Mémento du Langage C

Table des matières

Lexique du langage C....3
Types de données simples....3
Affectation....3
Instruction 3
Arithmétiques....3
Séquence et Bloc....4
Opérateurs booléens....4
Tests et conditionnels....4
Expression et instruction 4
Structure d'un programme C 5
Type de données composées : tableau 6
Lecture - écriture simple 7
Chaînes de caractères (tableau de caractères)....7
Les fonctions de manipulation de chaînes de caractères....8
Enregistrement (struct)....9
Déclarations de types....9
Tailles des types de base....9
Variables locales - globales....10
Conversion de types....11
Production de programmes....12
Gestion des tableaux 15
Fonctions et paramètres 18
Opérateurs ++, --, +=, -=, ... (la forme générale @=)....20
Conditionnel "? :" 20
Boucles et itérations....20
Définition de constantes par #define....22
Boucle de lecture au clavier (EOF, ^Z)....23
Tableaux multidimensionnels (matrices) 26
Exemples et exercices....27
Exercices sur les Tableaux....29
Exercice sur Enregistrements 32
Type énuméré....33
Schéma CASE 33
Allocation dynamique....35
Applications : deux algorithmes de tri 38
1- méthode de tri par sélection 38
Exercice 38
2- méthode de tri par insertion 39
ANALYSE RECURSIVE 40

Décomposition récurrente 41
Exemple 41
Exemples et exercices 42
Récurrence structurelle 48
Application 50
Exercices 50
Gestion des fichiers en C....................56
Fonctions de la bibliothèque standard....................56

....................59
Travail à rendre 60
Suite : Listes, ...AEF, Analyseur syntaxique, ...60

Remarque : dans ce document, les mots clés et les exemples en langage C sont en *italique*.

Lexique du langage C

Comme le langage PASCAL (chiffres, lettres, virgule, point virgule, ponctuations, ...)
Mais en langage C :

- les majuscules et les minuscules sont différenciés ;
  => la variable "*b*" n'est pas la même que la variable "*B*".
- les commentaires sont entre "/*" et "*/" et peuvent tenir sur plusieurs lignes.
- les mots clés du langage sont en minuscules (*for, while, ...*).
- les constantes pré définies sont en général en majuscule (*EOF*).

Types de données simples

int => *int i ; int taux, intérêt ;*
float et double
char => *char c; double d; float f;*

NB : On peut **initialiser** chaque variable à la déclaration : *float f=1.2, g=5.4;*
NB : Le type **booléen** n'existe pas et l'on utilise le type *int* pour ce type.
Dans ce cas, **faux = 0 , autre => vrai**
Il existe pourtant les opérateurs booléens en C (voir plus bas).
NB : Le type **char** est en fait un entier sur un octet contenant le code ASCII du caractère.
On peut donc faire des opérations arithmétiques sur un caractère :
Exemple : *char C = 'a' ; C = C + 1 ; /* C contient 'b' */*

Affectation

= /* à ne pas confondre avec l'égalité notée == */
Exemple *a = b;*

Instruction

Expression terminée par ";". L'affectation (suivie de ';') est une instruction.
Exemple : ***a = b*** est une expression ; ***a = b ;*** est une instruction.
Voir plus loin pour les expressions.

Arithmétiques

Les opérateurs **+, -, *, / , %, ++, --**
Le résultat de ces opérations dépendent des type des opérandes.
Le type float l'emporte sur les entiers et caractères.
Exemple : *int i=1 ; float f=1.5 ; f+i est de type float*

Séquence et Bloc

Séquence d'instructions entre **{ }** et séparées par ";" (cf. *begin ... end* de Pasacl)

Exemple : *{int R; float Pi=3.14, Surface;*
*R=12; Surface = Pi * R * R; ......*
}

Opérateurs booléens

Les opérateurs logiques **!, &&, ||** (non, et, ou)

Exemple :

```
if ( (a> b) && (a>c )) max = a;
if ( ( a <= b) || (b <= c) ) min = ...
if ( ! (a>b) )  max = ...
```

Tests et conditionnels

Avec **if-else** et les opérateurs **==, !=, >, >=, <, <=**

Le format : *if (expression) ....*

NB : Il n'y a pas le mot clé *then*.
Un *point virgule* est nécessaire après l'instruction précédant *else*.
Les tests sont toujours placés entre parenthèses.

Exemple :

```
if (a == b) a = a+1;
if (a > b) a = a+1; else a = a-1;
if (a != b) {a = a+1; b=b-a;}  else {a= a-1; b=a; }
```

NB : Un des pièges de C est la confusion entre = (affectation) et == (égalité).

Exemple : *if (a = 1) ...* /* ce qui est toujours vrai */

Expression et instruction

Une expression peut être une constante (numérique, booléenne, caractère), une variable, une opération composée d'opérandes et d'opérateur, un appel de fonction, etc.

Une affectation est aussi une expression (au moins deux opérandes et l'opérateur '=').

Une expression donne lieu à une valeur lorsqu'elle est évaluée.

Selon la syntaxe de C/C++, une expression (éventuellement à effet de bord), devient une instruction (une action) lorsqu'elle est suivie de ';'.

Un effet de bord est un changement de valeurs des variables ou autre modification sur l'environnement du travail tel qu'une lecture ou écriture.

Pour simplifier et comparer au langage Pascal, on dira que les instructions de C/C++ sont comme celles de Pascal mais en C/C++, il est possible de construire des expressions à effet de bord.

Dit d'une manière différente, les expressions en C/C++ représentent chacune une valeur et certaines peuvent devenir des instructions si elles sont suivies de ';' (lors que la syntaxe du langage le permet).

Pour cela, le langage C/C++ utilise la notion d'*instruction - expression*. Une instruction - expression est une expression terminée par ';'. C'est en général le cas des affectations et des appels de fonctions.

Par exemple, *a=b* est une expression dont la valeur est b et dont l'effet de bord est de modifier la valeur de a.

On peut par exemple écrire :

if (x=y) c = 5;

Ici, la valeur de y est affectée à x puis la valeur de l'expression (la valeur de b) est comparée à 0.

De même, on peut écrire en C/C++

x=y=z=k=12;

la valeur 12 est affectée à k puis le résultat (12) est affecté à z, etc. L'ensemble devient une instruction (grâce à ';').

Comme cela a été dit plus haut, toute expression ne peut pas être une instruction. C'est la syntaxe du langage qui le permet. Par exemple, on ne peut pas avoir l'instruction

a + 5;

Un autre exemple d'instruction - expression est I+++ (voir plus loin). L'effet en est d'incrémenter la valeur de I mais l'expression représente la valeur de I avant cette incrémentation. Par exemple :

int I = ... ;
if (I++ == 5) ...

la valeur de I est comparée à 5 puis I est incrémentée (voir aussi ++I, I--, --I, ...)

Structure d'un programme C

En C, tout sous-programme est une fonction.
Il n'y a pas d'imbrication de sous programmes.
Une fonction renvoie une valeur par **return**.*valeur_du_type_de_retour;*
Le début du programme est le début de la fonction particulière **main**.
Une fonction de type "void" est une procédure sans valeur de retour;
(voir plus loin les exemples de fonction).

NB : Fonction sans paramètre.

Exemple :

```
int ma_fonction()           /* une fonction sans paramètre qui renvoie un entier */
{char c;                    /* variable locale */
 c = 'A';
     /* .... le reste du corps de la fonction ... */
return un_entier;
}

void une_proc()                 /* une fonction sans valeur de retour = une procédure */
{
     /* .....  */
}

int main()                  /* la fonction  principale = le point d'entrée du programme */
{int i;
 i = ma_fonction();             /* appel de ma_fonction : on met les () */
une_proc() ;                    /* appel de une_proc */
     /* ...... */
}
```

NB : Il faut déclarer ou définir une fonction pour s'en servir (règle de précédence).
On peut déclarer l'entête d'une fonction et définir la fonction elle-même plus tard ;
Ne pas oublier '()' lors de l'appel d'une fonction sans paramètre.

Type de données composées : tableau

- Tableaux (**[]**)
  => *int tab[20];* /* déclare un tableau de 20 entiers. */

NB : le premier élément est toujours à l'indice 0 ; le dernier à n-1
Ici, le premier élément de tab est *tab[0]*, le dernier *tab[19]*

Exemple :

```
void main()
{int tab[10];
tab[0]=12;
tab[9]=15;
int i = tab[3] + 12 ;
.......
}
```

N.B. :

- la taille d'un tableau (entre []) doit être une expression constante. La taille ne peut pas être une variable.
- on peut initialiser un tableau à la déclaration en mettant les valeurs entre {}:
Exemple : *int t[5] = {3, 5, 2, 9, 7} ;*

-la taille d'un tableau initialisé à la déclaration peut être omise :
Exemple **:** *float F[] = {3.1, 5.0, 2.2, 9.1} ;* **=>** le compilateur calcule la taille.

Lecture - écriture simple

Les entrées sorties simples sont faites à l'aide des fonctions **scanf** et **printf**.
Les entrées sorties sont faites à l'aide d'un *format* de lecture/écriture :
%d pour les entiers
%c pour les caractères
%f et %e %g pour les réels
%s pour les chaînes de caractères

Exemple : lire un entier et décider s'il est pair ou impair. (opérateur **%** : modulo)

```
void main()
{int nombre;
 printf("donner un nombre : ");
 scanf("%d", &nombre);          /* le & devant nombre */
 if (nombre % 2 == 0) printf("pair");
 else printf("impair");
}
```

NB : La variable de lecture dans *scanf* est précédée de **&** (sauf pour les tableaux et les adresses).

Cas de chaînes (tableaux) de caractères

```
char chaine[20];
printf("%s", chaine);
scanf("%s", chaine);          % pas de & devant chaîne */
```

Chaînes de caractères (tableau de caractères)

Une chaîne de caractères est un tableau de caractères particulier.

Exemple :

char chaine[10]; */* tableau de 10 caractères */*

- caractère NULL (code ASCII 0)
- une chaîne constante correspond à une adresse en mémoire.
=> On ne peut donc pas affecter une chaîne constante à un tableau de caractères.
Donc, l'instruction *chaine = " ecole";* est interdite

NB : à ne pas confondre avec l'initialisation à la déclaration : *char t[] = "école";*
- fonctions String.h : strcpy, strcat, strlen, strcmp, ...

Exemple :

```
char chaine[15] ="Ecole Centrale";        /* déclaration suivie d'initialisation */
                                  /* Une autre manière : laisser le compilateur compter la taille
                                          /* char chaine[] ="Ecole  Centrale";

printf("%s", chaine);                     /* afficher chaîne */
chaine[0]='E';                            /* changer le 1er caractère de chaîne */
...
```

Les fonctions de manipulation de chaînes de caractères

On ne peut pas faire d'affectation directe dans une chaîne de caractères.
L'affectation, la comparaison, la concaténation et le calcul de la taille d'une chaîne de caractères sont réalisés à l'aide des fonctions de la bibliothèque C/C++ dont les déclarations d'en-tête se trouvent dans le fichier <string.h>.

