Mise à niveau en langage C
Cours/TP 2

LES TABLEAUX

Les tableaux correspondent aux matrices (ou vecteurs) en mathématiques. Un tableau est caractérisé par sa taille et par ses éléments.

Les tableaux à une dimension

Déclaration : **type nom[dim];**

Exemples :

```
int compteur[10];
float nombre[20];
```

Cette déclaration signifie **que le compilateur réserve 10 ou 20 places en mémoire pour ranger les éléments du tableau.**

Exemples :

```
int compteur[10];
```

le compilateur réserve des places pour 10 entiers, soit 20 octets en TURBOC et 40 octets en C standard

```
float nombre[20];
```

le compilateur réserve des places pour 20 réels, soit 80 octets

Remarque : la dimension est nécessairement une VALEUR NUMERIQUE. Ce ne peut être en aucun cas une combinaison des variables du programme.

Utilisation : un élément du tableau est repéré par son indice. En langage C les tableaux commencent à l'indice 0. L'indice maximum est donc dim-1 si la dimension est dim.
Appel : **nom[indice]**
Exemples :

```
compteur[2] = 5;
nombre[i] = 6.789;
printf("%d",compteur[i]);
scanf("%f",&nombre[i]);
```

D'une façon générale, les tableaux consomment beaucoup de place en mémoire. On a donc intérêt à les dimensionner au plus juste.

Les tableaux à plusieurs dimensions

Tableaux à deux dimensions

Déclaration : `type nom[dim1][dim2];`

Exemples :

```
int compteur[4][5];
float nombre[2][10];
```

Utilisation : un élément du tableau est repéré par ses indices. En langage C les tableaux commencent aux indices 0. Les indices maximums sont donc dim1-1, dim2-1.

Appel : `nom[indice1][indice2]`

Exemples :

```
compteur[2][4] = 5;
nombre[i][j] = 6.789;
printf("%d",compteur[i][j]);
scanf("%f",&nombre[i][j]);
```

Tableaux à plus de deux dimensions

On procède de la même façon en ajoutant les éléments de dimensionnement ou les indices nécessaires.

Initialisation des tableaux

On peut initialiser les tableaux au moment de leur déclaration

Exemples :

```
int liste[10] = {1,2,4,8,16,32,64,128,256,528};

float nombre[4] = {2.67,5.98,-8,0.09};

/* tableau de 2 lignes et 3 colonnes */
int x[2][3] = {{1,5,7},{8,4,3}};
```

TABLEAUX ET POINTEURS

Un **pointeur** n'est autre qu'une variable qui contient l'**adresse** d'une autre variable.

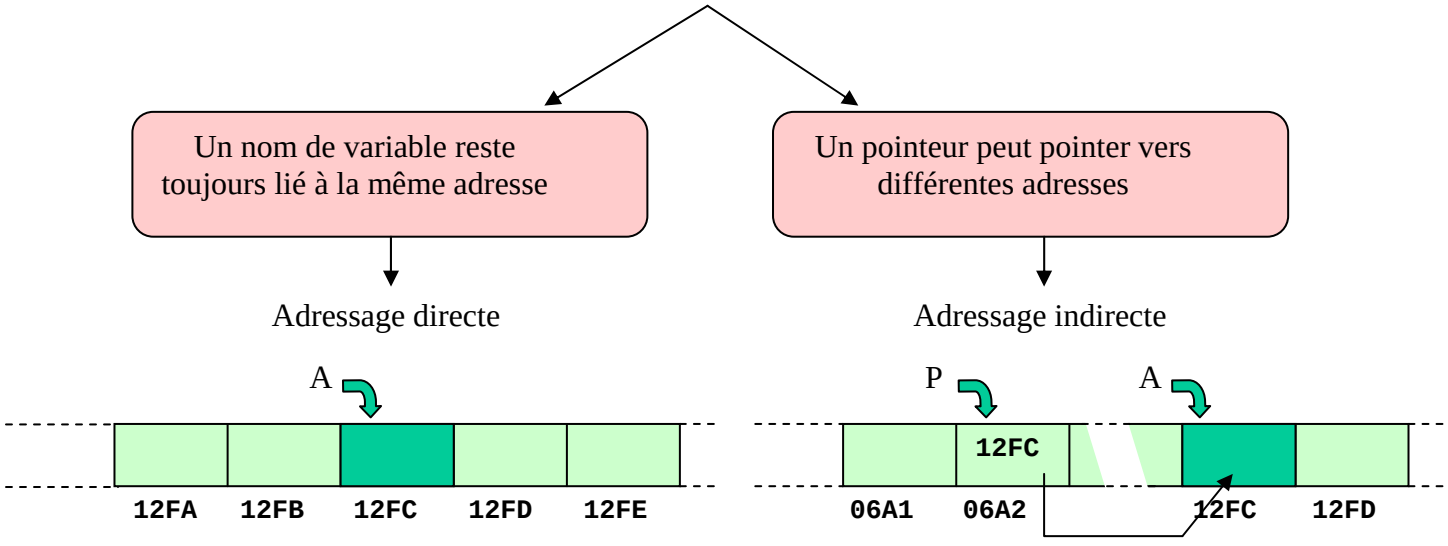
Mémoire : ce terme désigne d'une façon générale tout ce qui peut stocker de l'information, mais utilisé seul (i.e. « la mémoire »), il représente la mémoire centrale d'un ordinateur.

