TP 3 : Compilation séparée, Allocation dynamique de mémoire, types structurés

Dans ce TP vous allez tout d'abord apprendre à compiler séparément des programmes et utiliser l'outil `make`. Ensuite vous verrez comment utiliser l'API `malloc/free` pour réserver/libérer de l'espace pendant l'exécution et y stocker par exemple des tableaux dont la taille n'est pas connue à l'avance. Vous implémenterez même un tableau dynamique, c'est à dire un tableau dont la taille augmente au fur et à mesure qu'on y ajoute des valeurs.
Dans la partie 2, vous utiliserez les mots-clés `typedef` et `struct` pour définir de nouveaux types de variables, ce qui permet d'implémenter toutes sortes de structures de données. Dans la partie 4 on s'intéressera de plus près à une structure de données qui fait un usage intensif de l'allocation dynamique et que nous avons déjà vu en cours, la liste chaînée.

1 Compilation séparée

Jusqu'ici vous avez construit des programmes qui fonctionnent seuls : ils contiennent la fonction `main()` ainsi que toutes les fonctions intermédiaires dont vous avez besoin. Le langage C permet la compilation séparée. Le but est de répartir les fonctions dans des modules contenant des bibliothèques de fonctions qui pourront être définies indépendamment du programme principale. Cela représente plusieurs intérêts. D'une part on peut ainsi partager du code plus facilement (en évitant la méthode sauvage de copier/coller le texte des programmes d'un programme à l'autre). D'autre part la modification d'une bibliothèque ne nécessite pas de tout recompiler. Il suffit de recompiler la partie qui a été modifiée. Dans les programmes "à taille réelle" c'est très important car ces derniers peuvent faire des centaines de millier de ligne de codes et être réparties dans des dizaines de fichiers différents.

Attention ! l'outil `make` n'est pas intégré au compilateur C. Il s'agit d'un outil à part entière que vous devrez peut être installer avant de pouvoir commencer. Suivant le système d'exploitation que vous utilisez les procédures peuvent différer. Sous Linux cela se fera avec une ligne de commande comme :

```
mon-ordinateur% sudo apt-get update
mon-ordinateur% sudo apt-get install make
```

QUESTION 1 ► Écrire un programme réparti en deux fichiers `main.c` et `biblio.c`. Dans `biblio.c` intégrez des fonctions permettant de rendre la somme, le produit et la moyenne d'un tableau d'entier. Déclarez un fichier `biblio.h` contenant le profil de ces fonctions. Utilisez ces fonctions dans `main.c` en incluant une phase de lecture d'un tableau d'une taille définie par l'utilisateur et inférieure à une valeur maximum définie par une constante.
Une fois ceci réalisé compilez les programmes séparément avec l'option de compilation `-c`, puis effectuez l'édition de lien pour obtenir l'éxécutable par :

```
gcc -c biblio.c
gcc -c main.c
gcc -o main main.o biblio.o
```

tester votre programme. Puis modifiez `biblio.c` et testez cette modification en faisant tout le nécessaire pour bien recompiler le projet (normalement il suffit de recompiler `biblio.c` puis de refaire l'édition de liens). Pour simplifier cette dernière tâche de compilation on peut utiliser un `Makefile` qui contient les dépendances et les actions à réaliser. Pour cette exemple le `Makefile` correspondant est :

```
main: main.o biblio.o
        gcc -o main main.o biblio.o

main.o: main.c
        gcc -c main.c

biblio.o: biblio.c
        gcc -c biblio.c
```

Pour produire une étiquette il suffit d'entrer la commande `make etiquette`, par exemple `make main` pour obtenir l'éxécutable.
Il est très important d'utiliser des tabulations pour les commandes et non des espaces dans les `Makefile` (faites attention parfois certains éditeurs de texte changent les tabulations en espaces sans le dire...).

- Modifiez `biblio.c` (en ajoutant un espace par exemple) puis relancez `make`.
- Modifiez `main.c` (en ajoutant un espace) puis relancez `make`.

