TD/TME Semaine 1
Remise en route, mémoire, tableaux et compilation

Version du 16 septembre 2019

Objectif(s)

- ✯ Remise en route
- ✯ Tableau 1D, malloc et mémoire
- ✯ Lecture simple de fichiers
- ✯ arguments du main (très simples)
- ✯ Compilation séparée et utilitaire make
- ✯ Bonnes pratiques de programmation : les tests
- ✯ Bonnes pratiques de programmation : ddd/gdb et valgrind
- ✯ Bonnes pratiques de programmation : Makefile

TD : Manipulation de tableaux, mémoire, compilation séparée et tris

L'objectif de cette semaine est d'écrire un programme de tri que nous nommerons *sort* c'est-à-dire qui lit un fichier contenant des entiers, les trie par ordre croissant et les écrits dans un autre fichier. Vous prendrez soin de bien tester vos fonctions.

Exercice 1 *(obligatoire)* – Manipulation de tableaux

1. Soit un tableau d'entiers de taille `n` connue. Ecrivez une fonction `void afficher_tab(int tab[], int n)` qui permet d'afficher sur la sortie standard (*i.e.* dans le terminal) un tableau d'entiers.
2. Donnez le prototype et le corps d'une fonction nommée `echanger_elem_tab` permettant d'échanger les valeurs de deux cases d'un tableau. Ecrivez un `main` avec un exemple d'appel de cette fonction et dessinez la mémoire (pile) lors de l'exécution de ce programme.
3. Proposez des tests en utilisant la fonction `assert` permettant de vérifier que cette fonction donne des résultats corrects.
4. Ecrivez la fonction `int *nouveau_tableau(int n)` qui alloue un tableau de `n` entiers et retourne un pointeur vers celui-ci. Modifiez la fonction `main` précédent pour qu'elle alloue dynamiquement le tableau et dessinez à nouveau la mémoire.
5. Ecrivez une fonction `int tab_trie(int tab[], int n)` retournant 1 si le tableau `tab` est bien trié par ordre croissant et 0 sinon. Ecrivez aussi les tests avec `assert`.

Exercice 2 *(obligatoire)* – Lecture de fichier

Les valeurs à trier sont dans un fichier à raison d'une valeur par ligne, sans ligne vide. Comme nous ne savons pas *a priori* le nombre de valeurs nous allons procéder en deux temps : d'abord compter le nombre de lignes (valeurs) dans le fichier puis allouer le tableau et lire les valeurs.

1. Ecrire une fonction `int compter_lignes(char *nomFichier)` qui compte le nombre de lignes du fichier dont le nom est donné en argument.
2. Lors du TME, vous aurez à écrire une fonction `lire_tableau` qui permet de lire les valeurs contenue par le fichier et de retourner à la fois le nombre de valeurs lues ainsi que le tableau que vous aurez alloué pour les stocker. Quel est le prototype de cette fonction ? Quelle fonction de lecture allez vous utiliser ? Décrivez brièvement l'algorithme de votre fonction.

Exercice 3 *(obligatoire)* – Compilation, Makefile

Les fonctions que vous récupérerez ou que vous écrirez lors du TME sont organisées en plusieurs fichiers :

— `utilitaires_tableaux.c` et `.h` contiennent les fonctions de manipulation de tableau (affichage, allocation, lecture de fichier, ecriture...)

— `tri.c` contient la ou les fonctions de tris

— `main_tests.c` et `main_sort.c`, contiennent chacun une fonction `main` permettant soit de tester les fonctions, soit de faire ce qui est demandé pour cette semaine.

Il vous sera demandé de créer deux programmes, un où les tests des fonctions que vous avez écrites sont faits et les résultats sont affichés de manières explicites dans le terminal, et un programme nommé *sort* qui lit un fichier contenant des entiers, les trie par ordre croissant et les écrit dans un autre fichier.

1. Comment allez-vous compiler votre programme de test et votre programme *sort* ? Donnez les commandes de compilation.
2. Supposons à présent qu'à l'exécution vous observez un bug qui vous amène à modifier une de vos fonctions dans `tri.c`, quelles sont les étapes de la compilation que vous devez à nouveau effectuer ? Proposer une méthode pour automatiser ces tâches.

   Pour éviter d'avoir à ré-écrire ces lignes de commandes et ne pas oublier de recompiler après chaque modification d'un fichier, il est pratique d'utiliser l'utilitaire make. Ecrivez le `Makefile` permettant de réaliser les opérations précédentes.

Algorithmes de tri

Il existe de très nombreux algorithmes de tri plus ou moins rapides. Vous en choisirez un parmi ceux ci-dessous que vous ferez tourner à la main sur un petit exemple (exercice ci dessous) et que vous programmerez ensuite en TME. Si vous avez le temps, vous pourrez en programmer plusieurs et comparer leur efficacité.

Exercice 4 *(base)* – Tri par sélection du minimum

Le principe du tri par sélection du minimum est de parcourir plusieurs fois le tableau, de la gauche vers la droite en se décalant à chaque fois d'une case. A chaque parcours, on permute le premier élément lu, avec le plus petit des éléments qui le suivent. Ainsi, après le premier passage, l'élément minimum est en première position du tableau et après k passages, les k premiers éléments sont bien placés.

1. Appliquer ce principe de tri pour les tableaux suivants et montrer leur évolution (pour l'instant avec votre tête et votre crayon uniquement !).

6	5	4	3

3	2	1	6	5	4

2. Réfléchissez maintenant à l'implémentation de votre tri. De quelle(s) fonction(s) aurez vous besoin ? Quels seront leurs prototypes ? Ecrivez le code ou au moins l'algorithme de ce que vous implémenterez.
3. Évaluer, en fonction du nombre d'éléments du tableau, le nombre de tests et d'échanges effectués lors de ce tri.

Exercice 5 *(obligatoire)* – Tri à Bulle

Le principe du tri à bulle est de parcourir le tableau du premier au dernier élément en permutant toutes les paires de deux éléments consécutifs non ordonnés. Ainsi, après le premier passage, l'élément maximum est en dernière position du tableau et après k passages, les k derniers éléments sont bien placés. Lorsque, lors d'un passage, aucune permutation n'est effectuée, le tableau est alors trié, et les passages ultérieurs sont inutiles.