Adresse : valeur numérique désignant un emplacement en *Mémoire*. Souvent exprimée en Hexadécimal.

Il existe deux **modes d'adressage** sur ordinateur (particulièrement en C)

- adressage **direct** : L'adresse est donnée directement, sans calcul intermédiaire ni *Pointeur*. Le contenu de la *variable* est accessible directement par le nom de cette *variable.*
- adressage **indirect :** L'accès au contenu d'une variable est obtenu en passant par un *Pointeur* qui contient l'adresse de la variable.

Pointeurs et noms de variables jouent le même rôle : ils donnent accès à un Emplacement dans la mémoire.



L'opérateur d'adresse & :

Il permet de connaître l'adresse de n'importe quelle variable ou constante.

Exemple:

```
#include <stdio.h>
/* Déclaration d'un entier */
int i;
void main()
{
i=3;
/* Affichage de la valeur et de l'adresse de i */
printf("La valeur de i est %d\n",i);
printf("L'adresse de i est %d\n",&i);
}
```

Pour les tableaux:
L'adresse du premier élément est ***L'identificateur*** ou ***&identificateur[0]***.
L'adresse de l'élément ***n*** est ***&identificateur[n-1]***.

Exemple:

```
#include <stdio.h>
/* Déclaration d'un tableau d'entiers */
int tab[3];
/* Déclaration d'une chaîne */
char nom[10]="Bonjour";
void main()
{
/* Affichage de l'adresse de tab */
printf("La valeur de tab est %d\n",tab);
printf("\t\t ou \n");
printf("L'adresse de tab est %d\n",&tab[0]);
/* Affichage de l'adresse de nom */
printf("La valeur de nom est %d\n",nom);
printf("\t\t ou \n");
printf("L'adresse de nom est %d\n",nom);
}
```

Déclaration et manipulation de pointeur :

Un pointeur est une variable contenant l'adresse d'une autre variable, on parle encore d'indirection. La déclaration de pointeur se fait en rajoutant une étoile * devant l'identificateur de pointeur.

Syntaxe: ***<classe> <type>* ptr_identificateur_de_pointeur; /*** <classe> est facultatif */

Exemple:

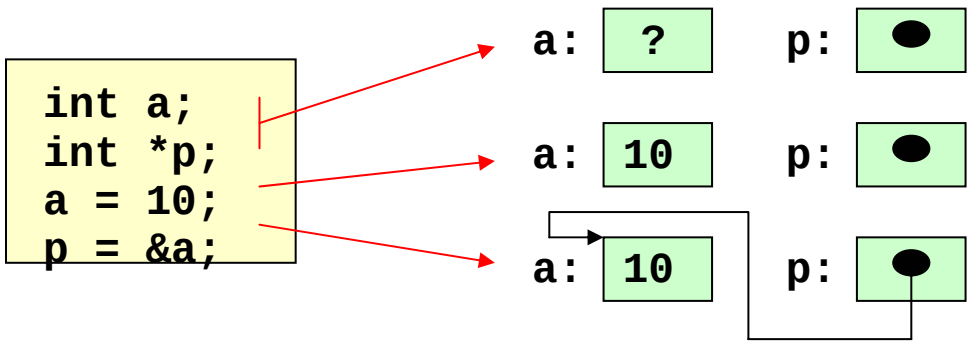
```c
#include <stdio.h>
/* Déclaration d'un entier */
int i;
/* Déclaration d'un pointeur d'entier */
int *ptr_i;
void main()
{
/* Initialisation de ptr_i sur l'adresse de i */
ptr_i=&i;
}
```

Pour pouvoir accéder à la valeur d'un pointeur il suffit de rajouter une étoile ***** devant l'identificateur de pointeur. L'étoile devant l'identificateur de pointeur veut dire objet (valeur) pointé par.

