

Travaux Pratiques n°1

Programmation C

—L2 MIDO—

Premiers pas

Il est conseillé de créer un répertoire sur votre compte pour l'UE de C, puis un répertoire par TP.

Vous n'avez pas le droit d'utiliser d'IDE type Geany ou Codeblocks. Vous devez utiliser un éditeur de texte et un terminal. Vous êtes libre de choisir l'éditeur que vous souhaitez, du moment qu'il dispose au moins des fonctions suivantes :

- indentation automatique ;
- coloration syntaxique ;
- police à chasse fixe ;
- tabulation produit des espaces programmables ;
- annulation ;
- correspondance entre parenthésage ;
- affichage des numéros de lignes ;
- copie de sauvegarde.






Sur les machines du CRIO, il y a par exemple *Gedit*. Sous Mac, *bbedit* par exemple. Sur tous les systèmes (Windows, Linux, Mac), *Atom* ou *Visual Studio*.





Vous compilerez avec la commande `gcc` (ou `clang`). Voir la fiche d'aide mémoire sur Moodle pour le reste.

On conseille l'utilisation de l'option `-Wall` lors de la compilation. L'usage de variables globales est à proscrire. Vous devez choisir des noms de variables et de fonctions les plus clairs et pertinents possibles. Vous devez vous tenir à une seule convention de nommage.

Vous devez tester chaque programme que vous écrivez, en vous assurant au préalable qu'il compile sans erreur ni warning.

► Exercice 1. - de 18

1. Lancer le terminal via l'icône ou le menu.
2. Par défaut, vous êtes actuellement dans votre répertoire utilisateur. Aller dans le dossier `Documents` en tapant `cd Documents`  (la commande `cd` permet de changer de répertoire).
3. Lister le contenu de ce répertoire avec la commande `ls`.
4. Créer un répertoire avec la commande `mkdir TPC`  (`mkdir` permet de créer un répertoire).
5. Aller dans ce répertoire avec `cd TPC` . Noter qu'en tapant `T` , vous avez l'auto-complétion.
6. Créer un répertoire avec `mkdir TP1` .

7. Aller dans ce répertoire avec `cd TP1` .
8. Si vous êtes sous Linux, taper `gedit exo1.c &` . Si vous êtes sous Mac, taper `open exo1.c` . ⁽¹⁾ Gedit est un éditeur de texte (si vous avez l'habitude d'utiliser un autre éditeur de texte, vous pouvez l'utiliser à la place). `exo1.c` est le fichier que l'on crée. Ajouter `&` à la fin permet de garder la main dans le terminal.
9. Écrire un programme qui demande l'âge de l'utilisateur, puis qui affiche MINEUR ou MAJEUR selon sa réponse.
10. Dans le terminal, **compiler** votre programme avec `gcc -Wall exo1.c`. S'il y a des erreurs, corriger et relancer la commande (plutôt que de retaper, faites ). Si rien n'est écrit, c'est que la compilation de votre programme s'est bien passée.
11. La *compilation* a créé un *exécutable* appelé par défaut `a.out` ⁽²⁾. L'**exécuter** avec la commande `./a.out` dans le terminal.

► Exercice 2. Debugage

L'équipe C a écrit une solution pour l'exercice calculant les solutions d'une équation du second degré. Malheureusement le programme ne fonctionne pas ! Vous allez devoir le corriger et le faire fonctionner. Pour récupérer le fichier, aller sur le répertoire ETUD/DEMI2E_L2_ProgC/ (ou sur Moodle) et copier, dans votre répertoire personnel, le fichier `debug_secondDegre.c`.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     double a, b, c;
6     double x = 0, x1 = 0, x2 = 0;
7     printf("a ?");
8     scanf("%lf", &a);
9     printf("b ?");
10    scanf("%lf", &b);
11    printf("c ?");
12    scanf("%d", &c);
13
14    double delta = b*b - 4ac;
15    if (delta < 0)
16        printf("Il n'y a pas de solutions reelles.\n");
17    else if ((delta == 0)) {
18        x = -b / 2*a;
19        printf("Solution double : %f\n", x);
20    } else
21        x1 = (-b + sqrt(delta))/2*a;
22        x2 = (-b - sqrt(delta))/2*a;
23        printf("Deux solutions : %f et %f\n", x1, x2);
24    return 0;
25 }
```

(1). `open` sous Mac ouvre l'application par défaut pour ce fichier. Pour choisir un programme en particulier, par exemple `bbedit`, faire `open -a bbedit exo1.c`

(2). Probablement `a.exe` sous windows

Compiler le programme avec la commande :
`gcc -Wall debug_secondDegre.c -o exo2 -lm`

1. Que vous indique le premier warning ? Quelle est la ligne erronée ? Est-ce un warning important ou peut-on le négliger ? Corriger l'erreur.
2. À quelle ligne se situe l'erreur suivante ? Comment la corriger ?
3. Quelle est la fonction inconnue du dernier warning ? Que vous indique la note située en dessous ? Corriger l'erreur.

Le programme compile maintenant sans erreur, mais il faut absolument le tester !

4. On va tester le cas $a = 1$, $b = 2$ et $c = 1$. Quel est le résultat attendu ? Exécuter le programme ; les solutions sont-elles correctes ? l'affichage est-il correct ? Simuler le fonctionnement du programme pour comprendre d'où vient l'erreur et corriger-la.
5. Retester le même cas. L'affichage est-il correct ? Pourquoi ce second `printf()` s'affiche-t-il ? Corriger l'erreur.
6. On va tester le cas $a = 2$, $b = -8$ et $c = 6$. Quel est le résultat attendu ? Exécuter le programme ; les solutions sont-elles correctes ? Bien lire le code pour corriger l'erreur. Vérifier en retestant le cas.
7. Pensez-vous que le programme fonctionne maintenant correctement ? Tester le cas $a = 4$, $b = -8$ et $c = 4$. Quel est le résultat attendu ? Exécuter le programme ; la solution est-elle correcte ? Auriez-vous pu penser à ce problème avant même de tester ? Corriger le code.

► Exercice 3. Comptage

Écrire un programme qui demande à l'utilisateur le nombre d'entiers qu'il désire saisir, puis compte le nombre d'entiers positifs, négatifs ou nuls entrés par l'utilisateur. Ne pas utiliser de tableau.

► Exercice 4. Akinator

Écrire un programme qui demande à l'utilisateur de trouver un nombre entre 0 et 100. Ce programme indique à chaque étape si la valeur saisie est inférieure ou supérieure au nombre secret. L'utilisateur doit trouver en au plus 10 essais. S'il gagne, le programme doit l'informer du nombre de coups qui ont été nécessaires.

Le nombre secret est une valeur inscrite dans le code. Il peut être défini aléatoirement (entre 0 et 99), en utilisant le code suivant (faire `man 3 random` pour plus d'information...) :

```
1 #include <stdlib.h>
2 #include <time.h>
3
4 int main() {
5     ...
6     srand(time(NULL)); // initialisation de la graine de l'aléa
7     int guess = random()%100; // tirage aléatoire
8     ...
9 }
```

► **Exercice 5. Rectangles**

1. Écrire un programme qui demande la saisie de deux entiers positifs *hauteur* et *largeur* par l'utilisateur, puis affichant un rectangle composé d'étoiles de la hauteur et de la largeur saisie.
2. Écrire ensuite le programme dessinant un rectangle de même taille, mais vide au milieu.
Par exemple, pour les valeurs 5 et 7, on attend :