1. Appliquer ce principe de tri pour les tableaux suivants et montrer leur évolution (pour l'instant avec votre tête et votre crayon uniquement !).

6	5	4	3

3	2	1	6	5	4

1	2	3	4

2. Réfléchissez maintenant à l'implémentation de votre tri. De quelle(s) fonction(s) aurez vous besoin ? Quels seront leurs prototypes ? Ecrivez le code ou au moins l'algorithme de ce que vous implémenterez.
3. Évaluer, en fonction du nombre d'éléments du tableau, le nombre de tests et d'échanges effectués lors de ce tri.

Exercice 6 *(entraînement)* – Tri rapide

Le tri rapide (*quicksort*) est un algorithme de type " diviser pour régner ". Il est considéré comme le plus performant dans le cas général pour les ensembles de taille moyenne à grande.

Le principe du tri rapide (*quicksort*) consiste à choisir un élément quelconque de l'ensemble à trier, on nomme cet élément le pivot, puis à partitionner en deux sous-ensembles les éléments restants de part et d'autre du pivot. Le premier sous-ensemble contient tous les éléments inférieurs ou égaux au pivot, le second contient tous les éléments strictement supérieurs au pivot. On applique à nouveau le partitionnement des deux sous-ensembles créés et on continue ainsi récursivement jusqu'à ce que les sous-ensembles créés soient réduits à 1 seul élément. On rassemble enfin le tout dans l'ordre et c'est fini.

1. Appliquer ce principe de tri sur le tableau suivant et montrer son évolution (avec votre tête et votre crayon pour l'instant) :

6	5	4	8	1	3	2	9	7

   Afin de faire le partitionnement, nous allons définir deux nouveaux tableaux. Le premier est destiné à contenir les éléments supérieurs au pivot et le second les éléments inférieurs au pivot [1].

   ATTENTION : un tableau qui a été créé dans une fonction doit être soit retourné soit détruit lorsqu'on quitte la fonction.
2. Réfléchissez maintenant à l'implémentation de votre tri. De quelle(s) fonction(s) aurez vous besoin ? Quels seront leurs prototypes ? Ecrivez le code ou au moins l'algorithme de ce que vous implémenterez.

Exercice 7 *(approfondissement)* – Tri rapide amélioré

Découper un tableau en trois (les éléments inférieurs ou égaux au pivot - le pivot - les éléments strictement supérieurs au pivot) est facile à imaginer, mais sa mise en oeuvre informatique est plus délicate si l'on souhaite éviter de passer par

1. Cette solution est loin d'être la plus efficace car elle nécessite de manipuler la mémoire plus que nécessaire. Dans la suite, nous verrons une version de cet algorithme de tri qui trie le tableau en place.

des tableaux temporaires. En effet, on travaille sur un tableau à une dimension et dans lequel on ne peut faire que des échanges de cases.

L'algorithme vise à partitionner en deux un tableau `tab` de `n` élements. Ce tableau doit avoir au moins deux cases pour être partitionné. On choisit le pivot parmi les éléments du tableau. Ce pivot peut être quelconque mais on espère que sa valeur permettra de fabriquer des partitions de taille équivalente. A l'issue du partitionnement on aura déterminé l'indice p (pivot) de la case contenant le pivot, et on aura :

— 0 <= p < n
— quel que soit k parmi [0..p] tab[k] <= pivot
— quel que soit k parmi [p+1, n[tab[k] > pivot (à condition que p < n-1)

Comme on ne fait aucune hypothèse sur l'ordre des éléments dans le tableau, on choisit arbitrairement comme pivot le premier élément du tableau tab[0]. On parcourt le tableau de gauche à droite (de 1 à n-1).

Soit i l'indice de la case courante et p l'indice du pivot (initialement 0), le traitement de la case i consiste à :

— tester si sa valeur est inférieure ou égale au pivot
  — si oui
    incrementer p
    échanger le contenu de la case p et le contenu de la case i,
  — si non
    ne rien faire

A l'issue du parcours, il reste à placer le pivot à sa place et donc à échanger les cases d'indice 0 et p.

1. Appliquer ce principe de tri sur le tableau suivant et montrer son évolution :

6	5	4	8	1	3	2	7

2. Ecrire une fonction `partitionner(int tab[],int s,int n)` qui, en appliquant l'algorithme décrit, partitionne le tableau `tab` et retourne l'indice définitif du pivot. `s` indique l'indice du premier élément à prendre en compte et `n` est la taille du tableau à considérer.
3. L'algorithme de tri rapide dans son ensemble est une fonction qui appelle la fonction de partitionnement récursivement sur les partitions du tableau, à condition que le tableau initial contienne au moins deux cases. Écrire une fonction récursive `triRapideAmeliore(int tab[],int s, int n)` qui trie en ordre croissant un tableau de `n` éléments en commençant à l'indice `s` selon la méthode du tri rapide.
4. Évaluer, en fonction du nombre d'éléments du tableau, la complexité en temps de cette fonction. L'objectif n'est pas de trouver des valeurs exactes, mais d'estimer cette complexité en fonction de la taille du tableau (à des constantes près).

TME : Implantation d'un tri

Les fichiers sont fournis dans `/Infos/lmd/2019/licence/ue/LU2IN018-2019oct/fournis/Sn` n étant le numéro de la séance. Pensez à les récupérer.

L'objectif de cette ue est notamment de vous apprendre à coder en C proprement et efficacement. Il est donc **obligatoire** de tester chaque fonction que vous écrivez et d'utiliser, lorsque cela est possible, la fonction `assert`, comme vous aviez l'habitude de le faire dans le module `python` de L1 (vous devez pour cela inclure le fichier `assert.h`).

Il vous est aussi demandé d'utiliser régulièrement `valgrind` pour vérifier que votre programme n'a pas de problème mémoire et conseillé d'utiliser `ddd` ou `gdb` pour identifier les *bugs* mémoire. Une présentation rapide de `ddd` est fournie en annexe.

Exercice 8 *(obligatoire)* – Allocation, lecture et écriture de tableaux

La plupart des fonctions que vous avez écrites en TD vous sont données dans le fichier `utilitaires_tableaux.c` et leurs prototypes sont dans `utilitaires_tableaux.h`, mais pas toutes. Dans cet exercice, vous allez les compléter. Vous veillerez à les tester dans le main du fichier `main_tests.c`.