Exemple :

```
#include <stdio.h>            /* fonctions pour lire et écrire */
#include <string.h>           /* fonctions de traitement des chaînes */

void main()
{int L;
char nom1[15], nom2[20];

strcpy(nom1, "toto");

L = strlen(nom1);             /* L= le nombre de caractères dans nom1 */
printf("la taille de la chaine %s est %d \n", nom1, L);

strcpy(nom2, nom1);
printf("la taille de la chaine %s est %d \n", nom2, strlen(nom2));

strcat(nom2, "et titi");

if (strcmp(nom1, nom2) == 0)             /* comparaison : 0 => identiques */
      printf("les deux chaines identiques \n");
else  printf("les deux chaines différentes \n");
}
```

Enregistrement (struct)

Syntaxe :
```
struct
{       /* les champs de l'enregistrement */
} nom_record;
```

Exemple :
```
#include <stdio.h>                        /* pour lire et écrire*/

void main()
{
 struct
        {       int num;
                float salaire;
                char code;
        } my_record;
my_record.num = 10;
my_record.salaire = 111.6;
my_record.code = 'A';
printf("%d %f %c  \n", my_record.num, my_record.salaire, my_record.code);        /* écrire  */
}
```

Déclarations de types

un type se définit par **typedef**

Exemples :
```
        typedef int entier;             /* le type entier est le même que le type int */
        typedef char chaine[10];        /* le type tableau de 10 caractères */

        typedef struct personne         /* un type enregistrement */
                {chaine nom;
                 float salaire;
                 entier labo;
                };
```

Tailles des types de base

La taille d'un type varie d'une machine à une autre (selon le processeur ou le compilateur).
Habituellement, les tailles en octets sont :

char =1, short int = 1, int = 2, long int = 4, float = 4, double = 8, long double = 10...

Les nombre réels ont une organisation mantisse - exposant.
La taille d'un enregistrement = somme des tailles de ses composants.

La taille d'un type peut être obtenue avec la fonction **sizeof**.

Variables locales - globales

Toute variable à l'intérieur d'une fonction est locale à cette fonction.
Les variables globales sont déclarées hors les fonctions.
Une variable déclarée dans la fonction main est locale à main (pas globale).

Variables statiques

Une variable déclarée *static* est locale à la fonction où elle est déclarée mais elle est gérée comme une variable globale.

Exemple :

```
#include <stdio.h>
int next()
        {static int val = 0;
         int i = 1;
          val = val +1;
          retrn val;
        }

void main()
        {printf("Appel 1 : %d \n", next());
         printf("Appel 2 : %d \n", next());
        }

Résultats :
        Appel 1 : 0
        Appel 2 : 1;
```

La variable *val* déclarée *static* n'est pas créée dans le bloc de la fonction *next()* mais dans la zone des variables globales (voir plus loin). Elle est ensuite initialisée **une seule fois** (ici à 0) et conserve sa valeur dans les appels successifs de la fonction *next()*.
Par contre, la variable locale *i* est créée et initialisée à 1 à chaque fois qu'on appelle la fonction *next()*.
Cette propriété des variables statiques permet par exemple de conserver le pointeur d'enregistrement d'un fichier d'une lecture à une autre ou de générer par une fonction une valeur aléatoire qui dépend de la valeur précédente.

Conversion de types

Syntaxe : *(type) valeur*

Exemples :

int i=1; float f= (float) i;
float f=1.5; int i= (int) f;

Exemple : calcul de la moyenne de 2 nombres

```
int somme = 7;
float moyenne;

Moyenne = Somme / 2;            /* un entier qui sera converti en réel => perte de précision */
Moyenne = (float)Somme / N;     /* correcte. Une seule conversion suffit (float x int => float) */
.....
```

Conversions automatiques

Dans une opération arithmétique, les valeurs des opérandes sont converties dans un type commun. En général, les types plus 'petits' sont convertis en types plus 'larges' afin de ne pas perdre en précision. Par exemple, *float* est plus large que *int*.
Dans une affectation, l'opérande droit est converti dans le type de l'opérande gauche. On peut dans ce cas avoir des pertes de précision comme dans l'exemple ci-dessus (moyenne).

Exemple :

```
int I=8;
float X = 12.5;
double Y;
Y = I * X;        // donne Y=100.00
```

Ici, la valeur de I est convertie en float (le plus large des deux). Le résultat de la multiplication sera de type float qui sera converti en double avant l'affectation.

Règles de conversion automatiques dans les opération arithmétiques :
- dans une opération avec char, short et int, les valeurs sont converties en **int** (plus large).
- sur les entiers, on a l'échelle : **int, unsigned int, long, unsigned long.**
- dans une opération avec un entier et un réel (float ou double), l'entier est converti en **réel**.
- dans une opération avec deux réel, le type le plus large sera utilisé dans l'échelle : **float, double, long double.**

Cas d'affectation : le résultat est toujours converti dans le type de la destination et on peut perdre en précision si ce type est plus faible.

Exemple :

```
unsigned int A = 70000;      /*  Ici, A contiendra : 70000 modulo 65536 = 4464. */
```

Production de programmes

Cycle de production d'un programme exécutable : Edition - Compilation - Edition de liens.

Edition : production d'un ou plusieurs fichiers sources ou modules (extension *.c ou .cpp*) à l'aide d'un éditeur.
On crée souvent un fichier d'en-tête ou header (extension *.h ou .hpp*) contenant des déclarations de types, de constantes, de variables et de sous programmes. Un fichier d'en-tête peut être partagé par plusieurs modules.
Un programme peut inclure des déclarations prédéfinies dans des fichiers d'en-tête. Pour cela, on utilise la directive *#include fichier-entête.*
Par exemple, pour les entrées sorties, on utilise :

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

On utilise <fichier> lorsque le fichier header est un prédéfini de C/C++ et "fichier" lorsqu'il a été défini et crée par l'utilisateur.

Compilation : le compilateur vérifie la syntaxe des déclaration et autres constructions.
Il signale les erreurs syntaxiques, les erreurs de déclaration, certaines incompatibilités de types dans les opérations, certaines incompatibilités de paramètres dans les appels des fonctions, ...
Le résultat de la compilation est un fichier binaire d'extension *.obj* (sous Dos/Windows) ou d'extension *.o* (sous Unix).

Edition des liens (link) : un programme C/C++ peut utiliser des déclarations prédéfinis de variables, de types ou de sous programmes.
Par exemple, les fonctions *scanf* ou *printf* sont définies dans les bibliothèques d'entrées sorties; des fonctions mathématiques telles que *sqrt* (racine carrée) est dans la bibliothèque mathématique; des constantes (*EOF, NULL*) ou le type *FILE* (type fichier) sont définis dans des fichiers header,...
Ces informations ont été incluses par *#include* dans le programme source et le code des fonctions doivent être assemblés avec le fichier binaire issu de la compilation.
Lors de l'édition des liens, plusieurs fichiers *.obj* (ou *.o*) seront regroupés pour produire un fichier exécutable *.exe* sous Dos/Windows ou un fichier de nom quelconque sous Unix.

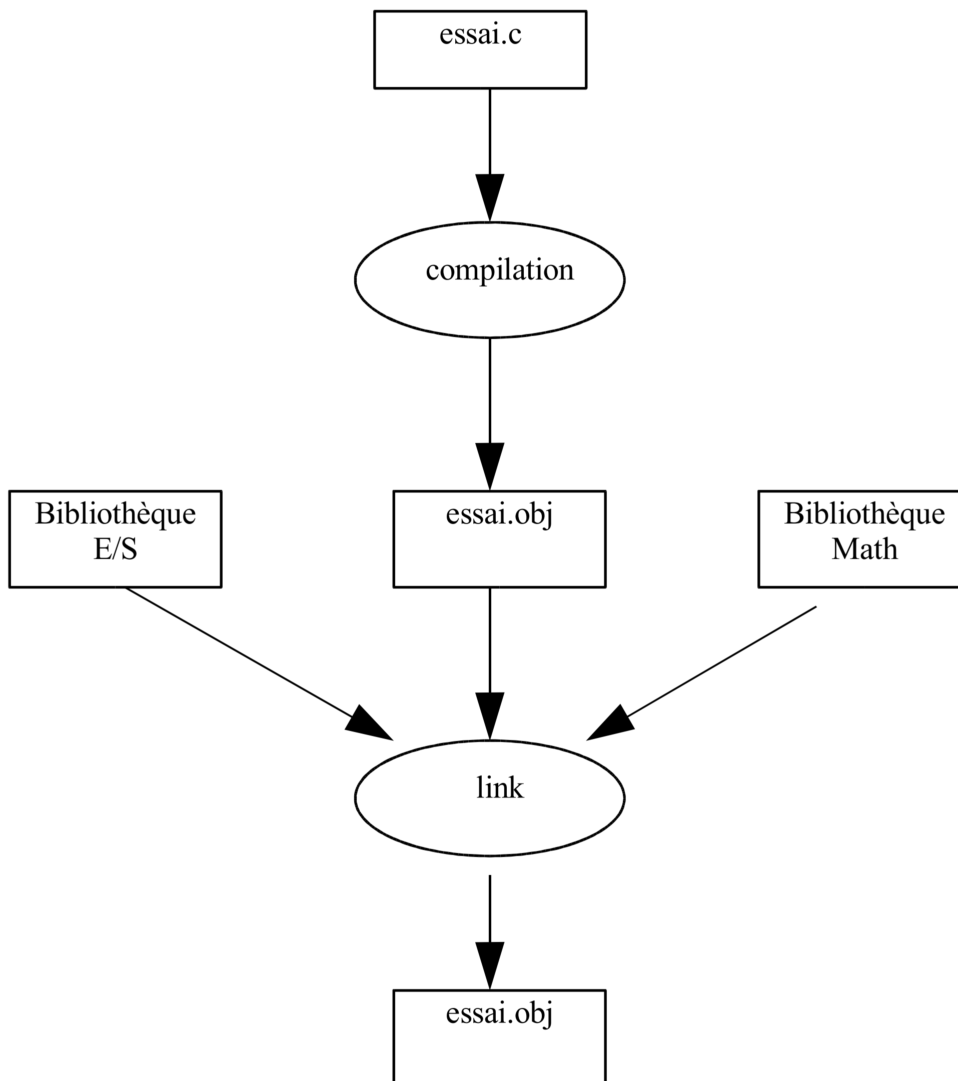
Projet : les séquences d'exécution des opérations de compilation et d'édition de liens peuvent être automatisées par un fichier appelé *Projet* (sous Dos/Widows et Turbo C/C++) ou par un fichier *Makefile* sous Unix.
Un fichier projet contient (selon certaine syntaxe) les opérations à effectuer sur des fichiers pour obtenir un exécutable.

L'exemple ci-dessous montre les étapes de création d'un exécutable.

Exemple : soit le programme *essai.c* suivant qui lit un nombre et affiche sa racine carrée.

```
#include <stdio.h>
#include <math.h>
void main()
{float f;
 printf("donner un nombre reel : ");
 scanf("%g", &f);
 printf("la racine carree de %g = %g \n", f,  sqrt(f));
}
```

Ce programme utilise les routines prédéfinies d'entrées sorties (*printf et scanf*) dont les déclarations se trouvent dans *stdio.h*. Il utilise aussi la fonction *sqrt* dont la déclaration se trouve dans *math.h*.
Les codes binaires des ces fonctions se trouvent dans les bibliothèques prédéfinis.
Les opérations de compilation et d'édition de lien pour créer *essai.exe* (sous Dos/Windows) sont données par la figure suivante.



Remarque : la phase de compilation peut détecter la majorité des erreurs dites statiques. Cependant, l'édition de liens et même l'exécution peuvent révéler d'autres erreurs.
L'exemple simple suivant montre quelques types d'erreurs possibles d'un programme.

Exemple de programme erroné

```
#include <stdio.h>
int i , 12;
void main()
{int j = 0;
  i = f(j);
  k = i / j;          // division par zéro
}
```

La production d'un exécutable détectera les erreurs selon les différentes phases.

