# Grenoble INP – Esisar
# CS351
# Rappels algorithmiques et Introduction à la programmation en langage C

*Stéphanie Chollet, Ioannis Parissis*

## Objectifs

Les objectifs de cet enseignement qui se déroule, essentiellement, de manière pratique sont les suivants :

- Rappeler les principes algorithmiques de construction de programmes simples ;
- Initier les élèves à l'utilisation du langage C et les rendre capables de pouvoir progresser de manière autonome dans la suite de leur cursus ;
- Leur donner l'occasion de se confronter à la réalisation d'un projet relativement conséquent et en autonomie,
- Enfin, se familiariser avec l'utilisation du système Unix.

Des cours dits "de soutien" sont proposés aux élèves n'ayant pas d'expérience de programmation en C pour leur permettre une meilleure progression. **Ils sont ouverts à tous** (sous réserve de compatibilité de l'emploi du temps).
Etant donné le niveau relativement hétérogène en début d'année, il est probable que l'avancement des différents binômes soit différent et c'est une caractéristique intrinsèque de cet enseignement. L'objectif est d'arriver, en fin de semestre, non pas à une totale homogénéité mais à l'acquisition d'un socle commun de connaissances et de compétences.

## Organisation et évaluation

L'enseignement est organisé en deux parties :

1. Une partie constituée de TP assez classiques organisée en 8 chapitres (et autant de thèmes). Chaque chapitre est calibré pour être traité en une séance (si vous n'avez pas terminé le chapitre en fin de séance il est plus que conseillé de le terminer hors séance et, dans tous les cas, avant la séance suivante et de **déposer votre travail sur chamilo – voir modalités sur le site**). *Il est également conseillé de préparer (raisonnablement) les TP (i.e. une lecture du sujet et des parties de la documentation C correspondante).*
   *Deux évaluations écrites* sont organisées au milieu et à la fin de cette première partie (à des dates qui vous seront communiquées). Le travail de TP que vous rendez doit être réalisé consciencieusement (*cela vaut également pour les élèves ayant déjà une expérience de programmation en C : cela permet de consolider les connaissances du socle commun*), en particulier parce qu'ils peuvent être pris en compte dans la note finale.

   Cette première partie (évaluations incluses) devrait, au plus, occuper 10 des 17 séances de travail prévues.

2. La deuxième partie de l'enseignement consiste en la réalisation d'un unique projet.

Tout au long de l'enseignement aura également lieu une initiation au système UNIX, selon des modalités qui vous seront communiquées en séance et par courriel. **Les évaluations écrites porteront également sur ce point.**

La note finale de l'enseignement est composée des notes des évaluations des TP (30%+30%) et de la note du projet (40%).

## Informations pratiques

*Les travaux pratiques sont à effectuer sous Linux (système UNIX sur PC : voir la documentation qui vous est fournie à ce sujet).*
*Pour créer/éditer des programmes, utiliser un éditeur de texte.*
*Pour lancer une compilation ou exécuter votre programme, utiliser un terminal[1].*
*On se réfère au polycopié « Introduction au langage C » de B. Cassagne.*

## Table des matières



[1] *Menu Accessoires>Terminal.*

# Chapitre 1 : Ecriture, compilation et exécution de programmes C simples

Objectifs :

- Découvrir la syntaxe et la sémantique du langage C.
- Être capable d'écrire en C un algorithme simple.
- Maîtriser l'écriture de fonctions.
- Être capable d'écrire en C un programme structuré en plusieurs fonctions.

## 1. Mon premier programme C

Créer un répertoire CS351 puis un sous répertoire TP1. Dans TP1, créer un fichier `prog1.c` et y ajouter le texte suivant :

```
#include <stdio.h>
int main() {
     printf("Bonjour\n"); /* \n signifie " passage à la ligne " */
     return (0);
}
```

Ensuite exécuter la commande suivante[2] :

```
gcc prog1.c -Wall -ansi -pedantic -o prog1
```

Vous venez de compiler votre premier programme C. A l'issue de l'exécution de cette commande, le fichier `prog1` a été ajouté dans votre répertoire courant. Il s'agit du programme exécutable correspondant à `prog1.c`. Pour exécuter ce programme, taper simplement :

```
./prog1
```

## 2. Notions de base du langage C : types, constantes, variables, expressions, instructions conditionnelles

*Ce paragraphe est un résumé très compact du polycopié, chapitre 1, sections 1.1 à 1.13, dont la lecture est nécessaire.*

Les *identificateurs* du langage (noms de variables, fonctions etc.) sont des suites formées de lettres, chiffres (pas en première position) et du caractère '_' (p.ex. bonjour, b3jour, bon_jour, _bonjour).

Un *commentaire* est compris entre /* et */ (/* Ceci est un commentaire */ et ceci une faute de syntaxe */).

Les *types de base* du langage sont : caractère (char), entier (int, short int, long int), flottants (float, double, long double).

L'écriture de *valeurs constantes* se fait naturellement pour les cas simples : 2, 2.6, 'a', "toto". De nombreuses écritures spécifiques sont disponibles (p.ex. 4L désigne l'entier 4 représenté

[2] *Voir TP2 pour des explications sur la commande gcc.*

comme un entier long ; '\n' correspond au caractère *newline*…) à découvrir dans le polycopié. Le moyen le plus simple de définir une *constante nommée* est d'utiliser la directive `#define` :

```
#define PI 3.14159 /* sans ';' à la fin */
```

Cette ligne fait de sorte que toute occurrence de l'identificateur `PI` dans le programme est remplacée par `3.14159`.

L'affectation en C s'effectue à l'aide de l'opérateur = (égal).

```
x = 5 ; /* affectation à x de la valeur 5 */
```

Une affectation est une expression retournant une valeur :

```
int k = 1 ;
int i = (j = k) + 1 ; /* j prend la valeur de k, soit 1 ; i prend
la valeur 2, soit la valeur de (j = k) + 1 */
```

Les opérateurs de comparaison sont == (égalité), >, <, >=, <=, != (différence).

L'instruction conditionnelle `if` :

```
if(condition) {
     instruction
}
/* noter la parenthèse obligatoire autour de la condition*/
if(condition) {
     instruction
} else {
     instruction
}
```

Une instruction peut prendre la forme d'un *bloc d'instructions* délimitées par des accolades (qui correspondent à « début » et « fin » en langage algorithmique).

```
if(x > 0) {
   x = x + 1 ;
   y = 0 ;
} else {
   x = y = 1;
}
```

L'instruction de choix `switch` (voir polycopié 3.15.1) :

```
/* x, y sont des int */
switch (x) {
     case 0: y = 1; break;
     case 1: y = 2; break;
     default: y = 0;
}
```

## 3. Entrées sorties élémentaires

L'affichage à l'écran ou dans un fichier, la lecture depuis le clavier ou depuis un fichier se font au moyen d'appels à des fonctions des bibliothèques de C (en d'autres termes, il n'existe pas d'instruction de base du langage permettant d'effectuer une action d'entrée ou sortie). Avant une étude plus approfondie de ces fonctions, on se contente ici d'utiliser deux d'entre elles,

`printf` et `scanf`. Leur utilisation est rapidement expliquée dans les sections 1.16 et 3.14 du polycopié. L'exemple ci-dessous en illustre quelques utilisations simples :

```
#include <stdio.h>

#define INC 2

int main() {
   int i;
   printf("Donnez un entier : ");
   scanf ("%d", &i); /* &i : i par adresse (voir plus loin) */
   i = i + INC ;
   printf("valeur de i=%d et son successeur i+1=%d\n", i, i+1);

     return (0);
}
```

### Exercice 1 : Appréciation

Nous souhaitons écrire un programme qui lit une note exprimée sous la forme d'une lettre de A à E, puis affiche un message correspondant à la lettre saisie (« Très bien », « Bien », « Assez bien », « Passable » et « Insuffisant »). On fera trois versions de programme :

- Une version utilisant des instructions `if` imbriquées (if… else… if… etc);
- Une version utilisant des `if` en séquence (sans imbrication : if ...; ; if… ;);
- Une version utilisant l'instruction `switch`.

Que doit-il se passer si l'utilisateur entre une autre lettre que A à E ?

### Exercice 2 : Année bissextile

Une année bissextile comprend un jour de plus que les années normales. On dit couramment que les années bissextiles reviennent tous les quatre ans, ce qui n'est pas tout à fait exact. La définition complète est la suivante :
Une année bissextile est divisible par 4 ; mais si elle est également divisible par 100, alors elle doit aussi être divisible par 400.

1. Etablir un ensemble de tests (entiers correspondant à une année) qui permettront de vérifier la correction du programme. Ajouter, sous forme de commentaire, ces tests dans le programme de la question suivante en expliquant leur choix.
2. Écrire un programme qui lit un entier correspondant à une année et affiche un message approprié en fonction (« L'année 1996 est bissextile », « L'année 2001 n'est pas bissextile »).

## 4. Itérations

Il existe trois instructions permettant de réaliser des itérations en langage C : `while`, `do`, `for` (lire les sections 2.2 et 2.3 du polycopié).

```
/* condition est une expression booléenne */
while(condition) {
     instruction
```

```
}
```

```
do {
     instruction
} while (condition) ;
```

```
for(expression1 ; expression2 ; expression3) {
     instruction
}
```

IMPORTANT : Nous rappelons qu'une boucle bien construite **ne doit pas** contenir de `return`, `break` ou `continue` dans son corps. La sortie de la boucle doit se faire **exclusivement** au moment de l'évaluation de la condition, quel que soit le type de boule utilisé.

### Exercice 3 : Somme des n premiers entiers

Nous souhaitons écrire un programme calculant la somme des n premiers entiers. La valeur de n est fournie par l'utilisateur.

(i) Réaliser une version de ce programme utilisant l'instruction `while` et une autre utilisant `do`.

(ii) Que se passe-t-il si la valeur entrée par l'utilisateur est négative ?

### Exercice 4 : Opérateurs de post et pré incrémentation et décrémentation

Pour bien comprendre comment fonctionnent les opérateurs ++ et --, exécuter et commenter le programme suivant (assurez-vous d'avoir bien compris !) :

```
#include <stdio.h>

int main() {
     int i, j, k, l;

     i = j = k = l = 0;

     while(i < 9) {
          printf("i++ = %d, ++j = %d, k-- = %d, --l = %d\n",
                  i++, ++j, k--, --l);
     }

     printf ("i = %d, j = %d, k = %d, l = %d\n", i, j, k, l);

     return (0);
}
```

### Exercice 5 : Opérateurs binaires

Exécuter et commenter le programme suivant (en profiter pour lire la section 7.2 du polycopié) :

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

```
int main() {
     int i = 2, j = 3, k = 4;

     printf("%d, %d, %d\n", i & j, i | 1, k | j);

     if ((i & k) || (i & j))
        printf("ok");

     return (0);
}
```

## 5. Fonctions et procédures : notions de base

Lire les sections 1.15.1 à 1.15.3 du polycopié.

Exemple de définition d'une fonction :

```
/* la fonction somme retourne un int */
/* ses paramètres formels sont les int i et j */
int somme(int i, int j) {
     int res; /* variable locale */
     res = i + j;
     return(res); /* retour de la valeur de res */
}
```

`int somme(int i, int j)` est le *prototype* de la fonction `somme`.

Appel de fonction :

```
#include <stdio.h>

int main(){
     int r, a, b;

     a = 3;
     b = 2;
     r = somme(a, b);
     printf("a + b = %d\n", r);

     return (0);
}
```

En C, il n'existe pas de procédure, à proprement parler. En fait, une procédure est une fonction qui ne retourne pas de valeur (auquel cas, on spécifie `void` comme type de retour) :

```
#include <stdio.h>

void aff_somme(int i, int j) {
     int res = i + j;
     printf("somme = %d\n", res);
}

int main(){
     int r, a, b;
```

```
    a = 3;
    b = 2;
    aff_somme(a, b);

    return (0);
}
```

Pour chacun des exercices, on écrira, **dans un même fichier**, la ou les fonctions demandées ainsi que la fonction `main` les appelant.

## Exercice 1 : Fibonacci

On rappelle la suite de Fibonacci définie par :

$$u_0 = 0$$
$$u_1 = 1$$
$$u_n = u_{n-1} + u_{n-2} \text{ si } n > 1$$

- Nous souhaitons écrire une fonction `fibonacci` calculant le terme de rang n de la suite dont le prototype est : `int fibonnacci(int n);`

- Ecrire cette fonction en utilisant une instruction `while`, `for` ou `do`.
- Ecrire ensuite une fonction `main` demandant la valeur de n à l'utilisateur et affichant le terme correspondant de la suite.
- Quelles sont les valeurs de n que l'utilisateur doit entrer pour bien tester cette fonction ?

## Exercice 2 : PGCD

On rappelle que le pgcd est défini par les relations suivantes (a et b étant des entiers naturels) :

pgcd(a, 0) = a

pgcd(a, b) = pgcd(b, r) avec r = a mod b, si b ≠ 0 (mod est le reste de la division entière).

- Ecrire une fonction `pgcd` (**utilisant une itération**), à deux paramètres entiers, retournant le pgcd de ses paramètres. **Au préalable**, déterminer un ensemble de triplets de valeurs (a, b, p), tels que p = pgcd(a, b) qui permettent de tester cette fonction.
- Ecrire une fonction `main` demandant deux valeurs entières à l'utilisateur et affichant leur pgcd. Vérifier la correction de la fonction en entrant les valeurs précédemment déterminées.

