Cours de Programmation II (*Module M21*)

[Filière SMI, Semestre 4]
Département d'Informatique
Faculté des sciences de Rabat

Par
B. AHIOD
(ahiod@fsr.ac.ma)
2014-2015

Objectifs

- **Approfondir les connaissances de la programmation en langage C :**
  - pointeurs, fonctions et chaînes de caractères
  - enregistrements et fichiers
  - …
- **Utiliser le langage de programmation C pour implémenter :**
  - les structures de données
  - les algorithmes qui les manipulent
  - …

Pré-requis

- **Notions de base d'algorithmique [*Algorithmique I (S2), Algorithmique II (S3)*] :**
  - Conception d'algorithmes itératifs et récursifs
  - Analyse de complexité d'algorithmes
  - Structures de données élémentaires
  - …
- **Programmation en langage C [*Programmation I (S3)*] :**
  - Programmation structurée
  - Notions de tableaux, de fonctions, …
  - Manipulation des pointeurs et allocation dynamique
  - …

Contenu

Introduction :
- **Algorithme vs Programme, Itératif vs Récursif, …**

Rappels :
- **Phases de programmation en C, structure et composants d'un programme en C, types de base, instructions, …**

Pointeurs et allocation dynamique :
- **Pointeurs et tableaux, pointeurs et fonctions, allocation de mémoire , …**

Types structures, unions et synonymes :
- **Notion de structure, union et type synonyme, énumérés, structures auto-référentielles, …**

Chaînes de caractères :
- **Définition, manipulation, tableaux de chaînes de caractères, …**

Structures de données linéaires en C :
- **type abstrait de données, structure de données, implémentation en C, exemples et applications (piles, files, listes)**

Fichiers :
- **Types de fichiers (textes et binaires), accès (séquentiel et direct), manipulation (ouvrir, fermer, lire, écrire)**

Compléments :
- **Compilation séparée, directives du préprocesseur , …**

Introduction

- ✓ **Algorithme vs Programme**
- ✓ **Itératif vs Récursif**
- ✓ **…**

Notion de programme

Algorithmes + structures de données

=

Programme

[*Wirth*]

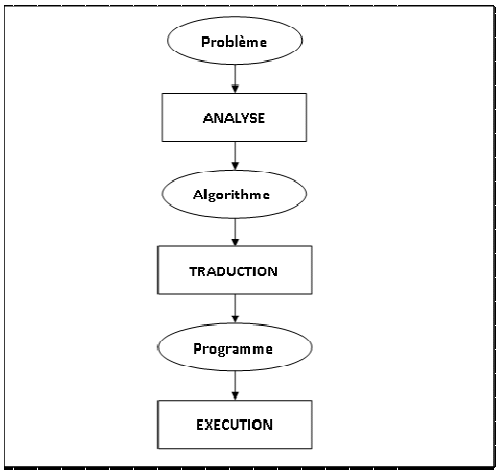
- **Un programme informatique est constitué d'algorithmes et de structures de données manipulées par des algorithmes**

Notion de programme

- **Synonymes**
  - Programme, application, logiciel
- **Objectifs des programmes**
  - Utiliser l'ordinateur pour traiter des données afin d'obtenir des résultats
  - Abstraction par rapport au matériel
- **Un programme est une suite logique d'instructions que l'ordinateur doit exécuter**
  - Chaque programme suit une logique pour réaliser un traitement qui offre des services (*obtention des résultats souhaités à partir de données*)
  - Le processeur se charge d'effectuer les opérations arithmétiques et logiques qui transformeront les données en résultats
- **Programmes et données sont sauvegardés dans des fichiers**
  - Instructions et données doivent résider en mémoire centrale pour être exécutées

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 7

De l'Analyse à l'Exécution



[SMI4-fsr] Programmation II (M21-S4) 2014-2015 8

Notion d'Algorithme

- **Origine :**
  - Le terme algorithme vient du nom du mathématicien Al-Khawarizmi (*820 après J.C.)*
- **Définition :**
  - Un algorithme est une suite finie de règles à appliquer dans un ordre déterminé à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat, et cela indépendamment des données
- **Rôle fondamental :**
  - Sans algorithme il n'y aurait pas de programme
- **Un algorithme est indépendant :**
  - de l'ordinateur qui l'exécute
  - du langage dans lequel il est énoncé et traduit

Spécifier/Exprimer/Implémenter un algorithme

- **Spécification d 'un algorithme :**
  - ce que fait l'algorithme
  - cahier des charges du problème à résoudre
- **Expression d 'un algorithme :**
  - comment il le fait
  - texte dans un pseudo langage
- **Implémentation d 'un algorithme :**
  - traduction du texte précédent
  - dans un langage de programmation

Exemple : Recherche d'un élément

Spécification d'un algorithme

```
/* Cet algorithme recherche la place d'un élément val
   dans un tableau tab contenant n éléments */
```

```
Algorithme recherche_sequentielle(tab: entier[]; n, val: entier) : entier
 entrées : tab, n et val
 sortie : indice de val dans le tableau tab, sinon -1
Début
  variables locales : i: entier
  i ← 0;                                          // 1
  tant que ((i<n) et (tab[i] <> val)) faire       // 2
       i ← i+1                                    // 3
  ftq                                             // 4
  si (i = n) alors retourner -1                   // 5
  sinon retourner i                               // 6
Fin
```

L'algorithme traduit en C

L'algorithme en pseudo code

```
int recherche_sequentielle(int *tab, int n, int val) {
 int i;                                           // 1
 i = 0;                                           // 2
 while ((i<n) && (tab[i] != val))                 // 3
    i ++;                                         // 4
 if (i == n)                                      // 5
   return(-1);                                    // 6
 else return(i);                                  // 7
}
```


Analyse descendante

- **Consiste à décomposer un problème en sous problèmes, eux-mêmes, à décomposer en sous problèmes, et ainsi de suite jusqu'à descendre à des actions dites primitives**
  - ***Les étapes successives de décomposition donnent lieu à des sous algorithmes pouvant être considérés comme des actions dites intermédiaires***
  - ***Ces étapes sont appelées fonctions ou encore procédures***

Notion d'algorithme récursif

- **Un algorithme est dit *récursif* lorsqu'*il s'appelle lui même* de façon directe ou indirecte.**

- **Pour trouver une solution récursive d'un problème, on cherche à le décomposer en plusieurs sous problèmes de même type, mais de taille inférieure.**

  On procède de la manière suivante :

  – **Rechercher un** (*ou plusieurs*) **cas de base** et sa (*ou leur*) solution (*évaluation sans récursivité*)

  – **Décomposer le cas général** en cas plus simples eux aussi décomposables pour *aboutir au cas de base*.

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 13

Itératif vs Récursif (Exemple)

```
/* Calcul de la somme des carrés des entiers entre m et n (version itérative) */
Algorithme SommeCarres_iter(m: entier; n: entier) : entier
 entrées : m et n
 sortie : somme des carrés des entiers entre m et n inclus,
          si m<=n, et 0 sinon
Début
  variables locales : i, som: entier

  som ← 0                          // 1
  pour i de m à n faire            // 2
       som ← som + (i*i)           // 3
  fpour                            // 4
  retourner som                    // 5
Fin
```

```
/* Calcul de la somme des carrés des entiers entre m et n (version récursive) */
Algorithme SommeCarres_rec(m: entier; n: entier) : entier
 entrées : m et n
 sortie : somme des carrés des entiers entre m et n
 pré-condition : m<=n
Début
  si (m<>n) alors                                  // 1
       retourner ((m*m)+SommeCarres_rec(m+1,n)     // 2
  sinon                                            // 3
       retourner (m*m)                             // 4
  fsi
Fin
```

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 14

Programmation Procédurale vs Programmation Orientée-Objet

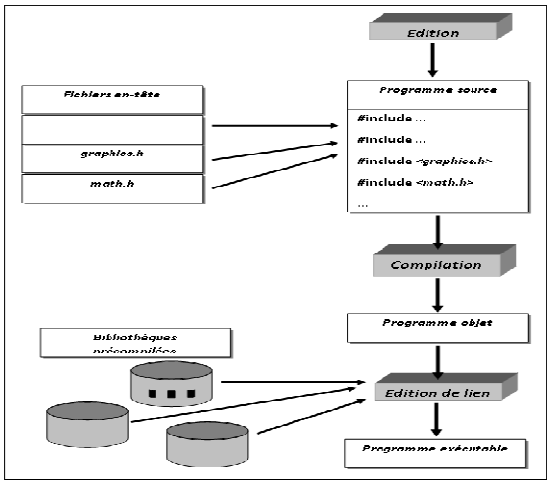
- **Programmation Procédurale :**
  - Centrée sur **les procédures** (ou **opérations**)
  - Décomposition des fonctionnalités d'un programme en procédures qui vont s'exécuter séquentiellement
  - Les données à traiter sont passées en arguments aux procédures
  - Des langages procéduraux : **C**, **Pascal**, **…**
- **Programmation Orientée-Objet :**
  - Centrée sur **les données**
  - Tout tourne autour des "**objets**" qui sont des petits ensembles de données représentants leurs propriétés
  - Des langages orientés-objets : **C++**, **Java**, **…**

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 15

Rappels

- ✓ **Phases de Programmation en C**
- ✓ **Structure de Programme en C**
- ✓ **Types de Base des Variables**
- ✓ **Instructions**
- ✓ **Pointeurs et Allocation Dynamique**
- ✓ **…**

Phases de programmation en C



Un programme en C

```
                                    /* Exemple de programme en C */
#include <stdio.h>
#include <math.h>
#define NFOIS 5

int main() {
 int i ;
 float x ;
 float racx ;

 printf ("Bonjour\n") ;
 printf ("Je vais vous calculer %d racines carrées\n", NFOIS) ;

 for (i=0 ; i<NFOIS ; i++) {
  printf ("Donnez un nombre : ") ;
  scanf ("%f", &x) ;
 if (x < 0.0)
    printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;
 else {
   racx = sqrt (x) ;
  printf ("Le nombre %f a pour racine carrée : %f\n", x, racx) ;
 }
}
printf ("Travail terminé - Au revoir") ;
return 0;
}
```

Composantes d'un Programme en C

- En ***C***, les programmes sont composés essentiellement de ***fonctions*** et de ***variables.***
- ***Définition d'une fonction en C :***

```
<TypeRésultat> <Nomfonction> (<TypePar1>, <TypePar2>, …)
{
 <déclarations locales> ;
 <instructions> ;
}
```

- En ***C***, toute instruction simple est terminée par un point virgule (**;**).

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 19

Composantes d'un Programme en C

- ***La fonction main :***
  - Une fonction et une seule s'appelle **main.**
  - C'est la fonction principale des programmes en ***C*** ; elle se trouve obligatoirement dans tous les programmes.
  - L'exécution d'un programme entraîne automatiquement l'appel de la fonction **main**.
- ***Les variables :***
  - Contiennent les ***valeurs*** utilisées pendant l'exécution du programme.
  - Les noms des variables sont des ***identificateurs*** quelconques.
  - Toute variable doit être déclarée avant les instructions et son ***type*** spécifié dès la déclaration.

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 20

Types base des variables en C

- **Toutes les variables doivent être explicitement typées**
- **Types de base des variables :**
  - Les entiers : **int, short int, long int**
  - Les réels : **float, double, long double**
  - Les caractères : **char**

Exemples :

```
short int mon_salaire;
double cheese;
char avoile;
```

- **Remarque :**
  - La présence d 'une ou plusieurs étoiles devant le nom d'une variable indique un pointeur.
  - Exemple :

```
double **mat;
        // permet de déclarer une matrice (tableau à deux dimensions)
```


Instructions

- **Tests :**

```
If (expression) {bloc} else {bloc};

switch (expression)
{
case const1: instructions; break;
case const2: instructions; break;
..
default: instructions;
}
```

- **Boucles :**

```
while (expression) {instructions;}

for (expr1 ; expr2 ; expr3) { instructions;}

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

- **Quitter une boucle (for, do, while) ou un switch :**

```
break;
```

- **Passer à l'itération suivante, mais ne quitte pas la boucle :**

```
continue:
```

- **…**

Pointeurs & Allocation dynamique

- ✓ **Pointeurs & Tableaux**
- ✓ **Pointeurs & Fonctions**
- ✓ **Allocation de mémoire**
- ✓ **…**

Pointeurs & Tableaux

- **Notion de tableau :**
  - Un tableau est *une variable structurée* formée d'un ensemble de variables du même type, appelées ***les composantes du tableau***.
  - Chaque élément est repéré par ***un indice*** précisant sa position.
  - Le nom du tableau est son ***identificateur***.

- **Tableaux à une dimension (vecteurs)**
  - ***Déclaration en C :***

    <TypeSimple> <NomTableau> [<NombreComposantes>] ;

    Exemple :

    float B[200] ;

  - ***Mémorisation :***
    - Les éléments d'un tableau sont rangés à des adresses consécutives dans la mémoire.
    - le nom du tableau est *le représentant de l'adresse du premier élément.*
    - Si un tableau est formé de N composantes, chacune ayant besoin de M octets en mémoire, alors le tableau occupera (N * M) octets.
    - L'adresse de la composante numéro i de du tableau A se calcule :

      A + (i * taille-de-la-composante)

  - ***Accès aux composantes d'un tableau :***
    - Pour accéder à un élément on utilise ***un indice*** selon la syntaxe suivante :

      <Nomtableau> [<indice>]

      où <indice> : expression entière positive ou nulle.

    Exemple :

    Pour un tableau **T** de **N** composantes :
    - ***l'accès au premier élément*** se fait par **T[0]**
    - ***l'accès au dernier élément*** se fait par **T[N-1]**

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 24

Pointeurs & Tableaux

- ***Tableaux à deux dimensions (matrices) :***
  - **Déclaration en C :**
    - <TypeSimple> <NomTableau> [NombreLignes] [NombreColonnes] ;

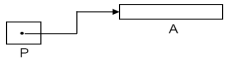
    Exemple :

    int A[10][20] ; /* matrice de 200 entiers (ayant 10 lignes et 20 colonnes */
  - **Accès aux composantes :**

    <NomMatrice> [<Ligne>] [<Colonne>] ;
  - Pour une matrice **M** formée de **L** lignes et **C** colonnes :
    - La ***première*** composante de la matrice est **A[0][0]**
    - La composante de la ***L**ème ligne* et ***C**ème colonne* est notée : **A[L-1][C-1]**

Pointeurs & Tableaux

- **Notion de pointeur :**
  - Un pointeur est une variable qui peut ***contenir l'adresse d'une autre variable***.
  - Si un pointeur P contient l'adresse d'une variable A, on dit que '***P pointe sur** A*':



- **Déclaration d'un pointeur :**

  <type> *<NomPointeur> ;

  Exemple :

  int *Pnum ;

  On dira que :
  - " ***Pnum est du type int** ", ou bien
  - " *Pnum est un pointeur sur **int** *", ou bien
  - " *Pnum peut contenir l'adresse d'une variable du type **int** *"

  Pnum ne pointe sur aucune variable précise : Pnum est ***un pointeur non initialisé***.

  Soit la déclaration : **int** A ;
  - L'initialisation du pointeur Pnum avec la variable se fait par :
  - Pnum = &A ; /* adresse de la variable A */
  - **Un pointeur est lié explicitement à un type de données**. Ainsi, Pnum ne peut recevoir l'adresse d'une variable d'un autre type que **int**.

Pointeurs & Tableaux

- **Un tableau est une zone mémoire qui peut être identifiée par l'adresse du 1er élément du tableau**
- ***Adressage des composantes d'un tableau :***
  - La déclaration : **int** A[10] ;
    définit un tableau de 10 composantes : A[0], A[1], …, A[9]
  - Si p est pointeur d'entiers déclaré par : **int** *p ;
    alors,
    l'instruction : p = A ; est équivalente à : p = &A[0] ;
    p *pointe sur* A[0]
    *(p+1) *désigne le contenu de* A[1]
    *(p+2) *désigne le contenu de* A[2]
    ...
    *(p+i) *désigne le contenu de* A[i]

- **Dans une expression, une écriture de la forme Expr1[Expr2] est remplacée par :**
  *((Expr1) + (Expr2))

- **Il existe une différence entre un pointeur P et le nom d'un tableau A :**
  - ***Un pointeur est une variable*** , donc les opérations comme P = A ou P++ sont permises.
  - ***Le nom d'un tableau est une constante***, donc les opérations comme A = P ou A++ sont impossibles.

Pointeurs & Tableaux

