}

void LIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
. . .
}

void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
. . .
}

```

Exercice 10.42 :

```

#include <stdio.h>
main()
{
  /* Prototypes des fonctions appelées */
  int TRANSPON_MATRICE (int *MAT, int *L, int LMAX, int *C, int
CMAX);
  void LIRE_DIM      (int *L, int LMAX, int *C, int CMAX);
  void LIRE_MATRICE  (int *MAT, int L, int C, int CMAX);
  void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX);
  /* Variables locales */
  int M[30][30]; /* Matrice d'entiers */
  int L, C;      /* Dimensions de la matrice */
  /* Traitements */
  LIRE_DIM (&L,30,&C,30);
  LIRE_MATRICE ((int*)M,L,C,30 );
  printf("Matrice donnée : \n");
  ECRIRE_MATRICE ((int*)M,L,C,30);
  if (TRANSPON_MATRICE ((int*)M,&L,30,&C,30))
    {
      printf("Matrice transposée : \n");
      ECRIRE_MATRICE ((int*)M,L,C,30);
    }
  else
    printf("\aLa matrice n'a pas pu être transposée\n");
  return (0);
}

int TRANSPON_MATRICE (int *MAT, int *L, int LMAX, int *C, int
CMAX)
{
  /* Prototypes des fonctions appelées */
  void PERMUTER(int *A, int *B);
  /* Variables locales */
  int I,J;
  int DMAX; /* la plus grande des deux dimensions */
  /* Transposition de la matrice */
  if (*L>CMAX || *C>LMAX)
    return (0);
  else

```

```

    {
        DMAX = (*L>*C) ? *L : *C;
        for (I=0; I<DMAX; I++)
            for (J=0; J<I; J++)
                PERMUTER (MAT+I*CMAX+J, MAT+J*CMAX+I);
        PERMUTER(L,C); /* échanger les dimensions */
        return 1;
    }
}

void PERMUTER(int *A, int *B)
{
    . . .
}

void LIRE_DIM (int *L, int LMAX, int *C, int CMAX)
{
    . . .
}

void LIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
    . . .
}

void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
    . . .
}

```

Exercice 10.43 :

```

#include <stdio.h>
main()
{
    /* Prototypes des fonctions appelées */
    void MULTI_2_MATRICES (int *MAT1, int *MAT2, int *MAT3,
                          int N, int M, int P, int CMAX);
    void LIRE_DIM (int *L, int LMAX, int *C, int CMAX);
    void LIRE_MATRICE (int *MAT, int L, int C, int CMAX);
    void ECRIRE_MATRICE (int *MAT, int L, int C, int CMAX);
    /* Variables locales */
    /* Les matrices et leurs dimensions */
    int M1[30][30], M2[30][30], M3[30][30];
    int N, M, P;
    int DUMMY; /* pour la lecture de la première dimension de */
              /* MAT2 à l'aide de LIRE_DIM. */

    /* Traitements */
    printf("*** Matrice 1 ***\n");
    LIRE_DIM (&N, 30, &M, 30);
    LIRE_MATRICE ((int*)M1, N, M, 30 );
    printf("*** Matrice 2 ***\n");
}

```

```

LIRE_DIM (&DUMMY,30,&P,30);
LIRE_MATRICE ((int*)M2,M,P,30);
printf("Matrice donnée 1 : \n");
Ecrire_MATRICE ((int*)M1,N,M,30);
printf("Matrice donnée 2 : \n");
Ecrire_MATRICE ((int*)M2,M,P,30);
MULTI_2_MATRICES ((int*)M1 , (int*)M2 , (int*)M3 ,
N,M,P,30);
printf("Matrice résultat : \n");
Ecrire_MATRICE ((int*)M3,N,P,30);
return (0);
}

void MULTI_2_MATRICES (int *MAT1, int *MAT2, int *MAT3,
int N, int M, int P, int CMAX)
{
/* Variables locales */
int I,J,K;
/* Multiplier MAT1 et MAT2 en affectant le résultat à MAT3
*/
for (I=0; I<N; I++)
for (J=0; J<P; J++)
{
*(MAT3+I*CMAX+J)=0;
for (K=0; K<M; K++)
*(MAT3+I*CMAX+J) += *(MAT1+I*CMAX+K) *
*(MAT2+K*CMAX+J);
}
}

void LIRE_DIM (int *L, int LMAX, int *C, int CMAX)
{
. . .
}

void LIRE_MATRICE (int *MAT, int L, int C, int CMAX)
{
. . .
}

void Ecrire_MATRICE (int *MAT, int L, int C, int CMAX)
{
. . .
}

```

Chapitre 11 : LES FICHIERS SEQUENTIELS :

En C, les communications d'un programme avec son environnement se font par l'intermédiaire de fichiers. Pour le programmeur, tous les périphériques, même le clavier et l'écran, sont des fichiers. Jusqu'ici, nos programmes ont lu leurs données dans le fichier d'entrée standard, (c'est-à-dire : le clavier) et ils ont écrit leurs résultats dans le fichier de sortie standard (c'est-à-dire : l'écran). Nous allons voir dans ce chapitre, comment nous pouvons créer, lire et modifier nous-mêmes des fichiers sur les périphériques disponibles.

I) Définitions et propriétés :

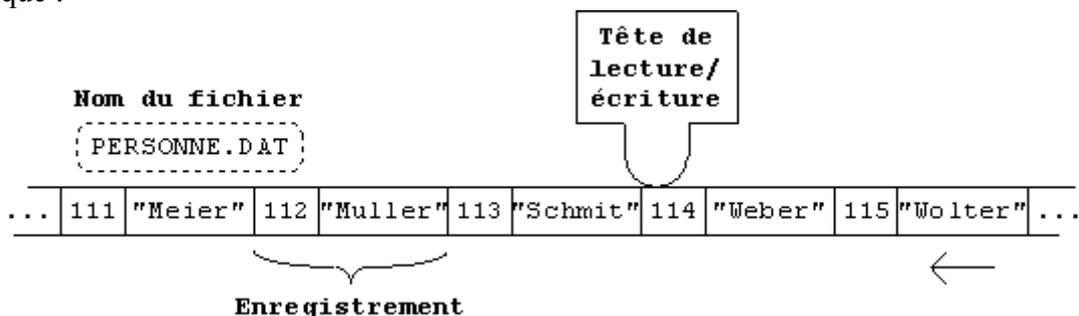
1) Fichier :

Un **fichier** (angl.: *file*) est un ensemble structuré de données stocké en général sur un support externe (disquette, disque dur, disque optique, bande magnétique, ...). Un **fichier structuré** contient une suite *d'enregistrements* homogènes, qui regroupent le plus souvent plusieurs composantes appartenant ensemble (*champs*).

2) Fichier séquentiel :

Dans des **fichiers séquentiels**, les enregistrements sont mémorisés consécutivement dans l'ordre de leur entrée et peuvent seulement être lus dans cet ordre. Si on a besoin d'un enregistrement précis dans un fichier séquentiel, il faut lire tous les enregistrements qui le précèdent, en commençant par le premier.

En simplifiant, nous pouvons nous imaginer qu'un fichier séquentiel est enregistré sur une bande magnétique :



Propriétés :

Les fichiers séquentiels que nous allons considérer dans ce cours auront les propriétés suivantes :

- * Les fichiers se trouvent ou bien en état d'écriture ou bien en état de lecture; nous ne pouvons pas simultanément lire et écrire dans le même fichier.
- * A un moment donné, on peut uniquement accéder à un seul enregistrement; celui qui se trouve en face de la tête de lecture/écriture.
- * Après chaque accès, la tête de lecture/écriture est déplacée derrière la donnée lue en dernier lieu.

3) Fichiers standards :

Il existe deux fichiers spéciaux qui sont définis par défaut pour tous les programmes :

- **stdin** le fichier d'entrée standard
- **stdout** le fichier de sortie standard

En général, *stdin* est lié au clavier et *stdout* est lié à l'écran, c'est-à-dire les programmes lisent leurs données au clavier et écrivent les résultats sur l'écran.

En UNIX et en MS-DOS, il est possible de dévier l'entrée et la sortie standard vers d'autres fichiers ou périphériques à l'aide des symboles < (pour *stdin*) et > (pour *stdout*) :

Exemple :

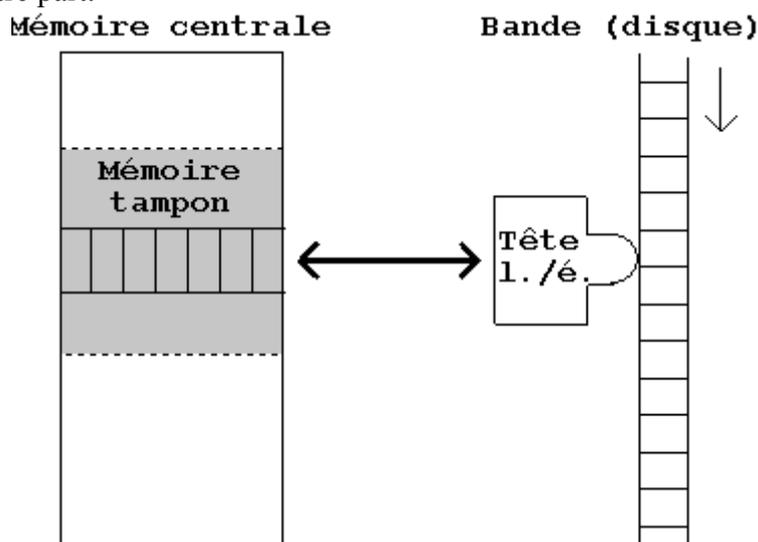
L'appel suivant du programme PROG lit les données dans le fichier C:\TEST.TXT au lieu du clavier et écrit les résultats sur l'imprimante au lieu de l'écran.

```
PROG <C:\TEST.TXT >PRN:
```

En fait, l'affectation de *stdin* et *stdout* est gérée par le système d'exploitation; ainsi le programme ne 'sait' pas d'où viennent les données et où elles vont.

II) La mémoire tampon :

Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une *mémoire tampon* (angl.: *buffer* ; allem.: *Pufferspeicher*). La mémoire tampon est une zone de la mémoire centrale de la machine réservée à un ou plusieurs enregistrements du fichier. L'utilisation de la mémoire tampon a l'effet de réduire le nombre d'accès à la périphérie d'une part et le nombre des mouvements de la tête de lecture/écriture d'autre part.



III) Accès aux fichiers séquentiels :

Les problèmes traitant des fichiers ont généralement la forme suivante : un fichier donné par son nom (et en cas de besoin le chemin d'accès sur le médium de stockage) doit être créé, lu ou modifié. La question qui se pose est alors :

Comment pouvons-nous relier le nom d'un fichier sur un support externe avec les instructions qui donnent accès au contenu du fichier ?

En résumé, la méthode employée sera la suivante :

Avant de lire ou d'écrire un fichier, l'accès est notifié par la commande **fopen**. **fopen** accepte le nom du fichier (p.ex : "A:\ADRESSES.DAT"), négocie avec le système d'exploitation et fournit un pointeur spécial qui sera utilisé ensuite lors de l'écriture ou la lecture du fichier. Après les traitements, il faut annuler la liaison entre le nom du fichier et le pointeur à l'aide de la commande **fclose**.

On peut dire aussi qu'entre les événements **fopen()** et **fclose()** le fichier est ouvert.

1) Le type FILE* :

Pour pouvoir travailler avec un fichier, un programme a besoin d'un certain nombre d'informations au sujet du fichier :

- adresse de la mémoire tampon,
- position actuelle de la tête de lecture/écriture,
- type d'accès au fichier : écriture, lecture, ...
- état d'erreur,
- ...

Ces informations (dont nous n'aurons pas à nous occuper), sont rassemblées dans une structure du type spécial **FILE**. Lorsque nous ouvrons un fichier avec la commande **fopen**, le système génère automatiquement un bloc du type **FILE** et nous fournit son adresse.

Tout ce que nous avons à faire dans notre programme est :

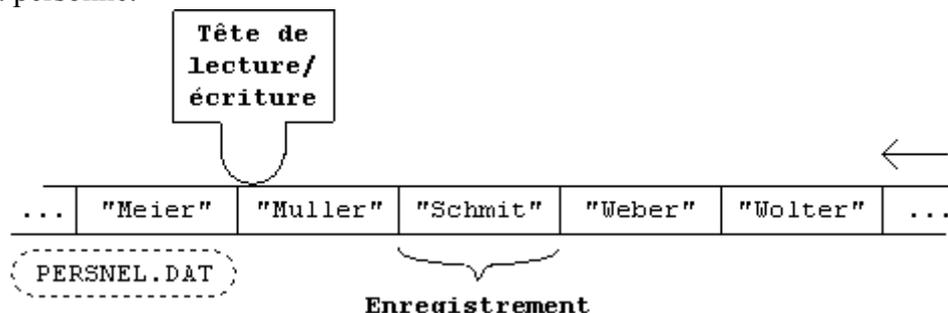
- * déclarer un pointeur du type **FILE*** pour chaque fichier dont nous avons besoin,
- * affecter l'adresse retournée par **fopen** à ce pointeur,
- * employer le pointeur à la place du nom du fichier dans toutes les instructions de lecture ou d'écriture,
- * libérer le pointeur à la fin du traitement à l'aide de **fclose**.

2) Exemple : Créer et afficher un fichier séquentiel :

Avant de discuter les détails du traitement des fichiers, nous vous présentons un petit exemple comparatif qui réunit les opérations les plus importantes sur les fichiers.

Problème :

On se propose de créer un fichier qui est formé d'enregistrements contenant comme information le nom d'une personne. Chaque enregistrement est donc constitué d'une seule rubrique, à savoir, le nom de la personne.



L'utilisateur doit entrer au clavier le nom du fichier, le nombre de personnes et les noms des personnes. Le programme se chargera de créer le fichier correspondant sur disque dur ou sur disquette.

Après avoir écrit et fermé le fichier, le programme va rouvrir le même fichier en lecture et afficher son contenu, sans utiliser le nombre d'enregistrements introduit dans la première partie.

Solution en langage algorithmique :