```
*****
*****
*****
*****
*****

*****
*       *
*       *
*       *
*       *
*****
```

► **Exercice 6. Nombre parfait**

Un entier positif est dit parfait s'il est égal à la somme de ses diviseurs, excepté lui-même. Par exemple, 6 est parfait car $6 = 1 + 2 + 3$.

1. Écrire un programme déterminant si un entier entré par l'utilisateur est parfait.
2. En déduire un programme qui affiche les nombres parfaits inférieurs à n , et un autre affichant les n premiers nombres parfaits. On rappelle que `ctrl+c` permet d'interrompre un programme.

► Exercice 7. Gare aux flottants

Le but de cet exercice est de faire comprendre qu'il faut faire attention lorsque les flottants sont utilisés. Des arrondis peuvent être effectués par le programme C si le nombre n'est pas rationnel car il n'est pas possible de stocker un nombre infini de bits...

1. Afficher le résultat de l'opération $1.2e18 + 55.55$ (e18 équivaut à 10^{18} , soit un nombre très grand).
2. Essayer de deviner le résultat du programme suivant :

```
1 #include <stdio.h>
2 int main() {
3     float f = 0.0;
4     int i;
5     for(i = 0; i < 10; i++)
6         f += 0.1;
7     if (f == 1.0)
8         printf("f egal 1.0\n");
9     else
10        printf("f different de 1.0\n");
11    printf("%.12f, %.12f\n", f, 1.);
12    return 0;
13 }
```

Tester et expliquer.

Indice : remplacer l'instruction d'affichage après la boucle par

```
printf("%.12f, %.12f\n", f, 1.);
```

L'option `-Wfloat-equal` ajouté à *gcc* permet de détecter les tests d'égalités entre flottants.

Pour plus d'explications et des exemples (à base de fusée Ariane qui explose) :

<https://introcs.cs.princeton.edu/java/91float/>

► Exercice 8. Gare aux entiers aussi

Le but de cet exercice est de comprendre les erreurs de calcul provenant de dépassement de capacité. Ainsi, même lorsqu'on manipule des entiers, il ne faut pas oublier que la capacité de représentation des nombres par un ordinateur est limitée.

1. Exécuter le programme suivant :

```
1 #include <stdio.h>
2 #include <limits.h>
3
4 int main() {
5     int n = INT_MAX;
6     printf("n vaut : %d\n", n);
7     n++;
8     printf("n + 1 vaut : %d\n", n);
9     return 0;
10 }
```

Que se passe-t-il ? Que représente `INT_MAX` ?

- Écrire un programme qui déclare deux entiers non signés, initialise l'un à 1 et l'autre à -1, et affiche leur valeur (avec `%u` dans un `printf()`). Tester et expliquer l'affichage obtenu.
- Exécuter le programme suivant :


```
1 #include <stdio.h>
2
3 int main() {
4     for (unsigned int i = 10; i >= 0; i--)
5         printf("%u\n", i);
6     return 0;
7 }
```

Que se passe-t-il ? Pourquoi ?

- Écrire un programme qui affiche les valeurs limites et le nombre d'octets utilisés des types `int`, `unsigned int` et `long`. Il faudra utiliser la fonction `sizeof(type)` qui renvoie le nombre d'octets utilisés pour représenter le type passé en paramètre. Tester et vérifier que les valeurs affichées sont bien celles attendues. N'hésitez pas à tester d'autres types dans votre programme.

NB : les valeurs limites des `int`, `unsigned int` et `long` sont : `INT_MIN`, `INT_MAX`, `UINT_MAX`, `LONG_MIN` et `LONG_MAX` ; elles sont définies dans le fichier d'entête `<limits.h>`

► Exercice 9. [Bonus] Mini Calculette

Écrire une mini calculette permettant les opérations de bases (addition, soustraction, multiplication, division). Les opérations seront entrées via un seul `scanf` par l'utilisateur selon la forme suivante : `operande1 operateur operande2` , où les opérands sont des entiers et l'opérateur un caractère. Utiliser un `switch`.

► Exercice 10. [Bonus] Puissances de 5

Il n'existe que 3 nombres pouvant être écrits comme étant la somme de leurs chiffres à la puissance 4.

$$1634 = 1^4 + 6^4 + 3^4 + 4^4$$

$$8208 = 8^4 + 2^4 + 0^4 + 8^4$$

$$9474 = 9^4 + 4^4 + 7^4 + 4^4$$

On ignore $1 = 1^4$ car ce n'est pas une somme.

La somme de ces 3 nombres est $1634 + 8208 + 9474 = 19316$.

Trouver la somme de tous les nombres pouvant être écrits comme étant la somme de leurs chiffres à la puissance 5. (Commencer par trouver quel pourrait être le plus grand de ces nombres.) (Pour mettre un nombre à la puissance 5 il suffit de le multiplier 5 fois avec lui-même...).

Travaux Pratiques n°2

Programmation C

—L2 MIDO—

Tableaux

Ce TP aborde les tableaux en C. Dans la suite, tous vos tableaux auront pour taille maximum N , où N est une constante définie en début de programme avec `#define`. Tout ce qui est écrit doit être testé.

► Exercice 1. Tableaux

On désire faire fonctionner le programme suivant qui se trouve aussi dans le répertoire ETUD/DEMI2E_L2_ProgC/ et sur Moodle :

```
#define N 50

int main() {
    int tab1[N], tab2[N], tab3[N];
    int nb;
    srand(time(NULL));

    do {
        printf("Entrez le nombre de valeurs que vous désirez\n");
        scanf("%d", &nb);
    } while (nb > N);

    Remplissage(tab1, nb);
    Affichage(tab1, nb);
    printf("%d %d\n", Position(tab1, nb, 2), Position(tab1, nb, 4));

    Remplissage(tab2, nb);
    Somme(tab1, tab2, tab3, nb);
    Affichage(tab3, nb);