1. Ecrivez la fonction `int *nouveau_tableau(int n)` qui alloue et retourne un pointeur vers un tableau de `n` entiers.
2. Ecrivez la fonction `int * detruire_tableau(int *t)` qui libère la mémoire allouée pour le tableau pointé par `t` et retourne `NULL`.
3. Ecrivez la fonction `int *lire_tableau(char *nomFichier, int *p_nbVal)` qui lit les valeurs entières contenues dans le fichier dont le nom est dans la variable `nomFichier` (une valeur par ligne) et qui retourne un tableau alloué dynamiquement contenant toutes les valeurs lues et le nombre de valeurs lues (dans la variable pointée par `p_nbVal`). Des fichiers exemple vous sont fournis (par exemple `10_valeurs.txt`).
4. Ecrivez la fonction `void ecrire_tableau(char *nomFichier, int *tab, int n)` qui écrit dans un fichier dont le nom est stocké dans `nomFichier` les `n` premières valeurs du tableau `tab`.
5. Ecrivez la fonction `main` dans le fichier `main_tests.c` pour tester les fonctions que vous avez écrites avec `assert` lorsque cela est possible. Comment pourriez vous vérifier que vos fonctions de lecture et d'écriture sont correctes ? Après compilation à l'aide du `Makefile`, votre programme se nommera vous nommerez `tests_tableaux`.

Exercice 9 *(obligatoire)* – Tri

1. Ecrivez la ou les fonctions permettant de trier un tableau d'entiers dans le fichier `tris.c`. Ajoutez le prototype dans le fichier `tris.h`. Testez votre fonction dans le `main` de `main_tests.c`. Vous utiliserez la fonction `assert` pour effectuer des tests complets pour la ou les fonctions que vous aurez écrites.
2. Il est possible de fournir des arguments à votre programme sur la ligne de commande de vos programmes. En effet, la fonction main a deux arguments et son vrai prototype est `int main(int argc, char **argv`. Ainsi, chaque argument de votre ligne de commande est stocké dans la variable `argv` sous la forme d'une chaine de caractères. La variable `argc` contient le nombre d'arguments de la ligne de commande. Il y a au minimum un argument, le nom du programme.
   Lisez la fonction main qui vous est fournie dans le fichier `main_sort.c` puis compilez ce fichier (directement dans le terminal avec gcc) et exécutez le programme. Comprenez-vous ce qu'il affiche ?
3. Modifiez la fonction `main` de `main_sort.c` pour que le programme puisse être appelé ainsi : `sort fichier1 fichier2` et qu'il lise les valeurs contenues dans `fichier1`, les trie et les ecrive dans `fichier2`.

Exercice 10 *(approfondissement)*

Si vous avez terminé, vous pouvez implémenter d'autres algorithmes de tri et comparer leurs performances pour vérifier que votre évaluation de leur complexité est juste.

Annexe : l'environnement de travail

Pour les TME vous aurez besoin de savoir utiliser les trois éléments suivants :
— le terminal et les commandes shell minimales ;
— un éditeur de texte ;
— le compilateur `gcc` et ses options.

Terminal et commandes shell

Le terminal peut être lancé à partir de l'icône représentant un écran noir. Son emplacement varie selon les distributions linux. Elle peut être directement dans la barre du haut, ou accessible depuis le menu contenant les applications. Mac OS X dispose également d'un terminal que vous trouverez dans Applications/Utilitaires. Windows propose une "invite de commande", mais dont les fonctionnalités sont très limitées par rapport au terminal Linux/MacOS X, nous supposerons dans la suite que vous êtes sur Linux ou MacOS X.

Lorsqu'un terminal est ouvert, une ligne est affichée qui ressemble en général à cela :

```
bash$
```

Il s'agit de ce que l'on appelle une invite de commande. Le mot `bash` peut être remplacé par un nom de répertoire ou une autre chaîne de caractère selon votre configuration. Le terminal attend que vous tapiez des commandes shell ou des noms de programmes. Pour cela, il suffit de taper le nom de la commande, suivi éventuellement des arguments que prend la commande (séparés par des espaces) et d'appuyer sur entrée pour en déclencher l'exécution.

Le terminal se trouve initialement dans votre répertoire HOME (le répertoire qui contient tous vos fichiers), mais vous pouvez vous déplacer dans l'arborescence de votre disque dur avec la commande `cd` que nous allons évoquer ci-dessous. Les commandes que vous exécutez dans le terminal sont lancées depuis le répertoire courant dans lequel se trouve le terminal.

Voilà la liste des commandes de base dont vous aurez besoin pour les TP :
— `mkdir nom_du_repertoire` : créé le répertoire `nom_du_repertoire` dans le répertoire courant du terminal. Exemple :
  — `mkdir LU2IN018` : créé le répertoire `LU2IN018` dans le répertoire courant du terminal
  — `mkdir LU2IN018/S1` : créé le répertoire `S1` dans le répertoire `LU2IN018`
  — `mkdir LU2IN018/S2` : créé le répertoire `S2` dans le répertoire `LU2IN018`
— `cd nom_du_repertoire` : change le répertoire courant du terminal. Le nouveau répertoire courant est `ancien_repertoire/nom_du_repertoire` (`ancien_repertoire` est le répertoire dans lequel se trouvait le terminal avant d'appeler cette commande). ATTENTION : pour que cela fonctionne, le répertoire `ancien_repertoire` doit contenir un répertoire nommé `nom_du_repertoire`. Le répertoire '..' correspond au répertoire parent dans l'arborescence des fichiers. Le répertoire '.' correspond au répertoire courant. Ci-après sont donnés quelques exemples si le terminal se trouve dans un répertoire `/home/utilisateur` et en supposant que ces commandes sont exécutées dans cet ordre :
  — `cd LU2IN018` : aller dans le répertoire `/home/utilisateur/LU2IN018`
  — `cd ..` : retourner dans `/home/utilisateur`
  — `cd LU2IN018/S1` : aller dans le répertoire `/home/utilisateur/LU2IN018/S1`
  — `cd ../S2` : aller dans le répertoire `/home/utilisateur/LU2IN018/S2`
  — `cd ../..` : retour dans `/home/utilisateur`
— `ls [repertoire]` : liste le contenu d'un répertoire. Le nom de répertoire donné en argument est optionnel. Si on ne le donne pas, c'est le contenu du répertoire courant qui est listé. Exemples :
  — `ls` : liste le contenu du répertoire courant
  — `ls LU2IN018` : liste le contenu du répertoire `LU2IN018`
  — `ls LU2IN018/S1` : liste le contenu du répertoire `LU2IN018/S1`
— `pwd` : affiche le répertoire courant. Si toutes les commandes ci-dessus ont été exécutées, `pwd` doit afficher `/home/utilisateur`.

