

Programmation C Avancée

Ce module de cours offre un rapide rafraichissement des notions requises pour programmer en langage C dans un environnement de programmation, et permet ensuite de découvrir des notions plus avancées du langage C.

Site: Espadon
Cours: PROGRAMMATION C AVANCÉE
Livre: Programmation C Avancée
Imprimé par: Thomas Stegen
Date: mercredi 23 mars 2016, 15:42



Table des matières

- 1 Consignes et Programme du cours
- 2 Rappels
 - 2.1 Création d'un projet en C sur Dev C++
 - 2.2 Analyse des paramètres d'entrée
 - 2.3 Pour se remettre dans le bain
- 3 Rappels sur la compilation
- 4 Debugger avec Dev C++
- 5 Types de Données - Structures
 - 5.1 Types de données
 - 5.2 Structures de données
 - 5.3 Définitions de types
 - 5.4 Exercice : Structures
- 6 Les Tableaux
 - 6.1 Les tableaux à une dimension
 - 6.2 Les tableaux à deux dimensions
 - 6.3 Exercice : Tableaux à deux dimensions
- 7 Fonctions avancées
 - 7.1 Blocs et portée
 - 7.2 Déclaration et définition des fonctions
 - 7.3 Renvoyer un résultat
 - 7.4 Paramètres des fonctions
- 8 Pointeurs
 - 8.1 Les pointeurs
 - 8.2 Exercices
- 9 Pointeurs et tableaux
 - 9.1 Adressage des composantes d'un tableau
 - 9.2 Arithmétique des pointeurs
 - 9.3 Pointeurs et chaînes de caractères
 - 9.4 Pointeurs et tableaux à deux dimensions
 - 9.5 Exercice : Pointeurs et Tableaux
 - 9.6 Exercice : Formalismes
 - 9.7 Exercice : Adressage d'un tableau
 - 9.8 Exercice : Elimine Occurence
 - 9.9 Exercice : Inverse Tableau
 - 9.10 Exercice : Tableaux à 2 dimensions
- 10 Pointeurs et chaînes de caractères
 - 10.1 Pointeurs sur char et Chaînes de Caractères Constantes
 - 10.2 Avantages des Pointeurs sur char
 - 10.3 Tableaux de pointeurs
 - 10.4 Exercice : Palindrome
 - 10.5 Exercice : Longue chaîne
 - 10.6 Exercice : Nombre de mots
 - 10.7 Exercice : Compter les lettres
 - 10.8 Exercice : Chaînes de caractères constantes
- 11 Allocation dynamique de mémoire
 - 11.1 Déclaration statique de donnée
 - 11.2 Allocation dynamique
 - 11.3 La fonction malloc et l'opérateur sizeof
 - 11.4 La fonction Free
 - 11.5 Exercice : Inverse phrases
 - 11.6 Exercice : Effacer Phrases
 - 11.7 Exercice : Concaténer phrases
- 12 Fichiers séquentiels
 - 12.1 Définitions et Propriétés
 - 12.2 La mémoire Tampon
 - 12.3 Accès aux fichiers séquentiels
 - 12.4 Le type FILE*
 - 12.5 Exemple: Créer et Afficher un Fichier Séquentiel
 - 12.6 Ouvrir et fermer des fichiers séquentiels
 - 12.7 Lire et écrire dans des fichiers séquentiels
 - 12.8 Traitement par enregistrements
 - 12.9 Traitement par caractères

12.10 Détection de la Fin d'un Fichier Séquentiel

12.11 Résumé

13 Mise à jour d'un fichier séquentiel en C

14 Listes chaînées

14.1 Ajouter un enregistrement à un fichier

14.2 Supprimer un enregistrement dans un fichier

14.3 Modifier un enregistrement dans un fichier

14.4 Exercice : Notes d'un étudiant

14.5 Exercice : Notes de plusieurs étudiants

15 Fichiers Avancés

15.1 Les Fichiers sur Disque

15.2 Les Messages d'Erreurs

15.3 Ouverture d'un fichier

15.4 Exemple : Copier un fichier

15.5 Les Erreurs d'Entrée/Sortie

15.6 Positionnement dans un Fichier

15.7 Le Traitement par Blocs

15.8 Opérations sur les fichiers

15.9 Les fichiers temporaires

15.10 Rediriger un flux d'E/S

15.11 Exercice

15.12 Exercice : Fichiers Avancés

16 Projets Contrôle Continu

1 Consignes et Programme du cours

Consignes

Au fil du cours de Programmation C avancée, créez pour chaque chapitre un rapport en Microsoft Word, dans lequel vous mettrez des copies d'écrans de vos actions, des copies de vos codes sources et les réponses aux questions qui sont posées.

Prérequis

Les éléments suivants sont censés avoir été abordé pendant le cours "*Programmation niveau 1*"

Ce module est une introduction à la programmation C Ansi.

1 - Le préprocesseur	6 - Les instructions conditionnelles
2 - Type de données	6.1 Blocs d'instructions
2.1 Conversions	6.2 Saut incondtionnel arrêt incondtionnel
3 - Les constantes	7 - Les tableaux
4 - Les variables	7.1 Unidimensionnels
4.1 Déclarations	7.2 Multidimensionnels
4.2 Initialisation	8 - Les chaînes de caractères
4.3 Portée	9 - Les fonctions
5 - Les opérateurs	9.1 Prototype
5.1 Calcul	9.2 Déclaration
5.2 Assignment	9.3 Appel
5.3 Incrémentation	9.4 Arguments
5.4 Comparaison	9.5 Passage par valeur
5.5 Logiques	9.6 Passage des tableaux
5.6 Bit à bit	9.7 Renvois d'une valeur
5.7 Priorités	9.8 Variables locales et globales

Sans reprendre le cours point par point, ces éléments seront utilisés dans la section *Rappels*.

Le tableau ci-dessous résume les concepts qui seront abordés dans le cours de *Programmation C Avancée*. Sont indiqué en rouge les notions critiques en informatique.

1 - Types de données	4 - Pointeurs
1.1 Base (Rappels)	4.1 Généralités
1.2 Structures de données	4.2 Opérations élémentaires
1.3 Tableaux de structures	4.3 Pointeurs et tableaux
1.4 Définition de types	4.4 Pointeurs et chaînes de caractères
2 -Tableaux	4.5 Tableaux de pointeurs
2.1 Bases (Rappels)	5 - Allocation dynamique de mémoire
2.2 Tableaux à 1 dimension	5.1 Allocation
2.3 Tableaux à n dimensions	5.2 Libération de la mémoire
3 - Fonctions avancées	6 - Listes chaînées
3.1 Bases (Rappels)	7 - Fichiers séquentiels
3.2 Blocs et portées	7.1 Ouvrir et fermer
3.3 Passages par valeur	7.2 Lire et écrire
3.4 Passage par adresse	7.3 Mettre à jour
3.5 Passage de tableaux	8 - Fichiers avancés
	8.1 Fichiers non séquentiels

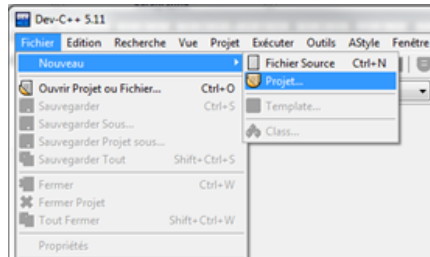
2 Rappels

Le but de cette partie est de se remémorer les bases de la programmation en C.

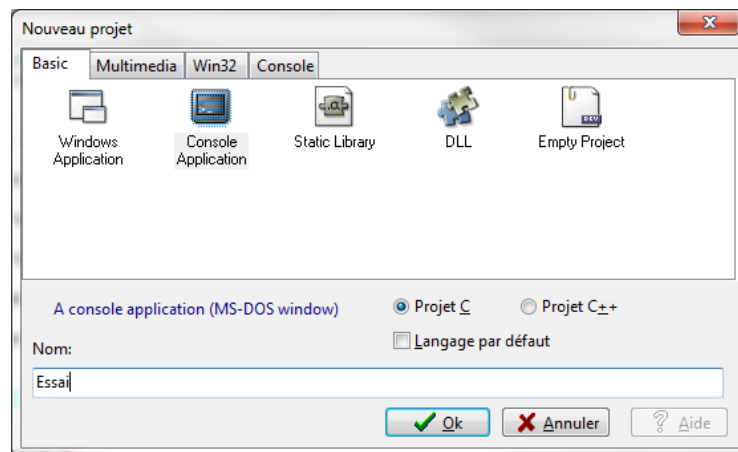
L'environnement utilisé est Dev-C++.

2.1 Création d'un projet en C sur Dev C++

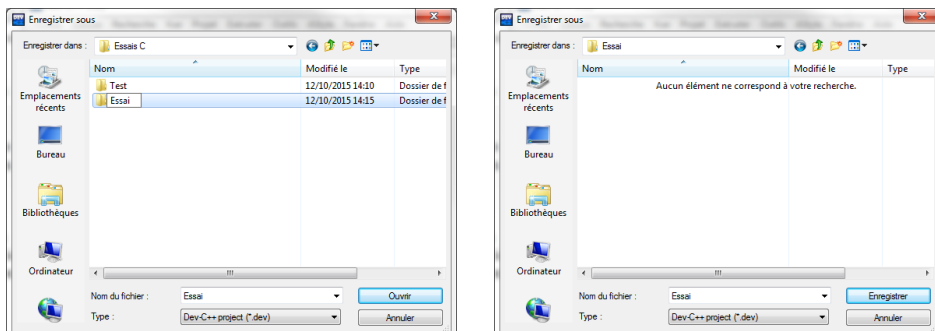
Lancez Dev C++ et créez un nouveau projet:



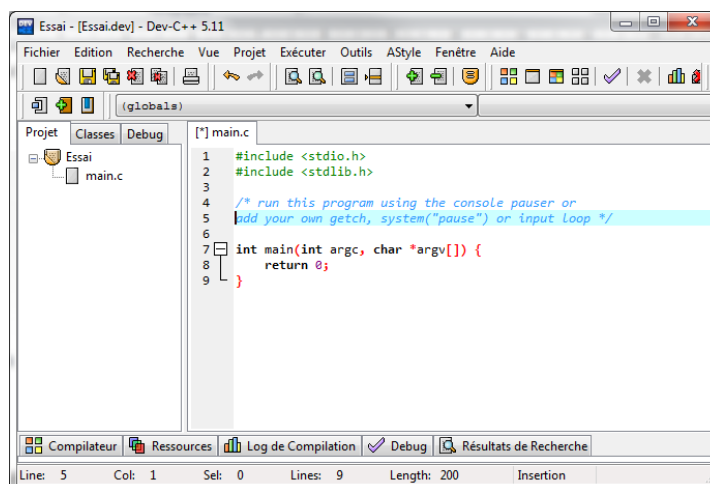
Créez un projet de type **Console** en C nommé **Essai**:



Enregistrez votre projet dans un répertoire à son nom (il vaut toujours mieux créer un projet dans un répertoire indépendant):



Dev C++ vous crée un projet vierge avec le point d'entrée du programme à venir:



Les lignes :

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

signifient que pour s'exécuter, le programme aura besoin des bibliothèques de fonction **stdio** (STanDard Input/Output) et **stdlib** (STanDard LIBrary). Ces bibliothèques contiennent les fonctions de base qui sont nécessaires à tous les programmes en C, et leur inclusion dans le projet sont donc extrêmement courante, pour ne pas dire obligatoire. Voir [ici](#) pour plus d'informations.

La partie entre /* et */ sera ignorée par le compilateur et permet d'inclure des larges portions de commentaires dans le programme. Les lignes uniques sont commentées avec //.

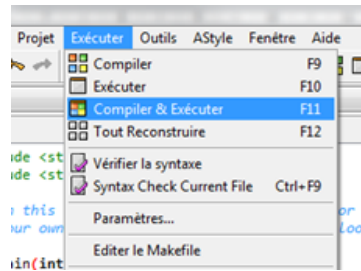
La ligne :

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

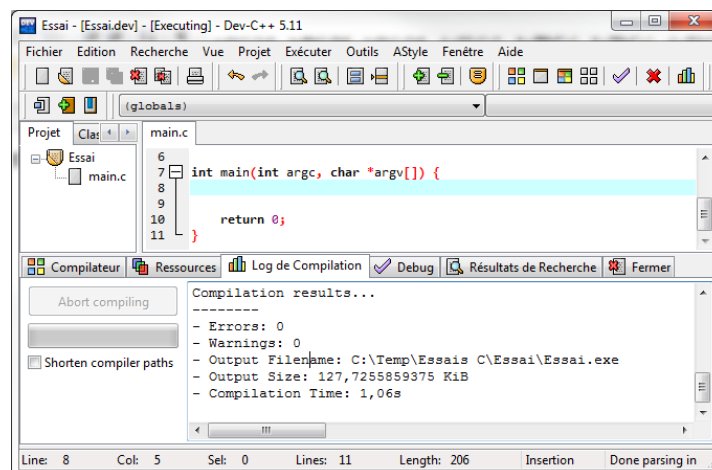
indique au programme qui transformera par la suite votre code en langage machine que le point d'entrée (là où débute l'exécution du programme) se trouve ici (en C, la fonction qui est exécutée en premier dans un programme s'appelle toujours 'main'; voir sur [Wikipedia](#)). Comme un programme peut être appelé en lui passant des paramètres, il reçoit comme informations *argc*, le nombre (int: entier) de paramètres, et *argv*, les valeurs de ces paramètres (des caractères; *char* signifie que les valeurs reçues sont des caractères, et le '*' précise que ce sont les adresses des premiers caractères de chaque paramètre qui seront fournies dans un tableau '[]').

Le *int* au début de la ligne déclare que la fonction renverra un nombre. Cela est obligatoire pour le main, et le plus souvent, un zéro indique que tout s'est bien passé.

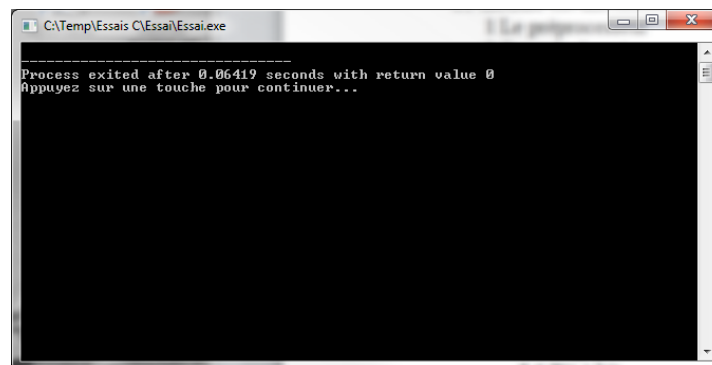
Compilez (le programme sera découpé et transformé en langage machine par le compilateur de Dev-C++) et exécutez votre programme (même vide !) :



Dev C++ affiche ceci:



et une fenêtre s'ouvre (c'est votre programme) qui affiche cela:

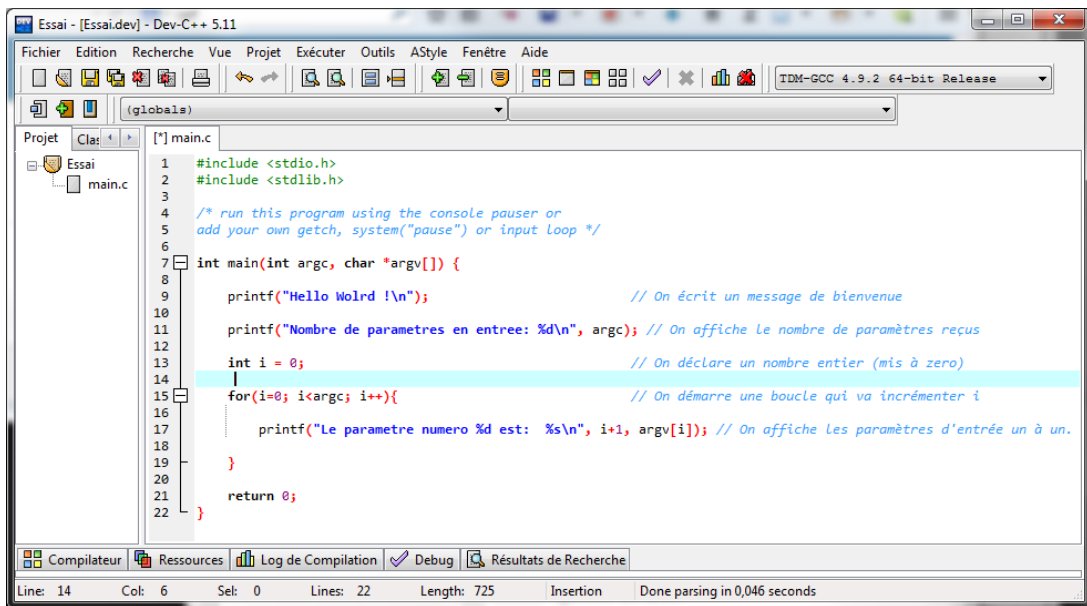


Le système affiche un message de fin de programme par défaut; vous pouvez soit appuyer sur une touche, soit fermer la fenêtre pour qu'il soit tout à fait terminé. En pratique un fichier avec l'extension '**Essai.exe**' a été créé dans le répertoire où vous avez enregistré votre projet, et c'est ce fichier dont l'exécution a été lancée par Dev C++ (voir le nom de la fenêtre d'exécution).

2.2 Analyse des paramètres d'entrée

Nous avons vu comment créer un projet et quels sont les éléments qui sont créés par défaut par Dev-C++. Mais nous avons vu que le 'main' reçoit des paramètres. Quels sont ils ?

Modifiez le programme de tests précédent de la manière suivante:

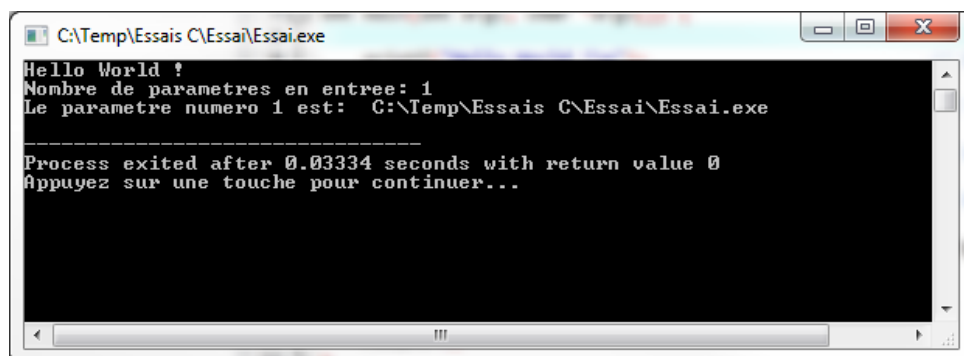


```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* run this program using the console pauser or
5 add your own getch, system("pause") or input loop */
6
7 int main(int argc, char *argv[]) {
8
9     printf("Hello Wolrd !\n");           // On écrit un message de bienvenue
10
11     printf("Nombre de parametres en entree: %d\n", argc); // On affiche Le nombre de parametres reçus
12
13     int i = 0;                          // On déclare un nombre entier (mis à zero)
14
15     for(i=0; i<argc; i++){              // On démarre une boucle qui va incrémenter i
16
17         printf("Le parametre numero %d est:  %s\n", i+1, argv[i]); // On affiche Les parametres d'entrée un à un.
18     }
19
20     return 0;
21 }
```

Pour plus d'information sur les fonctions utilisées, voir:

- la fonction **printf**.
- les boucles **for**.

Lancez l'exécution (**Compiler & Exécuter**, ou **F11**). Vous obtenez :

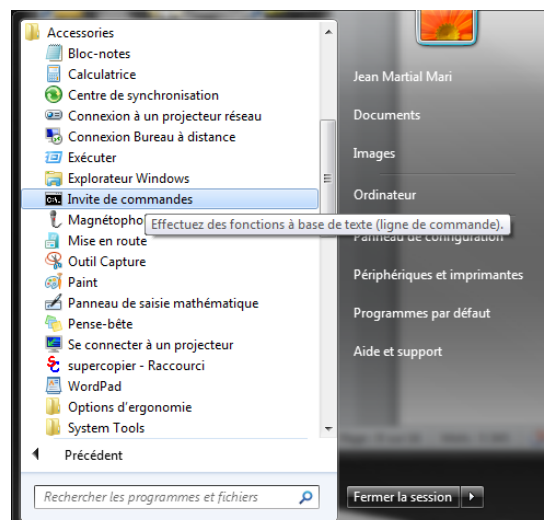


```
C:\Temp\Essais C\Essai\Essai.exe
Hello World !
Nombre de parametres en entree: 1
Le parametre numero 1 est: C:\Temp\Essais C\Essai\Essai.exe

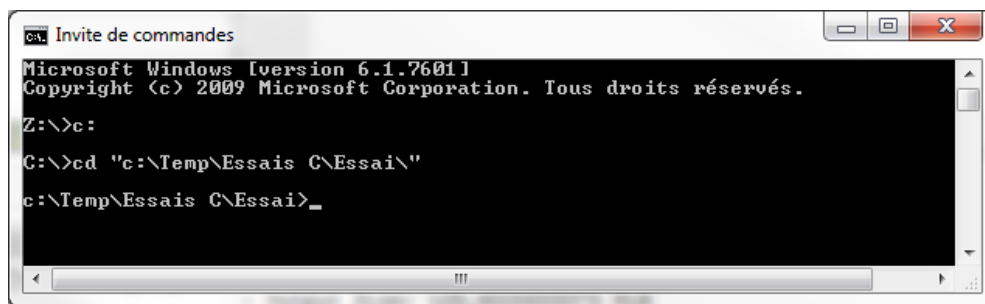
Process exited after 0.03334 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Vous constatez que si vous ne précisez rien de plus, le programme reçoit déjà par défaut le chemin d'accès à l'endroit où il est enregistré. On peut aussi voir qu'il a été créé avec le nom 'Essai.exe', l'extension 'exe' précisant qu'il s'agit d'un programme exécutable par le système Windows.

Lancez maintenant une fenêtre DOS:



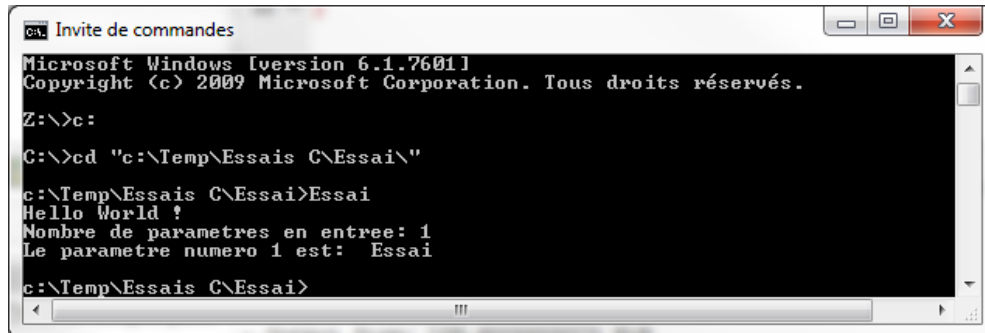
et allez dans le répertoire de travail (là où se trouve votre exécutable):



```
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

Z:\>cd "c:\Temp\Essais C\Essai\"
c:\Temp\Essais C\Essai>
```

et lancez le programme 'Essai.exe' :

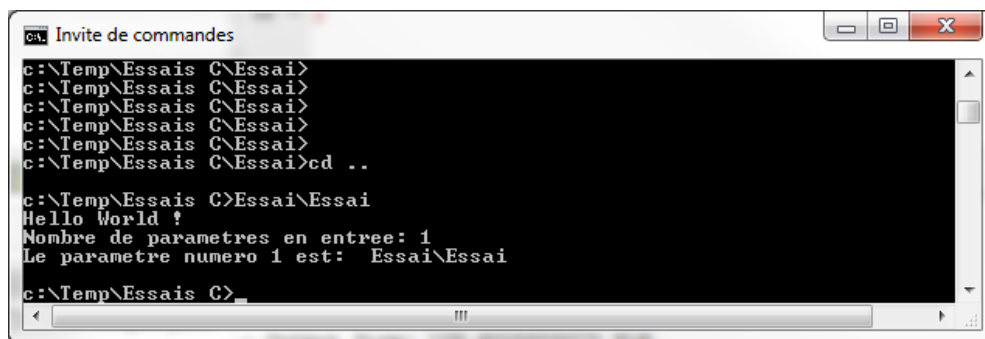


```
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

Z:\>cd "c:\Temp\Essais C\Essai\"
c:\Temp\Essais C\Essai>Essai
Hello World !
Nombre de parametres en entree: 1
Le parametre numero 1 est: Essai
c:\Temp\Essais C\Essai>
```

On peut voir que le nombre de paramètres par défaut est toujours 1, mais que ici la valeur du paramètre fourni par défaut est uniquement le nom à l'exécution: Comme on se trouve dans le répertoire où se trouve le fichier, le chemin d'accès pour le programme est nul, et seul son nom est fourni.

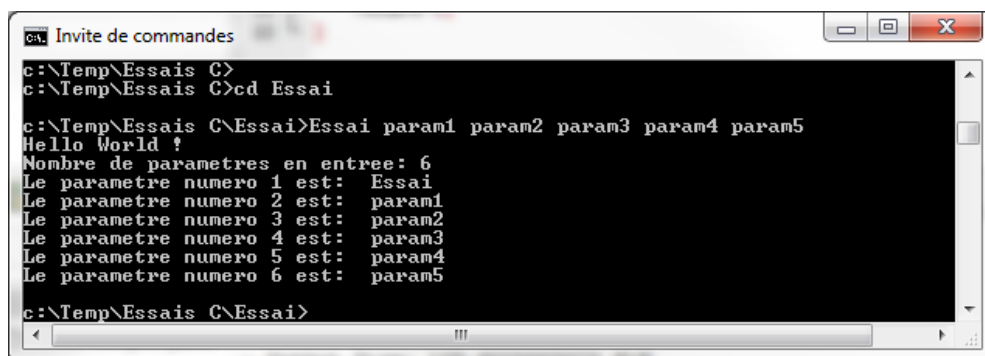
Si on remonte dans l'arborescence (en tapant 'cd ..') et qu'on relance l'exécution en précisant dans quel répertoire se situe le programme, on voit bien que c'est le chemin d'accès relatif qui est fourni au programme:



```
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

c:\Temp\Essais C\Essai>
c:\Temp\Essais C\Essai>
c:\Temp\Essais C\Essai>
c:\Temp\Essais C\Essai>
c:\Temp\Essais C\Essai>
c:\Temp\Essais C\Essai>cd ..
c:\Temp\Essais C>Essai\Essai
Hello World !
Nombre de parametres en entree: 1
Le parametre numero 1 est: Essai\Essai
c:\Temp\Essais C>
```

Lancez maintenant le programme en tapant à la suite du nom du programme des noms de paramètres (aléatoires):

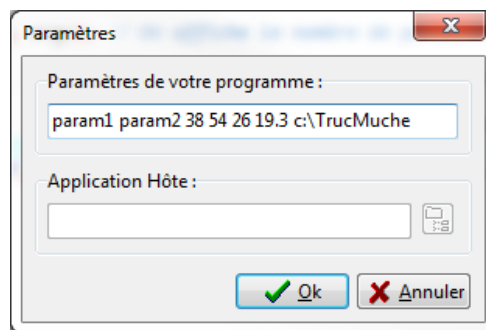


```
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

c:\Temp\Essais C>
c:\Temp\Essais C>cd Essai
c:\Temp\Essais C\Essai>Essai param1 param2 param3 param4 param5
Hello World !
Nombre de parametres en entree: 6
Le parametre numero 1 est: Essai
Le parametre numero 2 est: param1
Le parametre numero 3 est: param2
Le parametre numero 4 est: param3
Le parametre numero 5 est: param4
Le parametre numero 6 est: param5
c:\Temp\Essais C\Essai>
```

Vous constatez que tous les paramètres passés en ligne de commande sont donnés au programme et peuvent être récupérés. **Tous les paramètres d'entrée sont reçus par le programme sous forme de chaînes de caractères** (ont reçu leurs adresses en mémoire, d'où le **char** *), même les nombres (on reçoit une représentation sous forme de chaîne de caractère d'un nombre quand c'en est un), qui doivent donc être transformés en format numérique avant de pouvoir être utilisés dans des formules mathématiques.

Les valeurs de lignes de commande peuvent être passées directement depuis Dev-C++, en utilisant l'option **Paramètres** du menu **Exécuter**.
Exemple:



Et le résultat après compilation et exécution est le suivant:

```
Hello World ?
Nombre de parametres en entree: 8
Le parametre numero 1 est: C:\Temp\Essais C\Essai\Essai.exe
Le parametre numero 2 est: param1
Le parametre numero 3 est: param2
Le parametre numero 4 est: 38
Le parametre numero 5 est: 54
Le parametre numero 6 est: 26
Le parametre numero 7 est: 19.3
Le parametre numero 8 est: c:\TrucMuche

-----
Process exited after 0.02252 seconds with return value 0
Appuyez sur une touche pour continuer... _
```

Mais attention, les nombres affichés n'en sont pas: encore une fois, tous les paramètres sont traités comme des suites de caractères séparées par des espaces.

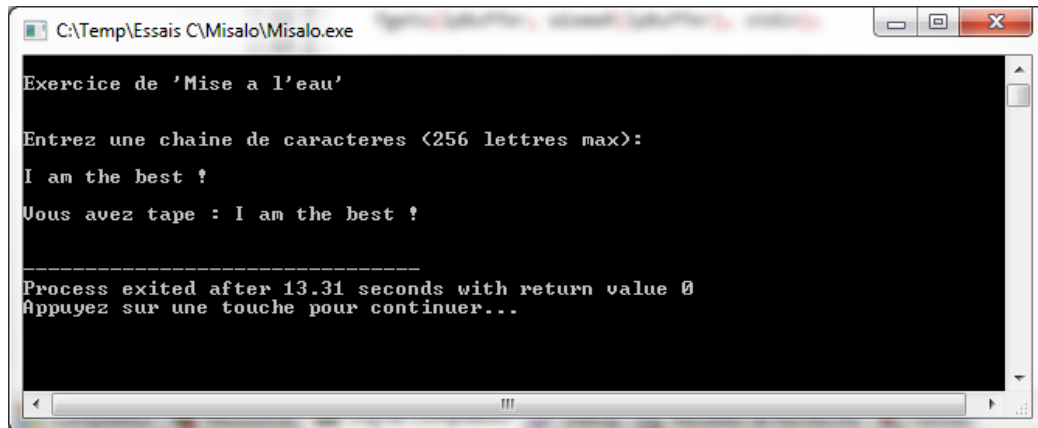
2.3 Pour se remettre dans le bain

Entrées/sorties

Une des tâches les plus courantes en programmation est d'effectuer des traitements en fonction des valeurs, texte ou nombre, entrées au clavier par l'utilisateur, et souvent d'afficher le résultat à l'écran. Il convient donc de savoir comment réaliser de telles entrées/sorties.

Exercice de *Mise à l'eau*:

Créez un petit programme en C (Console, langage C) qui crée un tableau de 256 caractères, demande à l'utilisateur de taper une phrase de moins de 256 lettres (et espaces!) et affiche ensuite la chaîne à l'écran:



```
C:\Temp\Essais C\Misalo\Misalo.exe

Exercice de 'Mise a l'eau'

Entrez une chaine de caracteres <256 lettres max>:
I am the best !
Vous avez tape : I am the best !

