Le TP a pour but de continuer la prise en main du langage C et en particulier de commencer à prendre en main les pointeurs. Les premiers exercices sont faits pour être faciles, il est indispensable que vous ayez fait les exercices 1 à 5. Les exercices 6 et 7 vous permettront d'approfondir votre compréhension du langage. Les exercices 8 et suivant sont là pour vous présenter une mise en pratique de tous les outils vus dans ce TP et le TP précédent afin de réaliser un démineur.

Exercice 1 : Prise en main des fonctions

Dans cet exercice, nous allons écrire nos premières fonctions en C. Je vous encourage à regarder l'exemple "fonctions" du Moodle qui vous éclairera sur la syntaxe des fonctions.

Q1. Écrivez une fonction `void affiche_entier (int n);` qui affiche la valeur de l'entier `n` précédé de "`La fonction vous affiche l'entier :` ". Testez votre fonction en l'appelant plusieurs fois depuis votre fonction `main`.

Dans la description de la fonction, `void` signifie que votre fonction ne retourne aucune valeur, ici elle ne fait qu'afficher à l'écran. C'est le seul cas pour lequel vous n'avez pas besoin de terminer votre fonction par un `return`.

Q2. Écrivez une fonction `int produit (int a, int b);` qui retourne le produit des deux entiers donnés en paramètre. Testez votre fonction en l'appelant plusieurs fois depuis votre fonction `main`. Attention, la fonction `produit` retourne une valeur mais ne doit rien afficher elle-même !

Q3. Écrivez une fonction `void modifie_a (int a);` qui prend en paramètre un entier `a` et lui réaffecte une autre valeur au sein de la fonction (par exemple : `a = a + 1`). Affichez la nouvelle valeur de `a` grâce à un `printf` présent à la fin de la fonction `modifie_a` puis par un `printf` présent dans la fonction `main` après l'appel à la fonction `modifie_a`. Que constatez-vous ? Comme l'expliquez-vous ?

Exercice 2 : Prise en main des pointeurs

Cet exercice a pour but de prendre en main la notion de pointeurs. Je vous encourage à regarder l'exemple "pointeurs" du Moodle qui vous éclairera sur la syntaxe des pointeurs.

Q1. Dans une fonction `void test_pointeur (void);`, déclarez et initialisez un entier `a` puis déclarez et initialisez un pointeur `pt_a` pointant vers `a`. Affichez ensuite la valeur de `a` ainsi que son adresse. Affichez pour finir la valeur de la variable `pt_a` ainsi que le contenu de la case mémoire vers laquelle pointe `pt_a`.

Testez votre fonction en l'appelant depuis votre fonction `main`.

Q2. Écrire une fonction `int plus_egal (int* pt_a, int b);` qui prend la valeur qui se trouve à l'adresse pointée par `pt_a`, additionne `b` à cette valeur et place le résultat à l'adresse pointée par `pt_a`. La fonction retournera la valeur `0` si elle s'est exécutée correctement jusqu'au bout.

Testez votre fonction en l'appelant depuis votre fonction `main` et en observant la valeur de `pt_a` avant et après l'appel de la fonction `plus_egal`. De même pour la valeur de la case mémoire ciblée par `pt_a`. Attention : Pour faire le test dans la fonction `main` il vous faut créer une variable `a` puis définir un pointeur `pt_a` qui pointe vers `a`.

Q3. De la même manière que précédemment, écrivez une fonction `int fois_egal (int* pt_a, int b);` qui prend la valeur qui se trouve à l'adresse pointée par `pt_a`, la multiple `b` et place le résultat à l'adresse pointée par `pt_a`. La fonction retournera la valeur `0` si elle s'est exécutée correctement jusqu'au bout.

Exercice 3 : Fonctions calculant plusieurs résultats

En manipulant les fonctions, vous vous rendez compte que vous ne pouvez retourner qu'une seule valeur à la fois, un `int`, un `float` ou un `char`. Une technique pour contourner ce problème consiste à utiliser les pointeurs. On va donner à la fonction un pointeur vers une case mémoire et faire en sorte que la fonction remplisse la case mémoire ciblée. En procédant ainsi, la fonction appelante pourra retrouver le résultat mis dans cette case mémoire et l'utiliser.