    return 0;
}
```

1. Écrire une fonction **Remplissage** prenant un tableau et sa taille en arguments et qui remplit le tableau avec des valeurs aléatoires entre 0 et 10 (voir TP1).
2. Écrire une fonction **Affichage** qui prend en arguments un tableau et sa taille et qui l'affiche.
3. Écrire une fonction **Position** prenant un tableau en paramètre, sa longueur, ainsi qu'un entier e , et renvoyant la première position de l'entier e dans le tableau, ou -1 si e n'est pas présent.
4. Écrire une fonction **Somme** prenant en argument deux tableaux **tabA** et **tabB** (que l'on suppose de même taille), ainsi qu'un tableau **resultat**, t.q. $\forall i \in [0..N-1], resultat[i] = tabA[i] + tabB[i]$.

► **Exercice 2. Histogramme**

Écrire une fonction `Histogramme` recevant deux tableaux `notes` et `histo` ainsi que la longueur du tableau `notes`. Ce dernier contient des notes entre 0 et 20 dans chaque case. Le tableau `histo` a pour taille 21 et représente, après appel à la fonction `Histogramme`, l'histogramme du premier tableau, i.e. `histo[i]` donne le nombre de valeurs égales à i dans `notes`. Vous devez utiliser **une seule boucle** pour remplir `histo` après son initialisation. Tester.

► **Exercice 3. Ératosthène**

Le crible d'Ératosthène est une méthode efficace pour calculer les entiers premiers inférieurs à N . Pour cela, il suffit d'écrire les entiers entre 2 et N , puis de rayer tous les multiples de 2 du tableau, puis tous les multiples du premier entier non rayé (3, puis 5...), jusqu'à ce que l'entier dont il faut supprimer les multiples soit supérieur à \sqrt{N} (vous n'avez pas besoin de la fonction racine carrée pour cela, car si $i < \sqrt{N}$, $i^2 < N$). À la fin de cette opération, tous les entiers non rayés sont des entiers premiers inférieurs à N . Écrire une fonction remplissant un tableau de N entiers tel que la case i contienne 1 si i est premier, 0 sinon. Vérifier en affichant les entiers premiers inférieurs à 100.

► **Exercice 4. [Bonus] Pascal**

Écrire un programme demandant à l'utilisateur la saisie d'un numéro n puis affichant la ligne numéro n du triangle de Pascal. On suppose que ce numéro de ligne est compris entre 1 et 20. Afin d'afficher cette ligne, vous devez calculer toutes les lignes du triangle de Pascal de 1 à n puis afficher la ligne n . Vous utiliserez pour cela **un seul** tableau, en remarquant que dans ce triangle, la ligne i est de taille i .

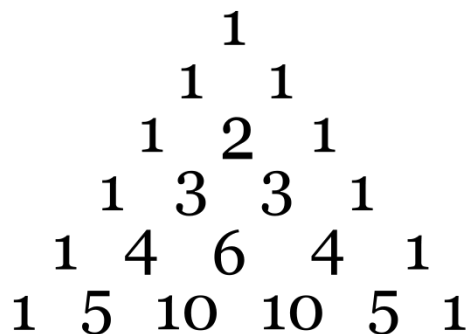


FIGURE 1 – Les 6 premières lignes du triangle de Pascal

► **Exercice 5.** [*Bonus 2*] **Somme à k et comparaison**

Reprendre l'exercice du TD demandant de trouver l'ensemble des paires de cases d'un tableau dont la somme vaut k . On désire comparer en pratique les deux algorithmes proposés lors du TD.

Pour cela, on va créer un tableau de taille N , où N est très grand (au moins 150000). On va remplir le tableau avec des nombres aléatoires entre 0 et $N * 4$ (par exemple).

Puis, on va calculer le temps pris par chacun des deux algorithmes (sur le même tableau) à l'aide de la fonction `time` utilisé comme suit :

```
1 #include <time.h>
2 int main() {
3     time_t start, end;
4
5     time(&start);
6     // code à mesurer
7     time(&end);
8
9     // ne fonctionne qu'à partir d'une seconde
10    printf("%f sec\n", difftime(end, start));
11    return 0;
12 }
```

Travaux Pratiques n°3

Programmation C

—L2 MIDO—

Chaînes de caractères

Une fois encore, on utilisera des tableaux dont la taille max est définie par une constante. On rappelle qu'une chaîne de caractères est un tableau de `char` terminé par le caractère `'\0'`.

► Exercice 1. Concaténation

Écrire une fonction recevant deux chaînes de caractères et recopiant la seconde à la suite de la première. Attention à vérifier qu'il y a assez d'espace dans le tableau recevant le résultat ! La fonction renvoie 0 s'il n'y avait pas assez de place et 1 sinon. Le tableau obtenu doit rester une chaîne de caractères. On vous donne le `main()` suivant :

```
#define N 13

int main() {
    char s1[N] = "salut ";
    char s2[N] = "c'est cool!";
    printf("%s%s\n", s1, s2);
    if (concat(s1, s2))
        printf("%s\n", s1);
    else
        printf("Not enough space\n");
    return 0;
}
```

Note : cette fonction existe dans la bibliothèque `string` sous le nom de `strcat()` (sans le contrôle de longueur).

► Exercice 2. Remplacer

Écrire une fonction recevant une chaîne de caractères et deux caractères, qui remplace toutes les occurrences du premier caractère par le second dans la chaîne. La fonction renvoie le nombre de remplacements effectués.

► Exercice 3. César

Écrire une fonction prenant en arguments une chaîne de caractères `ch` et un entier `d` et modifiant `ch` selon le codage de César, c'est-à-dire en décalant de `d` lettres dans l'alphabet chaque caractère de la chaîne. Par exemple, pour `XYLOPHONE` et un décalage de 2, le résultat est `ZANQRJQPG`. Écrire la fonction permettant de faire l'opération inverse. Pour cet exercice, on suppose que tous les caractères de la chaîne sont des lettres majuscules.

► Exercice 4. Anagrammes

Écrire une fonction recevant deux chaînes de caractères et déterminant (en renvoyant 1 ou 0) si elles sont des anagrammes (deux mots utilisant les mêmes lettres, par exemple *TSARINE* et *ENTRAIS*). Vous pourrez par exemple utiliser un tableau local de 256 entiers stockant le nombre d'occurrences de chaque code ASCII. Il est plus élégant d'utiliser **un seul** tableau annexe.

► Exercice 5. Occurrence joker

Écrire une fonction prenant deux chaînes de caractères en argument. Cette fonction renvoie l'indice dans la première chaîne où se trouve n'importe lequel des caractères de la seconde, ou -1 si aucun n'est présent.

Tester avec un `main()`. Par exemple, pour `("bonjour", "xo")` on attend 1, pour `("bonjour", "zaf")` on attend -1.

► Exercice 6. Grep

Écrire une fonction recevant deux chaînes de caractères et déterminant si la seconde apparaît dans la première. Si oui, elle renvoie l'indice de sa première occurrence, -1 sinon. Par exemple, l'appel avec `"ma belle bababouche me bouche !"` et `"babouche"` doit renvoyer 11.

Comparer votre fonction à la fonction `strstr()` de la bibliothèque `string` de prototype : `char* strstr(const char* chaine, const char* chaineARechercher);`. Attention, cette fonction ne renvoie pas l'indice mais un pointeur sur le début de la chaîne trouvée (ou NULL si non présent).



► Exercice 7. [Bonus] Latin cochon

Écrire une fonction qui prend deux chaînes de caractères en argument et qui place dans la seconde la version *pig latin* de la première (on suppose qu'il y a assez d'espace dans la seconde chaîne).

Les règles de modification sont les suivantes :

- si le mot commence par une voyelle, on ajoute *way* à la fin du mot ;
- sinon, on déplace toutes les consonnes précédant la première voyelle à la fin du mot et on ajoute *ay*.

On pourra utiliser la fonction d'occurrence faite précédemment et la fonction de concaténation de la bibliothèque `string` qui est `strcat()`.

Par exemple :

- *egg* devient *eggway*,
- *simple* devient *implesay*,
- *string* devient *ingstray*.

Travaux Pratiques n°4

Programmation C

—L2 MIDO—

Récurtivité

Ce TP introduit la notion de récursivité en C, notion puissante mais parfois dangereuse.

► Exercice 1. Croissons

Recopier et tester la fonction suivante :