```
/* Exemple : Lecture et affichage d'une matrice */
#include <stdio.h>
#define n 4
#define p 10
main() {
 float A[n][p] ;
 float *pA ;
 int i, j ;
  /* lecture d'une matrice */
 pA = &A[0][0] ;                    /* ou bien pA = (float *) A ; */
 for (i = 0 ; i<n ; i++)  {
     printf("\t ligne n° %d\n", i+1) ;
     for (j = 0 ; j<p ; j++)
        scanf("%f", pA + i * p + j) ;
  }
 for (i = 0 ; i<n ; i++) {  /* 1ère façon : affichage */
     for (j = 0 ; j<p ; j++)
        printf("%7.2f", *(pA + i * p + j)) ;
     printf("\n") ;
  }
 for (i = 0 ; i<n ; i++)  {  /* 2ème façon : affichage */
     pA = &A[i][0] ;
     for (j = 0 ; j<p ; j++)
        printf("%7.2f", pA[j]) ;
     printf("\n") ;
  }
 return 0 ;
}
```

Pointeurs & Fonctions

- En ***C***, la structuration d'un programme en ***sous-programmes*** (***modules***) se fait à l'aide de ***fonctions***
- **Notion de fonction :**
  - Une fonction est définie par ***un entête*** appelé ***prototype*** et ***un corps*** contenant les instructions à exécuter :

    [<ClasseAllocation>] [<Type>] <NomFonction> ([ListeParamètres])

    <CorpsFonction>

  - ***Prototype de fonction :***
    - Indique le type de données transmises et reçues par la fonction :
    - Chaque paramètre (***formel***) ou argument doit être fourni avec son type, qui peut être quelconque
  - ***Corps d'une fonction :***
    - ***Un bloc*** d'instructions. à l'intérieur duquel, on peut :
      - déclarer des variables externes
      - déclarer des fonctions
      - définir des variables locales au bloc
      - ***Mais il est interdit de définir des fonctions***.

Pointeurs & Fonctions

- ***Déclaration d'une fonction :***
  - Il faut déclarer une fonction avant de l'utiliser.
  - La déclaration informe le compilateur du type des paramètres et du résultat de la fonction
  - Si la fonction est définie avant son premier appel, alors pas besoin de la déclarer
  - Déclarer une fonction, c'est fournir son prototype
- ***Utilisation d'une fonction :***
  - se traduit par ***un appel à la fonction*** en indiquant son nom suivi de parenthèses renfermant éventuellement ***des paramètres effectifs***.
  - Les *paramètres formels* et *effectifs* doivent correspondre en nombre et en type (*les noms peuvent différer*).
  - L'appel d'une fonction peut être utilisé ***dans une expression*** ou ***comme une instruction***.

Pointeurs & Fonctions

- **Passage des paramètres d'une fonction**
  - A l'appel *d'une fonction avec paramètres*, la valeur ou l'adresse du paramètre effectif est transmise au paramètre formel correspondant.
    - Si *la valeur* est transmise, on a *un passage par valeur*.
    - Si *l'adresse* est transmise, on a *un passage par adresse* (ou par référence)
  - ***Passage par valeur :***
    - Si le nom d'une variable (***sauf le nom d'un tableau***) apparaît dans l'appel d'une fonction, comme paramètre effectif, alors la fonction appelée reçoit la valeur de cette variable.
    - Cette valeur *sera recopiée* dans le nom du paramètre formel correspondant.
    - Après l'appel de cette fonction, la valeur du paramètre effectif *n'est pas modifiée*
  - ***Passage par adresse :***
    - Lorsqu'on veut qu'une fonction *puisse modifier* la valeur d'une variable passée comme paramètre effectif, il faut transmettre l'adresse de cette variable.
    - La fonction appelée range l'adresse transmise dans une variable pointeur et la fonction *travaille directement* sur l'objet transmis.

Pointeurs & Fonctions

```
/* Exemple : Calcul de la moyenne d'un tableau de réels : */
#include <stdio.h>
#define max 50
void main() {
 int n ;
 float t[max] ;
 void lire_tab(float *, int *) ;
  /* 2ème façon : void lire2_tab(float [], int *) ; */
 float moyenne(float [], int) ;
 lire_tab(t, &n) ;
 printf("\n \n \t moyenne = %7.2f\n", moyenne(t, n)) ;
}
void lire_tab(float *ptab, int *pn) {
  /* 2ème façon : void lire2_tab(float tab[], int *pn) */
int i ;
 printf("Nombre de notes ? : ") ; scanf("%d", pn) ;
 for (i = 0 ; i<*pn ; i++)  {
    printf("Note n° %d : ", i + 1) ;
    scanf("%f", ptab++) ; /* 2ème façon: scanf("%d",&tab[i]) ; */
   }
}
float moyenne(float X[], int nb) {
 float s ;
 int i ;
 for (s = 0, i = 0 ; i<nb ; i++)
  s += X[i] ;
 return (s/nb) ;
}
```

Allocation de Mémoire

- ***Déclaration statique de données :***
  - Chaque variable dans un programme induit une réservation automatique d'un certain nombre d'octets en mémoire.
  - Le nombre d'octets à réserver *est connu pendant la compilation* : c'est la "***déclaration statique de données***".

Exemples :

```
float A, B, C ;   /* réservation de 12 octets */
short D[10][20] ; /* réservation de 200 octets */
double *G ;  /* réservation de p octets (p = taille d'un mot  machine. Dans notre cas, p = 2) */
```

- ***Allocation dynamique de la mémoire :***
  - La déclaration d'un tableau définit un tableau "***statique***" (*il possède un nombre figé d'emplacements*). Il y a donc un gaspillage d'espace mémoire en réservant toujours l'espace maximal prévisible.
  - Il serait souhaitable que l'allocation de la mémoire dépend du nombre d'éléments à saisir. Ce nombre ne sera *connu qu'à l'exécution* : c'est l' "***allocation dynamique***«

- **Fonctions d'allocation dynamique de la mémoire (malloc, calloc et realloc) :**
  - Chaque fonction prend une zone d'une taille donnée dans l'espace mémoire libre réservé pour le programme (appelé ***tas*** ou ***heap***) et affecte l'adresse du début de la zone à une variable pointeur.
  - S'il n'y a pas assez de mémoire libre à allouer, la fonction renvoie le pointeur **NULL**.

Allocation de Mémoire

- ***Fonction malloc***

```
<pointeur> = [<type>] malloc(<taille>) ;
```

<type> est un type pointeur définissant l'objet pointé par <pointeur>.
<taille> est le nombre d'octets alloués pour <pointeur>.

Exemple :

```
char *pc ;
pc = (char *) malloc(4000) ;
soit à pc est affectée l'adresse d'un bloc mémoire de 4000 octets. Soit pc contient la valeur 0 s'il n'y a pas assez de mémoire libre.
```

- ***Fonction calloc***

```
<pointeur> = [<type>] calloc(<nb_elts>, <taille_elt>) ;
```

  - S'il y a assez de mémoire libre, la fonction retourne un pointeur sur une zone mémoire de <nb_elts> éléments de <taille_elt> octets chacun initialisés à 0.

Exemple :

```
pt = (int *) calloc(100, sizeof(int)) ;   /* allocation dynamique d'un tableau de 100 entiers égaux à 0 */
```

- ***Fonction realloc***

```
<pointeur> = [<type>] realloc(<pointeur>, <nouvelletaille>);
```

  - Permet de modifier la taille d'une zone précédemment allouée par **malloc**, **calloc** ou **realloc**.
  - Si <pointeur> est **NULL**, alors **realloc** équivaut à **malloc**

- **Libération de la mémoire (la fonction free) :**
  - Un bloc de mémoire *réservé dynamiquement par* **malloc**, **calloc** ou **realloc**, peut être libéré à l'aide de la fonction **free**

```
free <pointeur> ;
```

Libère le bloc de mémoire désigné par le pointeur <pointeur>.

Exercice

Ecrire un programme C qui :

- ***demande à l'utilisateur de saisir tant qu'il le souhaite des nombres entiers au clavier***
- ***au fur est à mesure de la saisie, remplit, en utilisant l'allocation dynamique, un tableau initialement vide***
- ***effectue un tri par insertion des éléments du tableau, une fois la saisie des nombres est terminée***
- ***affiche les éléments du tableau.***

Types Structures, Unions et Synonymes

- ✓ **Types Structures**
- ✓ **Types Unions**
- ✓ **Types Enumérés**
- ✓ **Type Synonymes**
- ✓ **…**

Types structures (struct)

- **Une structure :**
  - est un nouveau *type de données composé de plusieurs **champs*** (ou ***membres***)
  - sert à représenter un objet réel.
- Chaque champ est de type quelconque pouvant être, lui aussi, une structure.
- Le nom d'une structure n'est pas un nom de variable (***c'est un nom de type***).
- **Exemple :**
  - Une date peut être représentée par les renseignements : jour, mois et année

Structures *(Exemples) (1)*

```
struct date
{
 int jour ;
 int mois ;
 int annee ;
} ;
struct adresse
{
 char nom[25], prenom[25] ;
 int n_rue ; /* numéro de rue */
 char rue[30] ;
 char ville[20] ;
} ;
struct date date_de_naissance ;
struct adresse adr1, adr2 ;
```

Structures *(Exemples) (2)*

```
struct complexe
{
 double re ;
 double im ;
} z1, z2 ;
/* z1 et z2 deux variables de type complexe */
```


Structure Anonyme

- ***Définition :***

```
struct
{
 <type1> <nom_champ1> ;
 <type2> <nom_champ2> ;
 …
 <typeN> <nom_champN> ;
} <liste_de_variables> ;
```

- ***Exemple :***

```
struct
{
 int heure ;
 int minute ;
 int seconde ;
} t1, t2 ;
```

Portée d'une Structure

- **Dépend de l'emplacement de sa déclaration :**
  - Au sein d'une fonction, elle n'est accessible que dans cette fonction
  - En dehors d'une fonction, elle est accessible de toute la partie du fichier source qui suit l'emplacement de la déclaration

Tableau de structures

<type_structure> <NomTableau> [<dimension>] ;

- **Exemple :**

```
struct client
{
 int compte ;
 char nom[20], prenom[20] ;
 float solde ;
} banque[1000] ; /* un tableau de 1000 clients au plus */
```

Imbrication/Pointeur

- **Imbrication de Structures :**

```
struct stage
{
 char nom[40] ;
 struct date debut, fin ;
} s, ts[10] ;
```

- **Pointeur de Structure :**

```
struct date *pd ;
```


Opérations sur les champs (1)

- ***Accès à un champ d'une structure :***

  <variable_structure>.<champ_structure>

- **Exemple :**

```
struct date d ;
d.jour = 2 ;  /* accès au champ jour de la date d */
scanf("%d", &d.jour) ;
printf("%d", d.jour) ;
```

Opérations sur les champs (2)

- ***Accès à un champ d'un pointeur de structure :***

  <pointeur_structure>-><champ_structure>

- **Exemple :**

```
struct date *pd, d ;
pd = &d ;
pd->jour = 5 ;   /* accès au champ jour */
```

- **Remarque :**
  - Il y a équivalence entre (*pd).jour et pd->jour

Opérations sur les variables structures (1)

- ***Initialisation à la déclaration :***

```
struct date d = {4, 10, 1999} ;
```

- ***Affectation :***
  - Les variables structures doivent être de même type (*à condition que des champs de la structure ne soient pas déclarés comme constantes*)
  - **Exemple :**

```
struct date d1, d2 = {4, 10, 1999} ;
d1 = d2 ;
```

Opérations sur les variables structures (2)

- ***Opérateur d'adresse & :***

```
struct date d, *pd ;
pd = &d ;
```

- ***Opérateur sizeof :***

```
printf("taille structure date : %d\n", sizeof(struct date)) ;
```

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 47

Structure Récursive

- Un ou plusieurs champs de la structure est ***un pointeur sur elle-même***.
- Permet de représenter des suites (*finies*) de taille quelconque avec ajouts et suppressions efficaces d'éléments.
- Ces structures requièrent généralement l'allocation dynamique pour allouer et libérer explicitement de la mémoire

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 48

Structure Récursive (Exemple)

- ***Liste chaînée de réels***

```
struct cellule
{
 double elt ;
 struct cellule *suiv ;
} ;
```

- Chaque cellule a deux champs, elt et suiv :
  - elt est un réel
  - suiv est un pointeur sur une structure cellule
- La valeur de suiv est soit l'adresse en mémoire d'une cellule soit le pointeur **NULL**.

Fonctions et structures (1)

- ***Retour d'une variable structure par une fonction :***
  - ***Exemple :***

```
struct date newdate()
{
 struct date d ;
 printf("Jour (1, 2, …, 31) : ") ; scanf("%d", &d.jour) ;
 printf("Mois (1, 2, …, 12) : ") ; scanf("%d", &d.mois) ;
 printf("Année (1900, …, 1999) : ") ; scanf("%d", &d.annee) ;
 return d ;
}
```

Fonctions et structures (2)

- ***Passage par valeur en argument d'une variable structure à une fonction :***
  - **Exemple :**

```
int chekdate(struct date) ;
```

- ***Passage par adresse en argument d'une variable structure à une fonction :***
  - **Exemple :**

```
void lire_date(struct date *pd)
{
 printf("Jour (1, 2, …, 31) : ") ; scanf("%d", &(*pd).jour) ;
 printf("Mois (1, 2, …, 12) : ") ; scanf("%d", &(*pd).mois) ;
 printf("Année (1900, …, 1999) : ") ; scanf("%d", &pd->annee) ;
}
```


Champs de bits

- ***Syntaxe de définition :***

```
struct <nom_structure>
{
 unsigned int <nom_champ1> : <nombre_de_bits> ;
 unsigned int <nom_champ2> : <nombre_de_bits> ;
 …
 unsigned int <nom_champN> : <nombre_de_bits> ;
} ;
```

Champs de bits (Exemple)