```

programme PERSONNEL
  chaîne NOM_FICHIER, NOM_PERS
  entier C, _NB_ENREG

  (* Première partie :
   Créer et remplir le fichier *)
  écrire "Entrez le nom du fichier à créer : "
  lire NOM_FICHIER
  ouvrir NOM_FICHIER en écriture
  écrire "Nombre d'enregistrements à créer : "
  lire NB_ENREG
  en C ranger 0
  tant que (C < NB_ENREG) faire
  | écrire "Entrez le nom de la personne : "
  | lire NOM_PERS
  | écrire NOM_FICHIER:NOM_PERS
  | en C ranger C+1
  ftant (* C=NB_ENREG *)
  fermer NOM_FICHIER

  (* Deuxième partie :
   Lire et afficher le contenu du fichier *)
  ouvrir NOM_FICHIER en lecture
  en C ranger 0
  
```

```

    tant que non(finfichier(NOM_FICHIER)) faire
    | lire NOM_FICHIER:NOM_PERS
    | écrire "NOM : ",NOM_PERS
    | en C ranger C+1
ftant
fermer NOM_FICHIER
fprogramme (* fin PERSONNEL *)

```

Solution en langage C :

```

#include <stdio.h>

main()
{
    FILE *P_FICHIER; /* pointeur sur FILE */
    char NOM_FICHIER[30], NOM_PERS[30];
    int C,NB_ENREG;

    /* Première partie :
       Créer et remplir le fichier */
    printf("Entrez le nom du fichier à créer : ");
    scanf("%s", NOM_FICHIER);
    P_FICHIER = fopen(NOM_FICHIER, "w"); /* write */
    printf("Nombre d'enregistrements à créer : ");
    scanf("%d", &NB_ENREG);
    C = 0;
    while (C<NB_ENREG)
    {
        printf("Entrez le nom de la personne : ");
        scanf("%s", NOM_PERS);
        fprintf(P_FICHIER, "%s\n", NOM_PERS);
        C++;
    }
    fclose(P_FICHIER);

    /* Deuxième partie :
       Lire et afficher le contenu du fichier */
    P_FICHIER = fopen(NOM_FICHIER, "r"); /* read */
    C = 0;
    while (!feof(P_FICHIER))
    {
        fscanf(P_FICHIER, "%s\n", NOM_PERS);
        printf("NOM : %s\n", NOM_PERS);
        C++;
    }
    fclose(P_FICHIER);
    return (0);
}

```

>> Voir aussi : [Chapitre 11.4.3](#). Exemples : Ouvrir et fermer des fichiers en pratique

IV) Ouvrir et fermer des fichiers séquentiels :

Avant de créer ou de lire un fichier, nous devons informer le système de cette intention pour qu'il puisse réserver la mémoire pour la zone d'échange et initialiser les informations nécessaires à l'accès du fichier. Nous parlons alors de l'*ouverture* d'un fichier.

Après avoir terminé la manipulation du fichier, nous devons vider la mémoire tampon et libérer l'espace en mémoire que nous avons occupé pendant le traitement. Nous parlons alors de la *fermeture* du fichier.

L'ouverture et la fermeture de fichiers se font à l'aide des fonctions **fopen** et **fclose** définies dans la bibliothèque standard `<stdio>`.

1) Ouvrir un fichier séquentiel :

Ouvrir un fichier en langage algorithmique :

ouvrir <Nom> en écriture
ou bien

ouvrir <Nom> en lecture

<Nom> est une chaîne de caractères constante ou une variable de type chaîne qui représente le nom du fichier sur le moyen de stockage.

Ouvrir un fichier en C - fopen :

Lors de l'ouverture d'un fichier avec **fopen**, le système s'occupe de la réservation de la mémoire tampon dans la mémoire centrale et génère les informations pour un nouvel élément du type **FILE**. L'adresse de ce bloc est retournée comme résultat si l'ouverture s'est déroulée avec succès. La commande **fopen** peut ouvrir des fichiers en écriture ou en lecture en dépendance de son deuxième paramètre ("r" ou "w") :

```
<FP> = fopen ( <Nom> , "w" );
```

ou bien

```
<FP> = fopen ( <Nom> , "r" );
```

* <Nom> est une chaîne de caractères constante ou une variable de type chaîne qui représente le nom du fichier sur le moyen de stockage,

* le deuxième argument détermine le mode d'accès au fichier :

"w" pour 'ouverture en écriture' - write -

"r" pour 'ouverture en lecture' - read -

* <FP> est un pointeur du type **FILE*** qui sera relié au fichier sur le médium de stockage. Dans la suite du programme, il faut utiliser <FP> au lieu de <Nom> pour référencer le fichier.

* <FP> doit être déclaré comme :

```
FILE *FP;
```

Le résultat de fopen :

Si le fichier a pu être ouvert avec succès, **fopen** fournit l'adresse d'un nouveau bloc du type **FILE**. En général, la valeur de cette adresse ne nous intéresse pas; elle est simplement affectée à un pointeur <FP> du type **FILE*** que nous utiliserons ensuite pour accéder au fichier.

A l'apparition d'une *erreur* lors de l'ouverture du fichier, **fopen** fournit la valeur numérique zéro qui est souvent utilisée dans une expression conditionnelle pour assurer que le traitement ne continue pas avec un fichier non ouvert (voir 11.4.3. Exemples).

Ouverture en écriture :

Dans le cas de la création d'un nouveau fichier, le nom du fichier est ajouté au répertoire du médium de stockage et la tête de lecture/écriture est positionnée sur un espace libre du médium.

Si un fichier *existant* est ouvert en écriture, alors son contenu est perdu.

Si un fichier *non existant* est ouvert en écriture, alors il est créé automatiquement. Si la création du fichier est impossible alors **fopen** indique une erreur en retournant la valeur zéro.

Autres possibilités d'erreurs signalées par un résultat nul :

- - chemin d'accès non valide,
- - pas de disque/bande dans le lecteur,
- - essai d'écrire sur un médium protégé contre l'écriture,
- - . . .

Ouverture en lecture :

Dans le cas de la lecture d'un fichier existant, le nom du fichier doit être retrouvé dans le répertoire du médium et la tête de lecture/écriture est placée sur le premier enregistrement de ce fichier.

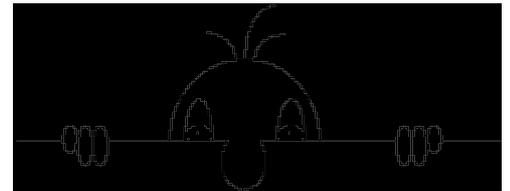
Possibilités d'erreurs signalées par un résultat nul :

- - essai d'ouvrir un fichier non existant,
- - essai d'ouvrir un fichier sans autorisation d'accès,
- - essai d'ouvrir un fichier protégé contre la lecture,
- - ...

Remarque avancée :

Si un fichier n'a pas pu être ouvert avec succès, (résultat NUL), un code d'erreur est placé dans la variable **errno**. Ce code désigne plus exactement la nature de l'erreur survenue. Les codes d'erreurs sont définis dans **<errno.h>**.

- L'appel de la fonction **strerror(errno)** retourne un pointeur sur la chaîne de caractères qui décrit l'erreur dans **errno**.
- L'appel de la fonction **perror(s)** affiche la chaîne **s** et le message d'erreur qui est défini pour l'erreur dans **errno**.



2) Fermer un fichier séquentiel :

Fermer un fichier en langage algorithmique :

```
fermer <Nom>
```

<Nom> est une chaîne de caractères constante ou une variable de type chaîne qui représente le nom du fichier que l'on désire fermer.

Fermer un fichier en langage C :

```
fclose ( <FP> );
```

<FP> est un pointeur du type **FILE*** relié au nom du fichier que l'on désire fermer.

La fonction **fclose** provoque le contraire de **fopen** :

Si le fichier a été ouvert en écriture, alors les données non écrites de la mémoire tampon sont écrites et les données supplémentaires (longueur du fichier, date et heure de sa création) sont ajoutées dans le répertoire du médium de stockage.

Si le fichier a été ouvert en lecture, alors les données non lues de la mémoire tampon sont simplement 'jetées'.

La mémoire tampon est ensuite libérée et la liaison entre le pointeur sur **FILE** et le nom du fichier correspondant est annulée.

Après **fclose()** le pointeur <FP> est invalide. Des erreurs graves pourraient donc survenir si ce pointeur est utilisé par la suite!

3) Exemples : Ouvrir et fermer des fichiers en pratique :

En langage algorithmique, il suffit de simplement ouvrir et fermer un fichier par les commandes respectives :

```
programme PERSONNEL  
chaîne NOM_FICHER  
.  
.  
.  
  
écrire "Entrez le nom du fichier : "  
lire NOM_FICHER  
ouvrir NOM_FICHER en écriture  
(* ou bien *)  
(* ouvrir NOM_FICHER en lecture *)
```

```

. . .
    fermer NOM_FICHIER
fprogramme (* fin PERSONNEL *)

```

En pratique, il faut contrôler si l'ouverture d'un fichier a été accomplie avec succès avant de continuer les traitements. Pour le cas d'une erreur, nous allons envisager deux réactions différentes :

a) Répéter l'essai jusqu'à l'ouverture correcte du fichier :

```

#include <stdio.h>
main()
{
    FILE *P_FICHIER;          /* pointeur sur FILE */
    char NOM_FICHIER[30]; /* nom du fichier */
    . . .

    do
    {
        printf("Entrez le nom du fichier : ");
        scanf("%s", NOM_FICHIER);
        P_FICHIER = fopen(NOM_FICHIER, "w");
        /* ou bien */
        /* P_FICHIER = fopen(NOM_FICHIER, "r"); */
        if (!P_FICHIER)
            printf("\aERREUR : Impossible d'ouvrir "
                "le fichier : %s.\n", NOM_FICHIER);
    }
    while (!P_FICHIER);

    . . .

    fclose(P_FICHIER);
    return (0);
}

```

b) Abandonner le programme en retournant un code d'erreur non nul – exit :

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE *P_FICHIER;          /* pointeur sur FILE */
    char NOM_FICHIER[30]; /* nom du fichier */
    . . .

    printf("Entrez le nom du fichier : ");
    scanf("%s", NOM_FICHIER);
    P_FICHIER = fopen(NOM_FICHIER, "w");
    /* ou bien */
    /* P_FICHIER = fopen(NOM_FICHIER, "r"); */
    if (!P_FICHIER)
    {
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", NOM_FICHIER);
        exit(-1); /* Abandonner le programme en */
    }             /* retournant le code d'erreur -1 */
}

```

```

. . .

fclose(P_FICHIER);
return (0);
}

```

V) Lire et écrire dans des fichiers séquentiels :

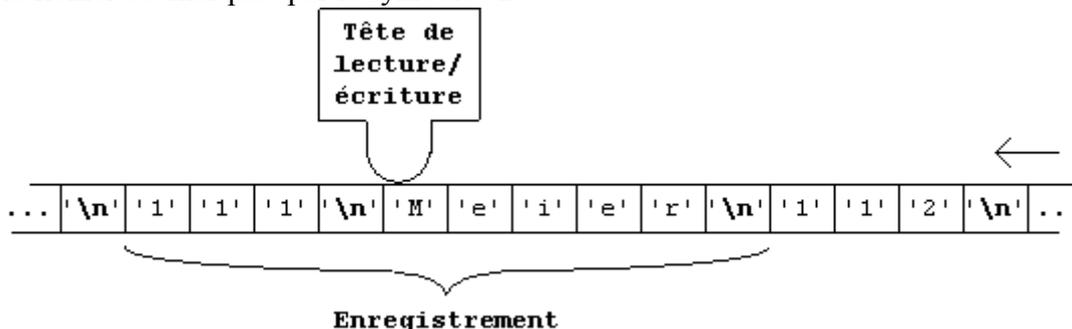
Fichiers texte :

Les fichiers que nous employons dans ce manuel sont des fichiers texte, c'est-à-dire toutes les informations dans les fichiers sont mémorisées sous forme de chaînes de caractères et sont organisées en lignes. Même les valeurs numériques (types **int**, **float**, **double**, ...) sont stockées comme chaînes de caractères.