-----
Process exited after 13.31 seconds with return value 0
Appuyez sur une touche pour continuer...
```

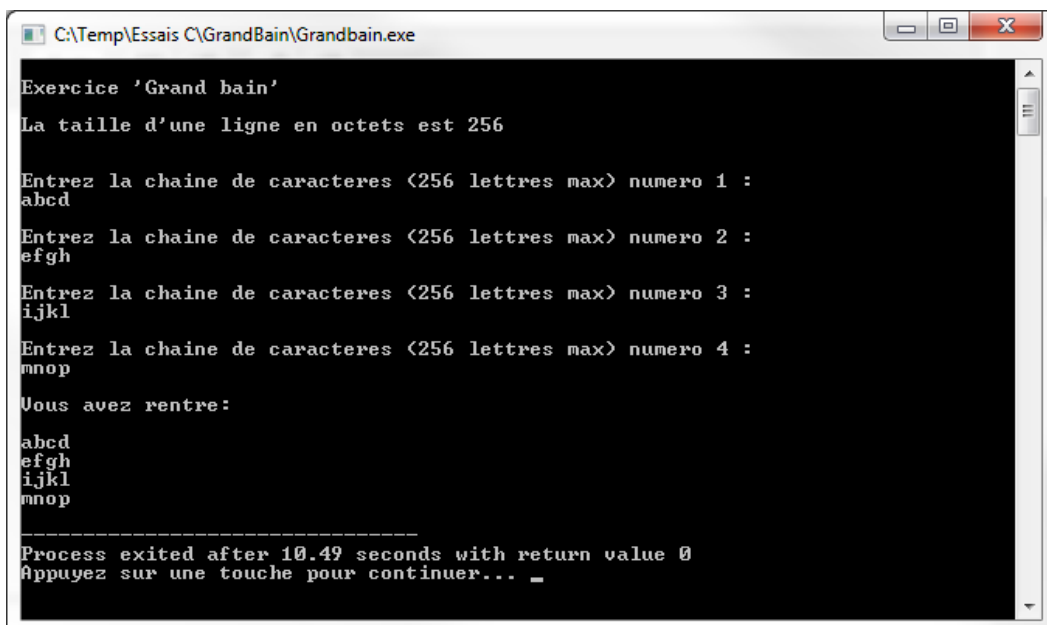
Indice: Effectuez la recherche Google suivante "initiation langage C saisie d'une chaîne caractères entrée clavier".

Questions:

- Cherchez sur Google comment la fonction **fgets** marque la fin de la chaîne de caractères dans le tableau de stockage.
- Cherchez sur Google les propriétés de la fonction **sizeof()**.

Exercice en *Grand bain*:

Créez un nouveau projet (avec son propre répertoire, console, langage C) intitulé *Grandbain*, et modifiez-y le programme précédent pour ne pas lire une, mais plusieurs lignes rentrées successivement à l'écran:



```
C:\Temp\Essais C\GrandBain\Grandbain.exe

Exercice 'Grand bain'
La taille d'une ligne en octets est 256

Entrez la chaine de caracteres <256 lettres max> numero 1 :
abcd
Entrez la chaine de caracteres <256 lettres max> numero 2 :
efgh
Entrez la chaine de caracteres <256 lettres max> numero 3 :
ijkl
Entrez la chaine de caracteres <256 lettres max> numero 4 :
mnop

Vous avez rentre:

abcd
efgh
ijkl
mnop

-----
Process exited after 10.49 seconds with return value 0
Appuyez sur une touche pour continuer... _
```

Question: Que deviennent les caractères correspondant à la frappe de la touche "Entrée" à la fin d'une ligne entrée au clavier ?

Indice: voici une partie du programme, qu'il convient de compléter:

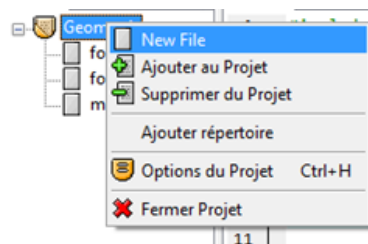
```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 4 // On définit un symbole auquel sera substitué la valeur indiquée à la compilation.
5
6 /* run this program using the console pauser or add your own getch, system("pause") or input loop */
7
8 int main(int argc, char *argv[]) {
9
10     int i;
11     char lpBuffer[N][256]; // On définit un tableau à deux dimensions de N lignes par 256 colonnes.
12
13     printf("\nExercice 'Grand bain' \n\n");
14
15     int tailleLigne = sizeof(lpBuffer[0]); // Pour ne pas recalculer la longueur d'une ligne
16                                           // à chaque fois dans la boucle.
17     printf("La taille d'une ligne en octets est %d\n", tailleLigne); // Par curiosité.
18
19     for(i=0; i<N; i++){
20         // ?
21     }
22
23     printf("\nVous avez rentre:\n\n");
24
25     for(i=0; i<N; i++){
26         // ?
27     }
28
29     return 0;
30 }

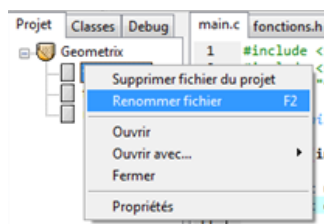
```

Exercice des Suites géométriques:

Créez dans son répertoire dédié un projet intitulé **Geometrix** (Console, langage C). Dans ce projet, par le biais d'un "clic droit", ajoutez au projet deux nouveaux fichiers



et renommez les *fonctions.h* et *fonctions.c*:



Dans le fichier **fonctions.h**, déclarez les prototypes de trois fonctions tel que ci-dessous:

```

Projet Classes Debug main.c fonctions.h fonctions.c
1
2 float getNext(float reason, float un);
3 float geometrique(float reason, float u0, int order);
4 float sommeGeometrique(float reason, float u0, int order);

```

Ce prototypage permet de déclarer au compilateur C le format des fonctions qui seront appelées dans le programme principal sans que le compilateur ait besoin de les chercher dans le reste du listing du programme.

Codez ces fonctions dans le fichier **fonction.c** en complétant le code ci-dessous (sans utiliser de fonction de type *puissance*):

```

1  #include "fonctions.h"
2
3  float getNext(float reason, float un)
4  {
5      return (/* ??? */);
6  }
7
8  float geometrique(float reason, float u0, int order){
9
10     int i;
11
12     for (i=0; i<order;i++){
13
14         /* ??? */
15
16     }
17
18     return u0;
19 }
20
21 float sommeGeometrique(float reason, float u0, int order){
22
23     int i;
24     float somme = u0;
25
26     for (i=0; i<order;i++){
27
28         /* ??? */
29
30     }
31
32     return somme;
33 }

```

et programmez le main pour avoir la sortie suivante :

```

C:\Temp\Essais C\Geometrix\Geometrix.exe

  Application des fonctions en C aux suites geometriques
  Soit une suite geometrique  $U_n = q * U_{n-1}$ 
  de raison  $q = 0.500$  et de valeur initiale  $U_0 = 3.000$ 
   $U_1 = q * U_0 = getNext(q, u0) = 1.500$ 
   $U_1 = geometrique(q, u0, 1) = 1.500$ 
   $U_2 = q * U_1 = getNext(q, getNext(q, u0)) = 0.750$ 
   $U_2 = q * U_1 = geometrique(q, u0, 2) = 0.750$ 
   $U_3 = q * U_2 = getNext(q, getNext(q, getNext(q, u0))) = 0.375$ 
   $U_3 = q * U_2 = geometrique(q, u0, 3) = 0.375$ 
   $U_0 + U_1 + U_2 + U_3 = sommeGeometrique(q, u0, 3) = 5.625$ 

  -----
  Process exited after 0.03387 seconds with return value 0
  Appuyez sur une touche pour continuer... _

```

Indice: le formatage de **printf** est le même que pour **sprintf**.

Only the brave: proposez à l'utilisateur de spécifier U_0 , q et le rang jusqu'auquel calculer la somme de la suite géométrique, et affichez le résultat.

Corrections:

Ne regardez pas les corrections sans avoir cherché, ce qui inclue une recherche d'exemples sur internet. Surtout ne recopiez pas les corrections, mais essayez plutôt de la comprendre (dites-vous à vous-même dans votre tête ce que signifie chaque portion et détail) et de la refaire vous-même.

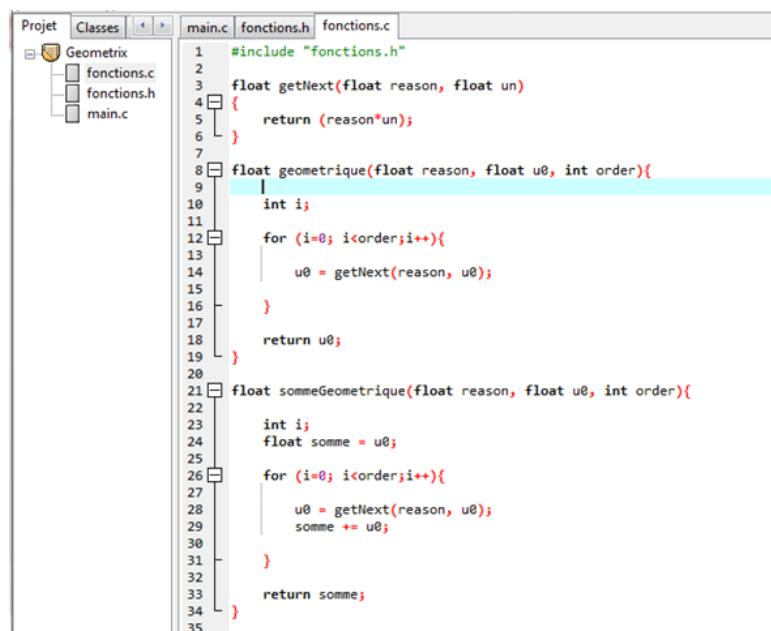
Mise à l'eau:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* run this program using the console pauser or add your own getch, system("pause") or input loop */
5
6 int main(int argc, char *argv[]) {
7     char lpBuffer[256];
8
9     printf("\nExercice de 'Mise à l'eau' \n\n");
10
11     printf("\nEntrez une chaîne de caractères (256 lettres max): \n\n");
12
13     fgets(lpBuffer, sizeof(lpBuffer), stdin);
14
15     printf("\nVous avez tapé : %s\n", lpBuffer);
16
17     return 0;
18 }
```

Grand bain:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 4 // On définit un symbole auquel sera substitué la valeur indiquée à la compilation.
5
6 /* run this program using the console pauser or add your own getch, system("pause") or input loop */
7
8 int main(int argc, char *argv[]) {
9     int i;
10     char lpBuffer[N][256]; // On définit un tableau à deux dimensions de N lignes par 256 colonnes.
11
12     printf("\nExercice 'Grand bain' \n\n");
13
14     int tailleLigne = sizeof(lpBuffer[0]); // Pour ne pas recalculer la longueur d'une ligne
15                                           // à chaque fois dans la boucle.
16     printf("La taille d'une ligne en octets est %d\n", tailleLigne); // Par curiosité.
17
18     for(i=0; i<N; i++){
19         printf("\nEntrez la chaîne de caractères (256 lettres max) numéro %d : \n", i+1);
20         fgets(lpBuffer[i], tailleLigne, stdin);
21     }
22
23     printf("\nVous avez rentré:\n\n");
24
25     for(i=0; i<N; i++){
26         printf("%s", lpBuffer[i]);
27     }
28
29     return 0;
30 }
```

Suites géométriques



Projet

Classes

main.c

fonctions.h

fonctions.c

Geometrix

fonctions.c

fonctions.h

main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "fonctions.h"
4
5 /* run this program using the console pauser or add your own getch, system("pause") or input loop */
6
7 int main(int argc, char *argv[]) {
8
9     float u0 = 3;
10    float q = 0.5;
11
12    printf("\n Application des fonctions en C aux suites geometriques\n");
13
14    printf("\n Soit une suite geometrique Un = q * Un-1\n");
15    printf("\n de raison q = %.3f et de valeur initiale U0 = %.3f\n", q, u0);
16
17    printf("\n U1 = q*U0 = getNext(q, u0) = %.3f\n", getNext(q, u0));
18    printf("\n U1 = geometrique(q, u0, 1) = %.3f\n", geometrique(q, u0, 1));
19    printf("\n U2 = q*U1 = getNext(q, getNext(q, u0)) = %.3f\n", getNext(q, getNext(q, u0)));
20    printf("\n U2 = q*U1 = geometrique(q, u0, 2) = %.3f\n", geometrique(q, u0, 2));
21    printf("\n U3 = q*U2 = getNext(q, getNext(q, getNext(q, u0))) = %.3f\n", getNext(q, getNext(q, getNext(q, u0))));
22    printf("\n U3 = q*U2 = geometrique(q, u0, 3) = %.3f\n", geometrique(q, u0, 3));
23
24    printf("\n U0 + U1 + U2 + U3 = sommeGeometrique(q, u0, 3) = %.3f\n", sommeGeometrique(q, u0, 3));
25
26    return 0;
27 }
```



3 Rappels sur la compilation

LA COMPILATION, L'EDITION DE LIEN, LES EXECUTABLES

Utilisation des Bibliothèques de Fonctions

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

#include

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

Identification des Fichiers

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

*.C	fichiers source
*.OBJ	fichiers compilés (versions objet)
*.EXE	fichiers compilés et liés (versions exécutables)
*.LIB	bibliothèques de fonctions précompilées
*.H	fichiers en-tête (<i>header files</i>)

Exemple

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

```
MATHS.LIB  
GRAPHICS.LIB
```

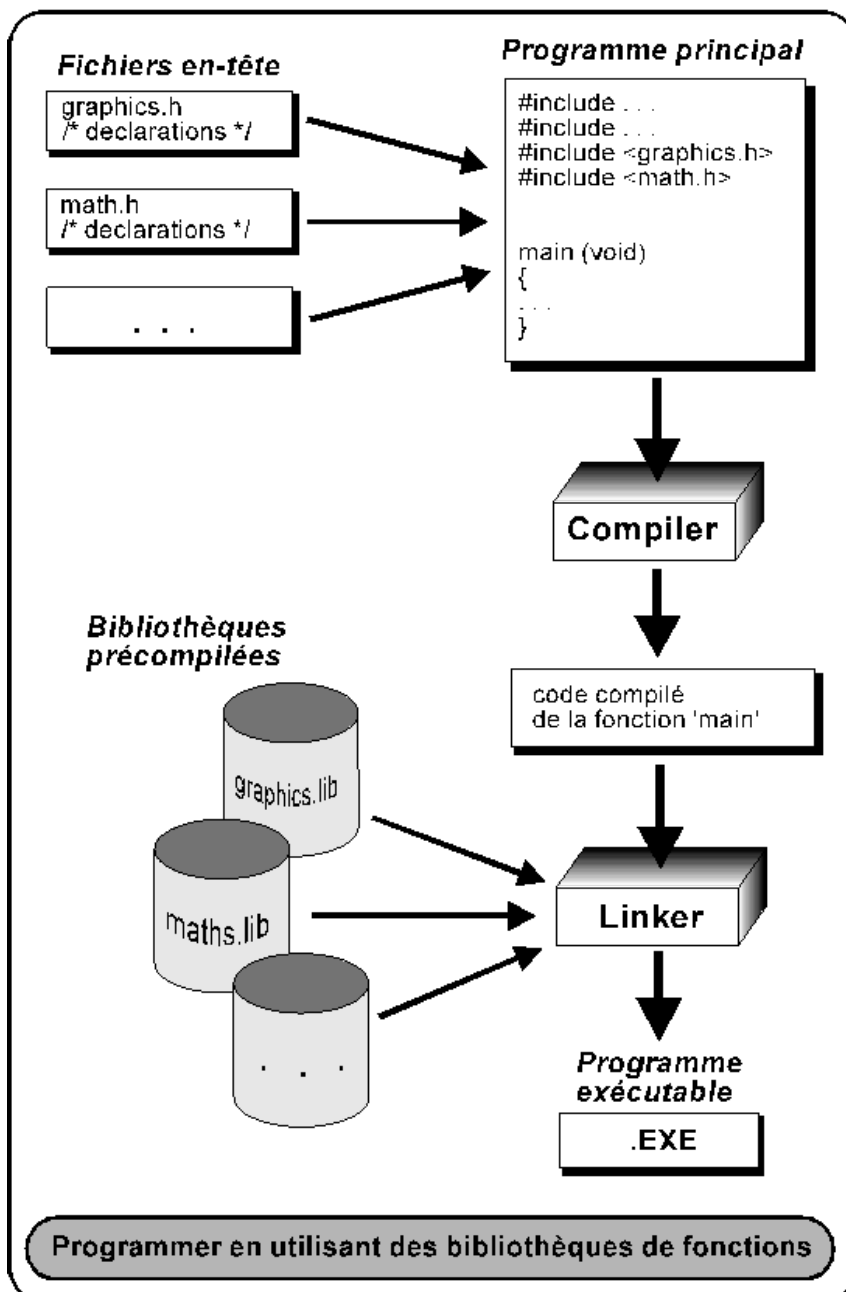
Nous devons donc inclure les fichiers en-tête correspondants dans le code source de notre programme à l'aide des instructions:

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

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

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

Schéma: Bibliothèques de Fonctions et Compilation



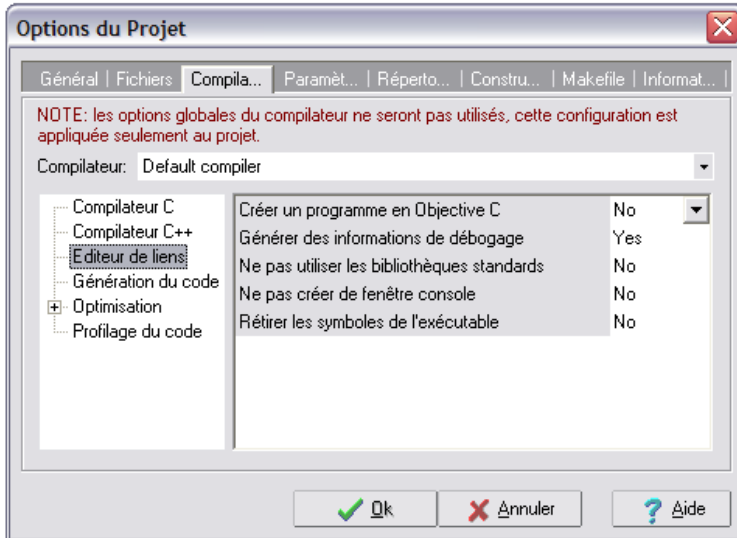
Exercice: cherchez sur votre disque et dans votre arborescence quels sont les fichiers créés par la compilation, leurs liens et où sont ils placés.

4 Debugger avec Dev C++

Déboguer votre programme

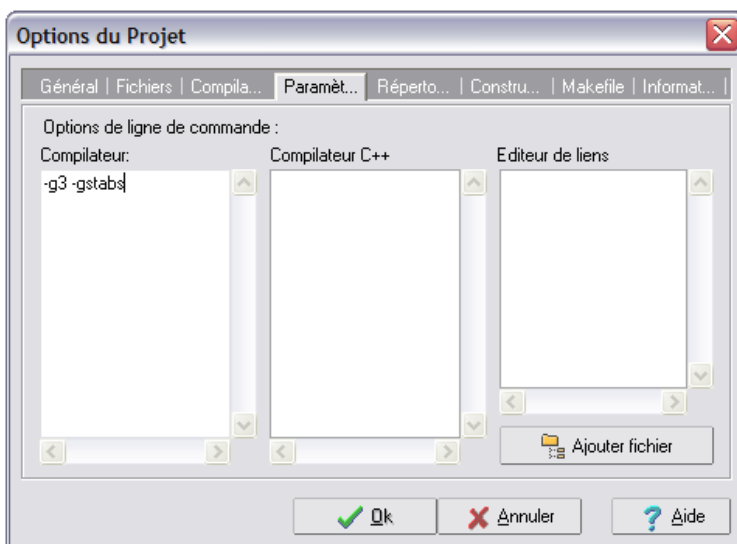
Un *débogueur* est un outil pour exécuter un programme pas à pas et en permettant d'examiner le contenu des variables. Cela permet de comprendre le comportement de l'application et comment ses variables évoluent. C'est un moyen précieux pour trouver les fautes de programmation, et aussi pour parfaire sa connaissance de la programmation en examinant de l'intérieur comment les programmes marchent.

Pour qu'un programme puisse être contrôlé par le débogueur il faut que le fichier exécutable ait gardé certaines informations symboliques, comme les noms des variables et des fonctions, qui sont habituellement éliminées durant la compilation. A cet effet il faut positionner une option de l'éditeur de liens : commande **Options du Projet** du menu **Projet**, volet **Compilation**, choisir **Editeur de liens** et donner la valeur **Yes** à l'option **Générer des informations de débogage** (laisser les autres options à **No**).

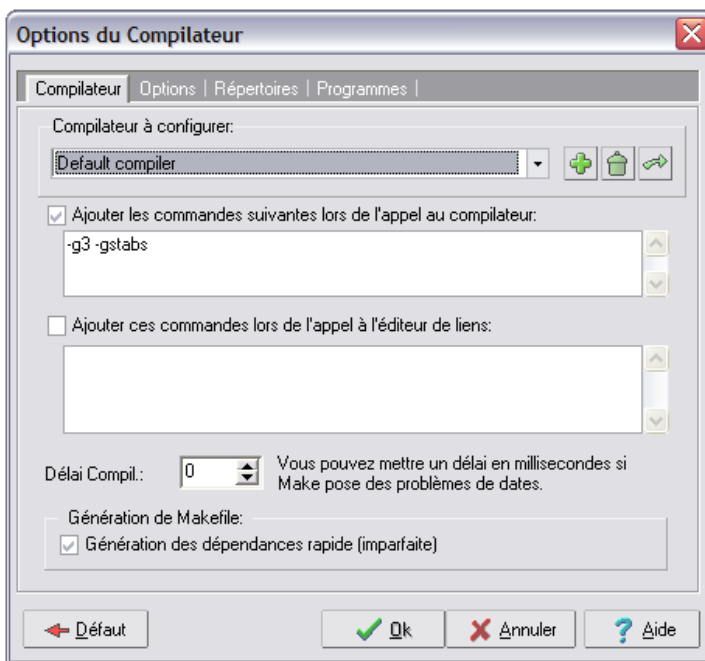


Après avoir mis à *Yes* l'option *Générer les informations de débogage* il faut recompiler le programme avec la commande **Tout Reconstruire** du menu **Exécuter** (la commande *Compiler* risquerait de ne pas faire le travail).

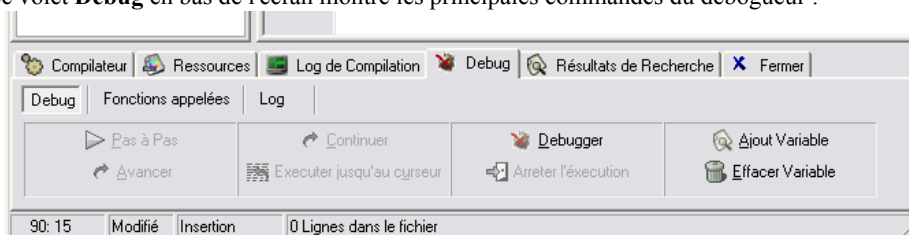
Note 1. Dans certains cas, les actions précédentes ne suffisent pas à mettre Dev-C++ dans un état rendant possible le débogage. Une manière d'atteindre cet état à coup sûr consiste à ajouter la ligne "**-g3 -gstabs**" dans la fenêtre **Compilateur:** du volet **Paramètres** du panneau **Options du projet** (commande **Options du Projet** du menu **Projet**) :



Note 2. L'une et l'autre des manipulations précédentes peuvent se faire en agissant sur des panneaux plus ou moins analogues obtenus à travers la commande **Options du compilateur** du menu **Outils**. Ces actions portent alors sur tous les projets que vous créerez et non uniquement sur le projet en cours :



Le volet **Debug** en bas de l'écran montre les principales commandes du débogueur :



Attention. Il faut être tolérant, le débogueur n'est pas un programme très robuste et, dans certaines circonstances, ses commandes semblent ne pas avoir d'effet. En outre, faites attention à ne pas laisser des sessions de débogage actives par inadvertance, car cela met Dev-C++ dans un état malsain. En principe, la commande **Arrêter l'exécution** du menu **Debug** fait quitter le débogage et remet Dev-C++ dans l'état " normal ".

Il a deux manières principales de lancer le débogueur :

- placer un point d'arrêt (*breakpoint*) puis actionner la commande **Debugger**
- placer le curseur au début d'une instruction puis actionner la commande **Exécuter jusqu'au curseur**

La manière la plus simple de placer un point d'arrêt consiste à cliquer dans la gouttière (la marge de gauche). Une marque dans la gouttière indique le point d'arrêt, ainsi qu'un surlignage de la ligne concernée. D'autre part, une flèche dans la gouttière montre constamment la ligne sur laquelle l'exécution est arrêtée. Par exemple, la figure ci-dessous montre un moment d'une session de débogage, avec l'exécution arrêtée à la ligne 20, un point d'arrêt étant placé à la ligne 24 (les couleurs avec lesquelles sont surlignées certaines lignes peuvent être redéfinies par la commande **Options de l'éditeur** du menu **Outils**, volet **Syntaxe**, types **Breakpoints** et **Active breakpoints**) :

```

17 int main(int argc, char *argv[]){
18     double u, v;
19     int k;
20     u = 2;
21     k = 10;
22     v = puiss(u, k);
23
24     printf("%g ^%d = %g\n", u, k, v);
25     return 0;
26 }

```

Un programme ne peut être arrêté que sur des instructions, évitez de mettre des points d'arrêt sur des lignes constituées de déclarations (des déclarations il ne reste aucune trace après la compilation).

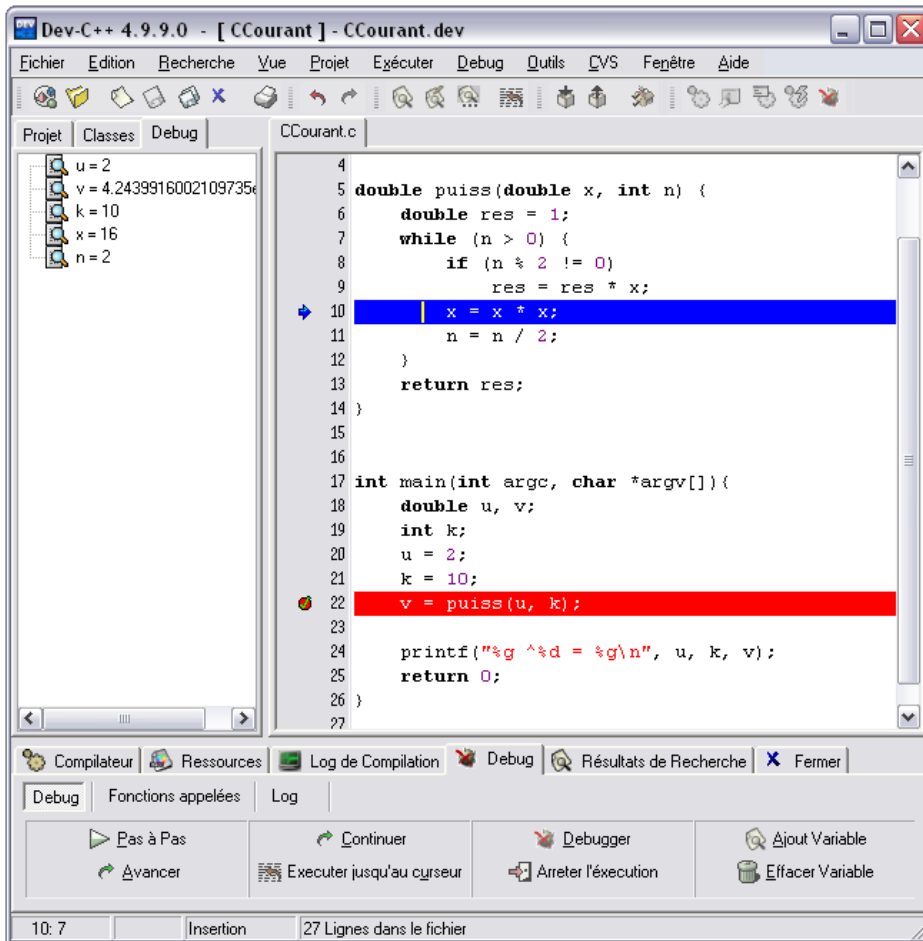
Lorsque le débogueur est bloqué (sur un point d'arrêt ou consécutivement à l'emploi de la commande *Exécuter jusqu'au curseur*) on doit le débloquer par une des commandes :

- **Pas à Pas** (*Next Step*) : exécuter une instruction, en considérant qu'un appel de fonction est une instruction atomique qu'il n'y a pas lieu de détailler,
- **Avancer** (*Step Into*) : avancer d'une instruction, en s'arrêtant, le cas échéant, à l'intérieur des fonctions appelées,
- **Continuer** : relancer l'exécution du programme, jusqu'au prochain point d'arrêt ou, s'il n'y en a plus, jusqu'à la fin.