Vous pourrez trouver plus d'explication sur la commande `make` et la composition des `Makefile` ici :

`https://gl.developpez.com/tutoriel/outil/makefile/`

2 Allocation de mémoire

«Les gens qui ont bonne conscience ont souvent mauvaise mémoire.»
– Jacques Brel

Échauffement : allocation statique

QUESTION 2 ► Écrire un programme qui lit successivement dans une bouble `for` dix nombres entiers (sur l'entrée standard) et les stocke dans un tableau alloué *statiquement*. Autrement dit, votre tableau doit être déclaré de la façon suivante : `int montableau[10]`.
Pour lire un nombre et ranger sa valeur en mémoire (par exemple dans une variable `x`) vous utiliserez la primitive `scanf()` en lui passant l'adresse mémoire souhaitée, par exemple `scanf("%d", &x)`.

QUESTION 3 ► Modifiez votre programme pour que, une fois la lecture terminée, il calcule le minimum, le maximum, et la moyenne des valeurs contenues dans le tableau, comme illustré ci-dessous.

Remarque : pour tester, vous pouvez exécuter votre programme de façon interactive, et taper les nombres un par un au clavier. Ou alors, vous pouvez brancher l'entrée de votre programme sur la sortie d'une autre commande ! Comme la fonction `scanf` ignore les espaces et les retours à la ligne, elle ne verra aucune différence et consommera toujours les nombres un par un, exactement de la même manière.

```
% echo 1 2 3 4 5 6 7 8 9 10 | ./monprogramme
min=1 max=10 moy=5
```

ou encore :

```
echo -1000 1 1 1 1 1 1 1 1 1| ./monprogramme
min=-1000 max=1 moy=-99
```

Allocation dynamique avec `malloc()`

Dans cet exercice, on va modifier notre programme pour qu'il puisse traiter une quantité arbitraire de valeurs. On ne peut plus déclarer notre tableau comme une variable statique, puisque sa taille ne sera déterminée qu'au moment de l'exécution. Il faudra donc déclarer notre variable avec un type *pointeur* (en écrivant quelque chose comme `int * montableau`) et l'initialiser avec `malloc()`. Pour simplifier l'exercice, on suppose ici que la première valeur lue (toujours sur l'entrée standard) nous indique la quantité *N* de nombres à traiter.

Remarque : La signature de malloc est `void* malloc(size_t size)` où le paramètre size indique le *nombre d'octets* qu'on souhaite réserver. Il ne faudra donc pas utiliser la valeur de *N* directement, mais d'abord la «traduire» en une taille exprimée en nombre d'octets. Pour cela, vous utiliserez l'opérateur `sizeof()` qui prend comme paramètre un nom de type, et qui renvoie la taille (en octets) occupée par une donnée de ce type.

QUESTION 4 ► Modifiez votre programme pour qu'il alloue dynamiquement le tableau. On veut obtenir par exemple les comportements suivants :

```
% echo 1 2 3 4 5 6 7 8 9 10 | ./monprogramme
N=1 min=2 max=2 moy=2
```

ou encore :

```
% echo 4 100 -33 -33 -33 | ./monprogramme
N=4 min=-33 max=100 moy=0
```

Implémentation d'un tableau dynamique

Dans l'exercice précédent, la taille *N* du tableau nous était indiquée explicitement au début de l'exécution. Ici, on suppose au contraire que tous les nombres lus sur l'entrée standard sont des valeurs à traiter. Vous devrez donc remplacer votre boucle `for` par une boucle `while`, qui se répètera autant de fois que nécessaire, comme illustré ci-dessous :

```
while(1)
{
    int number;
    int nbtokens = scanf("%d", &number);

    if(nbtokens != 1 || feof(stdin))
       break;

    ... do something with ‘number‘ ...
 }
```

Remarque : le test sur `nbtokens` vérifie que `scanf()` a bien reconnu un nombre. Si la saisie est incorrecte (par ex. si elle contient des lettres) alors on s'arrête. De même, le test sur `feof(stdin)` vérifie que le flux d'entrée est encore «ouvert», sinon on s'arrête. Lorsqu'on utilise un tuyau (ce qui est le cas avec la commande «`echo truc | ./monprogramme`») ce flux d'entrée est fermé automatiquement, après la fin de `echo`. Lorsqu'on exécute notre programme de façon interactive, on peut fermer l'entrée standard avec la combinaison de trouches Ctrl+D.

