IN 301 Langage C – TD

Version du 06 octobre 2019

Youssef Ait El Mahjoub – youssef.ait-el-mahjoub2@uvsq.fr
Pierre Coucheney – pierre.coucheney@uvsq.fr
Coline Gianfrotta – coline.gianfrotta@ens.uvsq.fr
Franck Quessette – franck.quessette@uvsq.fr
Yann Strozecki – yann.strozecki@uvsq.fr

Table des matières

Durant ce semestre vous travaillerez sous l'environnement Linux de la machine virtuelle que ce soit pour les TDs ou pour le projet.

Dans cet environnement, vous utiliserez le terminal afin de vous déplacer dans l'arborescence des fichiers, d'éditer, compiler, débugger votre programme, et de mettre vos fichiers sous contrôle de version sur un répertoire distant.

Pour cela, vous utiliserez les programmes suivants :
— `geany` pour l'édition,
— `gcc` pour la compilation,
— `gdb` pour le débuggage,
— `git` pour le contrôle de version.

Biblio

— **Le langage C, 2ème édition** – Brian W. Kernighan, Dennis M. Ritchie
— **Programmer en langage C - 5ème édition** – Claude Delannoy

Plan du Cours

Cours 1

Environnement de programmation, éditeur, compilateur, arborescence des fichiers, fenêtre shell, ligne de commande, commandes shell de base, commande de compilation.

Cours 2

Entrées sorties dans les fichiers, sur l'écran, principe et intérêt du Makefile.

Cours 3

Adresse mémoire, Opérateurs `*` et `&`, malloc, free.

Cours 4
Chaînes de caractères, Implémentation des tableaux multidimmensionnels, arguments de la ligne de commande, atoi, atof, constructeur de type (tableaux et `struct`).

Cours 5
Compilation séparée

Cours 6
Listes chaînees simples

Cours 7
Implémentations de piles et files

Cours 8
Contrôle

Cours 9
Processus, `fork()`, `exec()`, Environnement d'exécution d'un processus

Cours 10
Cours 11
Cours 12
Cours 13
Cours 14
Cours 15

1 Environnement de programmation

Dans un premier temps, vous allez apprendre les fonctionnalités de base du terminal. Les commandes que vous utiliserez sont :

```
man, ls, cd, pwd, mkdir, mv, rm, wc, cat
```

leur description peut être obtenue avec la commande `man commande`. Deux fonctionnalités du terminal qui s'avèrent très utiles à l'usage, et que vous vous efforcerez d'utilisez dès le début, sont la complétion automatique avec la touche tabulation, et le parcours de l'historique des commandes avec les flêches haut et bas.

Exercice 1. Terminal et compilation

a. Ouvrez un terminal. Dans quel répertoire vous trouvez-vous ? Que se passe-t'il si vous appuyer sur la touche `w` puis deux fois sur `TAB` dans le terminal ? Puis `wh` et deux fois sur `TAB` ? Puis `whe` et une fois sur `TAB` ? Que fait la commande que vous venez de taper ? Essayer `whereis ls`.

b. Déplacez vous vers le répertoire `Bureau`. Revenez dans le répertoire initial, puis encore dans le répertoire `Bureau` sans taper complètement au clavier `Bureau`.

c. Affichez ce que contient le répertoire `Bureau`. Créez un répertoire `essai`, et vérifiez que celui-ci a bien été créé.

d. Déplacez vous dans le répertoire `essai`, puis éditez un fichier vide `fic1` avec `geany` en tapant la commande :

```
geany fic1 &
```

puis fermez `geany`. Que contient le répertoire maintenant ?

e. Copiez le fichier vers un fichier `fic2`. Changez le nom de `fic2` en `fic3`. Supprimez le fichier initial `fic1`.

f. Supprimez tous les fichiers et ajoutez le programme `debug.c` qui est sur `e-campus2`. Affichez le programme dans le terminal. Combien de lignes et de caractères contient-il ?

g. Nous allons maintenant compiler le programme. Taper la commande

```
gcc debug.c
```

Vérifiez qu'un fichier `a.out` a été créé. Executez le programme en tapant la commande `./a.out`. Manifestement ce programme contient un bug... mais vous ne le corrigerez pas pour l'instant.

h. Pour choisir un nom à votre exécutable, tapez la commande `gcc -o nom debug.c`. Vérifiez qu'un exécutable ayant le nom que vous avez choisi a bien été créé, et exécutez le.

i. Compilez en ajoutant les warnings de compilation avec la commande

```
gcc -o nom -Wall debug.c
```

Que constatez-vous ?

Notez bien que pour compiler un programme qui utilise la librairie mathématique, il faut ajouter l'option `-lm`. Au final, cela donne la commande suivante :

```
gcc -o nom -Wall debug.c -lm
```

Exercice 2.

Organisation de vos dossiers Afin de de stocker correctement vos fichiers vous devez créer les dossiers suivants :

- un dossier `IN301`
- dans ce dossier un dossier par TD, commencez donc par un dossier `TD 01 - Environnement de programmation` correspondant au TD de cette section
- Vous pouvez également créer un dossier par séance de TD avec la commande :

```
for x in $(seq  1 12); do mkdir "td_$x"; done
```

- Vous pouvez créer un dossier `Notes de Cours` et un dossier `Projet`.
- Pour cette séance vous travaillerez dans le dossier `TD 01 - Environnement de programmation`.
- Vous sauvegarderez tous les fichiers de ce TD dans ce dossier.

Exercice 3.