Compilation :
ligne 2 : erreur syntaxique pour **12**
ligne 6 : erreur de **k** : variable non déclarée

Edition de liens :
ligne 5 : la fonction **f** non définie

Exécution :
ligne 6 : division par 0

Remarque : en C++, on peut noter les commentaire en commençant par // .... jusqu'à la fin de la ligne.

Gestion des tableaux

Rappels :

int TabInt[10]; // déclaration d'un tableau de 10 entiers

La taille d'un tableau doit être une valeur connue.
Elle est vérifiée au moment de la compilation.

Si l'on initialise le tableau immédiatement après sa déclaration, on peut laisser le compilateur calculer cette taille.

int TabInt[] = {1, 5, 3, 2, 10 }; // déclaration d'un tableau de 5 entiers

Ou pour un tableau de caractères :

char TabChar[] = "abcd"; // même chose que {'a', 'b', 'c', 'd', '\0'};

Remarques :

L'expression **&X** représente l'adresse de **X**.
L'expression ***Y** représente le contenu de la zone mémoire dont l'adresse est **Y**.

Le nom d'un tableau :

a/ est le nom de la zone allouée au tableau en mémoire (allocation à l'exécution)

b/ correspond à l'adresse du premier élément du tableau
=> **TabInt** est équivalent à **&TabInt[0]**

c/ est une constante non modifiable. On ne peut donc pas écrire l'instruction:

TabInt = ...; // ERREUR : affectation à une constante

d/ on peut faire de l'arithmétique avec cette adresse :

TabInt +1 = &TabInt[1];

D'une manière générale, si le tableau **Tab** est implanté à l'adresse **xxxx** en mémoire :

Tab + 1 = xxxx + taille du type de base du tableau
Tab + i = xxxx + i * taille du type de base du tableau

Exemple : écrire les éléments d'un tableau d'entiers *TabInt* de taille N:

Solution simple :

```
for (i=0; i < N ; i++)   printf("%d", TabInt[i]);
```

Autre solution :

```
for (i=0; i < N ; i++)   printf("%d", *(TabInt + i));
```

L'expression *TabInt + i* permet de pointer les entiers du tableau (saut de 2 octets).

Exemple : même chose avec un tableau de caractères :

```
int TabChar[] = "abcd";                 // la chaîne se termine avec '\0'
for (i=0; *(TabChar + i) != '\0'; i++)
        printf("%c", *(TabChar + i));   // ou putchar(*(TabChar + i))
```

L'expression *TabChar + i* permet de pointer les caractères du tableau (saut d'1 octet).

C'est le compilateur qui calcule le saut en fonction du type des éléments du tableau.

Pointeurs, Adresses et Références

Exemple :
On déclare un entier I et l'on suppose qu'il est implanté à l'adresse 2000 en mémoire.

int I = 5;

On déclare PI comme un pointeur sur un entier :

int * PI;

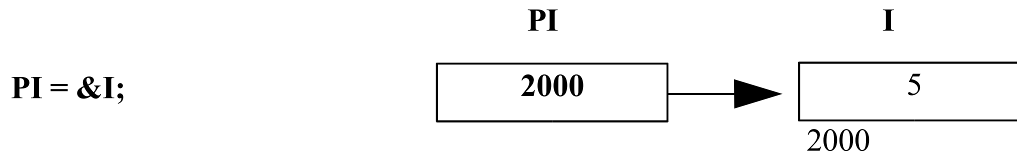
PI est de type pointeur sur entier. Elle peut contenir l'adresse d'un entier.

On peut faire deux lectures équivalentes de cette déclaration :

```
int * PI;    =>  PI est de type pointeur sur entier     => (int *) PI
int *PI;     =>  *PI est un entier                      => int (*PI)
```

On affecte l'adresse de I à PI :

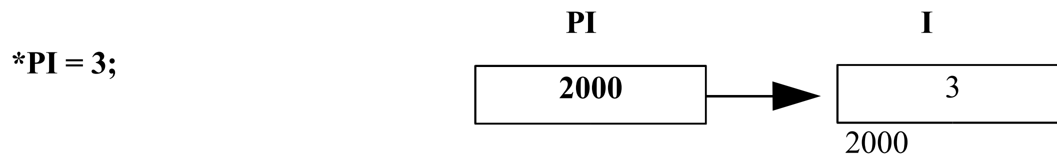
PI = &I;



Cette affectation range dans PI l'adresse de I. On dit que PI pointe I.

On peut modifier la valeur de I en utilisant PI. L'expression *PI représente l'entier pointé par PI :

***PI = 3;**



Maintenant, I contient 3, c'est à dire, la case N° 2000 de la mémoire contient 3.

Rappel : en C/C++,
- L'expression **&X** représente l'adresse de **X**
- ***Y** représente le contenu de la zone pointée par **Y**
- Ne pas confondre le rôle de '*' :

  la déclaration : int * PI; /* PI est pointeur sur entier */

  et *PI = 3; /* la zone pointée par PI reçoit une valeur */

Fonctions et paramètres

En C, un paramètre de fonction est passé par valeur. C++ propose le passage par référence.
Le passage par référence est réalisé par le passage de l'adresse du paramètre.
Lorsque ce paramètre est une adresse, on peut manipuler l'objet pointé.
Une fonction dont le type de retour est *void* ne renvoie pas de valeur.

Exemples en C

Passage de paramètre par valeur :

```
    int incremente1 (int X)          /* renvoyer X+1*/
    {return X+1 ; }

Appel :
     int A;
    A=incremente1(5);
```

Dans l'exemple suivant, on transmet l'adresse d'un entier à la fonction :

```
    void incremente2 (int * X)       /* incrémenter X */
    { *X = *X +1; }                  /* X est un pointeur sur entier*/

Appel :
     int A;
    incremente2(&A);             /µ on transmet l'adresse de A */
```

Dans l'exemple suivant, on échange les valeurs de 2 entiers. Pour cela, on a besoin de modifier les arguments, d'où le passage de paramètre par référence.

```
    void permuter(int * X, int * Y)  /* inversion de 2 nombres */
    {int Z;
      Z = *X; *X = *Y; *Y = Z ;
    }

Appel :
     int A, B;
    permuter(&A, &B);
```

Passage de paramètres avec & (à la C++)

En plus du passage de paramètre par valeur,
En C++, le passage par référence se fait à l'aide de &
NB : même principe qu'en Pascal

Exemple : une autre version de permute.
La fonction permute reçoit deux paramètres transmis par référence. Toute modification des paramètres i et j dans la fonction permute est répercutée sur les arguments A et B.

```
void permute(int & i, int & j)
{int temp = i;
 i = j;
 j = temp;
}

Appel : int A, B;
  permute(A, B);        // remarquer l'appel.
```

Remarque :

Il ne faut pas confondre les deux cas d'utilisation de '&' :

1- Dans l'entête d'une fonction :

la fonction : void f(int & **A**)
{A = 1;}

et l'appel depuis main: int B;
f(**B**);

2 - dans les opérations :

int I = 5;
int * PI = &I;

Dans le premier cas, *A* est un paramètre modifiable. A l'appel *f(B)*, le compilateur passe l'argument *B* par référence à *f*. Dans la fonction *f*, *A* ne représente pas l'adresse de *B* mais un paramètre passé par référence qui correspond à *B*. Si le paramètre *A* est modifié, l'argument *B* de l'appel l'est aussi.

Dans le second cas, **&I** représente bien l'adresse de *I*.

Opérateurs ++, --, +=, -=, ... (la forme générale @=)

```
int I = 1;
I++;        // incrémenter I
++I;        // idem
int J;
J += I;     // J = J +1
```

Conditionnel "? :"

Syntaxe : *expression ? si_oui : si_ non*

Exemple : calcul du maximum de deux nombres A et B dans C

```
Avec if-else
    if (A > B) then C = A; else C = B;

Avec ? :
    C = (A > B) ? A : B;
```

Exemple : écrire N entiers du tableau d'entiers Tab en écrivant 10 entiers séparés par un espace par ligne.

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

Exemple : convertir la caractère car1 (éventuellement) en caractère car2 majuscule :

```
car2 = (car1 >= 'a' && car1 <= 'z') ? ('A' + car1 - 'a') : car1;
```

Boucles et itérations

while, do while, for

Exemple : la somme des éléments d'un tableau de 20 entiers
Syntaxe de while : **while (expression) { instructions }**

```
void main()
{int tab[20];
 int i=0, somme=0;
 while (i < 20)
        {scanf("%d", &tab[i]); i = i+1;}
 i = 0;
 while (i < 20)
        {somme += tab[i]; i++;}
 printf("la somme = %d", somme);
}
```

Même exemple avec **do-while**

Syntaxe de do - while : **do { instructions } while (expression);**

```
void main()
{int tab[20];
 int i=0, somme;
 do
        {scanf("%d", &tab[i]); i = i+1;}
while (i < 20);

i = somme = 0;
do     somme += tab[i++];           /* une seule instruction */
while (i<20);
printf("la somme = %d", somme);
}
```

Même exemple avec **for**

Syntaxe de for : **for (initialisations ; conditions d'itération; post instructions) { instructions}**

```
void main()
{int tab[20];
int i, somme;
for( i=0; i<20; i++) scanf("%d", &tab[i]);
for( i=0 , somme=0; i<20; i++) somme += tab[i];          /* remarquer la virgule */
printf("la somme = %d", somme);
}
```

Autre exemple : calcul de la valeur maximum des entiers d'un tableau tab[N] avec for :

```
int i, Max=tab[0];              /* on suppose que le 1er élément est le plus grand */
for (i=1; i<N; i++)
        Max = (tab[i] > Max) ? tab[i] : Max;

printf("le maximum est %d \n", Max);
```

NB : **break, continue**

break permet de sortir de la boucle dans laquelle il se trouve.

continue permet de passer à l'itération suivante dans la boucle où il se trouve.

Exemple de *break* :vérifier si un tableau d'entier tab[N] est ordonné croissant.

```
int i, est_ordonne=1;           /* un booléen déclaré comme un entier init vrai*/
for (i=0; i < N-1; i++)
    if (tab[i] > tab[i+1])
           {est_ordonne = 0; break;}           /* on arrête dès que est_ordonne = faux */

if (est_ordonne)          /* ou  if (i == N-1) */
        printf("le tableau est ordonné \n");
else .....
```

Exemple de *continue* : même exercice.

```
int i=0, est_ordonne=1;        /* un booléen déclaré comme un entier init vrai*/
while (( i < N-1) && est_ordonne)
    {if (tab[i] <= tab[i+1])
         {i++; continue;}
      est_ordonne = 0;
     }

if (est_ordonne)          /* ou  if (i == N-1) */
       printf("le tableau est ordonné \n");
else ....
```

Remarque : on peut éviter d'utiliser le booléen et se servir de la valeur de i à la sortie de la boucle.

Autre solution avec *break* et *continue* sans utiliser le booléen :

```
int i, est_ordonne=1;           /* un booléen déclaré comme un entier init vrai*/
for (i=0; i < N-1; i++)
   { if (tab[i] <= tab[i+1]) continue;      /* il faut continuer */
     break;                    /* on arrête si l'on arrive ici */
   }

if ( if (i == N-1)
       printf("le tableau est ordonné \n");
else .....
```

Autre solution : utilisation d'une boucle sans corps.

```
int i;
for(i=0;  (i < N-1) && (tab[i] <= tab[i+1]); i++) ;          /* boucle vide */
if (i == N-1) printf("le tableau est ordonné \n");
else .....
```

Remarque : sur le ';' après une boucle.