Exemple:

```c
#include <stdio.h>
/* Déclaration d'un entier */
int i;
/* Déclaration d'un pointeur d'entier */
int *ptr_i;
void main()
{
/* Initialisation de i */
i=5;
/* Initialisation de ptr_i sur l'adresse de i */
ptr_i=&i;
/* Affichage de la valeur de i */
printf("La valeur de i est %d\n",i);
/* Changement de valeur i par le pointeur ptr_i */
*ptr_i=56;
/* Affichage de la valeur de i par le pointeur ptr_i */
printf("La valeur de i est %d\n",*ptr_i);
}
```

Exemple:



Affectation de l'adresse de **a** *à* **p**

Tableaux et pointeurs :

En déclarant un tableau, on définit automatiquement un pointeur (on définit en fait l'adresse du premier élément du tableau).
Les écritures suivantes sont équivalentes :

```
/* déclaration */
int *tab;                                          int tab[10];
tab=(int*)malloc(sizeof(int)*10);

/* le 1er élément */
*tab                                               tab[0]

/* un autre élément */
*(tab+i)                                           tab[i]

/* adresse du 1er élément */
tab                                                &tab[0]

/* adresse d'un autre élément */
(tab + i)                                          &(tab[i])
```

Il en va de même avec un tableau de réels (float)

L'opérateur de déférence *

Définition
Un pointeur est une adresse mémoire. On dit que le pointeur pointe sur cette adresse.

Déclaration des pointeurs
Une variable de type pointeur se déclare à l'aide de l'objet pointé précédé du symbole * (opérateur d'indirection et de déférence)

Exemple:

```
char *pc;          pc est un pointeur pointant sur un objet de
                   type char
int *pi;           pi est un pointeur pointant sur un objet de
                   type int
float *pr;         pr est un pointeur pointant sur un objet de
                   type float
```

L'opérateur * désigne en fait le contenu de l'adresse.

Exemple:

```
char *pc;

*pc = 34;
printf("CONTENU DE LA MEMOIRE: %c\n",*pc);
printf("ADRESSE EN HEXA: %p\n",pc);
```

ARITHMETIQUE DES POINTEURS

On peut essentiellement **déplacer** un pointeur dans un plan mémoire à l'aide des opérateurs d'addition, de soustraction, d'incrémentation, de décrémentation.

On ne peut le déplacer que **d'un nombre de cases mémoire multiple du nombre de cases réservées en mémoire pour la variable sur laquelle il pointe.**

Exemples :

```
int *pi;          /* pi pointe sur un objet de type entier */
float *pr;        /* pr pointe sur un objet de type réel   */
char *pc;         /* pc pointe sur un objet de type caractère */

*pi=421;          /* 421 est le contenu de la case mémoire pi et des 3
suivantes */

*(pi+1)=53;       /* on range 53  4 cases mémoire plus loin */
*(pi+2)=0xabcd;   /* on range 0xabcd  8 cases mémoire plus loin */
*pr=45.7;         /* 45,7 est rangé dans la case mémoire pr et les 3 suivantes
*/
pr++;             /* incrémente la valeur du pointeur pr de 4 cases mémoire */

/* affichage de la valeur de l'adresse pr */
printf("L'ADRESSE r VAUT: %p\n",pr);

*pc = 'j';        /* le contenu de la case mémoire c est le code ASCII de 'j'*/
pc--;             /* décrémente la valeur du pointeur c d'une case mémoire   */
```

ALLOCATION DYNAMIQUE DE MEMOIRE

Lorsque l'on déclare une variable char, int, float, ..., un nombre de cases mémoire bien défini est **réservé** pour cette variable. Il n'en est pas de même avec les pointeurs.

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 l'***allocation dynamique*** de la mémoire.

Attention !

```
char *pc;

*pc = 'a';          /* le code ASCII de a est rangé dans la case mémoire pointée
                    par pc */
*(pc+1) = 'b';      /* le code ASCII de b est rangé une case mémoire plus loin  */
*(pc+2) = 'c';      /* le code ASCII de pc est rangé une case mémoire plus loin */
*(pc+3) = 'd';      /* le code ASCII de d est rangé une case mémoire plus loin  */
```

Dans cet exemple, le compilateur a attribué une valeur au pointeur pc, les adresses suivantes sont donc bien définies. Mais le compilateur n'a pas **réservé** ces places : il pourra très bien les attribuer un peu plus tard à d'autres variables. Le contenu des cases mémoires pc pc+1 pc+2 pc+3 sera donc perdu.

Ceci peut provoquer un blocage du système.

Il existe en langage C, des fonctions permettant **d'allouer de la place en mémoire** à un pointeur. Il faut absolument les utiliser dès que l'on travaille avec les pointeurs.

La fonction d'allocation malloc

Syntaxe :

```
void* malloc(size_t taille)
```

Cette fonction retourne un pointeur sur un objet non typé (mot clé void). L'argument en entrée spécifie le nombre total d'octets à réserver.