— `rm nom_de_fichier` : efface le fichier nommé `nom_de_fichier`. ATTENTION, contrairement à ce qui se passe avec la corbeille, il n'est pas possible d'annuler l'effacement d'un fichier par `rm` ! Cette commande est donc à utiliser avec la plus grande prudence. Il faut ajouter l'option "-r" pour effacer un répertoire.

Lorsque vous aurez créé un exécutable, vous pourrez l'exécuter depuis le terminal en tapant son nom, précédé de "./" [2]. Exemple : si votre programme s'appelle "mon_prog", vous taperez dans le terminal `./mon_prog` .

IMPORTANT : Il est à noter que vous pourrez, depuis le terminal retrouver la documentation associée à une fonction standard du C en tapant [3] :

```
man 3 nom_de_fonction
```

Par exemple, pour obtenir la documentation de `printf` :

```
man 3 printf
```

Editeur

Vous pouvez utiliser l'éditeur que vous souhaitez. Nous vous recommandons néanmoins d'utiliser un éditeur offrant quelques facilités lorsqu'il s'agit de taper un fichier source en C. Certains éditeurs offrent notamment une coloration syntaxique, mettant en évidence les noms de variables, de fonction, etc. Des éditeurs proposent également une indentation automatique et la mise en évidence des parenthèses ou accolades (lorsque vous passez sur une parenthèse ouvrante, par exemple, la parenthèse fermante correspondante clignote). Voilà une liste non exhaustive des éditeurs que vous pouvez utiliser :

— `emacs` (ou `xemacs`) : il s'agit là d'un éditeur extrêmement puissant. Il dispose des fonctionnalités listées ci-dessus et de bien plus encore. De nombreux raccourcis claviers permettent d'effectuer les opérations courantes (ouvrir un fichier, passer d'un fichier ouvert à un autre, etc), mais force est de constater qu'ils sont loin d'être intuitifs. De nombreux tutoriels sont disponibles sur le web, par exemple `http://www.tuteurs.ens.fr/unix/editeurs/emacs.html` ou encore `http://www.linux-france.org/article/appli/emacs/tut/emacs-tut.html` ;

— `gvim` : gvim est une version étendue de vi, ayant des capacités de complétion, coloration syntaxique, indentation automatique ;

— `geany` : il peut également être utilisé comme éditeur.

Le compilateur

`gcc` est le compilateur standard des distributions linux. Il est également disponible sur d'autres systèmes d'exploitation, notamment MacOS et Windows (au travers de Cygwin ou Mingw).

Nous aurons l'occasion de revenir sur l'utilisation de `gcc` et sur ses options. Pour l'instant, tout ce que vous avez besoin de savoir, c'est que pour obtenir un exécutable `mon_prog` depuis un fichier source `mon_prog.c`, il faut taper la ligne suivante dans le terminal, depuis le répertoire contenant `mon_prog.c` :

```
gcc -Wall -o mon_prog mon_prog.c
```

Si votre programme contient plusieurs fichiers sources (`source1.c`, `source2.c`, par exemple), vous pouvez taper la ligne suivante :

```
gcc -Wall -o mon_prog source1.c source2.c
```

Pour des raisons sur lesquelles nous reviendrons plus tard dans cette séance, il est préférable de compiler séparément chaque fichier source en passant par un outil comme `make`. Cependant, s'il s'agit de compiler "à la main" depuis un terminal, cette façon de faire est plus pratique.

2. Le "." désigne le répertoire courant. L'écriture "./" sert à indiquer au terminal que le programme en question se trouve bien dans le répertoire courant. Il n'est pas nécessaire de spécifier l'emplacement des commandes `ls`, `cd` et autres car elles se trouvent dans un répertoire connu du shell et, pour des raisons de sécurité, le shell ne va pas chercher les commandes dans le répertoire courant, sauf si on le précise avec "./".

3. Pour que cela fonctionne, il faut installer les paquetages de pages de manuel (man-pages).

Le debugger symbolique DDD

`ddd` (data display debugger) est un outil permettant de débugger efficacement des programmes écrits en C ou C++ (entre autres). Il s'agit en fait d'une interface permettant d'appeler un debugger non graphique, tel que `gdb`.

Pour pouvoir débugger un programme à l'aide de `ddd`, il est impératif que le code source ait été compilé avec l'option -g. On lance ensuite `ddd` sur le programme `mon_programme` en tapant `ddd mon_programme`. Par exemple,

```
gcc -g -o mon_programme mon_programme.c
```

puis

```
ddd mon_programme
```

On voit alors s'ouvrir la fenêtre de la Figure 1.