## Exercice 3 : Factorielles

- Ecrire une fonction `factorielle` qui calcule et retourne la valeur de n! (1 x 2 x 3 x…x n). Cette fonction doit être réalisée à l'aide d'une instruction `for` ou `while`.
- Ecrire une fonction `factorielleBis` à un paramètre entier m qui calcule et retourne la valeur du plus petit entier positif n tel que n! (factorielle de n) soit supérieur à m. Cette fonction ne doit pas faire appel à la fonction `factorielle`.

- Ecrire ensuite une fonction `main` demandant la valeur de n à l'utilisateur et affichant le résultat de chaque fonction ci-dessus. Préciser, à l'aide de commentaires précédant la fonction `main`, les tests que vous avez effectués.

### Exercice 4 : Jeu de multiplication

On veut écrire une procédure `jeuMulti` qui demande à l'utilisateur de réciter sa table de multiplication. L'utilisateur commence par entrer un nombre entre 2 et 9 (si le nombre est incorrect, le programme le redemande). Ensuite l'algorithme affiche une à une les lignes de la table de multiplication de ce nombre, en laissant le résultat vide et en attendant que l'utilisateur entre le résultat. Si celui-ci est correct, on passe à la ligne suivante, sinon on affiche un message d'erreur donnant la bonne valeur et on termine. Si toutes les réponses sont correctes, on affiche un message de félicitations. On représente ci-dessous une exécution possible (les entrées de l'utilisateur sont affichées en italiques) :

Valeur de n : *12*
Réessayez : la valeur doit être comprise entre 2 et 9
Valeur de n : *6*
1 x 6 = *6*
2 x 6 = *12*
3 x 6 = *21*
Erreur ! 3 x 6 = 18 et non 21

1. Ecrire la procédure `jeuMulti`.
2. Ecrire une nouvelle procédure `jeuMultiPoints` qui ne s'arrête pas quand une réponse fausse est donnée, mais affiche à la fin le nombre d'erreurs commises et un message éventuel de félicitations.
3. Testez ces deux procédures en précisant les tests que vous avez réalisés et en justifiant leur choix.

## 6. Rappel sur la construction des itérations

Dans l'exercice précédent, l'itération construite dans la question 2 est un *parcours de l'intégralité de la séquence* des réponses fournies par l'utilisateur. De ce fait, l'itération est de la forme :

```
Tant qu'il y a des éléments dans la séquence
        Traiter l'élément courant
        Passer à l'élément suivant
```

Par contre, dans la question 1, on s'arrête dès qu'une mauvaise réponse est donnée. En d'autres mots, on *recherche* la première erreur dans la séquence des réponses fournies par l'utilisateur. Plus généralement, une itération de recherche d'une valeur dans une séquence aura la forme :

```
Tant qu'il y a des éléments dans la séquence et l'élément
recherché n'est pas trouvé
        Traiter l'élément courant
        Passer à l'élément suivant
```

## 7. Méthodologie à suivre pour la réalisation des exercices

Dans les exercices précédents, un accent a été mis sur le **choix des tests** à réaliser pour vérifier la correction des programmes ou fonctions demandés. Désormais, pour chaque exercice que vous réaliserez, une réflexion préalable (et documentée) sur les tests que vous allez utiliser doit être menée. Le choix des tests (entrée de la fonction/programme, résultats attendus) doit être précisément décrit et justifié.

# Chapitre 2 : Compilation séparée
# Makefile, fichiers .h et .c

Objectifs :
- Maîtriser l'outil make et les fichiers Makefile.
- Maîtriser la compilation séparée.
- Savoir utiliser les bibliothèques prédéfinies en C.

## Compilation séparée

### 1. Introduction

#### 1.1. Comment marche la compilation ?

La compilation avec gcc s'effectue en quatre étapes pour produire, à partir d'un programme C, un code exécutable :

1. passage au pré-processeur (*preprocessing*) ;
2. compilation en langage assembleur (*compiling*) ;
3. conversion du langage assembleur en code machine (*assembling*) ;
4. édition des liens (*linking*).

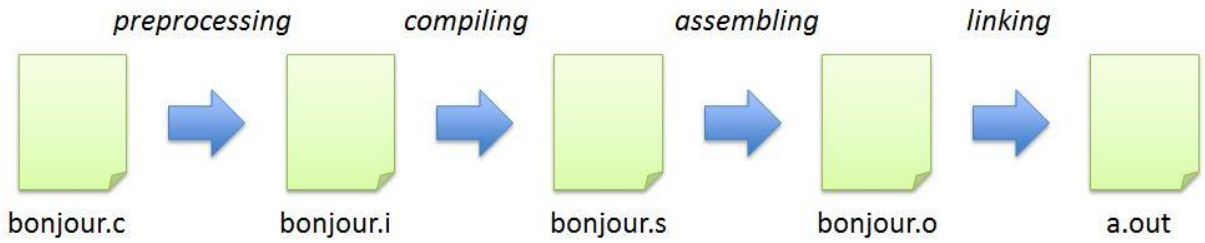



**Figure 1 : Chaîne de compilation.**

Le prétraitement (*preprocessing*) consiste à transformer le code source d'un programme C en un autre code source C débarrassé des directives de prétraitement et où les macros du processeur ont été développées. Les directives de prétraitement permettent d'inclure d'autres fichiers contenant des déclarations (ex : `#include <fichier.h>`) et de définir de nouvelles macros (ex : `#define PI 3.14`). Les directives du préprocesseur commencent toutes par un caractère dièse (#).

La compilation (*compiling*) du code C prétraité en assembleur est la phase la plus compliquée. Pendant cette phase, on passe d'un langage de haut niveau vers un langage de très bas niveau : l'assembleur.

L'assemblage (*assembling*) permet la traduction du langage assembleur (texte) en langage machine binaire (fichier "objet").

La commande gcc utilisée avec l'option –c permet de réaliser ces trois premières étapes. Ainsi, la commande `gcc –c bonjour.c` permet de produire le fichier `bonjour.o` (la commande `gcc –c bonjour.c –o bonjour.o` a le même résultat).

D'autres options de compilation peuvent être utilisées. Nous en utilisons systématiquement trois dans le cadre de cet enseignement :

```
gcc –c bonjour.c -Wall -ansi -pedantic –o bonjour.o
```

Elles permettent de garantir que la syntaxe utilisée est la plus standard et la plus correcte possible (ce qui est important en phase d'apprentissage).

Enfin, l'édition de liens (*linking*) permet de combiner éventuellement plusieurs fichiers "objet" produits par la phase d'assemblage ainsi que les éventuelles bibliothèques utilisées (p.ex. pour les entrées et sorties) en un seul programme exécutable. Elle fait le lien entre des symboles définis dans les fichiers objets et ces mêmes symboles utilisés dans d'autres objets. La commande suivante effectue l'édition des liens entre les fichiers objet `bonjour.o` et `bonsoir.o` et produit le fichier exécutable `progexe` : `gcc bonjour.o bonsoir.o –o progexe`. Si le nom de fichier résultant de la compilation n'est pas spécifiée à l'aide de l'option `–o (gcc bonjour.o bonsoir.o)`, il prend par défaut la valeur "a.out".

On peut également écrire `gcc bonjour.c –o bonjour`. Dans ce cas, la compilation et l'édition des liens est faite est le programme exécutable `bonjour` est produit (ceci n'a évidemment pas de sens si notre application est composée de plusieurs fichiers .c).

### 1.2. Intérêt de la compilation séparée

La compilation séparée est nécessaire à partir du moment où une application est écrite à l'aide de plusieurs fichiers sources (.c), ce qui est en général le cas, pour plusieurs raisons :

- la programmation est modulaire, donc plus compréhensible ;
- la séparation en plusieurs fichiers produit des listings plus lisibles ;
- la maintenance est plus facile car seuls les modules modifiés sont recompilés.

Sous UNIX, la commande `make`[3] permet d'automatiser l'exécution des étapes de compilation séparée. Cette commande :

- effectue la compilation séparée grâce à la commande gcc ;
- utilise des macro-commandes et des variables ;
- permet de ne recompiler que les fichiers sources modifiés ;
- permet d'utiliser des commandes shell.

### 1.3. Mise en œuvre

La commande make requiert pour son fonctionnement un fichier nommé (par défaut) Makefile (avec un M majuscule et le reste en minuscule). Il doit se trouver dans le répertoire

[3] make est essentiel lorsque l'on veut effectuer un portage, car la plupart des logiciels libres UNIX (c'est-à-dire des logiciels qui sont fournis avec le code source) l'utilisent pour leur installation.

courant lorsque l'on appelle la commande make à l'invite du shell. Les instructions contenues dans ce fichier doivent respecter une syntaxe particulière expliquée ci-après.

## 2. Règles

Les fichiers Makefile sont structurés grâce aux *règles* ; ce sont elles qui définissent ce qui doit être exécuté.

### 2.1. Définition

Une règle est une suite d'instructions qui seront exécutées pour construire *une cible*, mais uniquement si des *dépendances* ont été modifiées depuis la dernière construction de la cible. La syntaxe d'une règle est la suivante :

```
cible : dépendances
     commandes
```

L'espace avant commandes est obligatoire et doit être fait avec une tabulation.

### 2.2. Cible

La cible est souvent (mais pas nécessairement) le nom d'un fichier qui va être généré par les commandes qui vont suivre.

### 2.3. Dépendances

Les dépendances sont les fichiers nécessaires à la création de la cible. Par exemple, pour la compilation C, les dépendances peuvent contenir un fichier d'en-tête ou un fichier source.

### 2.4. Commandes

Les commandes sont des commandes du shell (langage de commande) Unix qui seront exécutées au moment de la construction de la cible. La tabulation avant les commandes est obligatoire et si la commande dépasse une ligne, il est nécessaire de signaler la fin de ligne avec un caractère antislash "\".

## 3. Un exemple de fichier Makefile

### 3.1. Mon premier Makefile

```
# Mon premier Makefile

all: foobar.o main.o
     gcc –o main foobar.o main.o

foobar.o: foobar.c foobar.h
     gcc –c foobar.c –Wall –ansi –pedantic –o foobar.o

main.o: main.c
     gcc –c main.c –Wall –ansi –pedantic –o main.o
```

Les lignes commençant par le caractère # sont des commentaires.
Le Makefile ci-dessus a trois règles destinées à construire les cibles `all`, `foobar.o` et `main.o`. La cible principale `all` nécessite les fichiers `foobar.o` et `main.o` pour être

exécutée afin d'obtenir le fichier exécutable `main`. Pour obtenir les fichiers `foobar.o` et `main.o`, il faut utiliser les deux cibles suivantes.

### 3.2. Comment utiliser un Makefile

La commande `make` permet d'exécuter le fichier Makefile :

```
machine@esisar> make all
```

La commande `make` interprète le fichier Makefile et exécute les commandes contenues dans **la règle dont la cible est all**. Au préalable les dépendances `foobar.o` et `main.o` ont été vérifiées (i.e. les règles éventuelles dont la cible est `foobar.o` ou `main.o` ont été exécutées). Cela signifie, par exemple, que si `foobar.c` ou bien `foobar.h` ont été modifiés depuis la dernière compilation, alors `foobar.o` sera d'abord mis à jour avant que la commande de la règle all soit exécutée.
La commande `make` peut être exécutée sans argument. Dans ce cas, elle exécute la première règle rencontrée. Elle peut également utiliser un fichier appelé autrement que Makefile, auquel cas la syntaxe suivante est utilisée : `make -f monMakeFileAMoi`.

## 4. Variables

### 4.1. Définition et utilisation

Les variables doivent être vues comme des macro-commandes (comme #define en C). La déclaration se fait ainsi :

```
NOM = VALEUR
```

La valeur affectée à la variable peut comporter n'importe quels caractères. Elle peut être aussi une autre variable.
La syntaxe de l'appel de la variable est la suivante :

```
$(NOM)
```

### 4.2. Exemple de fichier Makefile avec des variables

```
# $(BIN) est le nom du fichier binaire généré
BIN = foo
# $(OBJECTS) sont les objets qui seront générés
# après la compilation
OBJECTS = main.o foo.o
# $(CC) est le compilateur utilisé
CC = gcc
# all est la première règle à être exécutée car elle est
# la première dans le fichier Makefile. Notons que les
# dépendances peuvent être remplacées par une variable,
# ainsi que n'importe quelle chaîne de caractères des
# commandes
all: $(OBJECTS)
     $(CC) $(OBJECTS) -o $(BIN)

main.o: main.c main.h
     $(CC) -c main.c
foo.o: foo.c foo.h main.h
```

```
	$(CC) -c foo.c
```

# 5. Caractères jockers et variables automatiques

## 5.1. Caractères jockers

Les caractères jockers s'utilisent comme en shell. Les caractères valides sont par exemple : *, ?. L'expression toto?.c représente tous les fichiers commençant par toto et finissant par .c avec une lettre entre ces deux chaînes (toto1.c, toto2.c…). Comme en shell, le caractère \ permet d'inhiber l'action des caractères jockers.