Examiner les variables. Pour faire afficher une variable dans le volet *Debug* à gauche de l'écran il suffit de presser le bouton **Ajout variable** ou bien de double-cliquer sur la variable. En fait, passer (lentement, soyez patients) le curseur sur la variable suffit la plupart du temps pour l'ajouter au volet *Debug*. La variable et sa valeur sont ensuite constamment affichées et on peut en observer l'évolution pendant que le

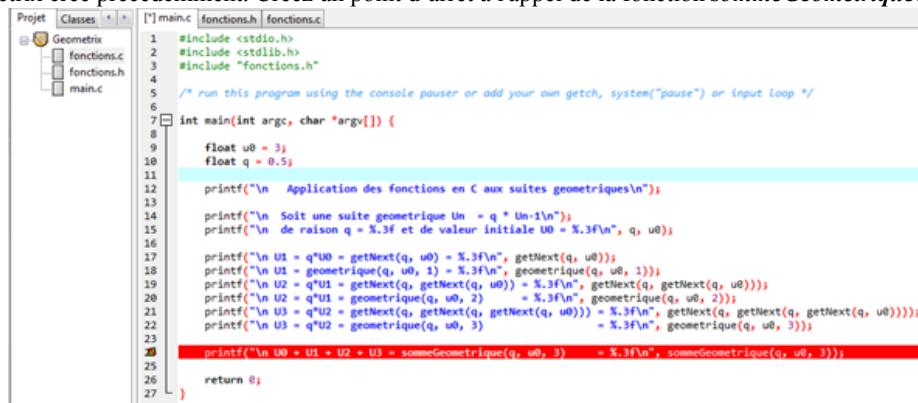
programme est exécuté.

Lorsque la variable est complexe, le volet *Debug* permet d'en examiner les éléments.

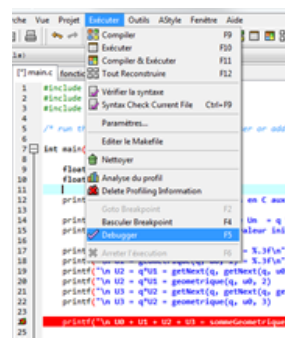


Débogage de Geometrix

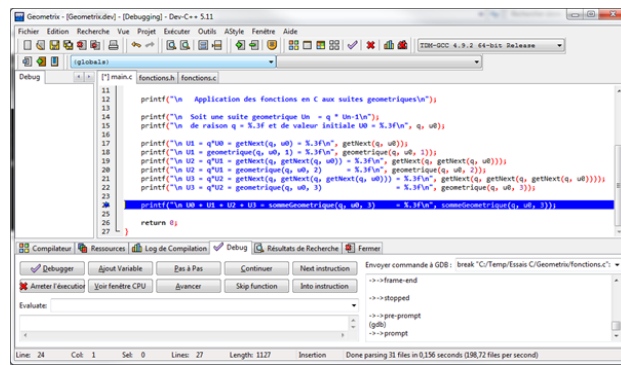
Rouvrez le projet Geometrix créée précédemment. Créez un point d'arrêt à l'appel de la fonction *sommeGeometrique*:



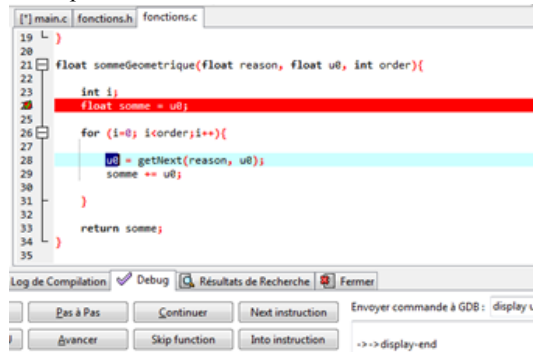
Et lancez le Débogage:



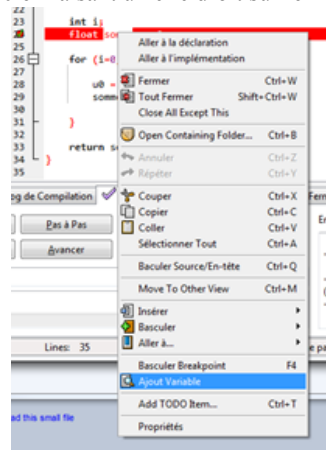
Si le logiciel est bien installé, l'exécution débute et s'arrête au point d'arrêt:



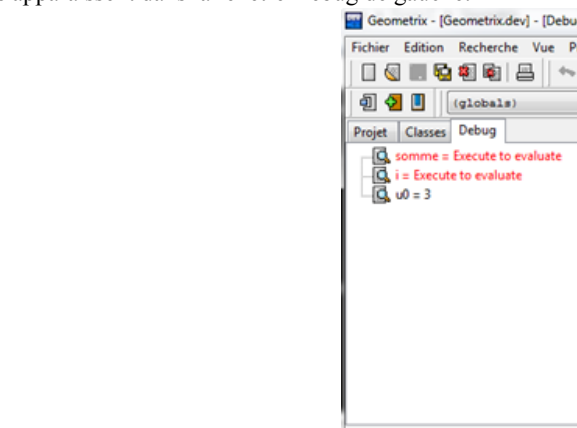
Allez dans *sommeGeometrique*, et placez un autre point d'arrêt:



puis planifiez l'observation des variables somme, i et u0 en faisant un clic droit sur le nom de ces variables dans le code:

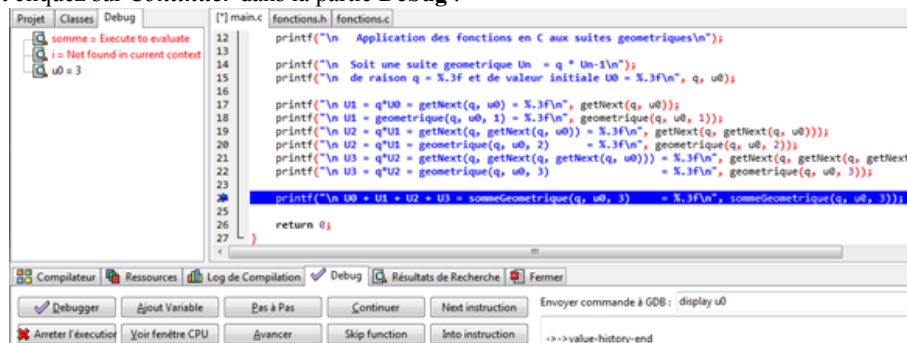


Et les variables observées apparaissent dans la fenêtre Debug de gauche:



On peut voir que la valeur de U_0 apparaît déjà (3), mais pas les deux autres. Pourquoi ?

Retournez dans le main et cliquez sur **Continuer** dans la partie **Debug** :



L'exécution continue jusqu'au point d'arrêt:

```

19 }
20
21 float sommeGeometrique(float reason, float u0, int order){
22
23     int i;
24     float somme = u0;
25     for (i=0; i<order; i++){
26         u0 = getNext(reason, u0);
27         somme += u0;
28     }
29     return somme;
30 }
31
32
33
34
35

```

Vous pouvez remarquer que les variables ont maintenant des valeurs, bien que la valeur attribuée à **somme** semble arbitraire et sans explication; la variable n'ayant pas été initialisée, c'est la valeur résiduelle stockée à l'emplacement mémoire de la variable qui est lue (une raison pour toujours initialiser ses variables). En cliquant sur **Avancer**, on passe la ligne d'initialisation de **somme** et la valeur lue au débogage est maintenant correcte:

```

19 }
20
21 float sommeGeometrique(float reason, float u0, int order){
22
23     int i;
24     float somme = u0;
25     for (i=0; i<order; i++){
26         u0 = getNext(reason, u0);
27         somme += u0;
28     }
29     return somme;
30 }
31
32
33
34
35

```

Allez placer un point d'arrêt dans la fonction **getNext**:

```

1 #include "fonctions.h"
2
3 float getNext(float reason, float un)
4 {
5     return (reason*un);
6 }
7
8 float geometrique(float reason, float u0, int order){

```

et cliquez sur **Continuer**: l'exécution continue jusqu'au point d'arrêt.

```

1 #include "fonctions.h"
2
3 float getNext(float reason, float un)
4 {
5     return (reason*un);
6 }
7
8 float geometrique(float reason, float u0, int order){

```

Passez la souris sur **reason** et sur **un**: leur valeur est affichée dans un popup:

```

1 #include "fonctions.h"
2
3 float getNext(float reason, float un)
4 {
5     return (reason*un);
6 }
7
8 float geometrique(float reason, float u0, int order){

```

Cela permet de contrôler l'évolution des valeurs des variables facilement pendant l'exécution.

Cliquez sur **Avancer**, et le compteur de programme (la flèche bleue) se place en position de sortie de la fonction:

```

2
3 float getNext(float reason, float un)
4 {
5     return (reason*un);
6 }
7
8 float geometrique(float reason, float u0, int order){

```

Cliquez encore une fois sur **Avancer**, et l'exécution se poursuit à l'instruction suivant l'appel de la fonction **getNext**:

```

19 }
20
21 float sommeGeometrique(float reason, float u0, int order){
22
23     int i;
24     float somme = u0;
25     for (i=0; i<order; i++){
26         u0 = getNext(reason, u0);
27         somme += u0;
28     }
29     return somme;
30 }
31
32
33
34
35

```

Cliquez encore une fois sur **Avancer**, la valeur de **somme** dans la fenêtre de **Debug** est mise à jour (vérifiez que la valeur est bonne); pourquoi

la valeur de u_0 n'a-t-elle pas changé, alors qu'en passant la souris sur u_0 dans le code, la valeur affichée est bien mise à jour ? A quel endroit dans le code cette valeur de u_0 est elle correcte ? Concluez.

```

19 }
20
21 float sommeGeometrique(float reason, float u0, int order){
22     int i;
23     float somme = u0;
24
25     for (i=0; i<order;i++){
26
27         u0 = getNext(reason, u0);
28         u0 = 1.5 * u0;
29     }
30
31     return somme;
32 }
33
34 }
  
```

Continuez à dérouler l'exécution du programme en cliquant sur **Avancer**, jusqu'à arriver à la ligne **return** de **sommeGeometrique**:

```

19 }
20
21 float sommeGeometrique(float reason, float u0, int order){
22     int i;
23     float somme = u0;
24
25     for (i=0; i<order;i++){
26
27         u0 = getNext(reason, u0);
28         somme += u0;
29     }
30
31     return somme;
32 }
33
34 }
  
```

Avancer encore deux fois et revenez au programme principal:

```

12 printf("\n Application des fonctions en C aux suites geometriques\n");
13
14 printf("\n Soit une suite geometrique Un = q * Un-1\n");
15 printf("\n de raison q = %.3f et de valeur initiale u0 = %.3f\n", q, u0);
16
17 printf("\n U1 = q*u0 = getNext(q, u0) = %.3f\n", getNext(q, u0));
18 printf("\n U1 = geometrique(q, u0, 1) = %.3f\n", geometrique(q, u0, 1));
19 printf("\n U2 = q*U1 = getNext(q, getNext(q, u0)) = %.3f\n", getNext(q, getNext(q, u0)));
20 printf("\n U2 = q*U1 = geometrique(q, u0, 2) = %.3f\n", geometrique(q, u0, 2));
21 printf("\n U3 = q*U2 = getNext(q, getNext(q, getNext(q, u0))) = %.3f\n", getNext(q, getNext(q, getNext(q, u0))));
22 printf("\n U3 = q*U2 = geometrique(q, u0, 3) = %.3f\n", geometrique(q, u0, 3));
23
24 printf("\n U0 + U1 + U2 + U3 = sommeGeometrique(q, u0, 3) = %.3f\n", sommeGeometrique(q, u0, 3));
25
26 return 0;
27 }
  
```

La valeur de u_0 est elle correcte ?

Cliquez sur **Continuer**, et finissez l'exécution.

5 Types de Données - Structures

Les types de données et leur structure.

5.1 Types de données

Récapitulation du vocabulaire

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

Motivation

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

LES TYPES ENTIERS

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

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

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

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

- **char** : caractère

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

- **int** : entier standard

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

- **short** : entier court

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

- Les modificateurs **signed/unsigned**:

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

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

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

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

LES TYPES RATIONNELS

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

	$\langle + - \rangle \langle \text{mantisse} \rangle * 10^{\langle \text{exposant} \rangle}$

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

Exemples :

<code>3.14159*10^0</code>	<code>1.25003*10^-12</code>
<code>4.3001*10^321</code>	<code>-1.5*10^3</code>

En C, nous avons le choix entre trois types de rationnels: **float**, **double** et **long double**. Dans le tableau ci-dessous, vous trouverez leurs caractéristiques (*mantisse* indique le nombre de chiffres significatifs de la mantisse.):

définition	précision	mantisse	domaine min	domaine max	nombre d'octets
float	simple	6	$3.4 * 10^{-38}$	$3.4 * 10^{38}$	4
double	double	15	$1.7 * 10^{-308}$	$1.7 * 10^{308}$	8
long double	suppl.	19	$3.4 * 10^{-4932}$	$1.1 * 10^{4932}$	10

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

LES CONSTANTES NUMERIQUES

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

Les Constantes Entières

Type automatique

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

Le type des constantes entières

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

Type forcé

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

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

Base octale et hexadécimale

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

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

Exemples

base décimale	base octale	base hexadécimale	représ. binaire
100	0144	0X64	1100100
255	0377	0xff	11111111
65536	0200000	0X10000	1000000000000000
12	014	0XC	1100

Les Constantes Rationnelles

Les constantes rationnelles peuvent être indiquées:

- *en notation décimale*, c.-à-d. à l'aide d'un point décimal: **123.4 ou -0.001 ou 1.0**
- *en notation exponentielle*, c.-à-d. à l'aide d'un exposant séparé du nombre décimal par les caractères 'e' ou 'E': **1234e-1 ou -1E-3 0.01E2**

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

Le type des constantes rationnelles

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

Les Caractères Constants

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

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

Séquences d'échappement

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

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

Le caractère NUL

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

LES CONVERSIONS DE TYPE

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

Les Conversions de Types Automatiques

Calculs et affectations

Si un opérateur a des opérandes de différents types, les valeurs des opérandes sont *converties automatiquement dans un type commun*. Ces manipulations implicites convertissent en général des types plus 'petits' en des types plus 'larges'; de cette façon on ne perd pas en précision.

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

Exemple : Considérons le calcul suivant:

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

Pour pouvoir être multiplié avec X , la valeur de I est convertie en **float** (le type le plus large des deux). Le résultat de la multiplication est du type **float**, mais avant d'être affecté à Y , il est converti en **double**. Nous obtenons comme résultat: **Y = 100.00**

Appels de fonctions

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

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

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

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

Règles de conversion automatique

Conversions automatiques lors d'une opération avec,

(1) deux entiers:

D'abord, les types **char** et **short** sont convertis en **int**.

Ensuite, l'ordinateur choisit le plus large des deux types dans l'échelle suivante: **int, unsigned int, long, unsigned long**

(2) un entier et un rationnel:

Le type entier est converti dans le type du rationnel.

(3) deux rationnels:

L'ordinateur choisit le plus large des deux types selon l'échelle suivante: **float, double, long double**

(4) affectations et opérateurs d'affectation:

Lors d'une affectation, le résultat est toujours converti dans le type de la destination. Si ce type est plus faible, il peut y avoir une perte de précision.

Exemple: Observons les conversions nécessaires lors d'une simple division:

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

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

Phénomènes imprévus ...

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

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

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

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

Perte de précision

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

Exemple:

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

Les Conversions de Type Forcées (Casting)

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

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

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

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

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

Résultat: *C=0.75*

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

5.2 Structures de données

Notion de Structure

Il est habituel en programmation d'avoir besoin d'un mécanisme permettant de grouper un certain nombre de variables de types différents au sein d'une même entité. On travaille par exemple sur un fichier de personnes et on voudrait regrouper une variable de type chaîne de caractères pour le nom, une variable de type entier pour le numéro d'employé, etc.

La réponse à ce besoin est le concept d'enregistrement : un enregistrement est un ensemble d'éléments de types différents repérés par un nom. Les éléments d'un enregistrement sont appelés des champs. Le langage C possède le concept d'enregistrement avec cependant un problème de vocabulaire :

ce que tout le monde appelle	le langage C l'appelle
enregistrement	structure
champ d'un enregistrement	membre d'une structure

Déclaration de Structure

Il y a plusieurs méthodes possibles pour déclarer des structures.

Première Méthode

La déclaration :

```
struct personne
{
    char nom[20];
    char prenom[20];
    int no_employe;
};
```

déclare l'identificateur `personne` comme étant le nom d'un type de structure composé de trois membres, dont le premier est un tableau de 20 caractères nommé `nom`, le second un tableau de 20 caractères nommé `prenom`, et le dernier un entier nommé `no_employe`.

Dans le jargon du langage C, l'identificateur `personne` est une **étiquette de structure**. On peut ensuite utiliser ce type structure pour déclarer des variables, de la manière suivante :

```
struct personne p1,p2;
```

qui déclare deux variables de type `struct personne` de noms `p1` et `p2`;

Deuxième Méthode

On peut déclarer des variables de type structure sans utiliser d'étiquette de structure, par exemple :

```
struct
{
    char nom[20];
    char prenom[20];
    int no_employe;
} p1,p2;
```

déclare deux variables de noms `p1` et `p2` comme étant deux structures de trois membres, mais elle ne donne pas de nom au type de la structure. L'inconvénient de cette méthode est qu'il sera par la suite impossible de déclarer une autre variable du même type. En effet, si plus loin on écrit :

```
struct
{
    char nom[20];
    char prenom[20];
    int no_employe;
} p3;
```

les deux structures ont beau avoir le même nombre de champs, avec les mêmes noms et les mêmes types, elles seront considérées de types différents. Il sera impossible en particulier d'écrire `p3 = p1`;

Troisième Méthode

On peut combiner déclaration d'étiquette de structure et déclaration de variables, comme ceci :

```
struct personne
{
    char nom[20];
    char prenom[20];
    int no_employe;
} p1,p2;
```

déclare les deux variables p1 et p2 et donne le nom personne à la structure. Là aussi, on pourra utiliser ultérieurement le nom struct personne pour déclarer d'autres variables :

```
struct personne pers1, pers2, pers3;
```

qui seront du même type que p1 et p2.

De ces trois méthodes c'est la première qui est recommandée, car elle permet de bien séparer la définition du type structure de ses utilisations.

Initialisation d'une Structure

Une structure peut être initialisée par une liste d'expressions constantes à la manière des initialisations de tableau. Exemple :

```
struct personne p = {"Jean", "Dupond", 7845};
```

Opérateurs sur les Structures

Accès aux Membres des Structures

Pour désigner un membre d'une structure, il faut utiliser l'opérateur de sélection de membre qui se note **•** (point).

Par exemple, si p1 et p2 sont deux variables de type struct personne, on désignera le membre nom de p1 par **p1•nom** et on désignera le membre no_employe de p2 par **p2•no_employe**.

Les membres ainsi désignés se comportent comme n'importe quelle variable et par exemple, pour accéder au premier caractère du nom de p2, on écrira : **p2•nom[0]**.

Affectation de Structures

On peut affecter une structure à une variable structure de même type, grâce à l'opérateur d'affectation :

```
struct personne p1,p2
...
p1 = p2;
```

Comparaison de Structures

Aucune comparaison n'est possible sur les structures, même pas les opérateurs == et !=.

Tableaux de Structures

Une déclaration de tableau de structures se fait selon le même modèle que la déclaration d'un tableau dont les éléments sont de type simple. Supposons que l'on ait déjà déclaré la **struct personne**, si on veut déclarer un tableau de 100 structures de ce type, on écrira :

```
struct personne t[100];
```

Pour référencer le nom de la personne qui a l'index **i** dans **t** on écrira : **t[i]•nom**.

5.3 Définitions de types

Il existe en C un moyen de donner un nom à un type. Il consiste à faire suivre le mot clé `typedef` d'une construction ayant exactement la même syntaxe qu'une déclaration de variable. L'identificateur qui est le nom de la variable dans le cas d'une déclaration de variable, est le nom du type dans le cas d'un `typedef`.

Exemple :

```
typedef int tab[10];
```

déclare `tab` comme étant le type tableau de 10 entiers, et :

```
typedef struct
{
    char nom[20];
    int no_ss;
} personne;
```

déclare `personne` comme étant le type structure à deux champs : un tableau de 20 caractères et un `int`. Ces noms de type sont ensuite utilisables dans les déclarations de variables, exactement comme un type de base :

```
tab t1,t2;          /* t1 et t2 tableaux de 10 entiers */
personne *p1,*p2;    /* p1 et p2 pointeurs vers des struct */
```

Utilité des `typedef`: La principale utilité des `typedef` est, si l'on en fait une utilisation judicieuse, de faciliter l'écriture des programmes, et d'en augmenter la lisibilité.

Application à la Définition de Type Structure

Lorsqu'on donne un nom à un type structure par `typedef`, l'utilisation est beaucoup plus aisée. En effet, si on déclare :

```
struct personne
{
    ...
}
```

les déclarations de variables se feront par :

```
struct personne p1,p2;
```

alors que si on déclare :

```
typedef struct
{
    ...
} PERSONNE;
```

les déclarations de variables se feront par :

```
PERSONNE p1,p2;
```

on voit que la seconde méthode permet d'éviter d'avoir à répéter `struct`.

De la même manière, en ce qui concerne les pointeurs, il est plus difficile d'écrire et de comprendre :

```
struct personne
{
    ...
};

struct personne *p1,*p2;    /* p1 et p2 pointeurs vers des struct */
```

que la version suivante qui donne un nom parlant au type pointeur vers `struct` :

```
typedef struct
{
    ...
} PERSONNE;

typedef PERSONNE *P_PERSONNE; /* P_PERSONNE type pointeur vers struct */

P_PERSONNE p1,p2;           /* p1 et p2 pointeurs vers des struct */
```

5.4 Exercice : Structures

Exercice: Le but de l'exercice est d'écrire un programme de gestion des étudiants:

1) Déclarez une structure ***Etudiant*** comportant les champs suivants:

- Nom
- Prénom
- Numéro d'étudiant

Utilisez pour cela la définition de type.

2) Ecrivez une fonction **lire()** qui remplit et renvoie les informations d'un ***Etudiant*** dans une structure idoine.

3) Ecrivez une fonction **affiche()** qui reçoit une structure ***Etudiant*** et affiche les informations qu'elle contient.

4) Créez une structure ***Classe*** qui contient un entier **nbEtudiants**, et un tableau d'***Etudiants*** (le nombre maximal d'étudiants étant fixé à *N*).

5) Ecrivez une fonction **lire_Classe** qui reçoit une ***Classe classe*** et la renvoie complétée, et une fonction **affiche_Classe** qui affiche la Classe reçue.

6) Ecrivez une fonction **lire_tab** qui reçoit un tableau d'étudiants et le complète, et une fonction **affiche_tab** qui affiche le tableau d'étudiants reçu.

7) Reformulez le programme principal pour demander à l'utilisateur le nombre d'étudiants, lui faire remplir le tableau d'***Etudiants***, et ensuite proposer un menu (proposez d'utiliser les deux types de fonctions réalisées): 1) afficher tous les étudiants; 2) afficher un étudiant dans le tableau en précisant son numéro; ou 3) quitter le programme.

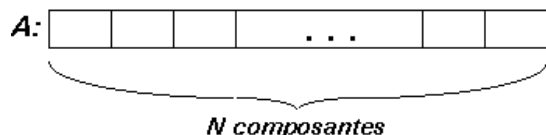
6 Les Tableaux

Les tableaux et leurs types.

6.1 Les tableaux à une dimension

Définitions

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



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

Exemple: La déclaration

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

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

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

Déclaration et Mémorisation

Déclaration

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

Les noms des tableaux sont des *identificateurs* qui doivent correspondre aux restrictions définies auparavant.

Exemples:

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

`float B[100];` ou bien `double B[100];` ou bien ...

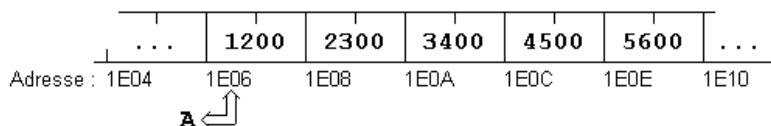
`int C[10];`

`char D[30];`

Mémorisation

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

Exemple:



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

Exemple

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

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

Initialisation et Réserve Automatique

Initialisation

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

Exemples:

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

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

Réserve automatique

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

Exemples:

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

=> réservation de $5 * \text{sizeof}(\text{int})$ octets (dans notre cas: 10 octets)

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

=> réservation de $4 * \text{sizeof}(\text{float})$ octets (dans notre cas: 16 octets)

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

=> réservation de $10 * \text{sizeof}(\text{int})$ octets (dans notre cas: 20 octets)

Exemples:

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



A:	1200	2300	3400	4500	5600
----	------	------	------	------	------

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



A:	1200	2300	3400	0	0
----	------	------	------	---	---

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



A:	1200	2300	3400	☠	☠
----	------	------	------	---	---



ERREUR !

Accès aux Composantes

En déclarant un tableau par:

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

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

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

Exemple:

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

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

Exemples:

```
MAX = (A[0]>A[1]) ? A[0] : A[1];  
A[4] *= 2;
```

ATTENTION!!! Considérons un tableau T de dimension N:

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

Affichage et Affectation

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

Affichage du contenu d'un tableau

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

Remarques

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

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

Affectation avec des valeurs provenant de l'extérieur

Reprenons l'exemple précédent

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

Remarques:

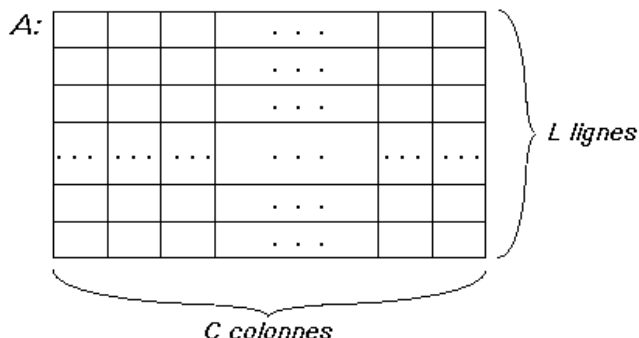
- Comme **scanf** a besoin des adresses des différentes composantes du tableau, il faut faire précéder le terme **A[I]** par l'opérateur adresse **'&'**.
- La commande de lecture **scanf** doit être informée du type exact des données à lire. (Ici: **%d** ou **%i** pour lire des valeurs du type **int**)

6.2 Les tableaux à deux dimensions

Définitions

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

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



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

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

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

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

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

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

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

NOTE :	45	34	...	50	48
	39	24	...	49	45

	40	40	...	54	44

10 lignes

20 colonnes

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

Déclaration et Mémorisation

Déclaration de tableaux à deux dimensions

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

Exemples:

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

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

```
int C[3][3];
```

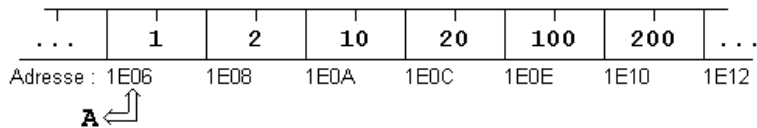
```
char D[15][40];
```

Mémorisation

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

Exemple: Mémorisation d'un tableau à deux dimensions


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



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

Exemple

En supposant qu'une variable du type **double** occupe 8 octets (c.-à-d: `sizeof(double)=8`), pour le tableau T déclaré par: **double T[10][15];** C réservera L*C*M = 10*15*8 = 1200 octets en mémoire.

Initialisation et Réserve Automatique

Initialisation

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

Exemples:

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

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

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

Exemples:

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



C:

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

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



C:

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

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



C:

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

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



ERREUR !



Réserve automatique

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

nécessaires.

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

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

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

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

Exemple:

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



C:

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

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

Accès aux Composantes

Accès à un tableau à deux dimensions

<code><NomTableau>[<Ligne>][<Colonne>]</code>

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

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

ATTENTION!!! Considérons un tableau A de dimensions L et C.