Autre exemple : utilisation de continue dans l'exemple de calcul de Max d'un tableau tab[N]

```
int i, Max=tab[0];               /* on suppose que le 1er élément est le plus grand */
for (i=1; i<N; i++)
       {if (Max >= tab[i]) continue;
       Max = tab[i];
       }
printf("le maximum est %d \n", Max);
```

Définition de constantes par #define

#define N 10
#define True 1
#define False 0
#define Bool int
.....

Boucle de lecture au clavier (EOF, ^Z)

```
while ((c=getchar()) != EOF) ....
while (scanf("%c",&c) != EOF) ....
```

Exemple : lire une suite de caractères et constituer les mots et les afficher

NB : les mots sont séparés par un seul espace et il n'y a pas d'autre espace.

```
void main()
{char c, mot[20];
 int i=0;            /* indice dans mot */
 while ((c=getchar()) != EOF)
      {if (c == ' ')           /* séparateur de mots */
        {mot[i]='\0';          /* terminer la chaîne mot */
         printf("le mot construit = %s", mot);
         i=0;                  /* pour le prochain mot */
        }
       else {mot[i]=c; i++;}
        }

        /* ici, traiter le dernier mot */
        if (i != 0)        /* un dernier mot avait commencé sans terminer avec un ' '*/
        {mot[i]='\0';          /* terminer la chaine mot */
         printf("le mot construit = %s", mot);
        }
}
```

Exercices : palindrome

```
char chaine[20];
  scanf("%s", chaine);
int i=0;
int L= strlen(chaine);
while(i < L /2)
       {if (chaine[i] != chaine[L-i-1])
              {printf("%s n'est pas un palindrome", chaine); break;}
       else i += 1;
       }
```

Exercices : calcul de la moyenne (un entier) de N entiers.
Remarque : 2 méthodes :

- lecture et calcul à la volée ou
- lecture et stockage dans un tableau puis calcul.

```
#include <stdio.h>
#define N 8

int moyen1()
{int tab[N];
 int i, somme=0;
 for (i=0; i<N; i++)
        {printf("la valeur de tab[%d] = ",i+1);
           scanf("%d",&tab[i]); }
 for (i=0; i<N; i++) somme +=tab[i];
return (somme / N);
}

int moyen2()
{ int i,j, somme=0;
 for (i=0; i<N; i++)
        {printf("la %deme valeur = ",i+1);
         scanf("%d",&j);
         somme += j;
        }
return (somme / N);
}

int main( void )    /* remarque sur void */
{       printf("la moyenne par tableau = %d\n", moyen1());
        printf("la moyenne par tableau = %d\n", moyen2());
        return 0;
}
```

Remarque : codage peu lisible de la boucle de lecture de N entiers avec calcul de leur somme :

```
for (i=0, Somme=0; i<N , scanf("%d", &x); Somme += x; i++)  ;
```

Exercices : calcul du minimum et du maximum de N entiers.
Remarque : 2 méthodes :

- lecture et calcul à la volée ou
- lecture et stockage dans un tableau puis calcul.

Remarque : procédures avec passage de paramètre à la C++. Ne pas oublier de nommer le fichier par l'extension "cpp" pour invoquer le compilateur C++.
Dans le compilateur C++, les commentaires //....... sont permis.

```
#include <stdio.h>
#define N  8

void minmax1(int & min, int & max)
{ int i,j;
 for (i=0; i<N; i++)
        {printf("la %deme valeur = ",i);
         scanf("%d",&j);
         if (i==0) min=max=j;
         else {        min= (min < j) ? min : j;
                       max = (max > j) ? max : j;
               }
        }
}

void minmax2(int & min, int & max)
{ int i, min,max,tab[N];
 for (i=0; i<N; i++)
        {printf("la %deme valeur = ",i);
         scanf("%d",&tab[i]);
         }
 min=max=tab[0];
 for (i=1; i<N; i++)
         {min= (min < tab[i]) ? min : tab[i];
          max = (max > tab[i]) ? max : tab[i];
         }
}

int main( void )
{int min,max;
        minmax1(min,max);
        printf("par la methode sans tableau, min= %d, max = %d\n",min,max);
        minmax2(min, max);
        printf("par la methode avec tableau, min= %d, max = %d\n",min,max);
        return 0;
}
```

Exercices : calcul du produit scalaire de deux vecteurs de N entiers.
Remarque : sous forme d'une procédure avec un paramètre.

```
#include <stdio.h>
#define N 8

void produit_scal(int & resultat)
{int i,t1[N], t2[N];
 resultat=0;
 for (i=0; i<N; i++)
        {printf("la %deme valeur du T1 = ",i);
         scanf("%d",&t1[i]);
         }
 for (i=0; i<N; i++)
        {printf("la %deme valeur du T2 = ",i);
         scanf("%d",&t2[i]);
         }
 for (i=0; i<N; i++)
        resultat += t1[i] * t2[i];
}
int main( void )
{ int produit;
  produit_scal(produit);
  printf("produit scalaire = %d\n",produit);
  return 0;
}
```

Tableaux multidimensionnels (matrices)

Exemple :

```
int tab[10];                tab[3] = ...
int matrice[5][5];          matrice[2][3] = ...
char cube[3][3][3];         cube[2][1][2] = ...
```

Exemples et exercices

Exemple-1 Calcul de la somme de deux nombres:

```
void Somme(int X, int Y, int *Z)
{ *Z = X + Y ;}
```

Exemple d'appel:

```
int A , B , C :...
A = ...; B = ...
Somme(A,B, &C);
printf(" %d \n", C);
```

NB: faire le dessin de correspondance des paramètres.
Même exemple sous forme de fonction :

```
int Somme(int X, int Y)
{ return X + Y ;}
```

Exemple d'appel:

```
int A , B , C :...
A = ...;  B = ...
printf("%d \n", Somme(A,B));
```

Exemple-2 Calcul de 2^i :

$U_0 = 1$

$U_n = U_{n-1} * 2$

```
void puiss(int n, int  &  r)
{ int i;
r = 1;
for (i=0; i<= n; i++)           { r = r * 2 ; }
}
```

```
int puiss(int n)
{ int X = 1;
for (i=0; i<= n; i++)  { X = X * 2 ;}
return X;
}
```

```
int puiss(int n)             % récursive
{       if ( n == 0)  puiss = 1;
        else return 2 * puiss(n-1);
}
```

Remarque : donner la solution avec While et do-while

Exemple-3 Calcul de Ui :

U0 = a

Un = 2^n . Un-1 / n

```
void Suite(int a, int n , int  & S)          % calcul progressif de 2^n
{ int I = 0; int S = a; int T = 1;

% I > 0
for (i=0; i<= n; i++)
        {       T = 2 * T;
                S = T * U / I;
        }
}
```

```
int Suite(int a, int n)            % Réutilisation de la fonction  puiss  ci-dessus
{ int i;  int  S = a;
for (i=0; i<= n; i++)   S = puiss(I) * U / I;
return  S ;
}
```

```
void Suite(int a,int n, int & S)             % Récursive + Réutilisation de puiss
{ if ( n == 0 )  S = a;
 else  { T = suite(n-1); S = puiss(n)  * T / n;}
}
```

Remarque: On recalcule 2^n pour chaque pas

Exercice 3 bis: la suite de Syracus (conjecture)

U0 = a

Un = Un-1 / 2 si Un est pair, 1 + 3 * Un-1 sinon

Montrer (par programme) que Un tend vers 1.

```
Vois syracus(int X)
{do
        {if (X % 2 == 0) X = X/2;
         else X = 3*X +1 ;
        }
 while (X != 1);
}
```

Exemple : U0 = 3.   On a la suite : 3 10 5 16 8 4 2 1

Exercices :

1- calculer l'epsilon de la machine sur laquelle vous travaillez.

Indication : partir de X= 1.0 en divisant X par 2 jusqu'à ce que X == X/2

2- Calculer le point fixe de la fonction e^{-X}. C'est à dire, trouver X tel que X = e^{-X}.

Indication : e^A est calculé par la fonction exp(A) de la bibliothèque math de C++.

Exercices sur les Tableaux

Schéma général de parcours d'un tableau

```
int T[Bsup];
int I = 0;
while (I < Bsup)
{       traiter T[I];
        I++;
        }
```

Exemple-4 Afficher les éléments d'un tableau T[N] de caractères

```
void affiche(char T[], int N)
{int i = 0;
while ( i < N)
{       printf("%c", T[i]);
        i = i + 1;
}
{ i >= N  &&  i > 1 }
}
```

Remarque : Solution avec FOR.

Exemple-4bis Mettre tous les éléments d'un tableau d'entiers à 0

Exemple-5 Calculer la somme des éléments d'un tableau d'entiers

```
void Somme(int T[], int N , int & S)
{S = 0;
for (i=0; i<N; i++)
         { S = S + T[i]; }
}
```

Remarque : Solution avec While

Exemple-5bis Calculer le nombre d'éléments nuls, positifs et négatifs d'un tableau d'entiers

Exemple-6 Schéma général de recherche de ELE dans un tableau T[N]

```
#define bool int
#define faux 0
#define vrai 1

int T[Bsup];
int I = 0 ; bool trouve =  faux;
while (( I < Bsup) && (! trouve))
        {       if ( T[I] == X)  trouve = vrai;
                else I++;
          }
          .....
```

Remarque : table de vérité de la boucle.

Exemple-7 Rechercher X dans un tableau T[N] et rendre l'indice

```
bool rech(int T[], int N, int X, int & ind)
{int i = 0; trouve = faux;
while ((i < N) && (! trouve))
        { if ( X == T[i])  trouve = vrai;
          else i = i +1;
        }
{(trouve &  i<N & T[i]=X) | (i == N & ! trouve)}
{ TABLE DE VERITE DE SORTIE }
ind = i;
return trouve;
}
```

Remarque : solution avec For; utiliser break.

La table de vérité :

trouve	I >= N	Traitement
V	V	Impossible
V	F	OK : ind =i
F	V	Pas trouvé
F	F	On tourne !

Remarque : En générale, dans une table de vérité, on considère l'inverse de la condition de l'itération mais on peut aussi raisonner sur la condition de la boucle.

Remarque: On peut utiliser ind localement
Modifier l'algorithme pour trouver la dernière occurrence.

Exemple-8 compter les mots d'une phrase représentée par un tableau T de caractères
- la phrase se termine par un point et contient au moins un mot;
- on considère l'espace comme le seul séparateur et 2 mots sont séparés par un seul espace;
- il n'y a pas d'espace en tète de la phrase ni avant le point final.

Remarque : On n'a pas besoin de la taille N
Le '.' final devient un séparateur particulier

```
void compte(char T[], int & NBM)
{NBM = 0 ; I = 0 ;
while (T[I]  != '.' )
        {if ( T[I] == ' ' )  NBM ++;
         I ++;
        }
 {I>1 & T[I]='.' & NBM = ? }
 NBM++;
}
```

Exemple 8 bis : la fonction *strlen*

```
int strlen (char Ch[])                  // calculer la taille de Ch
{int i = 0;
  while (Ch[i] != '\0') i++;
  return i;
}
```

```
int strlen(char * Ch)
{int i;
 for (i=0; *Ch++ ; i++);
 return i;
}
```

Exemple 9s : la fonction *strcpy*

```
void strcpy(char S1[], char S2[])            // copier S2 dans S1
{int i=0;
  while (S2[i] != 0) {S1[i] = S2[i]; i = i +1 ; }          // le code de '\0' = 0
  S1[i] = 0;
}
```

```
void strcpy(char S1[], char S2[])            // version plus compacte
{while (*S1++ = *S2++) ;}
```

Exerciece : lire K chaînes de caractères (de longueur <= 25) et les stocker dans un tableau de chaînes de caractères (utiliser l'allocation dynamique pour minimiser l'occupation mémoire).