## 5.2. Variables automatiques

Les variables automatiques sont variables qui sont actualisées au moment de l'exécution de chaque règle en fonction de la cible et des dépendances.

| | |
|---|---|
| **$@** | Nom de la cible |
| **$<** | Première dépendance de la liste des dépendances |
| **$?** | Les dépendances les plus récentes de la cible |
| **$^** | Toutes les dépendances |

# 6. Conventions de nommage

## 6.1. Noms d'exécutables et arguments

| | |
|---|---|
| **CC** | Compilateur C (cc ou gcc) |
| **CXX** | Compilateur C++ (c++ ou g++) |
| **RM** | Commande pour effacer un fichier (rm) |
| **CFLAGS** | Paramètres à passer au compilateur C |
| **CXXFLAGS** | Paramètres à passer au compilateur C++ |

## 6.2. Noms des cibles

Un utilisateur de make peut donner à ses cibles le nom qu'il désire, mais pour des raisons de lisibilité, on donne toujours un nom standard à certaines cibles :

| | |
|---|---|
| **all** | Compile tous les fichiers source pour créer l'exécutable principal |
| **install** | Exécute all et copie l'exécutable, les bibliothèques et les fichiers d'en-tête s'il y en a dans le répertoire de destination |
| **uninstall** | Détruit les fichiers créés lors de l'installation, mais pas les fichiers du répertoire d'installation où se trouvent les fichiers sources et le Makefile |
| **clean** | Détruit tous les fichiers créés par all |
| **dist** | Crée un fichier tar de distribution |

## Exercice 1 : Makefile

Ecrire un fichier Makefile simple qui permette de compiler le code de l'exercice du jeu de la multiplication du TP 1.

# Organisation d'un programme

Jusqu'à présent, tout votre code était dans un unique fichier source .c. La structure d'un fichier *monProgramme.c* est en général la suivante :

```
// Inclusion de fichiers permettant d'utiliser des fonctions
// de bibliothèques
#include <stdio.h>
#include <stdlib.h>

// Déclarations des constantes
#define TAILLE 100

// Déclarations des variables globales
int entier1 ;
// Déclarations de toutes les fonctions
// Le détail des fonctions se trouve après la fonction main()
void fonction1(void) ;
void fonction2(void) ;

// Programme principal
int main(int argc, char * argv[]) {
...
...
}

// Implémentation des fonctions déclarées précédemment
void fonction1(void) {
...
}

void fonction2(void) {
...
}
```

Cette méthode fonctionne bien pour de petits programmes assez simples. Mais dans le cas de plus grosses applications, on organise le code en plusieurs fichiers qu'on appelle "modules". Chaque module contient des fonctions offrant un ensemble de fonctionnalités cohérent (on parle souvent de "séparation des préoccupations"). Par exemple, pour un logiciel de jeu, il est possible d'avoir des fonctions pour le graphisme et d'autres pour gérer les règles du jeu et les différents joueurs.

## 1. Découpage d'un programme en modules

En C, un module est réalisé à l'aide de deux sortes de fichiers reconnaissables à leur extension :

- les fichiers .c qui sont des *fichiers source* ;

- les fichiers .h (h pour header en anglais) qui sont les *fichiers d'en-tête*.

Le fichier .h contient les déclarations de type, de fonctions, de constantes du module ; en d'autres termes, tout ce qu'un autre module aurait besoin pour utiliser les fonctions proposées par celui-ci. La réalisation (ou implémentation) des fonctions du module est, quant à elle, contenue dans le fichier .c.

Le principe généralement appliqué consiste à écrire un fichier .h pour chaque fichier .c et à déclarer dans le fichier .h tout ce que le module souhaite "exporter" : lorsqu'un module A utilise une fonction déclarée dans un module B, alors A.h doit inclure (#include) le fichier B.h.

**Exemple :** découpage en modules de l'exemple précédent.

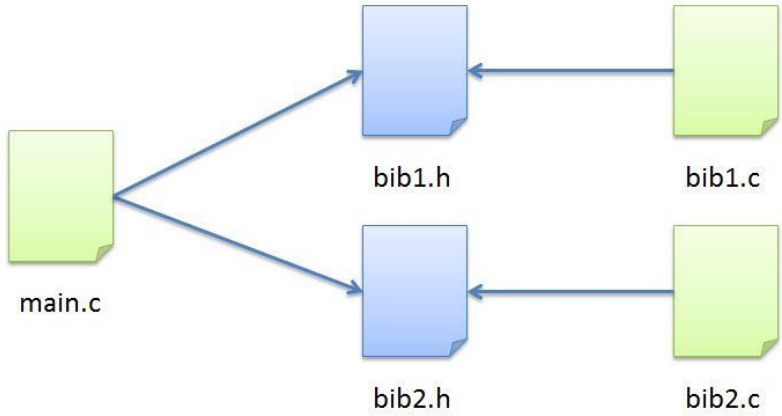



Figure 2 : Architecture du programme.

Les flèches → représentent les inclusions.

Les fichiers bib1.h et bib2.h contiennent les déclarations qui sont utiles aux modules bib1 et bib2 :

Le fichier bib1.h :

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

#define TAILLE=100

void fonction1(void) ;
```

Le fichier bib2.h :

```
#include <stdio.h>

int entier1 ;

void fonction2(void) ;
```

Les fichiers .c contiennent les implémentations des fonctions :

Le fichier bib1.c

```
#include "bib1.h"

void fonction1(void) {
...
}
```

```
#include "bib2.h"

void fonction2(void) {
...
}
```

Le fichier bib2.c

Le programme principal ne contient plus que les inclusions nécessaires ainsi que la fonction main().

```
// Inclusion de fichiers permettant d'utiliser des fonctions
// de bibliothèques

#include "bib1.h"
#include "bib2.h"

// Programme principal
int main(int argc, char * argv[]) {
...
...
}
```

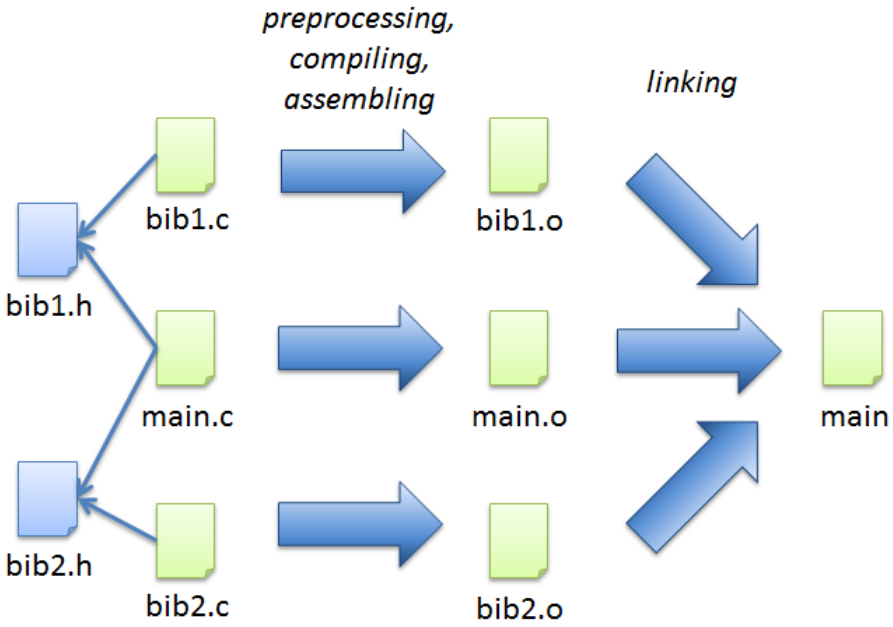



**Figure 3 : Chaîne de compilation à plusieurs fichiers.**

## 2. Compilation conditionnelle

Dans l'exemple précédent, on remarque que des bibliothèques identiques ont été incluses dans différents modules (par exemple, la bibliothèque stdio.h). Par conséquent, lors de la compilation du module principal, ces bibliothèques seront lues plusieurs fois. Ceci peut faire échouer la compilation (par exemple, des éventuelles déclarations de constante seront lues plusieurs fois, ce qui équivaut à une déclaration multiple, interdite en C). Pour éviter au compilateur de lire plusieurs fois les mêmes fichiers, on utilise le mécanisme de *compilation conditionnelle*. Le compilateur tient compte d'un ensemble de commandes pour savoir ce qui doit être compilé ou non.

Des *directives* permettent d'incorporer ou d'exclure des portions du fichier dans le texte qui sera analysé par le préprocesseur. Ces directives se classent en deux catégories selon la condition qui régit l'incorporation :

- Existence ou inexistence des symboles ;
- Valeur d'une expression.

Le détail des *directives* sont présentés dans la section 8.2 du document « *Introduction au langage C* » à la page 139.

Pour résumer, les fichiers .h doivent avoir cette forme (illustrée sur l'exemple du fichier bib1.h) :

```
#ifndef __BIB1_H__
#define __BIB1_H__

void fonction1(void) ;