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

A[N-1][M-1]

Affichage et Affectation

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

Affichage du contenu d'un tableau à deux dimensions

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

        /* Retour à la ligne */
        printf("\n");
    }
    return 0;
}
```

Remarques:

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

```
printf("%7d", A[I][J]);
```

Affectation avec des valeurs provenant de l'extérieur

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

6.3 Exercice : Tableaux à deux dimensions

Exercice 1:

Ecrivez un programme qui lit les dimensions L et C d'un tableau T à deux dimensions du type **int** (dimensions maximales: 50 lignes et 50 colonnes).

Remplissez le tableau par des valeurs entrées au clavier et affichez le tableau

Calculez et affichez la somme des éléments du tableau

Calculez et affichez la somme de chaque ligne et de chaque colonne en n'utilisant qu'une variable d'aide pour la somme.

Exercice 2:

Les équipes de foot

1) Créez un type de structure nommée **Joueur** qui stocke les éléments d'un joueur de foot:

- son **nom**, son **prenom**
- son **age**.
- son nombre de sélections **nbSel**.
- le nombre de but marqués au dernier match **derniersButs**.
- le nombre de fois ou il a eu la balle **nbBal**.
- un tableau à deux dimensions **buts** stockant le nombre de but pour chaque mois au cours des trois dernières années.

2) Créez un type de structure nommée **Equipe** qui stocke les éléments suivant:

- le **nom** de l'équipe de foot.
- le nombre total de buts marqués **nbButs**.
- le nombre de fois ou l'équipe a eu la balle **nbBal**.
- l'age moyen de l'équipe **ageMoy**.
- l'age du plus vieux **ageMax**.
- l'age du plus jeune **ageMin**.
- le nombre de Joueurs **nbJoueurs**.
- le tableau des **Joueurs**.

3) Créez une fonction **getPlayer** qui renvoie une structure joueur à l'aide des valeurs entrées au clavier par l'utilisateur (les buts du tableau **buts** seront tirés au hasard entre 0 et 10 à l'aide de la fonction **rand**).

4) Créez une fonction **getButs** qui renvoie le nombre de buts marqués au cours d'un certain mois **m** d'une des années **a** précédentes pour un **joueur** donné.

5) Créez une fonction **getTeam** qui reçoit une **Equipe** et un nombre de joueur et renvoie une structure **Equipe** remplie.

6) Créez un type de structure nommée **Saison** qui stocke les éléments suivant:

- l'année de début de la saison **debut**.
- l'année de fin de la saison **fin**.
- le nombre d'équipes **nbEquipes**.
- le tableau des équipes nommé **equipes**.

7) Créez une fonction **statEquipe** qui compile les données d'une équipe à partir des données des joueurs (la fonction reçoit une **Equipe** et renvoie une **Equipe**).

8) Créez une fonction **afficheJoueur** et une fonction **afficheEquipe**.

9) Créez une fonction **ajouteJoueur** qui permet d'ajouter un **Joueur** à une équipe d'une **Saison** saison. La fonction demandera à l'utilisateur l'index de l'équipe en question.

10) Créez une fonction **statsSaison** qui reçoit une **Saison**, compile les statistiques de chaque équipe, détermine l'équipe qui a gagné la saison de championnat, et renvoie la **Saison** complétée.

11) Modifiez le programme principal pour proposer un menu qui permette de renseigner les équipes d'une saison, ajouter (ou optionnellement supprimer) une **Equipe** ou un joueur dans une équipe, compiler les données de la saison, et d'afficher son contenu et ses résultats. Ajoutez une portion de code qui donne les noms des meilleurs joueurs de la saison au cours des années précédentes et pour l'année en cours.

Pendant la programmation, créez éventuellement les fonctions supplémentaires nécessaire à une bonne structuration de votre programme.

7 Fonctions avancées

Les fonctions avancées.

7.1 Blocs et portée

LA NOTION DE BLOCS ET LA PORTEE DES IDENTIFICATEURS

Les fonctions en C sont définies à l'aide de blocs d'instructions. Un bloc d'instructions est encadré d'accolades et composé de deux parties:

Blocs d'instructions en C

```
{  
<déclarations locales>  
<instructions>  
}
```

Par opposition à d'autres langages de programmation, ceci est vrai pour *tous* les blocs d'instructions, non seulement pour les blocs qui renferment une fonction. Ainsi, le bloc d'instructions d'une commande **if**, **while** ou **for** peut théoriquement contenir des déclarations locales de variables et même de fonctions.

Exemple: La variable d'aide I est déclarée à l'intérieur d'un bloc conditionnel. Si la condition (N>0) n'est pas remplie, I n'est pas défini. A la fin du bloc conditionnel, I disparaît.

```
if (N>0)  
{  
    int I;  
    for (I=0; I<N; I++)  
        ...  
}
```

Variables Locales

Les variables déclarées dans un bloc d'instructions sont *uniquement visibles à l'intérieur de ce bloc*. On dit que ce sont des *variables locales* à ce bloc.

Exemple: La variable NOM est définie localement dans le bloc extérieur de la fonction HELLO. Ainsi, aucune autre fonction n'a accès à la variable NOM:

```
void HELLO(void);  
{  
    char NOM[20];  
    printf("Introduisez votre nom : ");  
    gets(NOM);  
    printf("Bonjour %s !\n", NOM);  
}
```

Exemple: La déclaration de la variable I se trouve à l'intérieur d'un bloc d'instructions conditionnel. Elle n'est pas visible à l'extérieur de ce bloc, ni même dans la fonction qui l'entoure.

```
if (N>0)  
{  
    int I;  
    for (I=0; I<N; I++)  
        ...  
}
```

ATTENTION!!! Une variable déclarée à l'intérieur d'un bloc *cache* toutes les variables du même nom des blocs qui l'entourent.

Exemple: Dans la fonction suivante,

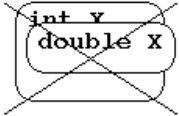
```
int y;  
double x;
```

```

int FONCTION(int A);
{
    int X;
    ...
    X = 100;
    ...
    while (A>10)
    {
        double X;
        ...
        X *= A;
        ...
    }
}

```

la première instruction **x=100** se rapporte à la variable du type **int** déclarée dans le bloc extérieur de la fonction; l'instruction **x*=A** agit sur la variable du type **double** déclarée dans la boucle **while**. A l'intérieur de la boucle, il est impossible d'accéder à la variable X du bloc extérieur.



Ce n'est pas du bon style d'utiliser des noms de variables qui cachent des variables déclarées dans des blocs extérieurs; ceci peut facilement mener à des malentendus et à des erreurs.

La plupart des programmes C ne profitent pas de la possibilité de déclarer des variables ou des fonctions à l'intérieur d'une boucle ou d'un bloc conditionnel. Dans la suite, nous allons faire toutes nos déclarations locales *au début des fonctions*.

Variables Globales

Les variables déclarées au début du fichier, à l'extérieur de toutes les fonctions *sont disponibles à toutes les fonctions du programme*. Ce sont alors des **variables globales**. En général, les variables globales sont déclarées immédiatement derrière les instructions **#include** au début du programme.

ATTENTION!!!

Les variables déclarées au début de la fonction principale **main** ne sont **pas** des variables globales, mais elles sont locales à **main**!

Exemple: La variable STATUS est déclarée globalement pour pouvoir être utilisée dans les procédures A et B.

```

#include <stdio.h>
int STATUS;

void A(...)
{
    ...
    if (STATUS>0)
        STATUS--;
    else
        ...
    ...
}

void B(...)
{
    ...
    STATUS++;
    ...
}

```

Conseils:

- Les variables globales sont à utiliser avec précaution, puisqu'elles créent des *liens invisibles entre les fonctions*. La modularité d'un programme peut en souffrir et le programmeur risque de perdre la vue d'ensemble.
- Il faut faire attention à ne pas cacher involontairement des variables globales par des variables locales du même nom.
- Le codex de la programmation défensive nous conseille *d'écrire nos programmes aussi 'localement' que possible*.

L'utilisation de variables globales devient inévitable, si

- plusieurs fonctions qui ne s'appellent pas ont besoin des mêmes variables, ou

- plusieurs fonctions d'un programme ont besoin du même ensemble de variables. Ce serait alors trop encombrant de passer toutes les variables comme paramètres d'une fonction à l'autre.

7.2 Déclaration et définition des fonctions

En général, le nom d'une fonction apparaît à trois endroits dans un programme:

- lors de la **déclaration**
- lors de la **définition**
- lors de l'**appel**

Exemple: Avant de parler des détails, penchons-nous sur un exemple. Dans le programme suivant, la fonction **main** utilise les deux fonctions:

- **ENTREE** qui lit un nombre entier au clavier et le fournit comme résultat. La fonction ENTREE n'a pas de paramètres.
- **MAX** qui renvoie comme résultat le maximum de deux entiers fournis comme paramètres.

```
#include <stdio.h>

main()
{
    /* Prototypes des fonctions appelées */
    int ENTREE(void);
    int MAX(int N1, int N2);
    /* Déclaration des variables */
    int A, B;
    /* Traitement avec appel des fonctions */
    A = ENTREE();
    B = ENTREE();
    printf("Le maximum est %d\n", MAX(A,B));
}

/* Définition de la fonction ENTREE */
int ENTREE(void)
{
    int NOMBRE;
    printf("Entrez un nombre entier : ");
    scanf("%d", &NOMBRE);
    return NOMBRE;
}

/* Définition de la fonction MAX */
int MAX(int N1, int N2)
{
    if (N1>N2)
        return N1;
    else
        return N2;
}
```

Définition d'une Fonction

Dans la définition d'une fonction, nous indiquons:

- le nom de la fonction
- le type, le nombre et les noms des paramètres de la fonction
- le type du résultat fourni par la fonction
- les données locales à la fonction
- les instructions à exécuter

Définition d'une fonction en C

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

Remarquez qu'il n'y a pas de point-virgule derrière la définition des paramètres de la fonction.

Les Identificateurs: Les noms des paramètres et de la fonction sont des identificateurs qui doivent correspondre aux restrictions définies auparavant. Des noms bien choisis peuvent fournir une information utile sur leur rôle. Ainsi, les identificateurs font aussi partie de la documentation d'un programme.

ATTENTION!!! Si nous choisissons un nom de fonction qui existe déjà dans une bibliothèque, notre fonction **cache** la fonction prédéfinie.

Type d'une Fonction: Si une fonction F fournit un résultat du type T, on dit que *'la fonction F est du type T'* ou que *'la fonction F a le type T'*.

Exemple: La fonction MAX est du type **int** et elle a besoin de deux paramètres du type **int**. Le résultat de la fonction MAX peut être intégré dans d'autres expressions.

```
int MAX(int N1, int N2)
{
    if (N1>N2)
        return N1;
    else
        return N2;
}
```

Exemple: La fonction PI fournit un résultat rationnel du type **float**. La liste des paramètres de PI est déclarée comme **void** (vide), c.-à-d. PI n'a pas besoin de paramètres et il faut l'appeler par: **PI()**

```
float PI(void)
{
    return 3.1415927;
}
```

Remarques: Une fonction peut fournir comme résultat:

- un type arithmétique,
- une structure (définie par **struct** - pas traité dans ce cours),
- une réunion (définie par **union** - pas traité dans ce cours),
- un pointeur,
- **void** (la fonction correspond alors à une 'procédure').

Une fonction **ne peut pas** fournir comme résultat

- des tableaux,
- des chaînes de caractères
- ou des fonctions.

(**Attention:** Il est cependant possible de renvoyer un pointeur sur le premier élément d'un tableau ou d'une chaîne de caractères.)

- Si une fonction ne fournit pas de résultat, il faut indiquer **void** (vide) comme type du résultat.
- Si une fonction n'a pas de paramètres, on peut déclarer la liste des paramètres comme (**void**) ou simplement comme **()**.
- Le type par défaut est **int**; autrement dit: si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type **int**.
- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).
- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée ou définie avant d'être appelée.

Rappel: La fonction principale **main** est du type **int**. Elle est exécutée automatiquement lors de l'appel du programme. A la place de la définition:

```
int main(void)
```

on peut écrire simplement:

```
main()
```

Déclaration d'une Fonction

En C, il faut déclarer chaque fonction avant de pouvoir l'utiliser. La déclaration informe le compilateur du type des paramètres et du résultat de la fonction. A l'aide de ces données, le compilateur peut contrôler si le nombre et le type des paramètres d'une fonction sont corrects. Si dans le texte du programme la fonction est définie avant son premier appel, elle n'a pas besoin d'être déclarée.

Prototype d'une Fonction: La déclaration d'une fonction se fait par un *prototype* de la fonction qui indique uniquement le type des données transmises et reçues par la fonction.

Déclaration de Prototype d'une Fonction

```
<TypeRés> <NomFonct> (<TypePar1>, <TypePar2>, ...);
```

ou bien

```
<TypeRés> <NomFonct> (<TypePar1> <NomPar1>, <TypePar2> <NomPar2>, ... );
```

ATTENTION!!! Lors de la *déclaration*, le nombre et le type des paramètres doivent nécessairement correspondre à ceux de la *définition* de la fonction.

Noms des Paramètres: On peut facultativement inclure les *noms des paramètres* dans la déclaration, mais ils ne sont pas considérés par le compilateur. Les noms fournissent pourtant une information intéressante pour le programmeur qui peut en déduire le rôle des différents paramètres.

Conseil pratique: Il est d'usage de copier (à l'aide de Edit - Copy & Paste) la première ligne de la définition de la fonction comme déclaration. (N'oubliez pas d'ajouter un point-virgule à la fin de la déclaration !)

Règles pour la déclaration des fonctions: De façon analogue aux déclarations de variables, nous pouvons déclarer une fonction localement ou globalement. La définition des fonctions joue un rôle spécial pour la déclaration. En résumé, nous allons considérer les règles suivantes:

Déclaration locale:

Une fonction peut être déclarée localement *dans la fonction qui l'appelle* (avant la déclaration des variables). Elle est alors disponible à cette fonction.

Déclaration globale:

Une fonction peut être déclarée globalement *au début du programme* (derrière les instructions **#include**). Elle est alors disponible à toutes les fonctions du programme.

Déclaration implicite par la définition:

La fonction est automatiquement disponible à toutes les fonctions qui suivent sa définition.

Déclaration multiple:

Une fonction peut être déclarée *plusieurs fois* dans le texte d'un programme, mais les indications doivent concorder.

Fonction main:

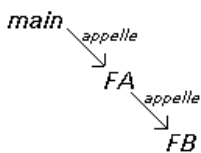
La fonction principale **main** n'a pas besoin d'être déclarée.

Discussion d'un Exemple

Considérons la situation suivante:

- La fonction **main** appelle la fonction FA.
- La fonction FA appelle la fonction FB.

Nous obtenons donc la hiérarchie suivante:



Il y a beaucoup de possibilités de déclarer et de définir ces fonctions. Nous allons retenir trois variantes qui suivent une logique conséquente:

a) Déclarations locales des fonctions et définition 'top-down'

La définition 'top-down' suit la hiérarchie des fonctions:

Nous commençons par la définition de la fonction principale **main**, suivie des sous-programmes FA et FB. Nous devons déclarer explicitement FA et FB car leurs définitions suivent leurs appels.

```

/* Définition de main */
main()
{
    /* Déclaration locale de FA */
    int FA (int X, int Y);
    ...
    /* Appel de FA */
    I = FA(2, 3);
    ...
}

/* Définition de FA */
int FA(int X, int Y)
{
    /* Déclaration locale de FB */
    int FB (int N, int M);
    ...
    /* Appel de FB */
    J = FB(20, 30);
    ...
}

/* Définition de FB */
int FB(int N, int M)
{
    ...
}

```

Cet ordre de définition a l'avantage de refléter la hiérarchie des fonctions: Ainsi l'utilisateur qui ne s'intéresse qu'à la solution globale du problème n'a qu'à lire le début du fichier. Pour retrouver les détails d'une implémentation, il peut passer du haut vers le bas dans le fichier. Sur ce chemin, il retrouve toutes les dépendances des fonctions simplement en se référant aux déclarations locales. S'il existe beaucoup de dépendances dans un programme, le nombre des déclarations locales peut quand même s'accroître dans des dimensions insoutenables.

b) Définition 'bottom-up' sans déclarations

La définition 'bottom-up' commence en bas de la hiérarchie:

La fonction **main** se trouve à la fin du fichier. Les fonctions qui traitent les détails du problème sont définies en premier lieu.

```

/* Définition de FB */
int FB(int N, int M)
{
    ...
}

/* Définition de FA */
int FA(int X, int Y)
{
    ...
    /* Appel de FB */
    J = FB(20, 30);
    ...
}

/* Définition de main */
main()
{
    ...
    /* Appel de FA */
    I = FA(2, 3);
    ...
}

```

Comme les fonctions sont définies avant leur appel, les déclarations peuvent être laissées de côté. Ceci allège un peu le texte du programme, mais il est beaucoup plus difficile de retrouver les dépendances entre les fonctions.

c) Déclaration globale des fonctions et définition 'top-down'

En déclarant toutes les fonctions globalement au début du texte du programme, nous ne sommes pas forcés de nous occuper de la dépendance entre les fonctions. Cette solution est la plus simple et la plus sûre pour des programmes complexes contenant une grande quantité de dépendances. Il est quand même recommandé de définir les fonctions selon l'ordre de leur hiérarchie:

```

/* Déclaration globale de FA et FB */
int FA (int X, int Y);
int FB (int N, int M);

/* Définition de main */
main()
{
    ...
    /* Appel de FA */
    I = FA(2, 3);
    ...
}

/* Définition de FA */
int FA(int X, int Y)
{
    ...
    /* Appel de FB */
    J = FB(20, 30);
    ...
}

/* Définition de FB */
int FB(int N, int M)
{
    ...
}

```

d) Conclusions

Dans la suite, nous allons utiliser l'ordre de définition 'top-down' qui reflète le mieux la structure d'un programme. Comme nos programmes ne contiennent pas beaucoup de dépendances, nous allons déclarer les fonctions localement dans les fonctions appelantes.

7.3 Renvoyer un résultat

Par définition, toutes les fonctions fournissent un résultat d'un type que nous devons déclarer. Une fonction peut renvoyer une valeur d'un type simple ou l'adresse d'une variable ou d'un tableau.

Pour fournir un résultat en quittant une fonction, nous disposons de la commande **return**:

La Commande Return

L'instruction

```
return <expression>;
```

a les effets suivants:

- *évaluation de l'<expression>*
- *conversion automatique du résultat de l'expression dans le type de la fonction*
- *renvoi du résultat*
- *terminaison de la fonction*

Exemples: La fonction CARRE du type **double** calcule et fournit comme résultat le carré d'un réel fourni comme paramètre.

```
double CARRE(double X)
{
    return X*X;
}
```

Nous pouvons définir nous-mêmes une fonction TAN qui calcule la tangente d'un réel X à l'aide des fonctions **sin** et de **cos** de la bibliothèque *<math>*.

```
#include <math.h>

double TAN(double X)
{
    if (cos(X) != 0)
        return sin(X)/cos(X);
    else
        printf("Erreur !\n");
}
```

Si nous supposons les déclarations suivantes,

```
double X, COT;
```

les appels des fonctions CARRE et TAN peuvent être intégrés dans des calculs ou des expressions:

```
printf("Le carre de %f est %f \n", X, CARRE(X));
printf("La tangente de %f est %f \n", X, TAN(X));
COT = 1/TAN(X);
```

La Commande Void

En C, il n'existe pas de structure spéciale pour la définition de *procédures* comme en Pascal et en langage algorithmique. Nous pouvons cependant employer une fonction du type **void** partout où nous utiliserions une procédure en langage algorithmique ou en Pascal.

Exemple: La procédure LIGNE affiche L étoiles dans une ligne, nous utilisons une fonction du type **void**:

```
void LIGNE(int L)
{
    /* Déclarations des variables locales */
    int I;
    /* Traitements */
    for (I=0; I<L; I++)
        printf("*");
    printf("\n");
}
```

La Fonction Main

Dans nos exemples, la fonction **main** n'a pas de paramètres et est toujours du type **int**. Typiquement, les programmes renvoient la valeur zéro comme code d'erreur s'ils se terminent avec succès. Des valeurs différentes de zéro indiquent un arrêt fautif ou anormal.

En MS-DOS, le code d'erreur retourné par un programme peut être contrôlé à l'aide de la commande IF ERRORLEVEL ...

Remarque avancée: Si nous quittons une fonction (d'un type différent de **void**) sans renvoyer de résultat à l'aide de **return**, la valeur transmise à la fonction appelante est indéfinie. Le résultat d'une telle action est imprévisible. Si une erreur fatale s'est produite à l'intérieur d'une fonction, il est conseillé d'interrompre l'exécution de tout le programme et de renvoyer un code erreur différent de zéro à l'environnement pour indiquer que le programme ne s'est pas terminé normalement.

Vu sous cet angle, il est dangereux de déclarer la fonction TAN comme nous l'avons fait plus haut: Le cas d'une division par zéro, est bien intercepté et reporté par un message d'erreur, mais l'exécution du programme continue 'normalement' avec des valeurs incorrectes.

La Fonction Exit

Pour remédier à ce dilemme, nous pouvons utiliser la fonction **exit** qui est définie dans la bibliothèque `<stdlib>`. **exit** nous permet d'interrompre l'exécution du programme en fournissant un code d'erreur à l'environnement. Pour pouvoir localiser l'erreur à l'intérieur du programme, il est avantageux d'afficher un message d'erreur qui indique la nature de l'erreur et la fonction dans laquelle elle s'est produite.

Une version plus solide de TAN se présenterait comme suit:

```
#include <math.h>

double TAN(double X)
{
    if (cos(X) != 0)
        return sin(X)/cos(X);
    else
    {
        printf("\aFonction TAN:\n"
               "Erreur: Division par zéro !\n");
        exit(-1); /* Code erreur -1 */
    }
}
```

Ignorer le Résultat

Lors de l'appel d'une fonction, l'utilisateur est libre d'accepter le résultat d'une fonction ou de l'ignorer.

Exemple: La fonction **scanf** renvoie le nombre de données correctement reçues comme résultat. En général, nous avons ignoré ce fait:

```
int JOUR, MOIS, ANNEE;
printf("Entrez la date actuelle : ");
scanf("%d %d %d", &JOUR, &MOIS, &ANNEE);
```

Nous pouvons utiliser le résultat de **scanf** comme contrôle:

```
int JOUR, MOIS, ANNEE;
int RES;
do
{
    printf("Entrez la date actuelle : ");
    RES = scanf("%d %d %d", &JOUR,&MOIS,&ANNEE);
}
while (RES != 3);
```

7.4 Paramètres des fonctions

Les *paramètres* ou *arguments* sont les 'boîtes aux lettres' d'une fonction. Elles acceptent les données de l'extérieur et déterminent les actions et le résultat de la fonction. Techniquement, nous pouvons résumer le rôle des paramètres en C de la façon suivante:

*Les paramètres d'une fonction sont simplement des **variables locales** qui sont initialisées par les **valeurs** obtenues lors de l'appel.*

Généralités

Conversion automatique: Lors d'un appel, le nombre et l'ordre des paramètres doivent nécessairement correspondre aux indications de la déclaration de la fonction. Les paramètres sont automatiquement convertis dans les types de la déclaration avant d'être passés à la fonction.

Exemple : Le prototype de la fonction **pow** (bibliothèque *<math></i>*) est déclaré comme suit:

```
double pow (double, double);
```

Au cours des instructions,

```
int A, B;  
...  
A = pow (B, 2);
```

nous assistons à trois conversions automatiques:

- Avant d'être transmis à la fonction, la valeur de B est convertie en **double**;
- la valeur 2 est convertie en 2.0 .
- Comme **pow** est du type **double**, le résultat de la fonction doit être converti en **int** avant d'être affecté à A.

Void: Evidemment, il existe aussi des fonctions qui fournissent leurs résultats ou exécutent une action sans avoir besoin de données. La liste des paramètres contient alors la déclaration **void** ou elle reste vide (P.ex.: **double PI(void)** ou **int ENTREE()**).

Passage des Paramètres par Valeur

En C, le passage des paramètres se fait toujours par la valeur, c.-à-d. les fonctions n'obtiennent que les *valeurs* de leurs paramètres et n'ont pas d'accès aux variables elles-mêmes.

Les paramètres d'une fonction sont à considérer comme des *variables locales* qui sont initialisées automatiquement par les valeurs indiquées lors d'un appel.

A l'intérieur de la fonction, nous pouvons donc changer les valeurs des paramètres sans influencer les valeurs originales dans les fonctions appelantes.

Exemple: La fonction ETOILES dessine une ligne de N étoiles. Le paramètre N est modifié à l'intérieur de la fonction:

```
void ETOILES(int N)  
{  
    while (N>0)  
    {  
        printf("*");  
        N--;  
    }  
    printf("\n");  
}
```

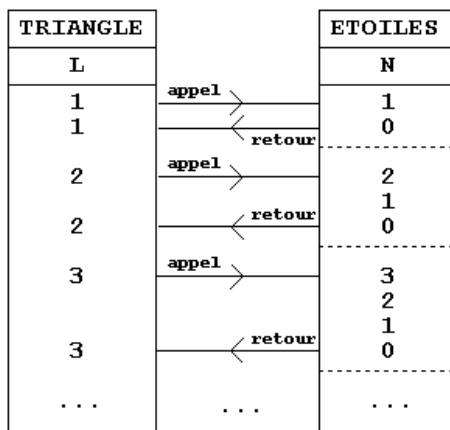
En utilisant N comme compteur, nous n'avons pas besoin de l'indice d'aide I comme dans la fonction LIGNES définie plus haut.

La fonction TRIANGLE, appelle la fonction ETOILES en utilisant la variable L comme paramètre:

```
void TRIANGLE(void)  
{  
    int L;  
    for (L=1; L<10; L++)  
        ETOILES(L);  
}
```

Au moment de l'appel, la *valeur* de L est copiée dans N. La variable N peut donc être décrémentée à l'intérieur de ETOILES, sans influencer la valeur originale de L.

Schématiquement, le passage des paramètres peut être représenté dans une 'grille' des valeurs:



Avantages: Le passage par *valeur* a l'avantage que nous pouvons utiliser les paramètres comme des variables locales bien initialisées. De cette façon, nous avons besoin de moins de variables d'aide.

Passage de l'Adresse d'une Variable

Comme nous l'avons constaté ci-dessus, une fonction n'obtient que les valeurs de ses paramètres.

ATTENTION!!! Pour changer la valeur d'une variable de la fonction appelante, nous allons procéder comme suit:

- la fonction appelante doit **fournir l'adresse de la variable** et
- la fonction appelée doit **déclarer le paramètre comme pointeur**.

On peut alors atteindre la variable à l'aide du pointeur.

Discussion d'un exemple: Nous voulons écrire une fonction PERMUTER qui échange le contenu de deux variables du type **int**. En première approche, nous écrivons la fonction suivante:

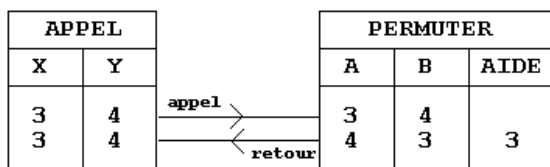
```
void PERMUTER (int A, int B)
{
    int AIDE;
    AIDE = A;
    A = B;
    B = AIDE;
}
```

Nous appelons la fonction pour deux variables X et Y par:

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

Résultat: X et Y restent inchangés !

Explication: Lors de l'appel, les *valeurs* de X et de Y sont copiées dans les paramètres A et B. PERMUTER échange bien contenu des variables *locales* A et B, mais les valeurs de X et Y restent les mêmes.



Pour pouvoir modifier le contenu de X et de Y, la fonction PERMUTER a besoin des adresses de X et Y. En utilisant des pointeurs, nous écrivons une deuxième fonction:

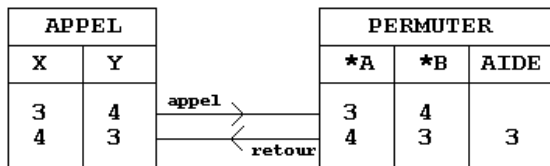
```
void PERMUTER (int *A, int *B)
{
    int AIDE;
    AIDE = *A;
    *A = *B;
    *B = AIDE;
}
```

Nous appelons la fonction par:

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

Résultat: Le contenu des variables X et Y est échangé !

Explication: Lors de l'appel, les *adresses* de X et de Y sont copiées dans les *pointeurs* A et B. PERMUTER échange ensuite le contenu des adresses indiquées par les pointeurs A et B.



Remarque: Dans les grilles, nous allons marquer les paramètres acceptant une adresse comme pointeurs et indiquer le contenu de ces adresses.

Passage de l'Adresse d'un Tableau à une Dimension

Méthode: Comme il est impossible de passer 'la valeur' de tout un tableau à une fonction, on fournit *l'adresse d'un élément du tableau*. En général, on fournit l'adresse du premier élément du tableau, qui est donnée par *le nom du tableau*.

Déclaration: Dans la liste des paramètres d'une fonction, on peut déclarer un tableau par le nom suivi de crochets,

```
<type> <nom>[]
```

ou simplement par un pointeur sur le type des éléments du tableau:

```
<type> *<nom>
```

Exemple: La fonction **strlen** calcule et retourne la longueur d'une chaîne de caractères fournie comme paramètre:

```
int strlen(char *S)
{
    int N;
    for (N=0; *S != '\0'; S++)
        N++;
    return N;
}
```

A la place de la déclaration de la chaîne comme

```
char *S
```

on aurait aussi pu indiquer

```
char S[]
```

comme nous l'avons fait dans l'exemple d'introduction. Dans la suite, nous allons utiliser la première notation pour mettre en évidence que le paramètre est un *pointeur variable* que nous pouvons modifier à l'intérieur de la fonction.

Appel: Lors d'un appel, l'adresse d'un tableau peut être donnée par le nom du tableau, par un pointeur ou par l'adresse d'un élément quelconque du tableau.

Exemple: Après les instructions,

```
char CH[] = "Bonjour !";
char *P;
P = CH;
```

nous pouvons appeler la fonction **strlen** définie ci-dessus par:

```
strlen(CH)      /* résultat: 9 */
strlen(P)       /* résultat: 9 */
strlen(&CH[4])  /* résultat: 5 */
strlen(P+2)     /* résultat: 7 */
strlen(CH+2)    /* résultat: 7 */
```

ATTENTION!!! Dans les trois derniers appels, nous voyons qu'il est possible de fournir *une partie* d'un tableau à une fonction, en utilisant l'adresse d'un élément à l'intérieur de tableau comme paramètre.

Remarque pratique: Pour qu'une fonction puisse travailler correctement avec un tableau qui n'est pas du type **char**, il faut aussi fournir la dimension du tableau ou le nombre d'éléments à traiter comme paramètre, sinon la fonction risque de sortir du domaine du tableau.

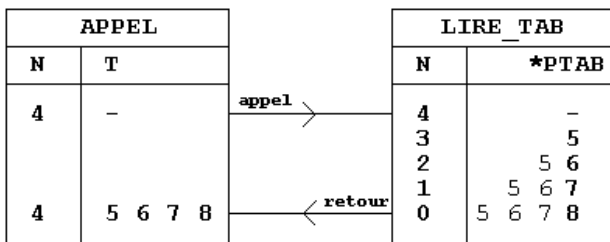
Exemple: La fonction LIRETAB lit N données pour un tableau (unidimensionnel) du type **int** et les mémorise à partir de l'adresse indiquée par le pointeur PTAB. PTAB et N sont fournis comme paramètres.