Exemple-9bis Compter les mots d'une phrase représentée par un tableau T[N] de caractères :
- la phrase se termine par un point et contient au moins un mot;
- on considère l'espace comme le seul séparateur et on peut avoir 2 mots séparés par plusieurs espaces.

```
void compte(char T[], int N , int & NBM)
{ NBM = 0; I = 0 ;
  bool MOT = faux;
 while ((I < N) && (T[I] != '.'))
        {       if  ( T[I] == ' ' )
                {       if ( MOT == vrai)   NBM ++;
                        MOT = faux;
                }
                else MOT = vrai;
                I ++;
        }
        {I>1 & I =N & T[I]='.' & (MOT= Faux  | MOT = Vrai) }
if ( MOT)  NBM++;
}
```

Exemple-9bis Avec la même structure de phrase, écrire une procédure qui récupère l'indice du premier caractère du mat ayant au plus L caractères.

Exemple-10 Ecrire la procédure **MOT_SVT(PH, N , Deb, Fin, MOT)** qui récupère le prochain mot de la phrase PH de taille de N caractères à partir de l'indice Deb et range dans Fin l'indice du début du prochain mot s'il existe (Fin=N+1 si plus de mot dans PH).

Exercice sur Enregistrements

Exemple-11

Soit T[N] un tableau déclaré comme suite:

```
typedef struct
        {char nom[12], prenom [12];
         float salaire;
        ...
        } Tpers;
Tpers T[N];
```

Rechercher le rang IND de l'enregistrement E dont le nom = "Dupont" dans T.

Type énuméré

En C : typedef enum {bleu, blanc, vert, rouge} Couleur;
En C++ : enum Couleur {bleu, blanc, vert, rouge};

Remarques :
1- Deux type énumérés ne peuvent pas avoir la même constante :

```
enum Couleur {bleu, vert, orange};
enum Fruit {banane, orange, citron};
```

Ici, la constante *orange* est commune aux deux types.

2- Le compilateur C++ vérifie l'affectation des types énumérés.

```
enum Couleur {bleu, vert, orange};
enum Fruit {banane, raisin, citron};
Couleur C;
Fruit F;
// ERREUR de la compilation :
F = C;
```

3- Le compilateur associe un entier aux constantes d'un type énuméré en commençant par zéro.

```
enum boolean {false, true};          // false =0, true = 1
```

On peut modifier cet ordre en précisant les valeurs :

```
enum mouvement {arriere = -1, arret = 0, avant = 1};
```

Schéma CASE

```
switch (constante)
{case val :     Action; break;     % break pour sortir de CASE (ne pas tester les autres)
 case vali :                              % les deux cas traités de
 case valj :    Action; break;     % la même manière
 case valk :    Action; break;
 default :      Action;
}
```

Exemple 12 : Ecrire la procédure **menu(char & C)** qui affiche l'écran:

A(jout, C(onsultation , S(uppression, F(in
Choix? : _

et récupère une réponse valide dans C (rejette les réponses invalides par un message d'erreur et redemande une nouvelle valeur).

```
    Char Choix;
    do
        effacer Ecran
        Afficher Menu + Choix
        Lire Choix
        switch (Choix)
            {case 'A' : ....;. break;
             case 'C' : .... ; break;
             case 'S' : ..... ; break;
             case'F' : break  ;
             default :  Afficher ERR
            }
   while (Choix !='F');
```

Remarque : Ici, **break** sort de *switch*, pas de la boucle.

Exemple 12 bis : avec un type énuméré

```
   enum Vehicule {voiture, camion, moto, velo};
   Vehicule mobile = ...;
   switch (mobile)
   {case voiture :  printf(" confortable \n"); break;
    case camion :
    case moto : printf(" avec moteur ! \n"); break;
    default : ....
   }
```

Exemple-13 Ecrire la procédure TASS(IND, T , Occupe) qui décale les éléments du tableau T rempli jusqu'à l'indice Occupe (les éléments allant de Occupe+1 à N-1 n'ont pas de signification) vers la gauche à partir de l'élément d'indice IND.T[IND] sera donc remplacé par T [IND+1]... et T[Occupe] se décale vers T[Occupe - 1].

```
void TASS(int Occupe, int IND , int & T[])
{int i;
 for (i = IND; i < Occupe; i++)
        {T[i] = T[i+1]; }
Occupe = Occupe-1;
}
```

Exemple-14 Schéma de HORNER (voir plus loin en récursif)

Allocation dynamique

Rappelons que la taille d'un tableau doit être connue au moment de la compilation.
Pour avoir de la souplesse dans les manipulations des tableaux, on peut utiliser l'allocation dynamique.

Exemple :

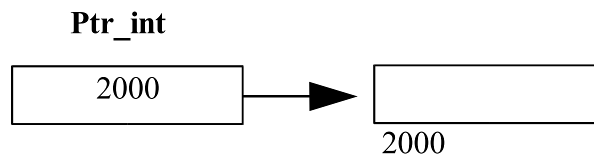
*int * Ptr_int;*

Ptr_int peut contenir l'adresse d'un entier. Mais elle peut aussi correspondre au début d'une zone (début d'un tableau) d'entiers.

On peut allouer un entier (par **new**) dont l'adresse sera rangée dans *Ptr_int.*

Ptr_int = new int;

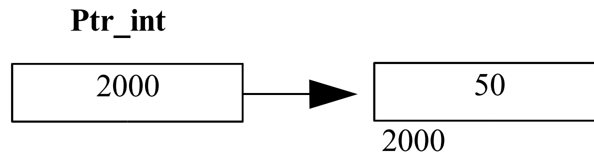
Ptr_int



Ici, on suppose que le gestionnaire de mémoire alloue l'entier demandé à l'adresse 2000. L'adresse de cet entier est rangée dans *Ptr_int*. Il faut noter que l'entier crée par *new* n'a pas de nom et il faut le manipuler à l'aide de *Ptr_int*.
Par exemple :

***Ptr_int = 50;**

Ptr_int

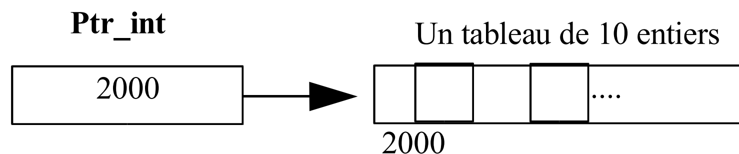


Ptr_int peut également être l'adresse (de début) d'un tableau d'entiers.

Ptr_int = new int [10];

Ptr_int

Un tableau de 10 entiers



Ptr_int est un pointeur qui contient l'adresse du premier élément du tableau :

Ptr_int ≡ &Ptr_int[0]

Important :
Ce qui intéressant avec l'allocation dynamique est de pouvoir manipuler un tableau dont la taille n'est pas connue au moment de la compilation.

Exemple : lire une taille N puis allouer un tableau de taille N :

```
int N, i;
int * Tab;
printf("la valeur de la taille N : ? ");
scanf("%d", &N);                // la taille du tableau
Tab = new int[N];
for(i=0; i< N; i++) scanf("%d", &Tab[i]);            // Tab[i] est UN entier
```

Remarques sur "&" dans l'exemple :

1- Dans

scanf("%d", &N);

on transmet à la fonction scanf l'adresse de la variable N. Car la fonction *scanf* possède un paramètre qui est de type "adresse de .." (ici int *). Ce qui est normal car ce paramètre (N ci-dessus) sera modifié et contiendra la valeur donnée au clavier.

2- De même, dans *scanf("%d", & Tab[i]);*
on transmet à *scanf* l'adresse du ième entier.

3- par contre, si 'argument de *scanf* est de type tableau (par exemple une chaîne de caractères) ou un pointeur, il n'est pas nécessaire d'utiliser "&".

4- il ne faut pas oublier que la lecture de tout un tableau d'entiers n'est pas possible en C/C++. Il faudra lire les éléments un par un.
Par contre, la lecture de tout un tableau de caractères (avec le format "%s") est possible grâce au statut particulier des chaînes de caractères (comme le *string* de Pascal).
De même, on peut écrire un tableau de caractères (avec "%s") mais l'écriture de tout un tableau d'entiers (ou de réels) n'est pas possible. Il faudra écrire les éléments un par un.

Exemple : lecture d'une chaîne de caractère statique, d'une chaîne dynamique et d'un entier accédé par pointeur.

```
char Chaine[10];
char * Ptr_ch;
int * Pi;

scanf("%s", Chaine);            // "&" n'est pas nécessaire

Ptr_ch = new char[25];
scanf("%s", Ptr_ch);            // "&" n'est pas nécessaire

Pi = new int;                   // allocation d'un entier
scanf("%d", Pi);                // "&" n'est pas nécessaire

printf("%s %s", Chaine, Ptr_chaine);      // écriture de toute la chaîne est possible
```

Suppression de zones allouées par new

La fonction **new** permet d'allouer une zone mémoire de taille quelconque.
Il est possible de libérer une zone allouée par la fonction **delete**.

On peut utiliser *delete* sous deux formes : delete et delete[]
La règle d'utilisation de ces deux formes est la suivante :
Si à l'appel de new, on a utilisé des [] (pour allouer un tableau), alors on libère la zone allouée par *delete[]*. Sinon, on utilise *delete*.

Exemple :

```
int *Pi, *TabInt , i;       // déclaration de deux pointeurs et d'un entiers. Remarquer les "*"

Pi = new Pi;
*Pi = 15;

TabInt = new int[*Pi];          // allocation de 15 entiers
for (i=0; i < *Pi; i++)  scanf("%d", &TabInt[i]);

delete Pi;
delete[] TabInt;
```

Exercice : on désire allouer un tableau d'entiers de taille N (N lue au clavier), remplir le tableau puis lire l'entier M au clavier et modifier la taille du tableau en M (M > N).

Cet exercice peut être utile par exemple lors des insertions (ajouts) dans un tableau. Lorsque la taille du tableau initialement alloué ne suffit pas, on peut agrandir celui-ci pour pouvoir continuer à insérer des éléments.

```
int * T1, *T2;      // deux pointeurs pour les tableaux
int N, M;           // les deux tailles.

printf("la taille N ? ");
scanf("%d", &N);
T1 = new int[N];

        /* ......N  insertions dans T1 .... */

/* T1 est complètement rempli et l'on doit augmenter sa taille (de N à M) entiers. */
/* On crée un nouveau tableau par T2, on copie T1 dans T2 puis on libère l'ancien tableau */

 printf("la taille M (M>N) ? "); scanf("%d", &M);
 T2 = new int[M];
 for (i=0; i < N; i++)
        T2[i] = T1[i];              // copie de T1 dans T2
 delete T1;          // le tableau T1 supprimé
 T1 = T2;            // T1 devient le nouveau tableau
/* Continuer à insérer dans T1 */
```

Applications : deux algorithmes de tri

1- méthode de tri par sélection

Pour trier un tableat T[inf .. sup], on trouve Min l'indice du plus petit élément de T[inf .. sup] puis on permute T[min] avec T[inf]. Ainsi T[inf] contient le plus petit élément. Puis on reprend la tri de la même manière sur le tableau T[inf+1 .. sup].
L'algorithme s'arrête quand inf=sup et à chaque étape I, T[inf..I] est trié.

```
#include <iostream.h>

char exemple[] = "asortingexample";

void swap(char a[],int i,int j)
{char temp=a[i];
 a[i] = a[j];
 a[j] = temp;
}

void selection(char a[], int N)
{int i,j,min;
 for (i=0; i<N; i++)
        {min = i;
          for (j=i+1; j < N; j++)
                  if (a[j] < a[min]) min=j;
          swap(a,min,i);
        }
}

int main()
{int i;
 printf(" on va trier = %s \n ",exemple);
 selection(exemple, 15);
 printf(" Le resultat  = %s \n ",exemple);
 return 0;
}
```

Exercice

Ecrire le programme complet qui permet de trouver l'indice de l'élément d'une matrice qui est à la fois le plus grand de sa ligne et le plus petit de sa colonne. Discuter de l'existence d'une solution ainsi que du nombre de solutions possibles selon les valeurs de la matrice.