Utilisation de commandes shell

a. Écrire un programme C qui génèrer dix milles nombres aléatoires entre 0 et 1 000. Utiliser la commande :

```
man 3 rand
```

pour voir la syntaxe des fonctions `rand()` et `srand()`. Dans votre programme pour afficher à l'écranla syntaxe est :

```
int a;
a = rand();
printf("%d\n",a);
```

b. Utiliser `srand()` avec comme argument `getpid()` pour changer l'initialisation des nombres aléatoires. Faites `man 2 getpid()`

c. Quelle est la commande de compilation correcte ?

d. Utiliser une redirection pour rediriger le résultat dans un fichier `nombres.data`.

e. Utiliser la command `wc` pour compter le nombre de nombres générer

f. Utiliser la command `sort -n` (faites `man sort` pour voir la doc) pour trier le fichier et rediriger le tri vers un autre fichier `nombres_tries.data`

g. Utiliser la command `uniq` pour supprimer les doublons et mettre le résultat dans `nombres_uniq.data`. Combien y a-t-il de nombres différents ?

h. mettre toutes ces commandes dans un fichier, le rendre exécutable et éécutez le.

i. Enchaîner toutes les commandes en une seule ligne pour compter le nombre de nombres différents.

Exercice 4. Contrôle de version : git

Vous allez utiliser un gestionnaire de version qui s'appelle `git`, et un dépôt distant qui s'appelle `github`. Il y a (au moins) 2 intérêts à cela par rapport à une sauvegarde classique : d'une part vous conservez l'historique de toutes les versions et pouvez revenir en arrière en cas d'erreur ou de suppression inopinée de votre travail, et d'autre part, vous faites des sauvegardes sur un dépôt distant qui vous prémunit des accidents que pourrait toucher votre machine personnelle. L'objectif de cet exercice est de vous familiariser avec l'utilisation de cet outil que vous utiliserez tout au long du semestre (et au-delà je l'espère !).

a. Ouvrez un navigateur et allez sur la page web : https://github.com/

b. Créez un compte en mémorisant bien votre nom d'utilisateur et votre mot de passe.

c. Une fois que vous êtes connecté, créez un dossier qui s'appelle `IN301` en cliquant sur la croix en haut à droite de l'écran (create new repository). Votre dépôt distant sous gestionnaire de version git est créé, vous allez maintenant le récupérer sur votre machine.

d. Dans votre terminal, déplacez vous à l'endroit où vous voulez ajouter le dossier `IN301`. Créez une copie locale en tapant la commande suivante dans le terminal :

```
git clone https://github.com/moi/IN301
```

et en remplaçant `moi` par votre nom d'utilisateur. Un répertoire `IN301` a normalement été créé, ce que vous pouvez vérifier en tapant la commande `ls`.

e. Déplacez vous dans le répertoire `IN301`. Créez le dossier `td0` (commande `mkdir`) [1]. Allez dans le dossier `td0` et créez le fichier vide `essai` avec la commande `touch essai`. Nous allons maintenant ajouter ce fichier au répertoire distant.

f. Pour mettre un nouvel élément en contrôle de version, il faut utiliser la commande `git add fichier`, donc, ici, `git add essai`. Pour valider cet ajout, utiliser la commande `git commit essai`, ou la commande `git commit -a` qui validera toutes les modifications faites sur des fichiers qui sont sous contrôle de version. Une fenêtre s'ouvre pour que vous renseigniez une description de la modification. Ecrivez, par exemple, "ajout fichier td0/essai", cliquez ensuite sur `CTRL+x`, puis sur `o` (pour OUI) et `ENTREE`. Pour propager cela au répertoire distant, il reste à taper la commande `git push`. Vérifiez ensuite sur votre compte `github` que le dossier a bien été ajouté.

g. Ajoutez le fichier `debug.c` que vous avez manipulé dans l'exercice précédent dans le répertoire `td0`. En suivant les mêmes instructions que pour `td0` faites en sorte de mettre ce fichier sous gestion de version et de l'ajouter dans `github`.

h. Une fois que cela a bien été réalisé, supprimez le répertoire `IN301` de votre machine (commande `rm -rf IN301`) et vérifiez que cela a bien fonctionné. Faites maintenant un clône de votre dépôt distant `github`, et vérifiez que vous avez bien récupéré le dossier `td0` et qu'il contient le fichier `debug.c`.

i. Ajoutez du texte dans le fichier `essai` et sauvegardez. Testez alors la commande `git status`. Commitez le changement. Que renvoie la commande `git status` ? Poussez votre changement sur `github` et testez de nouveau `git status`.

Dorénavant, pour ceux qui travaillent avec une machine en prêt, vous pourrez récupérer vos travaux en début de séance en clonant votre répertoire distant.

A tout moment, vous pouvez "commiter" vos changements (ne pas oublier de faire `git add` pour les dossiers et fichiers nouvellement créés), et les propager avec la commande `git push`. Localement vous pouvez vérifier si vos fichiers sont bien enregistré avec la commande `git status`.

Enfin, notez qu'il ne faut commiter que les fichiers textes (typiquement les programmes terminant par l'extension .c) ainsi que les répertoires qui les contiennent, et non les exécutables.

Exercice 5. Compilation et débug

Récupérez le fichier `debug.c` de l'exercice précédent. Nous allons utiliser le logiciel `gdb` afin de débugger ce programme.

a. Sans éditer le programme `debug.c`, compilez et exécutez le (voir exercice 1).

Les commandes de base du débugger `gdb` sont

```
run, quit, break, bt, print, step, next
```

b. Pour exécuter le programme dans le débugger, lancer la commande `gdb ./progDebug` dans le terminal, puis la commande `run`. Que se passe-t'il ?

c. Essayez d'ajoutez un point d'arrêt à la ligne 11 (commande `break 11`). Cela ne doit pas être possible car l'exécutable ne sait pas faire référence au fichier source. Il faut pour cela ajouter l'option `-g` à la compilation. Pour cela quittez `gdb` (commande `quit`) et recompilez en ajoutant l'option.