Pour l'écriture et la lecture des fichiers, nous allons utiliser les fonctions standard **fprintf**, **fscanf**, **fputc** et **fgetc** qui correspondent à **printf**, **scanf**, **putchar** et **getchar** si nous indiquons *stdout* respectivement *stdin* comme fichiers de sortie ou d'entrée.

1) Traitement par enregistrements :

Les fichiers texte sont généralement organisés en lignes, c'est-à-dire la fin d'une information dans le fichier est marquée par le symbole '\n' :



Attention ! :

Pour pouvoir lire correctement les enregistrements dans un fichier séquentiel, le programmeur doit connaître l'ordre des différentes rubriques (champs) à l'intérieur des enregistrements.

a) Ecrire une information dans un fichier séquentiel :

Ecrire dans un fichier séquentiel en langage algorithmique :

```
écrire <Nom>:<Expr1>
```

```
écrire <Nom>:<Expr2>
```

...

```
écrire <Nom>:<ExprN>
```

* <Nom> est une chaîne de caractères constante ou une variable de type chaîne qui représente le nom du fichier dans lequel on veut écrire.

* <Expr1>, <Expr2>, ..., <ExprN> représentent les rubriques qui forment un enregistrement et dont les valeurs respectives sont écrites dans le fichier.

Ecrire dans un fichier séquentiel en langage C - fprintf :

```
fprintf( <FP>, "<Form1>\n", <Expr1>);
```

```
fprintf( <FP>, "<Form2>\n", <Expr2>);
```

...

```
fprintf( <FP>, "<FormN>\n", <ExprN>);
```

ou bien

```
fprintf(<FP>, "<Form1>\n<Form2>\n... \n<FormN>\n", <Expr1>,
<Expr2>, ... , <ExprN>);
```

* <FP> est un pointeur du type FILE* qui est relié au nom du fichier cible.

* <Expr1>, <Expr2>, ... , <ExprN> représentent les rubriques qui forment un enregistrement et dont les valeurs respectives sont écrites dans le fichier.

* <Form1>, <Form2>, ... , <FormN> représentent les spécificateurs de format pour l'écriture des différentes rubriques (voir chapitre 4.3.).

Remarque :

L'instruction

```
fprintf(stdout, "Bonjour\n");
```

est identique à

```
printf("\Bonjour\n");
```

Attention ! :

Notez que **fprintf** (et **printf**) écrit toutes les chaînes de caractères *sans le symbole de fin de chaîne '\0'*. Dans les fichiers texte, il faut ajouter le symbole de fin de ligne '\n' pour séparer les données.

b) Lire une information dans un fichier séquentiel :

Lire dans un fichier séquentiel en langage algorithmique :

```
lire <Nom>:<Var1>
```

```
lire <Nom>:<Var2>
```

```
...
```

```
lire <Nom>:<VarN>
```

<Nom> est une chaîne de caractères constante ou une variable de type chaîne qui représente le nom du fichier duquel on veut lire.

<Var1>, <Var2>, ... , <VarN> représentent les variables qui vont recevoir les valeurs des différentes rubriques d'un enregistrement lu dans le fichier.

Lire dans un fichier séquentiel en langage C - fscanf :

```
fscanf(<FP>, "<Form1>\n", <Adr1>);
```

```
fscanf(<FP>, "<Form2>\n", <Adr2>);
```

```
...
```

```
fscanf(<FP>, "<FormN>\n", <AdrN>);
```

ou bien

```
fscanf(<FP>, "<Form1>\n<Form2>\n... \n<FormN>\n", <Adr1>,
<Adr2>, ... , <AdrN>);
```

<FP> est un pointeur du type FILE* qui est relié au nom du fichier à lire.

<Adr1>, <Adr2>, ... , <AdrN> représentent les adresses des variables qui vont recevoir les valeurs des différentes rubriques d'un enregistrement lu dans le fichier.

<Form1>, <Form2>, ... , <FormN> représentent les spécificateurs de format pour la lecture des différentes rubriques (voir chapitre 4.4.).

Remarque :

L'instruction

```
fscanf(stdin, "%d\n", &N);
```

est identique à

```
scanf("%d\n", &N);
```

Attention !

Pour les fonctions **scanf** et **fscanf** tous les signes d'espacement sont équivalents comme séparateurs. En conséquence, à l'aide de **fscanf**, il nous sera impossible de lire toute une phrase dans laquelle les mots sont séparés par des espaces.

2) Traitement par caractères :

La manipulation de fichiers avec les instructions **fprintf** et **fscanf** n'est pas assez flexible pour manipuler de façon confortable des textes écrits. Il est alors avantageux de traiter le fichier séquentiellement caractère par caractère.

a) Ecrire un caractère dans un fichier séquentiel - fputc :

```
fputc( <C> , <FP> );
```

fputc transfère le caractère indiqué par <C> dans le fichier référencé par <FP> et avance la position de la tête de lecture/écriture au caractère suivant.

<C> représente un caractère (valeur numérique de 0 à 255) ou le symbole de fin de fichier **EOF** (voir 11.5.3.).

<FP> est un pointeur du type **FILE*** qui est relié au nom du fichier cible.

Remarque :

L'instruction

```
fputc('a', stdout);
```

est identique à

```
putchar('a');
```

b) Lire un caractère dans un fichier séquentiel - fgetc :

```
<C> = fgetc( <FP> );
```

fgetc fournit comme résultat le prochain caractère du fichier référencé par <FP> et avance la position de la tête de lecture/écriture au caractère suivant. A la fin du fichier, **fgetc** retourne **EOF** (voir 11.5.3.).

<C> représente une variable du type **int** qui peut accepter une valeur numérique de 0 à 255 ou le symbole de fin de fichier **EOF**.

<FP> est un pointeur du type **FILE*** qui est relié au nom du fichier à lire.

Remarque :

L'instruction

```
C = fgetc(stdin);
```

est identique à

```
C = getchar();
```

3) Détection de la fin d'un fichier séquentiel :

Lors de la fermeture d'un fichier ouvert en écriture, la fin du fichier est marquée automatiquement par le symbole de fin de fichier **EOF** (*End Of File*). Lors de la lecture d'un fichier, les fonctions finfichier(<Nom>) respectivement **feof**(<FP>) nous permettent de détecter la fin du fichier :

Détection de la fin d'un fichier en langage algorithmique :

```
finfichier( <Nom> )
```

finfichier retourne la valeur logique vrai, si la tête de lecture du fichier référencé par <Nom> est arrivée à la fin du fichier; sinon la valeur logique du résultat est faux.

<Nom> est une chaîne de caractères constante ou une variable de type chaîne qui représente le nom du fichier duquel on veut lire.

Détection de la fin d'un fichier en langage C - feof :

```
feof( <FP> );
```

! **feof** retourne une valeur différente de zéro, si la tête de lecture du fichier référencé par <FP> est arrivée à la fin du fichier; sinon la valeur du résultat est zéro.

• <FP> est un pointeur du type **FILE*** qui est relié au nom du fichier à lire.

Pour que la fonction **feof** détecte correctement la fin du fichier, il faut qu'après la lecture de la dernière donnée du fichier, la tête de lecture arrive jusqu'à la position de la marque **EOF**. Nous obtenons cet effet seulement si nous terminons aussi la chaîne de format de **fscanf** par un retour à la ligne '\n' (ou par un autre signe d'espacement).

Exemple :

Une boucle de lecture typique pour lire les enregistrements d'un fichier séquentiel référencé par un pointeur FP peut avoir la forme suivante :

```
while (!feof(FP))
{
    fscanf(FP, "%s\n ... \n", NOM, ... );
    . . .
}
```

Exemple :

Le programme suivant lit et affiche le fichier "C:\AUTOEXEC.BAT" en le parcourant caractère par caractère :

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE *FP;
    FP = fopen("C:\\\\AUTOEXEC.BAT", "r");
    if (!FP)
    {
        printf("Impossible d'ouvrir le fichier\n");
        exit(-1);
    }
    while (!feof(FP))
        putchar(fgetc(FP));
    fclose(FP);
    return (0);
}
```

Dans une chaîne de caractères constante, il faut indiquer le symbole '\' (back-slash) par '\\', pour qu'il ne soit pas confondu avec le début d'une séquence d'échappement (p.ex : \n, \t, \a, ...).

VI) Résumé sur les fichiers :

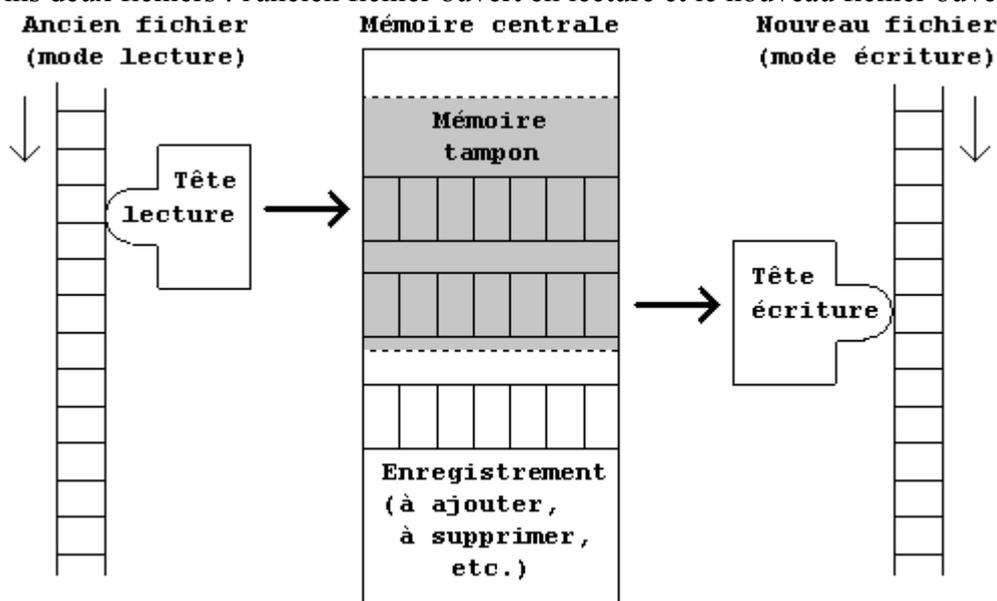
	Langage algorithmique	C
Ouverture en écriture	ouvrir <Nom> en écriture	<FP> = fopen(<Nom>,"w");
Ouverture en lecture	ouvrir <Nom> en lecture	<FP> = fopen(<Nom>,"r");
Fermeture	fermer <Nom>	fclose(<FP>);
Fonction fin de fichier	finfichier(<Nom>)	feof(<FP>)
Ecriture	écrire <Nom>:<Exp>	fprintf(<FP>,"...",<Adr>); fputc(<C>,<FP>);
Lecture	lire <Nom>:<Var>	fscanf(<FP>,"...",<Adr>); <C> = fgetc(<FP>);

VII) Mise à jour d'un fichier séquentiel en C :

Dans ce chapitre, nous allons résoudre les problèmes standards sur les fichiers, à savoir :
* l'ajoute d'un enregistrement à un fichier

- * la suppression d'un enregistrement dans un fichier
- * la modification d'un enregistrement dans un fichier

Comme il est impossible de lire et d'écrire en même temps dans un fichier séquentiel, les modifications doivent se faire à l'aide d'un fichier supplémentaire. Nous travaillons donc typiquement avec au moins deux fichiers : l'ancien fichier ouvert en lecture et le nouveau fichier ouvert en écriture :

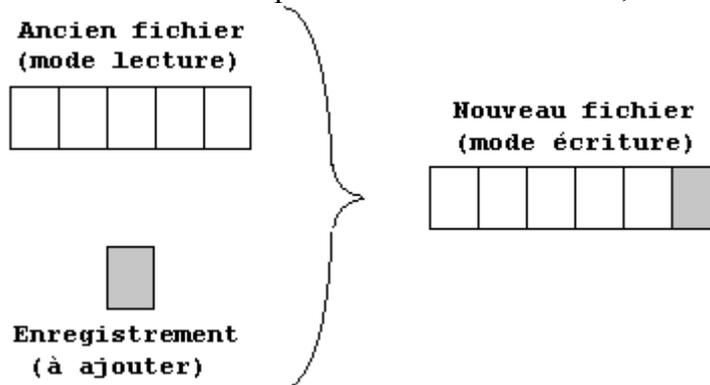


1) Ajouter un enregistrement à un fichier :

Nous pouvons ajouter le nouvel enregistrement à différentes positions dans le fichier :

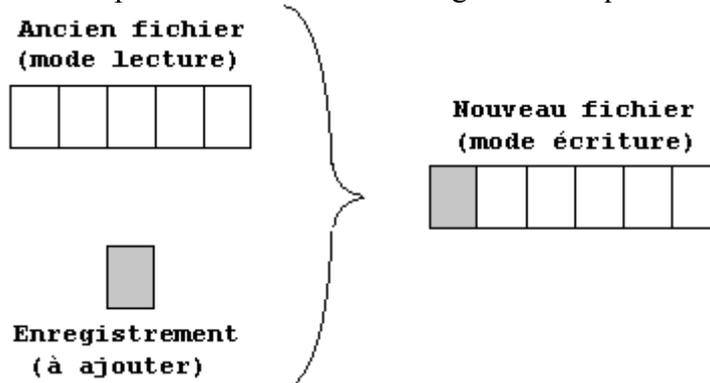
a) Ajoute à la fin du fichier :

L'ancien fichier est entièrement copié dans le nouveau fichier, suivi du nouvel enregistrement.



b) Ajoute au début du fichier :

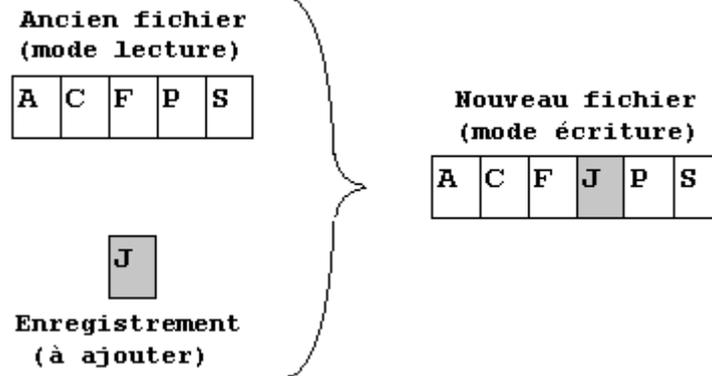
L'ancien fichier est copié derrière le nouvel enregistrement qui est écrit en premier lieu.



c) Insertion dans un fichier trié relativement à une rubrique commune des enregistrements :

Le nouveau fichier est créé en trois étapes :

- copier les enregistrements de l'ancien fichier qui précèdent le nouvel enregistrement,
- écrire le nouvel enregistrement,
- copier le reste des enregistrements de l'ancien fichier.



Le programme suivant effectue l'insertion d'un enregistrement à introduire au clavier dans un fichier trié selon la seule rubrique de ses enregistrements : le nom d'une personne. Le programme inclut en même temps les solutions aux deux problèmes précédents. La comparaison lexicographique des noms des personnes se fait à l'aide de la fonction **strcmp**.

Solution en C :

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

main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[30], NOUVEAU[30];
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char NOM_PERS[30], NOM_AJOUT[30];
    int TROUVE;

    /* Ouverture de l'ancien fichier en lecture */
    do
    {
        printf("Nom de l'ancien fichier : ");
        scanf("%s", ANCIEN);
        INFILE = fopen(ANCIEN, "r");
        if (!INFILE)
            printf("\aERREUR : Impossible d'ouvrir "
                "le fichier : %s.\n", ANCIEN);
    }
    while (!INFILE);
    /* Ouverture du nouveau fichier en écriture */
    do
    {
        printf("Nom du nouveau fichier : ");
        scanf("%s", NOUVEAU);
        OUTFILE = fopen(NOUVEAU, "w");
        if (!OUTFILE)
            printf("\aERREUR : Impossible d'ouvrir "
```

```

        "le fichier : %s.\n", NOUVEAU);
    }
    while (!OUTFILE);
    /* Saisie de l'enregistrement à insérer */
    printf("Enregistrement à insérer : ");
    scanf("%s",NOM_AJOUT);

    /* Traitement */
    TROUVE = 0;
    /* Copie des enregistrements dont le nom */
    /* précède lexicogr. celui à insérer.*/
    while (!feof(INFILE) && !TROUVE)
    {
        fscanf(INFILE, "%s\n", NOM_PERS);
        if (strcmp(NOM_PERS, NOM_AJOUT) > 0)
            TROUVE = 1;
        else
            fprintf(OUTFILE, "%s\n", NOM_PERS);
    }
    /* Ecriture du nouvel enregistrement, */
    fprintf(OUTFILE, "%s\n", NOM_AJOUT);
    /* suivi du dernier enregistrement lu. */
    if (TROUVE) fprintf(OUTFILE, "%s\n", NOM_PERS);
    /* Copie du reste des enregistrements */
    while (!feof(INFILE))
    {
        fscanf(INFILE, "%s\n", NOM_PERS);
        fprintf(OUTFILE, "%s\n", NOM_PERS);
    }
    /* Fermeture des fichiers */
    fclose(OUTFILE);
    fclose(INFILE);
    return (0);
}