```
struct langue
{
 unsigned int anglais : 1 ;
 unsigned int allemand : 1 ;
 unsigned int espagnol : 1 ;
 unsigned int japonais : 1 ;
 unsigned int russe : 1 ;
} ;
```

```
struct employe1 ;
{
 …
 int anglais ;
 int allemand ;
 int espagnol ;
 int japonais ;
 int russe ;
 …
} ListeEmpl1[1000] ;
struct employe2 ;
{
 …
 struct langue L ;
 …
] ListeEmpl2[1000] ;
```


Type union (1)

- Les unions permettent l'utilisation d'***un même espace mémoire*** par des données de types différents à des moments différents :
- Une union ne contient qu'une donnée à la fois.
- Le système alloue un emplacement mémoire tel qu'il pourra contenir le champ de plus grande taille appartenant à l'union.
- ***Syntaxe de définition :***

```
union <nom_union>
{
 <type1> <nom_champ1> ;
 <type2> <nom_champ2> ;
 …
 <typeN> <nom_champN> ;
} ;
```

Type union (2)

- **Exemple :**

```
union zone
{
 int entier ;
 long entlong ;
 float flottant ;
 double flotlong ;
} z1, z2;
```


Type enum (1)

- Permettent d'exprimer ***des valeurs constantes*** de type entier en associant ces valeurs à des noms.
- ***Syntaxe de définition :***

```
enum <nom_énumération>
{
 <identificateur1> ;
 <identificateur2> ;
 …
 <identificateurN> ;
} ;
```

  - Les identificateurs sont considérés comme des constantes entières.
  - Le compilateur associe au 1er identificateur la constante 0, au 2ème la constante 1, … et au N^{ème} la constante N+1.

Type enum (2)

- **Exemples :**

```
enum couleurs
{rouge, vert, bleu} rvb ;
enum jour
{Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi,
Dimanche};
enum jour j1, j2, j ;
```


Type enum (3)

- ***Opérations sur les variables de type énuméré :***
  - ***Affectation :***

```
j1 = Lundi ;
j2 = j1 ;
```

  - ***Comparaison :***

```
if (j == Lundi) printf("Le jour est un Lundi\n") ;
```

  - ***Incrémentation, décrémentation :***

```
j2 = Dimanche ; j2-- ;
for (j = Lundi ; j<Samedi ; j++)
```

Types synonymes (typedef)

- **typedef** permet de définir *des types nouveaux synonymes* à des types existants.
- **typedef** ne réserve pas d'espace mémoire. Le nom est un type ;
- ***Syntaxe de définition :***

typedef <type> <nom_de_remplacement1>,
<nom_de_remplacement2>,
…
<nom_de_remplacementN> ;

Type synonyme d'un type simple

```
typedef int entier, boolean ;
typedef float reel ;

entier e1 = 23, te[50] = {1, 2, 3, 4, 5, 6, 7} ;
int i ;
i = e1 + te[20] ;
te[20] = i - 60 ;
```

Type synonyme d'un type tableau

```
typedef int tab[10] ;
tab tt; /* tt est un tableau de 10 entiers */

typedef float matrice[10][20] ;
matrice a ;
```


Type synonyme à une structure

```
typedef struct
        {
         int jour ;
         int mois ;
         int annee date ;
        } date ;
date d, *ptd ;
```

Type synonyme d'un pointeur

```
typedef char *chaine ;
chaine ch ;
```


Chaînes de caractères

- ✓ **Définition, déclaration et mémorisation**
- ✓ **Chaînes constantes**
- ✓ **Initialisation**
- ✓ **Ordre alphabétique et lexicographique**
- ✓ **Manipulation des chaînes de caractères**
- ✓ **Tableaux de chaînes de caractères**

Définition

Une chaîne de caractères est :

- une suite de caractères alphanumériques (*du texte*)
- représentée sur une suite d'octets se terminant par un octet supplémentaire lié au symbole '**\0**'. *Celui-ci indique* ***une fin de chaîne***.
- considérée comme un tableau de caractères qui peut être *manipulé d'une manière globale* (*sans le faire caractère par caractère*).

Déclaration & Mémorisation

Déclaration :

char <NomChaine> [<longueur>] ; /* sous forme de tableau */

ou

char *<NomChaine>; /* sous forme de pointeur */

Exemples :

char Nom[20] ; /* Nom est un tableau ne pouvant contenir au plus que 19 caractères utiles */

char *Prenom ;

Mémorisation :

- Le nom d'une chaîne de caractères est le représentant de l'adresse du 1er caractère de la chaîne.
- Pour mémoriser une chaîne de N caractères, on a besoin de N+1 octets.

Chaînes constantes ou littérales

- **Sont représentées entre guillemets. *La chaîne vide* est notée "".**
- **Pour une chaîne constante, le compilateur associe un pointeur constant.**
- **Dans une chaîne, *les* caractères de contrôle peuvent être utilisés.**
  - **Exemple :**
    "Ce \ntexte \nsera réparti sur 3 lignes.«
- **Le symbole " peut être représenté à l'intérieur d'une chaîne constante par \"**
  - **Exemple :**
    "Affichage de \"guillemets\" \n"
- **Plusieurs chaînes de caractères constantes séparées par des espaces, des tabulations ou interlignes, dans le texte d'un programme, seront réunies en une seule chaîne constante lors de la compilation.**
  - **Exemple :**
    "un" "deux"
    "trois"
    sera évaluée comme : "un deux trois"

Initialisation à la définition

Exemples :

```
char ch1[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'} ;
char ch2[] = "Bonjour" ;              /* initialisation particulière aux chaînes de caractères */
char ch3[8] = "Bonjour" ;
char ch4[7] = "Bonjour" ;             /* Erreur pendant l'exécution */
char ch5[6] = "Bonjour" ;             /* Erreur pendant la compilation */
char *ch6 = "Bonjour" ;               /* pointeur sur char */
```

Remarques :

1.
```
char *ch1 = "une chaîne" ;
char *ch2 = "une autre chaîne" ;
ch1 = ch2 ;          /* ch1 et ch2 pointent sur la même chaîne  "une autre chaîne" */
```

2.
```
char ch1[20] = "une chaîne" ;
char ch2[20] = "une autre chaîne" ;
char ch3[30] ;
ch1 = ch2 ;          /* Impossible → Erreur */
ch3 = "Bonjour" ;    /* Impossible → Erreur */
```

Ordre alphabétique et lexicographique

Ordre alphabétique des caractères :

- dépend du code utilisé pour les caractères.
- pour le code ***ASCII***, on constate l'ordre suivant : …, 0, 1, 2, …, 9, …, A, B, C, …, Z, …, a, b, c, …, z, …

Exemple :

'0' ***est inférieur à*** 'z' et ***noté :*** '0' **<** 'z' (*code **ASCII** ('0') = 48 et code **ASCII**('z') = 90*)

Ordre lexicographique des chaînes de caractères :

- basé sur l'ordre alphabétique des caractères.
- suit l'ordre du dictionnaire et est défini comme suit :
  1. La chaîne vide "" ***précède lexicographiquement*** toutes les autres chaînes
  2. La chaîne "$a_1a_2…a_p$" (*p caractères*) ***précède lexicographiquement*** la chaîne "$b_1b_2…b_m$" (*m caractères*) si l'une des deux conditions suivantes est remplie :
     - 'a_1' < 'b_1'
     - 'a_1' = 'b_1' et "$a_2…a_p$" ***précède lexicographiquement*** "$b_2…b_m$"

Exemples :

"ABC" ***précède*** "BCD" car 'A' < 'B'
"ABC" ***précède*** "B"
"Abc" ***précède*** "abc"
"ab" ***précède*** "abcd" car "" *précède* "cd"
" ab" ***précède*** "ab" car ' ' < 'a' (***ASCII***(' ') = 12 et ***ASCII***('a') = 97))

Manipulation des chaînes de caractères (*Fonctions de stdio.h*) (1)

Affichage de chaînes de caractères :

1. Fonction printf

int printf(**const char** *format [, argument, …])

A utiliser avec le spécificateur de format %s

Exemple :

```
char ch[] = "Bonjour tout le monde" ;
printf("%s", ch) ; /* affichage normal */
printf("%7s", ch) ; /* largeur minimale de 7 caractères */
printf("%.7s", ch) ; /* largeur maximale de 7 caract. */
printf("%25s", ch) ; /* alignement à droite sur 25 caract. */
printf("%-25s", ch) ; /* alignement à gauche sur 25 caract. */
```

2. Fonction puts

int puts(**const char** *ch) ;

Exemple :

```
char *ch = "Bonjour" ;
puts(ch) ; est équivalente à printf("%s\n", ch) ;
```

Manipulation des chaînes de caractères (*Fonctions de stdio.h*) (2)

Lecture de chaînes de caractères :

1. ***Fonction scanf***

int scanf(**const char** *format [, adresse, …]) ;

A utiliser avec le spécificateur de format %s

Exemple :

```
char lieu[25] ;
printf("Entrez le lieu de naissance : ") ; scanf("%s", lieu) ;
```

2. ***Fonction gets***

char ***gets**(**char** *ch) ;

Exemple :

```
char string[80] ;
printf("Entrez une chaîne de caractères : ") ; gets(string) ;
printf("La chaîne lue est : %s\n", string) ;
```

Remarque :

Contrairement à **scanf**, la fonction **gets** permet de saisir des chaînes de caractères contenant des espaces et des tabulations.

Manipulation des chaînes de caractères (*Fonctions de string.h*) (1)

Longueur d'une chaîne de caractères :

1. ***Fonction strlen***

int strlen(**const char** *s) ;

Retourne le nombre de caractères présents dans la chaîne s (*sans compter* '\0').

Concaténation de chaînes de caractères :

1. ***Fonction strcat***

char ***strcat**(**char** *s1, **const char** *s2) ;

Ajoute une copie de la chaîne s2 à la fin de la chaîne s1. Le caractère final '\0' de s1 est écrasé par le 1er caractère de s2.
Retourne un pointeur sur s1.

Exemple :

```
char *ch1 = "Bonjour" ;
char *ch2 = " tout le monde" ;
strcat(ch1, ch2) ;
printf("%s", ch1) ;
```

2. ***Fonction strncat***

char ***strncat**(**char** *s1, **const char** *s2, **int** n) ;

Ajoute au maximum les n premiers caractères de la chaîne s2 à la chaîne s1.

Exemple :

```
char ch1[20] = "Bonjour" ;
char *ch2 = " tout le monde" ;
strncat(ch1, ch2, 5) ;
```

Manipulation des chaînes de caractères (*Fonctions de string.h*) (2)

Comparaison de chaînes de caractères :

1. ***Fonction strcmp***

int strcmp(**const char** *s1, **const char** *s2) ;

Compare *lexicographiquement* les chaînes s1 et s2, et retourne une valeur :

= 0	si s1 et s2 *sont identiques*
< 0	si s1 *précède* s2
> 0	si s1 *suit* s2

Exemple :

```
if (!strcmp(ch1, ch2))     printf("identiques\n") ;
else
  if (strcmp(ch1, ch2)>0)      printf("%s précède %s\n", ch2, ch1) ;
  else     printf("%s suit %s\n", ch2, ch1) ;
```

2. ***Fonction strncmp***

int strncmp(**const char** *s1, **const char** *s2, **int** n) ;

Ici, la comparaison est effectuée sur les n premiers caractères.

3. ***Fonction stricmp***

int stricmp(**const char** *s1, **const char** *s2) ;

Travaille comme **strcmp** *sans faire la distinction* entre majuscules et minuscules.

4. ***Fonction strnicmp***

int strnicmp(**const char** *s1, **const char** *s2, **int** n) ;

Travaille comme **strcnmp** *sans distinguer les* majuscules des minuscules.

Manipulation des chaînes de caractères (*Fonctions de string.h*) (3)

Copie de chaîne de caractères :

1. ***Fonction strcpy***

char *strcpy(**char** *s1, **const char** *s2) ;

Copie la chaîne s2 dans s1 y compris le caractère '\0'.
Retourne un pointeur sur s1

2. ***Fonction strncpy***

char *strncpy(**char** *s1, **const char** *s2, **int** n) ;

Copie au plus les n premiers caractères de la chaîne s2 dans s1 et retourne un pointeur sur s1.
La chaîne s1 peut ne pas comporter le caractère terminal si la longueur de s2 vaut n ou plus.

Exemple :

```
char ch1[8] ;
char *ch2 = "bonjour" ;
strncpy(ch2, ch1, 3) ;
ch2[3] = '\0' ;
printf("%s\n", ch2) ;
```

Manipulation des chaînes de caractères (*Fonctions de string.h*) (4)

Recherche d'un caractère dans une chaîne de caractères :

1. ***Fonction strchr***

char ***strchr**(**const char** *s, **char** c) ;

Recherche la 1ère occurrence du caractère c dans la chaîne s.
Retourne un pointeur sur cette 1ère occurrence si c'est un caractère de s, sinon le pointeur **NULL** est retourné.

2. ***Fonction strrchr***

char ***strrchr**(**const char** *s, **char** c) ;

Identique à **strchr** sauf qu'elle recherche la dernière occurrence du caractère c dans la chaîne s.

Exemple :

```
char *ch = "Bonjour" ;
strchr(ch, 'o') ;
puts(strchr(ch, 'o')) ;
strrchr(ch, 'o') ;
puts(strrchr(ch, 'o')) ;
```


Manipulation des chaînes de caractères (*Fonctions de string.h*) (5)

Recherche d'une sous-chaîne de caractères dans une chaîne de caractères :

1. ***Fonction strstr***

char ***strstr**(**const char** *s1, **const char** *s2) ;

Recherche la 1ère occurrence de la chaîne s2 dans la chaîne s1.
Retourne un pointeur sur cette 1ère occurrence si la chaîne s2 est une sous-chaîne de la chaîne s1, sinon le pointeur **NULL** est retourné.

Exemple :

```
#include <string.h>
…
char *s1 = "Bonjour tout le monde" ;
char *s2 = "tout" ;
char *pch ;
pch = strstr(s1, s2) ;
printf("La sous-chaîne est : %s\n", pch) ;
```

Manipulation des chaînes de caractères (*Fonctions de string.h*) (6)

Recherche d'une sous-chaîne de caractères dans une chaîne de caractères :

2. Fonction strpbrk

char ***strpbrk**(**const char** *s1, **const char** *s2) ;

Recherche dans la chaîne s1 la 1ère occurrence d'un caractère quelconque de la chaîne s2

Exemple :

```
char *ch1 = "abcdefghij" ;
char *ch2 = "123f" ;
char *pch ;
pch = strpbrk(ch1, ch2) ;
if (pch)
  printf("strpbrk trouve le premier caractère %c\n", *pch) ;
else
  printf("strpbrk ne trouve pas de caractères\n") ;
```


Manipulation des chaînes de caractères (*Fonctions de string.h*) (7)

Recherche d'une sous-chaîne de caractères dans une chaîne de caractères :

3. Fonction strtok

char ***strtok**(**char** *s1, **const char** *scp) ;

Recherche dans la chaîne s des éléments (*des chaînes da caractères*) séparés par des séparateurs définis dans la chaîne de caractères constante scp.

Le 1er appel à **strtok** renvoie un pointeur sur le 1er caractère du 1er élément de la chaîne s et écrit le caractère '\0' dans la chaîne s immédiatement après l'élément renvoyé.

D'autres appels à **strtok**, avec **NULL** comme 1er argument, traitent de la même manière, et jusqu'à épuisement, les autres éléments de la chaîne s.

Remarque :

strtok permet d'éclater la chaîne s en différentes sous-chaînes obtenues en considérant comme séparateurs les différents caractères de la chaîne scp.

Exemples :