Q1. Écrivez une fonction `void somme_et_produit (int a, int b, int* pt_somme, int* pt_produit);` qui prend en paramètre deux entiers et deux pointeurs vers des entiers et qui place dans la case mémoire ciblée par `pt_somme` la somme de `a` et `b` et dans la case mémoire ciblée par `pt_produit` leur produit.

Testez votre fonction en l'appelant depuis votre fonction `main` en affichant la valeur des pointeurs `pt_somme` et `pt_produit` avant et après l'appel à la fonction `somme_et_produit` ainsi que la valeur des cases mémoires qu'ils ciblent.

Exercice 4 : Les tableaux

Dans le TP précédent, vous avez écrit un programme qui permet d'afficher un tableau. Dans cet exercice nous allons utiliser deux manières différentes d'écrire une fonction permettant cet affichage.

Q1. Écrivez une fonction `void affiche_tableau (char* tableau, int taille);` qui prend en paramètre un tableau de caractères et sa taille et affiche le tableau en question. L'affichage devra être fait sur une seule ligne en séparant chaque caractère lu par un espace. Vous référerez à chaque case du tableau avec la notation de tableau `tableau[i]`.

Exemple : le tableau `char tableau[6] = {'a','z','e','r','t','y'};` sera affiché de la manière suivante : `a z e r t y .`

Q2. Écrivez une fonction `void affiche_tableau_pointeur (char* tableau, int taille);` qui a le même comportement que, `affiche_tableau` mais qui utilise la notation de pointeur pour accéder à chaque case du tableau : À savoir `tableau+i` est l'adresse de la ième case du tableau `tableau`.

Testez vos fonctions d'affichage et vérifiez qu'elles ont bien le même comportement.

Q3. Écrivez une fonction `void change_lettre (char* pt_lettre, char nouvelle_lettre);` qui prend en paramètre un pointeur vers une lettre et une lettre et place la nouvelle lettre à l'adresse ciblée par le pointeur.

Testez votre fonction en l'appelant depuis votre fonction `main` afin de modifier la première lettre de votre tableau de caractères. Faites de même pour modifier la troisième lettre de ce tableau.

Exercice 5 : Les chaînes de caractères

Comme présenté pendant le cours, les caractères en C sont stockés en mémoire comme des entiers suivant la convention donnée par la table ASCII. Ainsi le caractère `'a'` est stocké comme la valeur décimale `97`.

Q1. Écrivez une fonction `void affiche_caractere (char car);` qui prend en entrée un caractère `car` et l'affiche d'abord comme un caractère puis comme un entier. Pensez à consulter l'exemple "variables" en cas de difficulté.

Testez votre fonction en donnant différentes lettres et autres caractères de la table ASCII.

Q2. De par cette capacité à manipuler les caractères comme étant des entiers, il est possible d'utiliser des opérations sur les entiers et les appliquer à des `char`. Par exemple, `'a' + 1` correspond au caractère `'b'`.

Écrivez une fonction `void alphabet (char* tableau, int taille);` qui prend en argument un tableau de caractères et un entier et remplis les `taille` premières cases du tableau par les `taille` premières lettres de l'alphabet dans l'ordre alphabétique.

Q3. Écrivez une fonction `void alphabet_renverse (char* tableau, int taille);` qui prend en argument un tableau de caractères et un entier et remplis les `taille` premières cases du tableau par les `taille` dernières lettres de l'alphabet dans l'ordre inverse de l'ordre alphabétique.

Q4. Affichez votre tableau contenant l'alphabet comme si c'était une chaîne de caractères, puis placez la valeur `'\0'` au milieu de votre tableau et réaffichez la chaîne de caractères. Que se passe-t-il ?

Q5. Modifiez votre tableau en modifiant des caractères avant et après le `'\0'` et en ajoutant d'autres `'\0'` avant et après votre premier `'\0'`.

Exercice 6 : Tableaux à deux dimensions

En C les tableaux sont en réalité des pointeurs vers la première des cases mémoires réservées pour stocker le tableau. Ainsi lorsque l'on passe un tableau en paramètre d'une fonction on passe en réalité uniquement ce pointeur vers la première case. De ce fait, comme vu en cours et dans les exercices précédents, il faut toujours passer en paramètre la taille du tableau en question. De ce fait, même s'il est possible de créer des tableaux à plusieurs dimensions en C, la maitrise de ces dimensions est perdue dès que vous passez le tableau en paramètre d'une fonction.