2- méthode de tri par insertion

Cette méthode est employée par exemple lorsqu'on joue aux carte et on trie sa main.
Dans cette méthode, on insère T[inf+1] à sa place dans T[inf..inf] puis on s'intéresse à T[inf+2] qu'on insère à sa place dans T[inf..inf+1]. A chaque étape où on s'intéresse à T[i], T[inf..I-1] est trié (mais n'est pas la forme finale car des éléments peuvent venir s'y insérer) et on insère T[I] à sa place dans T [inf..I-1].

```
#include <iostream.h>

char exemple[] = "asortingexample";

void insertion(char a[], int N)
{int i,j; char v;
 for (i=1; i< N; i++)
        {v=a[i]; j=i;
         while ((j>0) && (a[j-1] > v))
                {a[j]=a[j-1]; j--;}
         a[j]=v;
        }
}

int main()
{int i;
 printf(" on va trier = %s \n ",exemple);
 selection(exemple, 15);
 printf(" Le resultat  = %s \n ",exemple);
 return 0;
}
```

ANALYSE RECURSIVE

L'analyse récursive est un type important d'analyse qui conduit à un sous problème similaire au problème initial.

Par exemple, le calcul de la dérivée d'une fonction somme de deux fonctions *f* et *g* se décompose en :

- calcul de la dérivée de f
- calcul de la dérivée de g
- calcul de la somme des deux fonctions ainsi obtenues :

$$d(f+g) = d(f) + d(g).$$

On se ramène donc par *récurrence* au même problème portant sur des données plus simples.

La notion d'algorithme (sous forme d'une fonction ou d'une procédure) fournit un bon outil pour résoudre de tels sous problèmes si l'on admet que le texte d'une abstraction contienne un ou plusieurs appels à elle-même.

Une telle construction s'appelle une abstraction RECURSIVE.

Exemple:

La définition récursive suivante de **factorielle** n (n!) :

0! = 1;
n! = (n-1)! * n pour n>= 1

Ce qui donne :

```
int fact (int n)
{
 /* n >= 0, fact(n) = n! */
  if ( n==0) return 1;
 else return ( n *  fact(n-1));
}
```

La fonction *fact* est utilisée dans sa propre définition.
Le calcul de la fonction *fact(n)* nécessite n multiplications : la longueur des calculs n'est donc pas bornée a priori puisqu'elle dépend de n.

L'écriture de la fonction est directement calquée sur l'énoncé du problème (comparer la définition de *fact* et la définition de la fonction *fact*).

Décomposition récurrente

Une décomposition récursive est fondée sur la démarche suivante :

Résoudre le problème dans le cas général en se ramenant aux cas particuliers où la solution est naturellement simple.

Plus précisément, pour résoudre un problème, nous suivons le cheminement suivant :

Trouver au moins un cas où la solution est déjà connue puis tenter de ramener le problème posé pour une valeur quelconque au même problème mais posé sur une valeur plus simple, i.e. plus " proche " du cas connu.

Exemple

calculer la somme des entiers naturels pairs, compris entre 0 et n inclus.

Solution: soit la fonction *som_1_n* répondant à la spécification.

Comment la construire?

- On connaît sa valeur pour certaines valeurs de n :
  som_1_n(0) = som_1_n(1) = 0
  som_1_n(n) n'uest pas définie si n n'est pas un entier naturel.

- Hypothèse à faire pour avancer :
  supposons savoir calculer la valeur de som_1_n(p-1).

On en déduit la valeur de *som_1_n(p)* et la définition de som_1_n :

som_1_n(p)	= 0	si p = 0
	= p + som_1_n(p-1)	si p est pair
	= som_1_n(p-1)	si p est impair

Ce qui se traduit par :

```
int som_1_n(int p)
{
 if (p<0) erreur;                /* arr    t total des calculs */
 else  if ( p==0)  return 0;
       else  if ( (p % 2 == 0)  return  (p + som_1_n(p-1));
             else  return  ( som_1_n(p-1));
}
```

Remarque sur la calculabilité des définitions récursives :

on a fact(n+1) = (n+1) * fact(n) => fact(n) = fact(n+1) / (n+1)
mais l'algorithme correspondant ne s'arrête jamais (incalculable).

Exemples et exercices

Exemple 1- Ecrire une procédure récursive qui affiche les valeurs N..1 d'entier (simulation du schéma "Pour i := N à 1 ").

Dans ce cas, le cas simple (dit cas terminal) est :
* descendant(0) = "ne rien faire "

et le cas récurrent pour nous mener au cas trivial :
* descendant(N) = "action à effet de bord d'écriture"
puis descendant(N-1) si n > 0

La présence de la condition n>0 est d'une importance capitale car sans elle, le calcul ne se terminera pas.

D'uoù la fonction :

```
void descendant(int N)
{if (N > 0 )
	{	écrire(N);		/* affichage quelconque */
		descendant(N-1);
	}
}
```

Trace d'exécution de descendant(4) :

```
descendant(4) :
écrire(4) ;
descendant(3)
            descendant(3) :
                écrire(3) ;
                descendant(2)
                        descendant(2) :
                            écrire(2) ;
                            descendant(1)
                                    descendant(1) :
                                        écrire(1) ;
                                        descendant(0)
                                            descendant(0) :
                                                sortie
                                    sortie (plus de commande
                                        après l'appel récursif)
                            sortie
                sortie
sortie
```

On obtient donc dans l'ordre les valeurs 4, 3, 2 et 1.

Exemple 2- Redéfinir la procédure pour afficher les valeurs de 1 à N (simulation du schéma "Pour i := 1 à N ").

La définition suit la même démarche. L'action d'écriture de N s'effectue après avoir visité les ascendants de celui-ci.

```
void ascendant(int N)
{if (N > 0 )
    {
        ascendant(N-1);
        écrire(N);
    }
}
```

Trace d'exécution de ascendant(4) :

ascendant(4) :

```
ascendant(3) î
écrire(4)       ascendant(2) î
"4"             écrire(3)       ascendant(1) î
sortie          "3"             écrire(2)
                sortie          "2"             ascendant(0) î
                                sortie          écrire(1)       sortie
                                                "1"
                                                sortie
```

Exemple 3- Calcul du pgcd, le plus grand diviseur commun de deux nombres M et N.
Algorithme d'Euclide : on procède aux soustractions successives.

La spécification du pgcd devient plus générale:

pgcd(x,y) = indéfini si x=0 ou y=0
pgcd(x,y) = x si y = x
pgcd(x,y) = pgcd(x-y, y) si x>y
pgcd(x,y) = pgcd(x, y-x) si y>x

```
int pgcd(int x,int y )
{   if ((x==0) || (y==0))
            { printf("pgcd pas défini");return -1; }
    else if ( x==y) return x;
    else if ( x>y)  return pgcd(x-y,y);
    elsereturn  pgcd(x,y-x);
}
```

Exemple 3'- Autre solution : on utilise l'opérateur % (mod) - Algorithme Fast Euclide.

Pour calculer le pgcd, on se ramène par des divisions successives au cas où l'un des nombres est diviseur de l'autre.

Spécification du pgcd(x,y) avec x, y $\neq$ 0 :

pgcd(x,y) = indéfini si x=0 ou y=0
pgcd(x,y) = x si y est multiple de x

pgcd(x,y) = y si x est multiple de y
pgcd(x,y) = pgcd(y, x mod y) si x>y
pgcd(x,y) = pgcd(x, y mod x) si y>x

```
int pgcd (int N, int M )
/* calcul du pgcd de deux entiers N et M /= 0 */
/* On suppose que le pgcd n'est pas défini si l'un des nombres (ou les deux) est nul. */
int  R;
{
        R:= N % M;                              /* R reçoit le reste de (N ÷  M) */
        if (R==0 )          return  M;
         else   return  pgcd(M,R);
}
```

Exemple 4- Ecrire une procédure récursive qui dessine un segment reliant deux points de coordonnées (x1,y1) et (x2,y2). On dispose d'une procédure utilitaire *dessine_point(x,y)* qui dessine un point (x,y). On trace le segment dans le plan XY avec x_i et $y_i > 0$

Pour dessiner un segment, on dessine le point du milieu, puis pour chaque demi segment, on applique la même procédure.

Spécification :

dessine(x1,y1,x2,y2) =
- * si x1≅ x2 et y1≅ y2 => dessine_point(x1,y1) /* les 2 points proches*/
- * sinon
  - xm = (x1+x2)/2
  - ym = (y1+y2)/2
  - dessine_point(xm,ym)
  - dessine(x1,y1,xm,ym)
  - dessine(x2,y2,xm,ym)

```
void dessine(int x1,int y1,int x2,int y2 )
{
        if ( (x1≅ x2) et (y1≅ y2))   dessine_point(x1,y1)        ;
        else
                {       xm = (x1+x2)/2;  ym = (y1+y2)/2; dessine_point(xm,ym);
                dessine(x1,y1,xm,ym);
                        dessine(x2,y2,xm,ym);
                }
}
```

Exemple 5- Schéma de **Horner** :

Calcul de la valeur d'un polynôme de degré N.

Soit P_n un un polynôme de degré N noté par :

$$P_n = a_0 + a_1X^1 + a_2X^2 + ... + a_nX^n$$

On peut décrire ce polynôme par le schéma général :

$$P_n = X(....(X(Xa_n + a_{n-1}) + a_{n-2})....+a_1) + a_0$$

L'adaptation du polynôme ci-dessus au schéma général de Horner donne (a'_i dans S_n correspond à a_{n-i} dans P_n) :

$$S_n = a'_n + a'_{n-1}X^1 + a'_{n-2}X^2 + ... + a'_0X^n$$

On décrit S_n par le schéma général :

$$S_n = X(....(X((Xa'_0 + a'_1)+a'_2)....).....+a'_{n-1}) + a'_n$$

```
                   |--| S0 = a'0
              |------------| S1 = XS0 + a'1
         |--------------------| S2 = XS1 + a'2
```

La définition : $\mathbf{S_0 = a'_0}$

$\mathbf{S_n = X\, S_{n-1} + a'_n}$

Par exemple, le nombre décimal 7534 est représenté par :

10*(10*(10*7+5)+3)+4

où, le nombre étant représenté par un tableau d'entiers A[0..n], X=10, N=3, a'_0=7, a'_1=5, a'_2=3 et a'_3=4.