#endif
```

## Exercice 2 : Exemple de compilation conditionnelle

Le but de cet exercice est d'écrire un programme qui a un comportement différent en fonction des options de compilation utilisées. On reprend, pour cela, le programme de calcul de pgcd du TP1.

- Ecrire une nouvelle version de ce programme telle que, quand il est compilé de la manière suivante: gcc –c –D MISEAUPOINT pgcd.c et après édition de liens, affiche, à chaque itération un message : "valeur courante de b = x" (x étant la valeur de b à l'entrée de l'itération). Ce message ne doit pas être affiché si l'option MISEAUPOINT n'est pas utilisée (gcc –c pgcd.c).

## Exercice 3 : Bibliothèque mathématique

Le but de cet exercice est d'écrire une bibliothèque de fonctions mathématiques. Vous devrez structurer correctement votre code (fichiers .h et .c). Pour chaque fonction, vous devez préalablement à la réalisation du code définir les tests que vous allez effectuer. Ces tests n'ont pas seulement vocation à "démontrer" que le programme fonctionne, mais aussi (et sans doute surtout) à détecter des situations où il ne fonctionnerait pas. Par exemple, dans une division, il est intéressant de tester avec un dénominateur nul, des nombres positifs mais aussi négatifs…

- Ecrire une fonction *quotient()* ayant pour paramètres deux entiers positifs a et b et qui retourne le quotient de a et b. Ce calcul doit se faire par soustractions successives (interdiction d'utiliser l'opérateur de division du langage C).
- Ecrire une fonction *reste()*, ayant pour paramètres deux entiers positifs a et b qui retourne le reste de la division de a par b. Cette fonction doit utiliser la fonction *quotient()* (et non pas l'opérateur %).
- Ecrire une fonction *valeurAbsolue*() qui calcule la valeur absolue d'un entier. Cette fonction peut utiliser la fonction abs() de la bibliothèque stdlib.h.

- Ecrire une fonction *ppcm()* qui calcule le plus petit commun multiple. Pour rappel, il est possible de le calculer ainsi :

$$ppcm(a,b) = \frac{|ab|}{pgcd(a,b)}$$

- Le calcul de la fonction puissance $f(x, n) = x^n$, où x et n sont des entiers positifs, peut se réaliser simplement par une itération réalisant n - 1 multiplications. Mais elle peut aussi se réaliser de sorte à avoir recours à moins d'opérations arithmétiques. L'algorithme suivant ("méthode binaire") permet de réduire considérablement le nombre de multiplications[4] :

  (i) N ← n, Y ← 1, Z ← x
  (ii) N ← N / 2. Si N était pair avant division, aller à l'étape 5
  (iii) Y ← Z * Y
  (iv) Si N = 0, fin de l'algorithme avec réponse Y
  (v) Z ← Z * Z, aller à l'étape 2.

  Réaliser une fonction *int puissanceMB (int x, int n)* qui calcule $x^n$ selon la méthode binaire (MB) présentée ci-dessus. Tester la fonction avec plusieurs valeurs de x et de n et évaluer le nombre total d'opérations effectuées pour chaque test.

- Le mathématicien grec Nikomakhos (Νικόμαχος, 1er siècle après J.C.) écrit dans son Introduction arithmétique que "tout cube est égal à la somme de nombres impairs consécutifs" ($1^3 = 1$, $2^3 = 8 = 3 + 5$, $3^3 = 27 = 7 + 9 + 11$).
  (i) Ecrire une fonction sommeDesImpairs qui reçoit deux paramètres entiers d et f, supposés impairs et tels que d < f. La fonction retourne la somme d + (d + 2) + … + f.
  (ii) Ecrire une fonction estUneDecompositionDe qui reçoit deux paramètres entiers d et f, supposés impairs et tels que d < f. La fonction retourne l'entier dont le cube se décompose en somme de tous les entiers impairs consécutifs entre d et f, s'il existe, -1 sinon. Par exemple, estUneDecompositionDe(7,13) retourne -1 tandis que estUneDecompositionDe(7,11) retourne 3. *NB : Il est interdit (et inutile) d'utiliser ou de réaliser une fonction calculant la racine cubique.*

- Ecrire une fonction *testBibliothèque*() qui effectue des tests des fonctions ci-dessus. Pour chaque test effectué on précisera (sous la forme d'un commentaire) le choix des valeurs des entrées. *Il faudra notamment que les tests couvrent tous les cas particuliers : division par un nombre plus grand, division par 0…*

## Exercice 4 : Mosaïque

Le but de cet exercice est d'utiliser une bibliothèque graphique afin de tracer un carré puis une mosaïque constituée de carrés.
La bibliothèque contient des fonctions qui permettent :

- d'ouvrir et d'initialiser une fenêtre pour dessiner : *gr_inits_w ()* ;
- de changer la couleur du pinceau : *set_blue(), set_red(), set_green(), set_yellow(), set_black()* ;
- de tracer une ligne : *line()* ;

[4] Source : D. Knuth, The Art of Program Computing, Vol. 2, Seminumerical Algorithms

- de récupérer les coordonnées d'un clic souris (cette commande attend qu'un clic se produise) : *cliquer_xy()* ;
- de bloquer l'exécution en attendant un clic de souris : *cliquer() – utile notamment en fin de dessin pour donner le temps au système de tout afficher*.

A partir de ces fonctions, vous devrez tracer un carré « sur la pointe » :

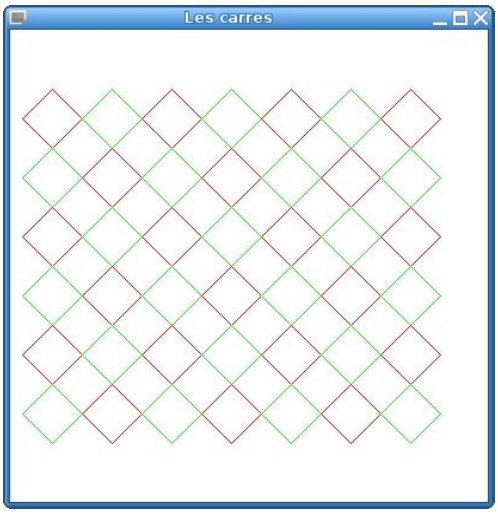
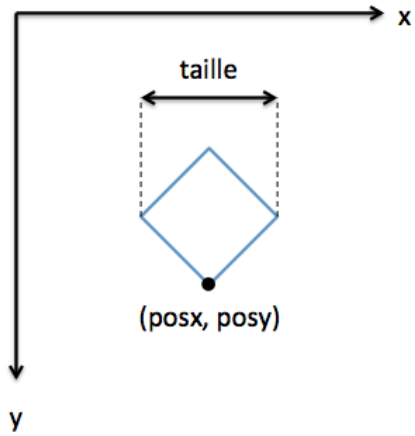



**Figure 4 : Mosaïque.**

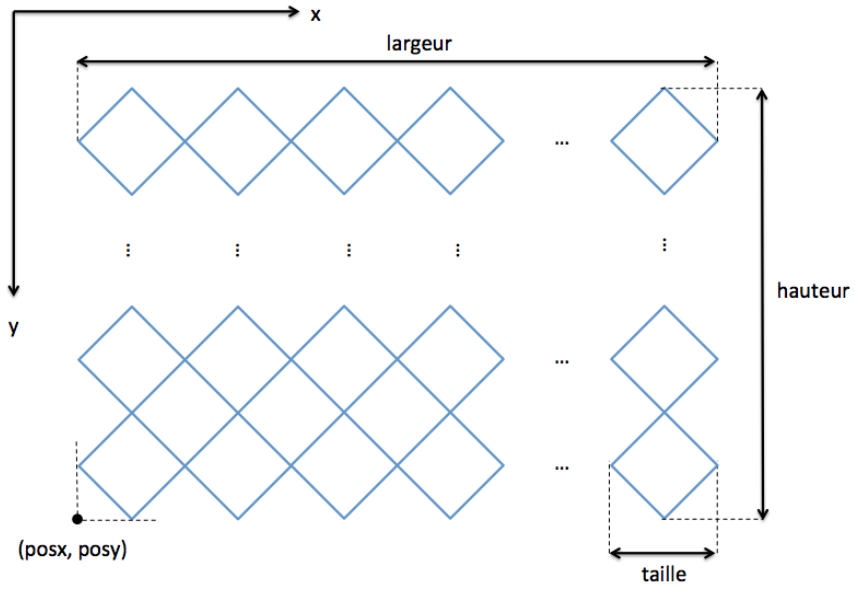



1. Ecrire le fichier *Makefile* qui permet de compiler les fichiers sources qui vous sont fournis. L'utilisation de la bibliothèque graphique nécessite d'utiliser les options `-lX11` et `-L/usr/X11R6/lib` lors de l'édition des liens[5].

Pour la suite, vous devez créer deux fichiers (*dessine.h* et *dessine.c*) dans lesquels vous allez déclarer et implémenter les fonctions suivantes qui seront testées dans le fichier *main.c* :

2. Ecrire une fonction *dessineCarre()* qui dessine un carré « sur la pointe ». Cette fonction prend en paramètre les coordonnées d'un point ainsi que la taille du carré.
3. Ecrire une fonction *dessineCarreDiagonale()* qui dessine un carré ainsi que ses diagonales. Cette fonction prend en paramètre les coordonnées d'un point ainsi que la taille du carré.
4. Ecrire une fonction *dessineMosaique()* qui dessine une mosaïque comme l'illustre la Figure 4. Cette fonction prend en paramètre la taille des carrés, la position de la mosaïque, sa largeur et sa hauteur en nombre de carrés :

[5] Ce sont des inclusions de librairies graphiques binaires nécessaires à graphlib.




5. Ecrire une fonction *dessineMosaiqueAvecSouris()*, similaire à la fonction précédente mais qui ne prend en paramètre que la taille des carrés, sa largeur et sa hauteur en nombre de carrés. La position de la mosaïque sera précisée par un clic souris de l'utilisateur.

# Chapitre 3 : Les tableaux

Objectifs :

- Maîtriser les tableaux à une dimension et à deux dimensions.
- Savoir passer des arguments en ligne de commande.

## Tableau à une dimension

### 1. Rappel algorithmique

Un tableau est caractérisé par trois éléments :

- son nom ;
- son nombre d'éléments ;
- et le type des éléments qu'il contient.

**Exemple :**

precipitations

| 66,0 | 72,6 | 82,8 | 81,9 | 107,2 | 94,1 | 63,4 | 74,9 | 91,4 | 94,1 | 80,3 | 70,7 |
|---|---|---|---|---|---|---|---|---|---|---|---|

Figure 5 : Tableaux des précipitations moyennes à Grenoble.

Le tableau *precipitations* contient *12* éléments de type *réel*. ■

### 2. Déclaration d'un tableau

En langage C, une variable de type tableau doit être déclarée au même titre que les autres variables dans la partie déclaration. La déclaration d'un tableau se fait ainsi :

```
TYPE nomTableau[n]
```

Où :

- TYPE représente le type des données contenues dans le tableau ;
- nomTableau est le nom de la variable de type tableau ;
- n est le nombre d'éléments du tableau.

**Exemple :**

```
float precipitations[12] ;
```

déclare un tableau precipitations à 12 éléments de type réel. ■

### 3. Utilisation d'un tableau

En langage C, l'utilisation d'un tableau se fait ainsi :

```
nomTableau[indice]
```

Où indice peut être :

- une variable simple : precipitations[i]
- une constante : precipitations[2]
- une expression arithmétique : precipitations[2*i]

L'indice doit être une valeur entière **comprise entre 0 et n-1 (n étant la taille du tableau)**.

**Exemple :** Parcours du tableau precipitations

Algorithme :

```
Pour i allant de 0 à 11 par pas de 1
faire
        Afficher(precipitations[i])
FinPour
```

En langage C :

```
for(i = 0 ; i < 12 ; i = i + 1) {
        printf(''%6.2f\n'', precipitations[i]);
}
```

## 4. Utilisation des tableaux dans les fonctions paramétrées

En langage C, le nom du désigne un pointeur[6] contenant l'adresse du premier élément du tableau.

**Exemple :** Fonction d'affichage d'un tableau

Soit une fonction *afficherTableau()* ayant deux paramètres d'entrée :

- *precipitations* : le tableau de réels ;
- *n :* le nombre d'éléments du tableau.

```
void afficherTableau(float precipitations[], int n) ;
```

Ainsi, on passe en paramètre le nom du tableau et son nombre d'éléments (c'est le seul moyen pour que la fonction `afficherTableau` connaisse la taille du tableau).

**Exemple :** Appel de la fonction d'affichage d'un tableau

```
afficherTableau(precipitations, 12);
```

## 5. Initialisation par défaut d'un tableau

Comme toute variable d'un type de base, un tableau peut être initialisé explicitement au moment de sa déclaration.

**Exemple :** Initialisation du tableau de précipitations

```
float precipitations[12] = {66.0, 72.6, 82.8, 107.2, 94.1, 63.4, 74.9,
91.4, 94.1, 80.3, 70.7};
```

Les valeurs utilisées pour l'initialiser doivent toujours être des constantes. De plus, il est possible d'omettre la dimension du tableau pour ce type de déclaration. L'expression suivante est équivalente :

```
float precipitations[] = {66.0, 72.6, 82.8, 107.2, 94.1, 63.4, 74.9, 91.4,
94.1, 80.3, 70.7};
```

[6] La notion de pointeur sera vue plus loin : un pointeur est une variable contenant une adresse mémoire

Le compilateur détermine la dimension du tableau en fonction du nombre de valeurs énumérées.

## Exercice 1 : Gestion de notes

Le but de cet exercice est d'implanter un programme de gestion des notes d'un groupe d'étudiants. Ce programme doit faciliter l'impression des statistiques concernant les notes obtenues par les étudiants pour un examen. Pour cet exercice, nous supposons que le nombre d'étudiants est fixe et que les notes doivent appartenir à l'intervalle [0 ; 20]. Etant donné que le programme a pour objectif de fournir des statistiques, nous ne tiendrons pas compte du nom des étudiants ; seules les notes nous intéressent.
Les statistiques attendues sont :

- La note la plus basse ainsi que la note la plus élevée ;
- La moyenne, la variance et l'écart-type ;

Pour rappel (voir bibliothèque standard math.h[2]) :

Variance :

$$\frac{\sum(x-\bar{x})^2}{n}$$

Ecart-Type :

$$\sqrt{\frac{\sum(x-\bar{x})^2}{n}}$$

**Jeu de tests**

Le programme réalisé devra être testé. Voici un exemple de test :

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 12,0 | 13,5 | 8,5 | 14,7 | 6,0 |

Quelles sont les valeurs attendues pour :

| | |
|---|---|
| Note la plus basse | |
| Note la plus élevée | |
| Moyenne des étudiants | |
| Variance | |
| Ecart-type | |
| Rang dans le tableau de la valeur 13,5 | |
| Rang dans le tableau de la valeur 10,5 | |

Avant d'aborder les questions suivantes, construire trois autres tests en justifiant leur choix.

### Partie obligatoire[7]

1. Ecrire une fonction *afficherNotes()* qui affiche à l'écran toutes les notes obtenues à l'examen.

2. Ecrire une fonction *minimumNote()* qui prend en entrée un tableau de notes et sa taille et qui retourne la note la plus basse obtenue à l'examen.

3. Ecrire une fonction *maximumNote()* qui prend en entrée un tableau de notes et sa taille et qui retourne la note la plus élevée obtenue à l'examen.

4. Ecrire une fonction *calculeMoyenne()* qui prend en entrée un tableau de notes et sa taille et qui retourne la moyenne des notes.

5. Ecrire une fonction *calculeVariance()* qui prend en entrée un tableau de notes et sa taille et qui retourne la variance.

6. Ecrire une fonction *calculeEcartType()* qui prend en entrée un tableau de notes et sa taille et qui retourne l'écart-type.

7. Ecrire une fonction *rechercherValeur()* qui prend en entrée un tableau de notes, sa taille et une valeur à rechercher. Cette fonction retourne la position de la valeur dans le tableau, -1 si la valeur n'a pas été trouvée.

### Partie optionnelle

Le but de cette partie est d'afficher graphiquement la répartition des notes. Cet affichage se fera sous la forme d'histogrammes (horizontal et vertical). Le principe est d'afficher autant de croix qu'il y a de notes par intervalle de deux points.

1. Ecrire une fonction qui permet d'afficher un histogramme horizontal. Le résultat attendu est le suivant pour les notes :
   0 ; 13.5 ; 8.5 ; 13.7 ; 20 ; 12 ; 8.5 ; 17 ; 11 ; 10 ; 9.5 ; 4 ; 14 ; 13.5 ; 12 ; 1 ; 15 ; 10.5 ; 7.5 ; 9.5.

```
[  0 ;  2 ] :   *       *
]  2 ;  4 ] :   *
]  4 ;  6 ] :
]  6 ;  8 ] :   *
]  8 ; 10 ] :   *       *       *       *       *
] 10 ; 12 ] :   *       *       *       *
] 12 ; 14 ] :   *       *       *       *
] 14 ; 16 ] :   *
] 16 ; 18 ] :   *
] 18 ; 20 ] :   *
```

2. Ecrire une fonction qui permet d'afficher un histogramme vertical. Le résultat attendu est le suivant pour les mêmes notes :

```
                                        *
                                *       *       *
                                *       *       *
    *                           *       *       *
    *       *           *       *       *       *       *       *       *
-------------------------------------------------------------------------------------------
[ 0 ;  2] ] 2 ;  4] ] 4 ;  6] ] 6 ;  8] ] 8 ; 10] ]10 ; 12] ]12 ; 14] ]14 ; 16] ]16 ; 18] ]18 ; 20]
```

[7] Pour utiliser la bibliothèque standard math.h penser à ajouter dans le Makefile l'option **–lm** lors de l'édition des liens; la documentation des bibliothèques standard peut être trouvée sur internet

## Exercice 2 : Polynômes

Un polynôme à coefficients réels de degré N-1 est défini par la donnée d'un **tableau A de taille N de ses coefficients** : A[0] + A[1]*x + … A[N-1]*$x^{N-1}$. Certains coefficients peuvent donc être nuls.

Ecrire une fonction *float valeurPolynome(float A[], int N, float x0)* qui reçoit en paramètres le tableau des coefficients du polynôme initialisé, le degré du polynôme et qui calcule sa valeur pour x = x0. **Pour réaliser cette fonction on ne calculera pas la puissance $x^N$ pour chaque terme du polynôme (pas d'utilisation de la fonction pow, pas de code calculant explicitement la puissance).** Indication : $6x^3 + 5x^2 + 3x + 1 = 1 + x * (3 + x * (5 + 6x))$.

# Passage de paramètres

Un programme contient obligatoirement une fonction principale : la fonction `main`. Celle-ci peut disposer d'arguments lui permettant de recueillir des informations venant de l'utilisateur.

La fonction `main` a obligatoirement une des signatures suivantes (selon que l'on souhaite, ou pas, qu'elle dispose d'arguments) :

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

Si la fonction main est de la deuxième forme, les arguments ont la signification suivante :

- argc : le nombre de paramètres fournies par l'utilisateur (entier positif ou nul).
- argv[] : contient les paramètres sous forme de chaînes de caractères. argv[0] contient toujours le nom du programme.

**Exemple :** Contenu des variables argc et argv

```
machine@esisar>./monProgramme 12 toto 20.5
```

| | |
|---|---|
| argc | 4 |
| argv[0] | ./monProgramme |
| argv[1] | 12 |
| argv[2] | toto |
| argv[3] | 20.5 |

**Exemple :** Utilisation des variables argc et argv

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    int i;
    printf("Nombre d'arguments : %d\n", argc);
    for(i = argc-1 ; i > 0 ; i--) {
          printf("%s\n", argv[i]);
    }
    return 0;
}
```

## Exercice 3

Tester le programme donné dans l'exemple précédent. Quels sont les résultats obtenus ?