Exemples :

```
char *pc;
int *pi,*pj,*pk;
float *pr;

/* on réserve 10 cases mémoire, pour 10 caractères */
pc = (char*)malloc(10);

/* on réserve 16 cases mémoire, pour 4 entiers    */
pi = (int*)malloc(16);

/* on réserve 24 places en mémoire, pour 6 réels  */
pr = (float*)malloc(24);
```

Utilisation de l'opérateur sizeof

Syntaxe : `sizeof(nom_du_type)`
Cet opérateur donne un entier égal à la taille en octets de l'objet ou du type indiqué

Exemples :

```
/* on réserve la taille d'un entier en mémoire   */
pj = (int*)malloc(sizeof(int));

/* on réserve la place en mémoire pour 3 entiers */
pk = (int*)malloc(3*sizeof(int));
```

Libération mémoire : la fonction free

Syntaxe : `free(void* ptr)`

Exemples :

```
/* on libère la place précédemment réservée pour pi */
free(pi);

/* on libère la place précédemment réservée pour pr */
free(pr);
```

AFFECTATION D'UNE VALEUR A UN POINTEUR

Dans les exemples précédents, l'utilisateur **ne choisit pas** la valeur des adresses mémoire **attribuées par le compilateur à chaque variable.** L'utilisateur se contente de **lire** la valeur de ces adresses en l'affichant sur l'écran. On ne peut pas affecter directement une valeur à un pointeur : **l'écriture suivante est interdite**

```
char *pc;
pc = 0xfffe;
```

On peut cependant être amené à **définir par programmation** la valeur d'une adresse. On utilise pour cela l'opérateur de "cast", jeu de deux parenthèses. Par exemple :

- pour adresser un périphérique (adressage physique)
- pour contrôler la zone DATA dans un plan mémoire

Exemples :

```
char *pc;
pc=(char*)0x1000; /* pc est l'adresse 0x1000    */
                  /* et pointe sur un caractère */

int *pi;
pi=(int*)0xfffa; /* pi est l'adresse 0xfffa     */
                 /* et pointe sur un entier     */
```

Lorsqu'on utilise une fonction d'allocation dynamique on ne peut affecter de valeur au pointeur à l'aide de l'opérateur de "cast" ()

L'opérateur de "cast", permet d'autre part, à des pointeurs de types différents de pointer sur la même adresse

```
char *pc;             /* pc pointe sur un objet de type caractère    */
```

```
int *pi;              /* pi pointe sur un objet de type entier       */
pi = (int*)malloc(4); /* allocation dynamique pour pi                 */
pc = (char*)pi;       /* pc et pi pointent sur la même adresse et pc sur un
caractère */
```

LES CHAINES DE CARACTERES

En langage C, les chaînes de caractères sont des **tableaux de caractères**. Leur manipulation est donc analogue à celle d'un tableau à une dimension.

Déclaration :

```
char nom[dim];
```

ou bien

```
char *nom;
nom = (char*)malloc(dim);
```

Exemple :

```
char texte[dim];
```

ou bien

```
char *texte;
texte = (char*)malloc(dim);
```

Le compilateur réserve (dim-1) places en mémoire pour la chaîne de caractères. En effet, il ajoute toujours le caractère **NULL ('\0')** à la fin de la chaîne en mémoire.

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.

Pointeurs sur char et chaînes de caractères constantes

Affectation

a) On peut attribuer *l'adresse d'une chaîne de caractères constante* à un pointeur sur **char**, par exemple :

```
char *C;
  C = "Ceci est une chaîne de caractères constante";
```

Nous pouvons lire cette chaîne constante (p.ex: 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.

Initialisation

b) Un pointeur sur **char** peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante :

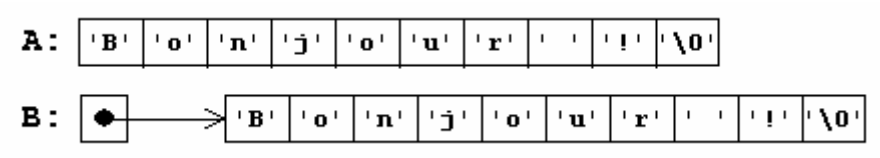
```
char *B = "Bonjour !";
```

Attention !
Il existe une différence importante entre les deux déclarations suivantes :

```
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.

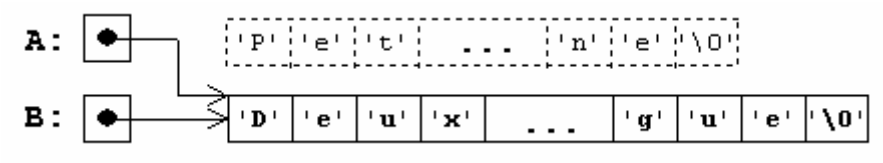


c) Si nous affectons une nouvelle valeur à un pointeur sur une chaîne de caractères constante, nous risquons de perdre la chaîne constante. D'autre part, un pointeur sur **char** a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur :

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 !!! */
```

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 (p.ex. dans une boucle) ou déléguer cette charge à une fonction de <*stdio*> ou <*string*>.

Les chaînes de caractères seront vues plus en détails en Travaux Pratiques.