1. Dorénavant, pour la feuille de td i, vous créerez un dossier `tdi` à cet emplacement

d. Retournez dans `gdb` et ajoutez un point d'arrêt à la ligne 11 puis lancez l'exécution du programme. Afficher la pile des appels (commande `bt`), et la valeur des variables dans la fonction courante. Continuez l'exécution du programme pas à pas en affichant les valeurs des variables régulèrement.

e. Corrigez la fonction `factorielle`, puis faites de même pour les fonctions `somme` et `maximum` en vous aidant si nécessaire de `gdb`.

2 Lecture et écriture dans les fichiers

Dans toute cette section, comme dans la suite du poly, vous écrirez un `Makefile` permettant de compiler et d'exécuter les programmes.

Exercice 6. Premier fichier `.h`

Créer un fichier `constantes.h` qui contient les lignes suivantes qui définissent trois constantes :

```
#define N 10000
#define MAX 1000000
#define NOMFIC "nombres.data"
```

Exercice 7. Écriture dans un fichier

a. Écrire un programme `genere.c` qui génère `N` entiers positifs aléatoires compris entre zéro et `MAX`. Ces nombres seront écrits dans le fichier `NOMFIC`. Afin de pouvoir utiliser les constantes dans ce programme, ajouter au début du programme, la ligne :

```
#include "constantes.h"
```

L'ouverture, l'écriture et la fermeture du fichier se feront dans une fonction :

```
void ecrire ()
```

Dans le `main` l'appel se fera par :

```
ecrire ();
```

Formattez l'affichage des nombres avec `%6d` pour que le fichier généré soit plus lisible.

Exercice 8. Lecture depuis un fichier

a. Écrire un programme `algos.c` qui déclare en variable globale un tableau de taille `N` :

```
int T[N];
```

et qui lit les nombres qui sont dans le fichier `NOMFIC` généré à l'exercice précédent et les stocke dans les cases du tableau de `0` à `N-1`.

Vous devez faire une fonction :

```
void lecture ()
```

qui lit dans le fichier `NOMFIC` fichier et remplit le tableau `T`. Vous devez comme dans l'exercice précédent écrire :

```
#include "constantes.h"
```

au début de votre programme.

b. Programmer une fonction qui écrit les nombres qui sont dans le tableau dans un fichier `nombres-verif.data`

c. Utiliser la commande `diff` pour vérifier que les fichiers sont identiques :

```
diff nombres.data nombres-verif.data
```

diff affiche les différences, donc si diff n'affiche rien c'est que les deux fichiers sont identiques.

d. Modifier le code en supposant que l'on ne connait pas le nombre de valeurs dans le fichier `NOMFIC`. On peut alors utiliser la valeur de retour de `fscanf` pour savoir «quand s'arrêter». Tapez

```
man 3 fscanf
```

et lisez la section `valeur de retour`.

Le `Makefile` sera commun avec celui de l'exercice précédent.

Exercice 9. premier algo

a. Dans le programme `algos.c` écrire une fonction :

```
int recherche (int x)
```

qui renvoie `1` si le nombre `x` est dans le tableau et `0` sinon.

b. Dans la fonction `recherche` ajouter un compteur pour compter le nombre de comparaisons que fait la fonction. Afficher ce nombre de comparaisons avant le retour de la fonction.

c. Écrire une fonction `stat_recherche(int x)` qui fait comme la fonction `recherche` mais renvoie la valeur du compteur du nombre de comparaisons.

d. Dans la fonction `main` appeler dix mille fois la fonction `stat_recherche` en lui donnant en argument une valeur aléatoire entre zéro et $5 \times$ `MAX`. Calculer le nombre comparaisons en moyenne. Ce nombre était-il prévisible ?

Exercice 10. Commande wc

Programmer l'équivalent de la commande du terminal `wc -c` qui donne le nombre de caractères d'un fichier texte passé en argument. Le programme sera appelé de la manière suivante :

```
./monwc fichier
```

3 Pointeurs, addresses

Exercice 11. Pointeurs

a. Utiliser l'operateur :

```
sizeof
```

qui prend en argument un `int` pour afficher les tailles (en nombres d'octets) des types suivants :

```
char; int; double; char*; void*; int*; double*; int**; int[10]; char[7][3]; int[]
```

b. Soit `tab` un tableau de 10 float déclaré par :

```
float tab[10];
```

à quelles cases mémoire correspondent les différentes expressions suivantes (faites un dessin) :

```
tab; tab[0]; &tab[0]; *&tab; *&tab[0]
```

Donner leur type et afficher leur contenu et leur taille.

Exercice 12. Tableau statique à deux dimensions

Soit la déclaration :

```
int T[7][3]
```

a. Afficher la valeur de `T`, et l'adresse de `T[0][0]`, que constatez-vous ?

b. Afficher les adresses de `T[0][1]`, `T[0][2]`, `T[1][0]`, `T[1][1]`, que constatez-vous ?

c. Calculez l'adresse de `T[6][2]`.

Exercice 13. Valeur et référence

Soit le code :

```
void plusun (int a, int *T) {
    a = a+1;
    T[0] = T[0]+1;
}

int main() {
    int a;
    int T[10];
    a = 7;
    T[0] = 12;
    plusun(a,T);
}
```

modifier le code pour afficher `a` et `T[0]` avant et après l'appel à la fonction `plusun`. Que constatez vous, pourquoi ?

Exercice 14. Échange

Écrire une fonction qui ne renvoie rien et qui permet d'échanger le contenu de deux variables.