```

2) Supprimer un enregistrement dans un fichier :

Le nouveau fichier est créé en copiant tous les enregistrements de l'ancien fichier qui précèdent l'enregistrement à supprimer et tous ceux qui le suivent :

Solution en C :

```

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

main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[30], NOUVEAU[30];
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char NOM_PERS[30], NOM_SUPPR[30];
    /* Ouverture de l'ancien fichier en lecture */
    do
    {

```

```

    printf("Nom de l'ancien fichier : ");
    scanf("%s", ANCIEN);
    INFILE = fopen(ANCIEN, "r");
    if (!INFILE)
        printf("\aERREUR : Impossible d'ouvrir "
               "le fichier : %s.\n", ANCIEN);
}
while (!INFILE);
/* Ouverture du nouveau fichier en écriture */
do
{
    printf("Nom du nouveau fichier : ");
    scanf("%s", NOUVEAU);
    OUTFILE = fopen(NOUVEAU, "w");
    if (!OUTFILE)
        printf("\aERREUR : Impossible d'ouvrir "
               "le fichier : %s.\n", NOUVEAU);
}
while (!OUTFILE);
/* Saisie de l'enregistrement à supprimer */
printf("Enregistrement à supprimer : ");
scanf("%s", NOM_SUPPR);
/* Traitement */
/* Copie de tous les enregistrements à */
/* l'exception de celui à supprimer. */
while (!feof(INFILE))
{
    fscanf(INFILE, "%s\n", NOM_PERS);
    if (strcmp(NOM_PERS, NOM_SUPPR) != 0)
        fprintf(OUTFILE, "%s\n", NOM_PERS);
}
/* Fermeture des fichiers */
fclose(OUTFILE);
fclose(INFILE);
return (0);
}

```

3) Modifier un enregistrement dans un fichier :

Le nouveau fichier est créé de tous les enregistrements de l'ancien fichier qui précèdent l'enregistrement à modifier, de l'enregistrement modifié et de tous les enregistrements qui suivent l'enregistrement à modifier dans l'ancien fichier :

Solution en C :

```

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

main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[30], NOUVEAU[30];
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char NOM_PERS[30], NOM_MODIF[30], NOM_NOUV[30];

```

```

/* Ouverture de l'ancien fichier en lecture */
do
{
printf("Nom de l'ancien fichier : ");
scanf("%s", ANCIEN);
INFILE = fopen(ANCIEN, "r");
if (!INFILE)
printf("\aERREUR : Impossible d'ouvrir "
"le fichier : %s.\n", ANCIEN);
}
while (!INFILE);

/* Ouverture du nouveau fichier en écriture */
do
{
printf("Nom du nouveau fichier : ");
scanf("%s", NOUVEAU);
OUTFILE = fopen(NOUVEAU, "w");
if (!OUTFILE)
printf("\aERREUR : Impossible d'ouvrir "
"le fichier : %s.\n", NOUVEAU);
}
while (!OUTFILE);
/* Saisie de l'enregistrement à modifier, */
printf("Enregistrement à modifier : ");
scanf("%s", NOM_MODIF);
/* et de sa nouvelle valeur. */
printf("Enregistrement nouveau : ");
scanf("%s", NOM_NOUV);
/* Traitement */
/* Copie de tous les enregistrements en */
/* remplaçant l'enregistrement à modifier */
/* par sa nouvelle valeur. */
while (!feof(INFILE))
{
fscanf(INFILE, "%s\n", NOM_PERS);
if (strcmp(NOM_PERS, NOM_MODIF) = 0)
fprintf(OUTFILE, "%s\n", NOM_NOUV);
else
fprintf(OUTFILE, "%s\n", NOM_PERS);
}
/* Fermeture des fichiers */
fclose(OUTFILE);
fclose(INFILE);
return (0);
}

```

VIII) Exercices d'application :

Exercice 11.1 :

Créer sur disquette puis afficher à l'écran le fichier INFORM.TXT dont les informations sont structurées de la manière suivante :

Numéro de matricule (entier)

Nom (chaîne de caractères)

Prénom (chaîne de caractères)

Le nombre d'enregistrements à créer est à entrer au clavier par l'utilisateur.

Exercice 11.2 :

Ecrire un programme qui crée sur disquette un fichier INFBIS.TXT qui est la copie exacte (enregistrement par enregistrement) du fichier INFORM.TXT.

Exercice 11.3 :

Ajouter un nouvel enregistrement (entré au clavier) à la fin de INFORM.TXT et sauver le nouveau fichier sous le nom INFBIS.TXT.

Exercice 11.4 :

Insérer un nouvel enregistrement dans INFORM.TXT en supposant que le fichier est trié relativement à la rubrique NOM et sauver le nouveau fichier sous le nom INFBIS.TXT.

Exercice 11.5 :

Supprimer dans INFORM.TXT tous les enregistrements :

a) dont le numéro de matricule se termine par 8

b) dont le prénom est "Paul" (utiliser **strempr**)

c) dont le nom est un palindrome. Définir une fonction d'aide PALI qui fournit le résultat 1 si la chaîne transmise comme paramètre est un palindrome, sinon la valeur zéro.

Sauver le nouveau fichier à chaque fois sous le nom INFBIS.TXT.

Exercice 11.6 :

Créer sur disquette puis afficher à l'écran le fichier FAMILLE.TXT dont les informations sont structurées de la manière suivante :

Nom de famille

Prénom du père

Prénom de la mère

Nombre d'enfants

Prénoms des enfants

Le nombre d'enregistrements à créer est entré au clavier.

Attention :

Le nombre de rubriques des enregistrements varie avec le nombre d'enfants !

Exercice 11.7 :

Ecrire un programme qui crée sur disquette le fichier MOTS.TXT contenant une série de 50 mots au maximum (longueur maximale d'un mot : 50 caractères). La saisie des mots se terminera à l'introduction du symbole '*' qui ne sera pas écrit dans le fichier.

Exercice 11.8 :

Ecrire un programme qui affiche le nombre de mots, le nombre de palindromes ainsi que la longueur moyenne des mots contenus dans le fichier MOTS.TXT. Utiliser les deux fonctions d'aide PALI et LONG_CH définies au chapitre 10.

Exercice 11.9 :

Ecrire un programme qui charge les mots du fichier MOTS.TXT dans la mémoire centrale, les trie d'après la méthode par propagation (méthode de la bulle - voir exercice 7.15) et les écrit dans un deuxième fichier MOTS_TRI.TXT sur la disquette. Les mots seront mémorisés à l'aide d'un tableau de pointeurs sur **char** et la mémoire nécessaire sera réservée de façon dynamique.

Exercice 11.10 :

A l'aide d'un éditeur de textes, créer un fichier NOMBRES.TXT qui contient une liste de nombres entiers. Dans le fichier, chaque nombre doit être suivi par un retour à la ligne. Ecrire un programme qui affiche les nombres du fichier, leur somme et leur moyenne.

Exercice 11.11 :

Ecrire un programme qui remplace, dans un fichier contenant un texte, les retours à la ligne par des espaces. Si plusieurs retours à la ligne se suivent, seulement le premier sera remplacé. Les noms des fichiers source et destination sont entrés au clavier.

Exercice 11.12 :

Ecrire un programme qui détermine dans un fichier un texte dont le nom est entré au clavier, le nombre de phrases terminées par un point, un point d'interrogation ou un point d'exclamation.

Utiliser une fonction d'aide FIN_PHRASE qui décide si un caractère transmis comme paramètre est un des séparateurs mentionnés ci-dessus. FIN_PHRASE retourne la valeur (logique) 1 si le caractère est égal à '.', '!' ou '?' et 0 dans le cas contraire.

Exercice 11.13 :

Ecrire un programme qui détermine dans un fichier un texte dont le nom est entré au clavier :

- le nombre de caractères qu'il contient,
- le nombre de chacune des lettres de l'alphabet (sans distinguer les majuscules et les minuscules),
- le nombre de mots,
- le nombre de paragraphes (c'est-à-dire : des retours à la ligne),

Les retours à la ligne ne devront pas être comptabilisés dans les caractères. On admettra que deux mots sont toujours séparés par un ou plusieurs des caractères suivants :

- fin de ligne
- espace
- ponctuation : . : , ; ? !
- parenthèses : ()
- guillemets : "
- apostrophe : '

Utiliser une fonction d'aide SEPA qui décide si un caractère transmis comme paramètre est l'un des séparateurs mentionnés ci-dessus. SEPA restituera la valeur (logique) 1 si le caractère est un séparateur et 0 dans le cas contraire. SEPA utilise un tableau qui contient les séparateurs à détecter.