PROCEDURES ET FONCTIONS

Jusqu'ici, nous avons résolu nos problèmes à l'aide de fonctions ou opérateurs prédéfinis 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.

MODULARISATION DE PROGRAMMES

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 efficientes.

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.

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.

- ***Simplification 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').*

LES MODULES EN LANGAGE ALGORITHMIQUE ET EN C

Cette partie résume les différences principales entre les modules (programme principal, fonctions, procédures) pour le langage de programmation que nous étudions.

Modules

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 :

a) Elle est exécutée lors de l'appel du programme.
b) 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 C, il est interdit de *définir* des fonctions à l'intérieur d'autres fonctions, mais nous pouvons *déclarer* des fonctions localement.

Variables locales

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 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 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 D'INTRODUCTION

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 principal

```
programme TEST_DIVI
        N,D,R  : ENTIER
début
        ECRIRE("Entrer numérateur et dénominateur : ")
        LIRE(N)
        LIRE(D)
        R ← DIVI(N,D)
        ECRIRE("Résultat : ",N,"  Reste : ",R)
fin
```

fonction DIVI

```
fonction DIVI(r A  : ENTIER, d B  : ENTIER) : ENTIER
        C : ENTIER
début
        C ← A modulo B
        A ← A div B

        retourner C
fin
```

- Le paramètre A est ***transféré par référence***: Il est déclaré avec le préfixe r dans la définition de la fonction.

- Le paramètre B est ***transféré par valeur*** : Il est déclaré avec le préfixe d dans la définition de la fonction.

- Le résultat de la fonction est ***affecté à la variable locale C*** de la fonction. Le contenu de cette variable est retourné (mot-clé ***retourner***) à la fin de 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

programme principal

```
#include <stdio.h>

main()
{
  int DIVI(int *A, int B);
  int N, D, R;

  printf("Entrer nominateur et dénominateur : ");
  scanf("%d %d", &N, &D);

  R = DIVI (&N, D);

  printf("Résultat : %d  Reste : %d\n", N, R);

  return 0;
}
```

fonction DIVI

```
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 (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.

DIFFERENCE ENTRE PROCEDURES ET FONCTIONS

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**.

Au contraire, une procédure est constituée d'un bloc comprenant des déclarations de variables une ou plusieurs instructions mais qui ne renvoie aucun résultat à la fin. En C, il n'existe pas de structure spéciale pour la définition de *procédures* comme en langage algorithmique. Nous pouvons cependant employer une fonction du type **void** partout où nous utiliserions une procédure en langage algorithmique ou même en Pascal.

La commande return

L'instruction

```
return <expression>;
```

a les effets suivants :

1. *évaluation de l'<expression>,*
2. *conversion automatique du résultat de l'expression dans le type de la fonction,*
3. *renvoi du résultat,*
4. *terminaison de la fonction.*

Exemples

1) 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);
}
```

ce code en C correspond en langage algorithmique à la fonction suivante :

```
fonction CARRE(d X  : REEL) : REEL

début
        retourner (X*X)
fin
```

2) La procédure LIGNE affiche une ligne de L étoiles.

procédure LIGNE

```
procédure LIGNE(d L : ENTIER)

        (* Déclarations des variables locales *)
        I : ENTIER
début
        (* Traitements *)
        I ← 0

        tant que I<>L faire
        début
                ECRIRE("*")
                I ← I+1
        fin (* I=L *)

        ECRIRE (* passage à la ligne *)
fin
```

Pour la traduction en C de cette ***procédure***, nous utilisons une ***fonction*** du type **void** :

fonction LIGNE

```
void LIGNE(int L)
{
       /* Déclarations des variables locales */
       int I;

       /* Traitements */

       for (I=0; I<L; I++)
              printf("*");

       printf("\n");
}
```

PARAMETRES 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.*

Passage des paramètres par valeur

En C, le passage des paramètres se fait toujours par la valeur, c.-à-d. 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");
}
```

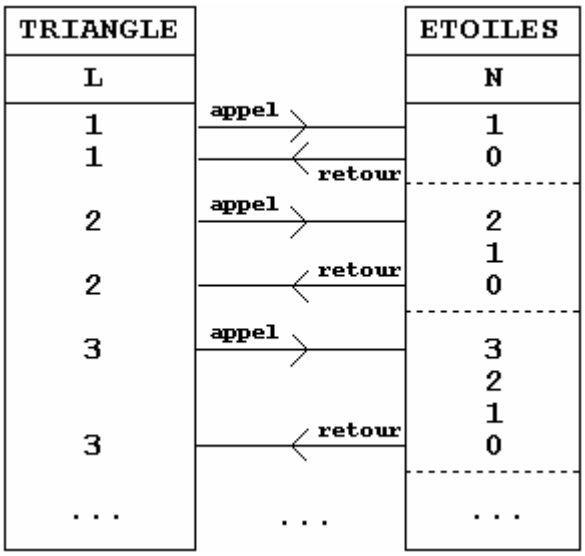
En utilisant N comme compteur, nous n'avons pas besoin de l'indice d'aide I comme dans la fonction LIGNES définie plus haut.

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 décrémentée 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 :



<u>Passage des paramètres par adresse</u>

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 :

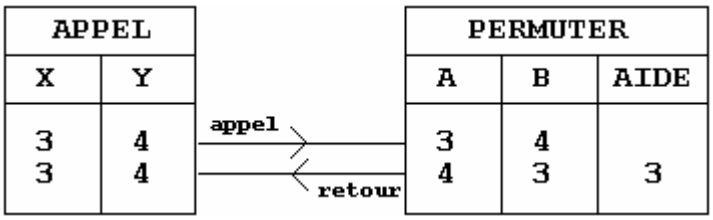
```
void PERMUTER (int A, int B)
{
        int AIDE;