Exercice 15. Tableaux

a. Écrire une fonction qui alloue dynamiquement un tableau de 10 entiers et le remplit avec 10 entiers choisis aléatoirement entre 0 et 20 et renvoie un pointeur sur ce tableau.

b. Écrire une fonction qui affiche les éléments du tableau.

c. Écrire une fonction qui retourne le produit des éléments du tableau.

d. Écrire une fonction qui retourne la valeur minimale du tableau.

e. Écrire une fonction qui effectue un décalage de 1 case à droite de tous les éléments d'un tableau. La valeur de droite est perdue.

f. Écrire une fonction qui alloue un tableau de 10 entiers et le remplit avec 10 entiers en ordre croissant, le premier étant entre 0 et 10, le deuxième entre le premier et le premier plus dix, le troisième entre le deuxième le deuxième plus 10 et ainsi de suite.

g. Écrire une fonction qui insère une valeur dans un tableau trié. Après l'insertion, le tableau est toujours trié. La valeur la plus à droite est perdue.

h. Écrire une fonction qui inverse les éléments d'un tableau. Cette inversion s'effectue sur le tableau lui-même sans utiliser de tableau intermédiaire.

i. Écrire une fonction qui supprime un élément choisi au hasard dans le tableau, le supprime et décale les valeurs vers la gauche. La valeur de la case la plus à droite reste inchangée.

j. Écrire une fonction qui désalloue la mémoire réservée pour le tableau.

Exercice 16. Back to the file

a. Ecrire une fonction

```
int nb_entiers(char * nom)
```

qui prend en argument un nom de fichier et qui renvoie le nombre d'entiers stockés dans ce fichier.

b. Déclarer en variables globales :

```
int N;
int *T;
```

c. Écrire une fonction qui prend en argument un nom de fichier et alloue dynamiquement le tableau `T` et stocke dans `N` le nombre d'entiers contenus dans le fichier et rempli `T` avec les `N` valeurs.

d. Écrire une fonction qui renvoie l'indice de la case qui contient le plus grand élément du tableau.

e. Écrire une fonction qui prend en argument un tableau et échange le contenu de deux cases du tableau, que proposez-vous de donner en argument à la fonction, en plus du tableau lui même.

f. Écrire une fonction qui prend en argument `k` compris entre `0` et `N` qui recherche le plus grand élement dans le tableau entre `0` et `k` et échange le plus grand élément avec la valeur stockée à l'indice `k`.

g. Proposer un algo pour trier le tableau en utilisant la fonction précédente.

4 Chaînes de caractères

En C, il est possible d'initialiser un tableau de caractères par une chaîne de caractère de la manière suivante :

```
char *c = "salut"
```

Le tableau se termine alors par le caractère de fin de chaîne `'\0'`. Sur l'exemple, il s'agit donc un tableau de taille six. Cela permet l'usage de fonctions adaptées aux chaînes de caractères, fournies dans la librairie `string.h`, telle que la fonction `strlen()` qui donne la taille du tableau sans le caractère de fin de chaîne.

Pour avoir la liste des différentes fonctions de la librairie `string.h` taper :

```
man 3 string
```

Pour avoir le manuel d'une fonction en particulier, taper :

```
man 3 strlen
```

Exercice 17. Arguments de la ligne de commande

Une utilisation des chaînes de caractères est le passage d'argument au programme dans la ligne de commande. Pour cela, le prototype de la fonction `main()` doit être adapté de la manière suivante :

```
int main(int argc, char** argv)
```

où `argv` est un tableau de chaînes de caractères, et `argc est la taille du tableau`.

a. Tester et comprendre le programme `testarg.c` suivant :

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

en l'appelant avec une ligne de commandes qui contient n'importe quel nombre d'arguments, par exemple :

```
./testarg 6 toto salut2016 ?!:;
```

b. Pourquoi la variable `argv` est-elle de type `char**` ?

c. En utilisant la fonction de conversion `atof()` (voir le manuel), modifier le programme pour qu'il affiche la somme des nombres passés en arguments. Par exemple l'exécution

```
./testarg 6.2 -8 2.0
```

doit afficher `0.2`.

d. Que se passe-t-il si les arguments ne sont pas des nombres flottants ?

e. En utilisant la fonction `isdigit()` (voir le manuel), vérifier au préalable que les arguments passés sont bien des nombres flottants éventuellement négatifs.

Exercice 18. Tableau et liste de caractères

L'objectif est d'écrire des fonctions simples de manipulation des chaînes de caractères. Nous travaillerons avec deux chaînes de caractères qui seront passées en argument de l'appel au programme de la manière suivante :

```
./programme chaine1 chaine2
```

Vous devez avoir dans la `Makefile` une cible `test` qui appelle plusieurs fois le programme avec à chaque fois des arguments pertinents pour tester ce programme.

a. Écrire une fonction **itérative** qui calcule la longueur d'une chaîne de caractère. Afficher la longueur de chacune des chaînes passées en argument.

b. Écrire une fonction **récursive** qui calcule la longueur d'une chaîne de caractère. Afficher la longueur de chacune des chaînes passées en argument.

c. Écrire une fonction **itérative** qui calcule la différence entre les deux chaînes comme le fait `strcmp()`. Comparer les deux chaînes passées en argument.

d. Écrire une fonction **récursive** qui calcule la différence entre les deux chaînes comme le fait `strcmp()`. Comparer les deux chaînes passées en argument.

e. Écrire une fonction qui renvoie vrai si les chaînes sont miroir l'une de l'autre. Comparer les deux chaînes passées en argument.

f. Écrire une fonction qui renvoie vrai si la chaîne 2 est une sous-chaîne de la chaîne 1. Tester sur les deux chaînes passées en argument.