Exemple :

Nom du fichier texte : A:LITTERA.TXT
Votre fichier contient:
12 paragraphes
571 mots
4186 caractères
dont
279 fois la lettre a
56 fois la lettre b
. . .
3 fois la lettre z
et 470 autres caractères

Exercice 11.14 :

Ecrire un programme qui affiche le contenu d'un fichier texte sur un écran de 25 lignes et 80 colonnes en attendant la confirmation de l'utilisateur (par 'Enter') après chaque page d'écran. Utiliser la fonction **getchar**.

Exercice 11.15 :

Ecrire un programme qui vérifie la validité d'une série de numéros de CCP mémorisés dans un fichier. Un numéro de CCP est composé de trois parties : un numéro de compte, un séparateur '-' et un numéro de contrôle. Un numéro de CCP est correct :

- si le reste de la division entière de la valeur devant le séparateur '-' par 97 est différent de zéro et égal à la valeur de contrôle.
- si le reste de la division par 97 est zéro et la valeur de contrôle est 97.

Exemple :

Nombre de CCP 15742-28 :	15742 modulo 97 = 28	correct
Nombre de CCP 72270-5 :	72270 modulo 97 = 5	correct
Nombre de CCP 22610-10 :	22610 modulo 97 = 9	incorrect
Nombre de CCP 50537-0 :	50537 modulo 97 = 0	
	nombre incorrect, car la valeur de contrôle devrait être 97.	
Nombre de CCP 50537-97 :	50537 modulo 97 = 0	correct

Utiliser une fonction CCP_TEST qui obtient comme paramètres les deux parties numériques d'un nombre de CCP et qui affiche alors un message indiquant si le numéro de CCP est valide ou non.

Pour tester le programme, créer à l'aide d'un éditeur de texte un fichier CCP.TXT qui contient les numéros ci-dessus, suivis par des retours à la ligne.

Exercice 11.16 :

Deux fichiers FA et FB dont les noms sont à entrer au clavier contiennent des nombres entiers triés dans l'ordre croissant. Ecrire un programme qui copie le contenu de FA et FB respectivement dans les tableaux TABA et TABB dans la mémoire centrale. Les tableaux TABA et TABB sont fusionnés dans un troisième tableau trié en ordre croissant TABC. Après la fusion, la tableau TABC est sauvé dans un fichier FC dont le nom est à entrer au clavier.

La mémoire pour TABA, TABB et TFUS dont les nombres d'éléments sont inconnus, est réservée dynamiquement après que les longueurs des fichiers FA et FB ont été détectées.

IX) Solutions des exercices du Chapitre 11 : LES FICHIERS SEQUENTIELS :

Exercice 11.1 :

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    /* Déclarations : */
    /* Nom du fichier et pointeur de référence */
    char NOM_FICH[] = "A:\\\\INFORM.TXT";
    FILE *FICHIER;
    /* Autres variables */
    char NOM[30], PRENOM[30];
    int MATRICULE;
    int I, N_ENR;

    /* Ouverture du nouveau fichier en écriture */
    FICHIER = fopen(NOM_FICH, "w");
    if (!FICHIER)
    {
        printf("\aERREUR : Impossible d'ouvrir "
               "le fichier : %s.\n", NOM_FICH);
        exit(-1);
    }
    /* Saisie des données et création du fichier */
    printf("*** Création du fichier %s ***\n", NOM_FICH);
    printf("Nombre d'enregistrements à créer : ");
    scanf("%d", &N_ENR);
    for (I=1; I<=N_ENR; I++)
    {
        printf("Enregistrement No: %d \n", I);
        printf("Numéro de matricule : ");
        scanf("%d", &MATRICULE);
        printf("Nom      : ");
        scanf("%s", NOM);
        printf("Prénom : ");
        scanf("%s", PRENOM);
        fprintf(FICHIER, "%d\n%s\n%s\n", MATRICULE, NOM,
PRENOM);
    }
    /* Fermeture du fichier */
    fclose(FICHIER);

    /* Ouverture du fichier en lecture */
    FICHIER = fopen(NOM_FICH, "r");
    if (!FICHIER)
    {
        printf("\aERREUR : Impossible d'ouvrir "
               "le fichier : %s.\n", NOM_FICH);
        exit(-1);
    }
    /* Affichage du fichier */
}
```

```

printf("*** Contenu du fichier  %s ***\n", NOM_FICH);
while (!feof(FICHIER))
{
    fscanf(FICHIER,    "%d\n%s\n%s\n",    &MATRICULE,    NOM,
PRENOM);
    printf("Matricule : %d\t", MATRICULE);
    printf("Nom et prénom : %s %s\n", NOM, PRENOM);
}
/* Fermeture du fichier */
fclose(FICHIER);
return (0);
}

```

Exercice 11.2 :

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[] = "A:\\INFORM.TXT";
    char NOUVEAU[] = "A:\\INFBIS.TXT";
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char NOM[30], PRENOM[30];
    int MATRICULE;

    /* Ouverture de l'ancien fichier en lecture */
    INFILE = fopen(ANCIEN, "r");
    if (!INFILE)
    {
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", ANCIEN);
        exit(-1);
    }
    /* Ouverture du nouveau fichier en écriture */
    OUTFILE = fopen(NOUVEAU, "w");
    if (!OUTFILE)
    {
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", NOUVEAU);
        exit(-1);
    }

    /* Copie de tous les enregistrements */
    while (!feof(INFILE))
    {
        fscanf    (INFILE,    "%d\n%s\n%s\n",    &MATRICULE,    NOM,
PRENOM);
        fprintf(OUTFILE,    "%d\n%s\n%s\n",    MATRICULE,    NOM,
PRENOM);
    }
    /* Fermeture des fichiers */

```

```

fclose(OUTFILE);
fclose(INFILE);
return (0);
}

```

Exercice 11.3 :

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[] = "A:\\\\INFORM.TXT";
    char NOUVEAU[] = "A:\\\\INFBIS.TXT";
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char NOM[30], PRENOM[30];

    int MATRICULE;
    char NOM_NOUV[30], PRE_NOUV[30];
    int MATRI_NOUV;
    /* Ouverture de l'ancien fichier en lecture */
    INFILE = fopen(ANCIEN, "r");
    if (!INFILE)
    {
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", ANCIEN);
        exit(-1);
    }
    /* Ouverture du nouveau fichier en écriture */
    OUTFILE = fopen(NOUVEAU, "w");
    if (!OUTFILE)
    {
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", NOUVEAU);
        exit(-1);
    }

    /* Saisie de l'enregistrement à ajouter */
    printf("Enregistrement à ajouter : \n");
    printf("Numéro de matricule : ");
    scanf("%d",&MATRI_NOUV);
    printf("Nom      : ");
    scanf("%s",NOM_NOUV);
    printf("Prénom : ");
    scanf("%s",PRE_NOUV);
    /* Copie des enregistrements de l'ancien fichier */
    while (!feof(INFILE))
    {
        fscanf (INFILE, "%d\n%s\n%s\n", &MATRICULE, NOM,
PRENOM);
        fprintf(OUTFILE, "%d\n%s\n%s\n", MATRICULE, NOM,
PRENOM);
    }
}

```

```

    }
    /* Ecriture du nouvel enregistrement à la fin du fichier */
    fprintf(OUTFILE, "%d\n%s\n%s\n", MATRI_NOUV, NOM_NOUV, PRE_NOUV);
    /* Fermeture des fichiers */
    fclose(OUTFILE);
    fclose(INFILE);
    return (0);
}

```

Exercice 11.4 :

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[] = "A:\\INFORM.TXT";
    char NOUVEAU[] = "A:\\INFBIS.TXT";
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char NOM[30], PRENOM[30];
    int MATRICULE;
    char NOM_NOUV[30], PRE_NOUV[30];
    int MATRI_NOUV;
    int TROUVE;

    /* Ouverture de l'ancien fichier en lecture */
    INFILE = fopen(ANCIEN, "r");
    if (!INFILE)
    {
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", ANCIEN);
        exit(-1);
    }

    /* Ouverture du nouveau fichier en écriture */
    OUTFILE = fopen(NOUVEAU, "w");
    if (!OUTFILE)
    {
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", NOUVEAU);
        exit(-1);
    }

    /* Saisie de l'enregistrement à insérer */
    printf("Enregistrement à ajouter : \n");
    printf("Numéro de matricule : ");
    scanf("%d", &MATRI_NOUV);
    printf("Nom      : ");
    scanf("%s", NOM_NOUV);
    printf("Prénom : ");

```

```

scanf ("%s", PRE_NOUV);

/* Copie des enregistrements dont le nom */
/* précède lexicogr. celui à insérer. */
TROUVE = 0;
while (!feof(INFILE) && !TROUVE)
{
    fscanf (INFILE, "%d\n%s\n%s\n", &MATRICULE, NOM,
PRENOM);
    if (strcmp(NOM, NOM_NOUV) > 0)
        TROUVE = 1;
    else
        fprintf(OUTFILE, "%d\n%s\n%s\n",
MATRICULE, NOM, PRENOM);
}

/* Ecriture du nouvel enregistrement, */

fprintf(OUTFILE, "%d\n%s\n%s\n", MATRI_NOUV, NOM_NOUV, PRE_NOUV);
/* et du dernier enregistrement lu. */
if (TROUVE)
    fprintf(OUTFILE, "%d\n%s\n%s\n", MATRICULE, NOM,
PRENOM);

/* Copie du reste des enregistrements */
while (!feof(INFILE))
{
    fscanf (INFILE, "%d\n%s\n%s\n", &MATRICULE, NOM,
PRENOM);
    fprintf(OUTFILE, "%d\n%s\n%s\n", MATRICULE, NOM,
PRENOM);
}

/* Fermeture des fichiers */
fclose(OUTFILE);
fclose(INFILE);
return (0);
}

```

Exercice 11.5 :

- a) Supprimer les enregistrements, dont le numéro de matricule se termine par 8

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[] = "A:\\INFORM.TXT";
    char NOUVEAU[] = "A:\\INFBIS.TXT";
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char NOM[30], PRENOM[30];
    int MATRICULE;

```

```

/* Ouverture de l'ancien fichier en lecture */
INFILE = fopen(ANCIEN, "r");
if (!INFILE)
{
    printf("\aERREUR : Impossible d'ouvrir "
           "le fichier : %s.\n", ANCIEN);
    exit(-1);
}
/* Ouverture du nouveau fichier en écriture */
OUTFILE = fopen(NOUVEAU, "w");
if (!OUTFILE)
{
    printf("\aERREUR : Impossible d'ouvrir "
           "le fichier : %s.\n", NOUVEAU);
    exit(-1);
}
/* Copie de tous les enregistrements à l'exception */
/* de ceux dont le numéro de matricule se termine */
/* par 8. */
while (!feof(INFILE))
{
    fscanf (INFILE, "%d\n%s\n%s\n", &MATRICULE, NOM,
PRENOM);
    if (MATRICULE%10 != 8)
        fprintf(OUTFILE, "%d\n%s\n%s\n",
MATRICULE, NOM, PRENOM);
}
/* Fermeture des fichiers */
fclose(OUTFILE);
fclose(INFILE);
return (0);
}

```

b) Supprimer les enregistrements, dont le prénom est "Paul" (utiliser strcmp)

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
    /* Déclarations */
    . . .
    /* Ouverture de l'ancien fichier en lecture */
    . . .
    /* Ouverture du nouveau fichier en écriture */
    . . .
    /* Copie de tous les enregistrements à l'exception */
    /* de ceux dont le prénom est 'Paul'. */
    while (!feof(INFILE))
    {
        fscanf (INFILE, "%d\n%s\n%s\n", &MATRICULE, NOM,
PRENOM);
        if (strcmp(PRENOM, "Paul") != 0)
            fprintf(OUTFILE, "%d\n%s\n%s\n",
MATRICULE, NOM, PRENOM);
    }
}

```

```

    }
    /* Fermeture des fichiers */
    . . .
}

```

c) Supprimer les enregistrements, dont le nom est un palindrome. Définir une fonction d'aide PALI qui fournit le résultat 1 si la chaîne transmise comme paramètre est un palindrome, sinon la valeur zéro.