Pour l'exemple ci-dessus, on a :

```
int A[4];     /* tableau des coefficients */
```

7	5	3	4
0	1	2	3

Pour 7534, on a (N=3, X=10) :

k=0	=> S_0	= a'_0	= A[0]	= 7
k=1	=> S_1	= $a'_0X + a'_1$	= 70 + A[1]	= 75
k=2	=> S_2	= S_1X + A[2]		= 753
k=3	=> S_n	= S_2X + A[3]		= 7534

Ecrire une fonction pour calculer, par le schéma de Horner, la valeur d'un polynôme de degré N. Les coefficients sont donnés dans le tableau A[N]; la base X est fournie.

```
#define N ..
int A[N];            /* tableau des coefficients */

int Horner(int A[],int  k , int X)              /* N non utilisé */
{              /* k>= 0, K<= N */
        if (k == 0)   return  A[k];
         else return ( X * Horner(A, k-1 , X) + A[k]);
}
```

Exemple 5'- Application : calculer en décimal (base 10) la valeur du nombre octal 705 (en base 8).

Exemple 6- Le triangle de Pascal : Déterminer de combien de manières distinctes on peut constituer une équipe de P personnes prises dans un ensemble de N joueurs.

Exemple : Soit {A,B,C,D} l'ensemble des joueurs (N=4).
On veut constituer des équipes de deux joueurs (P=2).

Les équipes seront :
{A,B} , {A,C} , {A,D}, {B,C}, {B,D} , {C,D}

Démarche (pour N et P donnés) :

♦ Cas général :
I- On retire une personne (par exemple A). On calcule le nombre d'équipes de P-1 personnes que l'on peut constituer avec les N-1 joueurs restant. On combinera ces équipes avec A. Ce sont les équipes dont A fait partie.

II- On retire une personne (par exemple A). On calcule le nombre d'équipes de P personnes que l'on peut constituer dans lesquelles A ne figure pas.

Le nombre total d'équipes est I + II .

♦ Cas triviaux : N<P => 0 équipe
P=0 ou N=P => 1 équipe

Le modèle athématique est donné par la formule de triangle de Pascal :

C_n^p = nombre de sous ensembles à P éléments d'un ensemble à N éléments est défini pour N>=0, P>=0 par :

- n<p => $C_n^p = 0$
- p = 0 ou n = p => $C_n^p = 1$
- 0<p<n => $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$

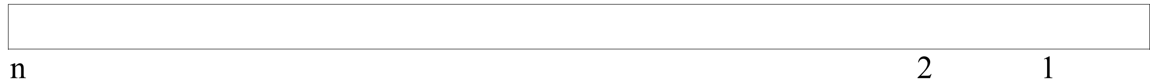
D'où la fonction :

```
int triangle (int n, int p )
{       /* n>=0, p>= 0 */
        if  (n<p)  return 0;
        else if         ((p==0) ||  (n==p))  return 1;
         else return ( triangle(n-1, p) + triangle(n-1, p-1));
}
```

* l'oubli des cas n<p, p=0 ou n = p conduit à la non-terminaison de certains appels à la fonction triangle.

Exemple 7 - La marelle de **Fibonacci** :

Des enfants jouent sur une marelle composée d'une suite de n cases numérotées de n à 1. *n* représente le départ et 1 l'arrivée.



n 2 1

Pour progresser du départ à l'arrivée, ils peuvent sauter à cloche-pied:

- soit d'une case à la suivante (petit saut)
- soit d'une case à la post-suivante (grand saut).

Combien y a-t-il de parcours distincts entre le départ et l'arrivée sur cette marelle?

Tout parcours sur cette marelle est formé (n > 2) :
- soit d'un petit saut suivi d'un parcours sur une marelle à n-1 cases ;

- soit d'un grand saut suivi d'un parcours sur une marelle à n-2 cases ;

Pour n=1, on considère qu'il y a un seul parcours à faire (c'est de rester sur la case 1)

Pour n=2, le premier parcours correspond à faire un petit saut; ce qui nous place dans le cas précédent où le seul parcours est de rester sur place (i.e. pour n=2, il y a deux parcours possibles);

si p(n) représente le nombre de parcours possibles, on a

n = 1 => p(n) = 1
n = 2 => p(n) = 2
n > 2 => p(n) = p(n-1) + p(n-2)

Cette définition de p conduit à la fonction :

```
int p(int n)
{            /*  n > 0 */
        if ( n <= 1)   return 1;
        else   if (  n== 2)  return   2;
        else return  (  p(n-1) + p(n-2));
}
```

Récurrence structurelle

Illustration sur les chaînes :

Une chaîne est soit une chaîne vide (notée "") soit une chaîne obtenue en ajoutant un élément (un caractère) en t te d'une chaîne déjà construite. On note la chaîne non-vide *ch* par $c \oplus ch1$ où c est la t te de la chaîne ch (tete(ch)) et ch1 est le reste de ch (reste(ch)).

De même, un entier naturel est soit 0 soit un entier obtenu en ajoutant 1 à un entier déjà connu. Cette similitude de construction se retrouve au niveau du raisonnement sur les valeurs.

Appelons longueur d'une chaîne le nombre de caractères de la chaîne. Pour définir une fonction *length* calculant la longueur d'une chaîne, nous sommes amenés à faire le raisonnement suivant.

D'une part, la valeur de length ("") ne peut être que 0.

D'uautre part, supposons savoir calculer la longueur L d'un chaîne ch, la longueur de la chaîne ($c \oplus ch$) est L + l.

Ce raisonnement ressemble fort à un raisonnement par récurrence mais au lieu d'être effectué sur des entiers, il est effectué sur des chaînes. Ce raisonnement utilise un principe de récurrence très général appelé *le principe de récurrence structurelle.*

Enonçons le sur des chaînes:

Soit P une propriété définie sur les chaînes :

1- si P("") est vérifiés

2- si, de l'hypothèse "p est vérifiée par ch ", *on peut déduire la conclusion "quelle que soit* c*, P est vérifiée par* (c ⊕ ch)*Ó alors quelle que soit* ch*, P(* ch*) est vérifiée.*

A l'aide de ce principe, montrons que la fonction *length* ci-dessous calcule effectivement la longueur de son argument :

```
int length(char ch[])
{
	if ( ch = "")   return  0;			/* égalité de chaînes avec strcmp */
	else
		/* ch = c ⊕ ch1, ch1 obtenu par une fonction Reste(ch) */
		return ( 1 + length(reste(ch)));
}
```

Soit P la propriété " length(ch) est égal au nombre de caractères de ch ".   P("") est vérifiée. De plus, P(ch) est vérifiée, le nombre de caractères de (c ⊕ ch1) est égal à 1+length(ch1) donc à length(c ⊕ ch1). Par suite, P(c⊕ch1) est aussi vérifiée et la propriété P est vraie pour toute chaîne.

La fonction length calcule bien le nombre de caractères d'une chaîne, nombre qui est, par définition la longueur de cette chaîne.

Cas des tableaux :

utilisation des tableaux comme des structures "faussement" récursives.

On peut considérer un tableau par la définition suivante :

<tableau> :: vide | un_élément <tableau>.

La structure est dite "faussement" récursive car la propriété "être vide" se vérifie par la manipulation des indices et non pas, comme pour les chaînes, par un test du contenu de la structure qui dénoterait sa vacuité par une valeur (e.g. une constante).

Application

1- Recherche du plus grand élément d'un tableau d'entiers T. Le tableau T est supposé non vide.

Spécification : pour un tableau T(inf..sup) où inf <= sup

Maximum(T,Inf,Sup) = indéfini si Inf > Sup
Maximum(T,Inf,Sup) = T(Inf) si Inf = Sup
Maximum(T,Inf,Sup) = Max(T(Inf),Maximum(T,Inf+1,Sup)) si Inf < Sup

Avec : Max(x,y) = x si x> y
Max(x,y) = y sinon

Algorithme :

```
int Maximum(int T[], int  Inf, int Sup)
{int M;
	if (Inf > Sup)   erreur
	else  if ( Inf == Sup)   return (T[Inf]);
	 else
	{
		M =  Maximum(T, Inf+1, Sup);
		if (T(Inf) > M)   M  =  T[Inf];
		return M;
	}
}
```

Exercices

1- Ecrire une procédure récursive qui détermine le rang K d'une valeur X dans un tableau T d'entiers de taille N.

2 - Ecrire une procédure récursive qui affiche les éléments d'un tableau de caractères T[N]. Réécrire l'algorithme pour afficher les caractères de ce tableau en commençant par le dernier.

2'- trouver le minimum et le maximum des éléments d'un tableau récursivement.

3- Définir si un mot, représenté par un tableau M de caractères, est un Palindrome (exemple de palindromes : radar, elle, tôt, ADA, laval, sus).

On suppose donnée une fonction *Longueur(Ch)* qui donne la longueur effective de la chaîne Ch.

```
#include <stdio.h>
#include <string.h>
#define vrai 1
#define faux 0

int Palindrome(char M[] ,int  debut,int  fin )
            /* on ne traite pas le cas d'espace dans M */
{       if (debut < fin)
            if (M[debut] == M[fin])
                  return (Palindrome(M,debut+1, fin-1));
             else return  faux;
        else return vrai;
}

int main( void )
{       char chaine[20];
        printf("une chaine ?\n");       scanf("%s",chaine);
        if (Palindrome(chaine, 0,strlen(chaine) -1))
            printf("ok\n");
        else printf("non\n");
        return 0;
}
```

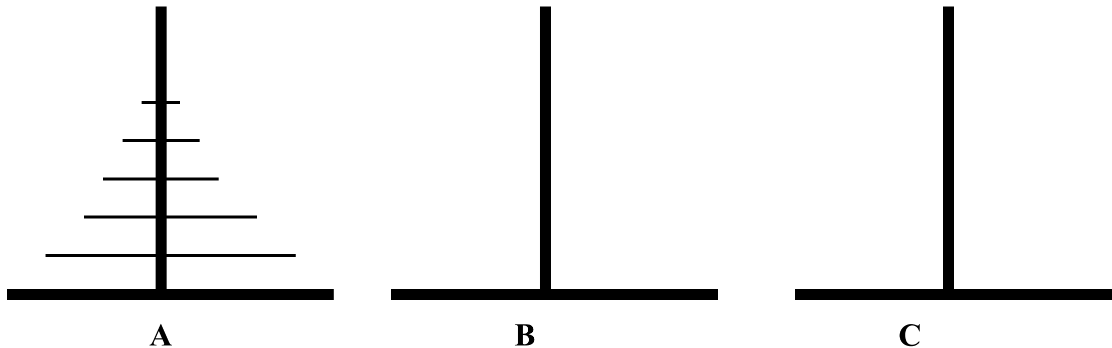
4- Les tours de Hanoï.

D'après une légende, pour entretenir une activité cérébrale, des moines chinoise ont entrepris la tâche suivante :

Soient 3 pieux A, B et C et N disques circulaires de dimension différentes. Au départ, les N disques sont placés sur le pieu A de telle sorte que le plus grand soit sous les autres puis le suivant etc. et le plus petit au sommet. Etant donnés deux autres pieux B et C, le but est de déplacer les N disques sur C en se servant de B et en respectant les contraintes suivantes :

- on ne déplace qu'un disque à la fois
- on ne place jamais un disque plus grand sur un disque plus petit.