```
#include <stdio.h>

void croissRec(int a) {
    if (a == 0) {
        return;
    } else {
        croissRec(a-1);
        printf("%d\n", a);
    }
}
```

1. Qu'effectue cette fonction si on l'appelle avec un entier positif?
2. Écrire la fonction `void decroissRec(int a)` qui affiche les entiers de a à 1, en ordre décroissant.

► Exercice 2. Factorielle

1. Écrire une fonction prenant un entier a et retournant $a!$ de manière itérative.
2. Écrire une fonction calculant cette même factorielle mais de manière récursive, sachant que $0! = 1$ et $n! = n \cdot (n - 1)!$.

► Exercice 3. PGCD

On rappelle qu'Euclide a dit que si $b = 0$, $PGCD(a, b) = a$. Et que si $a = qb + r$ où r est le reste de la division euclidienne, alors $PGCD(a, b) = PGCD(b, r)$.

1. Calculer à la main $PGCD(142, 12)$.
2. Que se passe-t-il si on applique l'algorithme aux entiers 12 et 142 ? (et plus généralement si $a < b$?).
3. Écrire une fonction récursive calculant le PGCD de deux entiers a et b . On suppose que l'un de ces deux entiers est non nul.

► **Exercice 4. La puissance (du C)**

1. Écrire une fonction prenant un entier a et un entier strictement positif n et qui renvoie a^n de manière itérative.
2. Écrire une fonction récursive calculant cette même puissance. Tester.
3. À l'aide d'un pointeur, calculer le nombre d'appels à la fonction récursive (faire l'affichage du compteur **après** le calcul de la fonction).
4. Sachant que $a^0 = 1$, $a^n = (a^2)^{n/2}$ si n est pair et $a^n = a \cdot (a^2)^{(n-1)/2}$ si n est impair, écrire une fonction récursive **maligne** calculant cette même puissance. Tester, et comparer le nombre d'appels effectués à la fonction récursive (cette fonction étant **maligne**, elle doit utiliser moins d'appels que la fonction précédente).

► **Exercice 5. Fibo**

La suite de Fibonacci est définie telle que $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$.

1. Écrire une fonction récursive qui calcule le n^{e} nombre de Fibonacci. À l'aide d'un pointeur, vérifier que le nombre d'appels à la fonction récursive est bien $2 \cdot F_{n+1} - 1$.
2. Écrire une fonction itérative qui calcule le n^{e} nombre de Fibonacci. Combien y a-t-il de passages dans la boucle ?

► **Exercice 6. Palindrome**

Une chaîne de caractères d'au moins deux caractères est un palindrome si son premier et son dernier caractères sont identiques et si la sous-chaîne obtenue en les retirant est elle aussi un palindrome.

1. Écrire une fonction récursive renvoyant 1 si la chaîne prise en argument est un palindrome et 0 sinon (on pourrait supposer que la chaîne ne contient pas d'espaces et est en minuscules). Tester avec `kayak` et `kiwi` par exemple.



► **Exercice 7. Clef USB [Optionnel]**

On suppose que l'on dispose d'un ensemble de N fichiers MP3 et d'une clef usb de capacité 100 Mo. On désire pouvoir remplir la clef USB au maximum de sa capacité. Par exemple, pour l'ensemble de fichiers $\{10, 20, 30, 50, 20\}$, votre programme renvoie 100 ; pour $\{103, 97, 50, 40, 20\}$, il renvoie 97 ; pour $\{94, 55, 50, 40, 20\}$, il renvoie 95.

Pour cela, on remarque qu'il faut, lors du traitement de l'élément d'indice i , comparer les valeurs de :

1. la meilleure solution (en terme d'optimalité d'espace occupé) pour les éléments de d'indice 0 à $i - 1$ avec un espace disponible de "taille courante - taille de l'élément i " + la taille de l'élément i (l'élément i est dans la solution),
2. la meilleure valeur de la solution pour les éléments de 0 à $i - 1$ avec un espace disponible de "taille courante" (l'élément i n'est alors pas dans la solution).



► **Exercice 8. Bière** [Optionnel, Coding battle 2017]

Une formule secrète indique qu'il est possible de synthétiser de la bière en grande quantité et de grande qualité grâce à deux essences savamment utilisées datant des druides gaulois : le dauphinège et le midorégène.

La manipulation de ces essences s'effectue dans un fût de volume V . On commence par introduire un volume V_a de dauphinège et un volume V_b de midorégène. Trois opérations sont alors possibles :

1. la 3-dauphino-2-midolifération : le dauphinège arrive à un volume de $V_a \cdot V_a \cdot V_a$ et le midorégène à un volume de $V_b \cdot V_b$.
2. la 2-dauphino-3-midolifération : le dauphinège arrive à un volume de $V_a \cdot V_a$ et le midorégène à un volume de $V_b \cdot V_b \cdot V_b$.
3. la rattrapagation : le volume des deux essences diminue, V_a devenant $\lfloor \sqrt{V_a} \rfloor$ et V_b devenant $\lfloor \sqrt{V_b} \rfloor$.

À tout moment, le volume de $V_a + V_b$ ne peut évidemment dépasser V .

Désirant synthétiser le maximum de bière étant donné un fût d'une certaine taille et des quantités initiales d'essences, donner le volume maximal atteignable en appliquant successivement une des trois opérations.

Par exemple, pour :

- $V = 1242$, $V_a = 2$, $V_b = 3$, le volume maximal est de 1241 (opération 1 puis 2 puis 3 puis 1).
- $V = 20$, $V_a = 2$, $V_b = 3$, le volume maximal est de 17 (opération 1).
- $V = 301$, $V_a = 14$, $V_b = 7$, le volume maximal est de 257.

On peut utiliser la fonction `sqrt` si on inclut `math.h` et que l'on compile avec `-lm` en plus.

Travaux Pratiques n°5

Programmation C

—L2 MIDO—

Manipulation de fichiers

Le but de ce TP est la manipulation de fichiers en C. On rappelle que les arguments de `main(int argc, char* argv[])` sont tels que `argv` est un tableau de `argc` chaînes de caractères, correspondant aux arguments donnés sur la ligne de commande.

► Exercice 1. Echo

Écrire un programme qui affiche sur la sortie standard les paramètres qu'il a reçus en argument de la ligne de commande.

Par exemple, si l'exécutable s'appelle `a.out`, la commande :

```
./a.out truc.txt machin.bla
```

obtiendra l'affichage suivant :

