Structures de données
TD2 Rappels de C

A ne pas oublier :

- Allocation statique = allocation dans la pile, mémoire libérée automatiquement à la fin de la fonction
- Allocation dynamique = allocation dans le tas = malloc, calloc, realloc... , c'est à vous de libérer la mémoire avec free
- **1 malloc = 1 free !!!!!!**
- parcours d'une matrice : toujours dans le sens ligne puis colonne sinon mauvaises performances

1 Exercice 1 Tableau dynamique à 2 dimensions

Un tableau 2D (= une matrice) correspond à un tableau de pointeurs (les lignes) dont chaque case pointe vers un tableau de valeurs (les colonnes).

1.0.1 Méthode 1

```
int** allouer_matrice(int n, int m)
{
    // Tableau 2d donc int** (tableau 3d int*** etc)
    // Attention au sizeof, ici nos cases seront des pointeurs (sur les colonnes)
    int** mat = (int**)malloc(n * sizeof(int*));
    if(mat == NULL) { // On verifie qu'il n'y a pas d'erreur
        return NULL; // NULL == adresse 0
    }
    for(int i=0;i<n;i++) {
        // Pour chaque ligne on alloue un tableau correspondant aux colonnes
        // sizeof(int) car dans ce tableau on stocke les valeurs
        mat[i] = (int*)malloc(m * sizeof(int));

        // On verifie que l'allocation s'est bien passee
        if(mat[i] == NULL) {
            // Si c'est pas le cas, il faut rendre la memoire de toutes les lignes
            // pointant vers un tableau de valeurs (colonnes) !!!
            for(int k=0;k<i;k++) {
                free(mat[k]);  1
            }
            free(mat); // et ne pas oublier le tableau de pointeurs (de lignes) !
            return NULL;
        }
    }
    return mat;
}
```

1.1 Méthode 2

On utilise le type int*** car ce qu'on attend en paramètre c'est la référence d'une variable pointant sur un tableau 2D.

```
int allouer_matrice(int*** matrice, int n, int m)
{
    // code identique en remplacant return NULL; par return 0;

    // et remplacer return mat; par :
    *matrice = mat;
    return 1;
}
```

Cette seconde méthode permet de retourner un code d'erreur (ou de succès) ou bien une autre information. Elle est souvent employée pour les fonctions de la bibiliothèque standard.

C'est par exemple le cas des fonctions de lecture/écriture qui retournent le nombre d'octets/caractères qui ont été lus ou écrits.

2 Exercice 2

2.1 Matrice triangulaire statique

Il n'est pas possible d'allouer une matrice triangulaire statiquement. Vous pouvez allouer une matrice rectangulaire et l'utiliser comme une matrirce triangulaire mais vous allez gâcher de la mémoire.

2.2 Représenter une matrice triangulaire avec une structure

```
typedef struct {
    int matrice**;
    int taille;
    int orientation; // 0 = low, 1 = sup
} MatTriangulaire
```

2.2.1 Allocation

```
MatTriangulaire* allouer_MatTriangulaire(int n, int o)
{
    // On alloue une structure MatTriangulaire
    MatTriangulaire* mat = (MatTriangulaire*)malloc(n * sizeof(MatTriangulaire));
    if(mat == NULL) {
        return NULL;
    }
```

```
    mat->taille = n;
    mat->orientation = o;

    mat->matrice = (int**)malloc(n * sizeof(int*)); // attention sizeof
    if(mat->matrice == NULL) {
        free(mat);
        return NULL;
    }
    if(o == 1) {
        for(int i=0;i<n;i++) {
            mat->matrice[i] = (int*)malloc( (n-i) * sizeof(int));
            // + Traitement des erreurs
        }
    } else {
        for(int i=0;i<n;i++) {
            mat->matrice[i] = (int*)malloc( (i+1) * sizeof(int));
            // + Traitement des erreurs
        }
    }
    return mat;
}
```

On a fait 1 + 1 + n mallocs.

2.2.2 Désallocation

```
void desallouer_MatTriangulaire(MatTriangulaire* mat)
{
    // n free
    for(int i=0;i<mat->taille;i++) {
        free(mat->matrice[i]);
    }
    // 1 free
    free(mat->matrice);
    // 1 free
    free(mat);
    // n + 1 + 1 free -> on a tout libere
}
```

2.2.3 Affichage

```
void afficher_MatTriangulaire(MatTriangulaire* mat)
{
    if(mat->orientation == 1) { // sup
        for(int i=0;i<mat->taille;i++) { // pour chaque ligne
            for(int j=0;j<mat->taille;j++) { // pour chaque colonne de la ligne
                if(j < i) {
                    printf("0\t");
                } else {
                    printf("%d\t", mat->matrice[i][j-i]);
                }
            }
            printf("\\n");
        }
    } else { // matrice inf
        for(int i=0;i<mat->taille;i++) {
            for(int j=0;j<mat->taille;j++) {
                if(j > i) {
                    printf("0\t");
                } else {
                    printf("%d\t", mat->matrice[i][j]);
                }
            }
            printf("\n");
        }
    }
}
```

3 Lecture et écriture dans un fichier

Rappel accès aux champs d'une structure Soit s une variable de type...