5 Structures de données, tri et complexité

5.1 Partie 1 : Tri d'entiers

Dans cette première partie, l'ensemble des fichiers nécessaires vous est fourni sur moodle uvsq dans un fichier `IN301_TD5_TRI_INT.zip`. Récupérez ce fichier et unzippez le et dans le terminal allez dans le dossier `IN301_TD5_TRI_INT`

Exercice 19. Compréhension de l'ensemble des fichiers

— Listez les fichiers du dossier `IN301_TD5_TRI_INT` et ouvrez le Makefile dans un éditeur.
— Faites un schéma présentant tous les fichiers et les relations entre fichiers. les relations sont de deux types : "est utilisé par" "est créé à partir de".
— Indiquez la cible du Makefile qui correspond à chacune des relations entre fichiers.

Exercice 20. Structure et premier tri

Le but de cet exercice est de faire une librairie de manipulation de tableaux d'entiers.

a. Dans le fichier `tabint.h` est défini le type :

```
struct tabint {
    int N;  // Taille du tableau
    int *T; // Pointeur vers le tableau
};
typedef struct tabint TABINT;
```

Expliquez cette définition.

b. Programmez dans le fichier `tabint.c` une fonction qui créé et remplit un tableau de `N` cases avec les valeurs aléatoires comprises dans l'intervalle `0..K` :

```
TABINT gen_alea_tabint (int N, int K)
```

c. Programmez dans le fichier `tabint.c` une fonction qui désalloue un tableau :

```
void sup_tabint (TABINT T)
```

d. Programmez dans le fichier `tabint.c` une fonction qui affiche un tableau :

```
void aff_tabint (TABINT T)
```

e. Programmez dans le fichier `tabint.c` une fonction `ech_tabint()` qui teste le contenu d'une case `i` du tableau avec le contenu de la case suivante `i+1`. Si le contenu de la case `i` est supérieure au contenu de la case `i+1`, les contenus sont échangés :

```
??? ech_tabint (TABINT T, int i)
```

A-t-on besoin de retourner le tableau pour qu'il soit modifié ?

f. Programmez dans le fichier `tabint.c` une fonction `scan_ech_tabint()` qui parcourt le tableau du début jusqu'à l'indice `fin` et effectue l'échange de la fonction précédente pour toute les cases de 0 à `fin-1` :

```
??? scan_ech_tabint (TABINT T, int fin)
```

A-t-on besoin de retourner le tableau pour qu'il soit modifié ?

g. Après un appel de cette fonction avec `fin` ayant comme valeur la taille du tableau, prouvez que la valeur maximum du tableau est à la fin du tableau.

h. Programmez dans le fichier `tri_bulle.c` une fonction qui appelle plusieurs fois la fonction `scan_ech_tabint()` et qui trie tout le tableau :

```
??? tri_bulle_tabint (TABINT T)
```

A-t-on besoin de retourner le tableau pour qu'il soit modifié ?

Vous avez programmé le tri à bulle.

i. Instrumentez le code pour compter le nombre de comparaisons entre cases et le nombre d'échanges de contenus de cases qui sont faits. Vous pourrez utiliser des variables globales.

Exercice 21. Analyse statistique

a. Dans le fichier `stat.h` est défini une structure permettant de stocker des stats :

```
struct stat {
    float nb_moy_comp;
    float nb_moy_ech;
};
```

b. Programmer dans le fichier `tri_bulle.c` une fonction qui prend en argument la taille du tableau `N` et un nombre de répétitions `A` et qui :

— génère et tri A fois un tableau de taille N

— renvoie le nombre moyen de comparaisons effectuées :

```
struct stat stat_moy (int N, int A)
```

c. Écrire un programme qui fait varier `N` entre 10 et 10^3 en le multipliant par 1.2, avec `A` fixé à 10^4 et qui calcule la moyenne du nombre de comparaisons et d'échanges.

Ce programme doit écrire les résultats dans un fichier `test_tri_bulle.data` en mettant sur chaque ligne, la valeur de `N` et la moyenne du nombre de comparaisons puis du nombre d'échanges.

d. Utilisez `gnuplot` pour afficher les courbes pertinentes et générer un pdf. Un fichier de commandes gnuplot vous est fourni : `stat.gplt`. Pour lancer gnuplot sur ce fichier de commandes, il suffit de taper :

```
gnuplot stat.gplt
```

e. Analysez de nouveau le Makefile.

Exercice 22. Comparaisons statistiques de tris

a. Programmer le tri fusion dans le fichier `tri_fusion.c`, vous utiliserez la fonction `fusionner` qui est à écrire dans `tabint.c`.

b. Programme la même chose que pour le tri à bulle pour faire des stats dans un fichier `test_tri_fusion.data`

5.2 Partie 2 : Tri de chaînes de caractères

Exercice 23. Trier des mots 1 : Quicksort

Dans cet exercice, nous allons voir comment trier des tableaux de chaînes de caractères selon l'ordre alphabétique.

a. Récupérer le programme source dans l'archive compressée `tri_mot.tar.gz` et le décompresser.

b. Éditer les fichiers `tableau.h` et `tableau.c` qui contiennent la structure à trier et des fonctions de manipulation qui sont fournies. Par défaut, le programme lit le fichier texte `romeoetjuliette` pour initialiser le tableau. Ce fichier peut être modifié dans le fichier `Makefile`.

c. Éditer le fichier `Makefile` et dessiner les liens entre les fichiers basé sur le Makefile

d. Dans le fichier `tri.c`, implémenter le tri rapide en utilisant la fonction `strcmp` de la librairie `string.h`.

Exercice 24. Trier des mots 2 : Tri par base

Nous allons maintenant implémenter le tri par base qui est une variante du tri par dénombrement adapté au tri de chaînes de caractères.