Pour N = 5, on a :



Le but est de ramener les 5 disques sur C en se servant de B comme poteau intermédiaire.
NB : il y a 2N -1 déplacements corrects (si l'on ne se trompe pas, sinon, beaucoup plus)
Le programme suivant donne la séquence des déplacements pour N=5.
Raisonnement : déplacer 4 disques de A sur B en se servant de C, puis placer le 5e de A sur C. Il reste à faire la même opération avec les 4 disques restants qui sont sur B vers C en se servant de A. Pour se faire, déplacer 3 disques de B sur A (en se servant de C qui contient le plus grand disque) puis transférer le disque 4 sur C...

```
#include <iostream.h>
int T[5]={1,2,3,4,5};            // les numéros = les diamètres

void hanoi(int N, char dep, char arr, char inter)
{if (N>=0)
        {hanoi(N-1, dep, inter, arr);
         printf(" %d  de %d  -->  %d \n, T[N], dep, arr);
         hanoi(N-1, inter, arr, dep);
        }
}

int main()
{hanoi(4,'A','C','B');
}
```

Résultat pour N=4 (5 disques), lire de gauche à droite et de haut vers le bas :

1 de A --> C
2 de A --> B

1 de C --> B
3 de A --> C
1 de B --> A
2 de B --> C
1 de A --> C
4 de A --> B
1 de C --> B
2 de C --> A
1 de B --> A
3 de C --> B
1 de A --> C
2 de A --> B
1 de C --> B
5 de A --> C Ici, le disque 5 est transféré sur C , on recommence avec les 4 autres.

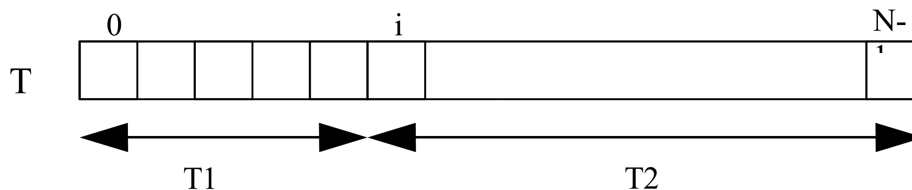
1 de B --> A
2 de B --> C
1 de A --> C
3 de B --> A
1 de C --> B
2 de C --> A
1 de B --> A
4 de B --> C Ici, le disque 4 est transféré sur C , on recommence avec les 3 autres.

1 de A --> C
2 de A --> B
1 de C --> B
3 de A --> C
1 de B --> A
2 de B --> C
1 de A --> C

5- Ecrire l'algorithme récursif de **tri par insertion** :

Méthode : pour trier T[inf .. sup], on trie T[inf+1 .. sup] puis on y insère T[inf] à sa place. De même, pour trier T[inf+1 .. sup], on trie T[inf+2 .. sup] et on y insère T[inf]. L'algorithme continue récursivement jusqu'à inf=sup, puis, par les retours arrières sur des appels récursifs, on insère les éléments.

Explication : on veut trier un tableau de N entiers par la méthode " tri par insertion ". Selon cette méthode, le tableau T à trier est considéré comme deux sous-tableaux T1= T[0..i-1] et T2= T[i.. N-1]. T1 est la partie triée et T2 reste à trier. Le but est de trier T en insérant à chaque étape T[i] à sa place dans la partie déjà triée T1. Par exemple, on procède ainsi pour trier une "main" dans un jeu de cartes. On commence avec T1 contenant le premier élément de T et T2 le reste de T. L'élément T[i] est insérés dans le tableau T[0..i] (T1 plus la case T[i]). Les éléments de T2 sont ainsi inséré un par un dans T1 (par la procédure **insère**). A chaque étape, i est incrémenté, la taille de T1 augmente de 1 et celle de T2 diminue de 1. La procédure de tri (procédure **tri**) s'arrête lorsque T2 est devenu vide; T1 est alors le résultat sous la forme d'un tableau trié.



Ecrire les procédures **insère** et **tri_ins** pour réaliser un tri par insertion. On remarque qu'une solution récursive est plus simple à trouver (20 points)

Exemple : T=<9,8,5,2,4,7,3,1> i=0, T1=<>, T2=T

1ère étape : T1<9>, T2=<8,5,2,4,7,3,1> T1 contient le 1er élément (trié)
2nde étape : T1<8,9>, T2=<5,2,4,7,3,1> On insère 8 à sa place dans T1
3ème étape : T1<5,8,9>, T2=<2,4,7,3,1> Un par un, les éléments sont
4ème étape : T1<2,5,8,9>, T2=<4,7,3,1> insérés dans T1
5e étape : T1=<2,4,5,8,9>, T2=<3,1>
6e étape : T1=<2,3,4,5,7,8,9>, T2=<1> Le dernier élément inséré sera 1
7e étape : T1=<1,2,3,4,5,7,8,9>, T2=<>

```
#include <iostream.h>
#define N 10

void echange(int  T[], int i, int j)
{int tmp=T[i]; T[i]=T[j]; T[j]=tmp;}

void insere(int  T[], int inf, int sup)
{if (inf < sup)
        {if (T[inf] > T[inf+1])
                {echange(T, inf,inf+1);
                 insere(T, inf+1, sup);
                }
        }
}

void tri_ins(int  T[], int inf, int sup)
{if (inf < sup)
        {tri_ins(T, inf+1, sup);
         insere(T, inf, sup);
        }
}
int main()
{int tab[N]={5,3,1,7,4,3,9,2,19,6};
 for(int i=0;i<N;i++)
        printf(" %d \n ", tab[i]);
 tri_ins(tab,0,9);
 for(i=0;i<N;i++)
        printf(" %d \n ", tab[i]);
 return 0;
}
```

Gestion des fichiers en C

En C , on dispose des fonctions de la bibliothèque standard pour manipuler les fichier comme vu dans la partie algorithmique. Il est aussi possible d'effectuer des traitements plus avancés comme l'écriture à la fin d'un fichier, le positionnement à l'intérieur d'un fichier, le stockage d'informations de types variés au sein d'un même fichier, etc.

Pour manipuler des fichier en C, comme dans tous les langages. il faut d'abord associer un "nom logique" au "nom physique" du fichier, puis utiliser le nom logique dans les fonctions d'entrée/sortie de la bibliothèque standard.

Nom physique et nom logique

Le nom physique est le nom utilisé par le système d'exploitation (ici le DOS) pour localiser un fichier (unité de disque. répertoires. fichier)

Un exemple de nom physique est : c:\TDlA\TD1\monTD.cpp

Le nom logique est une variable de type prédéfini FILE * (pointeur sur FILE), le nom logique est aussi appelé un **flot.**

Fonctions de la bibliothèque standard

Les principales fonctions d'E/S sur disque sont:

Ouverture d'un fichier

```
FILE *fopen(char * nom_fichier, char * mode)
```

Cette fonction permet d'associer le fichier de nom physique *nom_fichier* avec un nom logique dans le mode d'utilisation *mode.* Le paramètre *mode* peut prendre les valeurs suivantes

"r" ouverture en lecture, positionnement au début, le fichier doit déjà exister
"w" ouverture en écriture, positionnement au début, le fichier est créé s'il n'existait pas
"a" ouverture en écriture, positionnement à la fin

La fonction renvoie 0 Si le fichier ne peut pas être ouvert.

Exemple:

```
FILE *f ;
f = fopen("monfichier","r");
if (f == NULL)
        /*problème */
else
        /* on continue*/
```

Fermeture d'un fichier

```
flose(FILE *flot);
```

Cette fonction ferme le fichier repéré par le nom logique *flot*
Exemple: fclose(f);

Lecture dans un fichier

```
size_t fread(void *ptr, size_t taille, size_t nbelems, FILE *flot)
```

valeur de retour : nombre d'éléments effectivement lus (size_t = unsigned int)

ptr	adresse de la zone mémoire destinataire
taille	taille en octets de l'élément à lire
nbelems	nombre d'éléments à lire
flot	nom logique du fichier à lire

size_t est un type entier

Exemple:

```
struct elem
    {int numero;
     int code;
     float taille;
    };
typedef elem tab[10];

FILE *f;
tab t;

fread(t, sizeof(elem), 10,f);     /* sizeof renvoie la taille de son paramètre */
            .......
```

Lectures formatées dans un fichier

```
int fscanf(FILE *flot, char * format, void *advariablel,.. ,void * advariablen)
```

cf. scanf avec en plus l'indication de flot de lecture
NB. fscanf renvoie la constante EOF en cas de détection de la marque fin de fichier

Exemple:

```
int n;
fscanf(f,"%d" ,&n);
```

```
int getc(FILE *flot);
```

cf. getchar avec en plus l'indication du flot de lecture

Ecriture dans un fichier

```
size_t fwrite(void * ptr, size_t taille, size_t nbelems, FILE *flot)
```

cf. fread

Ecritures formatées dans un fichier

```
int fprintf(FILE *flot, char *format,.....)
```

cf printf avec en plus l'indication du flot d'écriture

```
int fputc(int c, FILE *flot)
```

cf. putchar

Test de fin de fichier

```
int feof(FILE *flot)
```

renvoie zéro si la fin du fichier n'est pas atteinte, 1 si elle est atteinte.

NB. Pour utiliser ces fonction il faut inclure le fichier entête stdio.h à l'aide de la directive
#include **<stdio.h>**

Exemple d'utilisation: Duplication d'un fichier contenant des entiers

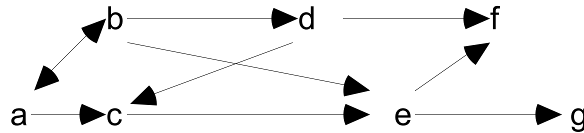
```
#include <stdio.h>

void main()
{
FILE *fs,*fd;
int x;
if (fs = fopen("source","r"))
	{
	fd = fopen("destination",  "w");
	while(fscanf(fs,"%d",&x) != EOF)
		fprintf(fd,"%d",x);
	fclose(fs);
	fclose(fd);
	}
else
	printf("impossible d'ouvrir le fichier source\n");
}
```

Travail à rendre

Un graphe est un ensemble de sommets et d'arcs qui relient ses sommets entre eux. Par exemple, le réseau routier peut être représenté par un graphe où les noeuds sont les villes et les arcs les routes qui les relient.

Exemple : dans le graphe suivant, a,b...g sont des noeuds.



On distingue différents types de graphes. Pour cet exercice, nous différencions les graphes de types suivants :

- graphe orienté : est un graphe où l'arc reliant deux noeuds a une direction précise. Dans l'exemple ci-dessus, les arcs sont orientés (certains ont une double direction).

- graphe cyclique (avec circuits) : est un graphe où il existe un chemin d'un noeud à lui même (sans utiliser la réflexivité)

- graphe valué : est un graphe où les arcs portent une valeur. Pour l'exemple du réseau routier, la valeur d'un arc peut être la distance qui sépare les deux villes reliées par cet arc.

Pour cet exercice, on représente le graphe par une matrice M de booléens où les lignes et les colonnes représentent les villes. On note M(i,j) = V s'il y a un arc reliant la ville i à la ville j, F sinon. De plus, on suppose qu'il n'y a pas de réflexivité, c'est à dire, $\forall i$, M(i,i) = F.

Noeuds	a	b	c	d	e	f	g
a	F	V	V				
b	V	F		V	V		
c			F		V		
d			V	F		V	
e					F	V	V
f						F	
g							F

On remarque que l'on peut avoir M(i,j)=V sans avoir forcément M(j,i)=V.

* Ecrire la procédure récursive *chemin(départ, arrivée)* qui détermine s'il y a un chemin entre les points départ et arrivée.
* Discuter des propriétés du graphe et de l'existence d'une solution (graphe orienté, cyclique);
* Pour un chemin existant, préciser la liste des points de passage.

* Proposer une structure pour évaluer les arcs et modifier les algorithmes pour proposer le chemin le plus court entre deux points départ et arrivée.

Suite : Listes, ...AEF, Analyseur syntaxique, ...