```
char ch[16] = "abc,d" ;
char *p ;
p = strtok(ch, ",") ; /* 1er appel à strtok */
if (p)
  printf("%s\n", p) ; /* il s'affichera la chaîne "abc" */
p = strtok(NULL, ",") ; /* 2ème appel à strtok avec comme 1er
                           argument NULL */
if (p)
  printf("%s\n", p) ; /* il s'affichera la chaîne "d" */
```

Manipulation des chaînes de caractères (Exercice)

- ***Ecrire une fonction qui affiche les mots d'une phrase :***
  - ***Une phrase est une chaîne de caractère constituée d'un ensemble de mots***
  - ***Les mots de la phrase sont séparés par un seul espace.***

Conversion nombres/chaînes de caractères (*Fonctions de stdlib.h) (1)*

Conversion d'une chaîne de caractères en une valeur numérique :

Fonctions atoi, atol, atof

int atoi(**const char** *s) ;
long atol(**const char** *s) ;
double atof(**const char** *s) ;

atoi (respectivement **atol**, **atof**) retourne la valeur numérique représentée par la chaîne s comme un **int** (respectivement **long int**, **double**).

Remarques :

Les espaces au début de la chaîne de caractères s sont ignorés.
La *conversion s'arrête* au 1er caractère non valide (c.-à-d. *non convertible*).
Si aucun caractère n'est valide, les fonctions retournent zéro.

Conversion nombres/chaînes de caractères (*Fonctions de stdlib.h) (2)*

Conversion d'une valeur numérique en une chaîne de caractères (non ANSI) :

Fonctions itoa, ltoa, ultoa

char ***itoa**(**int** n, **char** *s, **int** b) ;

char ***ltoa**(**long** n, **char** *s, **int** b) ;

char ***ultoa**(**unsigned long** n, **char** *s, **int** b) ;

Convertissent l'entier n, représenté en base de numération b, dans la chaîne s

Remarques :

Si n est un entier négatif et b = 10, **itoa** et **ltoa** (pas **ultoa**) utilisent le 1er caractère de la chaîne s pour le signe moins.

Si succès, les fonctions **itoa**, **ltoa** et **ultoa** renvoient un pointeur sur la chaîne résultante. Dans le cas contraire, elles retournent **NULL**.

Classification de caractères (*Fonctions de ctype.h)*

Fonctions de classification :

- retournent zéro si la condition respective n'est pas remplie.
- c est une valeur du type **int** qui peut être *représentée comme un caractère*

Fonction :	Retourne une valeur différente de zéro :
isupper(c)	si c est une lettre majuscule ('A', 'B', …, 'Z')
islower(c)	si c est une lettre minuscule ('a', 'b', …, 'z')
isdigit(c)	si c est un chiffre décimal ('0', '1', …, '9').
isalpha(c)	si islower(c) ou isupper(c).
isalnum(c)	si isalpha(c) ou isdigit(c).
isxdigit(c)	si c est un chiffre hexadécimal ('0', …, '9' ou 'A, 'B', …, 'F' ou 'a', 'b', …, 'f').
isspace(c)	si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f').

Conversion de caractères (*Fonctions de ctype.h)*

Fonctions de conversion :

– Retournent une valeur du type **int** qui peut être représentée comme caractère. La valeur originale de c est inchangée.

Fonction :	*Retourne :*
tolower(c)	la lettre minuscule si c est une majuscule.
toupper(c)	la lettre majuscule si c est une minuscule.

Tableaux de chaînes de caractères (1)

- Utiles pour mémoriser une suite de mots ou de phrases.

Exemples :

1. **char** Jour[7][9] = {"Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche"} ;

   Déclaration d'un tableau de 7 chaînes de caractères, chacune contenant au maximum 9 caractères (dont 8 significatifs).

2. Jour[4] = "Friday" ; /* affectation non valide ! */

   En effet Jour[4] représente l'adresse du 1er élément de la 4ème chaîne de caractères
   Pour faire ce type d'affectation, utiliser la fonction **strcpy** :

   strcpy(Jour[4], "Friday") ;

3. /* Affichage de la 1ère lettre des jours de la semaine */

   for (i = 0 ; i<7 ; i++)
   printf("%c\t", Jour[i][0]) ;

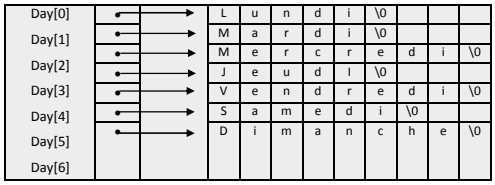
Tableaux de chaînes de caractères (2)

- Utiliser des *tableaux de pointeurs* pour *mémoriser de façon économique* des chaînes de caractères de différentes longueurs

Exemples :

```
char *Day[ ] = {"Lundi", "Mardi", "Mercredi", "Jeudi",
                "Vendredi", "Samedi", "Dimanche"} ;
```

Déclaration d'un tableau de 7 pointeurs sur **char**. Chacun des pointeurs est initialisé avec l'adresse de l'une des 7 chaînes de caractères constantes.



1. Day[4] = "Friday" ; /* Ici, affectation valable */
2. /* Affichage de la 1ère lettre des jours de la semaine */

```
for (i = 0 ; i<7 ; i++)
  printf("%c\t", *Day[i]) ;
```


Structures de Données Linéaires en C

- ✓ **Notion de Type Abstrait de Données**
- ✓ **Notion de Structure de Données**
- ✓ **Implémentation d'un TAD en C**
- ✓ **Exemples de Structures de Données Linéaires en C**

Notion de Type Abstrait de Données

- **Un type abstrait de données (TAD) :**
  - est un ensemble de valeurs muni d'opérations sur ces valeurs
  - sans faire référence à une implémentation particulière
- **Un TAD est caractérisé par :**
  - **sa signature :** définit la syntaxe du type et des opérations
  - **sa sémantique :** définit les propriétés des opérations

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 87

Notion de Structure de Données

- **On dit aussi :**
  - **structure de données concrète**
- **Correspond à :**
  - **l'implémentation d'un TAD**
- **Composée :**
  - d'un algorithme pour chaque opération
  - des données spécifiques à la structure pour sa gestion
- **Remarque :**
  - Un même TAD peut donner lieu à plusieurs structures de données, avec des performances différentes

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 88

Implémentation d'un TAD

- **Pour implémenter un TAD :**
  - Déclarer la structure de données retenue pour représenter le TAD : ***L'interface***
  - Définir les opérations primitives dans un langage particulier : ***La réalisation***
- **Exigences :**
  - Conforme à la spécification du TAD ;
  - Efficace en terme de complexité d'algorithme.
- **Pour implémenter, on utilise :**
  - Les types élémentaires (entiers, caractères, ...)
  - Les pointeurs ;
  - Les tableaux et les enregistrements ;
  - Les types prédéfinis.
- **Plusieurs implémentations possibles pour un même TAD**

Implémentation d'un TAD en C

- **Utiliser la programmation modulaire :**
  - Programme découpé en plusieurs fichiers, même de petites tailles (*réutilisabilité, lisibilité, etc.)*
  - Chaque composante logique (***un module***) regroupe les fonctions et types autour d'un même thème.
- **Pour chaque module truc, créer deux fichiers :**
  - **fichier truc.h : l'interface** (la partie publique) ; contient la spécification de la structure ;
  - **fichier truc.c : la définition** (la partie privée) ; contient la réalisation des opérations fournies par la structure. Il contient au début l'inclusion du fichier truc.h
- **Tout module ou programme principal qui a besoin d'utiliser les fonctions du module truc, devra juste inclure le truc.h**
- **Un module C implémente un TAD :**
  - **L'encapsulation :** détails d'implémentation cachés ; l'interface est la partie visible à un utilisateur
  - **La réutilisation :** placer les deux fichiers du module dans le répertoire où l'on développe l'application.

Structures de données linéaires

- **Structure linéaire :**
  - C'est un arrangement linéaire d'éléments liés par la **relation successeur**
- **Exemples :**
  - **Tableaux** (*la relation successeur est implicite*)
  - **Piles**
  - **Files**
  - **listes**

Notion de Pile (Stack)

- **Une pile est :**
  - une structure linéaire permettant de stocker et de restaurer des données selon **un ordre LIFO** (*Last In, First Out* ou « dernier entré, premier sorti »)
- **Dans une pile :**
  - Les insertions (***empilements***) et les suppressions (***dépilements***) sont restreintes à une extrémité appelée ***sommet*** de la pile.
- **Applications :**
  - Vérification du bon équilibrage d'une expression avec parenthèses
  - Evaluation des expressions arithmétiques postfixées
  - Gestion par le compilateur des appels de fonctions
  - …

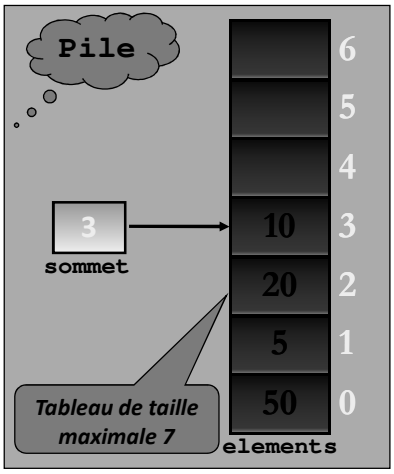
Type Abstrait Pile

```
Type Pile
Utilise Elément, Booléen
Opérations
    pile_vide  : → Pile
    est_vide   : Pile → Booléen
    empiler    : Pile x Elément → Pile
    dépiler    : Pile → Pile
    sommet     : Pile → Elément
Préconditions
    dépiler(p) est-défini-ssi est_vide(p) = faux
    sommet(p) est-défini-ssi est_vide(p) = faux
Axiomes
    Soit, e : Element, p : Pile
    est_vide(pile_vide) = vrai
    est_vide(empiler(p,e)) = faux
    dépiler(empiler(p,e)) = p
    sommet(empiler(p,e)) = e
```


Représentations d'une Pile

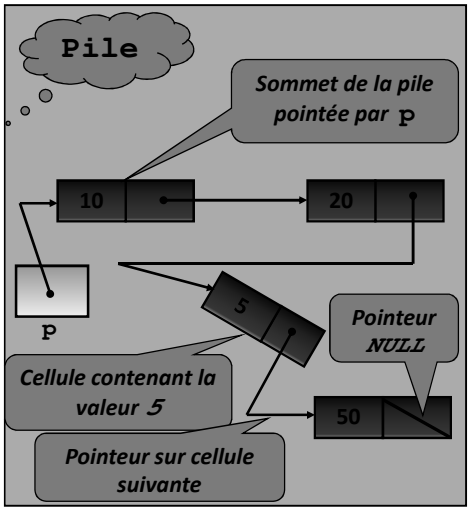
- **Représentation contiguë (*par tableau*) :**
  - ***Les éléments de la pile sont rangés dans un tableau***
  - ***Un entier représente la position du sommet de la pile***
- **Représentation chaînée (par pointeurs) :**
  - ***Les éléments de la pile sont chaînés entre eux***
  - ***Un pointeur sur le premier élément désigne la pile et représente le sommet de cette pile***
  - ***Une pile vide est représentée par le pointeur `NULL`***

Pile Contiguë en C




```
    /* Pile contiguë en C */

// taille maximale pile
#define MAX_PILE 7

// type des éléments
typedef int Element;

// type Pile
typedef struct {
 Element elements[MAX_PILE];
 int sommet;
} Pile;
```


Pile Chaînée en C




```
   /* Pile chaînée en C */

// type des éléments
typedef int element;

// type Cellule
typedef struct cellule {
  element valeur;
  struct cellule *suivant;
 } Cellule;

// type Pile
typedef Cellule *Pile;
```

Spécification d'une Pile Contiguë

```
                    /* fichier "Tpile.h" */
#ifndef _PILE_TABLEAU
#define _PILE_TABLEAU

#include "Booleen.h"

// Définition du type Pile (implémentée par un tableau)
#define MAX_PILE 7  /* taille maximale d'une pile */
typedef int Element;  /* les éléments sont des int */

typedef struct {
 Element elements[MAX_PILE]; /* les éléments de la pile */
 int sommet;  /* position du sommet */
} Pile;

// Déclaration des fonctions gérant la pile
Pile pile_vide ();
Pile empiler ( Pile p, Element e );
Pile depiler ( Pile p );
Element sommet ( Pile p );
Booleen est_vide ( Pile p );

#endif
```


Réalisation d'une Pile Contiguë

```
        /* fichier "Tpile.c" */

#include "Tpile.h"

// Définition des fonctions gérant la pile

// initialiser une nouvelle pile
Pile pile_vide() {
 Pile p;
 p.sommet = -1;
 return p;
}

// tester si la pile est vide
Booleen est_vide(Pile p) {
 if (p.sommet == -1) return vrai;
 return faux;
}

// Valeur du sommet de pile
Element sommet(Pile p) {
/* pré-condition : pile non vide ! */
 if (est_vide(p)) {
  printf("Erreur: pile vide !\n");
  exit(-1);
 }
 return (p.elements)[p.sommet];
}
```

```
// ajout d'un élément
Pile empiler(Pile p, Element e) {
 if (p.sommet >= MAX_PILE-1) {
  printf("Erreur : pile pleine !\n");
  exit(-1);
 }
 (p.sommet)++;
 (p.elements)[p.sommet] = e;
 return p;
}

// enlever un élément
Pile depiler(Pile p) {
/* pré-condition : pile non vide !*/
 if (est_vide(p)) {
  printf("Erreur: pile vide !\n");
  exit(-1);
 }
 p.sommet--;
 return p;
}
```

Utilisation d'une Pile Contiguë

```
                /* fichier "UTpile.c" */
#include <stdio.h>
#include "Tpile.h"

int main () {
 Pile p = pile_vide();

 p = empiler(p,50);
 p = empiler(p,5);
 p = empiler(p,20);
 p = empiler(p,10);
 printf("%d au sommet après empilement de 50, 5, 20 et"
        " 10\n", sommet(p));
 p = depiler(p);
 p = depiler(p);
 printf("%d au sommet après dépilement de 10 et 20\n",
        sommet(p));
 return 0;
}
```


Notion de File (Queue)

- **Une file est :**
  - une structure linéaire permettant de stocker et de restaurer des données selon un **ordre FIFO** (First In, First Out ou « premier entré, premier sorti »)
- **Dans une file :**
  - Les insertions (***enfilements***) se font à une extrémité appelée ***queue*** de la file et les suppressions (***défilements***) se font à l'autre extrémité appelée ***tête*** de la file
- **Applications :**
  - Gestion travaux d'impression d'une imprimante
  - Ordonnanceur (dans les systèmes d'exploitation)
  - …

Type Abstrait File