QUESTION 5 ► Votre travail consiste à modifier votre programme pour traiter une quantité «inconnue» de nombres. En particulier, on veut maintenant pouvoir augmenter la valeur de *N* lorsque le tableau est plein. Vous allez donc devoir garder un oeil sur le nombre *K* de cases occupées. Lorsque *K* atteint *N*, c'est que le tableau est plein, il faut donc en allouer un nouveau (par exemple avec $2 \times N$ cases) et recopier dans ce nouveau tableau les *K* premiers nombres. Il ne vous reste maintenant qu'à *libérer* le précédent tableau, avec la primitive `free()`, puis à mettre à jour votre pointeur.
On veut obtenir par exemple les comportements suivants :

```
% echo 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | ./monprogramme
N=20 min=1 max=20 moy=10
```

ou encore :

```
% echo 100 -33 -33 -33 | ./monprogramme
N=4 min=-33 max=100 moy=0
```

Minute culturelle : Le conteneur que vous avez implémenté dans cet exercice est très courant dans la vraie vie, c'est par exemple le `Vector` de C++, ou la classe `ArrayList` en Java. Le terme générique pour cette structure de donnéees est «*tableau dynamique*», par opposition aux tableaux «normaux» dont la taille est fixe. On parle aussi de «*liste contiguë*», par opposition aux listes dites «chaînées» (cf partie **??** p. **??**) dont les éléments ne sont pas voisins en mémoire.
Pour les curieux, allez donc lire `https://en.wikipedia.org/wiki/Dynamic_array`

3 Types structurés

«Qu'importe le flacon, pourvu qu'on ait l'ivresse.»
– Musset

Vous avez manipulé jusqu'ici uniquement des variables numériques (nombres entiers, nombres à virgule) ainsi que des tableaux de nombres. Pour permettre une représentation plus riche de nos données, le langage nous permet également de déclarer de nouveaux types dits «agrégés» (AKA structures, ou enregistrements) en regroupant dans un même objet plusieurs «champs» (AKA attributs, ou membres). Chacun de ces champs peut lui-même avoir n'importe quel type, y compris un type structure, comme illustré ci-dessous :

```
struct Point
{
   int x;
   int y;
};

struct Point p;
p.x = 3;
p.y = 5;

struct Point *q=malloc(...);
q->x=8;
q->y=10;
```

```
struct Rectangle
{
   struct Point p1;
   struct Point p2;
};

struct Rectangle r;
r.p1.x=2;
r.p1.y=1;

// l'affectation entre struct
// copie *tous* les champs
r.p2 = p;
```

Remarques :

- Le «vrai» nom du type qu'on vient de créer est «`struct Point`», c'est donc avec ce type qu'est déclarée la variable `p`, et c'est ce type qui est donné en argument de `sizeof()` pour le malloc.
- Cependant, le langage nous permet de définir des *synonymes* pour les noms de type, avec la syntaxe «`typedef anciennom nouveaunom`».
- Il est donc assez fréquent de rencontrer des déclarations comme «`typedef struct Point Point`», ce qui permet par la suite d'utiliser indifféremment les deux noms «`struct Point`» et «`Point`».
- Par ailleurs, il se trouve que le «nom de structure» (situé entre le mot-clé struct et l'accolade ouvrante) est facultatif dans la déclaration ! On aurait ainsi pu choisir d'écrire notre déclaration de type d'un seul tenant, ce qui donne `typedef struct {int x; int y;} Point;`
- Je vous recommande de privilégier autant que possible cette dernière forme, c'est à dire de toujours utiliser *ensemble* les mots-clés `typedef` et `struct` lorsque vous déclarez de nouveaux types.

Nombres Complexes

QUESTION 6 ► Définir un type structutre `Complexe` avec deux champs de type `float` pour les parties réelles et imaginaires. Écrire un programme qui lit deux nombres à virgule (i.e. `scanf()` avec `"%f"`) pour en faire une seule variable de type `Complexe`. Écrire ensuite une fonction `void affiche(Complexe z)` qui affiche `z` sous la forme $a + b.i$ si $b > 0$, ou $a - b.i$ si $b < 0$.

QUESTION 7 ► Écrire une fonction `Complexe ajoute(Complexe z1, Complexe z2)` qui renvoie la somme de ses deux arguments. De même, écrire une fonction `Complexe multiplie(Complexe z1, Complexe z2)` qui renvoie le produit de z1 et z2.
Utilisez ces fonctions pour afficher par exemple les valeurs de i^2, i^3, ou encore $(1 + i)(i - 1)$.