```
void LIRE_TAB(int N, int *PTAB)
{
    printf("Entrez %d valeurs : \n", N);
    while(N)
    {
        scanf("%d", PTAB++);
        N--;
    }
}
```

Dans l'appel de la fonction nous utilisons en général le nom du tableau:

```
LIRE_TAB(4, T);
```

Nous obtenons alors les grilles suivantes:



Passage de l'Adresse d'un Tableau à Deux Dimensions

Exemple: Imaginons que nous voulons écrire une fonction qui calcule la somme de tous les éléments d'une matrice de réels A dont nous fournissons les deux dimensions N et M comme paramètres.

Problème: Comment pouvons-nous passer l'adresse de la matrice à la fonction ?

Par analogie avec ce que nous avons vu au chapitre précédent, nous pourrions envisager de déclarer le tableau concerné dans l'en-tête de la fonction sous la forme `A[][]`. Dans le cas d'un tableau à deux dimensions, cette méthode ne fournit pas assez de données, parce que le compilateur a besoin de la deuxième dimension du tableau pour déterminer l'adresse d'un élément `A[i][j]`.

Une solution praticable consiste à faire en sorte que la fonction reçoive un pointeur (de type `float*`) sur le début de la matrice et de parcourir tous les éléments comme s'il s'agissait d'un tableau à une dimension `N*M`.

Cela nous conduit à cette fonction:

```
float SOMME(float *A, int N, int M)
{
    int I;
    float S;
    for (I=0; I<N*M; I++)
        S += A[I];
    return S;
}
```

Lors d'un appel de cette fonction, la seule difficulté consiste à transmettre l'adresse du début du tableau sous forme d'un pointeur sur **float**. Prenons par exemple un tableau déclaré par

```
float A[3][4];
```

Le nom A correspond à la bonne adresse, mais cette adresse est du type "*pointeur sur un tableau de 4 éléments du type float*". Si notre fonction est correctement déclarée, le compilateur la convertira automatiquement dans une adresse du type '*pointeur sur float*'.

Toutefois, comme nous l'avons déjà remarqué auparavant, on gagne en lisibilité et on évite d'éventuels messages d'avertissement si on utilise l'opérateur de conversion forcée ("*cast*").

Solution: Voici finalement un programme faisant appel à notre fonction SOMME:

```
#include <stdio.h>
main()
{
    /* Prototype de la fonction SOMME */
    float SOMME(float *A, int N, int M);
    /* Déclaration de la matrice */
    float T[3][4] = {{1, 2, 3, 4},
                     {5, 6, 7, 8},
                     {9,10,11,12}};
    /* Appel de la fonction SOMME */
    printf("Somme des éléments : %f \n",
           SOMME((float*)T, 3, 4) );
    return 0;
}
```

Rappel: Rappelons encore une fois que lors de l'interprétation d'un tableau à deux dimensions comme tableau unidimensionnel, il faut calculer les adresses des composantes à l'aide du *nombre de colonnes maximal réservé lors de la déclaration*.

8 Pointeurs

Les pointeurs

8.1 Les pointeurs

La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de **pointeurs**, c.-à-d. à l'aide de variables auxquelles on peut attribuer les **adresses d'autres variables**.

En C, les pointeurs jouent un rôle primordial dans la définition de fonctions: Comme le passage des paramètres en C se fait toujours par la valeur, les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions. Ainsi le traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs.

En outre, les pointeurs nous permettent d'écrire des programmes plus compacts et plus efficaces et fournissent souvent la seule solution raisonnable à un problème. Ainsi, la majorité des applications écrites en C profitent extensivement des pointeurs.

Le revers de la médaille est très bien formulé par Kernighan & Ritchie dans leur livre 'Programming in C':

" ... Les pointeurs étaient mis dans le même sac que l'instruction **goto** comme une excellente technique de formuler des programmes incompréhensibles. Ceci est certainement vrai si les pointeurs sont employés négligemment, et on peut facilement créer des pointeurs qui pointent 'n'importe où'. Avec une certaine discipline, les pointeurs peuvent aussi être utilisés pour programmer de façon claire et simple. C'est précisément cet aspect que nous voulons faire ressortir dans la suite. ... "

ADRESSAGE DE VARIABLES

Avant de parler de pointeurs, il est indiqué de brièvement passer en revue les deux modes d'adressage principaux, qui vont d'ailleurs nous accompagner tout au long des chapitres suivants.

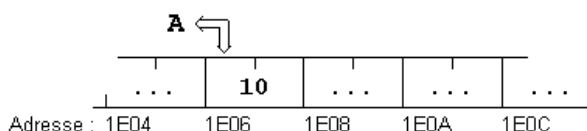
Adressage Direct

Dans la programmation, nous utilisons des variables pour stocker des informations. La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéder *directement* à cette valeur.

Adressage direct: Accès au contenu d'une variable par le nom de la variable.

Exemple:

```
short A;  
A = 10;
```



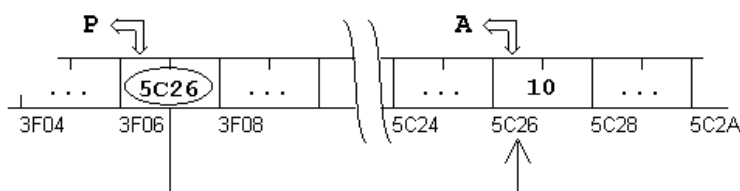
Adressage indirect

Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée **pointeur**. Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.

Adressage indirect: Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

Exemple

Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit:



Définition: Pointeur

Un **pointeur** est une variable spéciale qui peut contenir l'adresse d'une autre variable.

En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type.

Si un pointeur P contient l'adresse d'une variable A, on dit que

'P pointe sur A'.

Remarque

Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence:

* Un **pointeur** est une variable qui peut 'pointer' sur différentes adresses.

* Le **nom d'une variable** reste toujours lié à la même adresse.

Les opérateurs de base

Lors du travail avec des pointeurs, nous avons besoin



- d'un opérateur 'adresse de': **&**
pour obtenir l'adresse d'une variable.
- d'un opérateur 'contenu de': *****
pour accéder au contenu d'une adresse.
- d'une syntaxe de déclaration
pour pouvoir déclarer un pointeur.

L'opérateur 'adresse de' : &

```
&<NomVariable>
fournit l'adresse de la variable <NomVariable>
```

L'opérateur **&** nous est déjà familier par la fonction **scanf**, qui a besoin de l'adresse de ses arguments pour pouvoir leur attribuer de nouvelles valeurs.

Exemple

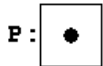
```
int N;
printf("Entrez un nombre entier : ");
scanf("%d", &N);
```

Attention !

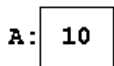
L'opérateur **&** peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c.-à-d. à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

Représentation schématique

Soit P un pointeur non initialisé



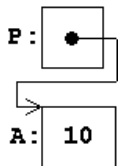
et A une variable (du même type) contenant la valeur 10 :



Alors l'instruction

```
P = &A;
```

affecte l'adresse de la variable A à la variable P. En mémoire, A et P se présentent comme dans le graphique à la fin du chapitre 9.1.2. Dans notre représentation schématique, nous pouvons illustrer le fait que 'P pointe sur A' par une flèche:

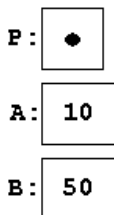


*L'opérateur 'contenu de' : **

```
*<NomPointeur>
désigne le contenu de l'adresse référencée par le pointeur <NomPointeur>
```

Exemple

Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur non initialisé:

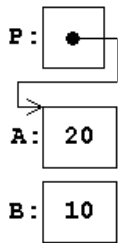


Après les instructions,

```
P = &A;
B = *P;
*P = 20;
```

- P pointe sur A,
- le contenu de A (référéncé par *P) est affecté à B, et

- le contenu de A (référéncé par *P) est mis à 20.



Déclaration d'un pointeur

`<Type> *<NomPointeur>`
déclare un pointeur `<NomPointeur>` qui peut recevoir des adresses de variables du type `<Type>`

Une déclaration comme

```
int *PNUM;
```

peut être interprétée comme suit:

*"*PNUM est du type **int**"*

ou

*"PNUM est un pointeur sur **int**"*

ou

*"PNUM peut contenir l'adresse d'une variable du type **int**"*

Exemple

Le programme complet effectuant les transformations de l'exemple ci-dessus peut se présenter comme suit:

<code> main()</code>	ou bien <code> main()</code>
<code> {</code>	<code> {</code>
<code> /* déclarations */</code>	<code> /* déclarations */</code>
<code> short A = 10;</code>	<code> short A, B, *P;</code>
<code> short B = 50;</code>	<code> /* traitement */</code>
<code> short *P;</code>	<code> A = 10;</code>
<code> /* traitement */</code>	<code> B = 50;</code>
<code> P = &A;</code>	<code> P = &A;</code>
<code> B = *P;</code>	<code> B = *P;</code>
<code> *P = 20;</code>	<code> *P = 20;</code>
<code> return 0;</code>	<code> return 0;</code>
<code> }</code>	<code> }</code>

Remarque

Lors de la déclaration d'un pointeur en C, ce pointeur est lié explicitement à un type de données. Ainsi, la variable PNUM déclarée comme pointeur sur **int** ne peut pas recevoir l'adresse d'une variable d'un autre type que **int**.

Nous allons voir que la limitation d'un pointeur à un type de variables n'élimine pas seulement un grand nombre de sources d'erreurs très désagréables, mais permet une série d'opérations très pratiques sur les pointeurs.

Les opérations élémentaires sur pointeurs

En travaillant avec des pointeurs, nous devons observer les règles suivantes:

Priorité de * et &

* Les opérateurs * et & ont la même priorité que les autres opérateurs unaires (la négation !, l'incrément ++, la décrémentation --). Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

* Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

Exemple

Après l'instruction

```
P = &X;
```

les expressions suivantes, sont équivalentes:

`Y = *P+1` \Leftrightarrow `Y = X+1`

`*P = *P+10` \Leftrightarrow `X = X+10`

`*P += 2` \Leftrightarrow `X += 2`
`++*P` \Leftrightarrow `++X`
`(*P)++` \Leftrightarrow `X++`

Dans le dernier cas, les parenthèses sont nécessaires:

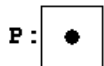
Comme les opérateurs unaires `*` et `++` sont évalués *de droite à gauche*, sans les parenthèses le *pointeur* `P` serait incrémenté, *non pas l'objet* sur lequel `P` pointe.



On peut uniquement affecter des adresses à un pointeur.

Le pointeur NUL

Seule exception: La valeur numérique 0 (zéro) est utilisée pour indiquer qu'un pointeur ne pointe 'nulle part'.



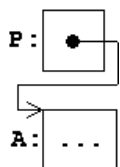
```
int *P;  
P = 0;
```

Finalement, les pointeurs sont aussi des variables et peuvent être utilisés comme telles. Soit `P1` et `P2` deux pointeurs sur **int**, alors l'affectation

```
P1 = P2;
```

copie le contenu de `P2` vers `P1`. `P1` pointe alors sur le même objet que `P2`.

Résumons:



Après les instructions:

```
int A;  
int *P;  
P = &A;
```

A désigne le contenu de `A`

&A désigne l'adresse de `A`

P désigne l'adresse de `A`

***P** désigne le contenu de `A`

En outre:

&P désigne l'adresse du pointeur `P`

***A** est illégal (puisque `A` n'est pas un pointeur)

8.2 Exercices

Exo 7.1

Ecrire un programme qui calcule la factorielle (faire une fonction) d'un nombre A entré par l'utilisateur à l'aide d'une boucle. Modifiez ce programme pour manipuler A grâce à un pointeur pA, et incrémentez A. Refaites une deuxième fonction qui calcule la même chose, mais en utilisant la récursivité. Testez la.

Exo 7.2

- Ecrire un programme qui récupère un entier dans la variable A, puis lance une fonction pour élever au carré la variable A. Par exemple si la variable vaut 5, elle doit ensuite valoir 25
- Modifiez ce programme pour que le paramètre passé en argument de la fonction soit l'adresse de la variable dont il faut calculer le carré (le paramètre est donc un pointeur).

Exo 7.3

- Ecrire un programme qui crée 5 variables A, B, C, D, E, puis affecte leurs adresses dans un tableau de pointeur *pTab[5].
- Donnez des valeurs à ces 5 variables (si ce n'est déjà fait), et élevez les au carré grâce au tableau de pointeurs.

9 Pointeurs et tableaux

En C, il existe une relation très étroite entre tableaux et pointeurs. Ainsi, chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs. En général, les versions formulées avec des pointeurs sont plus compactes et plus efficaces, surtout à l'intérieur de fonctions. Mais, du moins pour des débutants, le 'formalisme pointeur' est un peu inhabituel.

9.1 Adressage des composantes d'un tableau

Comme nous l'avons déjà constaté au chapitre 7, le nom d'un tableau représente l'adresse de son premier élément. En d'autres termes :

`&tableau[0]` et `tableau`

sont une seule et même adresse.

En simplifiant, nous pouvons retenir que *le nom d'un tableau est un **pointeur constant** sur le premier élément du tableau.*

Exemple

En déclarant un tableau A de type **int** et un pointeur P sur **int**,

```
int A[10];
int *P;
```

l'instruction :

`P = A;` est équivalente à `P = &A[0];`



Si P pointe sur une composante quelconque d'un tableau, alors P+1 pointe sur la composante suivante. Plus généralement,

P+i pointe sur la i-ième composante derrière P et

P-i pointe sur la i-ième composante devant P.

Ainsi, après l'instruction,

```
P = A;
```

le pointeur P pointe sur A[0], et

***(P+1)** désigne le contenu de A[1]

***(P+2)** désigne le contenu de A[2]

... ..

***(P+i)** désigne le contenu de A[i]

Remarque

Au premier coup d'oeil, il est bien surprenant que P+i n'adresse pas le i-ième *octet* derrière P, mais la i-ième *composante* derrière P ...

Ceci s'explique par la stratégie de programmation 'défensive' des créateurs du langage C :

Si on travaille avec des pointeurs, les erreurs les plus perfides sont causées par des pointeurs mal placés et des adresses mal calculées. En C, le compilateur peut calculer automatiquement l'adresse de l'élément P+i en ajoutant à P la grandeur d'une composante multipliée par i. Ceci est possible, parce que :

- chaque pointeur est limité à un seul type de données, et
- le compilateur connaît le nombre d'octets des différents types.

Exemple

Soit A un tableau contenant des éléments du type **float** et P un pointeur sur **float** :

```
float A[20], x;
float *P;
```

Après les instructions,

```
P = A;
x = *(P+9);
```

x contient la valeur du 10-ième élément de A, (c.-à-d. celle de A[9]). Une donnée du type **float** ayant besoin de 4 octets, le compilateur obtient l'adresse P+9 en ajoutant $9 * 4 = 36$ octets à l'adresse dans P.

Rassemblons les constatations ci dessus :

Comme A représente l'adresse de A[0],

***(A+1)** désigne le contenu de A[1]



***(A+2)** désigne le contenu de A[2]

...

***(A+i)** désigne le contenu de A[i]

Attention !

Il existe toujours une différence essentielle entre un pointeur et le nom d'un tableau:

- Un *pointeur* est une variable,
donc des opérations comme **P = A** ou **P++** sont permises.

- Le *nom d'un tableau* est une constante,
donc des opérations comme **A = P** ou **A++** sont impossibles.

Ceci nous permet de jeter un petit coup d'oeil derrière les rideaux:

Lors de la première phase de la compilation, toutes les expressions de la forme A[i] sont traduites en *(A+i). En multipliant l'indice i par la grandeur d'une composante, on obtient un indice en octets:

$$\langle \text{indice en octets} \rangle = \langle \text{indice élément} \rangle * \langle \text{grandeur élément} \rangle$$

Cet indice est ajouté à l'adresse du premier élément du tableau pour obtenir l'adresse de la composante i du tableau. Pour le calcul d'une adresse donnée par une adresse plus un indice en octets, on utilise un mode d'adressage spécial connu sous le nom '*adressage indexé*':

$$\langle \text{adresse indexée} \rangle = \langle \text{adresse} \rangle + \langle \text{indice en octets} \rangle$$

Presque tous les processeurs disposent de plusieurs registres spéciaux (*registres index*) à l'aide desquels on peut effectuer l'adressage indexé de façon très efficace.

Résumons Soit un tableau A d'un type quelconque et i un indice pour les composantes de A, alors

A désigne l'adresse de **A[0]**

A+i désigne l'adresse de **A[i]**

***(A+i)** désigne le contenu de **A[i]**

Si P = A, alors

P pointe sur l'élément **A[0]**

P+i pointe sur l'élément **A[i]**

***(P+i)** désigne le contenu de **A[i]**

Formalisme tableau et formalisme pointeur

A l'aide de ce bagage, il nous est facile de 'traduire' un programme écrit à l'aide du '*formalisme tableau*' dans un programme employant le '*formalisme pointeur*'.

Exemple

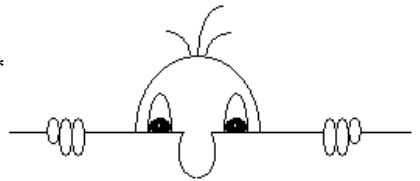
Les deux programmes suivants copient les éléments positifs d'un tableau T dans un deuxième tableau POS.

Formalisme tableau

```
main()
{
    int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
    int POS[10];
    int I,J; /* indices courants dans T et POS */
    for (J=0,I=0 ; I<10 ; I++)
        if (T[I]>0)
        {
            POS[J] = T[I];
            J++;
        }
    return 0;
}
```

Nous pouvons remplacer systématiquement la notation **tableau[I]** par ***(tableau + I)**, ce qui conduit à ce programme:

Formalisme pointeur



```

main()
{
    int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
    int POS[10];
    int I,J; /* indices courants dans T et POS */
    for (J=0,I=0 ; I<10 ; I++)
        if (*(T+I)>0)
        {
            *(POS+J) = *(T+I);
            J++;
        }
    return 0;
}

```

Sources d'erreurs

Un bon nombre d'erreurs lors de l'utilisation de C provient de la confusion entre soit contenu et adresse, soit pointeur et variable. Revoyons donc les trois types de déclarations que nous connaissons jusqu'ici et résumons les possibilités d'accès aux données qui se présentent.

Les variables et leur utilisation `int A;`

déclare une *variable simple* du type **int**

A désigne le contenu de *A*

&A désigne l'adresse de *A*

`int B[];`

déclare un *tableau* d'éléments du type **int**

B désigne l'adresse de la première composante de *B*.

(Cette adresse est toujours constante)

B[i] désigne le contenu de la composante *i* du tableau

&B[i] désigne l'adresse de la composante *i* du tableau

en utilisant le formalisme pointeur:

B+i désigne l'adresse de la composante *i* du tableau

***(B+i)** désigne le contenu de la composante *i* du tableau

`int *p;`

déclare un *pointeur* sur des éléments du type **int**.

p peut pointer sur des variables simples du type **int** ou
sur les composantes d'un tableau du type **int**.

P désigne l'adresse contenue dans *P*
(Cette adresse est variable)

***P** désigne le contenu de l'adresse dans *P*

Si *P* pointe dans un tableau, alors

P désigne l'adresse de la première composante

P+i désigne l'adresse de la *i*-ième composante derrière *P*

***(P+i)** désigne le contenu de la *i*-ième composante derrière *P*

9.2 Arithmétique des pointeurs

Comme les pointeurs jouent un rôle si important, le langage C soutient une série d'opérations arithmétiques sur les pointeurs que l'on ne rencontre en général que dans les langages machines. Le confort de ces opérations en C est basé sur le principe suivant:

Toutes les opérations avec les pointeurs tiennent compte automatiquement du type et de la grandeur des objets pointés.

- Affectation par un pointeur sur le même type

Soient P1 et P2 deux pointeurs sur le même type de données, alors l'instruction

```
P1 = P2;
```

fait pointer P1 sur le même objet que P2

- Addition et soustraction d'un nombre entier

Si P pointe sur l'élément A[i] d'un tableau, alors

```
P+n  pointe sur A[i+n]
```

```
P-n  pointe sur A[i-n]
```

- Incrémentation et décrémentation d'un pointeur

Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction

```
P++;  P pointe sur A[i+1]
```

```
P+=n; P pointe sur A[i+n]
```

```
P--;  P pointe sur A[i-1]
```

```
P-=n; P pointe sur A[i-n]
```

Domaine des opérations

L'addition, la soustraction, l'incrémentation et la décrémentation sur les pointeurs sont seulement définies à l'intérieur d'un tableau. Si l'adresse formée par le pointeur et l'indice sort du domaine du tableau, alors le résultat n'est pas défini.

Seule exception: Il est permis de 'pointer' sur le premier octet derrière un tableau (à condition que cet octet se trouve dans le même segment de mémoire que le tableau). Cette règle, introduite avec le standard ANSI-C, légalise la définition de boucles qui incrémentent le pointeur *avant* l'évaluation de la condition d'arrêt.

Exemples

```
int A[10];
```

```
int *p;
```

```
p = A+9;  /* dernier élément -> légal */
```

```
p = A+10; /* dernier élément + 1 -> légal */
```

```
p = A+11; /* dernier élément + 2 -> illégal */
```

```
p = A-1;  /* premier élément - 1 -> illégal */
```

- Soustraction de deux pointeurs

Soient P1 et P2 deux pointeurs qui pointent *dans le même tableau*:

P1-P2 fournit le nombre de composantes comprises entre P1 et P2.

Le résultat de la soustraction **P1-P2** est

- négatif, si P1 précède P2
- zéro, si P1 = P2
- positif, si P2 précède P1
- indéfini, si P1 et P2 ne pointent pas dans le même tableau

Plus généralement, la soustraction de deux pointeurs qui pointent dans le même tableau est équivalente à la soustraction des indices correspondants.

- Comparaison de deux pointeurs

On peut comparer deux pointeurs par <, >, <=, >=, ==, !=.

La comparaison de deux pointeurs qui pointent *dans le même tableau* est équivalente à la comparaison des indices correspondants. (Si les pointeurs ne pointent pas dans le même tableau, alors le résultat est donné par leurs positions relatives dans la mémoire).

9.3 Pointeurs et chaînes de caractères

De la même façon qu'un pointeur sur **int** peut contenir l'adresse d'un nombre isolé ou d'une composante d'un tableau, un pointeur sur **char** peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères. Un pointeur sur **char** peut en plus contenir l'adresse d'une chaîne de caractères constante et il peut même être *initialisé* avec une telle adresse.

A la fin de ce chapitre, nous allons anticiper avec un exemple et montrer que les pointeurs sont les éléments indispensables mais effectifs des fonctions en C.

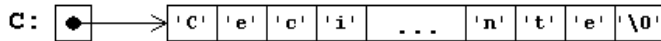
- Pointeurs sur char et chaînes de caractères constantes

Affectation

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

Exemple

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



Nous pouvons lire cette chaîne constante (p.ex: pour l'afficher), mais il n'est pas recommandé de la modifier, parce que le résultat d'un programme qui essaie de modifier une chaîne de caractères constante n'est pas prévisible en ANSI-C.

Initialisation

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

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

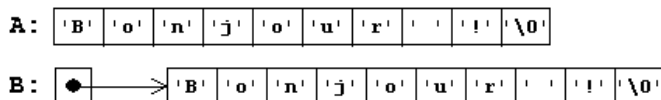
Attention !

Il existe une différence importante entre les deux déclarations:

```
char A[] = "Bonjour !"; /* un tableau */  
char *B = "Bonjour !"; /* un pointeur */
```

A est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '\0'. Les caractères de la chaîne peuvent être changés, mais le nom A va toujours pointer sur la même adresse en mémoire.

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.



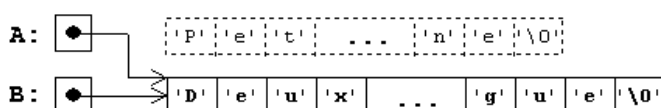
Modification

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

Exemple

```
char *A = "Petite chaîne";  
char *B = "Deuxième chaîne un peu plus longue";  
A = B;
```

Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue:



Attention !

Les affectations discutées ci-dessus ne peuvent pas être effectuées avec des tableaux de caractères:

Exemple




```
char A[45] = "Petite chaîne";
char B[45] = "Deuxième chaîne un peu plus longue";
char C[30];
A = B;          /* IMPOSSIBLE -> ERREUR !!! */
C = "Bonjour !"; /* IMPOSSIBLE -> ERREUR !!! */
```

A:

'P'	'e'	't'	...	'n'	'e'	'\0'
-----	-----	-----	-----	-----	-----	------

B:

'D'	'e'	'u'	'x'	...	'g'	'u'	'e'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	------

Dans cet exemple, nous essayons de copier l'adresse de B dans A, respectivement l'adresse de la chaîne constante dans C. Ces opérations sont impossibles et illégales parce que *l'adresse représentée par le nom d'un tableau reste toujours constante*.

Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre (p.ex. dans une boucle) ou déléguer cette charge à une fonction de `<stdio>` ou `<string>`.

Conclusions:

- Utilisons des *tableaux de caractères* pour déclarer les chaînes de caractères que nous voulons modifier.
- Utilisons des *pointeurs sur char* pour manipuler des chaînes de caractères constantes (dont le contenu ne change pas).
- Utilisons de préférence des *pointeurs* pour effectuer les manipulations à l'intérieur des tableaux de caractères. (voir aussi les remarques ci-dessous).

Perspectives et motivation

- Avantages des pointeurs sur char

Comme la fin des chaînes de caractères est marquée par un symbole spécial, nous n'avons pas besoin de connaître la longueur des chaînes de caractères; nous pouvons même laisser de côté les indices d'aide et parcourir les chaînes à l'aide de pointeurs.

Cette façon de procéder est indispensable pour traiter de chaînes de caractères dans des fonctions. En anticipant sur la matière du chapitre 10, nous pouvons ouvrir une petite parenthèse pour illustrer les avantages des pointeurs dans la définition de fonctions traitant des chaînes de caractères:



Pour fournir un tableau comme paramètre à une fonction, il faut passer *l'adresse du tableau* à la fonction. Or, les *paramètres des fonctions sont des variables locales*, que nous pouvons utiliser comme variables d'aide. Bref, une fonction obtenant une chaîne de caractères comme paramètre, dispose d'une *copie locale de l'adresse de la chaîne*. Cette copie peut remplacer les indices ou les variables d'aide du formalisme tableau.

Discussion d'un exemple

Reprenons l'exemple de la fonction **strcpy**, qui copie la chaîne CH2 vers CH1. Les deux chaînes sont les arguments de la fonction et elles sont déclarées comme *pointeurs sur char*. La première version de **strcpy** est écrite entièrement à l'aide du formalisme tableau:

```
void strcpy(char *CH1, char *CH2)
{
    int I;
    I=0;
    while ((CH1[I]=CH2[I]) != '\0')
        I++;
}
```

Dans une première approche, nous pourrions remplacer simplement la notation `tableau[I]` par `*(tableau + I)`, ce qui conduirait au programme:

```
void strcpy(char *CH1, char *CH2)
{
    int I;
    I=0;
    while ((*CH1+I)=*(CH2+I)) != '\0')
        I++;
}
```

Cette transformation ne nous avance guère, nous avons tout au plus gagné quelques millièmes de secondes lors de la compilation. Un 'véritable' avantage se laisse gagner en calculant directement avec les pointeurs CH1 et CH2 :

```
void strcpy(char *CH1, char *CH2)
{
    while ((*CH1=*CH2) != '\0')
    {
        CH1++;
        CH2++;
    }
}
```

Comme nous l'avons déjà constaté dans l'introduction de ce manuel, un vrai professionnel en C escaladerait les 'simplifications' jusqu'à obtenir:

```
void strcpy(char *CH1, char *CH2)
{
    while (*CH1++ = *CH2++)
        ;
}
```

Assez 'optimisé' - fermons la parenthèse et familiarisons-nous avec les notations et les manipulations du 'formalisme pointeur' ...

9.4 Pointeurs et tableaux à deux dimensions

L'arithmétique des pointeurs se laisse élargir avec *toutes* ses conséquences sur les tableaux à deux dimensions. Voyons cela sur un exemple:

Exemple

Le tableau M à deux dimensions est défini comme suit:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
```

Le nom du tableau M représente l'adresse du premier élément du tableau et pointe (oh, surprise...) sur le **tableau** M[0] qui a la valeur:

```
{0,1,2,3,4,5,6,7,8,9}.
```

L'expression (M+1) est l'adresse du deuxième élément du tableau et pointe sur M[1] qui a la valeur:

```
{10,11,12,13,14,15,16,17,18,19}.
```

Explication

Au sens strict du terme, un tableau à deux dimensions est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, le premier élément de la matrice M est le **vecteur** {0,1,2,3,4,5,6,7,8,9}, le deuxième élément est {10,11,12,13,14,15,16,17,18,19} et ainsi de suite.

L'arithmétique des pointeurs qui respecte automatiquement les dimensions des éléments conclut logiquement que:

M+I désigne l'adresse du tableau M[I]

Problème

Comment pouvons-nous accéder à l'aide de pointeurs aux éléments de chaque composante du tableau, c.à-d.: aux éléments M[0][0], M[0][1], ... , M[3][9] ?

Discussion

Une solution consiste à convertir la valeur de M (qui est un pointeur sur *un tableau du type int*) en un pointeur sur *int*. On pourrait se contenter de procéder ainsi:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};

int *P;
P = M; /* conversion automatique */
```



Cette dernière affectation entraîne une conversion automatique de l'adresse &M[0] dans l'adresse &M[0][0]. (Remarquez bien que l'adresse transmise reste la même, seule la nature du pointeur a changé).

Cette solution n'est pas satisfaisante à cent pour-cent: Généralement, on gagne en lisibilité en explicitant la conversion mise en oeuvre par l'opérateur de conversion forcée ("cast"), qui évite en plus des messages d'avertissement de la part du compilateur.

Solution

Voici finalement la version que nous utiliserons:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};

int *P;
P = (int *)M; /* conversion forcée */
```



Dû à la mémorisation ligne par ligne des tableaux à deux dimensions, il nous est maintenant possible traiter M à l'aide du pointeur P comme un tableau unidimensionnel de dimension 4*10.

Exemple

Les instructions suivantes calculent la somme de tous les éléments du tableau M:

```

int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};

int *P;
int I, SOM;
P = (int*)M;
SOM = 0;
for (I=0; I<40; I++)
    SOM += *(P+I);

```

Attention !

Lors de l'interprétation d'un tableau à deux dimensions comme tableau unidimensionnel **il faut calculer avec le nombre de colonnes indiqué dans la déclaration** du tableau.