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    /* Prototype de la fonction PALI */
    int PALI(char *CH);
    /* Déclarations */
    . . .
    /* Ouverture de l'ancien fichier en lecture */
    . . .
    /* Ouverture du nouveau fichier en écriture */
    . . .
    /* Copie de tous les enregistrements à l'exception */
    /* des palindromes. */
    while (!feof(INFILE))
    {
        fscanf (INFILE, "%d\n%s\n%s\n", &MATRICULE, NOM,
PRENOM);
        if (!PALI(NOM))
            fprintf(OUTFILE, "%d\n%s\n%s\n",
MATRICULE, NOM, PRENOM);
    }
    /* Fermeture des fichiers */
    . . .
}

int PALI(char *CH)
{
    /* Variable locale */
    char *CH2;
    /* Placer CH2 à la fin de la chaîne */
    for (CH2=CH; *CH2; CH2++)
        ;
    CH2--;
    /* Contrôler si la chaîne désignée par CH est un palindrome
    */
    for (; CH<CH2; CH++,CH2--)
        if (*CH != *CH2) return (0);
    return 1;
}

```

Exercice 11.6 :

```

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

```

```

main()
{
    /* Déclarations : */
    /* Nom du fichier et pointeur de référence */
    char NOM_FICH[] = "A:\\FAMILLE.TXT";
    FILE *FICHIER;
    /* Autres variables */
    char NOM[30], PERE[30], MERE[30], ENFANT[30];
    int J, N_ENFANTS;
    int I, N_ENR;

    /* Ouverture du nouveau fichier en écriture */
    FICHIER = fopen(NOM_FICH, "w");
    if (!FICHIER)
    {
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", NOM_FICH);
        exit(-1);
    }

    /* Saisie des données et création du fichier */
    printf("*** Création du fichier %s ***\n", NOM_FICH);
    printf("Nombre d'enregistrements à créer : ");
    scanf("%d", &N_ENR);
    for (I=1; I<=N_ENR; I++)
    {
        printf("Enregistrement No: %d \n", I);
        printf("Nom de famille : ");
        scanf("%s", NOM);
        printf("Prénom du père : ");
        scanf("%s", PERE);
        printf("Prénom de la mère : ");
        scanf("%s", MERE);
        printf("Nombre d'enfants : ");
        scanf("%d", &N_ENFANTS);
        fprintf(FICHIER, "%s\n%s\n%s\n%d\n",
            NOM, PERE, MERE,
N_ENFANTS);
        for (J=1; J<=N_ENFANTS; J++)
        {
            printf("Prénom %d. enfant : ", J);
            scanf("%s", ENFANT);
            fprintf(FICHIER, "%s\n", ENFANT);
        }
    }
    /* Fermeture du fichier */
    fclose(FICHIER);
    /* Réouverture du fichier */
    FICHIER = fopen(NOM_FICH, "r");
    if (!FICHIER)
    {
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", NOM_FICH);
        exit(-1);
    }
}

```

```

    }
    /* Affichage du fichier */
    printf("*** Contenu du fichier  %s ***\n", NOM_FICH);
    while (!feof(FICHIER))
    {
        fscanf (FICHIER, "%s\n%s\n%s\n%d\n",
                NOM,          PERE,          MERE,
        &N_ENFANTS);
        printf("\n");
        printf("Nom de famille : %s\n", NOM);
        printf("Nom du père   : %s %s\n", PERE, NOM);
        printf("Nom de la mère : %s %s\n", MERE, NOM);
        printf("Noms des enfants : \n", N_ENFANTS);
        for (J=1; J<=N_ENFANTS; J++)
        {
            fscanf(FICHIER, "%s\n", ENFANT);
            printf("\t%d. : %s %s\n", J, ENFANT, NOM);
        }
    }
    /* Fermeture du fichier */
    fclose(FICHIER);
    return (0);
}

```

Exercice 11.7 :

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    /* Déclarations : */
    /* Nom du fichier et pointeur de référence */
    char NOM_FICH[] = "A:\\MOTS.TXT";
    FILE *FICHIER;
    /* Autres variables */
    char CHAINE[50];

    /* Ouverture du nouveau fichier en écriture */
    FICHIER = fopen(NOM_FICH, "w");
    if (!FICHIER)
    {
        printf("\aERREUR : Impossible d'ouvrir "
               "le fichier : %s.\n", NOM_FICH);
        exit(-1);
    }
    /* Saisie des données et création du fichier */
    printf("*** Création du fichier %s ***\n", NOM_FICH);
    do
    {
        printf("Entrez un mot ('*' pour finir) : ");
        scanf("%s", CHAINE);
        if (CHAINE[0] != '*')
            fprintf(FICHIER, "%s\n", CHAINE);
    }
}

```

```

while (CHAINE[0] != '*');
/* Fermeture du fichier */
fclose(FICHIER);
return (0);
}

```

Exercice 11.8 :

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    /* Prototypes des fonctions appelées */
    int PALI(char *CH);
    int LONG_CH(char *CH);
    /* Déclarations : */
    /* Nom du fichier et pointeur de référence */
    char NOM_FICH[] = "A:\\MOTS.TXT";
    FILE *FICHIER;
    /* Autres variables */
    char CHAINE[50];
    int N_PALI; /* nombre de palindromes */
    int N_MOTS; /* nombre de mots */
    int L_TOT; /* longueur totale de tous les mots */

    /* Ouverture du nouveau fichier en écriture */
    FICHIER = fopen(NOM_FICH, "r");
    if (!FICHIER)
    {
        printf("\aERREUR : Impossible d'ouvrir "
               "le fichier : %s.\n", NOM_FICH);
        exit(-1);
    }
    /* Compter les palindromes et accumuler */
    /* les longueurs des mots. */
    L_TOT = 0;
    N_PALI = 0;
    N_MOTS = 0;
    while (!feof(FICHIER))
    {
        fscanf(FICHIER, "%s\n", CHAINE);
        N_MOTS++;
        L_TOT += LONG_CH(CHAINE);
        N_PALI += PALI(CHAINE);
    }
    /* Fermeture du fichier */
    fclose(FICHIER);
    /* Affichage des résultats */
    printf("Le fichier %s contient :\n", NOM_FICH);
    printf("\t%d \tmots d'une longueur moyenne de :\n",
N_MOTS);
    printf("\t%.1f \tcaractères et\n", (float)L_TOT/N_MOTS);
    printf("\t%d \tpalindromes\n", N_PALI);
    return (0);
}

```

```

}

int PALI(char *CH)
{
    /* Variable locale */
    char *CH2;
    /* Placer CH2 à la fin de la chaîne */
    for (CH2=CH; *CH2; CH2++)
        ;
    CH2--;
    /* Contrôler si la chaîne désignée par CH est un palindrome
    */
    for (; CH<CH2; CH++,CH2--)
        if (*CH != *CH2) return (0);
    return 1;
}

int LONG_CH(char *CH)
{
    char *P;
    for (P=CH ; *P; P++) ;
    return P-CH;
}

```

Exercice 11.9 :

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[] = "A:\\MOTS.TXT";
    char NOUVEAU[] = "A:\\MOTS_TRI.TXT";
    FILE *INFILE, *OUTFILE;
    /* Tableau de pointeurs */
    char *TAB[50];
    /* Autres variables */
    char CHAINE[50];
    char *AIDE; /* pour la permutation */
    int N_MOTS; /* nombre de mots du fichier */
    int I; /* ligne à partir de laquelle TAB est trié */
    int J; /* indice courant */
    int FIN; /* ligne où la dernière permutation a eu lieu */
    /* permet de ne pas trier un sous-ensemble déjà trié */

    /* Ouverture de l'ancien fichier en lecture */
    INFILE = fopen(ANCIEN, "r");
    if (!INFILE)
    {

```

```

        printf("\aERREUR : Impossible d'ouvrir "
               "le fichier : %s.\n", ANCIEN);
        exit(-1);
    }
    /* Initialisation du du compteur des mots */
    N_MOTS = 0;
    /* Lecture du fichier dans la mémoire centrale */
    while (!feof(INFILE))
    {
        fscanf (INFILE, "%s\n", CHAINE);
        /* Réserve de la mémoire */
        TAB[N_MOTS] = malloc(strlen(CHAINE)+1);
        if (TAB[N_MOTS])
            strcpy(TAB[N_MOTS], CHAINE);
        else
        {
            printf("\aPas assez de mémoire \n");
            exit(-1);
        }
        N_MOTS++;
    }
    /* Fermeture du fichier */
    fclose(INFILE);
    /* Tri du tableau par propagation de l'élément maximal. */
    for (I=N_MOTS-1 ; I>0 ; I=FIN)
    {
        FIN=0;
        for (J=0; J<I; J++)
            if (strcmp(TAB[J],TAB[J+1])>0)
            {
                FIN=J;
                AIDE = TAB[J];
                TAB[J] = TAB[J+1];
                TAB[J+1] = AIDE;
            }
    }
    /* Ouverture du nouveau fichier en écriture */
    OUTFILE = fopen(NOUVEAU, "w");
    if (!OUTFILE)
    {
        printf("\aERREUR : Impossible d'ouvrir "
               "le fichier : %s.\n", NOUVEAU);
        exit(-1);
    }
    /* Copie du tableau dans le nouveau fichier */
    for (I=0; I<N_MOTS; I++)
        fprintf(OUTFILE, "%s\n", TAB[I]);
    /* Fermeture du fichier */
    fclose(OUTFILE);
    return (0);
}

```

Exercice 11.10 :

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char NOM_FICH[] = "A:\\NOMBRES.TXT";
    FILE *FICHIER;
    /* Autres variables */
    int NOMBRE; /* nombre actuel lu dans le fichier */
    int N;      /* compteur des nombres */
    long SOMME; /* somme des nombres */

    /* Ouverture de l'ancien fichier en lecture */
    FICHIER = fopen(NOM_FICH, "r");
    if (!FICHIER)
    {
        printf("\aERREUR : Impossible d'ouvrir "
               "le fichier : %s.\n", NOM_FICH);
        exit(-1);
    }
    /* Lecture du fichier et comptabilité */
    N=0;
    SOMME=0;
    while (!feof(FICHIER))
    {
        fscanf (FICHIER, "%d\n", &NOMBRE);
        SOMME += NOMBRE;
        N++;
    }
    /* Fermeture du fichier */
    fclose(FICHIER);
    /* Affichage des résultats */
    printf("Le fichier %s contient %d nombres.\n", NOM_FICH,
N);
    printf("La somme des nombres est      : %ld\n", SOMME);
    printf("La moyenne des nombres est      : %f\n",
(float)SOMME/N);
    return (0);
}

```

Exercice 11.11 :

```

#include <stdio.h>

main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[30], NOUVEAU[30];
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char C; /* caractère lu dans le fichier */

```

```

char N_RET; /* Compteur des retours à la ligne consécutifs
*/

/* Ouverture de l'ancien fichier en lecture */
do
{
printf("Nom du fichier source : ");
scanf("%s", ANCIEN);
INFILE = fopen(ANCIEN, "r");
if (!INFILE)
printf("\aERREUR : Impossible d'ouvrir "
"le fichier : %s.\n", ANCIEN);
}
while (!INFILE);
/* Ouverture du nouveau fichier en écriture */
do
{
printf("Nom du nouveau fichier : ");
scanf("%s", NOUVEAU);
OUTFILE = fopen(NOUVEAU, "w");
if (!OUTFILE)
printf("\aERREUR : Impossible d'ouvrir "
"le fichier : %s.\n", NOUVEAU);
}
while (!OUTFILE);

/* Copier tous les caractères et remplacer le */
/* premier retour à la ligne d'une suite par */
/* un espace. */
N_RET=0;
while (!feof(INFILE))
{
C=fgetc(INFILE);
if (!feof(INFILE))
{
if (C == '\n')
{
N_RET++;
if (N_RET > 1)
fputc('\n', OUTFILE);
else
fputc(' ', OUTFILE);
}
else
{
N_RET=0;
fputc(C, OUTFILE);
}
}
}
/* Fermeture des fichiers */
fclose(OUTFILE);
fclose(INFILE);

```

```

    return (0);
}

```

Remarque :

Après la lecture par **fgetc**, il faut s'assurer encore une fois que le caractère lu est différent de **EOF**. Nous obtenons ainsi une construction un peu lourde :

```

while (!feof(INFILE))
{
    C=fgetc(INFILE);
    if (!feof(INFILE))
    {
        . . .
    }
}

```

Il est possible de réunir plusieurs instructions dans le bloc conditionnel de la structure **while**, en les séparant par des virgules. En pratique, on retrouve cette solution souvent pour éviter des constructions inutilement lourdes :

```

while (C=fgetc(INFILE), !feof(INFILE))
{
    . . .
}

```

Exercice 11.12 :

```

#include <stdio.h>
main()
{
    /* Prototype de la fonction FIN_PHRASE */
    int FIN_PHRASE(char C);
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char NOM_FICH[30];
    FILE *FICHIER;
    /* Autres variables */
    char C; /* caractère lu dans le fichier */
    char NP; /* Compteur de phrases */

    /* Ouverture de l'ancien fichier en lecture */
    do
    {
        printf("Nom du fichier texte : ");
        scanf("%s", NOM_FICH);
        FICHIER = fopen(NOM_FICH, "r");
        if (!FICHIER)
            printf("\aERREUR : Impossible d'ouvrir "
                "le fichier : %s.\n", NOM_FICH);
    }
    while (!FICHIER);
    /* Compter les symboles de fin de phrase */
    NP=0;
    while (!feof(FICHIER))
        NP += FIN_PHRASE(fgetc(FICHIER));
    /* Fermeture du fichier */
    fclose(FICHIER);
}

```