# Tableaux à deux dimensions

## 6. Déclaration et initialisation d'un tableau à deux dimensions

La déclaration d'un tableau à deux dimensions se fait ainsi :

```
TYPE nomTableau[n][m]
```

Où :

- TYPE représente le type des éléments du tableau ;
- nomTableau est l'identificateur de la variable de type tableau ;
- n et m sont les dimensions du tableau.

**Exemple :**

```
float T[3][4] ;
```

déclare un tableau de 3 x 4 éléments de type réel, notés :

| | | | |
|---|---|---|---|
| T[0][0] | T[0][1] | T[0][2] | T[0][3] |
| T[1][0] | T[1][1] | T[1][2] | T[1][3] |
| T[2][0] | T[2][1] | T[2][2] | T[2][3] |

Comme pour les tableaux à un indice, il est possible d'initialiser par défaut un tableau à deux indices.

**Exemple :** Les deux déclarations suivantes sont équivalentes :

```
int T[3][4] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 } } ;
int T[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 } ;
```

## 7. Utilisation d'un tableau à deux dimensions

Les tableaux à deux indices s'utilisent comme les variables simples ou comme les tableaux à un indice :

```
nomTableau[indice1][indice2]
```

où indice1 et indice 2 sont les indices, c'est-à-dire des expressions à valeurs entières comprises respectivement entre 0 et n-1 pour le premier indice et entre 0 et m-1 pour le deuxième.

## 8. Utilisation d'un tableau dans une fonction

Comme pour un tableau à un indice, pour passer en paramètre un tableau, on donne le pointeur sur le tableau ainsi que son nombre de colonnes et de lignes.

**Exemple :** Déclaration de la fonction d'affichage d'un tableau

```
void afficherTableau(int T[3][4], int nbColonnes, int nbLignes)
```

Il faut noter que *T* est défini avec sa taille (allouée en mémoire).

## 9. Tableau à *n* dimensions

Il est possible de composer à volonté les déclarations de telle sorte que l'on peut obtenir des tableaux de tableaux de tableaux…

**Exemple :** Un tableau à trois indices déclaré ainsi :

```
double tf[2][3][4] ;
```

- tf[2][3][4] est un double ;
- tf[2][3] est un tableau de 4 double ;
- tf[2] est un tableau de 3 tableaux de 4 double ;
- tf est est un tableau de 2 tableaux de 3 tableaux de 4 double.

## Exercice 4 : Matrices

Les matrices sont un exemple de tableaux à deux dimensions. Nous souhaitons réaliser un module qui permette de créer et d'initialiser une matrice, d'afficher une matrice, d'additionner deux matrices, de calculer la transposée d'une matrice et le produit de deux matrices.

1. Ecrire un programme qui crée une matrice de taille fixe, qui l'initialise et qui l'affiche.
2. Quels sont les résultats attendus dans les opérations suivantes?

Addition de deux matrices :

$$\begin{pmatrix} 1 & 0 & 4 \\ -1 & 2 & -5 \end{pmatrix} + \begin{pmatrix} 2 & -3 & 1 \\ 5 & 8 & 2 \end{pmatrix} = ?$$

- Quelles sont les contraintes pour réaliser une addition de deux matrices ?
- Quelle est la taille de la matrice résultat ?

Matrice transposée :

$$A = \begin{pmatrix} 1 & 0 & 4 \\ -1 & 2 & -5 \end{pmatrix} \qquad \text{et} \qquad {}^tA = ?$$

- Quelle est la taille de la matrice résultat ?

Produit matriciel :

$$\begin{pmatrix} 1 & 0 & 4 \\ -1 & 2 & -5 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & -1 \\ 3 & 4 \end{pmatrix} = ?$$

- Quelles sont les contraintes pour réaliser un produit de deux matrices ?
- Quelle est la taille de la matrice résultat ?

Pour chacune des questions suivantes, écrire au préalable des tests qui serviront à vérifier la correction des fonctions.

3. Ecrire une fonction *afficherMatrice()* qui affiche une matrice.
4. Ecrire une fonction *additionnerMatrices()* qui calcule la somme de deux matrices.
5. Ecrire une fonction *transposee()* qui calcule la matrice transposée.
6. Ecrire une fonction *produitMatriciel()* qui réalise un produit matriciel.

# Chapitre 4 : Entrées/Sorties, Fichiers, Chaînes de caractères

Objectifs :

- Maîtriser les entrées/sorties au clavier et à l'écran ;
- Savoir manipuler les fichiers de données ;
- Savoir manipuler des chaînes de caractères.

## Entrées/Sorties

### 1. Flux

En C, un *flux* est un canal destiné à transmettre ou à recevoir de l'information. Il peut s'agir de périphériques ou de fichiers. La bibliothèque *stdio.h* définit trois types de flux pour les périphériques de communication de base :

- *stdin* pour l'entrée standard ;
- *stdout* pour la sortie standard ;
- *stderr* pour la sortie d'erreur.

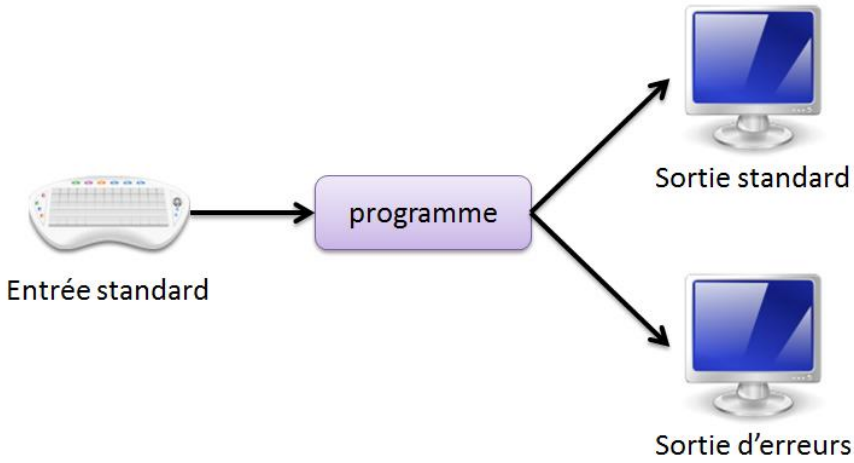



Figure 6 : Les différents flux.

Les fonctions qui sont présentées dans la première partie de ce TP permettent d'écrire et de lire sur ces différents flux. La suite du TP concerne ces mêmes actions mais pour les fichiers.

### 2. Les fonctions printf() et scanf()

Ces fonctions ont déjà été présentées lors du TP1. Pour rappel, le détail de ces fonctions est donné dans les sections 1.16 et 3.14 du document « Introduction au langage C ».

Ces fonctions nécessitent l'utilisation de la bibliothèque stdio.h. Leurs prototypes sont :

```
int printf(const char * format, liste d'expressions) ;
int scanf(const char * format, liste d'expressions) ;
```

La fonction *printf()* a été prévue pour afficher des caractères, des chaînes de caractères, des numériques ou des pointeurs. La fonction *printf()* fournit en retour le nombre de caractères qu'elle a écrits lorsque l'opération s'est bien déroulée.

La fonction *scanf()* a été prévue pour lire des caractères, des chaînes de caractères, des numériques ou des pointeurs; pour cela, on lui fournit les adresses des variables dans lesquelles on souhaite placer les valeurs lues – c'est la raison de l'utilisation de l'esperluette « & » devant les noms des variables[8]. La fonction *printf()* fournit en retour le nombre de valeurs lues convenablement.

Formats possibles :

| | |
|---|---|
| %c | 1 caractère |
| %d | 1 entier |
| %f | 1 réel, on peut préciser le nombre de décimales |
| %e | 1 réel en notation exponentielle |
| %s | 1 chaîne de caractères |

**Exemple :** Lecture et écriture

```
#include <stdio.h>


int main(int argc, char * argv[]) {
    int n ;
    printf("Donnez une valeur pour n :") ;
    scanf("%d", &n) ;
    printf("merci pour la valeur %d\n", n) ;
}
```

## Les fichiers

Dans cette partie, nous présentons les fonctions principales nécessaires pour la manipulation des fichiers, en particulier pour les fichiers en mode texte. Pour en savoir plus sur les fichiers et la manipulation des fichiers binaires, vous pouvez vous référer au chapitre 5 page 81 du document « *Introduction au langage C* » de B. Cassagne.

Les fonctions présentées nécessitent l'utilisation de la bibliothèque *stdio.h*. Il faut donc l'inclure avec la directive `#include <stdio.h>`.

### 3. Ouverture d'un fichier : fopen()

Le rôle de la fonction *fopen()* est d'ouvrir le fichier concerné dans *le mode* indiqué. Le prototype de la fonction est le suivant :

```
FILE * fopen(const char * nomFichier, const char * mode) ;
```

[8] Voir plus loin le TP sur les pointeurs, &a est une expression qui renvoie l'adresse mémoire de la variable a

Elle fournit comme valeur de retour un flux (pointeur sur une structure de type prédéfini FILE) qui sert à toutes les opérations de lecture ou écriture dans le fichier. En cas d'échec, un pointeur null est retourné. Les causes d'un tel échec peuvent être :

- le fichier est inexistant, en cas d'ouverture en lecture ou en mise à jour ;
- on ne dispose pas des droits d'accès voulus au fichier ou au répertoire ;
- …

La valeur de mode fait intervenir trois sortes d'indicateurs qu'il est possible de combiner.

| **Type d'indicateur** | **Valeurs** | **Signification** |
|---|---|---|
| Principal (obligatoire) | r | Ouverture en lecture seule d'un fichier existant. |
| | w | Ouverture en écriture seule dans un nouveau fichier ou **écrasement** d'un fichier existant. |
| | a | Ouverture en écriture à la fin d'un fichier existant ou d'un nouveau. |
| Mode d'ouverture (optionnel) | b | Manipulation d'un fichier binaire. En l'absence de cet indicateur, manipulation d'un fichier texte. |
| Mise à jour (optionnel) | + | Autorise à la fois la lecture et l'écriture. |

**Exemple :** Ouverture d'un fichier texte « monFichier.txt » en lecture/écriture.

```
FILE * fic = fopen("monFichier.txt", "r+") ;
```

Combinaisons possibles des indicateurs :

| | | | |
|---|---|---|---|
| **r** | Ouverture d'un fichier texte en lecture | **rb** | Ouverture d'un fichier binaire en lecture |
| **w** | Ouverture d'un fichier texte en écriture | **wb** | Ouverture d'un fichier binaire en écriture |
| **a** | Ouverture d'un fichier texte en écriture à la fin | **ab** | Ouverture d'un fichier binaire en écriture à la fin |
| **r+** | Ouverture d'un fichier texte en lecture/écriture | **rb+** | Ouverture d'un fichier binaire en lecture/écriture |
| **w+** | Ouverture d'un fichier texte en lecture/écriture | **wb+** | Ouverture d'un fichier binaire en lecture/écriture |
| **a+** | Ouverture d'un fichier texte en lecture/écriture à la fin | **ab+** | Ouverture d'un fichier binaire en lecture/écriture à la fin |

## 4. Fermeture d'un fichier : fclose()

La fonction *fclose()* permet de fermer un fichier. Son prototype est le suivant :

```
int fclose(FILE * fic) ;
```

**Exemple :** Fermeture du fichier « *monFichier.txt* » qui a été ouvert dans l'exemple précédent.

```
fclose(fic) ;
```

## 5. Ecriture dans un fichier en mode texte : fprintf()

Pour écrire dans un fichier de façon formatée, il faut utiliser la fonction *fprintf()* qui est, comme la fonction *printf()*, une fonction à arguments variables. La fonction *fprintf()* prend en premier argument le flux dans lequel le texte sera enregistré, les arguments suivants sont les mêmes que ceux de la fonction *printf()*.