```
truc.txt machin.bla
```

► Exercice 2. Affichage d'une image

La suite de ce TP travaille sur la manipulation des fichiers en C. En particulier, on travaille dans un premier temps sur des fichiers *PGM ASCII*. Le format de fichier PGM (*Portable graymap*) est un format d'images matricielles dans lequel un niveau de gris est associé à chaque pixel de l'image. Les lignes d'un fichier binaire PGM indiquent, dans cet ordre, les informations suivantes :

- le nom magique du format permettant d'identifier sous quel format sont encodées les informations : `P2` pour un fichier PGM ASCII ;
- un nombre indiquant la largeur de l'image, un espace, puis un nombre indiquant la hauteur de l'image ;
- une valeur maximale utilisée pour coder les niveaux de gris (un pixel noir est codé par la valeur 0, un pixel blanc par la valeur maximale) ;
- les données de l'image sous forme matricielle : chaque pixel est codé par une valeur en caractère ASCII, séparée par un caractère d'espacement.

Récupérer les fichiers PGM proposés sur Moodle et les enregistrer dans le même répertoire que votre code source. Ouvrez-les avec `gedit`. Les fichiers sont-ils bien au format PGM ASCII ? Afficher maintenant l'image grâce au programme unix `display` (on peut l'utiliser en tapant `display image.pgm` dans le terminal).

► Exercice 3. Lecture d'un fichier PGM

Pour manipuler des fichiers en C, il faut utiliser une variable de type `FILE *`. Les fonctions suivantes (on donne leur **signature**, pas la manière des les appeler!) permettent la lecture/écriture dans un fichier (**man fonction** pour plus d'informations sur **fonction**) :

- `FILE* fopen(char *nom, char *mode);` : ouvre le fichier dont le nom est le premier paramètre selon le mode spécifié par le deuxième paramètre. Les principaux modes sont : `"r"` pour la lecture, `"w"` pour écriture et `"a"` pour l'ajout. Cette fonction retourne `NULL` en cas d'erreur ;
- `int fprintf(FILE *f, char *format,...);` et `int fscanf(FILE *f, char *format,...);` : ont le même comportement que les fonctions `printf()` et `scanf()`, excepté le fait qu'elles écrivent/lisent dans le flux passé en premier argument. En particulier, `fscanf()` retourne la constante particulière `EOF` (de type `int`) lorsque la fin du fichier est atteinte ;
- `int fclose(FILE *f);` : ferme le flux `f`.

Vous prendrez soin de bien tester les valeurs de retour de ces fonctions et de fermer les fichiers que vous avez ouverts. Les éventuelles erreurs ne devront pas entraîner l'arrêt du programme.

`fopen()` ouvre le fichier et place le « curseur » en début de fichier. `fscanf()` lit depuis la position du curseur et « avance » le curseur de ce qu'il a lu.

1. Écrire un programme qui affiche sur la sortie standard la largeur et la hauteur d'un fichier PGM passé en argument de la ligne de commande. Attention si le fichier n'existe pas...
2. Transformer le programme précédent pour qu'il affiche la largeur et la hauteur de plusieurs fichiers PGM passés en argument de la ligne de commande.

► Exercice 4. Chargement et sauvegarde d'un fichier PGM

1. Écrire un programme qui stocke les données d'un fichier PGM ASCII passé en argument de la ligne de commande dans des entiers `w`, `h` et `vMax` (pour les largeur, hauteur et valeur maximale de l'image) et dans une matrice d'entiers `matrix` qui contiendra les valeurs des pixels de l'image.
2. Écrire une fonction `saveImage` qui prend en paramètres une matrice d'entiers `T`, trois entiers `h`, `l` et `vMax` et une variable `nomFichier` de type `char*` et qui crée un fichier PGM de nom `nomFichier` représentant l'image `T` de hauteur `h`, largeur `l` et valeur maximale `vMax`.

► Exercice 5. Traitement d'images PGM

Pour les trois questions suivantes, sauvegarder l'image modifiée dans un nouveau fichier image.

1. **Inversion.** Écrire un programme qui inverse les couleurs d'une image PGM passée en ligne de commande. L'opération d'inversion consiste à modifier la valeur du pixel v par $v_{\text{Max}} - v$ où v_{Max} est la valeur maximale utilisée pour coder les nuances de gris (donnée dans l'en-tête du fichier PGM). Utiliser la fonction `saveImage()` pour écrire le résultat dans un nouveau fichier s'appelant `inv.pgm`. Vérifier avec `display`.
2. **Noir et blanc.** Écrire un programme qui transforme une image PGM en une image noir et blanc. Pour cela, il suffit de remplacer chaque pixel par la valeur 0 s'il est inférieur à $v_{\text{Max}}/2$ ou v_{Max} sinon.
3. **Flou.** Écrire un programme qui transforme une image PGM en la rendant légèrement plus floue. L'opération à appliquer pour cela consiste à remplacer chaque pixel par la valeur moyenne de ses quatre pixels voisins (au dessus, en dessous, à droite et à gauche). Par simplification, on ne modifie pas un pixel au bord de l'image (n'ayant donc pas quatre voisins).
4. (Optionnel) Modifier votre code pour que la sauvegarde de l'image inversée se fasse maintenant dans un fichier appelé `fichierOriginal-inv.pgm` où `fichierOriginal` est le nom de l'image originale. Pour cela, on peut manipuler les tableaux de caractères à la main, ou alors on peut utiliser les fonctions `sscanf()` pour récupérer le nom du fichier original avec le format

```
sscanf(nomOriginal, "%[^.].%s", base, ext);
```

(On lit tout sauf un `'.'` puis un `'.'` et la suite), et `sprintf()` pour lui accoler `-inv.pgm` (faire `man sprintf`).

Pour ceux qui voudraient aller plus loin concernant la compression d'images, voir cet article (changement de base, etc).⁽¹⁾

(1). <http://accromath.uqam.ca/2012/07/les-images-sur-la-toile-un-defi-de-taille/>

► Exercice 6. Bufferisation

Les fonctions suivantes peuvent aussi être utiles pour la lecture/écriture dans un fichier (`man fonction` pour plus d'informations sur `fonction()`) :

- `int fgetc(FILE* f)` : lit et retourne un caractère depuis le flux `f`. Retourne EOF lorsque la fin du fichier est atteinte.
- `char* fgets(char* s, int size, FILE* f)` : lit au plus `size - 1` caractères de `f` et les stocke dans `s`. La lecture se termine lors d'un retour à la ligne ou d'une fin de fichier. Retourne NULL si la fin du fichier est atteinte.
- `fread` et `fwrite`, pouvant s'utiliser de la façon suivante :
`fread(buf, sizeof(char), n, f)` et `fwrite(buf, sizeof(char), n, f)`, où `buf` est un tableau de `n` `char` déclaré auparavant et `f` est le retour de `fopen`. La fonction `fread` lit également au plus `n` caractères, mais ne s'arrête pas lors d'un retour à la ligne.

Vous prendrez soin de bien tester les valeurs de retour de ces fonctions et de fermer les fichiers que vous avez ouverts. Les éventuelles erreurs ne devront pas entraîner l'arrêt du programme.

Commencer dans un premier temps par télécharger un gros fichier texte, par exemple celui *des Misérables* de Victor Hugo : [ici](#).

Dans cet exercice, on désire observer l'intérêt de la bufferisation pour les entrées-sorties. Le but sera de recopier le fichier original dans un nouveau fichier, à l'identique.

Pour cela, on va comparer le temps pris par le programme pour effectuer la recopie du texte d'un fichier dans un autre :

- caractère par caractère (en utilisant les fonctions `fgetc()` et `fputc()`) ;
- en utilisant un buffer (tableau de caractères) de taille `i` pour chaque `i` entre 1 et 32 (en utilisant les fonctions `fread()` et `fwrite()`).

Pour calculer le temps pris par une fonction, on peut utiliser la fonction `clock()` après avoir inclus `time.h`.

► Exercice 7. Métro [Bonus]

Créer un programme prenant en argument le nom d'un fichier contenant une ligne de métro. On considère qu'un tel fichier possède le format suivant :

- la première ligne contient le texte "**nom**" suivi du caractère ':' suivi d'un espace suivi du nom de la ligne ;
- la seconde ligne contient le texte **nombre** suivi du caractère ':' suivi d'un espace suivi du nombre de stations de cette ligne ;
- jusqu'à la fin du fichier, il y a un nom de station par ligne.

Par exemple 7bis.

Faites en sorte de lire le fichier, puis d'afficher le nom de la ligne ainsi que ses terminus (première et dernière stations de la liste). Faites attention à ne pas lire plus de caractères que la taille de vos chaînes !

Travaux Pratiques n°6

Programmation C

—L2 MIDO—

Structures C

Ce TP s'intéresse à l'utilisation de structures en C. On rappelle que l'utilisation de l'option `-Wall` à `gcc` est recommandée.

► Exercice 1. Complexes

1. Définir un type `Complexe` permettant la représentation d'un nombre complexe, avec deux flottants.
2. Écrire une fonction permettant de saisir au clavier un complexe. Cette fonction prend l'adresse de ce complexe à saisir. Ajouter le mot clé `const` avant le type pris en argument. Que se passe-t-il ? Conclure.
3. Écrire une fonction permettant l'affichage d'un complexe. Cette fonction prend également l'adresse du complexe à afficher, on rappelle qu'un passage par valeur COPIE entièrement le contenu de la variable en mémoire, ce qui est déroutant pour les grosses structures. Pour assurer l'absence de modification de la variable passée en argument, on ajoute le mot clef `const` avant celle-ci.
4. Écrire une fonction testant l'égalité entre deux complexes pris en arguments. Elle renvoie 1 s'ils sont égaux, 0 sinon.
5. Écrire une fonction effectuant la somme entre deux complexes.
6. Tester à l'aide d'une fonction `main()`.

► Exercice 2. Tableau de complexes

1. Définir une structure `EnsembleComplexes` contenant un tableau de `Complexes` (de taille maximale définie par une constante) et un entier correspondant à sa cardinalité.
2. Écrire une fonction prenant en argument un `EnsembleComplexes` qui demande la saisie de n complexes, à placer dans le tableau, où n est défini par l'utilisateur. On réutilisera les fonctions de l'exercice précédent.
3. Écrire une fonction qui prend en argument un `EnsembleComplexes` et qui l'affiche.
4. Tester à l'aide d'une fonction `main()`.

► Exercice 3. Jeu des 7 erreurs

Recopier le programme suivant ou le récupérer sur Moodle.