Le principe de ce tri est le suivant : on trie l'ensemble des mots selon leur dernier caractère en utilisant un tri par dénombrement. On fait de même sur l'avant dernier caractère et ainsi de suite jusqu'au premier caractère. Attention, à chaque fois, le tri utilisé doit être un tri stable.

a. Rappeler ce qu'est un tri stable.

b. Exécuter alors les étapes du tri par base sur les chaînes suivantes : $ca, cb, aa, ba, bc, bb, ac, aa, cb$.

c. Montrer que cet algorithme est correct.

d. Rappeler le principe du tri par dénombrement lorsque l'on trie des entiers entre 0 et K. Quelle est sa complexité ? En déduire la complexité du tri par base.

e. Afin d'adapter le tri par dénombrement pour trier des caractères, suivez les indications suivantes (on suppose que l'on est en train de trier selon le $i$ème caractère) :

- Indication 1 : Créer un tableau qui contient les fréquences d'apparition des 26 lettres de l'alphabet en position i dans l'ensemble des mots.
- Indication 2 : Si le '`a`' apparait $n_{\texttt{a}}$ fois ($n_{\texttt{a}}$ est donc la fréquence du '`a`'), et le '`b`' $n_{\texttt{b}}$ fois, où doit-on "ranger" tous les mots ayant un '`b`' en $i$ème position et dans quel ordre.
- Même question pour n'importe quelle lettre.
- Indication 3 : En supposant que l'on a déjà trié les mots selon les positions jusqu'à $i-1$, écrire l'algo pour trier jusqu'à la position i.

f. Écrire la fonction

```
int indice(char c)
```

qui convertit des caractères en indice dans le fichier `tri.c`.

g. Écrire la fonction

```
void tri_base_indice(Tableau t, int i)
```

qui implémente un tri par dénombrement stable des mots du tableau `t` selon le caractère `i` de chaque mot. Pour implémenter une version stable de ce tri, vous devrez utiliser un tableau auxiliaire (on dit que le tri n'est pas en place).

h. Comparer les temps d'exécution des deux algorithmes de tri (quicksort et tri par base).

Exercice 25. Trier des mots 3 : MSD

Le tri par base de l'exercice précédent est en fait la variante LSD (Least Significant Digit !). Il existe une autre variante appelée MSD (Most Significant Digit) qui peut être vue comme une adaptation du tri rapide (ou plus précisément du tri par paquets). Celle-ci consiste à trier les mots selon le premier caractère puis à trier récursivement l'ensemble des mots qui commencent par le plus petit caractère (c'est-à-dire le caractère `0'`), puis le deuxième plus petit (le caractère `'a'`), etc...

a. Programmer cet algorithme et comparer son temps d'exécution aux deux autres.

b. Écrire une fonction qui supprime les doublons.

6 Listes chaînées

Dans toute cette section vous devrez, pour chaque fonction que vous écrivez, la tester en l'appelant dans le `main`.

Toutes ces fonctions sont des fonction de base de manipulation des listes chaînées. Elles doivent absolument être maitrisées pour aborder la suite du cours

Exercice 26. Manipulations de base

On considère le type suivant :

```
struct liste {
    int val;            // Valeur stockée
    struct liste *suiv; // Pointeur vers le suivant
};
```

a. Déclarer et initialiser une variable pour avoir une liste vide. Dessiner la liste.

b. Créer une liste avec un seul élément contenant la valeur 17. Dessiner la liste.

c. Ajouter un élément ayant la valeur 23 avant la valeur 17. Dessiner la liste.

d. Ajouter un élément de valeur 42 entre la valeur 23 et la valeur 17. Dessiner la liste.

e. Écrire une fonction **itérative** et une fonction **récursive** qui affiche les éléments d'une liste :

```
void affiche_iter (struct liste *l);
void affiche_rec  (struct liste *l);
```

f. Écrire une fonction **itérative** et une fonction **récursive** qui compte le nombre d'éléments d'une liste :

```
int nb_elem_iter (struct liste *l);
int nb_elem_rec  (struct liste *l);
```

g. Écrire une fonction **itérative** et une fonction **récursive** qui renvoie vrai si une valeur donnée en argument est présente dans la liste et faux sinon :

```
int est_present_iter (struct liste *l, int val);
int est_present_rec  (struct liste *l, int val);
```

Exercice 27. Insertion et suppression au début de la liste

a. Écrire une fonction qui ajoute en début de liste une valeur donnée en argument :

```
struct liste *inserer_deb (struct liste *l, int val);
```

b. Écrire la même fonction que précédemment mais sans utiliser la valeur de retour :

```
void inserer_deb2 (???? l, int val);
```

c. Écrire une fonction qui supprime l'élément en début de liste (pensez à libérer la mémoire) :

```
struct liste *supprimer_deb (struct liste *l);
```

Quelles sont les choix possibles lors de l'implémentation en cas de suppression dans une liste vide ?

d. Écrire la même fonction que précédemment mais sans utiliser la valeur de retour :

```
void supprimer_deb2 (???? l);
```

Exercice 28. Insertion et suppression en fin de liste

a. Écrire une fonction **itérative** et une fonction **récursive** qui ajoute en fin de liste une valeur donnée en argument :

```
struct liste *inserer_fin_iter (struct liste *l, int val);
struct liste *inserer_fin_rec  (struct liste *l, int val);
```

Tester cette fonction sur une liste vide, une liste à un élément et une liste à 10 éléments.

b. Écrire une fonction **itérative** et une fonction **récursive** qui supprime le dernier élément de la list :

```
struct liste *supprimer_fin_iter (struct liste *l);
struct liste *supprimer_fin_rec  (struct liste *l);
```

Tester cette fonction sur une liste vide, une liste à un élément et une liste à 10 éléments.