Si vous voulez utiliser un tableau à deux dimensions, il faut donc définir un tableau classique puis écrire les fonctions qui vous permettent de manipuler ce tableau. C'est ce que nous allons faire dans cet exercice.

Pour commencer, on va prendre pour exemple un tableau de 4 lignes et 4 colonnes. On donc un tableau de taille 16. À chaque fois que l'on va utiliser une fonction qui travaille sur un tableau à deux dimensions, nous devrons indiquer le nombre de lignes et le nombre de colonnes du tableau en question (tout comme on indique la taille d'un tableau à une dimension).

```
a  b  c  d
e  f  g  h
i  j  k  l
m  n  o  p
```

Q1. Écrivez une fonction : `char valeur_tableau_2d (char* tableau, int nb_ligne, int nb_colonne, int pos_ligne, int pos_colonne);` qui prend un tableau, son nombre de lignes, son nombre de colonnes et deux entiers et retourne la valeur présente à la ligne `pos_ligne` et la colonne `pos_colonne`.

Aide : Le calcul savant à effectuer pour trouver le bon numéro de case est :
`pos_ligne * nb_colonne + pos_colonne`.

Q2. Écrivez une fonction `void affiche_tableau_2d (char* tableau, int nb_ligne, int nb_colonne);` qui prend en paramètre un tableau et ses dimensions et l'affiche comme un tableau à deux dimensions. Vous utiliserez la fonction `valeur_tableau_2d` pour vous faciliter la tâche.

Aide : Pour parcourir un tableau à une dimension, vous utilisez une boucle `for`. Si vous passez à deux dimensions, il est plus simple d'en utiliser deux.

Testez votre fonction d'affichage en construisant un tableau contenant les 16 premières lettres de l'alphabet grâce à la fonction `alphabet` puis affichez ce tableau comme un tableau à deux dimensions de 4 lignes et 4 colonnes. Vous devez obtenir un affichage proche de celui donné en exemple.

Q3. Écrivez une fonction : `void place_valeur_tableau_2d (char* tableau, int nb_ligne, int nb_colonne, int pos_ligne, int pos_colonne, char car);` qui prend un tableau et ses dimensions et place la valeur `car` dans le tableau à la ligne `pos_ligne` et la colonne `pos_colonne`.

Testez votre fonction en modifiant certaines cases et en affichant à nouveau votre tableau à deux dimensions.

Exercice 7 : Entrée utilisateur

Dans cet exercice, nous allons nous intéresser aux entrées utilisateur ainsi qu'à leur traitement. Pour cela nous allons utiliser la fonction `scanf`. La fonction `scanf` est une fonction qui va enregistrer la chaine de caractères donnée par l'utilisateur dans le tableau qui lui est donné en paramètre. Il nous faut donc définir un tableau de caractères.

Q1. En utilisant `scanf`, écrivez une fonction `void demande_valeur(char* entree, int taille);` qui copie le texte entré par le joueur dans le tableau `entree`. (Aide : `scanf("%s",entree);` fait le travaille voulu, il vous reste plus qu'à l'encapsuler dans une fonction.)

Pour sélectionner une case, l'utilisateur doit écrire la lettre de la colonne en minuscule suivie du numéro de ligne (ex : "a3" sélectionne la ligne 3 colonne 0).

Q2. Écrire une fonction `int numero_colonne (char* entree, int taille);` qui va chercher le premier caractère de `entree` (qui doit être la lettre correspondant à la colonne voulue) et retourne un entier correspondant au numéro de la colonne demandée.

Q3. Écrire une fonction `int numero_ligne (char* entree, int taille);` qui va chercher le deuxième caractère de `entree` (qui doit être le numéro correspondant à la ligne voulue) et retourne un entier correspondant au numéro de la ligne demandée.

Q4. Écrivez une fonction `char valeur_demande_tableau_2d (char* tableau, int nb_ligne, int nb_colonne` qui prend un tableau `tableau` à deux dimensions, demande à l'utilisateur d'entrer une case et la fonction retourne le caractère du tableau contenu à la position demandée.

Démineur

En utilisant toutes les fonctions que nous avons vu précédemment, nous allons programmer un démineur. Le Démineur est un jeu vidéo de réflexion dont le but est de localiser des mines cachées dans une grille représentant un champ de mines virtuel, avec pour seule indication le nombre de mines dans les zones adjacentes. (merci Wikipédia !)