        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 ! Pourquoi ?
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;
}
```

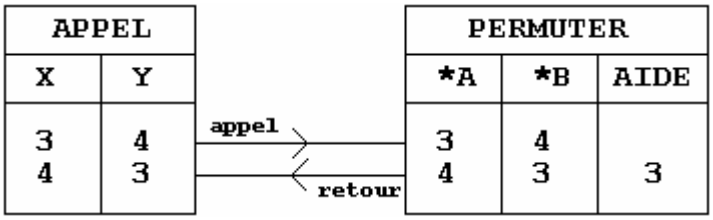
Nous appelons la fonction par :

```
PERMUTER(&X, &Y);
```

Résultat: Le contenu des variables X et Y est cette fois 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.



Passage de l'adresse d'un tableau à une dimension

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***.

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[]**

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.

CHOISIR ENTRE UNE PROCEDURE ET UNE FONCTION ?

Discussion de deux problèmes

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 :

a) Nous pouvons fournir la valeur comme résultat de la fonction.
b) 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.

Résultat int ==> "fonction"

Définissons d'abord la fonction ENTREE suivante :

```
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();
```

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 algorithmique, nous dirions que nous pouvons remplacer une ***fonction*** par une ***procédure***.

En général, la préférence pour l'une ou l'autre variante dépend le 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.

Exemples

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);
```

La fonction MOY fournit la valeur moyenne des valeurs d'un tableau T de N d'entiers. Cette fois, il s'agit d'une seule valeur à retourner. Nous pouvons donc définir MOY comme une fonction qui retourne un objet de type double :

```
double MOY(int N, int *T);
{
        int I;
        double S = 0.0;

        for (I=0; I<N; I++)
                S += *(T+I);

        return (S/N);
}
```

Lors d'un appel de MOY, il suffit de récupérer le résultat dans une variable *ad hoc* :

```
int TAB[8] = {2,5,-1,0,6,9,-4,6};
int N = 8;
double M;

M=MOY(N, TAB);
```

Bibliographie

- E. Deléchelle, *Algorithmique et programmation*, Université Paris 12.
- B.W. Kernighan, D.M. Ritchie, *Le langage C – Norme ANSI*, Edition MASSON (2ème édition).

Exercices

Exercice 1 :
Écrire un programme qui affiche les nombres pairs entre 0 et 100 en utilisant :

- Une boucle for.
- Une boucle while.
- Une boucle do … while.

Exercice 2 :
Pour un vecteur x = {x1, x2, ….., xn}, on a :

$$\|x\| = \left(\sum_{i=1}^{n} x_i{}^2 \right)^{1/2}$$

Ecrire un programme qui :

- Saisi n la dimension de x.
- Saisi les coordonnées de x (en utilisant une boucle do … while).
- Calcule et affiche la norme de x.

Exercice 3 :
Ecrire un programme permettant de :

- Saisir un vecteur V1[10] d'entier.
- Saisir un vecteur V2[10] d'entier.
- Calculer et afficher la différence de ces deux vecteurs donnée comme suit :

$$\text{Difference} = \frac{1}{100} \sum_{i=0}^{9} \left(V1[i] - V2[i] \right)^2$$

Exercice 4 :
Ecrire un programme permettant de :

- Saisir la taille d'un vecteur n (n<10).
- Saisir un vecteur V d'entier.
- Diviser ce vecteur en deux sous vecteurs VP et VI, avec VP contient les éléments pairs de V et VI contient les éléments impairs de V.
- Afficher les deux vecteurs VP et VI.