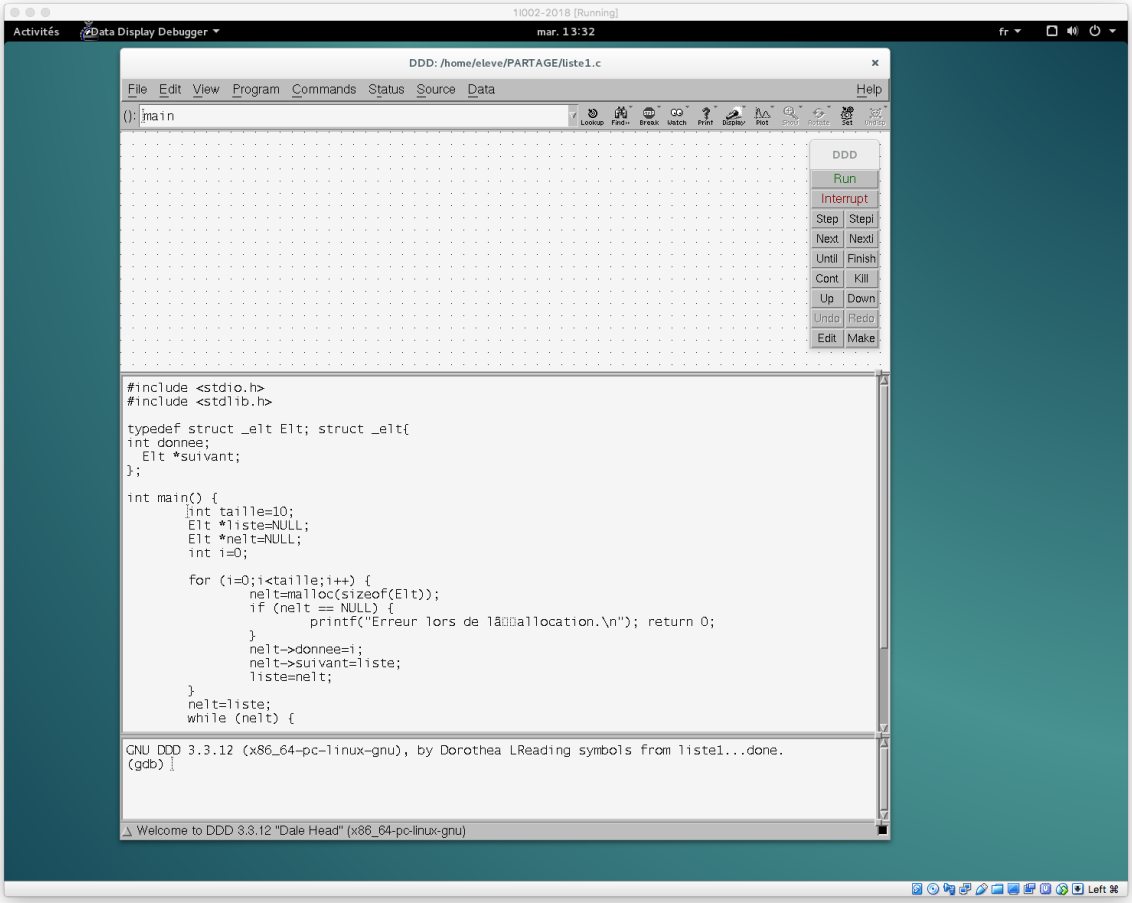

FIGURE 1 – Le debugger `ddd`

— la barre de menus contient des boutons permettant de configurer le déroulement de l'exécution. Lorsque le bouton est accompagné d'un triangle, il permet d'accéder à un sous-menu en maintenant le bouton gauche de la souris enfoncé.

— la fenêtre elle-même est divisée en trois sections : une zone d'affichage (*View* ▷ *Data Window*), la zone contenant le code source (*View* ▷ *Source Window*) et la console (*View* ▷ *GDB Console*) qui permet d'agir sur l'exécution. C'est là par exemple que l'utilisateur saisit des valeurs lorsque le programme en demande. On peut aussi y taper des commandes sous forme textuelle, ou y afficher des résultats.

— la sous-fenêtre DDD (*View* ▷ *Command Tool*) propose des boutons de commande permettant de contrôler l'exécution du programme.

Progression dans l'exécution du programme

Pour exécuter le programme avec `ddd`, il suffit de cliquer sur `Run`. Cette opération exécute le programme d'une traite : vous n'avez rien vu passer ! On définit donc des *points d'arrêt* (voir ci-dessous) dans le programme. `Run` exécute alors le programme jusqu'au premier point d'arrêt. On peut ensuite poursuivre l'exécution de plusieurs manières :

Cont : poursuit l'exécution jusqu'au prochain point d'arrêt.

Next : exécute l'instruction suivante. Si cette instruction est un appel de fonction, la fonction est exécutée d'une traite (i.e., le debugger ne déroule pas l'exécution de la fonction pas à pas.

Step : exécute l'instruction suivante. Si cette instruction est un appel de fonction, on entre dans la fonction et on s'arrête à la première instruction de celle-ci. On peut à tout instant terminer l'exécution de la fonction en cliquant sur Finish.

NB : `Step` est incompatible avec l'exécution des fonctions de bibliothèque, type `printf` ou `malloc`, dont on ne connaît pas le code source.

Until : exécute le programme jusqu'à atteindre une instruction dont le numéro de ligne est strictement supérieur à celui de l'instruction courante. Cette commande permet en particulier de terminer l'exécution d'une boucle.

Insertion et destruction de points d'arrêt

L'insertion d'un point d'arrêt se fait avec le bouton `Break` après avoir placé le curseur au début de la ligne où l'on veut s'arrêter. Un stop s'affiche au début de la ligne. L'exécution du programme sera interrompue avant chaque exécution de l'instruction figurant sur cette ligne. En plaçant le curseur au niveau d'un point d'arrêt existant, on remarque que le bouton `Break` du menu se transforme en `Clear`. On peut alors détruire le point d'arrêt ou, en accédant au sous-menu, le désactiver ou modifier ses caractéristiques. En particulier, on peut rendre un point d'arrêt temporaire : le programme ne sera interrompu que lors de son premier (ou prochain) passage à cet endroit. Le menu *Source* ▷ *Breakpoints* permet d'accéder à la liste des points d'arrêt du programme, ce qui se révèle utile lorsqu'on veut en modifier plusieurs.

Visualisation de la valeur des variables

Il existe plusieurs manières de visualiser la valeur d'une variable. La plus simple consiste à amener le curseur dessus : au bout d'environ 1s, la valeur de la variable s'affiche. Un clic droit sur une variable permet aussi d'accéder à un menu d'affichage, en particulier aux fonctions :

Print : afffiche la valeur de la variable dans la fenêtre *GDB Console*. Si cette variable est un pointeur, la valeur affichée est une adresse. Il faut utiliser `Print*` pour accéder à la valeur de la variable pointée, c'est-à-dire au contenu de la case mémoire.

Display : affiche dans la fenêtre *Data Window* l'évolution de la valeur de la variable, mise à jour au fur et à mesure des modifications. Comme pour `Print`, `Display*` permet d'accéder à l'objet pointé lorsque la variable est un pointeur. L'affichage peut être supprimé au moyen du bouton `Undisp` (ou par un clic droit sur l'affichage).