Dans ce jeu, le joueur est face à un terrain représenté par un quadrillage. Chaque case de ce quadrillage contient, de manière cachée au joueur, soit une mine, soit rien du tout. En sélectionnant une case, le joueur va la révéler. S'il révèle une mine, il a perdu, sinon le jeu continu. Le jeu se termine lorsque toutes les cases ne contenant pas de mines ont été révélées. Pour l'aider, lorsque le joueur révèle une case sans mine, un numéro s'inscrit sur cette case correspondant au nombre de mines se trouvant autour de cette case, diagonales comprises.

Dans ce jeu, il y a deux types d'informations. On retrouve les informations cachées au joueur et celles connues par le joueur. Pour les représenter, nous allons donc utiliser deux structures de données, sous forme de tableau :

- Le tableau `char mines[];` contient la position des mines.
  - 'M' → la case contient une mine.
  - 'V' → la case est vide.
- Le tableau `char terrain[];` correspond à l'information que possède le joueur.
  - '?' → le joueur n'a pas d'information sur cette case.
  - '@' → la case vient d'être révélée et contenait une mine qui vient d'exploser (le joueur à perdu).
  - '0' → la case est révélée et il y a 0 mine autour de cette case.
  - '1' → la case est révélée et il y a 1 mine autour de cette case.

— '2' → la case est révélée et il y a 2 mines autour de cette case.
— etc.

Lorsque l'on va définir les tableaux `mines` et `terrain`, nous allons devoir leur donner des tailles fixes. Or il est fort probable que l'on veuille a posteriori modifier ces tailles pour avoir un terrain de jeu plus grand ou plus petit. Pour cela, nous allons utiliser les `#define`. La syntaxe est la suivante :

```
#define MOT_CLEF VALEUR
```

Cette instruction doit être placée juste après le bloc d'instruction `#include`. Elle sera exécutée avant la compilation et va remplacer toutes les occurrences de `MOT_CLEF` dans votre code par `VALEUR`. Cela permet de pouvoir facilement changer une valeur partout dans le code sans avoir à vérifier que le changement a bien été fait partout. De plus comme le remplacement est fait avant la compilation, il est possible d'utiliser `MOT_CLEF` pour définir la taille d'un tableau. Nous nous servirons de ces `#define` pour fixer le nombre de lignes et de colonnes du terrain de notre démineur. Pour commencer, nous utiliserons donc :

```
#define NB_LIGNES 4
#define NB_COLONNES 4
```

Par la suite vous utiliserez les constantes `NB_LIGNES` et `NB_COLONNES` à chaque fois que vous aurez besoin de faire référence au nombre de lignes ou de colonnes de votre terrain. Cela permettra de redimensionner facilement votre démineur plus tard.

Exercice 8 : Initialisation du démineur

Q1. Écrivez une fonction `int demineur (void);` dans laquelle vous définissez le tableau `mines` où toutes les cases sont initialisées à `'V'` et le tableau `terrain` où toutes les cases sont initialisées à `'?'`.

Q2. Ajoutez trois mines en position (1,3), (2,0), (3,2) en utilisant la fonction `place_valeur_tableau_2d`. Affichez à nouveau le tableau de mines.

Exercice 9 : Dévoillage du terrain

Q1. Écrivez une fonction `int combien_de_mines_autour(char* mines, int nb_lignes, int nb_colonnes, int ligne, int colonne);` qui regarde les 8 cases autour de la position indiquée et compte le nombre de mines qu'elles contiennent. Attention au bord du plateau !

Q2. Écrivez une fonction `void revelation_du_terrain(char* terrain, char* mines, int nb_lignes, int nb_colonnes, int ligne, int colonne);` qui révèle la case indiquée en mettant à jour le tableau `terrain`. N'oubliez pas que le tableau `terrain` est un tableau de caractères.

Exercice 10 : Un démineur fonctionnel

Q1. Écrivez une fonction `int est_ce_fini (char* mines, char* terrain, int nb_lignes, int nb_colonnes)` qui regarde si toutes les cases ne contenant pas de mines ont été dévoilées. Elle retourne 1 si c'est le cas et 0 sinon.