Exercice 29. Insertion et suppression en milieu de liste

a. Écrire une fonction **itérative** et une fonction **récursive** qui supprime la première occurence d'une valeur donnée en argument :

```
struct liste *supprimer_mil_iter (struct liste *l, int val);
struct liste *supprimer_mil_rec  (struct liste *l, int val);
```

b. Écrire une fonction **itérative** et une fonction **récursive** qui insère une valeur passée en argument dans une liste triée et qui renvoie une liste triée.

```
struct liste *inserer_trie_iter (struct liste *l, int val);
struct liste *inserer_trie_rec  (struct liste *l, int val);
```

Exercice 30. Manipulations de listes

a. Écrire une fonction qui prend en argument une liste et renvoie une liste avec les éléments en ordre inverse

```
struct liste *retourner (struct liste *l);
```

b. Écrire une fonction qui prend en argument deux listes et qui renvoie la concaténation de ces deux listes :

```
struct liste *concat (struct liste *l1, struct liste *l2);
```

c. Écrire une fonction qui prend en argument deux listes dont les éléments sont triés en ordre croissant et et qui renvoie la fusion de ces deux listes qui est une liste triée qui contient tous les éléments des deux listes :

```
struct liste *fusion (struct liste *l1, struct liste *l2);
```

d. Écrire une fonction qui prend en argument une liste triée et qui renvoie la liste avec les doublons supprimés.

```
struct liste *uniq (struct liste *l);
```

e. Écrire une fonction qui échange les deux premiers éléments de la liste s'il y en a au moins deux et qui ne modifie pas la liste sinon :

```
struct liste *echange (struct liste *l);
```

Écrire une fonction qui effectue le tri à bulle sur une liste

```
struct liste *tri_a_bulle (struct liste *l);
```

7 Arbre et expression arithmétique

Exercice 31. Lecture et écriture d'expression arithmétique

Le but est lire une expression arithmétique donnée sur la ligne de commande et de calculer sa valeur L'objectif est de la représenter par un arbre. Les opérateurs sont $+$ et $*$ et les valeurs manipulées sont des flottants.

a. Proposer un type d'arbre :

```
struct Expression_Arithmetique {

};
typedef struct Expression_Arithmetique *EA;
```

qui permet de stocker une expression arithmétique.

b. Soit `s` une chaîne de caractères. On suppose que `s` commence par une parenthèse ouvrante. Écrire une fonction qui renvoie l'indice de la position de la parenthèse fermant la première parenthèse. Attention il peut y avoir d'autres paires de parenthèses dans la chaîne. Si la parenthèse ouvrante n'a pas de correspondance, la fonction renvoie `-1`

c. On suppose pour simplifier que les expressions arithmétiques sont complètement parenthèsées et sans espace. Une chaîne de caractère `s` est une expression arithmétique **valide** si :

— `s` est un float sans espace ni avant ni après
— `s` = `(`S_1`)op(`S_2`)` ou op est un des deux opérateurs et S_1 et S_2 sont des chaînes de expressions arithmétiques **valides**.

Par exemple `((0.1)+(2.23))*(5.0)` est une expression arithmétique **valide**. Écrire une fonction qui prend en entrée une chaîne de caractères représentant une expression arithmétique, et renvoie une variable de type `EA` stockant la même expression arithmétique. Si la chaîne de caractère n'est pas **valide**, cette fonction termine le programme avec un message d'erreur.

d. Écrire une fonction qui affiche le contenu d'un arbre de type `EA`. Comment s'en servir pour écrire la chaîne de caractères **valide** correspondant ?

e. Écrire une fonction de test qui à partir d'une chaîne de caractère `s` la stocke dans une variable de type `EA` puis transforme le contenu de cette variable dans une chaîne `s'` et vérifie si les deux chaînes sont égales.

Exercice 32. Évaluation

Écrire un programme qui calcule la valeur d'une expression arithmétique. L'expression sera en argument de la ligne de commande. L'exécution devra afficher la valeur de l'expression arithmétique.

Tester votre code sur différents exemples :

— `((0.1)+(2.23))*(5.0)`
— `((1.0)+(2.0))+(3.0)`
— `(1.0)+((2.0)+(3.0)`

Exercice 33. Simplifications, améliorations et libération de mémoire

a. Écrire une fonction de simplification qui dans un arbre de type `EA` :

— remplace `0+expr` ou `expr+0` par `expr`
— remplace `1*expr` ou `expr*1` par `expr`
— remplace `0*expr` ou `expr*0` par `0`

Attention ces simplifications doivent être toutes faites. Par exemple `((2.0)*(1))+(0)` doit donner `2.0`.

b. Rajouter les opérateurs $-$, $/$ et $\sqrt{}$.

c. Écrire une fonction qui libère la mémoire utilisée par une variable de type `EA`.

d. Modifier le programme pour qu'il puisse prendre en argument un fichier contenant plusieurs expressions arithmétiques sous forme de chaînes de caractères, une par ligne et affiche la valeur de chacune.

Exercice 34. Génération aléatoire

Écrire un autre programme qui crée un arbre aléatoire complet de hauteur n dont les feuilles ont des valeurs flottantes tirées aléatoirement entre $-m$ et m et les nœuds internes sont un opérateur tiré aléatoirement. Les valeurs n et m sont données en argument au programme et le programme écrit la chaîne de caractère **valide** correspondant à l'arbre.

Exercice 35. Pour aller plus loin

a. Autoriser les espaces dans les chaînes de caractères **valides**.

b. Ajouter l'opérateur `if then else`, les booléens et les comparaisons.

A Révisions du L1

Les exercices de cette section sont faits pour se mettre à jour du niveau de début de ce cours.