```
#include <stdio.h>

typedef struct {
    char nom[10];
    char prenom[10];
    int numero;
} Etudiant

Etudiant CreerEtudiant(char *p, char *n, double num) {
    Etudiant res;
    res.numero = num;
    strncpy(res.nom,n,10);
    strcpy(res.prenom,p);
    return res;
}

void AfficherEtudiant(const Etudiant *e) {
    printf("%d %d %s\n",e.numero, e->prenom, e->nom);
}

int main() {
    Etudiant e1;
    Etudiant e2;
    e1 = CreerEtudiant("Nada","Mimouni",2);
    AfficherEtudiant(e1);
    e2 = CreerEtudiant("Ernest-Antoine","Seilliere",1);
    AfficherEtudiants(&e2);
    return 0;
}
```

1. Essayer de compiler.
2. Utiliser les erreurs données par le compilateur pour corriger le programme (il y a 7 problèmes).
3. Exécuter le programme. Affiche t-il ce qu'il devrait ?
4. Faire un man de `strcpy()`. Débugger le programme.

Travaux Pratiques n°7

Programmation C

—L2 MIDO—

Formes, voitures, garage, tris et makefile

Structures complexes, tris, programmation modulaire.

Dans le cadre de la programmation modulaire, on utilise plusieurs fichiers pour un seul programme final afin de séparer les fonctionnalités.

On place dans des fichiers `.h` (dits *headers*) les structures, ainsi que les signatures ou prototypes des fonctions pouvant être utilisées par l'extérieur. Dans les fichiers `.c`, on place le code des fonctions, ainsi que d'éventuelles fonctions utilitaires internes (dont le reste du programme n'a pas besoin de connaître l'existence).

L'inclusion d'un fichier *header* personnalisé dans un fichier se fait avec la commande `#include "monHeader.h"` (des guillemets plutôt que les chevrons réservés aux fichiers *headers* du langage).

Pour ne pas redéfinir plusieurs fois les mêmes fonctions, il faut que le compilateur ne recopie qu'une seule fois le code des fichiers *headers*. Pour cela, on encadre le contenu de chaque fichier `.h` par :

```
#ifndef FICHIER_H
#define FICHIER_H
et
#endif (en remplaçant FICHIER par le nom du fichier).
```

Pour **compiler**, on doit compiler l'ensemble des fichiers `.c` en même temps, en faisant :

```
gcc fic1.c fic2.c main.c -o sortie
```

ou bien plus simplement :

```
gcc *.c -o sortie
```

qui prend tous les fichiers `.c` du répertoire. Dans le dernier exercice, on verra comment automatiser cette phase.

► Exercice 1. Types

On va définir une voiture à l'aide de plusieurs autres structures intermédiaires. On désire les structures correspondant aux types suivants :

- un **cercle** est défini par son centre (deux entiers indiquant ses coordonnées) et son rayon ;
- un **rectangle** est caractérisé par les coordonnées de son coin supérieur gauche, sa longueur et sa hauteur ;
- une **voiture** (vue de côté) est composée de deux cercles (les roues), une partie basse (un rectangle), une partie haute (un rectangle) et deux fenêtres (deux rectangles) (vous utiliserez des pointeurs vers ces structures).

Pour vous aider, on donne le code de `cercle.h` et `voiture.h` :

```
#ifndef CERCLE_H
#define CERCLE_H
#include <stdio.h>

typedef struct {
    int x;
    int y;
    double r;
} cercle;

/* Initialise une variable cercle aux valeurs passees en entree */
cercle *creer_cercle(int x, int y, double r);

/* Prends en entree un cercle et un fichier HTML et dessine un cercle
 * dans le fichier HTML selon la valeur des champs du cercle */
void afficher_cercle(FILE *html, cercle *c);

/* desalloue le cercle de la memoire */
void libere_cercle(cercle *c);
#endif
```

```
#ifndef VOITURE_H
#define VOITURE_H
#include "cercle.h"
#include "rectangle.h"
typedef struct {
    cercle *avant;
    cercle *arriere;
    rectangle *bas;
    rectangle *haut;
    rectangle *fenetre_avant;
    rectangle *fenetre_arriere;
} voiture;

/* Initialise une variable voiture avec les valeurs passees en entree
 * (deja alloues) */
voiture *creer_voiture(cercle *avant, cercle *arriere, rectangle *bas,
    rectangle *haut, rectangle *f1, rectangle *f2);

/* Prends en entree une voiture et un fichier HTML ouvert en écriture
 * et y dessine une voiture selon ses champs dans le fichier */
void afficher_voiture(FILE *html, voiture *v);

/* Libere tout l'espace memoire utilise pour stocker une voiture */
void libere_voiture(voiture *v);
#endif
```

Écrire le code de `rectangle.h`.

► Exercice 2. Des fonctions pour chaque

Pour chacune des structures définies à l'exercice précédent, on désire avoir dans les fichiers `.c` correspondants, les fonctions de :

- création : prend en entrée les paramètres nécessaires à l'initialisation des champs, alloue l'espace mémoire et renvoie un pointeur vers une variable du type correspondant ;
- affichage : prend en entrée un pointeur vers un type, un fichier HTML ouvert en écriture et y affiche sa valeur ;
- libération : prend en entrée un pointeur et libère l'espace (et bien tout l'espace).

Pour l'affichage, la libération etc. vous devez utiliser les fonctions déjà écrites dans les autres fichiers (par exemple, pour afficher une voiture, on utilisera l'affichage d'un cercle). Cet affichage se fait en SVG dans un fichier HTML (afin d'avoir un rendu visuel). Pour cela, on écrit du code HTML dans ce fichier avec `fprintf()`.

Pour vous aider, voici le code de `cercle.c` :

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

cercle *creer_cercle(int x, int y, double r) {
    cercle *res = malloc(sizeof(cercle));
    if (res == NULL) return NULL;
    res->x = x;
    res->y = y;
    res->r = r;
    return res;
}

void afficher_cercle(FILE *html, cercle *c) {
    fprintf(html,
        "<circle cx='%d' cy='%d' r='%f' stroke='black' fill='grey' />\n",
        c->x, c->y, c->r);
}

void libere_cercle(cercle *c) {
    free(c);
}
```

1. Compiler uniquement `cercle` avec la commande `gcc -c cercle.c`. Tout se passe bien s'il n'y a pas de message (note : il n'y a pas d'exécutable créé avec l'option `-c`).
2. Écrire le fichier `rectangle.c`. Pour afficher une voiture, on utilisera le code HTML suivant dans le `fprintf()` :

```
"<rect x='%d' y='%d' width='%f' height='%f' stroke='black' fill='white' />"
```

3. Écrire le fichier `voiture.c`. L’affichage d’une voiture utilise les fonctions précédentes ! Pour éviter d’avoir les voitures toutes affichées au même endroit, on utilisera la code suivant (`nbV` est une variable globale à `voiture.c`) :

```
int nbV = 0;

void afficher_voiture(FILE *html, voiture_t *v) {
    fprintf(html, "<g transform=\"translate(0, %d)\">", nbV*50);

    // TODO: l'affichage de la voiture

    fprintf(html, "</g>");
    nbV++;
}
```