```

    /* Affichage du résultat */
    printf("Le fichier %s contient %d phrases.\n",
                                                NOM_FICH,
NP);
    return (0);
}

int FIN_PHRASE(char C)
{
    return (C=='.' || C=='!' || C=='?');
}

```

Exercice 11.13 :

```

#include <stdio.h>
main()
{
    /* Prototypage de la fonction FIN_PHRASE */
    int SEPA(char C);
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char NOM_FICH[30];
    FILE *FICHIER;

    /* Autres variables */
    char C;          /* caractère lu dans le fichier */
    int ABC[26];    /* compteurs des lettres de l'alphabet */
    int NTOT;       /* nombre total des caractères */
    int NAUTRES;    /* nombre des caractères qui ne font pas
                    partie de l'alphabet */
    int NMOTS;      /* nombre des mots */
    int NPARA;      /* nombre de paragraphes (retours à la ligne)
*/
    int I;          /* indice d'aide */
    int DANS_MOT;  /* indicateur logique : */
                    /* vrai si la tête de lecture se trouve */
                    /* actuellement à l'intérieur d'un mot. */

    /* Ouverture de l'ancien fichier en lecture */
    do
    {
        printf("Nom du fichier texte : ");
        scanf("%s", NOM_FICH);
        FICHIER = fopen(NOM_FICH, "r");
        if (!FICHIER)
            printf("\aERREUR : Impossible d'ouvrir "
                    "le fichier : %s.\n", NOM_FICH);
    }
    while (!FICHIER);
    /* Initialisations des variables */
    for (I=0; I<26; I++)
        ABC[I]=0;
    NTOT =0;
    NAUTRES =0;

```

```

NMOTS    =0;
NPARA    =0;
DANS_MOT=0;
/* Examenation des caractères du fichier */
while (!feof(FICHIER))
{
    C=fgetc(FICHIER);
    if (!feof(FICHIER))
    {
        /* Comptage au niveau caractères */
        if (C=='\n')
            NPARA++;
        else
        {
            NTOT++;
            if (C>='a' && C<='z')
                ABC[C-'a']++;
            else if (C>='A' && C<='Z')
                ABC[C-'A']++;
            else
                NAUTRES++;
        }
    }

/* Comptage des mots */
    if (SEPA(C))
    {
        if (DANS_MOT)
        {
            NMOTS++;
            DANS_MOT=0;
        }
    }
    else
        DANS_MOT=1;
}

/* Fermeture du fichier */
fclose(FICHIER);
/* Affichage du résultat */
printf("Votre fichier contient :\n");
printf("\t%d paragraphes\n", NPARA);
printf("\t%d mots\n", NMOTS);
printf("\t%d caractères\ndont\n", NTOT);
for (I=0; I<26; I++)
    printf("\t%d fois la lettre %c\n", ABC[I], 'a'+I);
printf("et %d autres caractères\n", NAUTRES);
return (0);
}

int SEPA(char C)
{
    /* Tableau contenant tous les séparateurs de mots */
    char SEP[12] = { '\n', ' ', ',', '.', ';', ':', '?', '!', '(', ')', '"', '\\'};
}

```

```

int I;

/* Comparaison de C avec tous les éléments du tableau */
for (I=0 ; C!=SEP[I] && I<12 ; I++) ;
if (I==12)
    return (0);
else
    return 1;
/* ou bien simplement : */
/* return (I != 12);    */
}

```

Exercice 11.14 :

```

#include <stdio.h>
main()
{
    /* Noms des fichiers et pointeurs de référence */
    char NOM_FICH[30];
    FILE *FICHIER;
    /* Autres variables */
    char C; /* caractère actuel */
    int NLIGNE, NCOLO; /* position actuelle sur l'écran */

    /* Ouverture de l'ancien fichier en lecture */
    do
    {
        printf("Nom du fichier texte : ");
        scanf("%s", NOM_FICH);
        FICHIER = fopen(NOM_FICH, "r");
        if (!FICHIER)
            printf("\aERREUR : Impossible d'ouvrir "
                "le fichier : %s.\n", NOM_FICH);
    }
    while (!FICHIER) ;
    getchar();
    /* Initialisations des variables */
    NLIGNE = 0; NCOLO = 0;
    /* Affichage du fichier */
    while (!feof(FICHIER))
    {
        C=fgetc(FICHIER);
        if (!feof(FICHIER))
        {
            NCOLO++;
            if (NCOLO==80 || C=='\n') /* fin de la ligne */
            {
                NLIGNE++;
                if (NLIGNE==25) /* fin de l'écran */
                {
                    getchar();
                    NLIGNE=1;
                }
            }
            printf("%c",C);

```

```

        NCOLO=1;
    }
    else
        printf("%c",C);
    }
}
/* Fermeture du fichier */
fclose(FICHIER);
return (0);
}

```

Exercice 11.15 :

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    /* Prototype de la fonction CCP_TEST */
    void CCP_TEST(long COMPTE, int CONTROLE);
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char NOM_FICH[] = "A:\\\\CCP.TXT";
    FILE *FICHIER;
    /* Autres variables */
    long COMPTE; /* nombre du compte CCP */
    int CONTROLE; /* nombre de contrôle */

    /* Ouverture du fichier CCP.TXT en lecture */
    FICHIER = fopen(NOM_FICH, "r");
    if (!FICHIER)
    {
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", NOM_FICH);
        exit(-1);
    }

    /* Lecture des nombres et appel de la fonction CCP_TEST */
    /* A l'aide de la chaîne de format, scanf lit les deux */
    /* parties du nombre de CCP, les convertit en long resp. */
    /* en int et affecte les résultats aux variables COMPTE */
    /* et CONTROLE. */
    while (!feof(FICHIER))
    {
        fscanf (FICHIER, "%ld-%d\n", &COMPTE, &CONTROLE);
        CCP_TEST(COMPTE, CONTROLE);
    }
    /* Fermeture du fichier */
    fclose(FICHIER);
    return (0);
}

```

```
void CCP_TEST(long COMPTE, int CONTROLE)
```

```

{
  int RESTE;
  RESTE = COMPTE % 97;
  if (RESTE == 0)
    RESTE = 97;
  if (RESTE == CONTROLE)
    printf ("Le nombre CCP %ld-%d est valide\n",
  COMPTE, CONTROLE);
  else
    printf ("Le nombre CCP %ld-%d n'est pas valide\n",
  COMPTE, CONTROLE);
}

```

Exercice 11.16 :

```

#include <stdio.h>
#include <stdlib.h>
main()
{
  /* Prototypage de la fonction FUSION */
  void FUSION(int *A, int *B, int *FUS, int N, int M);
  /* Déclarations : */
  /* Noms des fichiers et pointeurs de référence */
  char FICH_A[30], FICH_B[30], FICH_FUS[30];
  FILE *FA, *FB, *FFUS;
  /* Autres variables */
  int *TABA, *TABB, *TFUS; /* pointeurs pour les tableaux */
  int LA, LB; /* Longueurs de FA et FB */
  int N; /* Nombre lu ou écrit dans un fichier */
  int I; /* Indice d'aide */

  /* Ouverture du fichier FA en lecture */
  do
  {
    printf("Nom du fichier FA : ");
    scanf("%s", FICH_A);
    FA = fopen(FICH_A, "r");
    if (!FA)
      printf("\aERREUR : Impossible d'ouvrir "
      "le fichier : %s.\n", FICH_A);
  }
  while (!FA);
  /* Détection de la longueur de FA */
  for (LA=0; !feof(FA); LA++)
    fscanf(FA,"%d\n", &N);
  /* Fermeture du fichier FA */
  fclose(FA);
  /* Allocation de la mémoire pour TABA */
  TABA = malloc (LA*sizeof(int));
  if (!TABA)
  {
    printf("\a Pas assez de mémoire pour TABA\n");
    exit(-1);
  }
}

```

```

/* Réouverture du fichier FA en lecture */
FA = fopen(FICH_A, "r");
/* Copie du contenu de FA dans TABA */
for (I=0; I<LA; I++)
    fscanf(FA,"%d\n", TABA+I);
/* Fermeture du fichier FA */
fclose(FA);

/* Mêmes opérations pour FB : */
/* Ouverture du fichier FB en lecture */
do
{
    printf("Nom du fichier FB : ");
    scanf("%s", FICH_B);
    FB = fopen(FICH_B, "r");
    if (!FB)
        printf("\aERREUR : Impossible d'ouvrir "
            "le fichier : %s.\n", FICH_B);
}
while (!FB);
/* Détection de la longueur de FB */
for (LB=0; !feof(FB); LB++)
    fscanf(FB,"%d\n", &N);
/* Fermeture du fichier FB */
fclose(FB);
/* Allocation de la mémoire pour TABB */
TABB = malloc (LB*sizeof(int));
if (!TABB)
{
    printf("\a Pas assez de mémoire pour TABB\n");
    exit(-1);
}
/* Réouverture du fichier FB en lecture */
FB = fopen(FICH_B, "r");
/* Copie du contenu de FB dans TABB */
for (I=0; I<LB; I++)
    fscanf(FB,"%d\n", TABB+I);
/* Fermeture du fichier FB */
fclose(FB);

/* Allocation de la mémoire pour TFUS */
TFUS = malloc ((LA+LB)*sizeof(int));
if (!TFUS)
{
    printf("\a Pas assez de mémoire pour TFUS\n");
    exit(-1);
}

/* Fusion des tableaux TA et TB dans TFUS */
FUSION (TABA, TABB, TFUS, LA, LB);

```

```

/* Ouverture du fichier FFUS en écriture */
do
{
    printf("Nom du fichier FFUS : ");
    scanf("%s", FICH_FUS);
    FFUS = fopen(FICH_FUS, "w");
    if (!FFUS)
        printf("\aERREUR : Impossible d'ouvrir "
               "le fichier : %s.\n", FICH_FUS);
}
while (!FFUS);
/* Copier le contenu de TFUS dans FFUS */
for (I=0; I<(LA+LB); I++)
    fprintf(FFUS,"%d\n", *(TFUS+I));
/* Fermeture du fichier FFUS */
fclose(FFUS);
return (0);
}

void FUSION(int *A, int *B, int *FUS, int N, int M)
{
    /* Variables locales */
    /* Indices courants dans A, B et FUS */
    int IA,IB,IFUS;

    /* Fusion de A et B dans FUS */
    IA=0, IB=0; IFUS=0;
    while ((IA<N) && (IB<M))
        if (*(A+IA)<*(B+IB))
            {
                *(FUS+IFUS)=*(A+IA);
                IFUS++;
                IA++;
            }
        else
            {
                FUS[IFUS]=B[IB];
                IFUS++;
                IB++;
            }
    /* Si A ou B sont arrivés à la fin, alors */
    /* copier le reste de l'autre tableau. */
    while (IA<N)
        {
            *(FUS+IFUS)=*(A+IA);
            IFUS++;
            IA++;
        }
    while (IB<M)
        {
            *(FUS+IFUS)=*(B+IB);
            IFUS++;
            IB++;
        }
}

```

}

Chapitre 12 : ANNEXES :

Remarque : Dans la suite, les indications en bas à droite des cadres renvoient aux chapitres où les sujets respectifs sont traités.

ANNEXE A : Les séquences d'échappement :

<code>\n</code>	NL(LF)	Nouvelle ligne
<code>\t</code>	HT	Tabulation horizontale
<code>\v</code>	VT	Tabulation verticale (descendre d'une ligne)
<code>\a</code>	BEL	Sonnerie
<code>\b</code>	BS	Curseur arrière
<code>\r</code>	CR	Retour au début de ligne
<code>\f</code>	FF	Saut de page
<code>\\</code>	\	Trait oblique (back-slash)
<code>\?</code>	?	Point d'interrogation
<code>\'</code>	'	Apostrophe
<code>\"</code>	"	Guillemets
<code>\0</code>	NUL	Fin de chaîne

2.4. / 3.2.1.

ANNEXE B : Les priorités des opérateurs :

	<i>Classes de priorités :</i>	<i>Ordre de l'évaluation :</i>
Priorité 1 (la plus forte)	()	->
Priorité 2	! ++ --	<-
Priorité 3	* / %	->
Priorité 4	+ -	->
Priorité 5	< <= > >=	->
Priorité 6	== !=	->
Priorité 7	&&	->
Priorité 8		->
Priorité 9 (la plus faible)	= += -= *= /= %=	<-

3.5.

ANNEXE C : Les prototypes des fonctions traitées :

I) Entrée et sortie de données : <stdio.h>

Traitement de fichiers :

Le fichier en-tête <stdio.h> contient les déclarations d'un ensemble de fonctions qui gèrent des fichiers ainsi que l'indispensable structure **FILE** et les constantes **stdin** et **stdout**. Dans la suite, **FP** désigne une source ou une destination de données qui est liée à un fichier sur disque dur ou sur un autre périphérique.

Principe : On associe une structure **FILE** au fichier par l'intermédiaire d'un pointeur sur cette structure.

FILE *fopen(const char *NOM_FICH, const char *MODE) 11.4.1.

fopen ouvre le fichier indiqué et fournit un pointeur sur **FILE** ou zéro en cas d'échec.