Visualisation d'un tableau :

Il est possible de visualiser directement le contenu d'un tableau, en sélectionnant `Display` après un clic droit sur le nom du tableau. On peut passer d'un affichage vertical à horizontal (ou l'inverse), en sélectionnant `Rotate` au moyen d'un clic droit sur l'affichage du tableau.

Visualisation d'une liste chaînée :

Lorsqu'un pointeur est affiché (`Display`), on peut sélectionner ce pointeur dans la zone d'affichage et cliquer sur `Disp*`. La valeur de l'objet pointé s'afffiche alors, liée au pointeur par une flèche. Lorsque différents pointeurs référencent une même case mémoire, la "factorisation" n'est pas automatique. Le menu *Data* ▷ *Detect Aliases* permet de ramener les différentes flèches sur le même objet.

Exécution d'un programme avec arguments

L'exécution de certains programmes nécessite des arguments passés par la ligne de commande. Ces paramètres ne peuvent pas être transmis lors de l'appel à `ddd`. Ils sont transmis lors du lancement de l'exécution en accédant au menu *Program* ▷ *Run* et en entrant l'ensemble des paramètres (séparés par un espace) dans la case *Run with Arguments*.

TD/TME Semaine 2
Listes chaînées et simulation d'un écosystème

Version du 16 septembre 2019

Objectif(s)

- ✯ Structures
- ✯ Gestion de la mémoire
- ✯ Passage de paramètres et pointeurs
- ✯ tableaux 2D statiques (un peu)
- ✯ Listes chaînées
- ✯ Makefile item Bonnes pratiques de programmation : les tests
- ✯ Bonnes pratiques de programmation : ddd/gdb et valgrind
- ✯ Bonnes pratiques de programmation : Makefile

L'objectif de ce sujet est la réalisation d'un petit projet de programmation d'un écosystème virtuel utilisant des listes chaînées. Nous allons d'abord commencer par un petit exercice de rappels sur les listes puis aborderons le problème de l'écosystème.

TD Rappels sur les listes chaînées

L'objectif de cette première partie est de prendre en main le concept de listes chaînées au travers de quelques exercices pédagogiques.

Soit la structure de données suivante :

```
typedef struct _elt Elt;
struct _elt{
  int donnee;
  Elt *suivant;
};
```

Exercice 1 *(base)* – Analyse d'un exemple

Soit le programme suivant :

```
int main() {
  int taille=10;
  Elt *liste=NULL;
  Elt *nelt=NULL;
  int i=0;
```

```
  for (i=0;i<taille;i++) {
    nelt=malloc(sizeof(Elt));
    if (nelt == NULL) {
      printf("Erreur lors de l'allocation.\n");
      return 0;
    }
    nelt->donnee=i;
    nelt->suivant=liste;
    liste=nelt;
  }

  nelt=liste;
  while (nelt) {
    printf("%d ",nelt->donnee);
    nelt=nelt->suivant;
  }
  printf("\n");

  return 0;
}
```

1. Que fait ce programme ? Qu'est-ce qui est affiché à l'écran ?
2. Quelle est la place mémoire occupée par la liste chaînée créée dans ce programme ? Quelle taille ferait un tableau contenant les mêmes données ?
3. La mémoire allouée pour cette liste n'a pas été libérée. Ajoutez des instructions permettant de libérer toute la mémoire qui a été allouée.

TD Ecosystème - Mise en place

L'objet de cette partie du sujet est d'écrire des fonctions de manipulation de listes chaînées et de les utiliser pour programmer une simulation simple d'écosystème.

Le modèle d'écosystème que vous allez programmer n'a aucune prétention à être réaliste, mais il permet de se familiariser avec le concept d'équilibre, primordial dans un écosystème. L'évolution de votre écosystème dépendra de quelques variables que vous pourrez modifier pour observer l'impact qu'elles peuvent avoir. Cet écosystème contiendra deux types d'entités virtuelles : des proies et des prédateurs, susceptibles de manger ces dernières.

Notre écosystème est un monde discret (un tore, que nous afficherons comme un rectangle) contenant un certain nombre de cases, identifiées par leurs coordonnées (entières) x et y. Chaque proie (et chaque prédateur) est dans une case donnée et peut se déplacer. A un instant donné, une case peut contenir plusieurs proies et plusieurs prédateurs.

```
Nb proies (*):         20
Nb predateurs (O):     20
@ = * et O
+----------+
| *        |
|* *O * **O|
| O*O      |
|  O  O    |
|  @  O  O |
|  **    *O|
|     @ O @|
|    *   O@|
|  O *O    |
|  O O @*  |
+----------+
```

La simulation de notre écosystème repose sur plusieurs structures de données et sur des fonctions que vous allez écrire dans la suite de cette séance. Les données utilisées pour la simulation sont une liste chaînée contenant les proies et une autre liste chaînée contenant les prédateurs.

Exercice 2 *(obligatoire)* – Structure de données

Les proies et les prédateurs seront représentés par une même structure de données contenant les coordonnées entières `x` et `y`, un nombre réel `energie` (tel qu'un animal dont l'énergie tombe en dessous de 0 est mort, ce point sera géré plus tard). Il contiendra ensuite un tableau `dir` de deux entiers qui représentera sa direction (nous y reviendrons également plus tard). Comme nous souhaitons stocker des variables de ce type dans des listes simplement chaînées, la structure contiendra enfin un pointeur nommé `suivant` sur cette même structure.

Ces déclarations seront réalisées dans `ecosys.h`, de même que les prototypes des fonctions que vous écrirez par la suite. Les fonctions elles-mêmes seront écrites dans un fichier `ecosys.c`.

1. Ecrivez la déclaration de cette structure, à laquelle vous donnerez, via un `typedef` le nom équivalent `Animal`.
2. Ecrivez la fonction de création d'un élément de type `Animal` (allocation dynamique). Vous initialiserez les champs `x`, `y`, et `energie` à partir des arguments de la fonction. Les cases du tableau `dir` seront initialisées aléatoirement avec les valeurs -1, 0 ou 1.

   La fonction aura le prototype suivant :

   ```
   Animal *creer_animal(int x, int y, float energie);
   ```

3. Ecrivez la fonction d'ajout en tête dans la liste chaînée.

   La fonction aura le prototype suivant :

   ```
   Animal *ajouter_en_tete_animal(Animal *liste, Animal *animal);
   ```

4. Ecrivez la fonction permettant d'enlever un élément de la liste chaînée et de libérer la mémoire associée. Comme précédemment, la liste sera passée par adresse.