```
Type File
Utilise Elément, Booléen
Opérations
    file_vide   : → File
    est_vide    : File → Booléen
    enfiler     : File x Elément → File
    défiler     : File → File
    tête        : File → Elément
Préconditions
    défiler(f) est-défini-ssi est_vide(f) = faux
    tête(f) est-défini-ssi est_vide(f) = faux
Axiomes
    Soit, e : Element, f : File
    est_vide(pile_vide) = vrai
    est_vide(enfiler(f,e)) = faux
    si est_vide(f) = vrai alors tête(enfiler(f,e)) = e
    si est_vide(f) = faux alors tête(enfiler(f,e)) = tête(f)
    si est_vide(f) = vrai alors défiler(enfiler(f,e)) = file_vide
    si est_vide(f) = faux
      alors défiler(enfiler(f,e)) = enfiler(défiler(f),e)
```


Représentations d'une File

- **Représentation contiguë (*par tableau*) :**
  - ***Les éléments de la file sont rangés dans un tableau***
  - ***Deux entiers représentent respectivement les positions de la tête et de la queue de la file***
- **Représentation chaînée (par pointeurs) :**
  - ***Les éléments de la file sont chaînés entre eux***
  - ***Un pointeur sur le premier élément désigne la file et représente la tête de cette file***
  - ***Un pointeur sur le dernier élément représente la queue de file***
  - ***Une file vide est représentée par le pointeur NULL***

Notion de Liste

- **Généralisation des piles et des files**
  - Structure linéaire dans laquelle les éléments peuvent être traités les uns à la suite des autres
  - Ajout ou retrait d'éléments n'importe où dans la liste
  - Accès à n'importe quel élément
- **Une liste est :**
  - une suite finie, éventuellement vide, d'éléments de même type repérés par leur **rang** dans la liste
- **Dans une liste :**
  - Chaque élément de la liste est rangé à une certaine **place**
  - Les éléments d'une liste sont donc ordonnés en fonction de leur place
- **Remarques :**
  - Il existe une fonction notée **succ** qui, appliquée à toute place sauf la dernière, fournit la place suivante
  - Le nombre total d'éléments, et par conséquent de places, est appelé **longueur de la liste**
- **Applications :**
  - Codage des polynômes, des matrices creuses, des grands nombres, …

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 103

Type Abstrait Liste

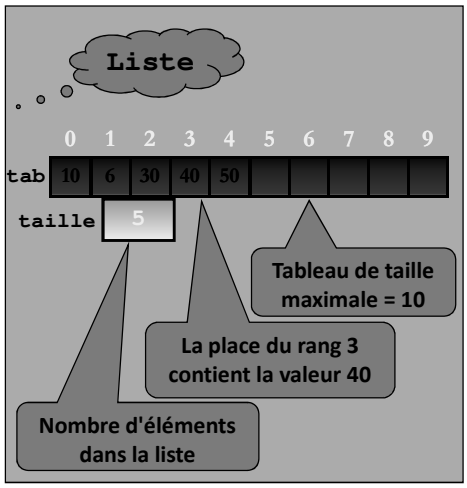
```
Type Liste
Utilise Elément, Booléen, Place
Opérations
    liste_vide : → Liste
    longueur   : Liste → Entier
    insérer    : Liste x Entier x Elément → Liste
    supprimer  : Liste x Entier → Liste
    kème       : Liste x Entier → Elément
    accès      : Liste x Entier → Place
    contenu    : Liste x Place → Elément
    succ       : Liste x Place → Place
Préconditions
    insérer(l,k,e) est-défini-ssi 1 ≤ k ≤ longueur(l)+1
    supprimer(l,k) est-défini-ssi 1 ≤ k ≤ longueur(l)
    kème(l,k) est-défini-ssi 1 ≤ k ≤ longueur(l)
    accès(l,k) est-défini-ssi 1 ≤ k ≤ longueur(l)
    succ(l,p) est-défini-ssi p ≠ accès(l,longueur(l))
```

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 104

Représentation contigüe d'une Liste

- **Les éléments sont rangés les uns à côté des autres dans un tableau**
  - ***La ième case du tableau contient le ième élément de la liste***
  - ***Le rang est donc égal à la place ; ce sont des entiers***
- **La liste est représentée par une structure en langage C :**
  - ***Un tableau représente les éléments***
  - ***Un entier représente le nombre d'éléments dans la liste***
  - ***La longueur maximale, MAX_LISTE, de la liste doit être connue***

Liste Contiguë en C

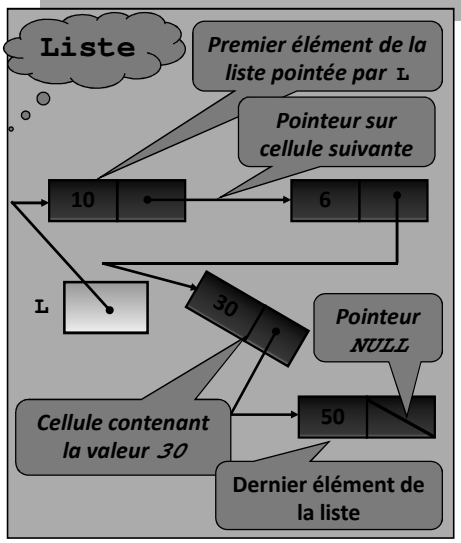



```
 /* Liste contiguë en C */
// taille maximale liste
#define MAX_LISTE 10

// type des éléments
typedef int Element;

// type Place
typedef int Place;

// type Liste
typedef struct {
 Element tab[MAX_LISTE];
 int taille;
} Liste;
```

Représentation chaînée d'une Liste

- **Les éléments ne sont pas rangés les uns à côté des autres**
  - ***La place d'un élément est l'adresse d'une structure qui contient l'élément ainsi que la place de l'élément suivant***
  - ***Utilisation de pointeurs pour chaîner entre eux les éléments successifs***
- **La liste est représentée par un pointeur sur une structure en langage C**
  - ***Une structure contient un élément de la liste et un pointeur sur l'élément suivant***
  - ***La liste est déterminée par un pointeur sur son premier élément***
  - ***La liste vide est représentée par la constante prédéfinie NULL***

Liste Chaînée en C




```
   /* Liste chaînée en C */

// type des éléments
typedef int element;

// type Place
typedef struct cellule* Place;

// type Cellule
typedef struct cellule {
  element valeur;
  struct cellule *suivant;
 } Cellule;

// type Liste
typedef Cellule *Liste;
```

Spécification d'une Liste Chaînée

```
                                /* fichier "CListe.h" */
#ifndef _LISTE_CHAINEE
#define _LISTE_CHAINEE

// Définition du type liste (implémentée par pointeurs)
typedef int element;  /* les éléments sont des int */

typedef struct cellule *Place;  /* la place = adresse cellule */

typedef struct cellule {
 element valeur;  // un éléments de la liste
 struct cellule *suivant;   // adresse cellule suivante
} Cellule;

typedef Cellule *Liste;

// Déclaration des fonctions gérant la liste
Liste liste_vide (void);
int longueur (Liste l);
Liste inserer (Liste l, int i, element e);
Liste supprimer (Liste l, int i);
element keme (Liste l, int k);

Place acces (Liste l, int i);
element contenu (Liste l, Place i);
Place succ (Liste l, Place i);

#endif
```

type Liste : un pointeur de Cellule

Réalisation d'une Liste Chaînée (1)

```
Liste liste_vide(void) {
 return NULL;
}

int longueur(Liste l) {
 int taille=0;
 Liste p=l;
 while (p) {
   taille++;
   p=p->suivant;
 }
 return taille;
}

Liste inserer(Liste l, int i, element e) {
// précondition :0 ≤ i < longueur(l)+1
 if (i<0 || i>longueur(l)) {
  printf("Erreur : rang non valide !\n");
  exit(-1);
 }
 Liste pc = (Liste)malloc(sizeof(Cellule));
 pc->valeur=e;
 pc->suivant=NULL;
 if (i==0){
   pc->suivant=l;
   l=pc;
 }
```

```
 else {
   int j;
   Liste p=l;
   for (j=0; j<i-1; j++)
     p=p->suivant;
   pc->suivant=p->suivant;
   p->suivant=p;
 }
 return l;
}

Place acces(Liste l, int k) {
// pas de sens que si 0 ≤ k ≤ longueur(l)-1
 int i;
 Place p;
 if (k<0 || k>=longueur(l)) {
   printf("Erreur: rang invalide !\n");
   exit(-1);
 }
 if (k == 0)
   return l;
 else {
   p=l;
   for(i=0; i<k; k++)
     p=p->suivant;
   return p;
 }
}
```

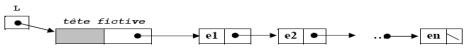
Réalisation d'une Liste Chaînée (2)

```
element contenu(Liste l, Place p) {
// pas de sens si longueur(l)=0 (liste vide)
 if (longueur(l) == 0) {
   printf("Erreur: liste vide !\n");
   exit(-1);
 }
 return p->valeur;
}

Place succ(Liste l, Place p) {
// pas de sens si p dernière place de liste
 if (p->suivant == NULL) {
   printf("Erreur: suivant dernière
    place!\n");
   exit(-1);
 }
 return p->suivant;
}

element keme(Liste l, int k) {
// pas de sens que si 0 <= k <= longueur(l)-1
 if (k<0 || k>longueur(l)-1) {
   printf("Erreur : rang non valide !\n");
   exit(-1);
 }
 return contenu(l, acces(l,k));
}
```

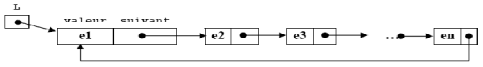
```
Liste supprimer(Liste l, int i) {
// précondition : 0 ≤ i < longueur(l)
 int j;
 Liste p;
 if (i<0 || i>longueur(l)+1) {
   printf("Erreur: rang non valide!\n");
   exit(-1);
 }
 if (i == 0) {
   p=l;
   l=l->suivant;
 }
 else {
   Place q;
   q=acces(l,i-1);
   p=succ(l,q);
   q->suivant=p->suivant;
 }
 free(p);
 return l;
}
```


Variantes de Listes Chaînées

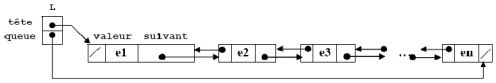
- ***Liste avec tête fictive***
  - Eviter d'avoir un traitement particulier pour le cas de la tête de liste (*opérations d'insertion et de suppression*)



- ***Liste chaînée circulaire***
  - Le suivant du dernier élément de la liste est le pointeur de tête



- ***Liste doublement chaînée***
  - Faciliter le parcours de la liste dans les deux sens (utilisation de deux pointeurs…)



- ***Liste doublement chaînée circulaire***
- ***Liste triée***
  - L'ordre des enregistrements dans la liste respecte l'ordre sur les clés

Fichiers

- ✓ **Définition et propriétés**
- ✓ **Fichiers de texte et fichiers binaires**
- ✓ **Fichiers standards**
- ✓ **Déclaration, ouverture et fermeture d'un fichier**
- ✓ **Traitement du contenu d'un fichier**
- ✓ **Déplacement dans le fichier (accès direct)**
- ✓ **…**

Définition et Propriétés

Définition :

- Un fichier est *une suite de données **homogènes conservées en permanence** sur un support externe* (*disque dur, clef USB, …).*
- Ces données regroupent, le plus souvent, plusieurs composantes (*champs*) d'une structure.

Exemples :

- Un fichier d'étudiants.
- Un fichier d'entiers.

Propriété :

- En ***C***, les fichiers sont considérés comme une suite d'octets (*1 octet = caractère*)

Manipulation d'un Fichier

Principe de manipulation d'un fichier :

1. ***ouverture du fichier***
2. ***lecture, écriture, et déplacement dans le fichier***
3. ***fermeture du fichier***

Manipulation d'un Fichier

Deux types de fonctions permettent de manipuler un fichier :

- ***fonctions de bas niveau :*** dépendent du système d'exploitation et font un accès direct sur le support physique de stockage du fichier.
- ***fonctions de haut niveau :*** l'accès au fichier se fait par l'intermédiaire d'une zone mémoire de stockage (***la mémoire tampon***). Ces fonctions sont construites à partir des fonctions de bas niveau.

Remarque :

- Dans ce cours, seules les fonctions de haut niveau seront étudiées et utilisées.

Manipulation d'un Fichier

Deux techniques pour manipuler un fichier :

- ***l'accès séquentiel :*** pour atteindre l'information souhaitée, il faut passer par la première puis la deuxième et ainsi de suite.
- ***l'accès direct :*** consiste à se déplacer directement sur l'information souhaitée sans avoir à parcourir celles qui la précèdent.

Mémoire Tampon

- Les accès à un fichier (*en vue d'une lecture ou écriture d'informations*) se font par l'intermédiaire *d'une* ***mémoire tampon*** (***buffer***).
- Il s'agit d'une zone de la mémoire centrale qui stocke une quantité, assez importante, de données du fichier.
- Son rôle est *d'**accélérer** les entrées/sorties à un fichier*.

Types de Fichiers

- **Deux types de fichiers :**
  - Fichiers de texte
  - Fichiers binaires
- ***Un fichier de texte*** est une suite de lignes ; chaque ligne est une suite de caractères terminée par le caractère spécial '\n'.
- ***Un fichier binaire*** est une suite d'octets pouvant représenter toutes sortes de données. (*le système n'attribue aucune signification aux octets échangés*)

Fichiers Standards

- Des fichiers spéciaux sont prédéfinis et ouverts automatiquement lorsqu'un programme commence à s'exécuter :
  - ***stdin :*** entrée standard (*par défaut, lié au clavier*)
  - ***stdout :*** sortie standard (*par défaut, lié à l'écran*)
  - ***stderr :*** sortie d'erreur standard (*par défaut, lié aussi à l'écran*)
- Ces fichiers peuvent être redirigés au niveau de l'interprète de commandes par l'utilisation de symboles **>** et **<** à l'appel du programme.

Fichiers Standards

Exemples :

1. Soit le fichier de texte "*c:\essai.txt*"

   Considérons les appels suivants du programme exécutable *Prog* :

   Prog > c:\essai.tx

   Prog écrira dans c:\essai.txt au lieu de l'écran

   Prog < c:\essai.txt

   Prog fera ses lectures dans *c:\essai.tx)*

2. Soient *Prog1* et *Prog2* deux programmes exécutables.

   Soit l'appel suivant :

   Prog1 | Prog2

   Prog1 a sa sortie standard redirigée dans l'entrée standard de *Prog2*

Fichiers Standards

Aux fichiers standards :

sont associées des fonctions prédéfinies permettant de réaliser les opérations suivantes :

- ***lecture et écriture caractère par caractère***
- ***lecture et écriture ligne par ligne***
- ***lecture et écriture formatées***

Fichiers Standards

Echanges caractère par caractère :

Fonction getchar

int getchar() ;

Permet de lire un caractère sur *stdin*.

Retourne la valeur du caractère lu ou **EOF** (*si fin du fichier ou erreur*)

Exemple :