Exercice 5 :
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:

```
a b c d
e f g h   ==>    a b c d e f g h i j k l
i j k l
```

Exercice 6 :
Ecrire un programme qui réalise l'addition de deux matrices carrées A et B, tous les deux saisit au clavier

Exercice 7 :
Ecrire un programme qui lit 4 mots, séparés par des espaces et qui les affiche ensuite dans une ligne, mais dans l'ordre inverse. Les mots sont mémorisés dans 4 variables M1, ... ,M4.
Exemple :
section imagerie industrielle 1
1 industrielle imagerie section

Exercice 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:

Donner un verbe : fêter
je fête
tu fêtes
il fête
nous fêtons
vous fêtez
ils fêtent

Exercice 9 : *Palindromes*
Ecrire et programmer l'algorithme récursif ou non d'une fonction *palindrome(…)* qui détermine si une suite de caractère est un palindrome (c'est à dire symétrique par rapport à son milieu). Les chaînes *ABCBA* et *00100100* sont des palindromes.

Exercice 10 : *Palindromes par morceax*
Après avoir programmé l'algorithme récursif d'une fonction *palindrome(…)*, on peut compliquer le problème en écrivant l'algorithme d'une fonction *palParMorceau(…)* qui reconnaît si une chaîne est un palindrome par morceau (cas où la chaîne est une concaténation de plusieurs palindromes). Par exemple, la chaîne *01010010* est un palindrome par morceau : le premier palindrome est formé des 5 premiers caractères 01010, et le second est formé des 3 derniers 010.

- **Remarques :**
  a. On ne cherchera pas à traiter le cas des palindromes imbriqués (cas de *ABACA*, constitué des palindromes *ABA* et *ACA*) mais uniquement celui des palindromes **disjoints**.
  b. On cherchera uniquement les palindromes **maximaux** (le cas de *AAAAA* doit donner un seul palindrome).
  c. La fonction *palParMorceau(…)* peut appeler la fonction *palindrome(…)*.

Exercice 11 : ***Drapeau hollandais à 1 dimension***

On rappelle l'algorithme permettant de construire le *drapeau hollandais* à une dimension ; on supposera que l'on range les couleurs bleu, blanc et rouge dans cet ordre, et que ces couleurs sont disposés initialement de manière quelconque dans le tableau.
Dans cet algorithme, on suppose que ***b*** représente l'indice dans le tableau du dernier bleu rangé, ***r*** représente l'indice du premier rouge rangé, ***i*** l'indice de la couleur à ranger et ***n*** représente la taille du tableau.

```
i = 1
b = 0
r = n+1
tant que (i < r) faire
        cas drapeau[i] = blanc  :  i = i+1
                         bleu   :  b = b+1
                                   drapeau[i] = drapeau[b]
                                   drapeau[b] = "bleu"
                                   i = i+1
                         rouge  :  r = r-1
                                   drapeau[i] = drapeau[r]
                                   drapeau[r] = "rouge"
        fincas
finTantQue
```

a. Ecrire une fonction *echange(x,y,T)* qui échange le contenu des deux cases *x, y* (du tableau *T*) données en paramètres.
b. Ecrire une fonction *DrapeauHollandais(tableau_a_ordonner, b, r)* qui est la transcription de l'algorithme présenté ci-dessus. Utiliser la fonction *echange()*.
c. Ecrire la fonction *main()* qui représente le « programme principal ».

Exercice 11 : ***Nombres Romains***

Le but de cet exercice est de calculer la valeur entière d'un nombre écrit en chiffres romains. Pour cela, on décide que les chiffres romains utilisés seront exclusivement les caractères suivant :

Nombre	I	V	X	L	C	D	M
valeur	1	5	10	50	100	500	1000

La règle d'évaluation est la suivante : en général, les chiffres romains sont écrits de gauche à droite dans l'ordre décroissant de leur valeur ; dans le cas contraire, un chiffre romain qui se trouve à gauche d'un chiffre de valeur strictement supérieur est compté négativement.
Ainsi, IV vaut 4, mais VI vaut 6. De même CCCXXXIX vaut 339, MCMLXXXVII vaut 1987, MCMXCIX vaut 1999.

On se contentera de cette règle sans chercher si les autres règles traditionnelles d'écriture sont ou non satisfaites. On suppose également que les nombres romains sont correctement écrits, c'est à dire qu'il n'y a pas de « blancs » contenus dans les nombres romains, de même que tout autre caractère est invalide.

On rappelle l'algorithme de conversion d'un nombre romain en sa valeur décimale :

```
romain = lire (nombre_romain)
decimal = 0
pour chaque chiffre de romain faire
        si valeur_decimale(romain[i]) < valeur_decimale(romain[i+1])
                alors decimal = decimal – valeur_decimale(romain[i])
                sinon decimal = decimal + valeur_decimale(romain[i])
        finSi
finPour
```

Exercice 12 : ***Tri par sélection***

L'algorithme de tri le plus simple est le tri par sélection. Il consiste à trouver l'emplacement de l'élément le plus petit du tableau (***tab***), c'est-à-dire l'entier ***m*** tel que ***tab(i) ≥ tab(m)*** pour tout ***i***. Une fois cet emplacement trouvé, on échange les éléments ***tab(1)*** et ***tab(m)***. Puis on recommence ces opérations sur la suite ***(tab(2), tab(3), …, tab(n))***, ainsi on recherche le plus petit élément de cette nouvelle suite et on l'échange avec ***tab(2)***. Et ainsi de suite… jusqu'au moment où on n'a plus qu'une suite composée d'un seul élément.