Q2. Vous avez désormais tous les éléments pour finir votre première version du démineur. Pensez à indiquer au joueur lorsqu'il a gagné ou perdu.

Q3. Testez votre démineur.

Exercice 11 : Des fonctionnalités bien pratiques

Maintenant que vous avez un démineur fonctionnel, il est temps de rajouter quelques fonctionnalités pour améliorer l'expérience de jeu.

Q1. Lorsque vous dévoilez une case sans mine avec une valeur affichée qui est 0, vous savez qu'il n'y aucune mine autour. Vous pouvez donc faire en sorte que votre jeu dévoile directement les cases adjacentes. Si l'une des cases dévoilées est aussi un 0, vous devez répéter l'opération. Attention aux boucles infinies.

Nous allons désormais nous intéresser à générer aléatoirement le placement des mines. Pour cela vous devez initialiser votre générateur de nombre aléatoire à l'aide de l'instruction `srand(graine);` où `graine` est un entier que vous demandez à l'utilisateur. Vous pouvez ensuite obtenir des valeurs aléatoires en faisant appel à la fonction `rand`. Si vous voulez une valeur entre 0 et `NB_LIGNES-1` (valeurs choisies totalement au hasard bien sûr), l'instruction usuelle est `rand()%NB_LIGNES`.

Q2. Écrivez une fonction `int demande_entier(void);` qui demande un entier à l'utilisateur et retourne cette valeure.

Q3. Vous utiliserez `demande_entier` au début du programme pour initialiser votre générateur aléatoire grâce à l'instruction `srand(graine);`. Ainsi si vous utilisez deux fois la même graine, les mines seront placées aux mêmes endroits.

Q4. Écrivez une fonction `int place_random_mines(char* mines, int nb_lignes, int nb_colonnes, int nb_mines);` qui prend en entrée un nombre de mines et place ce nombre de mines de manière aléatoire sur le terrain. Demandez au joueur le nombre de mines qu'il veut pour sa partie et placez-les. Vérifiez que vous avez le bon nombre de mines (que vous n'avez pas placé deux mines au même endroit).

Q5. Votre démineur fonctionne-t-il toujours si vous augmentez la taille du terrain en modifiant vos `#define`, en écrivant par exemple :

```
#define NB_LIGNES 8
#define NB_COLONNES 8
```

Si ce n'est pas le cas, c'est que votre code n'a pas été écrit proprement en utilisant les constantes `NB_LIGNES` et `NB_COLONNES`. Corrigez-le pour que ce soit désormais le cas.

Exercice 12 : Pour aller toujours plus loin

Q1. Lorsque vous perdez, les mines sans drapeau doivent s'afficher avec un 'X'.

Q2. Ajoutez la possibilité de mettre des drapeaux, symbolisée par un `'P'`, pour indiquer l'emplacement de mines que vous avez déjà trouvé. Vérifiez que les drapeaux ne perturbent pas votre vérification de victoire.

Q3. Faites en sorte que si vous sélectionnez un drapeau, vous ne révélez pas la case, mais simplement vous retirez le drapeau.

Q4. Afficher le nombre de mines restantes, c'est-à-dire le nombre total de mines moins le nombre de drapeaux placés.

Q5. Modifier votre programme pour qu'il ne soit pas possible de tomber sur une mine au premier coup.

Q6. Modifier votre programme pour que le premier coup tombe forcément sur une case qui ne contient pas de mine et dont les 8 voisins ne contiennent pas de mines non plus.

Q7. Faites en sorte que si vous sélectionnez une case déjà découverte et que le nombre de drapeaux autour de cette case correspond au nombre inscrit sur cette case, alors votre jeu découvre toutes les cases autour ne contenant pas de drapeaux.

Q8. Faites en sorte que si vous sélectionnez une case déjà découverte et que le nombre de cases non découvertes autour de cette case correspond au nombre inscrit sur cette case, alors votre jeu place un drapeau sur chacune de ces cases non découvertes.

Q9. (Bonus) Comment vérifier que votre démineur est soluble sans avoir besoin de faire appel à la chance ?

Exercice 13 :

Votre code est-il utilisateur-proof ?

Q1. Faite en sorte qu'un utilisateur ne puisse pas faire planter votre démineur.

Q2. Vérifier en faisant tester votre démineur par votre voisin malveillant.