Exemple

Pour la matrice A, nous réservons de la mémoire pour 3 lignes et 4 colonnes, mais nous utilisons seulement 2 lignes et 2 colonnes:

```

int A[3][4];
A[0][0]=1;
A[0][1]=2;
A[1][0]=10;
A[1][1]=20;

```

Dans la mémoire, ces composantes sont stockées comme suit :

A[0][0]A[0][1]				A[1][0]A[1][1]				A[2][0]A[2][1]			
...	1	2	?	?	10	20	?	?	?	?	...
Adresse : 1E06		1E0A		1E0E		1E12		1E16		1E1A	1E1E

A ↖ ↗

L'adresse de l'élément A[I][J] se calcule alors par:

$$A + I*4 + J$$

Conclusion

Pour pouvoir travailler à l'aide de pointeurs dans un tableau à deux dimensions, nous avons besoin de quatre données:

- l'adresse du premier élément du tableau converti dans le type simple des éléments du tableau
- la longueur d'une ligne réservée en mémoire
(- voir déclaration - ici: 4 colonnes)
- le nombre d'éléments effectivement utilisés dans une ligne
(- p.ex: lu au clavier - ici: 2 colonnes)
- le nombre de lignes effectivement utilisées
(- p.ex: lu au clavier - ici: 2 lignes)

9.5 Exercice : Pointeurs et Tableaux

Exercice: Soit P un pointeur qui "*pointe*" sur un tableau A:

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};  
int *P;  
P = A;
```

Quelles valeurs ou adresses fournissent ces expressions:

- a) *P+2
- b) *(P+2)
- c) &P+1
- d) &A[4]-3
- e) A+3
- f) &A[7]-P
- g) P+(*P-10)
- h) *(P+(P+8)-A[7])

9.6 Exercice : Formalismes

Exercice: Vous devez écrire un programme qui remplit un tableau puis l'affiche en utilisant :

- le **formalisme tableau**: en utilisant `T[i]` pour accéder à chaque élément du tableau
- le **formalisme pointeur**: on utilise le nom du tableau comme un pointeur fixe
- les **pointeurs**:
 - un **pointeur statique**: on utilise un pointeur initialisé au début du tableau pour parcourir le tableau, ce pointeur ne doit pas évoluer
 - un **pointeur dynamique**: on initialise un pointeur au début du tableau, puis on l'incrémente pour accéder à chaque élément du tableau.

9.7 Exercice : Adressage d'un tableau

Exercice:

Ecrire un programme qui lit deux tableaux **A** et **B** et leurs dimensions **N** et **M** au clavier et qui ajoute les éléments de **B** à la fin de **A**.

Utiliser le formalisme pointeur à chaque fois que cela est possible. Faites deux versions, une avec l'utilisation d'un ou plusieurs pointeurs statiques, l'autre avec des pointeurs dynamiques

9.8 Exercice : Elimine Occurence

Ecrire un programme qui lit un entier X et un tableau A du type **int** au clavier et élimine toutes les occurrences de X dans A en tassant les éléments restants.

Le programme utilisera les pointeurs $P1$ et $P2$ pour parcourir le tableau (statique et dynamique).

9.9 Exercice : Inverse Tableau

Exercice:

Ecrire un programme qui range les éléments d'un tableau **A** du type **int** dans l'**ordre inverse**.

Le programme utilisera des pointeurs **P1** et **P2** et une variable numérique **AIDE** pour la permutation des éléments.

9.10 Exercice : Tableaux à 2 dimensions

Exercice: Vous allez écrire un programme qui calcule la somme des éléments d'un tableau à 2 dimensions en utilisant 2 boucles, une pour les lignes et une pour les colonnes... avec des pointeurs bien sur...

Donc il faut rentrer les valeurs dans le tableau (passer un pointeur en argument de la fonction pour faire passer le tableau, faire aussi passer le nombre de colonnes et le nombre de lignes)

afficher le tableau avec une fonction dédiée

calculer la somme des éléments du tableau..

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
```

Eventuellement, une fois fini, écrivez une fonction qui calcule la somme d'une colonne du tableau ou d'une ligne du tableau... à vous d'imaginer comment faire cela...

10 Pointeurs et chaînes de caractères

De la même façon qu'un pointeur sur **int** peut contenir l'adresse d'un nombre isolé ou d'une composante d'un tableau, un pointeur sur **char** peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères. Un pointeur sur **char** peut en plus contenir l'adresse d'une chaîne de caractères constante et il peut même être **initialisé** avec une telle adresse.

10.1 Pointeurs sur char et Chaînes de Caractères Constantes

Affectation: On peut attribuer *l'adresse d'une chaîne de caractères constante* à un pointeur sur **char**:

Exemple:

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



ATTENTION!!! Nous pouvons lire cette chaîne constante (p.ex: pour l'afficher), mais il n'est pas recommandé de la modifier, parce que le résultat d'un programme qui essaie de modifier une chaîne de caractères constante n'est pas prévisible en ANSI-C.

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

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

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

```
char A[] = "Bonjour !"; /* un tableau */  
char *B = "Bonjour !"; /* un pointeur */
```

A est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '\0'. Les caractères de la chaîne peuvent être changés, mais le nom A va toujours pointer sur la même adresse en mémoire.

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.



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

Exemple:

```
char *A = "Petite chaîne";  
char *B = "Deuxième chaîne un peu plus longue";  
A = B;
```

Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue:



Attention !!! Les affectations discutées ci-dessus ne peuvent pas être effectuées avec des tableaux de caractères:

Exemple de Solution Incorrecte:

```
char A[45] = "Petite chaîne";  
char B[45] = "Deuxième chaîne un peu plus longue";  
char C[30];  
A = B; /* IMPOSSIBLE -> ERREUR !!! */  
C = "Bonjour !"; /* IMPOSSIBLE -> ERREUR !!! */
```



Dans cet exemple, nous essayons de copier l'adresse de B dans A, respectivement l'adresse de la chaîne constante dans C. Ces opérations sont impossibles et illégales parce que *l'adresse représentée par le nom d'un tableau reste toujours constante*.

Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre (p.ex. dans une boucle) ou déléguer cette charge à une fonction de `<stdio>` ou `<string>`.

Conclusions:

- Utilisons des **tableaux de caractères** pour déclarer les chaînes de caractères que nous voulons modifier.
- Utilisons des **pointeurs sur char** pour manipuler des chaînes de caractères constantes (dont le contenu ne change pas).

- Utilisons de préférence des **pointeurs** pour effectuer les manipulations à l'intérieur des tableaux de caractères. (voir aussi les remarques ci-dessous).

10.2 Avantages des Pointeurs sur char

REMARQUE!!! Comme la fin des chaînes de caractères est marquée par un symbole spécial, nous n'avons pas besoin de connaître la longueur des chaînes de caractères; nous pouvons même laisser de côté les indices d'aide et parcourir les chaînes à l'aide de pointeurs.

Cette façon de procéder est indispensable pour traiter de chaînes de caractères dans des fonctions.

Pour fournir un tableau comme paramètre à une fonction, il faut passer *l'adresse du tableau* à la fonction. Or, les *paramètres des fonctions sont des variables locales*, que nous pouvons utiliser comme variables d'aide. Bref, une fonction obtenant une chaîne de caractères comme paramètre, dispose d'une *copie locale de l'adresse de la chaîne*. Cette copie peut remplacer les indices ou les variables d'aide du formalisme tableau.

Discussion d'un exemple: Reprenons l'exemple de la fonction **strcpy**, qui copie la chaîne CH2 vers CH1. Les deux chaînes sont les arguments de la fonction et elles sont déclarées comme *pointeurs sur char*. La première version de **strcpy** est écrite entièrement à l'aide du formalisme tableau:

```
void strcpy(char *CH1, char *CH2)
{
    int I;
    I=0;
    while ((CH1[I]=CH2[I]) != '\0')
        I++;
}
```

Dans une première approche, nous pourrions remplacer simplement la notation **tableau[I]** par ***(tableau + I)**, ce qui conduirait au programme:

```
void strcpy(char *CH1, char *CH2)
{
    int I;
    I=0;
    while ((*CH1+I)=*(CH2+I)) != '\0')
        I++;
}
```

Cette transformation ne nous avance guère, nous avons tout au plus gagné quelques millièmes de secondes lors de la compilation. Un 'véritable' avantage se laisse gagner en calculant directement avec les pointeurs CH1 et CH2 :

```
void strcpy(char *CH1, char *CH2)
{
    while ((*CH1=*CH2) != '\0')
    {
        CH1++;
        CH2++;
    }
}
```

Un vrai professionnel en C escaladerait les 'simplifications' jusqu'à obtenir:

```
void strcpy(char *CH1, char *CH2)
{
    while (*CH1++ = *CH2++)
        ;
}
```

10.3 Tableaux de pointeurs

Si nous avons besoin d'un ensemble de pointeurs du même type, nous pouvons les réunir dans un tableau de pointeurs.

Déclaration d'un tableau de pointeurs

`<Type> *<NomTableau>[<N>]` déclare un tableau `<NomTableau>` de `<N>` pointeurs sur des données du type `<Type>`.

Exemple: `double *A[10];` déclare un tableau de 10 pointeurs sur des rationnels du type **double** dont les adresses et les valeurs ne sont pas encore définies.

Remarque: Le plus souvent, les tableaux de pointeurs sont utilisés pour mémoriser de façon économique des **chaînes de caractères de différentes longueurs**. Dans la suite, nous allons surtout considérer les tableaux de pointeurs sur des chaînes de caractères.

Initialisation: Nous pouvons initialiser les pointeurs d'un tableau sur **char** par les adresses de chaînes de caractères constantes.

Exemple:

```
char *JOUR[] = {"dimanche", "lundi", "mardi",  
               "mercredi", "jeudi", "vendredi",  
               "samedi"};
```

déclare un tableau `JOUR[]` de 7 pointeurs sur **char**. Chacun des pointeurs est initialisé avec l'adresse de l'une des 7 chaînes de caractères.



On peut afficher les 7 chaînes de caractères en fournissant les adresses contenues dans le tableau `JOUR` à **printf** (ou **puts**) :

```
int I;  
for (I=0; I<7; I++) printf("%s\n", JOUR[I]);
```

Comme `JOUR[I]` est un pointeur sur **char**, on peut afficher les premières lettres des jours de la semaine en utilisant l'opérateur 'contenu de' :

```
int I;  
for (I=0; I<7; I++) printf("%c\n", *JOUR[I]);
```

L'expression `JOUR[I]+J` désigne la J-ième lettre de la I-ième chaîne. On peut afficher la troisième lettre de chaque jour de la semaine par:

```
int I;  
for (I=0; I<7; I++) printf("%c\n", *(JOUR[I]+2));
```

Résumons: Les tableaux de pointeurs

`int *D[];` déclare un **tableau de pointeurs** sur des éléments du type **int**

`D[i]` peut pointer sur des variables simples ou

sur les composantes d'un tableau.

`D[i]` désigne l'adresse contenue dans l'élément `i` de `D`
(Les adresses dans `D[i]` sont variables)

`*D[i]` désigne le contenu de l'adresse dans `D[i]`

Si `D[i]` pointe dans un tableau,

<code>D[i]</code>	désigne l'adresse de la première composante
<code>D[i]+j</code>	désigne l'adresse de la j-ième composante
<code>*(D[i]+j)</code>	désigne le contenu de la j-ième composante

10.4 Exercice : Palindrome

Exercice:

Ecrire de deux façons différentes, un programme qui vérifie sans utiliser une fonction de `<string>`, si une chaîne **CH** introduite au clavier est un palindrome:

a) en utilisant uniquement le formalisme pointeur (en utilisant le nom du tableau comme un pointeur fixe...)

b) en utilisant 2 pointeurs au lieu des indices numériques (le 1er pointeur est initialisé au début du tableau, le 2nd pointeur est initialisé à la fin du tableau)

Rappel: Un palindrome est un mot qui reste le même qu'on le lise de gauche à droite ou de droite à gauche:

Exemples: PIERRE ==> n'est pas un palindrome

OTTO ==> est un palindrome

23432 ==> est un palindrome

10.5 Exercice : Longue chaîne

Exercice:

Ecrire un programme qui lit une chaîne de caractères **CH** et détermine la longueur de la chaîne à l'aide d'un pointeur **P**.

Le programme n'utilisera pas de variables numériques.

10.6 Exercice : Nombre de mots

Exercice:

Ecrire un programme qui lit une chaîne de caractères **CH** et détermine le nombre de mots contenus dans la chaîne.

Utiliser un pointeur **P**, une variable logique, la fonction **isspace** de la bibliothèque **<ctype.h>** et une variable numérique **N** qui contiendra le nombre des mots.

10.7 Exercice : Compter les lettres

Exercice:

Ecrire un programme qui lit une chaîne de caractères **CH** au clavier et qui compte les occurrences des lettres de l'alphabet en ne distinguant pas les majuscules et les minuscules.

Utiliser un tableau **ABC** de dimension 26 pour mémoriser le résultat et un pointeur **PCH** pour parcourir la chaîne **CH** et un pointeur **PABC** pour parcourir **ABC**.

Afficher seulement le nombre des lettres qui apparaissent au moins une fois dans le texte.

Exemple:

Entrez un ligne de texte (max. 100 caractères) :

Jeanne

La chaîne "Jeanne" contient :

1 fois la lettre 'A'

2 fois la lettre 'E'

1 fois la lettre 'J'

3 fois la lettre 'N'

10.8 Exercice : Chaînes de caractères constantes

Exercice:

Soit **MOIS**[] un tableau de pointeurs qui "*pointent*" sur les chaînes de caractères constantes suivantes:

"Janvier", "Fevrier", "Mars", "Avril", "Mai", "Juin", "Juillet", "Aout", "Septembre", "Octobre", "Novembre", "Decembre"

Ecrire un programme qui demande à l'utilisateur une date au format JJ/MM/AA.
Puis la transforme en date au format Jour Mois Année.

Exemple:

Entrez une Date au format JJ/MM/AA: 06/02/06

Voici votre Date: 6 Fevrier 2006

11 Allocation dynamique de mémoire

Nous avons vu que l'utilisation de pointeurs nous permet de mémoriser économiquement des données de différentes grandeurs. Si nous générons ces données pendant l'exécution du programme, il nous faut des moyens pour réserver et libérer de la mémoire au fur et à mesure que nous en avons besoin. Nous parlons alors de **l'allocation dynamique** de la mémoire.

Revoyons d'abord de quelle façon la mémoire a été réservée dans les programmes que nous avons écrits jusqu'ici.

11.1 Déclaration statique de donnée

Chaque variable dans un programme a besoin d'un certain nombre d'octets en mémoire. Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des déclarations des données. Dans tous ces cas, le nombre d'octets à réserver était déjà connu pendant la compilation. Nous parlons alors de la **déclaration statique** des variables.

Exemples:

```
float A, B, C;          /* réservation de 12 octets */
short D[10][20];        /* réservation de 400 octets */
char E[] = {"Bonjour !"};
                        /* réservation de 10 octets */
char F[][10] = {"un", "deux", "trois", "quatre"};
                        /* réservation de 40 octets */
```

Pointeurs: Le nombre d'octets à réserver pour un *pointeur* dépend de la machine et du 'modèle' de mémoire choisi, mais il est déjà connu lors de la compilation. Un pointeur est donc aussi déclaré statiquement. Supposons dans la suite qu'un pointeur ait besoin de p octets en mémoire. (En DOS: p =2 ou p = 4)

Exemples:

```
double *G;              /* réservation de p octets */
char *H;                 /* réservation de p octets */
float *I[10];            /* réservation de 10*p octets */
```

Chaînes de caractères constantes: L'espace pour les *chaînes de caractères constantes* qui sont affectées à des pointeurs ou utilisées pour initialiser des pointeurs sur **char** est aussi réservé automatiquement:

Exemples

```
char *J = "Bonjour !";
                /* réservation de p+10 octets */
float *K[] = {"un", "deux", "trois", "quatre"};
                /* réservation de 4*p+3+5+6+7 octets */
```

11.2 Allocation dynamique

Problème: Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation. Ce serait alors un gaspillage de réserver toujours l'espace maximal prévisible. Il nous faut donc un moyen de gérer la mémoire lors de l'exécution du programme.

Exemple: Nous voulons lire 10 phrases au clavier et mémoriser les phrases en utilisant un tableau de pointeurs sur **char**. Nous déclarons ce tableau de pointeurs par:

```
char *TEXTE[10];
```

Pour les 10 pointeurs, nous avons besoin de 10*p octets. Ce nombre est connu dès le départ et les octets sont réservés automatiquement. Il nous est cependant impossible de prévoir à l'avance le nombre d'octets à réserver pour les phrases elles-mêmes qui seront introduites lors de l'exécution du programme ...

Allocation dynamique: La réservation de la mémoire pour les 10 phrases peut donc seulement se faire **pendant l'exécution du programme**. Nous parlons dans ce cas de l'**allocation dynamique** de la mémoire.

11.3 La fonction malloc et l'opérateur sizeof

La fonction **malloc** de la bibliothèque `<stdlib>` nous aide à localiser et à réserver de la mémoire au cours d'un programme. Elle nous donne accès au tas (*heap*); c.-à-d. à l'espace en mémoire laissé libre une fois mis en place le DOS, les gestionnaires, les programmes résidents, le programme lui-même et la pile (*stack*).

La Fonction malloc:

malloc(<N>) fournit l'adresse d'un bloc en mémoire de <N> octets libres ou la valeur zéro s'il n'y a pas assez de mémoire.

Attention !!! Sur notre système, le paramètre <N> est du type **unsigned int**. A l'aide de **malloc**, nous ne pouvons donc pas réserver plus de 65535 octets à la fois!

Exemple: Supposons que nous ayons besoin d'un bloc en mémoire pour un texte de 4000 caractères. Nous disposons d'un pointeur T sur **char** (**char *T**). Alors l'instruction:

```
T = malloc(4000);
```

fournit l'adresse d'un bloc de 4000 octets libres et l'affecte à T. S'il n'y a plus assez de mémoire, T obtient la valeur zéro.

Si nous voulons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine à l'autre, nous avons besoin de la grandeur effective d'une donnée de ce type. L'opérateur **sizeof** nous aide alors à préserver la portabilité du programme.

L'Opérateur Unaire sizeof

sizeof <var>	fournit la grandeur de la variable <var>
sizeof <const>	fournit la grandeur de la constante <const>
sizeof (<type>)	fournit la grandeur pour un objet du type <type>

Exemple: Après la déclaration,

```
short A[10];  
char B[5][10];
```

nous obtenons les résultats suivants sur un IBM-PC (ou compatible):

sizeof A	s'évalue à 20
sizeof B	s'évalue à 50
sizeof 4.25	s'évalue à 8
sizeof "Bonjour !"	s'évalue à 10
sizeof(float)	s'évalue à 4
sizeof(double)	s'évalue à 8

Exemple: Nous voulons réserver de la mémoire pour X valeurs du type **int**; la valeur de X est lue au clavier:

```
int X;  
int *PNum;  
printf("Introduire le nombre de valeurs :");  
scanf("%d", &X);  
PNum = malloc(X*sizeof(int));
```

REMARQUE!!! S'il n'y a pas assez de mémoire pour effectuer une action avec succès, il est conseillé d'interrompre l'exécution du programme à l'aide de la commande **exit** (de `<stdlib>`) et de renvoyer une valeur différente de zéro comme code d'erreur du programme.

Exemple: Le programme suivant lit 10 phrases au clavier, recherche des blocs de mémoire libres assez grands pour la mémorisation et passe les adresses aux composantes du tableau **TEXTE[]**. S'il n'y a pas assez de mémoire pour une chaîne, le programme affiche un message d'erreur et interrompt le programme avec le code d'erreur -1.

Nous devons utiliser une variable d'aide **INTRO** comme zone intermédiaire (non dynamique). Pour cette raison, la longueur maximale d'une phrase est fixée à 500 caractères.


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

main()
{
    /* Déclarations */
    char INTRO[500];
    char *TEXTE[10];
    int I;
    /* Traitement */
    for (I=0; I<10; I++)
    {
        gets(INTRO);
        /* Réserve de la mémoire */
        TEXTE[I] = malloc(strlen(INTRO)+1);
        /* S'il y a assez de mémoire, ... */
        if (TEXTE[I])
            /* copier la phrase à l'adresse */
            /* fournie par malloc, ... */
            strcpy(TEXTE[I], INTRO);
        else
        {
            /* sinon quitter le programme */
            /* après un message d'erreur. */
            printf("ERREUR: Pas assez de mémoire \n");
            exit(-1);
        }
    }
    return 0;
}
```

11.4 La fonction Free

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de **malloc**, alors nous pouvons le libérer à l'aide de la fonction **free** de la bibliothèque `<stdlib.h>`.

free(<Pointeur>) libère le bloc de mémoire désigné par le **<Pointeur>** n'a pas d'effet si le pointeur a la valeur zéro.

Attention !!!

- La fonction **free** peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par **malloc**.
- La fonction **free** ne change pas le contenu du pointeur; il est conseillé d'affecter la valeur zéro au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.
- Si nous ne libérons pas explicitement la mémoire à l'aide **free**, alors elle est libérée automatiquement à la fin du programme.

11.5 Exercice : Inverse phrases

Exercice:

Ecrire un programme qui lit 10 phrases d'une longueur maximale de 200 caractères au clavier et qui les mémorise dans un tableau de pointeurs sur **char** en réservant dynamiquement l'emplacement en mémoire pour les chaînes. Ensuite, l'ordre des phrases est inversé en modifiant les pointeurs et le tableau résultant est affiché.

11.6 Exercice : Effacer Phrases

Exercice:

Ecrire un programme qui lit 10 mots au clavier (longueur maximale: 50 caractères) et attribue leurs adresses à un tableau de pointeurs **MOT**.

Effacer les 10 mots un à un, en suivant l'ordre lexicographique et en libérant leur espace en mémoire.

Afficher à chaque fois les mots restants en attendant la confirmation de l'utilisateur (par 'Enter').

11.7 Exercice : Concaténer phrases

Exercice:

Ecrire un programme qui lit 10 mots au clavier (longueur maximale: 50 caractères) et attribue leurs adresses à un tableau de pointeurs **MOT**.

Copier les mots selon l'ordre lexicographique en une seule 'phrase' dont l'adresse est affectée à un pointeur **PHRASE**.

Réserver l'espace nécessaire à la PHRASE avant de copier les mots.

Libérer la mémoire occupée par chaque mot après l'avoir copié. Utiliser les fonctions de *<string>*.

12 Fichiers séquentiels

En C, les communications d'un programme avec son environnement se font par l'intermédiaire de fichiers. Pour le programmeur, tous les périphériques, même le clavier et l'écran, sont des fichiers. Jusqu'ici, nos programmes ont lu leurs données dans le fichier d'entrée standard, (c.-à-d.: le clavier) et ils ont écrit leurs résultats dans le fichier de sortie standard (c.-à-d.: l'écran). Nous allons voir dans ce chapitre, comment nous pouvons créer, lire et modifier nous-mêmes des fichiers sur les périphériques disponibles.

12.1 Définitions et Propriétés

Un **fichier** est un ensemble structuré de données stocké en général sur un support externe (disquette, disque dur, disque optique, bande magnétique, ...).

Un **fichier structuré** contient une suite *d'enregistrements* homogènes, qui regroupent le plus souvent plusieurs composantes appartenant ensemble (*champs*).

Dans des **fichiers séquentiels**, les enregistrements sont mémorisés consécutivement dans l'ordre de leur entrée et peuvent seulement être lus dans cet ordre. Si on a besoin d'un enregistrement précis dans un fichier séquentiel, il faut lire tous les enregistrements qui le précèdent, en commençant par le premier.

En simplifiant, nous pouvons nous imaginer qu'un fichier séquentiel est enregistré sur une bande magnétique:



Propriétés: Les fichiers séquentiels que nous allons considérer dans ce cours auront les propriétés suivantes:

- Les fichiers se trouvent ou bien en état d'écriture ou bien en état de lecture; nous ne pouvons pas simultanément lire et écrire dans le même fichier.
- A un moment donné, on peut uniquement accéder à un seul enregistrement; celui qui se trouve en face de la tête de lecture/écriture.
- Après chaque accès, la tête de lecture/écriture est déplacée derrière la donnée lue en dernier lieu.

Fichiers standards: Il existe deux fichiers spéciaux qui sont définis par défaut pour tous les programmes:

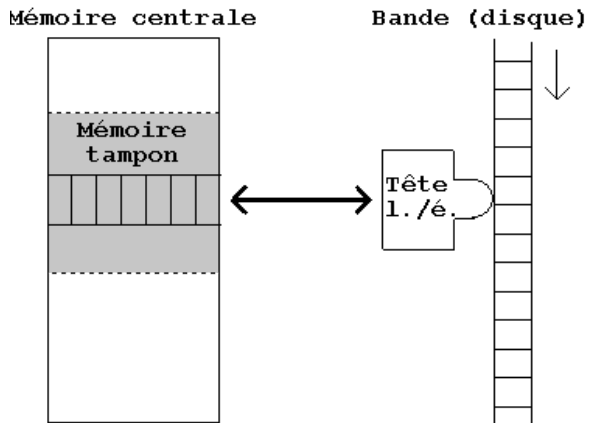
- ***stdin*** le fichier d'entrée standard
- ***stdout*** le fichier de sortie standard

En général, *stdin* est lié au clavier et *stdout* est lié à l'écran, c.-à-d. les programmes lisent leurs données au clavier et écrivent les résultats sur l'écran.

En fait, l'affectation de *stdin* et *stdout* est gérée par le système d'exploitation; ainsi le programme ne 'sait' pas d'où viennent les données et où elles vont.

12.2 La mémoire Tampon

Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une **mémoire tampon** (angl.: *buffer*). La mémoire tampon est une zone de la mémoire centrale de la machine réservée à un ou plusieurs enregistrements du fichier. L'utilisation de la mémoire tampon a l'effet de réduire le nombre d'accès à la périphérie d'une part et le nombre des mouvements de la tête de lecture/écriture d'autre part.



12.3 Accès aux fichiers séquentiels

Les problèmes traitant des fichiers ont généralement la forme suivante: un fichier donné par son nom (et en cas de besoin le chemin d'accès sur le médium de stockage) doit être créé, lu ou modifié. La question qui se pose est alors:

Comment pouvons-nous relier le nom d'un fichier sur un support externe avec les instructions qui donnent accès au contenu du fichier ?

En résumé, la méthode employée sera la suivante:

Avant de lire ou d'écrire un fichier, l'accès est notifié par la commande **fopen**. **fopen** accepte le nom du fichier (p.ex: "A:\ADRESSES.DAT"), négocie avec le système d'exploitation et fournit un pointeur spécial qui sera utilisé ensuite lors de l'écriture ou la lecture du fichier. Après les traitements, il faut annuler la liaison entre le nom du fichier et le pointeur à l'aide de la commande **fclose**.

On peut dire aussi qu'entre les événements **fopen()** et **fclose()** le fichier est ouvert.

12.4 Le type **FILE***

Pour pouvoir travailler avec un fichier, un programme a besoin d'un certain nombre d'informations au sujet du fichier:

- adresse de la mémoire tampon,
- position actuelle de la tête de lecture/écriture,
- type d'accès au fichier: écriture, lecture, ...
- état d'erreur,
- . . .

Ces informations (dont nous n'aurons pas à nous occuper), sont rassemblées dans une structure du type spécial **FILE**. Lorsque nous ouvrons un fichier avec la commande **fopen**, le système génère automatiquement un bloc du type **FILE** et nous fournit son adresse.

Tout ce que nous avons à faire dans notre programme est:

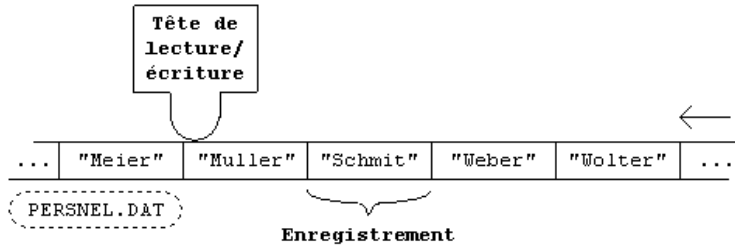
- * déclarer un pointeur du type **FILE*** pour chaque fichier dont nous avons besoin,
- * affecter l'adresse retournée par **fopen** à ce pointeur,
- * employer le pointeur à la place du nom du fichier dans toutes les instructions de lecture ou d'écriture,
- * libérer le pointeur à la fin du traitement à l'aide de **fclose**.

12.5 Exemple: Créer et Afficher un Fichier Séquentiel

Avant de discuter les détails du traitement des fichiers, nous vous présentons un petit exemple comparatif qui réunit les opérations les plus importantes sur les fichiers.

Problème

On se propose de créer un fichier qui est formé d'enregistrements contenant comme information le nom d'une personne. Chaque enregistrement est donc constitué d'une seule rubrique, à savoir, le nom de la personne.



L'utilisateur doit entrer au clavier le nom du fichier, le nombre de personnes et les noms des personnes. Le programme se chargera de créer le fichier correspondant sur disque dur ou sur disquette.

Après avoir écrit et fermé le fichier, le programme va rouvrir le même fichier en lecture et afficher son contenu, sans utiliser le nombre d'enregistrements introduit dans la première partie.

Solution en langage algorithmique

```
programme PERSONNEL
  chaîne NOM_FICHIER, NOM_PERS
  entier C, _NB_ENREG

  (* Première partie :
    Créer et remplir le fichier *)
  écrire "Entrez le nom du fichier à créer : "
  lire NOM_FICHIER
  ouvrir NOM_FICHIER en écriture
  écrire "Nombre d'enregistrements à créer : "
  lire NB_ENREG
  en C ranger 0
  tant que (C<NB_ENREG) faire
  | écrire "Entrez le nom de la personne : "
  | lire NOM_PERS
  | écrire NOM_FICHIER:NOM_PERS
  | en C ranger C+1
  ftant (* C=NB_ENREG *)
  fermer NOM_FICHIER

  (* Deuxième partie :
    Lire et afficher le contenu du fichier *)
  ouvrir NOM_FICHIER en lecture
  en C ranger 0
  tant que non(finfichier(NOM_FICHIER)) faire
  | lire NOM_FICHIER:NOM_PERS
  | écrire "NOM : ",NOM_PERS
  | en C ranger C+1
  ftant
  fermer NOM_FICHIER
fprogramme (* fin PERSONNEL *)
```

Solution en langage C