Un exemple illustrant les itérations de l'algorithme du tri par sélection est donné ci-dessous. En gras, les valeurs déjà traitées.

5	3	1	2	6	4
5	3	1	2	4	**6**
4	3	1	2	**5**	**6**
2	3	1	**4**	**5**	**6**
2	1	**3**	**4**	**5**	**6**
1	**2**	**3**	**4**	**5**	**6**

Exercice 13 : ***Tri « Bulle »***

Une variante du tri par sélection est le tri bulle (bubble sort). Son principe est de parcourir la suite ***(tab(1), tab(2), …, tab(n))*** en intervertissant toute paire d'éléments consécutifs ***(tab(i), tab(i+1))*** non ordonnées. Ainsi après un parcours l'élément maximum se retrouve en ***tab(n)***. On recommence alors avec le préfixe ***(tab(1), tab(2), …, tab(n-1))***, puis ***(tab(1), tab(2), …, tab(n-2))*** …

Le nom de tri bulle vient donc de ce que les plus grands nombres se déplacent vers la droite en poussant des bulles successives de la gauche vers la droite. La fonction correspondante utilise un indice ***i*** qui marque la fin du préfixe à trier, et un indice ***j*** qui permet de déplacer la bulle qui monte vers la borne ***i***. L'algorithme se termine lorsqu'il n'y a plus de permutations possibles. Ce fait sera constaté grâce à un dernier parcours du tableau où aucune permutation n'a lieu.

Un exemple illustrant les itérations de l'algorithme du tri Bulle est donné ci-dessous. En gras, les éléments qui sont comparés, et éventuellement permutés, pour passer à la ligne suivante.

5	**3**	1	2	6	4
3	**5**	**1**	2	6	4
3	1	**5**	**2**	6	4
3	1	2	**5**	**6**	4
3	1	2	5	**6**	**4**
3	**1**	2	5	4	6
1	**3**	**2**	5	4	6
1	2	**3**	**5**	4	6
1	2	3	**5**	**4**	6
1	2	3	4	**5**	**6**

Exercice 14 :

Soit le programme suivant :

```
void main()
  {
    int a = 10;
    int b = 15;
    int c = 20;
    int *p1, *p2;
    p1=&a;
    p2=&c;
    *p1=(*p2)+1;
    p1=p2;
    p2=&b;
    *p1-=*p2;
    ++*p2;
    *p1*=*p2;
    a=*p2**p1;
    p1=&a;
    *p2=*p1/=*p2;
  }
```

Complétez le tableau ci-dessous :

	a	***b***	***c***	***p1***	***p2***
Initiale	**10**	**15**	**20**	***I***	***I***
p1=&a	**10**	**15**	**20**	**&a**	***I***
p2=&c					
*p1=(*p2)+1					
p1=p2					
p2=&b					
*p1-=*p2					
++*p2					
p1=*p2					
a=*p2**p1					
p1=&a					
*p2=*p1/=*p2					

Exercice 15 :

Soit p un pointeur qui pointe sur un tableau T:

```
int T[9] = {12, 2, 34, 4, 15, 23, 57, 8, 13};
int *p;
p = T;
```

Quelles valeurs ou adresses fournissent ces expressions:

a) *p+2
b) *(p+2)
c) &T[4]
d) &T[4]-3
e) p+4
f) p+(*p-10)

g) *(p+*(p+8)-T[7])

Exercice 16 :
Ecrire une fonction Permute, avec passage d'arguments par adresse, permettant de permuter deux entiers a et b.

Dans le programme principal void main() :

- Saisir deux entiers x et y.
- Permuter les valeurs de x et y en utilisant la fonction Permute.

Exercice 17 :
Ecrire une fonction de prototype void maxmin(int t[], int N, int *admax, int *admin) qui ne renvoie aucune valeur et qui détermine la valeur maximale et la valeur minimale d'un tableau d'entiers de taille N.

Exercice 18 :*
Ecrire une fonction de saisie des éléments d'un tableau de réels. Le nombre d'éléments du tableau sera donné en argument de la fonction qui retournera l'adresse du tableau. Ce tableau devra être alloué dynamiquement. Une gestion des erreurs permettra de renvoyer le pointeur *NULL* en cas d'erreur d'allocation mémoire.

Exercice 19 :*
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 des 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.

Exercice 20 :*
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***.

Exemples :

Bonjour	Bravo	⇒	njou
abacab	aa	⇒	bcab

Exercice 21 :*
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 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 dans 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.