Exercice 36. Etoiles

Écrire un programme qui affiche à l'écran 10 étoiles sous la forme suivante :

Exercice 37. Conversions ....

Ecrire un programme qui convertit un temps donné en secondes en heures, minutes et secondes (avec l'accord des pluriels).

Exemple d'exécution :

```
3620 secondes correspond à 1 heure 0 minute 20 secondes
```

Exercice 38. Multiplication Egyptienne

Pour multiplier deux nombres, les anciens égyptiens se servaient uniquement de l'addition, la soustraction, la multiplication par deux et la division par deux. Ils utilisaient le fait que, si X et Y sont deux entiers strictement positifs, alors :

$$X \times Y = \begin{cases} (X/2) \times (2Y) & \text{pour X pair} \\ (X-1) \times Y + Y & \text{pour X impair} \end{cases}$$

Ecrire un programme qui, étant donnée deux nombres (dans l'exemple 23 et 87), effectue la mutliplication égyptienne, en affichant chaque étape de la façon suivante :

```
  23 x 87
= 22 x 87 + 87
= 11 x 174 + 87
= 10 x 174 + 261
= 5 x 348 + 261
= 4 x 348 + 609
= 2 x 696 + 609
= 1 x 1392 + 609
= 2001
```

Exercice 39. Limites ....

Calculez la limite de la suite

$$S_n = \sum_{i=1}^{i=n} \frac{1}{i^2}$$

en sachant que l'on arrête l'exécution lorsque $|S_{n+1} - S_n| < \epsilon$, ϵ étant la précision fixée à l'avance par une constante.

Exercice 40. Nombres premiers

Écrire un programme qui teste si un nombre est premier ou pas.

Exercice 41. Nombres amis

Soit n et m, deux entiers positifs. n et m sont dits *amis* si la somme de tous les diviseurs de n (sauf n lui-même) est égale à m et si la somme de tous les diviseurs de m (sauf m lui-même) est égale à n.

Écrire une fonction qui teste si deux entiers sont des nombres amis ou non.

Écrire une fonction qui, étant donné un entier positif $nmax$ affiche tous les couples de nombres amis (n, m) tels que $n \leq m \leq nmax$.

Aide : 220 et 284 sont amis.

Exercice 42. Racines

Ecrivez un programme qui calcule la racine d'un nombre à une erreur ε fixée par une méthode de dichotomie.

Exercice 43. Les suites de Syracuse

On se propose de construire un petit programme qui permet d'étudier les suites dites de Syracuse :

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

La conjecture de Syracuse dit que quelle que soit la valeur de départ, la suite finit par boucler sur les valeurs 4,2,1,4,2,1,...

a. Construire un programme qui, à partir d'une valeur de départ u_0, affiche les valeurs successives jusqu'à tomber sur la valeur 1 ;

b. modifier le programme pour qu'il compte le nombre d'itérations, sans affichage intermédiaire ;

c. modifier le programme pour qu'il affiche le nombre d'itérations pour toutes les valeurs de départ entre 1 et une valeur fixée.

Exercice 44. Factorielle

Pourquoi la fonction suivante peut donner des résultats faux ?

```
int factorielle (int n)
{
    int f;
    if (n <= 1)
        f = 1;
    else
        f = n * factorielle(--n);
   return f;
}
```

Exercice 45. Calcul de suite

Calculer les valeurs successives de la suite :

$$u_n = \sqrt{1 + \sqrt{2 + \sqrt{\ldots + \sqrt{n}}}}, \text{ pour } 1 \leq n \leq N.$$

Exercice 46. Table ASCII

Le code ASCII permet de coder chaque caractère (imprimable par une machine) à l'aide d'un octet. Ecrire un programme qui affiche le code ASCII. Par exemple, `65` code pour `A`, `48` code pour `0`, etc ....

Exercice 47. Tableau

Dans la suite vous utiliserez un tableau d'entiers comportant `N` cases où `N` est une constante.

a. Ecrire une fonction qui initialise toutes les cases du tableau à 1.

b. Ecrire une fonction qui affiche le produit des éléments d'un tableau.

c. Ecrire une fonction qui retourne le minimum d'un tableau

d. Ecrire une fonction qui effectue un décalage de 1 case à droite de tous les éléments d'un tableau. La case à gauche est affectée à 0. Le dernier élément est supprimé du tableau.

e. Ecrire une fonction qui insère une valeur dans un tableau trié. Après l'insertion, le tableau est toujours trié. Le dernier élément du tableau est supprimé.

f. Ecrire une fonction qui inverse les éléments d'un tableau. Cette inversion s'effectue sur le tableau lui-même (n'utilisez pas de tableau intermédiaire).

g. Ecrire une fonction qui élimine les valeurs en double (ou plus) d'un tableau d'entiers positifs en remplaçant ces valeurs en double par leur valeur négative. La première apparition de la valeur reste inchangée.

h. On suppose que le tableau est découpé en section de nombres significatifs. Chaque section est séparée par un ou plusieurs 0. Ecrire une fonction qui calcule **la moyenne des produits** de chaque section. Par exemple, si l'on dispose du tableau suivant : `1 2 3 0 0 5 4 0 0 8 0 10 11`. Il y a 4 sections. La moyenne des produits correspond à

$$\frac{(1.2.3) + (5.4) + 8 + (10.11)}{4} = 36,0$$

Exercice 48. Tri

a. Remplir un tableau de valeurs aléatoires comprises entre 0 et 99.

b. Calculer le nombre de valeurs différentes dans le tableau.

c. Calculer le tableau d'entiers de taille 10 dont l'élement indicé par i contient le nombre de valeurs aléatoires dont la division par 10 vaut i.