**Exemple :** Ecrire les 10 premiers entiers dans le fichier « monFichier.txt » qui a été ouvert dans l'exemple précédent.

```
for(i = 0 ; i < 10 ; i++) {
    fprintf(fic, "Valeur = %d\n", i) ;
}
```

## 6. Lecture dans un fichier en mode texte : fscanf()

La lecture dans un fichier se fait avec la fonction *fscanf()* qui est, comme la fonction *scanf()*, une fonction à arguments variables. La fonction *fscanf()* prend en premier argument le fichier dans lequel la lecture sera faite, les arguments suivants sont les mêmes que ceux de la fonction *scanf()*.

La lecture dans un fichier nécessite de détecter la fin du fichier. La fonction *feof()* prend la valeur vrai (non nul) lorsque la fin du fichier a été atteinte. Son prototype est le suivant :

```
int feof(FILE * fic) ;
```

Il faut noter que la fonction feof() ne travaille pas par anticipation, ce qui signifie qu'il n'est pas suffisant d'avoir lu le dernier octet du fichier pour que cette condition prenne la valeur vrai. Il est nécessaire d'avoir tenté de lire au-delà.

**Exemple :** Lecture d'entiers dans un fichier :

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

int main(int argc, char * argv[]) {
    FILE * fic;
    int nb;
    /* Ouverture du fichier */
    fic = fopen("monFichier.txt", "r");
    if(fic == NULL) {
          perror("Probleme ouverture fichier monFichier.txt");
          exit(1);
    }

    /* Lecture dans le fichier */
    while(!feof(fic)) {
          fscanf(fic, "%d", &nb);
          printf("Valeur lue = %d\n", nb) ;
```

```
    }

    /* Fermeture du fichier */
    fclose(fic);

    return 0;
}
```

**Remarque :** Les fonctions ci-dessus fonctionnent également sur les flux d'entrée/sortie introduits dans la section 1. Il suffit de passer en premier paramètre stdin, stdout ou stderr (par exemple `fprintf(stdout, "Valeur = %d\n", i) ;`).

### Exercice 1 : Manipulation de fichier et tri de tableau

Réaliser un module qui permet de lire dans un fichier une suite d'entiers puis de les trier avant de les enregistrer dans un autre fichier.

Dans un premier temps, préparer plusieurs tests sous la forme de plusieurs fichiers texte contenant des suites d'entiers différentes.

Ecrire les fonctions suivantes :

1. `int lireDonnees(char nomFichier[], int T[])` qui lit les données dans le fichier nommé *nomFichier* des entiers, puis les stocke dans un tableau *T*. La valeur de retour est le nombre d'entiers qui ont été lus (c'est-à-dire le nombre d'éléments utiles du tableau). On suppose la taille du tableau suffisante pour contenir tous les entiers.
2. `void afficherTableau(int T[], int nb)` qui affiche le contenu du tableau *T* qui comprend *nb* éléments.
3. `void triABulles(int T[], int nb)` qui trie le tableau *T* de *nb* éléments avec la méthode du tri à bulles[9] et qui retourne le tableau trié.
4. `void enregistrerDonnees(char nomFichier[], int T[], int nb)` qui enregistre les *nb* valeurs du tableau *T* dans le fichier nommé *nomFichier*.

Pour tester vos fonctions, réaliser un programme principal. L'exécution de ce programme doit prendre en paramètre le nom du fichier pour la lecture des données ainsi que celui pour l'enregistrement des valeurs.

*Conserver chaque fichier de données d'entrée créé pour le test. Préciser, dans un fichier à part (explicationsTest.txt), le choix des données d'entrée fait et pourquoi celles-ci testent correctement la fonction de tri.*

**Exemple d'exécution :**

```
machine@debian$./progTri donnees.txt resultat.txt
```

[9] Le principe du tri à bulles est le suivant : on imagine que le tableau est vertical, le plus petit indice en haut, et que les éléments les plus petits sont "moins lourds" que les autres. On effectue des passages successifs sur le tableau de bas en haut. À chaque étape, si deux éléments sont en ordre inverse (plus "léger" dessous), on les inverse. Les éléments "légers" remontent comme des bulles vers la surface tandis que les éléments "plus lourds" tombent lentement vers le fond. L'algorithme se termine après n-1 passages.

# Les chaînes de caractères

En langage C, on fait la distinction entre un tableau de caractères et une "chaîne de caractères" : une chaîne de caractères est contenue dans un tableau de caractères; par convention, sa fin est indiquée par le caractère particulier '\0' (caractère nul, différent de '0'). Le respect de cette convention est notamment indispensable si l'on souhaite utiliser les fonctions de la librairie string.h.

## 7. Déclaration et initialisation d'une chaîne de caractères

Une chaîne de caractères peut être allouée de manière statique[10] comme nous l'avons vu avec les tableaux :

```
char ch[TAILLE]
```

D'après cette déclaration, ce tableau peut contenir une chaîne **d'au plus TAILLE-1** caractères (plus le caractère '\0').

L'initialisation d'une chaîne de caractères peut se faire au moment de sa déclaration de deux façons différentes : soit comme une constante chaîne soit par énumération des caractères.

**Exemple :**

```
char ch[11] = "bonjour" ;
char ch[11] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'} ;
```

Ces deux déclarations sont équivalentes. Pour la seconde, il est important de noter qu'il faut ajouter le caractère **'\0'** à la fin alors qu'il est ajouté automatiquement dans la première déclaration. Le résultat de cette initialisation est le suivant :

| b | o | n | j | o | u | r | \0 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

## 8. Modification du contenu d'une chaîne de caractères

Une chaîne de caractères peut être modifiée caractère par caractère (comme les cellules d'un tableau).

**Exemple :** Transformer la chaîne contenue dans `ch` de « bonjour » en « bonsoir ».

```
ch[3] = 's';
ch[5] = 'i';
```

## 9. Manipulation de chaînes de caractères

La bibliothèque `string.h` contient un ensemble de fonctions permettant de manipuler des chaînes de caractères :

- la fonction *strlen()* renvoie la longueur d'une chaîne de caractères ;
- la fonction *strcpy()* permet de copier une chaîne de caractères dans une autre ;
- la fonction *strstr()* cherche la première occurrence d'un caractère dans une chaîne ;
- …

[10] Il est également possible de faire une allocation dynamique (cf. TP 7)

### Exercice 2 : Fonctions sur les chaînes de caractères

Réaliser un module offrant les fonctions suivantes. Pour chaque fonction, préciser au préalable les tests utilisés.

1. Ecrire deux fonctions *myStrlen()* et *myStrcpy()* qui réalisent le même travail que *strlen()* et *strcpy()* – on peut trouver la spécification de ces dernières dans *man*.
2. Ecrire une fonction *affichagerEnHexadecimal()* qui affiche à l'écran le mot en utilisant la valeur hexadécimale de chacune des lettres. Cette fonction prend en entrée un mot.
3. Ecrire une fonction *affichagerEnDecimal()* qui affiche à l'écran le mot en utilisant la valeur décimale de chacune des lettres. Cette fonction prend en entrée un mot.
4. Ecrire une fonction *mettreEnMajuscule()* qui transforme les caractères en minuscule en caractères en majuscule. Les autres caractères restent inchangés. Cette fonction prend en entrée une chaîne de caractères en minuscule et retourne une chaîne en majuscule.
5. Ecrire une fonction *mettreEnMinuscule()* qui transforme les caractères en majuscule en caractères en minuscule. Les autres caractères restent inchangés. Cette fonction prend en entrée une chaîne de caractères en majuscule et retourne une chaîne en minuscule.
6. Ecrire une fonction *transformerMinMaj()* qui change la casse des caractères. Les caractères en minuscule passent en majuscule et inversement. Cette fonction prend en paramètre une chaîne de caractères et retourne la chaîne de caractères modifiée.
7. Ecrire une fonction *retournerMot()* qui renvoie un mot écrit à l'envers (exemple : oiseau\0 devient uaesio\0). Cette fonction prend en entrée une chaîne de caractères et retourne la chaîne de caractères modifiée.
8. Ecrire une fonction *rechercherCaractereG()* qui renvoie la première occurrence d'un caractère dans une chaîne de caractères en partant de la gauche. Cette fonction prend en entrée la chaîne de caractères ainsi que le caractère recherché. Elle retourne la position dans la chaîne si le caractère est présent, -1 si le caractère n'a pas été trouvé.
9. Ecrire une fonction *rechercherCaractereD()* qui renvoie la première occurrence d'un caractère dans une chaîne de caractères en partant de la droite. Cette fonction prend en entrée la chaîne de caractères ainsi que le caractère recherché. Elle retourne la position dans la chaîne si le caractère est présent, -1 si le caractère n'a pas été trouvé.
10. Ecrire une fonction *estPalindrome(char * mot, int d, int f)* qui détermine si un palindrome (mot qui se lit de la même manière dans les deux sens ; p.ex. « eluparcettecrapule ») est présent entre les indices d et f de la chaîne. Cette fonction retourne la valeur 1 si c'est le cas, 0 sinon.
11. Ecrire une fonction *comparerChaine()* qui permet d'effectuer une comparaison de deux chaînes. Elle prend en paramètre d'entrée deux chaînes de caractères et elle retourne :
    - la valeur 0 en cas d'égalité : toto == toto ;
    - une valeur positive si la première chaîne de caractères est supérieure alphabétiquement à la seconde : toto > titi ;

- une valeur négative si la première chaîne de caractères est inférieure alphabétiquement à la seconde : titi < toto.

12. Ecrire une fonction int valeurDecimale() qui reçoit en paramètre une chaîne de caractères correspondant à un nombre (par exemple "2546") et qui retourne la valeur décimale correspondante (0 si la chaîne est vide).
13. Ecrire une fonction voit intVersChaine() qui reçoit en paramètre un entier (par exemple 2546) et une chaîne de caractères vide et remplit cette dernière avec la représentation textuelle de l'entier ("2546").

# Chapitre 5 : Structures, piles, listes

Objectifs :
- S'initier à la manipulation de structures de données fondamentales

## Structures

### Notions de base

Lire les sections 6.1 à 6.4 du polycopié ainsi que les sections 9.8 et 9.9.

### Exercice 1 : Fractions

On considère le type Fraction (portion de code à insérer dans un nouveau fichier Fraction.h) :

```
typedef struct {
  int num ; /* numérateur */
  int den ; /* dénominateur */
} Fraction ;
```

Nous souhaitons rédiger quatre procédures réalisant les quatre opérations arithmétiques sur des fractions et affichant le résultat. Le profil des procédures est le suivant :

```
void addFraction(Fraction f1, Fraction f2) ;
void subFraction(Fraction f1, Fraction f2) ;
void mulFraction(Fraction f1, Fraction f2) ;
void divFraction(Fraction f1, Fraction f2) ;
```

Chacune de ces procédures calcule la fraction correspondant à l'opération entre ses deux paramètres puis *affiche le résultat* sous la forme d'une fraction réduite (voir exemple ci-dessous). Pour cela, on utilisera une fonction qui calcule le pgcd de deux entiers. Ecrire ces procédures (dans nouveau fichier Fraction.c) et compléter la fonction main() (à insérer dans un nouveau fichier main.c) permettant de les tester comme suit (*entrées utilisateur en gras*):

*Entrer deux fractions :* ***3/4 4/8***
*Entrer une opération (+, -, /, *) :* *******
*Le résultat est 3/8.*

```
int main(int argc, char* argv[]){
      Fraction fa, fb ;
      printf("Entrer deux fractions :") ;
      scanf("%d/%d %d/%d", &fa.num, &fa.den, &fb.num, &fb.den) ;
      ...
}
```

### Exercice 2 : Polynômes

On considère les définitions de type suivantes :

```
#define NBMAX 100
typedef struct {
  float coeff ;/* coefficient du terme */
  int degre ;  /* degré du terme */
} Terme ;
typedef Terme Polynome[NBMAX];
```

Le type `Terme` représente un terme d'un polynôme. Un polynôme (type Polynome) est représenté par un tableau de termes *classés dans l'ordre décroissant de leurs degrés*. Il n'y a *pas de terme avec un coefficient nul (cela signifie que le polynôme* $2x^{154}+2x$ *est constitué de seulement deux termes). La fin d'un polynôme est marquée par un terme dont le degré est négatif.*
Rédiger une procédure réalisant la somme de deux polynômes ainsi qu'une fonction main la testant. On choisira, pour cela, plusieurs polynômes et on expliquera leurs choix pour tester la procédure :

```
void addPolynomes(Polynome p1, Polynome p2, Polynome res) ;
```

# Piles, files, listes

## Notions de base

Nous supposons que les structures de pile, file et liste ont été étudiées en algorithmique. Les exercices ci-dessous permettent d'aborder l'utilisation de ces structures en C (*l'utilisation de pointeurs est volontairement évitée ou « masquée »*).

## Exercice 3 : Calcul post-fixe

On veut évaluer des expressions arithmétiques écrites en notation polonaise inversée (ou notation post-fixe). Rappel du principe : tout opérateur arithmétique n'est pas placé entre ses arguments (comme avec la notation courante du type 4 + 7) mais après ses arguments. Ainsi 4 + 7 se note 4 7 +. Par exemple, l'expression (4+2)x5/(6-7) se note en post-fixe 4 2 + 5 x 6 7 - /. L'évaluation d'une telle expression se base sur l'utilisation d'une pile.
Écrire une fonction qui prend en paramètre une telle expression formée des quatre opérateurs +, -, x, / et des nombres entiers de 0 à 9, stockée dans un tableau de N caractères et qui retourne la valeur de l'expression. **On suppose que l'expression est correcte.**
*Pour réaliser cet exercice, vous trouverez dans le répertoire* pile *les fichiers pile.h et pile.c (que vous devez utiliser mais pas modifier), fournissant une réalisation simple d'une pile d'entiers (le fichier* main.c, *donné à titre d'exemple, en illustre l'utilisation).*

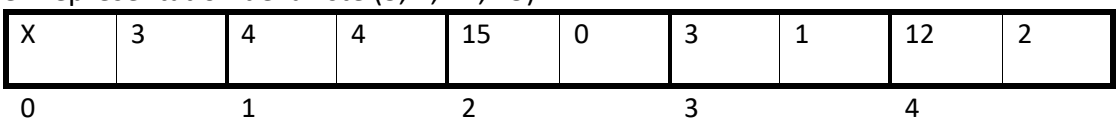
## Exercice 4 : Liste chaînée

On souhaite gérer une liste chaînée d'entiers, implémentée à l'aide d'un tableau. On considère les déclarations suivantes (présentes dans le fichier `listeTableau.h` fourni dans le répertoire `listeTableau`) :