Répertoire téléphonique

Dans cet exercice, vous allez écrire un programme qui manipule un type `Personne` avec deux champs : un nom, et un numéro de téléphone. Pour le numéro, vous utiliserez un tableau de dix `char` (on pourrait aussi le faire avec un unique `int`, mais cet exercice porte sur les tableaux et les pointeurs). Pour le nom c'est un peu plus compliqué : utiliser un type tableau nous forcerait à décider de sa taille une fois pour toutes. Or, un tableau trop petit (par exemple `char nom[20]`) ne nous permettrait pas de stocker les noms trop longs. Au contraire, fixer une taille trop importante (par exemple `char nom[1000]`) pose le risque

d'une consommation excessive d'espace. La bonne approche consiste donc à utiliser un pointeur (i.e. `char * nom`) et à allouer chaque chaîne dynamiquement à la bonne longueur.

QUESTION 8 ► Implémenter le type `Personne`. Écrire un programme qui lit successivement dix paires nom+numéro et les stocke dans un tableau alloué statiquement, par exemple `Personne repertoire[10]`. Dans une seconde boucle, affichez le contenu du tableau.

Remarque : Pour lire sur l'entrée standard, vous utiliserez `scanf("%s")`. Attention : de manière générale, `scanf()` ignore tous les «caractères blancs» : espace, retour charriot, etc. Avec `"%s"`, on demande à scanf de chercher un «mot», c'est à dire une séquence de caractères non-blancs, et rien de plus. Si l'utilisateur tape deux mots, scanf n'en «consommera» qu'un seul. Le second «restera dans le tuyau» et vous aurez des surprises dans la suite de l'exécution. Ce genre de problème est très courant dans la vraie vie, et constitue une difficulté majeure pour le développement logiciel (pour les curieux : faites donc une requête google avec «input sanitization»). Heureusement, le TP d'aujourd'hui ne porte pas du tout sur ce sujet. Pour simplifier l'exercice, on va supposer aujourd'hui que toutes nos entrées sont formattées correctement : pas d'espace à l'intérieur d'un nom, pas d'espace à l'intérieur d'un numéro.

4 Définition récursive de type

«Pour comprendre le principe de récursivité, il faut d'abord comprendre le principe de récursivité»
– Anonyme

On dit qu'un type T est *récursif* si la définition de T fait référence à T. Par exemple dans un arbre binaire, chaque *noeud* comporte deux références à d'autres noeuds. De même pour une liste chaînée : chaque élément contient un lien vers l'élément suivant. Lorsqu'on programme en C, ces types sont des `struct` et les liens sont des pointeurs de struct. Parmi les nombreuses façons d'écrire ce genre de déclarations, je vous recommande pour le TP d'aujourd'hui la syntaxe illustrée ci-dessous :

```
typedef struct _elem
{
    int value;
    struct _elem* next;
} elem;

// initialisation: liste vide
elem* list = NULL;

// creation d'un nouveau maillon
elem* e = malloc(...);
e->value=42;

// insertion en tete de liste
e->next=list;
list=e;
```

```
typedef struct _treenode
{
    int value;
    struct _treenode* left;
    struct _treenode* right;
} treenode;

// un arbre vide
treenode* root=NULL;

// creation d'un noeud racine
treenode* node=malloc(...);
node->value=42;
node->left=NULL;
node->right=NULL;

root=node;
```

Remarques :

— La coutume est d'utiliser un nom de structure «jetable» (par ex. commençant par un *underscore*) pour éviter de polluer inutilement l'espace des noms de type. Le nom «`struct _elem`» n'est utile qu'à l'intérieur du typedef proprement dit. Par la suite, on déclarera simplement des variables de type `elem`.

— On parle ici d'un type «élément de liste» et pas d'un type «liste». Chaque élément contient une valeur, et indique l'adresse de son successeur dans la liste. Une liste vide, c'est à dire qui ne comporte aucun élément, sera ainsi représentée par un simple pointeur nul. Une liste non-vide sera représentée par un pointeur vers son premier élément. Déclarer un autre type pour la liste elle-même ne présente pas d'intérêt, puisqu'il serait identique au type «`elem*`».