```
while ((c = getchar() != EOF) && (c != ' ')) ;
/* lit jusqu'au premier caractère non espace ou EOF */
```

Fonction putchar

int putchar(**int** c) ;

Permet d'écrire le caractère c sur *stdout*.

Retourne la valeur du caractère écrit c ou **EOF** en cas d'erreur

Fichiers Standards

Echanges ligne par ligne :

Une ligne est considérée comme une suite de caractères terminée par le caractère fin de ligne '\n' ou par la détection de la fin du fichier.

Fonction gets

char *gets(**char** *s) ;

Lit une ligne sur *stdin* et la place dans la chaîne s. Le caractère fin de ligne '\n' est remplacé dans s par le caractère fin de chaîne '\0'.

Retourne **NULL** à la rencontre de la fin de fichier ou en cas d'erreur

Fonction puts

int puts(**char** *s) ;

Permet d'écrire la chaîne de caractères s, suivie d'un saut de ligne sur *stdout*.

Retourne le dernier caractère écrit ou **EOF** en cas d'erreur.

Fichiers Standards

Echanges avec formats :

Fonction scanf

int scanf(char *, …) ;

Effectue une lecture formatée sur *stdin*.

Fonction printf

int printf(char *, …) ;

Effectue une écriture formatée sur *stdout*.

Déclaration d'un Fichier

FILE *<PointeurFichier> ;

- Le type **FILE** est défini dans <*stdio.h*> en tant que structure.
- A l'ouverture d'un fichier, la structure **FILE** contient un certain nombre d'informations sur ce fichier telles que :
  - adresse de la mémoire tampon,
  - position actuelle dans le tampon,
  - nombre de caractères déjà écrits dans le tampon, …,
  - type d'ouverture du fichier : écriture, lecture, …,
  - …
- Pour pouvoir travailler avec un fichier dans un programme, ranger l'adresse de la structure **FILE** dans le pointeur de fichier et tout accès ultérieur au fichier se fait par l'intermédiaire de ce pointeur.

Ouverture d'un Fichier

- Association d'un objet extérieur (*le fichier*) au programme en cours d'exécution.
- Réalisée par la fonction **fopen** selon la syntaxe :

  FILE ***fopen**(**char** *<NomFichier>, **char** *<TypeOuverture>) ;

Exemple :

```
pf = fopen("essai.dat", "rb") ;
```

- **fopen** tente d'ouvrir le fichier désigné par <NomFichier> pour le type d'ouverture spécifié <TypeOuverture>.
- Si succès, crée une structure de type **FILE**, y stocke les informations relatives à ce fichier et retourne l'adresse de cette structure.
- Sinon, **NULL** est retourné.
- Le type d'ouverture indique la nature des opérations que le programme devra exécuter après l'ouverture du fichier.

Types d'Ouverture d'un Fichier

Les différents types d'ouverture d'un fichier :

- **"r"** : ouverture en lecture seule. Si fichier inexistant, la fonction retourne **NULL**.
- **"w"** : création pour écriture. Si fichier préexistant, il est vidé (*son contenu est perdu*)
- **"a"** : ouverture pour ajout ; ouverture en écriture en fin de fichier ou création pour écriture si fichier inexistant.
- **"r+"** : ouverture de fichier préexistant pour mise à jour (*lecture/écriture*).
- **"w+"** : création pour mise à jour (*lecture/écriture*). Si fichier préexistant, le contenu est perdu.
- **"a+"** : ouverture pour ajout ; ouverture pour mise à jour en fin de fichier ou création si fichier inexistant.

Remarques :

- Pour indiquer qu'un fichier doit être ouvert ou créé en mode texte, ajouter **t** à la chaîne ("rt", "wt", "at", "rt+" ou "r+t", "wt+" ou "w+t", "at+" ou "a+t").
- Pour le mode binaire, ajouter **b** ("rb", "wb", …).

Fermeture d'un Fichier

- Termine la manipulation d'un fichier ouvert en faisant appel à la fonction **fclose** selon la syntaxe :

  int fclose(FILE *<PointeurFichier>) ;

Exemple :

fclose(pf) ; /* pf est un pointeur de fichier */

- **fclose** est la fonction inverse de **fopen** ; elle détruit le lien entre le pointeur de fichier et le nom du fichier.
- Retourne :
  - 0 dans le cas normal.
  - **EOF** en cas d'erreur.

Remarques :

- Quand un fichier ne sert plus, il est conseillé de le fermer.
- Dés qu'un fichier est fermé, la mémoire tampon est libérée.
- Après **fclose**(pf), le pointeur pf est ***invalide***. Des erreurs graves pourraient donc survenir si ce pointeur est utilisé par la suite.

Ouverture et Fermeture d'un fichier (Exemple)

```
#include <stdio.h>
#include<string.h>
void main()
{
 char nomfich[20] ; /* nom physique du fichier à traiter */
 FILE *pf ; /* pf est un pointeur de fichier */
 printf("Nom de sauvegarde du fichier : ") ;
 gets(nomfich) ;
 pf = fopen(nomfich, "r") ; /* Ouvre en lecture le fichier */
 if (pf == NULL)
   printf("Impossible d'ouvrir le fichier\n") ;
 else
   … /* traitement du fichier */
 fclose(pf) ; /* fermer le fichier référencé par pf */
}
```

Traitement du Contenu d'un Fichier

- Une fois le fichier ouvert, ***C*** permet plusieurs types de traitement du fichier :
  - ***par caractères***
  - ***par lignes***
  - ***par enregistrements***
  - ***par données formatées***
- Dans tous les cas, les fonctions de traitement du fichier (***sauf*** *les opérations de déplacement (voir plus loin)*) ont un *comportement séquentiel*. L'appel de ces fonctions provoque le déplacement du pointeur courant relatif au fichier ouvert.

Traitement Caractère par Caractère

Fonction fgetc

int fgetc(**FILE** *<PointeurFichier>) ;

- Lit un caractère dans le fichier référencé par le pointeur <PointeurFichier>
- Retourne :
  - Le caractère lu sous forme d'un **int**
  - **EOF** à la rencontre de la fin du fichier ou en cas d'erreur.

Fonction getc

int *getc*(**FILE** *<PointeurFichier>) ;

- Identique à **fgetc**() **sauf que** cette fonction est réalisée par ***une macro*** définie dans <*stdio.h*>.
- Pour une macro, les instructions sont générées en ligne (*et répétées à chaque appel*) ce qui évite un appel de fonction (*coûteux*).

Traitement Caractère par Caractère

Fonction fputc

int fputc(**int** <Caractere>, **FILE** *<PointeurFichier>) ;

- Ecrit dans le fichier référencé par le pointeur <PointeurFichier> le caractère placé dans la variable <Caractere>.
- Retourne :
  - La valeur sous forme d'**int** du caractère écrit dans le fichier.
  - **EOF** en cas d'erreur.

Fonction putc

int putc(**int** <Caractere>, **FILE** *<PointeurFichier>) ;

Identique à **fputc**() **sauf que** cette fonction est réalisée par une ***macro***

Traitement Caractère par Caractère (Exemple 1)

Création d'un fichier texte :

```
main() {
 FILE *pf ;
 char *nomf ; /* nom physique du fichier */
 int c ; /* le caractère à traiter */

 printf("Nom de sauvegarde : ") ; gets(nomf) ;
 if ((pf = fopen(nomf,"w")) != NULL) {
   printf("Entrez votre texte et terminez par CTRL-Z \n") ;
   while ((c = getchar() != EOF))
     fputc(c,pf) ;
   fclose(pf) ;
  }
 else printf("Problème d'ouverture") ;
 return 0 ;
}
```

Traitement Caractère par Caractère (Exemple 2)

Lecture d'un fichier texte :

```
main() {
 FILE *pf ;
 char *nomf ;
 int c ; /* caractère à traiter */
 printf("Nom du fichier à lire : ") ; gets(nomf) ;
 if ((pf = fopen(nomf,"r")) != NULL)  {
    while ((c = fgetc() != EOF))
      putchar(c) ;
    if (!feof(pf))   /* feof est une fonction qui détecte la fin d'un fichier (voir plus loin */
     printf("Erreur de lecture") ;
    fclose(pf) ;
   }
 else printf("Problème d'ouverture") ;
 return 0 ;
}
```


Traitement Caractère par Caractère

Remarques :

- c = **getchar**() équivalente à c = **getc**(**stdin**) ou c = ***fgetc***(**stdin**)
- **putchar**(c) équivalente à **putc**(c, **stdout**) ou **fputc**(c, **stdout**)

Traitement par Lignes (Lecture de Chaînes)

Fonction fgets

char ***fgets**(**char** *<Chaine>,**int**<Nbre>,**FILE** *<PointeurFichier>);

- Lit une ligne de caractères dans le fichier référencé par <PointeurFichier>. Cette ligne est stockée dans <Chaine>.
  <Nbre> est le nombre maximum de caractères à lire.
- Retourne :
  - Un pointeur vers le début de la chaîne.
  - **NULL** en cas d'erreur ou à la rencontre de la fin de fichier.
- La lecture s'arrête lorsque un des événements se produit :
  - Lecture de saut de ligne '\n' ('*n' est recopié dans <Chaine>*)
  - Lecture d'au plus (<Nbre> - 1) caractères (***fgets** termine <Chaine> par '\0'*)
  - Rencontre de la fin de fichier

Traitement par Lignes (Ecriture de Chaînes)

Fonction fputs

int fputs(**char** *<Chaine>, **FILE** *<PointeurFichier>) ;

- Ecrit la chaîne <Chaine> dans le fichier référencé par <PointeurFichier>.
- Retourne :
  - Une valeur positive (*code ASCII du dernier caractère écrit*) si l'écriture s'est correctement déroulée.
  - **EOF** en cas d'erreur.
- La chaîne <Chaine> doit être terminée par '\0'. Ce caractère n'est pas transféré dans le fichier. Il faut mettre explicitement la fin de ligne dans la chaîne pour qu'elle soit présente dans le fichier.

Traitement par Lignes (Exemple)

Lecture ligne à ligne d'un fichier après ouverture :

```
void main() {
 char nomfich[20] ; /* fichier à traiter */
 FILE *pf ;
 char BigBuf[256] ; /* pour stocker une ligne de caractères */
 printf("Nom de sauvegarde du fichier : ") ;
 gets(nomfich) ;
 pf = fopen(nomfich, "r") ;
 if (pf == NULL)  {
   printf("Impossible d'ouvrir le fichier %s \n", nomfich) ;
   return 1 ;
  }
 while (fgets(BigBuf, sizeof BigBuf, pf) != NULL)
  fputs(BigBuf, stdout) ; /* écrire la ligne lue à partir du
        fichier référencé par pf sur la sortie standard */
 fclose(pf) ; /* fermer le fichier référencé par pf */
}
```


Traitement par Enregistrements

– Permet de lire et écrire des objets,
  - le plus souvent représentés par *des structures (appelées **enregistrements**)* dans un fichier.

– *Pour ce type de traitement,*
  - ***le fichier doit être ouvert en mode binaire***.
  - Les données échangées ne sont pas traitées comme des caractères. Elles sont traitées sous forme de ***blocs d'octets***.

Traitement par Enregistrements

Fonction fread (*lecture d'un bloc d'octets d'un fichier*)

unsigned int fread(**void** *<pb>, **unsigned** <taille>,
unsigned <nb>, **FILE** *<pf>) ;

- Lit un certain nombre de données (*des enregistrements*) de taille identique depuis un fichier référencé par <pf> vers un bloc mémoire.
  - Le bloc mémoire d'adresse <pb> *reçoit* les enregistrements lus.
  - **<taille>** : taille d'un enregistrement en nombre octets.
  - **<nb>** : nombre d'enregistrements à échanger (*lire*).
  - **<pf>** : fait référence à un fichier ouvert en *mode binaire*.
  - Le nombre d'octets lus est (<nb> * <taille>)
- Retourne :
  - Le nombre d'enregistrements lus (*et non le nombre d'octets*).
  - Si **EOF** ou erreur, une valeur inférieure à <nb> (*ou même 0*).

Traitement par Enregistrements

Fonction fwrite (*écriture d'un bloc d'octets dans un fichier*)

unsigned int fwrite(**void** *<pb>, **unsigned** <taille>,
unsigned <nb>, **FILE** *<pf>) ;

- L'espace mémoire d'adresse <pb> *fournit* les données à écrire dans les enregistrements.
- Ecrit **<nb>** éléments (*enregistrements*) ayant chacun une taille de **<taille>** octets à la fin d'un fichier référencé par **<pf>**.

  Le nombre d'octets écrits est **(<nb> * <taille>)**
- Retourne :
  - Le nombre d'enregistrement écrits (*et non le nombre d'octets*).
  - Si erreur, une valeur inférieure à **<nb>** (*ou même **0***).

Traitement par Enregistrements (Exemple)

Lecture d'enregistrements dans un fichier :

- **Cet exemple :**
  - est une lecture du contenu d'un fichier appelé *FichParcAuto*
  - avec stockage du contenu de ce fichier dans un tableau en mémoire *ParcAuto*.

- **Les cases du tableau sont des structures contenant :**
  - un entier,
  - une chaîne de 20 caractères et
  - 3 chaînes de 10 caractères.

Traitement par Enregistrements (Exemple)

```
struct automobile {
  int age ;
  char couleur[20], numero[10], type[10], marque[10] ;
} ParcAuto[20] ;
main() {
 FILE *pf ; /* pointeur de fichier */
 int i ;
 unsigned fait ;
 pf = fopen("FicParcAuto","rb+") ;  /* Remarquer le type d'ouverture du fichier */
 if (pf == NULL) {
   printf("Can't open FicParcAuto\n") ;    return 1 ;
  }
 for (i=0 ; i<20 ; i++) {
   fait = fread(&ParcAuto[i], sizeof(struct automobile),1,pf) ;
   if (fait != 1) {
     printf("Erreur lecture fichier ParcAuto\n") ;     return 2 ;
    }
  }
 fclose(pf) ;
}
```

Traitement par Enregistrements (Exemple)

Remarque :

Il est possible de demander la lecture de 20 enregistrements en une seule opération, en remplaçant la boucle **for** par :

```
fait = fread(ParcAuto, sizeof(struct automobile), 20, pf) ;
```

ou bien par :

```
fait = fread(ParcAuto, sizeof ParcAuto, 1, pf) ;
```


Lecture et Ecriture Formatées dans les Fichiers

- **Sont utilisées les deux fonctions fprintf** et **fscanf**
- permettent de réaliser le même travail que **printf** et **scanf** sur ***des fichiers ouverts en mode texte*** :

Ecriture Formatée dans les Fichiers

Fonction fprintf *(écriture formatée sur un fichier ouvert en mode texte)*

int fprintf(**FILE** *<PointeurFichier>, **char** *<Format>, <Arguments>);

- Ecrit les données formatées dans un fichier.
- Fonctionne ainsi :
  - Accepte une série d'arguments (*les valeurs des données à écrire*).
  - Applique à chaque argument un spécificateur de format dans <Format>.
  - Envoie les données formatées dans un fichier.
- Retourne :
  - Le nombre de caractères écrits
  - Une valeur négative en cas d'erreur.

Ecriture Formatée dans les Fichiers

Remarques :

- Le nombre d'arguments doit satisfaire le nombre de formateurs :
  - S'il y a trop d'arguments (*pas assez de formateurs*), ceux en trop sont ignorés.
- En pratique,
  - les arguments représentent les rubriques qui forment un enregistrement et dont les valeurs respectives sont écrites dans le fichier.

Ecriture Formatée dans les Fichiers (Exemple)