```
typedef struct {
      int valeur ;
      int suivant ;
} element ;
typedef element* liste ;
```

Nous adoptons le principe de représentation suivant :
Soit L une variable de type liste.
L[0].valeur est non significatif ;
L[0].suivant est l'indice du tableau correspondant au premier élément de la liste.
L[i].suivant est l'indice du tableau contenant le successeur de celui situé à l'indice i.
L[i].suivant = 0 signifie que l'élément d'indice i est le dernier de la liste.
L[i].suivant = -1 signifie que l'élément d'indice i est inoccupé (donc, on ne devrait pas avoir d'élément j dans le tableau tel que L[j].suivant = i).

Ainsi, dans un tableau de taille N on peut représenter une liste d'au plus N-1 éléments.
Exemple : représentation de la liste (3, 4, 12, 15)




Compléter le fichier `listeTableau.c` du répertoire `listeTableau` avec les fonctions C permettant de gérer la liste et tester en complétant le programme `main.c`. *Ce dernier doit tester plusieurs cas (liste non exhaustive) : insertion et suppression et tête, en fin et en milieu de liste, insertion d'un élément déjà présent, suppression d'un élément qui n'existe pas… Justifier, sous forme de commentaires dans main.c, le choix de vos tests.S*

- *int elementLibre(liste l)*
- *void creerListeVide(liste l)* : crée une liste vide (initialise le premier élément du tableau ainsi que les éléments inoccupés).
- *void afficherListe(liste l)* : affiche tous les éléments de la liste dans l'ordre.
- *void insererElement(int x, liste l)* : en supposant l triée, insère x dans l, **en maintenant l triée**.
- *void supprimerElement(int i, liste l)* : supprime le i-ème élément de l.

**Question optionnelle** : écrire une fonction permettant de "compacter" le tableau contenant une liste de P éléments de sorte qu'elle occupe les P+1 premières cases du tableau.

- *void compacter (liste l)*

# Chapitre 6 : Récursivité
# Utilisation d'un débogueur

Objectifs :
- Pratiquer la programmation récursive en C
- Se familiariser avec l'utilisation d'un débogueur

## Débogueur ddd

### 1. Eléments pour commencer

#### 1.1. Principe et compilation

Un débogueur est un outil permettant d'exécuter un programme de manière contrôlée en observant, en particulier, l'évolution de ses variables. L'utilisation d'un débogueur n'est possible que si le code exécutable produit est enrichi avec des instructions complémentaires rendant ce contrôle possible. Le compilateur `gcc` fait cet enrichissement au moyen de l'option `-g`. La commande de compilation à utiliser lorsqu'on veut utiliser un débogueur est donc :

```
gcc -g -c prog.c -o prog.o
```

Nous utiliserons le débogueur `ddd` en exécutant : `ddd prog`

Il y a trois fonctions essentielles dans un débogueur :

- **La définition d'un point d'arrêt (breakpoint)** : pour cela on sélectionne la ligne du code où l'on souhaite que l'exécution s'arrête (soit N le numéro de cette ligne) et on pose un point d'arrêt ; la commande `run` démarre l'exécution du programme et la suspend à la ligne N.
- **L'exécution pas à pas** : à partir de la ligne N, on peut exécuter la commande `next`, auquel cas l'instruction de la ligne N sera exécutée et le débogueur passera à la prochaine instruction du programme où il suspendra son exécution en attendant une nouvelle commande. La commande `step` exécutera également l'instruction de la ligne N mais, contrairement à `next`, si N est un appel de fonction, le débogueur entrera dans cette fonction et s'arrêtera à sa première lige de code.
- **L'observation des variables** : on utilise les commandes `display` et `print` pour observer la valeur des différentes variables du programme.

#### Exercice 1 : Post-fixe avec ddd

Reprendre l'exercice sur le calcul post-fixe du TP 5 et modifier votre `Makefile` de sorte à rendre l'utilisation de `ddd` possible. En utilisant cet exemple, pratiquer des exécutions contrôlées avec `ddd` et déterminer l'utilité des commandes suivantes :

`next`, `step`, `stepi`, `nexti`, `print`, `display`, `continue`, `finish`.

## Récursivité

Rappel : Une fonction récursive est une fonction qui a pour propriété de s'appeler elle-même. Le principe de la récursivité est similaire à celui de la récurrence en mathématiques. On identifie un (ou plusieurs) cas de base (i.e. la définition du résultat de la fonction ne nécessite pas d'appel récursif) avant de définir la solution dans le cas général (récursif).

### Exercice 2 : Divers

1. Écrire une fonction récursive calculant le terme de rang n de la suite de Fibonacci.
2. Écrire une fonction récursive calculant le pgcd de deux entiers.
3. Écrire une fonction récursive vérifiant qu'un tableau de caractères contenant un mot de taille n donné est un palindrome.

### Exercice 3 : Tour de Hanoi

On dispose de trois "tours" A, B et C formées d'un empilement plus ou moins grand de disques de telle sorte que chaque disque ait un diamètre inférieur à son prédécesseur. Ainsi, le plus grand disque de chaque tour se situe à leur base et le plus petit sur leur sommet. Au départ, tous les disques (supposons qu'il y en ait n) se trouvent empilés suivant les règles précédemment décrites sur la tour A. L'objectif est de déplacer tous les disques de la tour A vers la tour C en s'aidant uniquement de la tour B, tout en respectant les règles suivantes :

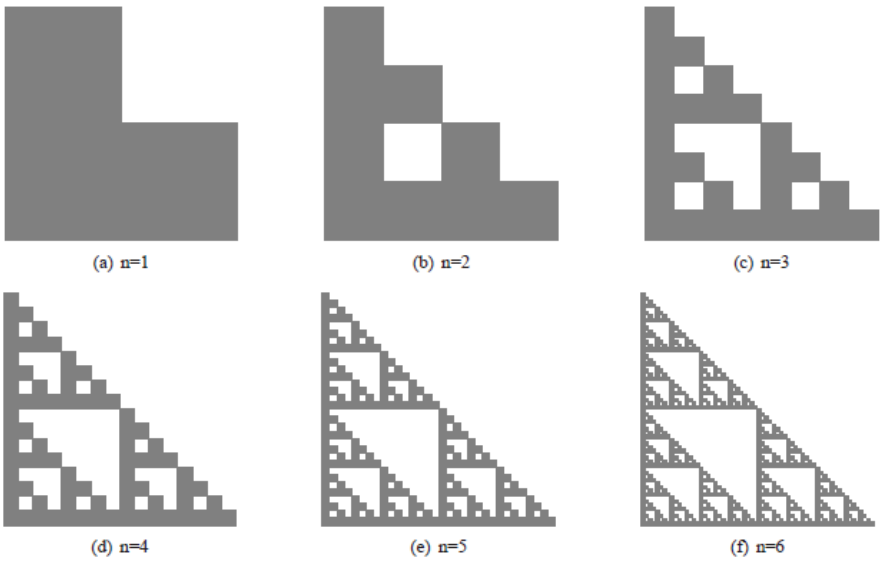
- On ne peut bouger qu'un seul disque par étape et cela doit toujours être le plus petit (celui qui se trouve au sommet).
- À chaque étape, la règle d'organisation des tours décrite précédemment doit être respectée.

a. Écrire une procédure récursive Hanoi recevant un paramètre entier (le nombre de disques initialement empilés sur la tour A) et les noms, dans l'ordre nécessaire, des trois tours ("A", "B" et "C") et affichant tous les déplacements de tours (par exemple "A -> C") jusqu'à ce que le problème soit résolu (i.e. tous les disques empilés sur la tour C).

b. (*optionnel*) En utilisant la bibliothèque graphlib proposer une version ou l'affichage des trois tours se fait de manière graphique.

### Exercice 4 : Dessin d'une forme récursive (fractale)

Une figure fractale est composée de figures identiques à une échelle réduite. Un exemple de telle figure est le triangle de Sierpinski. Le triangle de Sierpinski au niveau 0 est un carré plein, d'une taille T donnée. Pour passer au niveau 1, on décompose le carré en quatre carrés de taille T/2 et on "laisse de côté" le carré en haut à droite. Pour passer au niveau 2, on répète l'opération pour les trois autres carrés et ainsi de suite (voir figure).




Ecrire une procédure récursive à un paramètre entier n dessinant le triangle de Sierpinsky de rang n.

# Chapitre 7 : Les pointeurs

Objectif :

- Maîtriser les notions d'adresse et de pointeur

## Les pointeurs

### 10. Rappel sur les tableaux

La déclaration d'un tableau est faite ainsi :

```
TYPE nomTableau[n]
```

Où :

- TYPE représente le type des données contenues dans le tableau ;
- nomTableau est le nom du tableau ;
- n est le nombre d'éléments du tableau.

Cette déclaration de tableau correspond à :

1. la réservation en mémoire d'une zone de n × (taille du TYPE) octets contigus pour le tableau ;
2. deux informations supplémentaires :
   a. l'adresse du premier octet de la zone réservée ;
   b. le type de l'objet.

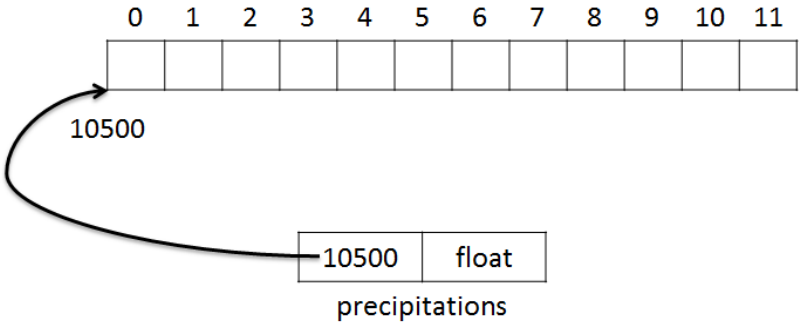
   Ces informations sons constantes : il n'est pas possible d'en modifier leurs valeurs (adresse et type).

**Exemple :**

La déclaration suivante :

```
float precipitations[12] ;
```

correspond à :



## 11. Adresses et pointeurs

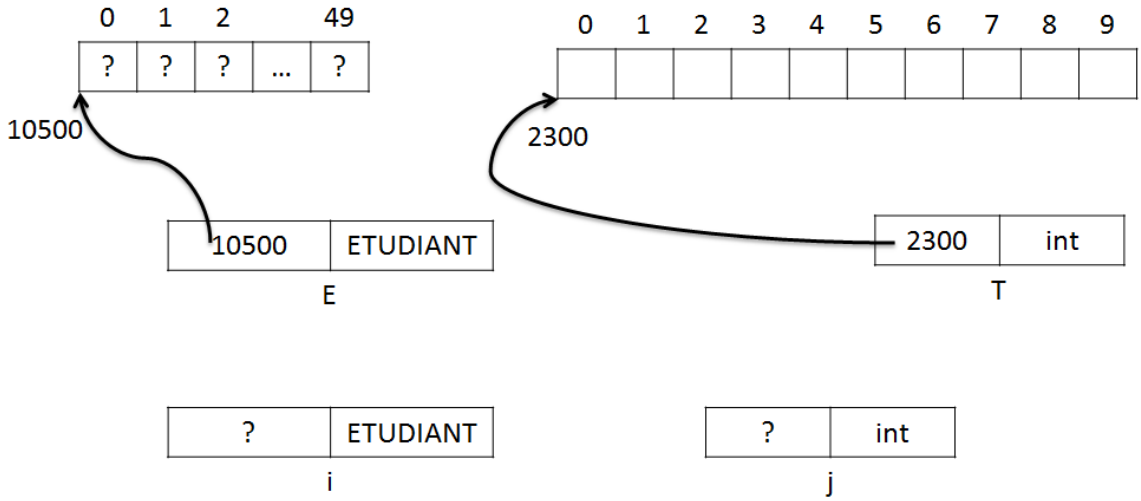
Chaque variable d'un programme C possède une adresse. Une adresse est un entier correspondant à l'emplacement du premier octet de la mémoire occupée par cette variable. Un pointeur est une variable contenant une adresse d'une variable d'un type donné.

**Exemple :**

Supposons que E soit un tableau d'étudiants, T un tableau d'entiers, i un pointeur vers un étudiant et j un pointeur vers un entier.
Les déclarations de ces variables se fait ainsi :