Liste chaînée de nombres entiers

Dans cet exercice, vous allez travailler avec la liste chaînée définie ci-dessus. Commencez donc par retaper le typedef pour elem, et également le squelette de programme principal qui vous est donné dans l'encadré ci-dessous. Dans les questions à venir, votre travail consistera à écrire les fonctions `afficher()`, `ajouter()`, etc. Toutes ces fonctions travailleront sur la variable globale `list`.

```
elem* list = NULL;

int main(void)
{
    while(1)
    {
        printf("menu:\n");
        printf("\t1: ajouter en tete\n");
        printf("\t2: afficher la liste\n");
        printf("\t0: quitter\n");

        int choix;
        scanf("%d", &choix);
        switch(choix)
        {
        case 0:
            goto fin;
        case 1:
            ajouter_en_tete();
            break;
        case 2:
            afficher();
            break;
        default:
            printf("choix incorrect\n");
            continue ; // revenir au menu
        }
    }

fin:
    printf("au revoir\n");

    return 0;
}
```

QUESTION 9 ► Implémenter les fonctions `void afficher(void)` et `void ajouter_en_tete(void)`. La première parcourt la liste et affiche la valeur de chaque élément rencontré. La seconde lit (sur l'entrée standard) un nombre tapé au clavier, crée un nouveau maillon avec cette valeur, puis insère le maillon au début de la liste.

On veut obtenir environ le comportement ci-dessous. Les lignes en gris représentent les entrées au clavier.

```
menu:
        1: ajouter en tete
        2: afficher la liste
        0: quitter
1
tapez un nombre:
123
menu:
        1: ajouter en tete
        2: afficher la liste
        0: quitter
1
tapez un nombre:
42
menu:
        1: ajouter en tete
        2: afficher la liste
        0: quitter
```

```
2
affichage de la liste:
42
123
fin de la liste
menu:
        1: ajouter en tete
        2: afficher la liste
        0: quitter
0
au revoir
```

QUESTION 10 ► Écrire une fonction `void rechercher(void)` qui parcourt la liste jusqu'à trouver un certain nombre (tapé au clavier) et affiche sa position, ou qui répond «pas trouvé» lorsque le nombre demandé n'est pas dans la liste. Ajouter une option correspondante dans le menu principal, et testez plusieurs scénarios pertinents : nombre présent plusieurs fois, nombre absent, etc.

QUESTION 11 ► Écrire une fonction `void ajouter_en_queue(void)` qui lit un nombre au clavier, crée un nouveau maillon, puis l'insère en *dernière* position dans la liste. Ajoutez un choix dans le menu principal, et testez plusieurs scénarios pertinents : insertion dans une liste vide, insertions multiples à la suite, etc.

QUESTION 12 ► Écrire une fonction `void supprimer(void)` qui lit un nombre au clavier, puis supprime *toutes* les occurences de ce nombre dans la liste en désallouant les maillons correspondants. Ajoutez un choix au menu principal, et testez plusieurs scénarios pertinents : nombre en première position, en dernière position, nombre absent de la liste, nombre présent plusieurs fois à la suite, etc.

QUESTION 13 ► Écrire une fonction `void dupliquer(void)` qui lit un nombre au clavier, puis qui parcourt la liste en «dédoublant» chaque occurence de cette valeur. Par exemple, dupliquer le nombre 2 dans la liste (1, 2, 3, 2, 2, 5) doit produire la liste (1, 2, 2, 3, 2, 2, 2, 2, 5).
Ajoutez un choix dans le menu principal, et testez plusieurs scénarios pertinents : nombre en première position, en dernière position, nombre absent de la liste, nombre présent plusieurs fois à la suite, etc.

QUESTION 14 ► Écrire et ajouter au menu une fonction `void est_triee(void)` qui affiche «oui» ou «non» selon que les éléments de la liste forment, ou pas, une suite croissante.

QUESTION 15 ► Écrire et ajouter au menu une fonction `void ajouter_en_place(void)` qui lit un nombre au clavier, crée un nouveau maillon, puis l'insère à la «bonne» position dans une liste triée. Testez sur plusieurs scénarios pertinents.
Remarque : on suppose que l'utilisateur n'a pas le droit d'appeler cette fonction si la liste n'est pas triée.

QUESTION 16 ► Écrire et ajouter au menu une fonction `void retourner(void)` qui inverse l'ordre des éléments de la liste. Par exemple la suite (1, 2, 3, 4) devient (4, 3, 2, 1). Cette fonction ne doit pas créer de nouveau maillon ni en supprimer. Testez sur plusieurs scénarios pertinents : liste vide, un seul élément, etc.