   La fonction aura le prototype suivant :

   ```
   void enlever_animal(Animal **liste, Animal *animal);
   ```

5. Ecrivez des fonctions permettant de compter le nombre d'éléments contenus dans une liste chaînée. Vous écrirez une fonction itérative et une fonction récursive.

   Les fonctions auront les prototypes suivants :

   ```
   unsigned int compte_animal_rec(Animal *la);
   unsigned int compte_animal_it(Animal *la);
   ```

Exercice 3 *(obligatoire)* – Affichage et main

1. Ecrivez une fonction d'affichage de votre ecosystème. Cette fonction affichera le contenu des différentes cases de votre monde simulé. Vous afficherez un espace pour les cases vides, un '*' pour les cases contenant au moins une proie, un 'O' pour les cases contenant au moins un prédateur et si une case contient des proies et des prédateurs, vous afficherez un '@'.

Exercice 4 *(obligatoire)* – Organisation du programme et main

Le programme est organisé en trois fichiers :

— `main_tests.c`, qui contiendra un main avec les tests de vos fonctions,

— `main_ecosys.c`, qui contiendra un main permettant de simuler un écosystème,

— `ecosys.c` et `ecosys.h` qui contiennent toutes les autres fonctions de manipulation de listes et de structures.

1. Ecrivez le `Makefile` permettant de compiler les programmes `tests_ecosys` et `ecosys`

TME : Ecosystème - Simulation

Cette partie est à faire en TME. Les fonctions de la partie précédente, corrigées en TD, sont fournies.

Exercice 5 *(obligatoire)* – Tests des fonctions et main

1. Ecrivez la fonction permettant d'ajouter un animal à la position `x`, `y`. Cet animal sera ajouté à la liste chaînée `liste_animal`. Plutôt que de renvoyer l'adresse du premier élément de la liste, vous passerez ce pointeur par adresse de façon à pouvoir le modifier directement dans la fonction : l'argument sera donc un pointeur sur liste chaînée, donc un pointeur sur un pointeur sur un Animal... La fonction ne fera rien si les coordonnées données sont incorrectes (négatives ou supérieures ou égales à `SIZE_X` ou `SIZE_Y` qui sont des étiquettes définies par ailleurs). Le niveau d'énergie initial de l'animal créé sera égal à une variable globale `energie`, déclarée dans le fichier contenant votre fonction main.

   La fonction aura le prototype suivant :

   ```
   void ajouter_animal(int x, int y, Animal **liste_animal);
   ```

2. Ecrivez la fonction `liberer_liste_animaux(Animal *liste)` qui permet de libérer la mémoire allouée pour la liste `liste`.
3. Ecrivez une fonction main qui va créer quelques proies et quelques prédateurs à des positions variées, vérifiez leur nombre en faisant appel aux fonctions de comptage que vous avez définies et enfin affichez l'état de votre écosystème puis libérez la mémoire allouée pour ces listes. Vous écrirez cette fonction dans le fichier `main_tests.c`. Vous complèterez aussi le `Makefile` pour créer le programme `tests_ecosys`.

Notre écosystème fonctionne en temps discret. A chaque pas de temps un certain nombre d'opérations devront être réalisées :

— toutes les proies se déplacent, leur énergie est décrémentée d'un montant `d_proie` ;
— les proies sont susceptibles de se reproduire avec une probabilité `p_reproduce` ;
— tous les prédateurs se déplacent, leur énergie est décrémentée d'un montant `d_predateur` ;
— les prédateurs qui sont sur la même case qu'une proie ont une probabilité `p_manger` de les dévorer. Dans ce cas la proie meurt et le prédateur augmente son énergie d'un montant valant l'énergie de la proie ;
— les prédateurs sont susceptibles de se reproduire avec une probabilité `p_reproduce`.

Les différentes probabilités évoquées ci-dessus seront des variables globales que vous déclarerez dans `ecosys.c`. Vous les déclarerez en tant qu'`extern` dans `ecosys.h`.

Nous allons à présent détailler les différentes fonctions permettant de programmer cette simulation.

Exercice 6 *(obligatoire)* – Déplacement et reproduction

Les mouvements seront gérés de la façon suivante : chaque proie (ou prédateur) dispose d'une direction indiquée dans le champs `dir`, qui est un tableau de deux entiers. Il indique la direction suivie sous la forme de deux entiers valant -1, 0 ou 1. Ces valeurs indiquent de combien les coordonnées de la proie doivent être décalées. Considérant que la première case de la prairie est en haut à gauche, une direction de $(1, -1)$ correspond, par exemple, à un mouvement vers la case en haut et à droite, une direction de $(0, 0)$ correspond à une proie immobile.
Un changement de direction s'opèrera avec une probabilité `p_ch_dir`, autrement dit si un nombre aléatoire compris entre 0 et 1 est inférieur à cette valeur. Pour obtenir ce nombre aléatoire, il faut utiliser la fonction `rand` qui renvoie un entier et le diviser par la valeur maximum, à savoir `RAND_MAX`. Rappel : toutes les "probabilités" seront déclarées sous forme de variables globales dans le fichier contenant la fonction main.

1. Ecrivez une fonction permettant de faire bouger les animaux contenus dans la liste chaînée passée en argument. Le déplacement de la proie se fera dans la direction indiquée par le champs `dir`. Le monde sera supposé torique, c'est-à-dire que si la proie essaie d'aller en haut alors qu'elle est sur la première ligne, elle se retrouvera automatiquement sur la dernière ligne. De même si une proie essaie d'aller à droite alors qu'elle est sur la dernière colonne de droite, elle réapparait sur la même ligne et sur la colonne la plus à gauche.

   La fonction aura le prototype suivant :

   ```
   void bouger_animaux(Animal *la);
   ```

2. Ecrivez une fonction permettant de gérer la reproduction des animaux. Vous parcourrez la liste passée en argument et, pour un animal `ani`, vous ajouterez un nouvel animal à la même position qu'`ani` avec une probabilité `p_reproduce`.

   Remarque : l'ajout se fera en tête, un animal qui vient de naître ne sera pas considéré par votre boucle et ne se reproduira donc pas.