```
#include <stdio.h>
int main(void) {
 FILE *pf ;
 int i = 100 ;
 char c = 'C' ;
 float f = 1.234 ;
 pf = fopen("Essai.txt", "w+") ; /* ouverture mise à jour */
 fprintf(pf, "%d %c %f", i, c, f) ;
 fclose(pf) ; /* fermer le fichier */
 return 0 ;
}
```


Lecture Formatée dans les Fichiers

Fonction fscanf *(lecture formatée dans un fichier ouvert en mode texte)*

int fscanf(**FILE** *<PointeurFichier>, **char** *<Format>, <Adresses>);

- Lit des données formatées dans un fichier :
  - <PointeurFichier> fait référence au fichier.
  - <Format> : format de lecture des données.
  - <Adresses> : adresses des variables à affecter à partir des données.
  - Un formateur et une adresse doivent être fournis pour chaque variable.
- Retourne :
  - Le nombre d'éléments lus (*0 si aucun élément n'a été traité totalement*)
  - **EOF** si fin de fichier.

Lecture Formatée dans les Fichiers (Exemple)

```
#include <stdlib.h>
#include <stdio.h>
int main(void)  {
 int i ;
 printf("Introduisez un entier : ") ;
 /* lire un entier à partir de l'entrée standard */
 if (fscanf(stdin, "%d", &i))
   printf("L'entier lu est : %i\n", i) ;
 else  {
    fprintf(stderr, "Erreur en lisant un entier sur stdin\n") ;
    exit(1) ;
   }
 return 0 ;
}
```


Lecture et Ecriture Formatées dans les Fichiers

Remarques :

- **fprintf**(**stdout**, "Bonjour\n")

  équivalente à **printf**("Bonjour\n")

  Dans les fichiers texte, il faut ajouter le symbole de fin de ligne '\n' pour séparer les données.

- **fscanf**(**stdin**, "%d", &N)

  équivalente à **scanf**("%d", &N)

  A l'aide de **fscanf**, il est impossible de lire toute une phrase dans laquelle les mots sont séparés par des espaces.

Détection de la Fin de Fichier

Function ***feof***

int feof(**FILE** *<PointeurFichier>) ;

– Consulte un "***indicateur de fin de fichier***" sur lequel agissent les différentes fonctions de manipulation de fichier.

– Retourne :

- Une valeur égale à 0 si la fin de fichier (**EOF**) n'a pas été détectée.
- Une valeur différente de 0 sinon.

Déplacement dans le Fichier (*Accès Direct*)

– Les fonctions précédentes modifient de manière automatique le pointeur courant dans le fichier correspondant (*adresse de l'octet dans le fichier à partir duquel se fera la prochaine opération de lecture ou écriture*).

– Après chaque opération de lecture ou d'écriture, ce pointeur de position (*défini dans* ***FILE***) est incrémenté du nombre de blocs transférés pour indiquer la prochaine opération de *lecture ou écriture* : C'est l'***accès séquentiel***.

– Les fonctions suivantes permettent de connaître la valeur de la position courante dans le fichier et de la modifier. Cela permettra de réaliser des lectures ou des écritures en n'importe quel endroit du fichier : C'est l'***accès direct***.

Déplacement dans le Fichier (*Accès Direct*) (*Fonctions associées à la position dans un fichier)*

Fonction fseek

int fseek(**FILE** *<PointeurFichier>, **long** <Offset>, **int** <Base>) ;

- Change la position courante dans le fichier référencé par <PointeurFichier> (*permet de placer le pointeur de position sur un octet quelconque du fichier*).
- <Offset> : déplacement à l'intérieur du fichier en nombre d'octets.
  - Si <Offset> est positif, le déplacement se fait vers la fin du fichier.
  - Si <Offset> est négatif, le déplacement se fait vers le début du fichier.
- <Base> : point de départ du déplacement. <Base> peut prendre les valeurs suivantes définies dans <*stdio.h*> :
  - 0 (ou **SEEK_SET**) : déplacement relatif au début du fichier.
  - 1 (ou **SEEK_CUR**) : déplacement relatif à la position courante.
  - 2 (ou **SEEK_END**) : déplacement relatif à la fin du fichier.
- Retourne :
  - 0 en cas de succès.
  - Une valeur différente de 0 si le déplacement ne peut être réalisé

Déplacement dans le Fichier (*Accès Direct*) (*Fonctions associées à la position dans un fichier)*

Remarques :

- Le 1er octet du fichier (*octet de rang 1*) est à la position 0.
- L'instruction :

  fseek(pf, 1L * **sizeof**(enrg)*(n-1), **SEEK_SET**) ;

  fait placer le pointeur de position sur le nème enregistrement enrg du fichier référencé par pf.
- Utiliser **fseek** avec précaution pour un *fichier texte*.

Déplacement dans le Fichier (*Accès Direct*)
(*Fonctions associées à la position dans un fichier)*

Utilité de fseek : *modification d'un enregistrement du fichier connaissant sa position dans le fichier*.

Exemple : ***Modifier le n^{ème} enregistrement***

```
fseek(pf, 1L * sizeof(enrg)*(n-1), 0) ;
fread(&enrg, sizeof(enrg),1 ,pf) ;
/* Instructions pour Modifier l'enregistrement enrg */
…
fseek(pf,1L * sizeof(enrg)*(n-1), 0) ;
fwrite(&enrg, sizeof(enrg),1 ,pf) ;
```


Déplacement dans le Fichier (*Accès Direct*)
(*Fonctions associées à la position dans un fichier)*

Fonction ftell

long ftell(FILE *<PointeurFichier>) ;

- Détermine la valeur de la position courante dans le fichier référencé par <PointeurFichier>.
- Retourne :
  - Sur *les fichiers binaires* : nombre d'octets entre la position courante et le début du fichier.
  - Sur *les fichiers texte* : une valeur permettant à **fseek** de repositionner le pointeur courant à l'endroit actuel.
  - -1L en cas d'erreur.

Déplacement dans le Fichier (*Accès Direct*)
(*Fonctions associées à la position dans un fichier)*

Remarque :

Pour connaître la taille (*le nombre d'octets*) d'un fichier, il suffit de faire :

```
long taille, nbre_enrg ;
…
fseek(pf, 0L, SEEK_END) ;
taille = ftell(pf) ;
nbre_enrg = taille / (sizeof(enrg)) ;
```


Déplacement dans le Fichier (*Accès Direct*)
(*Fonctions associées à la position dans un fichier)*

Fonction rewind

void rewind(**FILE** *<PointeurFichier>) ;

Permet de se placer en début de fichier.

rewind(pf) ;
est équivalente à **fseek**(pf, 0L, **SEEK_SET**) ;

Déplacement dans le Fichier (*Accès Direct*) (Exemple)

Modification de l'âge des voitures dans le fichier FicParcAuto :

Le programme correspondant procède de la manière suivante :

- Lit un enregistrement du fichier dans une zone mémoire
- Modifie la zone en mémoire
- Replace le pointeur courant sur le début de l'enregistrement pour pouvoir réécrire cet enregistrement
- Ecrit la zone mémoire dans le fichier.

Déplacement dans le Fichier (*Accès Direct*) (Exemple)

```
#include <stdio.h>
struct automobile  {
 int age ;
 char couleur[20], numero[10], type[10], marque[10] ;
} UneAuto ;
main() {
 FILE *pf ;
 int i ;
 unsigned fait ;
 pf = fopen("FicParcAuto","r+b") ;
 if (pf == NULL) {
   printf("Can't open FicParcAuto\n") ;    return 1 ;
  }
 for (i = 0 ; i<20 ; i++)  {
/* lecture d'un enregistrement du fichier dans la zone
mémoire   (variable) UneAuto du type struct automobile */
   fait = fread(&UneAuto, sizeof UneAuto, 1, pf) ;
   if (fait != 1)  {
     printf("Erreur lecture fichier ParcAuto\n") ;     return 2 ;
    }
  UneAuto.age++ ; /* modifier la valeur du champ age dans la
structure en mémoire */
```

```
/* Modifier la position courante du fichier pour
positionner   le pointeur courant à l'adresse de début de
l'enregistrement  qui est en mémoire */

  fait = fseek(pf, -1L * sizeof UneAuto, SEEK_CUR) ;
  if (fait != 0)  {
    printf("Erreur déplacement fichier ParcAuto\n") ;
    return 3 ;
   }

 /* Ecrire dans le fichier le contenu de la zone mémoire
   UneAuto. Cette écriture provoque la modification de
   l'enregistrement sur disque */
  fait = fwrite(&UneAuto, sizeof UneAuto, 1, pf) ;
  if (fait != 1)  {
    printf("Erreur écriture fichier ParcAuto fait=%d\n",
fait);     return 4 ;
   }
  }
 fclose(pf) ;
}
```

Gestion des erreurs

- Les erreurs des fonctions d'entrées/sorties *peuvent être récupérées par le programme*. Pour donner plus d'informations sur les causes d'erreur, les fonctions d'entrées/sorties utilisent *une variable globale* de type entier appelée **errno**.
- Par exemple, si un fichier n'a pas pu être ouvert avec succès, (*résultat **NULL***), un code d'erreur est placé dans la variable **errno**. Ce code désigne plus exactement la nature de l'erreur. Les codes d'erreurs sont définis dans <*errno.h*>.
- L'appel de la fonction **strerror**(**errno**) retourne un pointeur sur la chaîne de caractères qui décrit l'erreur dans **errno**.
- L'appel de la fonction **perror**(s) affiche la chaîne s suivie du signe deux-points (:), puis le message d'erreur qui est défini pour l'erreur dans **errno**, et enfin un caractère de saut de ligne.

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 163

Gestion des erreurs (Exemples)

```
#include <stdio.h>
#include <errno.h>
main()  {
 char *buffer ;
 buffer = strerror(errno) ;
 printf("Error : %s\n", buffer) ;
 return 0 ;
}
```

```
#include <stdio.h>
main()  {
 FILE *pf ;
 pf = fopen("Test.dat", "r") ;
 if (!pf)
   perror("Impossible d'ouvrir le fichier
en lecture") ;
 return 0 ;
}
```

[SMI4-fsr] Programmation II (M21-S4) 2014-2015 164

Vider le Tampon associé à un Fichier

Fonction fflush

int fflush(**FILE** *<PointeurFichier>) ;

– Force l'écriture (*physique*) sur disque des données en attente dans le tampon (*buffer*) associé au fichier référencé par le pointeur <PointeurFichier>.

– Retourne :

- 0 dans le cas normal.
- **EOF** en cas d'erreur (*si l'écriture physique s'est mal passée*)

Vider le Tampon associé à un Fichier (Exemple)

```
int a, int b, float c ;
char d ;
fflush(stdin) ; /* pour vider le buffer d'entrée standard */
d = getchar() ;
printf("%d %d %f %c\n",a,b,c,d) ;
fflush(stdout) ; /* pour vider le buffer de sortie standard et
        donc forcer l'affichage du contenu de ce buffer */
```

Suppression Physique d'un Fichier

Fonction fflush

int remove (**const char** *<NomFichier>)

- Supprime le fichier <NomFichier> sur disque.
- S'assurer que le fichier à supprimer a été fermé.
- Retourne :
  - 0 en cas de succès
  - -1 si erreur

Suppression Physique d'un Fichier (Exemple)

```
 fich char[80] ;
printf("Fichier à supprimer ? : ") ; gets(fich) ;
if (remove(fich) == 0)
  printf("Fichier \"%s\" supprimé\n", fich) ;
else
  perror("remove") ;
```

Compléments

- ✓ **Arguments de la fonction main**
- ✓ **Fonctions sprintf et sscanf**
- ✓ **Préprocesseur**
- ✓ **Compilation séparée**
- ✓ **…**

Arguments de la Fonction main

La fonction main() peut recevoir un certain nombre d'arguments :

- Ceux-ci doivent être transmis dans la ligne de commande (*la ligne destinée à appeler le programme exécutable*).
- La ligne de commande est considérée comme un tableau de chaînes de caractères
- deux identificateurs prédéfinis sont destinés à récupérer ces arguments : **argc** et **argv**

La définition de la fonction main() est alors :

```
main(int argc, char *argv[ ])
{
  /* code de la fonction main */
}
```

ou d'une manière équivalente :

```
main(int argc, char **argv)
{
  /* code de la fonction main */
}
```

Arguments de la Fonction main

Signification :

- **argc** : nombre d'arguments transmis dans la ligne de commande. Le nom du programme exécutable lui-même est pris en compte dans cette valeur *(le nombre d'arguments est-il toujours au moins égal à 1)*
- **argv** : pointeur sur les différentes chaînes de caractères passées dans la ligne de commande. Le premier argument, **argv**[0], contient le nom du programme.

Les arguments transmis dans la ligne de commande sont séparés par un espace

Arguments de la Fonction main (Exemple)

Soit la ligne de commandes suivante :

C:\>COPIE DE FICHA VERS FICHB

COPIE désigne le nom du programme exécutable
DE, *FICHA*, *VERS* et *FICHB* désignent les arguments de la commande

Supposons que le code source du programme *COPIE* est le suivant :

```
#include <stdio.h>
main(int argc, char *argv[])
{
 int i ;
 for (i = 1 ; i<argc ; i++) /* affichage des arguments */
   printf(%s \n", argv[i]) ;
}
```

Arguments de la Fonction main (Exemple)

argv[0] contient l'adresse du 1er caractère du nom du programme ("*COPIE*")

argv[1] celle du 1er argument ("*DE*")

argv[2] celle du 2ème argument ("*FICHA*")

argv[**argc**-1] celle du dernier argument ("*FICHB*")

argv[**argc**] contient **NULL**

Fonctions sprintf et sscanf

- Plusieurs fonctions font *des conversions de format* telles que :
  - **scanf** et **printf**.
  - **fprintf** et **fscanf** travaillant sur des fichiers ouverts en *mode texte*.
- Deux nouvelles fonctions appelées :
  - **sprintf** et **sscanf**

  s'utilisent pour faire de ***la conversion de données*** (ou ***formatage de données***) ***en mémoire***.

Fonction sprintf

Prototype :

int sprintf(**char** *string, **const char** *format, …) ;

- Permet de faire une conversion de données vers une zone mémoire (*string*) par transformation en chaîne de caractères.
- Possède trois arguments :
  - Zone dans laquelle les caractères sont *stockés* ;
  - Format d'écriture des données ;
  - Valeurs de données.

  sprintf convertit les arguments (*les valeurs de données*) suivant le format de contrôle et met le résultat dans la chaîne string.
- Retourne :
  - nombre de caractères stockés.
  - valeur négative en cas d'erreur.

Fonction sprintf

Exemple :