4. Compiler uniquement rectangle et voiture avec `gcc -c`.
5. Écrire dans un fichier `main.c` une fonction `main()` qui crée et affiche puis libère une voiture. Voici le code nécessaire pour l’affichage :

```
int main(void) {
    voiture *ferrari = ...

    FILE *html = fopen("voiture.html", "w");
    if (html == NULL) {
        printf("Error opening\n");
        return 0;
    }

    fprintf(html, "<!DOCTYPE html>\n<html>\n<body>\n");
    fprintf(html, "<svg height=\"800\" width=\"1000\">");

    afficher_voiture(html, ferrari);

    fprintf(html, "\n</svg></body>\n</html>");
    libere_voiture(ferrari);
    fclose(html);
}
```

Compiler et exécuter : la voiture sera dessinée dans un fichier `voiture.html` que vous pouvez visualiser avec un navigateur internet classique (par exemple en tapant dans le terminal `firefox voiture.html &`).

► **Exercice 3. Qui a la plus grosse voiture ?**

Ajouter pour chaque structure une **fonction de comparaison** qui prend en entrée deux pointeurs vers deux types et renvoie une valeur négative (resp. nulle, positive) si la valeur du premier type est inférieur (resp. égale, supérieure) au second.

Pour comparer des

- cercles : on compare les rayons ;
- rectangles : on compare les aires ;
- voitures : on compare la taille des parties basses.

► **Exercice 4. Seek and destroy**

1. Écrire une fonction qui prend en entrée un tableau de pointeurs sur des voitures et la taille du tableau, et renvoie l'indice du tableau contenant la plus petite voiture d'entre elles (au sens de la comparaison).
2. Écrire une fonction qui trie un tableau de voitures dans l'ordre croissant (sans utiliser d'indice pour indiquer le début du tableau). On utilisera un tri par sélection : on cherche d'abord la plus petite voiture que l'on place à l'indice 0 (par un échange), puis la seconde plus petite dans le sous-tableau commençant à l'indice 1 et ainsi de suite.

► **Exercice 5. Un garage bien rangé**

On se donne les deux fonctions suivantes :

```
voiture* creer_voiture_cabriolet(){
    cercle* avant = creer_cercle(50,50,6);
    cercle* arriere = creer_cercle(100,50,7);
    rectangle* bas = creer_rectangle(40,30,70,15);
    rectangle* haut = creer_rectangle(60,15,40,25);
    rectangle* f1 = creer_rectangle(70,25,10,15);
    rectangle* f2 = creer_rectangle(80,25,10,15);
    voiture* cabriolet = creer_voiture(avant,arriere,bas,haut,f1,f2);
    return cabriolet;
}

voiture* creer_voiture_familiale(){
    cercle* avant = creer_cercle(60,50,10);
    cercle* arriere = creer_cercle(110,50,12);
    rectangle* bas = creer_rectangle(40,20,80,20);
    rectangle* haut = creer_rectangle(50,15,60,25);
    rectangle* f1 = creer_rectangle(70,25,10,15);
    rectangle* f2 = creer_rectangle(80,25,10,15);
    voiture* cabriolet = creer_voiture(avant,arriere,bas,haut,f1,f2);
    return cabriolet;
}
```

1. Dans un fichier `garage.c`, écrire une fonction `main()` qui crée un garage de dix voitures sous forme d'un tableau. Pour chaque case du tableau, utiliser la fonction `random()` pour choisir aléatoirement entre une voiture familiale et une voiture décapotable.
2. Trier les voitures du garage.
3. Afficher le contenu du garage. Est-il bien trié ?
4. Vérifier qu'il n'y a pas eu de fuite mémoire avec `valgrind`.

► Exercice 6. Makefile

Pour éviter de compiler des parties du code déjà compilées et non modifiées, on peut écrire un *Makefile*.

Un fichier makefile est constitué de règles de la forme

cible: dépendances
commande

où commande est précédé d'une tabulation. Par défaut, `make` exécute la première règle rencontrée. Il regarde alors les dépendances. S'il s'agit de la cible d'autres règles, alors elles sont évaluées. Si un fichier dans les dépendances est plus récent que la cible, alors la commande est exécutée (donc, on ne recompile pas des parties du code qui n'ont pas été modifiées).

Pour un projet minimal contenant un fichier `main.c`, un fichier `hello.h` et un fichier `hello.c` (la fonction `main()` de `main.c` appelant une fonction située dans `hello.c` et déclarée dans `hello.h`), un makefile possible est le suivant :

```
CFLAGS=-Wall

all: monprog

monprog: hello.o main.o Makefile
    gcc -o monprog hello.o main.o

hello.o: hello.c
    gcc -o hello.o -c hello.c $(CFLAGS)

main.o: main.c hello.h
    gcc -o main.o -c main.c $(CFLAGS)

clean:
    rm -rf *.o monprog
```

1. Écrire un makefile pour compiler le garage sous la forme d'un fichier se nommant `Makefile` et se trouvant dans le même répertoire que les fichiers sources.

Travaux Pratiques n°8

Programmation C

—L2 MIDO—

Listes Chaînées

Nous continuons l'étude des listes chaînées débutée en TD.

Dans la suite, on utilise la définition suivante pour représenter une liste chaînée d'entiers :

```
typedef struct maillon {
    int val;
    struct maillon *suiv;
} Maillon;

typedef struct {
    Maillon *premier;
} Liste;
```

► Exercice 1. Opérations de base

Implémenter les fonctions suivantes et les tester dans une fonction `main()`. Les éventuelles erreurs seront renvoyées à cette fonction, où les actions seront prises en fonction de ces erreurs.

1. Une fonction `creer_maillon()` qui prend un entier et renvoie un pointeur sur un `Maillon` après avoir effectué les allocations nécessaires, ou `NULL` si une erreur a lieu.
2. Une fonction `liste_init()` qui alloue l'espace nécessaire pour une structure de liste et la renvoie, ou `NULL` si une erreur a lieu.
3. Une fonction `liste_ajout_debut()` qui prend un pointeur sur une liste ainsi qu'un entier et ajoute un maillon contenant cet entier au début de la liste.
4. Une fonction `liste_saisie()` qui prend un pointeur sur une liste et ajoute en tête un nombre fixé par l'utilisateur de maillons, chacun avec une valeur donnée par l'utilisateur.
5. (Optionnel) Une fonction `liste_lecture()` qui prend un pointeur sur une liste et un `FILE *` (représentant un fichier déjà ouvert contenant des entiers séparés par des espaces). La fonction lit les entiers contenus dans le fichier et insère chaque entier en tête de liste.
6. Une fonction `liste_affiche_it()` qui prend un pointeur sur une liste et l'affiche itérativement.
7. (Optionnel) En déduire une fonction `liste_ecrit_it()` qui prend un pointeur sur une liste et un `FILE *`, et qui écrit le contenu de la liste dans ce dernier. Remarquer que la fonction peut être appelée avec pour deuxième argument `stdout`.
8. Une fonction `liste_affiche_rec()` qui prend un pointeur sur une liste et appelle une fonction `maillon_affiche_rec()` affichant les maillons récursivement.
9. Une fonction `liste_affiche_inv()` qui prend un pointeur sur une liste et l'affiche récursivement manière inversée.