```

#include <stdio.h>

main()
{
    FILE *P_FICHIER; /* pointeur sur FILE */
    char NOM_FICHIER[30], NOM_PERS[30];
    int C,NB_ENREG;

    /* Première partie :
       Créer et remplir le fichier */
    printf("Entrez le nom du fichier à créer : ");
    scanf("%s", NOM_FICHIER);
    P_FICHIER = fopen(NOM_FICHIER, "w"); /* write */
    printf("Nombre d'enregistrements à créer : ");
    scanf("%d", &NB_ENREG);
    C = 0;
    while (C<NB_ENREG)
    {
        printf("Entrez le nom de la personne : ");
        scanf("%s", NOM_PERS);
        fprintf(P_FICHIER, "%s\n", NOM_PERS);
        C++;
    }
    fclose(P_FICHIER);

    /* Deuxième partie :
       Lire et afficher le contenu du fichier */
    P_FICHIER = fopen(NOM_FICHIER, "r"); /* read */
    C = 0;
    while (!feof(P_FICHIER))
    {
        fscanf(P_FICHIER, "%s\n", NOM_PERS);
        printf("NOM : %s\n", NOM_PERS);
        C++;
    }
    fclose(P_FICHIER);
    return 0;
}

```

12.6 Ouvrir et fermer des fichiers séquentiels

Avant de créer ou de lire un fichier, nous devons informer le système de cette intention pour qu'il puisse réserver la mémoire pour la zone d'échange et initialiser les informations nécessaires à l'accès du fichier. Nous parlons alors de l'*ouverture* d'un fichier.

Après avoir terminé la manipulation du fichier, nous devons vider la mémoire tampon et libérer l'espace en mémoire que nous avons occupé pendant le traitement. Nous parlons alors de la *fermeture* du fichier.

L'ouverture et la fermeture de fichiers se font à l'aide des fonctions **fopen** et **fclose** définies dans la bibliothèque standard `<stdio>`.

Ouvrir un Fichier Séquentiel

Ouvrir un fichier en C - fopen:

Lors de l'ouverture d'un fichier avec **fopen**, le système s'occupe de la réservation de la mémoire tampon dans la mémoire centrale et génère les informations pour un nouvel élément du type **FILE**. L'adresse de ce bloc est retournée comme résultat si l'ouverture s'est déroulée avec succès. La commande **fopen** peut ouvrir des fichiers en écriture ou en lecture en dépendance de son deuxième paramètre ("r" ou "w") :

```
<FP> = fopen ( <Nom> , "w" );
```

ou bien

```
<FP> = fopen ( <Nom> , "r" );
```

- **<Nom>** est une chaîne de caractères constante ou une variable de type chaîne qui représente le nom du fichier sur le médium de stockage,
- le **deuxième argument** détermine le mode d'accès au fichier:

"w" pour 'ouverture en écriture' - *write* -

"r" pour 'ouverture en lecture' - *read* -

- **<FP>** est un pointeur du type **FILE*** qui sera relié au fichier sur le médium de stockage. Dans la suite du programme, il faut utiliser **<FP>** au lieu de **<Nom>** pour référencer le fichier.
- **<FP>** doit être déclaré comme:

```
FILE *FP;
```

Le résultat de fopen:

Si le fichier a pu être ouvert avec succès, **fopen** fournit l'adresse d'un nouveau bloc du type **FILE**. En général, la valeur de cette adresse ne nous intéresse pas; elle est simplement affectée à un pointeur **<FP>** du type **FILE*** que nous utiliserons ensuite pour accéder au fichier.

À l'apparition d'une erreur lors de l'ouverture du fichier, **fopen** fournit la valeur numérique zéro qui est souvent utilisée dans une expression conditionnelle pour assurer que le traitement ne continue pas avec un fichier non ouvert.

Ouverture en écriture:

Dans le cas de la création d'un nouveau fichier, le nom du fichier est ajouté au répertoire du médium de stockage et la tête de lecture/écriture est positionnée sur un espace libre du médium.

- Si un fichier **existant** est ouvert en écriture, alors son contenu est perdu.
- Si un fichier **non existant** est ouvert en écriture, alors il est créé automatiquement.
- Si la création du fichier est impossible alors **fopen** indique une erreur en retournant la valeur zéro.

Autres possibilités d'erreurs signalées par un résultat nul:

- chemin d'accès non valide,
- pas de disque/bande dans le lecteur,
- essai d'écrire sur un médium protégé contre l'écriture,
- ...

Ouverture en lecture:

Dans le cas de la lecture d'un fichier existant, le nom du fichier doit être retrouvé dans le répertoire du médium et la tête de lecture/écriture est placée sur le premier enregistrement de ce fichier.

Possibilités d'erreurs signalées par un résultat nul:

- essai d'ouvrir un fichier non existant,
- essai d'ouvrir un fichier sans autorisation d'accès,
- essai d'ouvrir un fichier protégé contre la lecture,
- ...

Remarque: Si un fichier n'a pas pu être ouvert avec succès, (résultat NUL), un code d'erreur est placé dans la variable **errno**. Ce code désigne plus exactement la nature de l'erreur survenue. Les codes d'erreurs sont définis dans **<errno.h>**.

- L'appel de la fonction **strerror(errno)** retourne un pointeur sur la chaîne de caractères qui décrit l'erreur dans **errno**.
- L'appel de la fonction **perror(s)** affiche la chaîne **s** et le message d'erreur qui est défini pour l'erreur dans **errno**.

Fermer un Fichier Séquentiel

Fermer un fichier en langage C:

```
fclose( <FP> );
```

<FP> est un pointeur du type **FILE*** relié au nom du fichier que l'on désire fermer.

La fonction **fclose** provoque le contraire de **fopen**:

- Si le fichier a été ouvert en écriture, alors les données non écrites de la mémoire tampon sont écrites et les données supplémentaires (longueur du fichier, date et heure de sa création) sont ajoutées dans le répertoire du médium de stockage.
- Si le fichier a été ouvert en lecture, alors les données non lues de la mémoire tampon sont simplement 'jetées'.
- La mémoire tampon est ensuite libérée et la liaison entre le pointeur sur **FILE** et le nom du fichier correspondant est annulée.
- Après **fclose()** le pointeur <FP> est invalide. Des erreurs graves pourraient donc survenir si ce pointeur est utilisé par la suite!

Exemples: Ouvrir et Fermer des Fichiers en Pratique

En pratique, il faut contrôler si l'ouverture d'un fichier a été accomplie avec succès avant de continuer les traitements. Pour le cas d'une erreur, nous allons envisager deux réactions différentes:

Répéter l'essai jusqu'à l'ouverture correcte du fichier

```
#include <stdio.h>
main()
{
    FILE *P_FICHIER;      /* pointeur sur FILE */
    char NOM_FICHIER[30]; /* nom du fichier */
    . . .

    do
    {
        printf("Entrez le nom du fichier : ");
        scanf("%s", NOM_FICHIER);
        P_FICHIER = fopen(NOM_FICHIER, "w");
        /* ou bien */
        /* P_FICHIER = fopen(NOM_FICHIER, "r"); */
        if (!P_FICHIER)
            printf("\aERREUR: Impossible d'ouvrir "
                   "le fichier: %s.\n", NOM_FICHIER);
    }
    while (!P_FICHIER);

    . . .

    fclose(P_FICHIER);
    return 0;
}
```

Abandonner le programme en retournant un code d'erreur non nul - exit

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE *P_FICHIER;      /* pointeur sur FILE */
    char NOM_FICHIER[30]; /* nom du fichier */
    . . .

    printf("Entrez le nom du fichier : ");
    scanf("%s", NOM_FICHIER);
    P_FICHIER = fopen(NOM_FICHIER, "w");
    /* ou bien */
    /* P_FICHIER = fopen(NOM_FICHIER, "r"); */
    if (!P_FICHIER)
    {
        printf("\aERREUR: Impossible d'ouvrir "
               "le fichier: %s.\n", NOM_FICHIER);
        exit(-1); /* Abandonner le programme en */
    }           /* retournant le code d'erreur -1 */

    . . .

    fclose(P_FICHIER);
    return 0;
}

```

12.7 Lire et écrire dans des fichiers séquentiels

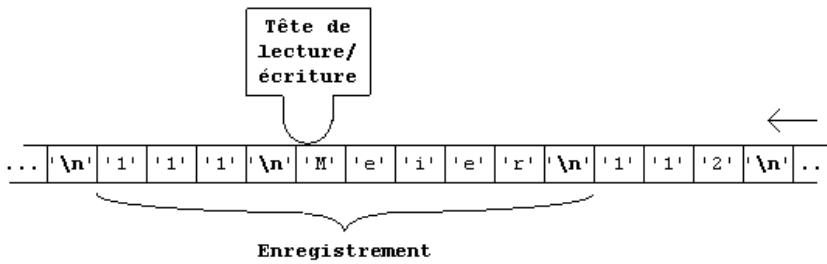
Fichiers texte

Les fichiers que nous employons dans ce manuel sont des fichiers texte, c.-à-d. toutes les informations dans les fichiers sont mémorisées sous forme de chaînes de caractères et sont organisées en lignes. Même les valeurs numériques (types **int**, **float**, **double**, ...) sont stockées comme chaînes de caractères.

Pour l'écriture et la lecture des fichiers, nous allons utiliser les fonctions standard **fprintf**, **fscanf**, **fputc** et **fgetc** qui correspondent à **printf**, **scanf**, **putchar** et **getchar** si nous indiquons *stdout* respectivement *stdin* comme fichiers de sortie ou d'entrée.

12.8 Traitement par enregistrements

Les fichiers texte sont généralement organisés en lignes, c.-à-d. la fin d'une information dans le fichier est marquée par le symbole '\n':



Attention !

Pour pouvoir lire correctement les enregistrements dans un fichier séquentiel, le programmeur doit connaître l'ordre des différentes rubriques (champs) à l'intérieur des enregistrements.

a) Ecrire une information dans un fichier séquentiel

Ecrire dans un fichier séquentiel en langage algorithmique

```
écrire <Nom>:<Expr1>
écrire <Nom>:<Expr2>
...
écrire <Nom>:<ExprN>
```

* <Nom> est une chaîne de caractères constante ou une variable de type chaîne qui représente le nom du fichier dans lequel on veut écrire.

* <Expr1>, <Expr2>, ..., <ExprN> représentent les rubriques qui forment un enregistrement et dont les valeurs respectives sont écrites dans le fichier.

Ecrire dans un fichier séquentiel en langage C - fprintf

```
fprintf( <FP>, "<Form1>\n", <Expr1>);
fprintf( <FP>, "<Form2>\n", <Expr2>);
...
fprintf( <FP>, "<FormN>\n", <ExprN>);
```

ou bien

```
fprintf(<FP>,"<Form1>\n<Form2>\n...\n<FormN>\n", <Expr1>, <Expr2>, ... , <ExprN>);
```

* <FP> est un pointeur du type **FILE*** qui est relié au nom du fichier cible.

* <Expr1>, <Expr2>, ..., <ExprN> représentent les rubriques qui forment un enregistrement et dont les valeurs respectives sont écrites dans le fichier.

* <Form1>, <Form2>, ..., <FormN> représentent les spécificateurs de format pour l'écriture des différentes rubriques (voir chapitre 4.3.).

Remarque

L'instruction

```
fprintf(stdout, "Bonjour\n");
```

est identique à

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

Attention !

Notez que **fprintf** (et **printf**) écrit toutes les chaînes de caractères *sans le symbole de fin de chaîne '\0'*. Dans les fichiers texte, il faut ajouter le symbole de fin de ligne '\n' pour séparer les données.

b) Lire une information dans un fichier séquentiel

Lire dans un fichier séquentiel en langage algorithmique

```
lire <Nom>:<Var1>
lire <Nom>:<Var2>
...
lire <Nom>:<VarN>
```

* <Nom> est une chaîne de caractères constante ou une variable de type chaîne qui représente le nom du fichier duquel on veut lire.

* <Var1>, <Var2>, ... , <VarN> représentent les variables qui vont recevoir les valeurs des différentes rubriques d'un enregistrement lu dans le fichier.

Lire dans un fichier séquentiel en langage C - fscanf

```
fscanf( <FP>, "<Form1>\n", <Adr1>);  
fscanf( <FP>, "<Form2>\n", <Adr2>);  
...  
fscanf( <FP>, "<FormN>\n", <AdrN>);
```

ou bien

```
fscanf(<FP>,"<Form1>\n<Form2>\n...\n<FormN>\n", <Adr1>, <Adr2>, ... , <AdrN>);
```

* <FP> est un pointeur du type **FILE*** qui est relié au nom du fichier à lire.

* <Adr1>, <Adr2>, ... , <AdrN> représentent les adresses des variables qui vont recevoir les valeurs des différentes rubriques d'un enregistrement lu dans le fichier.

* <Form1>, <Form2>, ... , <FormN> représentent les spécificateurs de format pour la lecture des différentes rubriques (voir chapitre 4.4.).

Remarque

L'instruction

```
fscanf(stdin, "%d\n", &N);
```

est identique à

```
scanf("%d\n", &N);
```

Attention !

Pour les fonctions **scanf** et **fscanf** tous les signes d'espacement sont équivalents comme séparateurs. En conséquence, à l'aide de **fscanf**, il nous sera impossible de lire toute une phrase dans laquelle les mots sont séparés par des espaces.

12.9 Traitement par caractères

La manipulation de fichiers avec les instructions **fprintf** et **fscanf** n'est pas assez flexible pour manipuler de façon confortable des textes écrits. Il est alors avantageux de traiter le fichier séquentiellement caractère par caractère.

a) Ecrire un caractère dans un fichier séquentiel - fputc

```
fputc( <C> , <FP> );
```

fputc transfère le caractère indiqué par <C> dans le fichier référencé par <FP> et avance la position de la tête de lecture/écriture au caractère suivant.

* représente un caractère (valeur numérique de 0 à 255) ou le symbole de fin de fichier **EOF** (voir 11.5.3.).

* <FP> est un pointeur du type **FILE*** qui est relié au nom du fichier cible.

Remarque

L'instruction

```
fputc('a', stdout);
```

est identique à

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

b) Lire un caractère dans un fichier séquentiel - fgetc

```
<C> = fgetc( <FP> );
```

fgetc fournit comme résultat le prochain caractère du fichier référencé par <FP> et avance la position de la tête de lecture/écriture au caractère suivant. A la fin du fichier, **fgetc** retourne **EOF** (voir 11.5.3.).

<C> représente une variable du type **int** qui peut accepter une valeur numérique de 0 à 255 ou le symbole de fin de fichier **EOF**.

<FP> est un pointeur du type **FILE*** qui est relié au nom du fichier à lire.

Remarque

L'instruction

```
C = fgetc(stdin);
```

est identique à

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

12.10 Détection de la Fin d'un Fichier Séquentiel

Lors de la fermeture d'un fichier ouvert en écriture, la fin du fichier est marquée automatiquement par le symbole de fin de fichier **EOF** (*End Of File*). Lors de la lecture d'un fichier, la fonction **feof(<FP>)** nous permet de détecter la fin du fichier:

Détection de la fin d'un fichier en langage C - feof

feof(<FP>);

feof retourne une valeur différente de zéro, si la tête de lecture du fichier référencé par <FP> est arrivée à la fin du fichier; sinon la valeur du résultat est zéro.

- <FP> est un pointeur du type **FILE*** qui est relié au nom du fichier à lire.

ATTENTION!!! Pour que la fonction **feof** détecte correctement la fin du fichier, il faut qu'après la lecture de la dernière donnée du fichier, la tête de lecture arrive jusqu'à la position de la marque **EOF**. Nous obtenons cet effet seulement si nous terminons aussi la chaîne de format de **fscanf** par un retour à la ligne '\n' (ou par un autre signe d'espacement).

Exemple: Une boucle de lecture typique pour lire les enregistrements d'un fichier séquentiel référencé par un pointeur FP peut avoir la forme suivante:

```
while (!feof(FP))
{
    fscanf(FP, "%s\n ... \n", NOM, ... );
    . . .
}
```

Exemple: Le programme suivant lit et affiche le fichier "C:\AUTOEXEC.BAT" en le parcourant caractère par caractère:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE *FP;
    FP = fopen("C:\\AUTOEXEC.BAT", "r");
    if (!FP)
    {
        printf("Impossible d'ouvrir le fichier\n");
        exit(-1);
    }
    while (!feof(FP))
        putchar(fgetc(FP));
    fclose(FP);
    return 0;
}
```

Dans une chaîne de caractères constante, il faut indiquer le symbole '\\' (back-slash) par '\\\\', pour qu'il ne soit pas confondu avec le début d'une séquence d'échappement (p.ex: \n, \t, \a, ...).

12.11 Résumé

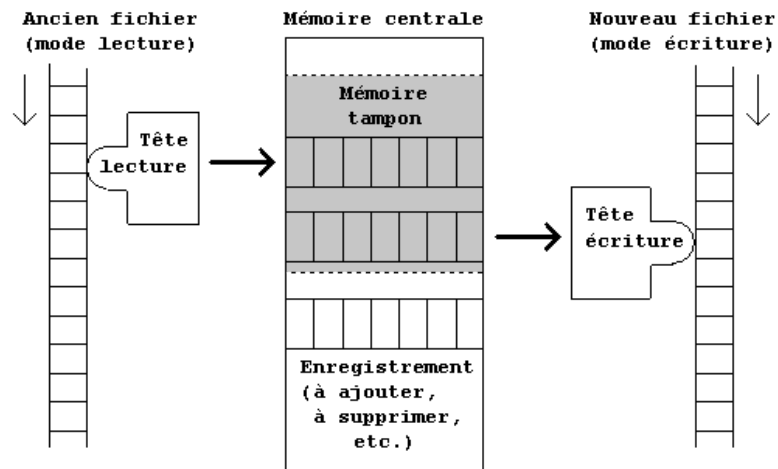
	C
Ouverture en écriture	<code><FP> = fopen(<Nom>,"w");</code>
Ouverture en lecture	<code><FP> = fopen(<Nom>,"r");</code>
Fermeture	<code>fclose(<FP>);</code>
Fonction fin de fichier	<code>feof(<FP>)</code>
Ecriture	<code>fprintf(<FP>,"...",<Adr>);</code> <code>fputc(<C>, <FP>);</code>
Lecture	<code>fscanf(<FP>,"...",<Adr>);</code> <code><C> = fgetc(<FP>);</code>

13 Mise à jour d'un fichier séquentiel en C

Dans ce chapitre, nous allons résoudre les problèmes standards sur les fichiers, à savoir:

- l'ajoute d'un enregistrement à un fichier
- la suppression d'un enregistrement dans un fichier
- la modification d'un enregistrement dans un fichier

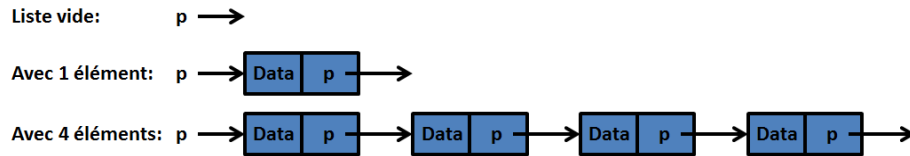
Comme il est impossible de lire et d'écrire en même temps dans un fichier séquentiel, les modifications doivent se faire à l'aide d'un fichier supplémentaire. Nous travaillons donc typiquement avec au moins deux fichiers: l'ancien fichier ouvert en lecture et le nouveau fichier ouvert en écriture.



14 Listes chaînées

Les listes chaînées permettent de créer des structures de stockage de l'information capables de grandir à mesure qu'on rajoute des données.

Le principe est simple: les données sont stockées dans des structures qui représentent chacune un maillon d'une chaîne: chaque maillon stocke un peu d'information, et contient un pointeur sur le maillon suivant (on parle de liste simplement chaînée). Quand chaque maillon stocke également un pointeur sur le maillon précédent, on parle de liste doublement chaînée. En général, on a accès à une liste chaînée car on dispose d'un pointeur sur le premier élément de la liste.



En terme de programmation, chaque élément de la liste est déclaré comme une structure. Une des listes les plus simple permet simplement de stocker des nombres, mais nous verrons que ces listes peuvent stocker des contenus beaucoup plus variés.

Exercice pour démarrer: Une liste simple

Créez un Projet Dev C++ appelé **SimpleList** avec un fichier **fonctions.h** et un **fonctions.c**.

- Dans le fichier **fonction.h**, définissez une structure nommée **Maillon**, qui contient une entier **nombre** et un pointeur sur un Maillon nommé **suivant**.

```
5  typedef struct Maillon{
6
7      int nombre;
8      struct Maillon * suivant;
9
10 } Maillon;
```

- Déclarez et créez une fonction **nouveauMaillon** qui reçoit un entier **nombre** et renvoie un pointeur sur un Maillon qui stocke ce nombre et a son pointeur suivant initialisé à 0.

```
Maillon * nouveauMaillon(int nombre){

    Maillon * m = /* ??? */
    return m;

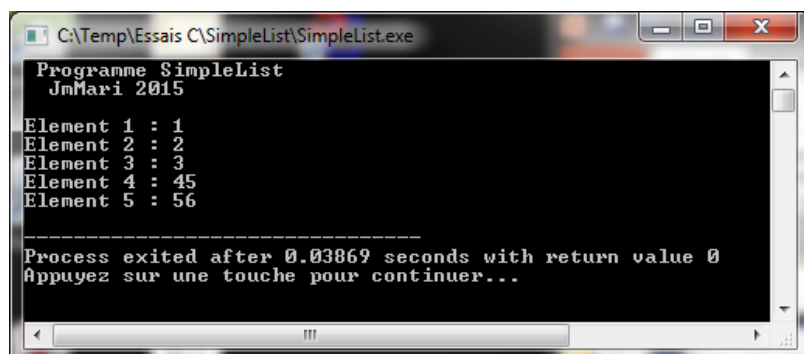
}
```

- Déclarez et créez une fonction **ajoutNombre** qui reçoit un pointeur sur un **Maillon** nommé **liste** et un entier **nombre**, ajoute un nouveau Maillon contenant le **nombre** à la fin de la liste et renvoie le pointeur **liste**.

```
Maillon * ajoutNombre(Maillon * liste, int nombre){

    if(liste == NULL){
        return /* ??? */
    }else{
        Maillon * m = /* ??? */
        while(m->suivant != NULL){
            /* ?? on avance d'un maillon ?? */
        }
        m->suivant = /* ?? on ajoute le nouveau nombre dans un maillon ?? */
        return liste;
    }
}
```

- Déclarez et créez une fonction **afficheListe** qui reçoit un pointeur sur un **Maillon** nommé **liste** et affiche tous les éléments de la liste chaînée:



```
void afficheListe(Maillon * liste){

    int i = 1;
    /* ??? */
    while(m != NULL){
        printf("Element %d : %d\n", i, m->nombre);
        /* ??? */
        /* ??? */
    }
}
```

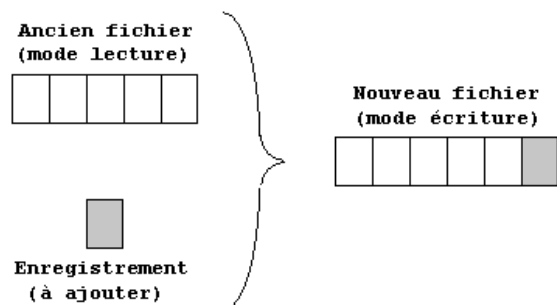
- Complétez le programme principal pour obtenir la sortie donnée à la question précédente.

14.1 Ajouter un enregistrement à un fichier

Nous pouvons ajouter le nouvel enregistrement à différentes positions dans le fichier:

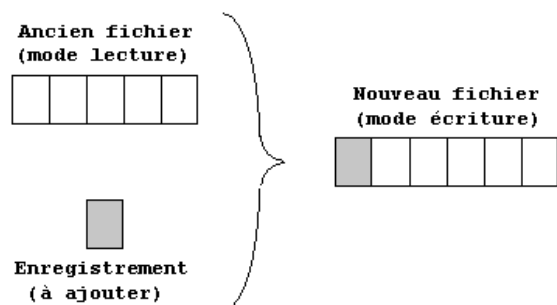
a) Ajoute à la fin du fichier

L'ancien fichier est entièrement copié dans le nouveau fichier, suivi du nouvel enregistrement.



b) Ajoute au début du fichier

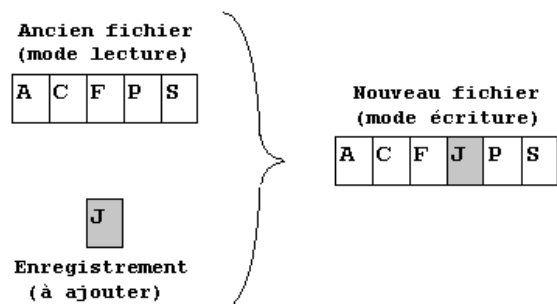
L'ancien fichier est copié derrière le nouvel enregistrement qui est écrit en premier lieu.



c) Insertion dans un fichier trié relativement à une rubrique commune des enregistrements

Le nouveau fichier est créé en trois étapes:

- copier les enregistrements de l'ancien fichier qui précèdent le nouvel enregistrement,
- écrire le nouvel enregistrement,
- copier le reste des enregistrements de l'ancien fichier.



Le programme suivant effectue l'insertion d'un enregistrement à introduire au clavier dans un fichier trié selon la seule rubrique de ses enregistrements: le nom d'une personne. Le programme inclut en même temps les solutions aux deux problèmes précédents. La comparaison lexicographique des noms des personnes se fait à l'aide de la fonction **strcmp**.

Solution en C

```

#include <stdio.h>
#include <string.h>

main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[30], NOUVEAU[30];
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char NOM_PERS[30], NOM_AJOUT[30];
    int TROUVE;

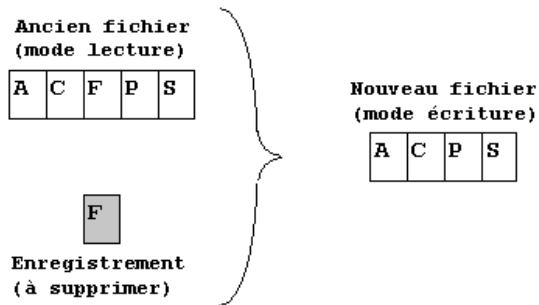
    /* Ouverture de l'ancien fichier en lecture */
    do
    {
        printf("Nom de l'ancien fichier : ");
        scanf("%s", ANCIEN);
        INFILE = fopen(ANCIEN, "r");
        if (!INFILE)
            printf("\aERREUR: Impossible d'ouvrir "
                "le fichier: %s.\n", ANCIEN);
    }
    while (!INFILE);
    /* Ouverture du nouveau fichier en écriture */
    do
    {
        printf("Nom du nouveau fichier : ");
        scanf("%s", NOUVEAU);
        OUTFILE = fopen(NOUVEAU, "w");
        if (!OUTFILE)
            printf("\aERREUR: Impossible d'ouvrir "
                "le fichier: %s.\n", NOUVEAU);
    }
    while (!OUTFILE);
    /* Saisie de l'enregistrement à insérer */
    printf("Enregistrement à insérer : ");
    scanf("%s", NOM_AJOUT);

    /* Traitement */
    TROUVE = 0;
    /* Copie des enregistrements dont le nom */
    /* précède lexicogr. celui à insérer.*/
    while (!feof(INFILE) && !TROUVE)
    {
        fscanf(INFILE, "%s\n", NOM_PERS);
        if (strcmp(NOM_PERS, NOM_AJOUT) > 0)
            TROUVE = 1;
        else
            fprintf(OUTFILE, "%s\n", NOM_PERS);
    }
    /* Ecriture du nouvel enregistrement, */
    fprintf(OUTFILE, "%s\n", NOM_AJOUT);
    /* suivi du dernier enregistrement lu. */
    if (TROUVE) fprintf(OUTFILE, "%s\n", NOM_PERS);
    /* Copie du reste des enregistrements */
    while (!feof(INFILE))
    {
        fscanf(INFILE, "%s\n", NOM_PERS);
        fprintf(OUTFILE, "%s\n", NOM_PERS);
    }
    /* Fermeture des fichiers */
    fclose(OUTFILE);
    fclose(INFILE);
    return 0;
}

```

14.2 Supprimer un enregistrement dans un fichier

Le nouveau fichier est créé en copiant tous les enregistrements de l'ancien fichier qui précèdent l'enregistrement à supprimer et tous ceux qui le suivent:



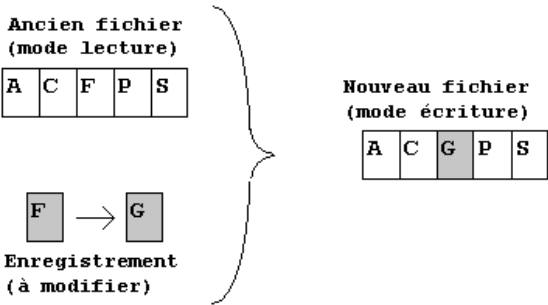
Solution en C

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

main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[30], NOUVEAU[30];
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char NOM_PERS[30], NOM_SUPPR[30];
    /* Ouverture de l'ancien fichier en lecture */
    do
    {
        printf("Nom de l'ancien fichier : ");
        scanf("%s", ANCIEN);
        INFILE = fopen(ANCIEN, "r");
        if (!INFILE)
            printf("\aERREUR: Impossible d'ouvrir "
                "le fichier: %s.\n", ANCIEN);
    }
    while (!INFILE);
    /* Ouverture du nouveau fichier en écriture */
    do
    {
        printf("Nom du nouveau fichier : ");
        scanf("%s", NOUVEAU);
        OUTFILE = fopen(NOUEAU, "w");
        if (!OUTFILE)
            printf("\aERREUR: Impossible d'ouvrir "
                "le fichier: %s.\n", NOUEAU);
    }
    while (!OUTFILE);
    /* Saisie de l'enregistrement à supprimer */
    printf("Enregistrement à supprimer : ");
    scanf("%s", NOM_SUPPR);
    /* Traitement */
    /* Copie de tous les enregistrements à */
    /* l'exception de celui à supprimer. */
    while (!feof(INFILE))
    {
        fscanf(INFILE, "%s\n", NOM_PERS);
        if (strcmp(NOM_PERS, NOM_SUPPR) != 0)
            fprintf(OUTFILE, "%s\n", NOM_PERS);
    }
    /* Fermeture des fichiers */
    fclose(OUTFILE);
    fclose(INFILE);
    return 0;
}
```

14.3 Modifier un enregistrement dans un fichier

Le nouveau fichier est créé de tous les enregistrements de l'ancien fichier qui précèdent l'enregistrement à modifier, de l'enregistrement modifié et de tous les enregistrements qui suivent l'enregistrement à modifier dans l'ancien fichier:



Solution en C

```

#include <stdio.h>
#include <string.h>

main()
{
    /* Déclarations : */
    /* Noms des fichiers et pointeurs de référence */
    char ANCIEN[30], NOUVEAU[30];
    FILE *INFILE, *OUTFILE;
    /* Autres variables */
    char NOM_PERS[30], NOM_MODIF[30], NOM_NOUV[30];
    /* Ouverture de l'ancien fichier en lecture */
    do
    {
        printf("Nom de l'ancien fichier : ");
        scanf("%s", ANCIEN);
        INFILE = fopen(ANCIEN, "r");
        if (!INFILE)
            printf("\aERREUR: Impossible d'ouvrir "
                "le fichier: %s.\n", ANCIEN);
    }
    while (!INFILE);

    /* Ouverture du nouveau fichier en écriture */
    do
    {
        printf("Nom du nouveau fichier : ");
        scanf("%s", NOUVEAU);
        OUTFILE = fopen(NOUEAU, "w");
        if (!OUTFILE)
            printf("\aERREUR: Impossible d'ouvrir "
                "le fichier: %s.\n", NOUEAU);
    }
    while (!OUTFILE);

    /* Saisie de l'enregistrement à modifier, */
    printf("Enregistrement à modifier : ");
    scanf("%s", NOM_MODIF);
    /* et de sa nouvelle valeur. */
    printf("Enregistrement nouveau : ");
    scanf("%s", NOM_NOUV);
    /* Traitement */
    /* Copie de tous les enregistrements en */
    /* remplaçant l'enregistrement à modifier */
    /* par sa nouvelle valeur. */
    while (!feof(INFILE))
    {
        fscanf(INFILE, "%s\n", NOM_PERS);
        if (strcmp(NOM_PERS, NOM_MODIF) == 0)
            fprintf(OUTFILE, "%s\n", NOM_NOUV);
        else
            fprintf(OUTFILE, "%s\n", NOM_PERS);
    }
    /* Fermeture des fichiers */
    fclose(OUTFILE);
    fclose(INFILE);
    return 0;
}

```

14.4 Exercice : Notes d'un étudiant

Exercice: Vous allez écrire un programme qui gère les notes d'un étudiant:

1) écrire dans un fichier "**notes.txt**" les notes suivantes que vous demanderez à l'utilisateur

MathET, MathCC, InfoET, InfoCC, Anglais

Les champs seront séparées par un caractère retour à la ligne '**\n**'

2) allez ouvrir le fichier "**notes.txt**" sur le disque dur avec un éditeur de texte comme **notepad**, vérifiez que le fichier correspond aux informations entrées par l'utilisateur

3) ouvrir ce fichier en lecture, récupérez toutes les notes, calculez et affichez la moyenne

4) ouvrir ce fichier en lecture, demandez pour chaque note à l'utilisateur s'il veut la modifier.

réenregistrez la nouvelle version du fichier avec les notes modifiées.