```
#include <stdio.h>
char s1[81], s2[81] ;
char *ch ;
int i, code ;
i = 15 ;
code = sprintf(s1, "%d", i) ;
code = sprintf(s2, "i vaut %d et sa moitié %f", i, i/2.0) ;
code = sprintf(ch, "%d", i) ;
/* Erreur ! on passe à sprintf un pointeur ch non initialisé */
/* Solution ! initialiser ch soit par un tableau de caractères
suffisamment grand, soit par appel à une fonction d'allocation
dynamique comme calloc */
```

Fonction sscanf

Prototype :

int sscanf(**char** *string, **const char** *format, …) ;

- Permet de faire une lecture formatée de données d'une zone mémoire.
- Possède trois arguments :
  - Zone dans laquelle les caractères sont *acquis* ;
  - Format de lecture des données ;
  - Adresse des variables à affecter à partir des données.

  sscanf extrait d'une chaîne de caractères (*string*) des valeurs qui sont stockées dans des variables suivant le format de contrôle.
- Retourne :
  - nombre de variables saisies.
  - **EOF** en cas d'erreur empêchant toute lecture.

Fonction sscanf

Exemple :

```
#include<stdio.h>
char *s ,
int code ;
double a, b, c ;
s = "12.5   12.3   11.6" ;
code = sscanf(s, "%f %f %f", &a, &b, &c) ;
                /* sscanf va lire la chaîne s pour
                  affecter les 3 valeurs a, b, c */
```

Fonctions sprintf et sscanf

Remarques :

- **scanf** *est une source permanente de problèmes* !

  Pour remédier à ceci, on fait recours à l'utilisation de **gets** et/ou de **sscanf**.

- **sprintf** et **sscanf** sont très utilisées pour convertir des numériques en chaîne de caractères et inversement.

Fonctions sprintf et sscanf (Exemple pratique)

```
#include <stdio.h>
#include <stdlib.h> /* contient le prototype de la fonction  random. random(n), avec n entier, permet de
        générer un nombre aléatoire a tel que 0 <= a < n */
char *noms[4] = {"Nom1", "Nom2", "Nom3", "Nom4"} ;
#define NBRE 4

void main(void) {
 int i ;
 char temp[4][80] ;
 char nom[20] ;
 int age ;
 long salaire ;
/* créer les données nom, âge et salaire */
 for (i = 0 ; i < NBRE ; ++i)
  sprintf(temp[i], "%s %d %ld", noms[i], random(10) + 20, random(5000) + 27500L) ;
 /* afficher une barre de titres */
 printf("%4s | %-20s | %5s | %15s\n", "#", "Nom", "Age",  "Salaire") ;
 printf(" ----------------------------------------------\n");
 /* lire et afficher les données nom, âge et salaire */
 for (i = 0 ; i < NBRE ; ++i) {
   sscanf(temp[i],"%s %d %ld", &nom, &age, &salaire) ;
   printf("%4d | %-20s | %5d | %15ld\n", i + 1, nom, age,  salaire) ;
  }
}
```

Préprocesseur

- Effectue *un prétraitement*, lors de la compilation d'un programme ***C***, :
  - en supprimant dans un premier temps tous les commentaires,
  - puis en traitant des "***directives de compilation***".
  - Enfin, envoie le programme ***C*** modifié au compilateur.
- Les directives de compilation, dans un programme ***C***, commencent toutes par un caractère ***#*** et sont de trois types :
  - ***directive d'inclusion de fichiers*** ;
  - ***directives de compilation conditionnelle*** ;
  - ***directives de substitution symbolique***. Ce type permet :
    - la définition de constantes ;
    - la définition de ***macros*** (***substitution avec arguments***)

Directive d'Inclusion de Fichier

#include

Exemples :

#include <fichier.h>

réalise l'inclusion du fichier *fichier.h* contenu dans un répertoire spécial (*connu par le préprocesseur*)

#include "fichier.h"

réalise l'inclusion du fichier *fichier.h* contenu dans le répertoire de travail ou, à défaut, dans le répertoire spécial. Il est également possible d'indiquer un chemin précis pour la recherche du fichier, soit par exemple, **#include** "c:\dev-cpp\include\fichier.h«

L'extension *.h* d'un fichier est l'abréviation de "*header*" (*entête*).

Directives de Compilation Conditionnelle

- **Rôle :**

  Incorporer ou exclure de la compilation des portions de texte de programme selon que l'évaluation de la condition donne 0 ou 1.

- **Plusieurs directives :**

Directive	Rôle
#ifdef	inclusion si symbole défini
#if defined	même chose
#ifndef	inclusion si symbole non défini
#if	inclusion si condition vérifiée
#else	sinon
#elif	else if, c.-à-d. sinon si
#endif	fin de si
#undef	met fin à l'existence d'un symbole

Directives de Compilation Conditionnelle (Exemple 1)

```
#define SYS1 1 /* définir symbole SYS1 et l'initialiser à 1 */
main() {
 #if defined (SYS1)
 … /* décl. Ou instr. C */ /* Ces lignes seront incluses dans la compilation */
 #endif
 #if defined (SYS1)
 … /* décl. Ou instr. C */ /* idem que précédemment */
 #else
 … /* décl. Ou instr. C */ /* Ces lignes auraient été incluses dans la compilation si SYS1 n'avait pas été défini */
 #endif
 #if !defined (SYS1)
 … /* décl. Ou instr. C */  /* idem que précédemment */
 #endif
 #if SYS1 == 1
 … /* décl. Ou instr. C */  /* Ces lignes seront incluses dans la compilation puisque SYS1 vaut 1*/
 #endif
 #if defined (SYS1) && defined (SYS2)
 … /* décl. Ou instr. C */  /* Ces lignes seraient été incluses dans la compilation si SYS1 et SYS2 seraient définis */
 #endif
 #if (sizeof(int) == 2)
 … /* décl. Ou instr. C */  /* lignes incluses si int est codé   sur 16 bits */
 #endif
 …
}
```

Directives de Compilation Conditionnelle (Exemple 2)

```
#define SYS1  1
#define SYS2..3
main() {
 #ifdef SYS1
 int i, j ;
 #else
 float i, j ;
 #endif
 j = i*2 ;
 #ifdef SYS1
 i = 5 ;
 #else
 i = 5.5 ;
 #endif
 #undef SYS2
 #if defined (SYS2)
 j = 6 ;
 #endif
}
```

Après traitement des directives par le préprocesseur, le texte résultat de ce programme est ainsi :

```
main() {
 int i, j ;
 j = i*2 ;
 i = 5 ;
}
```


Directives de Compilation Conditionnelle

Remarques :

- Les comparaisons effectuées par le préprocesseur ne peuvent porter que sur des constantes entières (*et pas sur des variables du programme dont l'évaluation n'est possible qu'à l'exécution...*)
- Les opérateurs interprétables par le préprocesseur sont : *!*, ~ (*complément à 1*), *-*, *+*, ***, */*, *|* (***ou*** *binaire inclusif*), *%*, ^ (***ou*** *binaire exclusif*), *&* (***et*** *binaire*), *<<*, , *<*, , *<=*, *=*, *!=*, *&&*, *||*, *==*, et l'opérateur ternaire conditionnel *?:*

Directives de Compilation Conditionnelle (Exemple 3)

```
#define _DEBUG_
...
#ifdef _DEBUG_
printf("Ligne 1234 : x = %d -- y = %d -- z = %d\n",x,y,z) ;
#endif
...
#ifdef _DEBUG_
printf("Ligne 2345 : x = %d -- y = %d -- z = %d\n",x,y,z) ;
#endif
```

- Les directives de compilation conditionnelles sont particulièrement utiles pour la "***mise au point***" d'un programme ***C*** :
  - Dans l'exemple ci-dessus :
    - La "***trace***" (*l'affichage de la valeur des variables dans les principales étapes d'un programme*) n'est effective que si le symbole _DEBUG_ est défini.
    - La suppression de la définition de ce symbole dans le programme produira la disparition de cet affichage après la prochaine compilation.

Directives de substitution symbolique (Définition de constantes)

#define *<symbole> <équivalent>*
#undef

- Le préprocesseur remplace dans un programme ***C*** (*en dehors des lignes commençant par un caractère* #) toutes les occurrences du symbole *<symbole>* par son équivalent *<équivalent> ,* en réitérant le processus si besoin est, sauf si cela engendre une infinité de remplacements.

- Le domaine de visibilité de la substitution d'un symbole s'étend entre la directive **#define** de ce symbole et la directive **#define** suivante de ce même symbole ou la directive **#undef** de ce même symbole ou, à défaut, jusqu'à la fin du programme.

Directives de substitution symbolique (Définition de constantes)

- **Le symbole** <symbole> est formé de lettres, de chiffres et du caractère _ et doit impérativement commencer par une lettre.
- **L'équivalent** <équivalent> doit être tapé sur une seule ligne. Si l'écriture de l'équivalent nécessite plusieurs lignes, il faut faire précéder la frappe de chaque caractère retour-chariot par un caractère ****

Directives de substitution symbolique (Définition de constantes)

Exemples :

```
#define PI 3.14159
#define FAUX 0
#define VRAI 1
...
#undef PI
```

Définition de macros (ou macro-instructions)

- Il existe une forme paramétrée pour la substitution symbolique :

#define <**symbole(param1, param2,...)**> <**équivalent**>

- Les paramètres qui suivent une occurrence de *<symbole>,* dans le programme, sont identifiés par le préprocesseur à *param1, param2,* …etc.
- L'équivalent *<équivalent>* est envoyé au compilateur par le préprocesseur, avec la même substitution des paramètres. On appelle cela une "***macro-instruction***" (ou "***macro***").
- La parenthèse ouvrante avant la liste des paramètres doit suivre immédiatement le symbole (*il ne doit pas y avoir d'espace*).

Définition de macros (ou macro-instructions) (Exemple 1)

#define ABS(n) ((n0) ? n : -n)

La séquence suivante :

```
L1 : main() {
L2 : int m,n = -8 ;
L3 : m = ABS(n) ;
L4 : printf("%d",m) ;
L5 : }
```

deviendra, après le passage du préprocesseur :

```
main() {
 int m, n = -8 ;
 m = ((n0) ? n : -n) ;
 printf("%d",m) ;
}
```

A l'exécution, la valeur *8* s'affichera à l'écran.

Définition de macros (ou macro-instructions)

Remarque :

– L'emploi des macro-instructions doit faire l'objet d'une attention particulière. Pour éviter de nombreux problèmes (*dus aux priorités des opérateurs*) :

- il est conseillé de parenthèser les paramètres de la macro-instruction
- Il faut, de plus, éviter de rendre le programme incompréhensible par l'abus de **#define**.

Définition de macros (ou macro-instructions) (Exemple 2)

```
#define SOMME(X,Y) X+Y
main() {
 int i, j, k ;
 float a, b, c ;
 …
 k = SOMME(i,j) ;
 c = SOMME(a,b) ;
 k = SOMME(5*i,b-a) ;
}
```

devient :

```
main() {
 int i, j, k ;
 float a, b, c ;
 …
 k = i+j ;
 c = a+b ;
 k = 5*i+b-a ;
}
```

Il est à noter que pour ce programme,

#define SOMME(X,Y) ((X)+(Y))

aurait été meilleur car à l'abri de toute erreur en cas d'utilisation à l'intérieur d'une expression arithmétique ou en cas d'utilisation avec des arguments qui sont eux-mêmes des expressions.

Utilité d'une macro-instruction

– permet d'optimiser le code compilé, en limitant le nombre d'appels à une fonction dans le programme exécutable.

– permet d'effectuer des actions sur des variables dont le type n'est pas connu *a priori* :
  • (*dans l'exemple 1 donné ci-dessus, la macro ABS peut calculer la valeur absolue d'un entier ou d'un réel*).

Macro-instruction sans paramètre

– Il est possible de définir *une macro-instruction sans paramètre*, comme dans l'exemple suivant, effectivement présent dans le fichier *stdio.h* :

```
#define getchar() getc(stdin)
```

Opérateurs et symboles prédéfinis

Opérateur #

– Dans une macro, il permet de substituer un paramètre par sa valeur convertie en chaîne de caractères.

Exemple :

```
#define chaine1(c) #c
#define chaine2(c) "c"
printf("%s\n",chaine1(Module 4)) ;
printf("%s\n",chaine2(Licence)) ;
```

produit l'affichage à l'écran de :

```
Module 4
c
```


Opérateurs et symboles prédéfinis

Opérateur ##

– Cet opérateur effectue la concaténation de deux symboles.

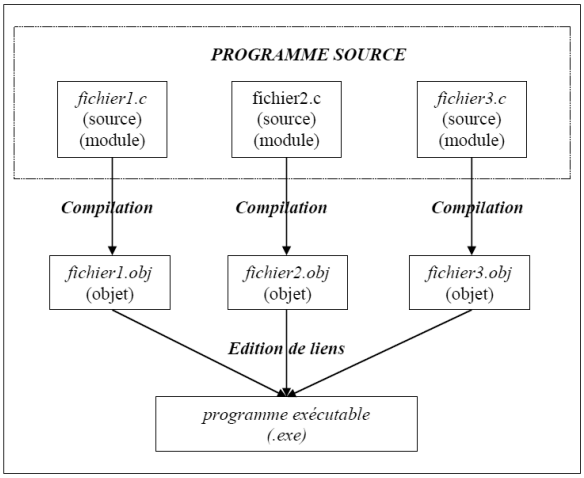
Exemple :

```
#define f(a,b) a##b
f(Module,4) ; /* est remplacé par Module4 */
```

Compilation séparée

- Permet de fragmenter un grand programme en *des parties qui peuvent être compilées indépendamment les unes des autres*.
- En ***C***, un programme source peut être décomposé en un ensemble de fichiers de texte (*aussi appelés source*) :
  - Ces fichiers pourront être compilés séparément et finalement reliés par *l'éditeur de liens* pour en faire un programme exécutable.
- Il sera ainsi possible :
  - d'apporter des modifications à un fichier *sans devoir recompiler l'ensemble*
  - de *créer des librairies de fonctions* (*sous forme de fichiers d'extension* ***.lib***) sans avoir à mettre le texte de ces fonctions à la disposition des utilisateurs.

Compilation séparée



Compilation séparée

- Chaque fichier source contient les éléments suivants dans un ordre quelconque :
  - déclarations de variables et de fonctions externes,
  - définitions de types synonymes ou de modèles de structures,
  - définitions de variables globales (*des demandes de réservation mémoire destinées à l'éditeur de liens*),
  - définitions de fonctions,
  - directives de précompilation et des commentaires (*les deux sont traités par le préprocesseur*).
- Le compilateur ne voit que les quatre premiers types d'objets.
- Les fichiers inclus par le préprocesseur ne doivent contenir que des déclarations externes ou des définitions de types et de modèles de structures.

Compilation séparée (Exemple)

module 1 (pr1.c)

```
void func(void) ;          /* fonction prototype */
int nb ;  /* provoque allocation de mémoire (2 octets) */
main()
{
 nb = 2 ;
 func() ; /* défini dans un module compilé séparément */
}
```

module 2 (pr2.c)

```
extern int nb ;   /* pas d'allocation de mémoire */
void func(void)
{
 printf("nb : % \n", nb) ;
}
```

Les deux modules sont compilés séparément. La compilation de pr1.c crée pr1.obj et celle de pr2.c crée pr2.obj. L'éditeur de liens génère le module exécutable.