```
ETUDIANT E[50], *i ;
int T[10], *j ;
```

i est un pointeur vers un objet de type ETUDIANT (on suppose que ce type a été défini par ailleurs) ; il est donc susceptible de contenir l'adresse d'un objet de type ETUDIANT.
j est un pointeur vers un objet de type entier (i.e. est susceptible de contenir l'adresse d'un emplacement mémoire occupé par un entier ; on dit que j « pointe vers un entier »).




Les affectations suivantes ont du sens :

- i ← E : affecte le contenu de E à i (E et i ont le même type) ;
- j ← T : affecte le contenu de T à j (T et j ont le même type).

Les affectations suivantes n'ont pas de sens :

- i ← T : impossible car T et i n'ont pas le même type (le type d'un pointeur est constant) ;
- E ← i : impossible car E est constant et ne peut être changé.

## 12. Récupération de l'adresse d'un élément

En langage C, il existe une fonction prédéfinie qui, étant donné un objet, rend l'adresse de cet objet. Cette fonction s'appelle : &.

L'opérateur * permet de manipuler un objet pointé. Si k est un pointeur vers un entier, l'expression *k désigne l'entier pointé.

Un pointeur (de n'importe quel type) contenant la valeur NULL par convention « ne pointe sur rien » (aucune adresse ne lui est associée).

**Exemple :**

Soient x un entier et p un pointeur sur un entier. Ils sont déclarés et initialisés ainsi :

```
int x, *p ;
x = 5 ;
```

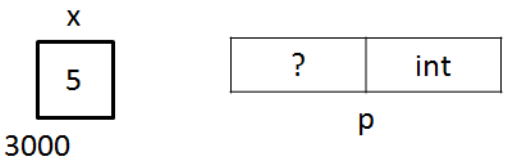



Figure 7 : Etat après la déclaration et l'affectation.

L'opération p ← &x affecte l'adresse de x à p. Après cette opération, *p aura pour valeur 5.

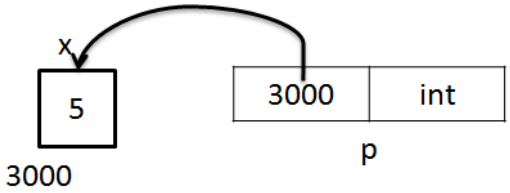



Figure 8 : Résultat de l'affection p ← &x.

**Exemple :**

```
int *k, l ; /* k est un pointeur, l est un entier : attention à cette
manière de déclarer ! */
l = 5 ;
k = &l ;
printf(" %d\n", l) ;
*k = 2;
printf("%d\n", l);
```

Le code ci-dessus affichera successivement les valeurs 5 et 2.

## 13. Arithmétique des pointeurs

Lorsque l'on déclare un tableau T, on définit en fait un doublet constant (adresse, type). La notation T[n], où n est un entier, désigne alors l'objet se trouvant à l'adresse :

*adresse du doublet + n × taille du type de T.*

Si i est un pointeur (adresse, type) et n un entier, alors la notation i + n désigne le doublet :

(*adresse du doublet + n × taille du type, type)*

Les expressions *(i+n) et i[n] désignent le même objet.

**Exemple :**

Pour un tableau E de 50 étudiants, E[4] désigne l'objet de type étudiant qui se trouve à l'adresse E + 4. Supposons que le tableau est stocké en mémoire à l'adresse 10500 et que la taille d'un objet de type Etudiant soit de 160.

E = (10500, Etudiant)

E + 4 = (11140, Etudiant)

L'expression *(E + 4) désigne également l'étudiant se trouvant à l'adresse 11140. Les deux expressions *(E + 4) et E[4] sont équivalentes.

**Exemple :**

Soient les déclarations et les opérations suivantes :

```
Etudiant E[50], *i, *j ;
i = E ;
j = i + 4 ;
```

Les expressions suivantes désignent toutes le même objet :

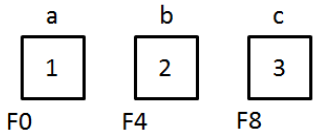
```
E[4], i[4], *j, *(E + 4) et *(i + 4)
```

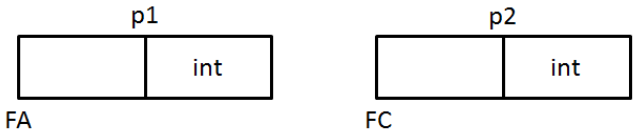
Les expressions suivantes désignent toutes un autre même objet :

```
E[5], (i+1)[4], *(i + 5), *(j + 1), j[1] et (j – 2)[3]
```

## Exercice 1 : Manipulation des pointeurs

Soient trois entiers a, b et c et deux pointeurs de type entier p1 et p2.




1. Déterminez quelle est la valeur des différents éléments donnés dans le tableau pour chaque opération.

| | a | b | c | &a | &b | &c | p1 | p2 | *p1 | *p2 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Initialisation** | | | | | | | | | | |
| **p1 = &a** | | | | | | | | | | |
| **p2 = &c** | | | | | | | | | | |
| ***p1 = *p2** | | | | | | | | | | |
| **(*p2)++** | | | | | | | | | | |
| **p1 = p2** | | | | | | | | | | |

| p2 = &b | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *p2 = *p1 – 2 * *p2 | | | | | | | | | | |
| (*p2)-- | | | | | | | | | | |
| *p1 = *p2 - c | | | | | | | | | | |
| a = (2 + *p2) * *p1 | | | | | | | | | | |
| | a | b | c | &a | &b | &c | p1 | p2 | *p1 | *p2 |
| p2 = &c | | | | | | | | | | |
| *p2 = *p1 / *p2 | | | | | | | | | | |
| *p1 = a + b | | | | | | | | | | |
| a += *p1 | | | | | | | | | | |
| b = *p1 + *p2 | | | | | | | | | | |
| *p1 = 2 * a | | | | | | | | | | |
| a = *p2 | | | | | | | | | | |
| *p2 = *p1 - *p2 | | | | | | | | | | |
| *p1 = 1 – c | | | | | | | | | | |
| *p2 += *p1 + a | | | | | | | | | | |
| p2 = p1 = &a | | | | | | | | | | |
| p2++ | | | | | | | | | | |
| p1 += 2 | | | | | | | | | | |
| c = p2 == &c | | | | | | | | | | |
| p1 = NULL | | | | | | | | | | |

2. Vérifiez que vos réponses sont correctes à partir du fichier *mainPointeur.c* et du débogueur.

## Exercice 2 : Passage de paramètres par adresse

En C, seul le passage de paramètres par valeur est disponible. Le passage de paramètres résultats (i.e. paramètres dont la valeur est modifiée par une fonction) n'est pas supporté. Le seul moyen dont on dispose pour approcher ce mode de passage est d'utiliser comme paramètre effectif une adresse.

```
void incr(int *a) {
  *a = (*a) + 1 ;
}
```

```
int main() {
 int x = 5;
 incr(&x);
 printf("%d", x); /* ce programme affiche 6 */
 return 0 ;
}
```

Observer l'exécution de ce programme avec le débogueur. Ainsi, on comprend mieux la fonction *scanf* qui reçoit les adresses des variables qui se voient affecter les valeurs saisies au clavier.

### Exercice 3 : Problème du drapeau hollandais

On veut trier un tableau dont les éléments ne prennent que dix valeurs (de 0 à 9) avec le critère suivant :

- tous les éléments dont la valeur est strictement inférieure à 3 sont au début du tableau ;
- tous les éléments dont la valeur est comprise entre 3 et 6 sont au milieu ;
- et tous les éléments dont la valeur est strictement supérieure à 6 sont à la fin.

Le programme affichera, par exemple :

Tableau initial :

| 5 | 8 | 1 | 4 | 3 | 9 | 2 | 7 | 3 | 8 | 1 | 4 | 5 | 3 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Tableau trié :

| 1 | 2 | 1 | 3 | 4 | 4 | 5 | 3 | 3 | 5 | 9 | 7 | 8 | 8 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Voici le pseudo-algorithme pour ce tri (source : Wikipédia):

```
Début
     b ← 1; w ← 1 ; r ← n
        tant que w ≤ r faire
                si T[w] = blanc alors w ← w+1
                sinon si T[w] = bleu alors
                        {échanger T[b] et T[w] ;
                        b ← b+1 ; w ← w+1 }
                sinon // T[w] = rouge
                        {échanger T[w] avec T[r] ;
                        r ← r-1}
        fintantque ;
Fin
```

Ecrire le programme qui permet de trier un tableau suivant l'algorithme proposé en utilisant les pointeurs.

# Chapitre 8 : Allocation dynamique de mémoire

Objectif :

- Maîtriser l'allocation dynamique de mémoire et son utilisation

## 1. Notions de base

Le détail des fonctions utilisées dans cette partie (*malloc*, *free*) est donné dans la section 6.11 du document « Introduction au langage C ».

### Exercice 1 : Liste chaînée avec allocation dynamique de mémoire

Dans cet exercice nous réalisons les fonctions de gestion d'une liste chaînée de manière similaire au TP6, mais en utilisant l'allocation dynamique de mémoire (au lieu d'un tableau). Nous considérons les déclarations de type suivantes :

```
typedef struct element element ;
struct element {
      int valeur ;                  /* valeur de l'élément */
      element* suivant ;            /* adresse du successeur */
};

typedef element* liste ;
```

La création d'une liste vide, revient à faire la déclaration :

```
liste L = NULL ;  /* L contient l'adresse du premier élément de la liste */
```

La création d'un nouvel élément se réalise de la manière suivante :

```
element *e = (element* ) malloc(sizeof(element)) ;
```

Ensuite, les champs de *e* peuvent être consultés ou modifiés à l'aide de l'opérateur -> :

```
e-> valeur = 12 ;
e-> suivant = NULL ;
```

Ou, de manière équivalente (à l'aide de l'opérateur de déréférencement *) :

```
(*e).valeur = 12 ;
(*e).suivant = NULL ;
```

Réaliser les fonctions C permettant de gérer une telle liste et tester les (au moins) dans les cas suivants : insertion et suppression et tête, en fin et en milieu de liste.

- *void afficherListe(liste *l)* : affiche tous les éléments de la liste dans l'ordre.

- *void insererElement(int x, liste *l)* : en supposant l triée, insère x dans l *en maintenant l triée*

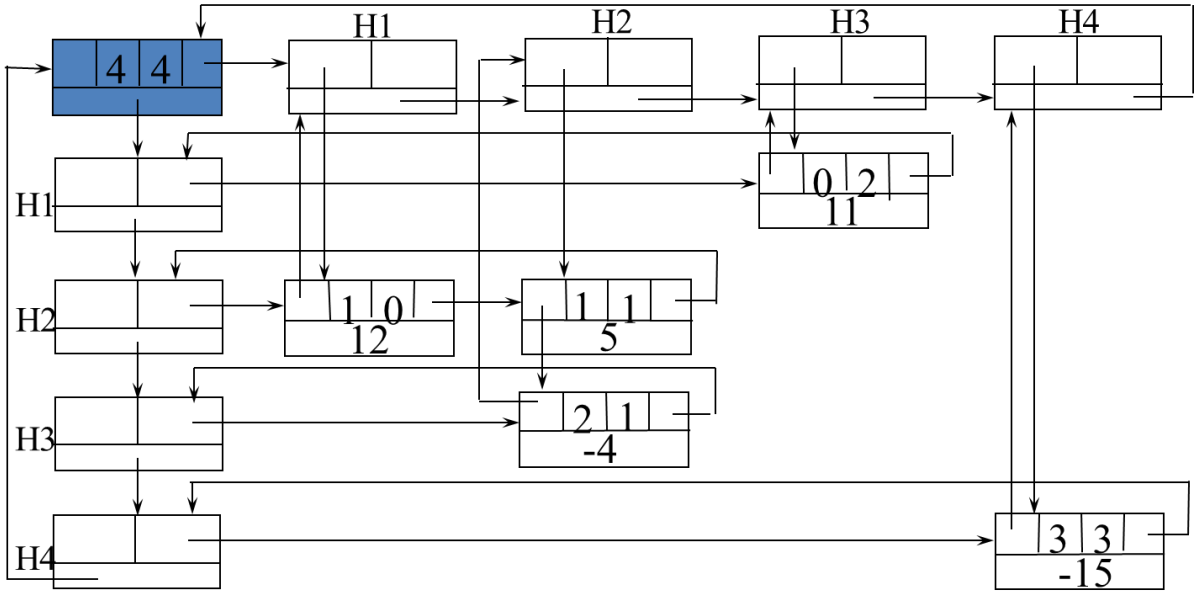
- *void supprimerElement(int i, liste *l)* : supprime le i-ème élément de l.

NB : le type du paramètre *l* des fonctions ci-dessus est de type *liste**. Ceci est nécessaire dans le cas d'insertion/suppression en tête de liste. *Pourquoi ?*

## Exercice 2 : Matrice creuse

Dans une *matrice creuse* N x M, la plupart des éléments sont nuls, si bien que la représenter en C par un tableau de N x M éléments a peu d'intérêt et termes d'occupation de la mémoire. Une solution d'implémentation à l'aide de plusieurs listes chaînées est illustrée sur l'exemple suivant :

$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 5 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$




1. En vous inspirant de cette figure, proposez les types nécessaires et une implémentation associée permettant de représenter des matrices creuses et de réaliser deux opérations :
   - Somme de deux matrices
   - Produit de deux matrices
2. En supposant que chaque matrice creuse a, en moyenne, 10% d'éléments non nuls, évaluer le nombre de multiplications effectuées avec cette nouvelle représentation lors d'un produit matriciel. Comparer avec le nombre de multiplications nécessaires pour cette même opération avec une représentation des matrices avec des tableaux à deux dimensions.