14.5 Exercice : Notes de plusieurs étudiants

Exercice: Vous allez écrire un programme qui gère les notes des étudiants d'une promotion.

1) déclarez une structure étudiant qui comporte les champs:

Nom;Prénom;MathET;MathC;InfoET;InfoCC;Anglais;

2) demandez à l'utilisateur le nom du fichier qu'il veut créer et combien il veut entrer d'étudiants dans ce fichier.

demandez lui d'entrer toutes les informations correspondantes pour chaque étudiant.

3) vérifiez sur le disque avec un éditeur de texte que le fichier contient bien toutes les informations

4) réouvrez le fichier et demandez à l'utilisateur s'il veut écrire de nouveaux étudiants. si oui, demandez les informations de ces nouveaux étudiants et sauvez les à la fin du fichier.

5) faites un affichage des informations du fichier à l'écran.

ps: il n'est pas question de stocker en mémoire vive tous les étudiants.

15 Fichiers Avancés

Introduction

Les **entrées/sorties (E/S)** ne font pas partie du langage C car ces opérations sont dépendantes du système. Néanmoins puisqu'il s'agit de tâches habituelles, sa bibliothèque standard est fournie avec des fonctions permettant de réaliser ces opérations de manière portable. Ces fonctions sont principalement déclarées dans le fichier **stdio.h**. Certaines ont déjà été présentées dans les tutoriels précédents, de même que quelques concepts relatifs aux entrées/sorties en langage C. Aucun rappel ne sera fait, sauf sur certains concepts jugés importants.

Les entrées/sorties en langage C se font par l'intermédiaire d'entités logiques, appelés **flux**, qui représentent des objets externes au programme, appelés **fichiers**. En langage C, un fichier n'est donc pas nécessairement un fichier sur disque (ou périphérique de stockage pour être plus précis). Un fichier désigne en fait aussi bien un fichier sur disque (évidemment) qu'un périphérique physique ou un tube par exemple. Selon la manière dont on veut réaliser les opérations d'E/S sur le fichier, qui se font à travers un flux, on distingue deux grandes catégories de flux à savoir **les flux de texte et les flux binaires**.

Les flux de texte sont parfaits pour manipuler des données présentées sous forme de texte. Un flux de texte est organisé en **lignes**. En langage C, une ligne est une suite de caractères terminée par le **caractère de fin de ligne** (inclus) : '\n'. Malheureusement, ce n'est pas forcément le cas pour le système sous-jacent. Sous Windows par exemple, **la marque de fin de ligne** est par défaut la combinaison de deux caractères : **CR** (Carriage Return) et **LF** (Line Feed) soit '\r' et '\n' (notez bien que c'est CR/LF c'est-à-dire CR suivi de LF pas LF suivi de CR). Sous UNIX, c'est tout simplement '\n'. On se demande alors comment on va pouvoir lire ou écrire dans un fichier, à travers un flux de texte, de manière portable. Et bien c'est beaucoup plus simple que ce à quoi vous-vous attendiez : lorsqu'on effectue une opération d'entrée/sortie sur un flux de texte, les données seront lues/écrites de façon à ce qu'elles correspondent à la manière dont elles doivent être représentées et non caractère pour caractère. C'est-à-dire par exemple que, dans une implémentation où la fin de ligne est provoquée par la combinaison des caractères CR et LF, l'écriture de '\n' sur un flux de texte va provoquer l'écriture effective des caractères '\r' et '\n' dans le fichier associé. Sur un flux binaire les données sont lues ou écrites dans le fichier caractère pour caractère.

15.1 Les Fichiers sur Disque

La communication avec une ressource externe (un modem, une imprimante, une console, un périphérique de stockage, etc.) nécessite un protocole de communication (qui peut être texte ou binaire, simple ou complexe, ...) spécifique de cette ressource et qui n'a donc rien à voir le langage C. La norme définit tout simplement les fonctions permettant d'effectuer les entrées/sorties vers un fichier sans pour autant définir la notion de matériel ou même de fichier sur disque afin de garantir la portabilité du langage (ou plus précisément de la bibliothèque). Cependant, afin de nous fixer les idées, nous n'hésiterons pas à faire appel à ces notions dans les explications.

Les fichiers sur disque servent à stocker des informations. On peut « naviguer » à l'intérieur d'un tel fichier à l'aide de **fonctions de positionnement** (que nous verrons un peu plus loin). A chaque instant, un « pointeur » indique la **position courante** dans le fichier. Ce pointeur se déplace, à quelques exceptions près, après chaque opération de lecture, d'écriture ou appel d'une fonction de positionnement par exemple. Bien entendu, cette notion de position n'est pas une spécificité exclusive des fichiers sur disque. D'autres types de fichiers peuvent très bien avoir une structure similaire.

15.2 Les Messages d'Erreurs

En langage C il est coutume de retourner un entier même quand une fonction n'est censée retourner aucune valeur. Cela permet au programme de contrôler les erreurs et même d'en connaître la cause. Certaines fonctions comme `malloc` par exemple retournent cependant une adresse, `NULL` en cas d'erreur. Pour fournir d'amples informations quant à la cause de l'erreur, ces fonctions placent alors une valeur dans une variable globale appelée **`errno`** (en fait la norme n'impose pas qu'`errno` doit être forcément une variable globale !), cette valeur bien entendu est à priori dépendante de l'implémentation. Ces valeurs doivent être déclarées dans le fichier **`errno.h`**. Par exemple, une implémentation peut définir un numéro d'erreur `ENOMEM` qui sera placée dans `errno` lorsqu'un appel à `malloc` a échoué car le système manque de mémoire.

La fonction **`strerror`**, déclarée dans **`string.h`**, permet de récupérer une chaîne à priori définie par l'implémentation décrivant l'erreur correspondant au numéro d'erreur passé en argument. La fonction **`perror`**, déclarée dans **`stdio.h`**, quant à elle utilise cette chaîne, plus une chaîne fournie en argument, pour afficher la description d'une erreur. Le texte sera affiché sur l'erreur standard (`stderr`). Pour résumer :

```
perror(msg);
```

est équivalent à :

```
fprintf(stderr, "%s: %s\n", msg, strerror(errno));
```

Si nous n'avons pas parlé de ces fonctions auparavant, c'est parce que nous n'en avons pas vraiment jusqu'ici besoin. A partir de maintenant, elles vont être très utiles car les fonctions d'entrées/sorties sont sujettes à de très nombreuses sortes d'erreurs : fichier non trouvé, espace sur disque insuffisant, on n'a pas les permissions nécessaires pour lire ou écrire dans le fichier, ...

15.3 Ouverture d'un fichier

Toute opération d'E/S dans un fichier commence par l'**ouverture** du fichier. Lorsqu'un fichier est ouvert, un **flux** lui est associé. Ce flux est représenté par un pointeur vers un objet de type **FILE**.

Pendant l'ouverture d'un fichier, on doit spécifier comment on désire l'ouvrir : en lecture (c'est-à-dire qu'on va lire dans le fichier), en écriture (pour écrire), ou en lecture et écriture. La fonction **fopen** :

```
FILE * fopen(const char * filename, const char * mode);
```

permet d'ouvrir un fichier dont le nom est spécifié par l'argument `filename` selon le mode, spécifié par l'argument `mode`, dans lequel on souhaite ouvrir le fichier. Elle retourne l'adresse d'un objet de type `FILE` qui représente le fichier à l'intérieur du programme. En cas d'erreur, `NULL` est retourné et une valeur indiquant la cause de l'erreur est placée dans `errno`.

En pratique, dans le cas d'un fichier sur disque, l'argument `nom` peut être aussi bien un chemin complet qu'un chemin relatif. Notez bien que les notions de chemin, répertoire (qui inclut également les notions de répertoire courant, parent, etc.), ... sont dépendantes du système, elles ne font pas partie du langage C. La plupart du temps, le répertoire courant est par défaut celui dans lequel se trouve le programme mais, si le système le permet, il est possible de spécifier un répertoire différent.

Fondamentalement, les valeurs suivantes peuvent être utilisées dans l'argument `mode` :

- **"r"** : ouvrir le fichier en **lecture**. Le fichier spécifié doit déjà exister.
- **"w"** : ouvrir le fichier en **écriture**. S'il n'existe pas, il sera créé. S'il existe déjà, son ancien contenu sera effacé.
- **"a"** : ouvrir le fichier en **mode ajout**, qui est un mode dans lequel toutes les opérations d'écriture dans le fichier se feront à la fin du fichier. S'il n'existe pas, il sera créé.

Quel que soit le cas, le fichier sera associé à un flux de texte. Pour spécifier qu'on veut ouvrir le fichier en tant que fichier binaire, il suffit d'ajouter le suffixe **"b"** (c'est-à-dire **"rb"**, **"wb"** ou **"ab"**).

Sauf dans le cas du mode **"ab"** et dérivés, dans lequel le pointeur pourrait, selon l'implémentation, être positionné à la fin du fichier, il sera positionné au début du fichier. On peut également ajouter un **"+"** (par exemple : **"wb+"**) dans l'argument `mode` ce qui aura pour effet de permettre d'effectuer aussi bien des opérations de lecture que d'écriture sur le fichier. On dit alors que le fichier est ouvert en **mode mise à jour**. Il y a cependant une remarque très importante concernant les fichiers ouverts en mode mise à jour :

- avant d'effectuer une opération de lecture juste après une opération d'écriture, il faut tout d'abord appeler `fflush` ou une fonction de positionnement
- avant d'effectuer une opération d'écriture juste après une opération de lecture, il faut d'abord appeler une fonction de positionnement, à moins d'avoir atteint la fin du fichier
- dans de nombreuses implémentations, tout fichier ouvert en mode mise à jour est supposé être un fichier binaire qu'on ait mis oui ou non la lettre **"b"**. Personnellement, je recommande donc de n'utiliser le mode mise à jour qu'avec les fichiers binaires.

Lorsqu'on n'en a plus besoin, il faut ensuite **fermer** le fichier :

```
int fclose(FILE * f);
```

Le programme suivant crée un fichier, `hello.txt`, pour y écrire ensuite une et une seule ligne : `Hello, world`.

```
#include <stdio.h>int main()
{
    FILE * f;

    f = fopen("hello.txt", "w");
    if (f != NULL)
    {
        fprintf(f, "Hello, world\n");
        fclose(f);
    }
    else
        perror("hello.txt");
    return 0;
}
```

En supposant que pour le système la fin de ligne est représentée par la combinaison des caractères CR et LF, ce programme est aussi équivalent au suivant :

```
#include <stdio.h>
int main()
{
    FILE * f;
    f = fopen("hello.txt", "wb");

    if (f != NULL)
    {
        fprintf(f, "Hello, world\r\n");
        fclose(f);
    }
    else
        perror("hello.txt");

    return 0;
}
```

15.4 Exemple : Copier un fichier

Il y a plusieurs moyens de réaliser la copie d'un fichier, le plus simple est peut-être de copier la source vers la destination octet par octet. Voici un programme qui réalise une telle copie :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char * saisir_chaine(char * lpBuffer, int buf_size);

int main()
{
    FILE * fsrc, * fdest;
    char source[FILENAME_MAX], dest[FILENAME_MAX];
    int c;

    printf("source : ");
    saisir_chaine(source, sizeof(source));
    fsrc = fopen(source, "rb");
    if (fsrc == NULL)
    {
        perror(source);
        exit(EXIT_FAILURE);
    }
    printf("dest : ");
    saisir_chaine(dest, sizeof(dest));

    fdest = fopen(dest, "wb");
    if (fdest == NULL)
    {
        perror(dest);
        exit(EXIT_FAILURE);
    }
    while ((c = getc(fsrc)) != EOF)
        putc(c, fdest);

    fclose(fdest);
    fclose(fsrc);
    return 0;
}

char * saisir_chaine(char * lpBuffer, int buf_size)
{
    char * p;

    fgets(lpBuffer, buf_size, stdin);

    if ((p = strchr(lpBuffer, '\n')) != NULL)
        *p = '\0';
    else
    {
        int c;
        do
            c = getchar();
        while (c != EOF && c != '\n');
    }

    return lpBuffer;
}
```

15.5 Les Erreurs d'Entrée/Sortie

Une **erreur d'E/S** est erreur qui peut se produire lors d'une tentative d'opération d'E/S, par exemple : fin de fichier atteinte, tentative d'écriture dans un fichier ouvert uniquement en lecture, tentative de lecture dans un fichier ouvert uniquement en lecture, etc.

La fonction feof :

```
int feof(FILE * f);
```

peut être appelé à n'importe quel moment pour connaître si l'on a atteint la fin du fichier. Je rappelle qu'un programme (du moins d'après les fonctions du C) ne peut affirmer que la fin d'un fichier a été atteinte qu'après avoir tenté de lire dans le fichier alors qu'on se trouve déjà à la fin c'est-à-dire derrière le dernier octet, pas juste après avoir lu le dernier octet, donc faites gaffe. Evidemment, une telle fonction ne sera utilisée que sur un fichier ouvert en lecture. Elle retourne VRAI si la fin de fichier a été atteinte et FAUX dans le cas contraire. Cette information est en fait maintenue par un **indicateur de fin de fichier** qui indique à tout moment si on a oui ou non déjà atteint la fin du fichier. Après un appel fructueux à une fonction de positionnement, l'indicateur de fin de fichier est remis à zéro.

La fonction ferror :

```
int ferror(FILE * f);
```

permet de connaître si une erreur s'est produite lors de la dernière opération d'E/S sur f. Cette information est maintenue en permanence par un **indicateur d'erreur**. Bien qu'une lecture à la fin d'un fichier soit également une erreur, elle n'est pas considérée par la fonction ferror car les indicateurs d'erreur et de fin de fichier sont des données bien distinctes.

Attention, une fois qu'une erreur s'est produite, l'indicateur d'erreur ne sera remis à zéro qu'après un appel à **clearerr**.

```
void clearerr(FILE * f);
```

Cette fonction remet à zéro l'indicateur d'erreur du fichier f.

15.6 Positionnement dans un Fichier

La fonction fseek

```
int fseek(FILE * f, long offset, int origin);
```

Permet de modifier la position courante dans un fichier. La nouvelle position dépend des valeurs des arguments offset qui représente le déplacement et origin qui représente l'origine. Les valeurs qu'on peut donner à origin sont :

- **SEEK_SET** : le pointeur sera ramené à offset caractères par rapport au début du fichier
- **SEEK_CUR** : le pointeur sera ramené à offset caractères par rapport à la position courante
- **SEEK_END** : le pointeur sera ramené à offset caractères par rapport à la fin du fichier.

Bien entendu, le déplacement peut être positif ou négatif. Si la fonction réussit, 0 est retourné (une valeur différente de zéro indique donc une erreur).

L'utilisation de cette fonction avec un flux de texte peut donner des résultats inattendus à cause du coup du '\n' ... Pour aller au n-ième caractère d'un fichier associé à un flux de texte (ce qui entraîne que la fin de ligne sera vue du programme comme un et un seul caractère : '\n') par exemple, utilisez la méthode suivante :

```
fseek(f, 0L, SEEK_SET);
for(i = 0; i < n; i++)
    getc(f);
/* On est maintenant a la position n dans le fichier */
```

Cette technique est aussi connue sous le nom d'**accès séquentiel**, par opposition à **accès aléatoire** (utilisant fseek, etc.).

Dans la pratique, on ne sera jamais confronté à un tel cas. Bon, « jamais » c'est peut être un peu trop mais en tout cas je peux vous affirmer que je n'ai jamais eu à le faire. Si on doit faire beaucoup de « va et vient » dans un fichier, c'est tout simplement une erreur de l'avoir ouvert en tant que fichier texte.

La fonction ftell

```
long ftell(FILE * f);
```

Permet de connaître la position courante dans le fichier. Dans le cas où le fichier est associé à un flux binaire, il s'agit du nombre de caractères entre le début du fichier et la position courante. Si le fichier est associé à un flux de texte, la valeur retournée par cette fonction est tout simplement une information représentant la position actuelle dans le fichier et on ne peut rien dire de plus. Quel que soit le cas, le retour de ftell pourra éventuellement être utilisée dans un futur appel fseek avec SEEK_SET pour revenir à la même position.

Les fonctions fgetpos et fsetpos

La fonction **fgetpos** permet de sauvegarder la position courante dans un fichier. On pourra ensuite ultérieurement revenir à cette position à l'aide de la fonction **fsetpos**.

```
int fgetpos(FILE * f, fpos_t * p_pos);
int fsetpos(FILE * f, const fpos_t * p_pos);
```

Le type **fpos_t** est un type limité à usage de ces fonctions. On ne doit passer à fsetpos qu'une valeur obtenue à l'aide de fgetpos. En cas de succès, 0 est retourné.

La fonction rewind

Cette fonction permet de « rembobiner » un fichier.

```
rewind(f);
```

est équivalent à :

```
fseek(f, 0L, SEEK_SET);
clearerr(f);
```

Position courante

En général, les opérations de lecture (respectivement d'écriture) commencent la lecture (respectivement l'écriture) à partir de la position courante puis ensuite déplacent le pointeur selon le nombre de caractères lus (respectivement écrits). Il existe toutefois des exceptions comme dans le cas où le fichier est ouvert en mode ajout par exemple, auquel cas toutes les opérations d'écriture se feront à la fin du fichier, indépendamment de la position courante.

15.7 Le Traitement par Blocs

La bibliothèque standard offre également des fonctions permettant de réaliser des opérations d'entrées/sorties par **blocs d'octets**. Il s'agit des fonctions **fread** et **fwrite**.

```
size_t fread(void * buffer, size_t size, size_t nobj, FILE * f);
size_t fwrite(const void * buffer, size_t size, size_t nobj, FILE * f);
```

fread lit **nobj** objets de taille **size** chacune à partir de la position courante dans **f** pour les placer dans un buffer. Elle retourne ensuite le nombre d'objets effectivement lus.

A l'inverse **fwrite** écrit **nobj** objets de **buffer** de taille **size** chacune en destination de **f**. Elle retourne ensuite le nombre d'objets effectivement écrits.

Par exemple, voici une manière d'écrire "Bonjour" dans un fichier en utilisant fwrite.

```
#include <stdio.h>
#include <string.h>
int main()
{
    FILE * f;
    char s[] = "Bonjour";

    f = fopen("bonjour.txt", "wb");

    if (f != NULL)
    {
        fwrite(s, sizeof(char), strlen(s), f);
        fclose(f);
    }
    else
        perror("bonjour.txt");

    return 0;
}
```

15.8 Opérations sur les fichiers

Renommer ou déplacer un fichier

La fonction **rename** :

```
int rename(const char * oldname, const char * newname);
```

permet de renommer un fichier lorsque cela est possible. Si la fonction réussit, 0 est retourné.

Supprimer un fichier

La fonction **remove** :

```
int remove(const char * filename);
```

permet de supprimer un fichier. Si la fonction réussit, 0 est retourné.

15.9 Les fichiers temporaires

Un **fichier temporaire** est un fichier utilisé par un programme puis supprimé lorsque celui-ci se termine. La fonction **tmpfile** :

```
FILE * tmpfile(void);
```

permet de créer un fichier en mode "wb+" qui sera automatiquement supprimé à la fin du programme.

On peut toujours bien sûr créer un fichier temporaire « manuellement ». Dans ce cas, il vaut mieux lui donner un nom généré par **tmpnam** afin de s'assurer qu'aucun autre fichier porte déjà ce nom.

```
char * tmpnam(char * name);
```

Cette fonction requiert que name soit un buffer d'au moins **L_tmpnam** octets autrement il est possible qu'elle ne pourra pas contenir le nom généré par cette fonction ce qui va provoquer un débordement de tampon !

15.10 Rediriger un flux d'E/S

La fonction **freopen** :

```
FILE * freopen(const char * filename, const char * mode, FILE * f);
```

ferme le fichier associé au flux *f*, ouvre le fichier dont le nom est spécifié par *filename* selon le mode spécifié par *mode* en lui associant le flux représenté par *f* puis retourne *f* ou NULL si une erreur s'est produite. Dans le programme suivant, la sortie standard est redirigée vers le fichier out.txt.

```
#include <stdio.h>
int main()
{
    if (freopen("out.txt", "w", stdout) != NULL)
    {
        printf("Hello, world\n");
        fclose(stdout);
    }
    else
        perror("out.txt");

    return 0;
}
```

15.11 Exercice

1. Ecrire un programme qui écrit le contenu entier d'un tableau de 10 entiers dans un fichier puis écrire un autre programme qui lit les dix entiers du fichier puis les affiche sur la sortie standard.
2. Ecrire un programme qui affiche la longueur d'un fichier dont le nom sera fourni par l'utilisateur (astuce : utiliser fseek/ftell).
3. Ecrire un programme qui permet de découper un fichier en un ou plusieurs morceaux puis écrire un autre programme qui permet de restituer le fichier original à l'aide des découpes.
4. Trouver puis implémenter une méthode simple permettant de compresser un fichier (même un programme qui n'est vraiment efficace que pour un type particulier de fichier).

15.12 Exercice : Fichiers Avancés

Exercice: Vous allez ré-écrire le programme qui gère les notes des étudiants:

Pour cela il faut utiliser les fonctions que vous auriez trouvé dans le cours sur les fichiers avancés... comme par exemple, la possibilité de créer un fichier temporaire, de l'effacer, d'ouvrir un fichier en lecture écriture, etc...

16 Projets Contrôle Continu

Sujet 1: Gestion d'un Annuaire Téléphonique

Le problème posé est la gestion d'un Annuaire Téléphonique

1. Définir les structures de données en mémoire nécessaire au stockage des informations **Nom, Prénom, Adresse, Téléphone**.
2. Développer un programme qui permet via un menu de:
 1. Ajouter un nouvel élément dans l'annuaire.
 2. Enlever une personne de l'annuaire.
 3. Modifier un des champs d'une personne de l'annuaire.
 4. Rechercher les coordonnées d'une personne dans l'annuaire (à partir du nom, prénom,...).
 5. Afficher le contenu de l'annuaire.

dans un deuxième temps et afin de donner de la persistance aux données de l'annuaire, vous allez stocker les informations dans un fichier.

3. Indiquez le type de fichier utilisé et la façon dont les données seront enregistrées.
4. Modifiez le programme précédent en ajoutant les fonctions:
 1. Chargement d'un fichier annuaire.
 2. Sauvegarde du fichier annuaire.
 3. Sortir de l'annuaire.

Sujet 2: Gestion de Bibliothèque

Le problème posé est la classification et le stockage d'ouvrages dans une bibliothèque.

1. Définir les structures de données en mémoire nécessaire au stockage des informations: **Titre, Auteur(s), Année de parution, , Numéro ISBN, Résumé, Style**.
2. Développer un programme qui permet via un menu de:
 1. Ajouter un ouvrage dans la bibliothèque.
 2. Modifier une fiche de la bibliothèque.
 3. Supprimer un ouvrage de la bibliothèque.
 4. Rechercher un ouvrage dans la bibliothèque (à partir d'une partie du titre, du nom, d'un auteur, de l'année de parution,...).
 5. Lister les ouvrages de la bibliothèque.

dans un deuxième temps et afin de donner de la persistance aux données de la bibliothèque, vous allez stocker les informations dans un fichier.

3. Indiquez le type de fichier utilisé et la façon dont les données seront enregistrées
4. Modifiez le programme précédent en ajoutant les fonctions:
 1. Chargement du fichier bibliothèque.
 2. Sauvegarde du fichier bibliothèque.
 3. Sortir du logiciel.

Sujet 3: Gestion de Dictionnaire

Le problème posé est la gestion d'un Dictionnaire Informatisé.

1. Définir les structures de données en mémoire nécessaire au stockage des informations: **Mot, Définition, Genre (Verbe, Nom Commun, Nom Propre)**.
2. Développer un programme qui permet via un menu de:
 1. Ajouter un mot
 2. Modifier un champ relatif à un mot
 3. Supprimer un mot
 4. Rechercher un mot dans la bibliothèque (à partir du mot)
 5. Lister les mots du dictionnaire.

dans un deuxième temps et afin de donner de la persistance aux données, vous allez stocker les informations dans un fichier.

3. Indiquez le type de fichier utilisé et la façon dont les données seront enregistrées
4. Modifiez le programme précédent en ajoutant les fonctions:
 1. Chargement du fichier.
 2. Sauvegarde du fichier.
 3. Sortir du logiciel.

Sujet 4: Gestion de Stock

Le problème posé est la Gestion d'un Stock.

1. Définir les structures de données en mémoire nécessaire au stockage des informations: **Désignation, Quantité, Numéro, Rayon**.
2. Développer un programme qui permet via un menu de:
 1. Ajouter un nouvel article.
 2. Modifier un champ relatif à un article.
 3. Augmenter ou diminuer le nombre d'articles stockés.
 4. Supprimer un article.

5. Rechercher une fiche dans le stock (à partir du numéro, de la désignation, du rayon, ...)
6. Lister les fiches.

dans un deuxième temps et afin de donner de la persistance aux données, vous allez stocker les informations dans un fichier.

3. Indiquez le type de fichier utilisé et la façon dont les données seront enregistrées
4. Modifiez le programme précédent en ajoutant les fonctions:
 1. Chargement du fichier.
 2. Sauvegarde du fichier.
 3. Sortir du logiciel.

Cahier des Charges:

Vous devrez dans la mesure du possible présenter votre travail de la façon suivante.

- Annotez votre programme de façon à le rendre compréhensible par une autre personne.
- Ecrivez le sous forme de fonctions, celles ci devront être stockées dans une bibliothèque de fonctions fournie avec le programme.
- Le programme fourni devra être le plus convivial possible.
- Le programme fourni devra être le plus robuste possible concernant la gestion des erreurs d'ouverture de fichiers et de lecture de paramètres utilisateurs.
- Fournir un fichier associé contenant les éléments demandés (annuaire ou bibliothèque...) pour essai par le correcteur.

Mise en Oeuvre du Projet:

Les impératifs de ce projet sont d'une part la durée limitée, d'autre part chaque projet sera mené à bien par un trinôme (ou un quadrinôme). Il vous faudra donc organiser votre travail, c'est à dire:

- Réfléchir ensemble sur le projet et la façon de le programmer
- Attribuer à chaque personne une tâche.
- Définir un cahier des charges par personne définissant les paramètres d'entrée, et de sortie.
- Réunir les différentes parties du projet (chaque fonction devra avoir en commentaire l'auteur de façon à garder une trace)

Travail à rendre:

Vous devrez rendre tous les fichiers nécessaire au bon fonctionnement de votre programme, fichiers sources, fichiers bibliothèques, ainsi qu'un fichier exemple déjà rempli par vos soins avec quelques fiches.

Vous devrez en outre présenter votre travail lors de la dernière séance de TD/TP, expliquant la philosophie de votre programme avec les principales fonctions, un organigramme sommaire expliquant les différentes composantes et les différentes fonctions utilisées. Dans ce compte rendu sera mentionné la part de travail des éléments du groupe.