— ... structure : on accède à ses champs avec un point
— ... pointeur de structure : on accède à ses champs avec une flèche

3.1 Ecriture d'un répertoire

```
void ecrire(char* fname, Repertoire* rep)
{
    FILE* file = fopen(fname, "w"); // "a" pour ajouter en fin de fichier
    if(file == NULL) {
        printf("Impossible d'ecrire le repertoire dans \"%s\"\n", fname);
    } else {
        for(int i=0;i<rep->taille;i++) {
            Personne p = rep->tab[i]; // == (rep->tab)[i];
            fprintf(file, "%s:0%ld\n", p.nom, p.tel);
        }
        fclose(file);
    }
}
```

3.2 Lecture d'un répertoire

```
Repertoire* lecture(char* fname)
{
    FILE* file = fopen(fname, "r");
    if(file == NULL) {
        printf("Impossible de lire le repertoire dans \"%s\"\n", fname);
        return NULL;
    } else {
        Repertoire* r = (Repertoire*)malloc(sizeof(Repertoire));
        r->tab = (Personne*)malloc(sizeof(Personne)*TAILLE);
        r->taille = TAILLE;
        char buff_ligne[200];
        char buff_nom[136];
        long buff_tel;
        int nbLignes = 0;
        while(fgets(buff_ligne,200,file)!=NULL) {
            if(nbLignes>=r->taille) { // Si notre tableau est plein on agrandit
                r->taille += TAILLE;
                r->tab = (Personne*)realloc(r->tab, sizeof(Personne) * r->taille);
            }
```

```
            // Pour chaque ligne du fichier on s'attend a lire 2 donnees
            if(sscanf(buff_ligne,"%[^:]:%ld\n", buff_nom, &buff_tel) == 2){
                Personne* p = &r->tab[nbLignes++];
                p->nom = strdup(buff_nom); // strdup necessite un free a la fin
                p->tel = buff_tel;
            } else {
                printf("Erreur de lecture a la ligne %d.", nbLignes);
            }
        }
        // On adapte la taille du tableau si trop grand
        r->tab = (Personne*)realloc(r->tab, sizeof(Personne) * nbLignes);
        r->taille = nbLignes;
        fclose(file);

        return r;
    }
}
```

Concernant la taille du répertoire (et donc la taille du tableau de personnes) on a trois solutions :

- Lire au fur et à mesure et faire des appels à realloc pour adapter la taille du tableau (comme ci-dessus) ⇒ les appels à realloc() vont ralentir le programme et peuvent échouer
- Parcours tout le fichier ⇒ peu performant si le répertoire est grand
- Utiliser la première ligne du fichier pour stocker la taille du répertoire : on stocke la valeur de la première ligne dans r→taille puis on crée r→tab de cette taille.

4 Exercice 4

Exemple de makefile :

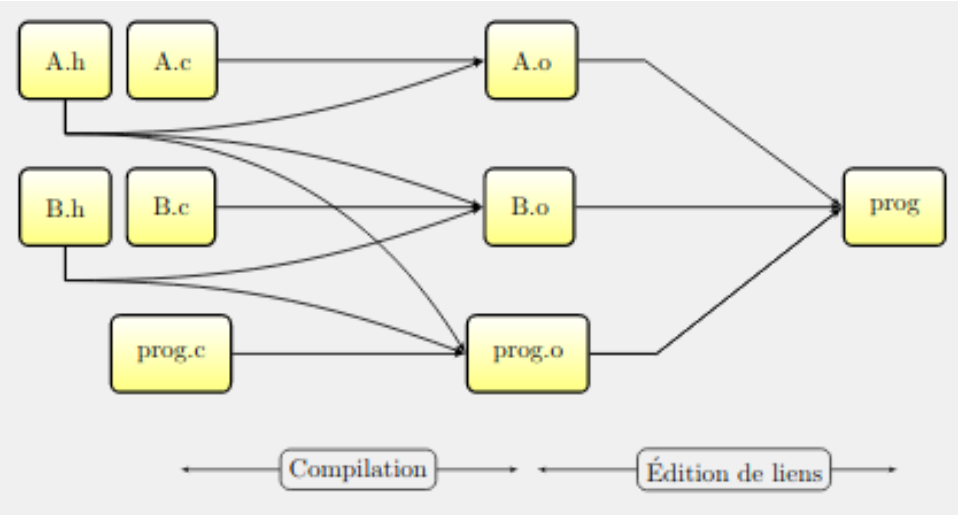
```
all: prog

prog: A.o B.o prog.o
    gcc A.o B.o prog.o -o prog
A.o: A.c A.h
    gcc -c A.c -o A.o
B.o: B.c B.h
    gcc -c B.c -o B.o
prog.o: prog.c A.h B.h
    gcc -c prog.c -o prog.o
```

```
clean:
    rm -f *.o prog
```

Ce makefile comporte 6 règles (all, prog, A.o, B.o, prog.o, clean).

Si vous faites *make all* vous allez appelez la règle *all*. Cette règle a une dépendance, la règle *prog* qui est donc appelée. Cette règle appelle à son tour ses dépendances puis fait appel à gcc pour compiler le binaire prog.



5 Erreur pendant le TD

Pendant le tp j'ai dit que si vous passiez un tableau statique en paramètre d'une fonction, celui-ci était dupliqué en mémoire. **C'est faux !** Passer un tableau statique en paramètre c'est comme si vous passiez son adresse. C'est donc un passage par référence et non par copie.