MODE est une des chaînes suivantes:

"r" (*read*) lecture seule, à partir du premier caractère du fichier

"w" (*write*) écriture après création ou écrasement du fichier, s'il existe déjà

int *fclose(FILE *FP) 11.4.2.

fclose écrit les données de *FP* qui n'ont pas encore été écrites, jette les données de la mémoire tampon qui n'ont pas encore été lues, libère l'espace pris par la mémoire tampon et ferme le fichier associé à *FP*. **fclose** retourne **EOF** en cas d'échec, sinon zéro.

int feof(FILE *FP) 11.5.3.

feof fournit une valeur différente zéro si *FP* est arrivé à la fin du fichier.

int fprintf(FILE *FP, const char *FORMAT, ...) 11.5.1.

fprintf convertit les données dans une suite de caractères et les écrit dans *FP* sous le contrôle de *FORMAT*. La valeur du résultat est le nombre de caractères écrits; le résultat est négatif dans le cas d'une erreur.

FORMAT contient deux sortes de données : des caractères qui sont copiés comme tels dans *FP* et des spécificateurs de format qui définissent la conversion du prochain paramètre de **fprintf**.

Spécificateurs de format pour fprintf :

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

4.1.

Pour traiter correctement les arguments du type **long**, il faut utiliser les spécificateurs **%ld**, **%li**, **%lu**, **%lo**, **%lx**.

Pour traiter correctement les arguments du type **long double**, il faut utiliser les spécificateurs **%Lf** et **%Le**.

int fscanf(FILE *FP, const char *FORMAT, ...) 11.5.1.

fscanf lit des données de *FP* sous le contrôle de *FORMAT* et les attribue aux arguments suivants qui doivent être des pointeurs. La fonction s'arrête si la chaîne de format a été travaillée jusqu'à la fin. La valeur du résultat est le nombre des données lues ou **EOF** au cas d'une erreur ou si on est arrivé à la fin du fichier.

FORMAT contient trois sortes de données : des caractères qui doivent être entrés exactement de la même façon qu'ils sont notés dans la chaîne *FORMAT*; des signes d'espacement qui correspondent à une suite quelconque de signes d'espacement lors de l'entrée; des spécificateurs de format qui définissent la conversion de la prochaine donnée.

Spécificateurs de format pour fscanf :

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

4.2.

Si nous voulons lire une donnée du type **long**, nous devons utiliser les spécificateurs **%ld**, **%li**, **%lu**, **%lo**, **%lx**.

Si nous voulons lire une donnée du type **double**, nous devons utiliser les spécificateurs **%le** ou **%lf**.

Si nous voulons lire une donnée du type **long double**, nous devons utiliser les spécificateurs **%Le** ou **%Lf**.

int fputc(int C, FILE *FP) 11.5.2.

fputc écrit le caractère *C* (converti en **unsigned char**) dans *FP*. La fonction retourne le caractère écrit comme résultat ou **EOF** lors d'une erreur.

int fgetc(FILE *FP) 11.5.2.

fgetc lit le prochain caractère de *FP* comme **unsigned char** (converti en **int**) ou **EOF** à la fin du fichier ou lors d'une erreur.

Lire et écrire dans les fichiers standard

int printf(const char *FORMAT, ...) 4.1. / 8.6.1.

printf(...) est équivalent à **fprintf(stdout, ...)**.

int scanf(const char *FORMAT, ...) 4.2. / 8.6.1.

scanf(...) est équivalent à **fscanf(stdin, ...)**.

int putchar(int C) 4.3.

putchar(C) est équivalent à **fputc(C, stdout)**

int getchar(void) 4.4.

getchar est équivalent à **fgetc(stdin)**. (Comme le fichier *stdin* travaille à l'aide d'une mémoire tampon qui est évaluée ligne par ligne, **getchar** attend un retour à la ligne avant de fournir le caractère lu comme résultat.)

int puts(const char *CH) 8.6.1.

puts écrit la chaîne *CH* et un retour à la ligne dans *stdout*. La fonction retourne **EOF** lors d'une erreur sinon une valeur non négative.

char *gets(char *CH) 8.6.1.

gets lit la prochaine ligne de *stdin* et l'attribue à *CH* en remplaçant le retour à la ligne final par un symbole de fin de chaîne '\0'. La fonction retourne *CH* ou le pointeur nul à la fin du fichier ou lors d'une erreur.

II) Lire un caractère : <conio.h>

La fonction suivante est réservée à DOS : elle n'est pas conforme au standard ANSI-C et elle n'est pas portable.

int getch(void) 4.4.

getch lit un seul caractère au clavier et le retourne comme résultat sans l'écrire sur l'écran et sans attendre un retour à la ligne.



III) Traitement de chaînes de caractères : <string.h>

int strlen(const char *CH1) 8.6.2.

fournit la longueur de *CH1* sans compter le '\0' final

char *strcpy(char *CH1, const char *CH2) 8.6.2.

copie *CH2* vers *CH1* ('\0' inclus); retourne *CH1*

char *strncpy(char *CH1, const char *CH2, int N) 8.6.2.

copie au plus *N* caractères de *CH2* vers *CH1*; retourne *CH1*. Remplit la fin de *CH1* par des '\0' si *CH2* a moins que *N* caractères

char *strcat(char *CH1, const char *CH2) 8.6.2.

ajoute *CH2* à la fin de *CH1*; retourne *CH1*

char *strncat(char *CH1, const char *CH2, int N) 8.6.2.

ajoute au plus *N* caractères de *CH2* à la fin de *CH1* et termine *CH1* par '\0'; retourne *CH1*

int strcmp(const char *CH1, const char *CH2)

8.5. / 8.6.2.

compare *CH1* et *CH2* lexicographiquement et fournit un résultat :

négatif	si <i>CH1</i> précède <i>CH2</i>
zéro	si <i>CH1</i> est égal à <i>CH2</i>
positif	si <i>CH1</i> suit <i>CH2</i>

IV) Fonctions d'aide générales : <stdlib.h>

Le fichier en-tête <stdlib.h> déclare des fonctions pour la conversion de nombres, pour la gestion de mémoire et des charges pareilles.

1) Conversion de chaînes de caractères en nombres :

int atoi(const char *CH) 8.6.3.

atoi retourne la valeur numérique représentée par *CH* comme **int**

long atol(const char *CH) 8.6.3.

atol retourne la valeur numérique représentée par *CH* comme **long**

double atof(const char *CH) 8.6.3.

atof retourne la valeur numérique représentée par *CH* comme **double**

2) Gestion de mémoire :

void *malloc(unsigned int N) 9.5.3.

malloc retourne un pointeur sur un bloc de mémoire libre de la grandeur *N*, ou zéro si la requête ne peut pas être remplie. (Remarque : En ANSI-C, on utilise le type **void*** pour déclarer des pointeurs qui peuvent pointer sur tous les types).

void free(void *P) 9.5.4.

free libère le bloc en mémoire référencé par *P*; la fonction n'a pas d'effet si *P* a la valeur zéro. *P* doit pointer sur un bloc réservé à l'aide de **malloc** (ou **calloc** ou **realloc** - fonctions non traitées dans ce manuel).

3) Abandon d'un programme :

void exit(int STAT) 9.5.3. / 10.4. / 11.4.3.

exit termine un programme, ferme tous les fichiers encore ouverts et redonne le contrôle à l'environnement du programme en retournant la valeur *STAT* comme code d'erreur. La valeur zéro pour *STAT* indique une terminaison normale du programme.

Les trois fonctions suivantes sont réservées à DOS : elles ne sont pas conformes au standard ANSI-C et elles ne sont pas portables.



4) Conversion de nombres en chaînes de caractères :

char *itoa (int VAL, char *CH, int B) 8.6.3.

itoa convertit *VAL* dans une chaîne de caractères terminée par '\0' et attribue le résultat à *CH*; **itoa** retourne *CH* comme résultat. *B* est la base utilisée pour la conversion de *VAL*. *B* doit être compris entre 2 et 36 (inclus). (Réservez assez de mémoire pour la chaîne résultat : **itoa** peut retourner jusqu'à 17 bytes.)

char *ltoa (long VAL, char *CH, int B) 8.6.3.

ltoa convertit *VAL* dans une chaîne de caractères terminée par '\0' et attribue le résultat à *CH*; **ltoa** retourne *CH* comme résultat. *B* est la base utilisée pour la conversion de *VAL*. *B* doit être compris entre 2 et 36 (inclus). (**ltoa** peut retourner jusqu'à 33 bytes.)

char *ultoa (unsigned long VAL, char *CH, int B) 8.6.3.

ultoa convertit *VAL* dans une chaîne de caractères terminée par '\0' et attribue le résultat à *CH*; **ultoa** retourne *CH* comme résultat. *B* est la base utilisée pour la conversion de *VAL*. *B* doit être compris entre 2 et 36 (inclus). (**ultoa** peut retourner jusqu'à 33 bytes.)

V) Classification et conversion de caractères : <ctype.h> :

1) Fonctions de classification et de conversion de caractères :

Les fonctions suivantes ont des arguments du type **int**, dont la valeur est **EOF** ou peut être représentée comme **unsigned char**.

int isupper(int C) 8.6.4.

retourne une valeur différente de zéro, si *C* est une majuscule

int islower(int C) 8.6.4.

retourne une valeur différente de zéro, si *C* est une minuscule

int isdigit(int C) 8.6.4.

retourne une valeur différente de zéro, si *C* est un chiffre décimal

int isalpha(int C) 8.6.4.

retourne une valeur différente de zéro, si **islower(C)** ou **isupper(C)**

int isalnum(int C) 8.6.4.

retourne une valeur différente de zéro, si **isalpha(C)** ou **isdigit(C)**

int isxdigit(int C) 8.6.4.

retourne une valeur différente de zéro, si *C* est un chiffre hexadécimal

int isspace(int C) 8.6.4.

retourne une valeur différente de zéro, si *C* est un signe d'espacement

Les fonctions de **conversion** suivantes fournissent une valeur du type **int** qui peut être représentée comme caractère; la valeur originale de *C* reste inchangée :

int tolower(int C) 8.5. / 8.6.4.

retourne *C* converti en minuscule si *C* est une majuscule, sinon *C*

int toupper(int C) 8.5. / 8.6.4.

retourne *C* converti en majuscule si *C* est une minuscule, sinon *C*

VI) Fonctions arithmétiques : <math.h> :

1) Fonctions arithmétiques :

Le fichier en-tête <math.h> déclare des fonctions mathématiques. Tous les paramètres et résultats sont du type **double**; les angles sont indiqués en radians.

Remarque avancée :

La liste des fonctions ne cite que les fonctions les plus courantes. Pour la liste complète et les constantes prédéfinies voir <math.h>.



double sin(double X) 3.6.

sinus de X

double cos(double X) 3.6.

cosinus de X

double tan(double X) 3.6.

tangente de X

double asin(double X) 3.6.

arcsin(X) dans le domaine $[-1/2, 1/2]$, $x[-1, 1]$

double acos(double X) 3.6.

arccos(X) dans le domaine $[0, 1]$, $x[-1, 1]$

double atan(double X) 3.6.

arctan(X) dans le domaine $[-1/2, 1/2]$

double exp(double X) 3.6.

fonction exponentielle : e^X

double log(double X) 3.6.

logarithme naturel : $\ln(X)$, $X > 0$

double log10(double X) 3.6.

logarithme à base 10 : $\log_{10}(X)$, $X > 0$

double pow(double X, double Y) 3.6.

X exposant Y : X^Y

double sqrt(double X) 3.6.

racine carrée de X : \sqrt{X} , $X \geq 0$

double fabs(double X) 3.6.

valeur absolue de X : $|X|$

double floor(double X) 3.6.

arrondir en moins : $\text{int}(X)$

double ceil(double X) 3.6.

arrondir en plus

ANNEXE D : Bibliographie :



Brian W. Kernighan, Dennis M. Ritchie
Programmieren in C, 2. Ausgabe, ANSI-C
Carl Hanser Verlag, 1990



Borland C++ Version 3.1, Programmer's Guide, 1992
Borland C++ Version 3.1, User's Guide, 1992
Borland C++ Version 3.1, Library Reference, 1992



Claude Delannoy
Exercices en Langage C
Edition EYROLLES, 1992



C auf der Überholspur, Kurs : C-Programmierung
DOS International (Sept.1992 - Apr.1993)



Jean-Michel Gaudin
Infoguide Turbo - C
Editions P.S.i, 1988



Niklaus Wirth
Algorithmen und Datenstrukturen
B.G.Teubner, 1983



Jürgen Franz, Christoph Mattheis
Objektorientierte Programmierung mit C++
Franzis Verlag, 1992



EST - Langage algorithmique - Manuel de référence
Ministère de l'Education Nationale, 1991



EST - Informatique, Initiation à l'algorithmique - Classe de 12^e
EST - Informatique, Initiation à l'algorithmique - Classe de 13^e
Ministère de l'Education Nationale, 1992