► Exercice 2. Recherches

1. Une fonction `liste_recherche()` qui prend en arguments un pointeur sur une liste et un entier et qui retourne l'adresse du premier maillon contenant une occurrence de l'entier, ou `NULL` si l'entier n'est pas présent.
2. Une fonction `liste_min()` qui prend en argument un pointeur sur une liste et retourne l'adresse du maillon contenant la valeur minimale de la liste passée en argument.
3. Une fonction `liste_dernier()` qui prend en argument un pointeur sur une liste et retourne l'adresse du dernier maillon de la liste.

► Exercice 3. Insertion

Implémenter les fonctions suivantes en utilisant ce qui a déjà été fait :

1. Une fonction `liste_insere_fin()` qui prend en arguments un pointeur sur une liste et un entier, et qui insère un nouveau maillon contenant la valeur prise en argument en fin de liste.
2. Une fonction `liste_insere_apres()` qui prend en arguments un pointeur sur une liste et deux entier *val* et *ap*, et qui insère un nouveau maillon contenant la valeur *val* après le maillon contenant *ap* (on n'insère pas si *ap* n'est pas présent).

► Exercice 4. Concaténation

Écrire une fonction `liste_concatenation()` prenant deux listes en arguments et qui ajoute la seconde liste à la première. On peut supposer que la première liste n'est pas vide.

► Exercice 5. Suppressions

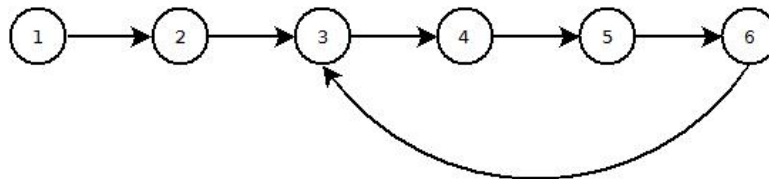
1. Écrire une fonction `libere_liste()` libérant l'espace alloué pour une liste donnée en paramètre (la liste et ses maillons).
2. Écrire une fonction `liste_suppr()` prenant en arguments une liste et un entier et qui supprime la première occurrence de l'entier donné dans la liste donnée.
3. Écrire une fonction `liste_suppr_inf()` supprimant toutes les occurrences inférieures à un entier donné dans une liste donnée. Attention, que se passe t-il si plusieurs occurrences du même entier à supprimer sont à la suite ?
4. Tester vos fonctions en ajoutant le contenu de ce fichier <https://www.lamsade.dauphine.fr/~sikora/ens/c/tp8-tests.c> à la place de votre main.
5. Vérifier que tout l'espace alloué a bien été libéré avec l'outil `valgrind` (le nombre de `free()` doit être égal au nombre de `malloc()`). Pour l'utiliser, il faut d'abord compiler avec l'option `-g`, puis lancer `valgrind` avec en argument le nom du programme. Par exemple :

```
gcc -g tp8.c
valgrind ./a.out
```

Vous pouvez aussi ajouter les options `-g -fsanitize=address` à `gcc` pour vérifier cela.

► **Exercice 6.** [*Bonus*] **LaFontaine**

1. Que se passerait-il si on fait l’affichage d’une liste comme celle-ci :



2. On veut donc faire une fonction `liste_cycle_detection()` qui renvoie 1 si une liste possède un cycle et 0 sinon, en temps linéaire et espace mémoire supplémentaire constant. Pour cela, on utilise l’algorithme suivant tiré du lièvre et de la tortue :
- (a) Le lièvre et la tortue vont tous les deux parcourir la liste, mais le lièvre ira deux fois plus vite que la tortue.
 - (b) S’il y a un cycle, le lièvre se retrouvera sur la même case que la tortue à un moment donné.
 - (c) S’il n’y a pas de cycle, le lièvre atteint la fin de la liste.

On veut que le code suivant fonctionne :

```
int main() {
    Liste *l = liste_init();
    liste_ajout_debut(l, 6);
    liste_ajout_debut(l, 5);
    liste_ajout_debut(l, 4);
    liste_ajout_debut(l, 3);
    liste_ajout_debut(l, 2);
    liste_ajout_debut(l, 1);
    liste_affiche_it(l);
    printf("%d\n", liste_cycle_detection(l));

    //on ajoute un cycle
    Maillon *a = liste_recherche(l, 3);
    Maillon *b = liste_dernier(l);
    b->suiv = a;

    //liste_affiche_it(l); //boucle infinie !
    printf("%d\n", liste_cycle_detection(l));
    return 0;
}
```

Travaux Pratiques n°9

Programmation C

—L2 MIDO—

Listes doublement chaînées

On implémente les listes doublement chaînées.

► Exercice 1.

1. Écrire une structure **Maillon** représentant un maillon, contenant la valeur **val** du maillon, ainsi qu'un pointeur **prec** vers le maillon précédent et un pointeur **suiv** vers le maillon suivant.
2. Écrire une structure **Liste** représentant une liste, contenant un entier **taille** stockant la taille de la liste, un pointeur **premier** vers le maillon de tête et un pointeur **dernier** vers le maillon de fin.
3. Écrire une fonction **creer_maillon()** qui prend un entier et crée un maillon contenant cet entier en effectuant les allocations nécessaires. La fonction renvoie un pointeur sur ce **Maillon** ou **NULL** si une erreur a lieu.
4. Écrire une fonction **liste_init()** qui crée et renvoie un pointeur sur une **Liste** vide après avoir effectué les allocations nécessaires, ou **NULL** si une erreur a lieu.
5. Écrire une fonction **liste_ajout_debut()** qui prend un pointeur sur une liste ainsi qu'un entier, et qui ajoute un maillon contenant cet entier au **début** de la liste.
6. Écrire une fonction **liste_ajout_fin()** qui prend un pointeur sur une liste ainsi qu'un entier, et qui ajoute un maillon contenant cet entier en **fin** de la liste.
7. Écrire une fonction **liste_affiche()** qui prend un pointeur sur une liste et l'affiche itérativement dans le **sens classique**.
8. Écrire une fonction **liste_affiche_inverse()** qui prend un pointeur sur une liste et l'affiche itérativement dans le **sens inversé**. Même question récursivement.
9. Écrire une fonction **liste_extraire_debut()** extrayant et retournant le **premier** maillon de la liste. Attention aux cas particuliers !
10. Écrire une fonction **liste_extraire_fin()** extrayant et retournant le **dernier** maillon de la liste.
11. Écrire une fonction **liste_supprimer()** supprimant le premier maillon correspondant à l'entier pris en argument (il retourne le maillon ou **NULL** si absent). Attention aux cas particuliers !
12. Écrire une fonction **liste_tourner_droite()** effectuant une permutation à **droite** de manière circulaire de la liste prise en paramètre, sans création de nouvelle cellule. Par exemple, la liste (1,2,3,4,5) devient (5,1,2,3,4).

13. Écrire une fonction `liste_tourner_gauche()` effectuant une permutation à **gauche** de manière circulaire de la liste prise en paramètre sans création de nouvelle cellule. Par exemple, la liste (1,2,3,4,5) devient (2,3,4,5,1). Quelles sont les complexités des deux dernières fonctions? Quelle aurait été cette complexité si on avait utilisé un tableau?
14. Écrire une fonction `liste_libere()` libérant l'espace alloué pour une liste donnée en paramètre.
15. Vérifier que tout l'espace alloué a bien été libéré avec l'outil `valgrind` (le nombre de `free()` doit être égal au nombre de `malloc()`) ou en ajoutant les options `-g` `-fsanitize=address` à `gcc`.
16. Testez votre programme en ajoutant le contenu de ce fichier <https://www.lamsade.dauphine.fr/~sikora/ens/c/tp9-tests.c> à la place de votre fonction `main()`.