   La fonction aura le prototype suivant :

   ```
   void reproduce(Animal **liste_animal);
   ```

3. Testez vos fonctions dans le main de tests en ne créeant qu'un animal à une position que vous aurez définie et en la déplaçant dans une direction que vous aurez aussi définie. Vérifiez bien la toricité du monde. Testez aussi votre fonction de reproduction.

Exercice 7 *(obligatoire)* – Simulation des proies

1. Écrivez une fonction de mise à jour des proies. Cette fonction devra :
   — faire bouger les proies en appelant la fonction `bouger_animaux` ;
   — parcourir la liste de proies et baisser leur énergie de `d_proie`, les proies dont l'énergie est inférieure à 0 seront supprimées ;
   — faire appel à la fonction de reproduction.

   La fonction aura le prototype suivant :

   ```
   void rafraichir_proies(Animal **liste_proie);
   ```

2. Ecrivez une nouvelle fonction main dans le fichier `main_ecosys.c`. Vous y ferez une boucle infinie pendant laquelle vous mettrez à jour les proies et afficherez l'écosystème résultant.

   Pour que vous ayez le temps de voir l'état de votre écosystème, vous pourrez ajouter des pauses en utilisant la fonction `usleep`.

   ```
   man 3 usleep
   ```

   pour avoir une description de son fonctionnement et de la façon de l'utiliser.

Exercice 8 *(entraînement)* – Gestion des prédateurs

1. Pour gérer les prédateurs, nous aurons besoin d'une fonction qui va vérifier s'il y a une proie sur une case donnée. Ecrivez cette fonction qui aura le prototype suivant :

   ```
   Animal *animal_en_XY(Animal *l, int x, int y);
   ```

   `l` est la liste chaînée des proies et la valeur renvoyée est un pointeur sur une proie dont les coordonnées sont `x` et `y` (la première rencontrée) ou `NULL` sinon.

2. Les prédateurs sont gérés comme les proies. Comme ils utilisent la même structure, les fonctions `creer_animal`, `ajouter_en_tete_animal`, etc. peuvent être réutilisées telles quelles. Écrivez la fonction de mise à jour de prédateurs qui respectera le prototype suivant :

```
void rafraichir_predateurs(Animal **liste_predateur, Animal **liste_proie);
```

Cette fonction est directement inspirée de la fonction de rafraichissement des proies. Vous pourrez copier-coller celle-ci et faire des modifications. Vous devrez notamment baisser l'énergie d'un prédateur de `d_predateur` et faire en sorte que, s'il y a une proie située sur la même case qu'un prédateur, elle soit "mangée" avec une probabilité `p_manger`.

3. Modifier votre fonction main (`main_ecosys.c`) pour voir évoluer également les prédateurs.

   Vous pouvez à présent observer l'évolution de votre petit écosystème et notamment tester différentes valeurs des constantes pour étudier l'impact que cela peut avoir sur votre écosystème.

SCIENCES SORBONNE UNIVERSITÉ

TD/TME Semaine 3
Bibliothèque générique de liste

Version du 16 septembre 2019

Objectif(s)

- ✯ Comprendre les pointeurs génériques et les pointeurs de fonction
- ✯ Savoir écrire une bibliothèque générique à partir de code existant
- ✯ Savoir utiliser une bibliothèque générique

Exercice(s)

TD : Conception d'une bibliothèque de liste chaînée générique

Exercice 1 *(base)* – Structures de données

Les principales fonctions de manipulation de liste vues jusqu'à présent sont les suivantes : `inserer_debut`, `inserer_fin`, `inserer_place`, `chercher`, `detruire_liste`, `afficher_liste`, `ecrire_liste` et `lire_liste`.

1. Rappelez ce que font ces différentes fonctions et identifiez celles qui ont besoin de connaitre la donnée portée par un élément de la liste.
2. Déduisez-en les fonctions de manipulation de donnée nécessaires auxquelles les fonctions évoquées ci-dessus pourront faire appel. Proposez-en des prototypes génériques, permettant de s'adapter à n'importe quel type de donnée.
3. Définissez le type décrivant un élément de la liste.

   Les fonctions de manipulation des données peuvent être définies de façon unique dans un programme. Cette solution a l'inconvénient d'empêcher d'utiliser, dans un même programme, notre structure de liste pour des données de type variées. Il ne serait ainsi pas possible de manipuler des listes d'entiers et des listes de chaînes de caractères en même temps, ce qui limiterait notre bibliothèque de liste.
4. Comment pourrait-on procéder pour pouvoir utiliser, dans un même programme, des listes chaînées de types divers ?
5. Proposez la structure de données `Liste` correspondante (`PListe` étant définie comme un pointeur sur `Liste`).

Exercice 2 *(base)* – Implémentation des fonctions de manipulation de la liste

Vous allez maintenant implémenter les fonctions de manipulation de liste. Vous prendrez soin de n'utiliser que les fonctions de manipulation des données évoquées précédemment.

1. Ecrivez la fonction d'insertion en début de liste. Prototype :

   ```
   void inserer_debut(PListe pliste, void *data);
   ```

   La fonction devra dupliquer la donnée passée en argument.

2. Ecrivez la fonction d'insertion en fin de liste. Prototype :
```
void inserer_fin(PListe pliste, void *data);
```
La fonction devra dupliquer la donnée passée en argument.
3. Ecrivez la fonction d'insertion en place. Prototype :
```
void inserer_place(PListe pliste, void *data);
```
La fonction devra dupliquer la donnée passée en argument. Elle ne fera rien si la donnée est déjà dans la liste.
4. Ecrivez la fonction de recherche dans la liste. Prototype :
```
PElement chercher_liste(PListe pliste, void *data);
```
5. Ecrivez la fonction de destruction de la liste. Prototype :
```
void detruire_liste(PListe pliste);
```
6. Ecrivez la fonction d'affichage de la liste. Prototype :
```
void afficher_liste(PListe pliste);
```
La fonction ira à la ligne suivante après chaque donnée.
7. Ecrivez une fonction permettant d'ajouter un nombre quelconque de données en utilisant le principe des fonctions variadiques. Prototype :
```
void ajouter_liste(PListe pliste, int nb_data, ...);
```
`pliste` est la liste à compléter, `nb_data` est le nombre de données à ajouter (c'est le nombre d'arguments après celui-ci).
8. Ecrivez une fonction permettant d'appliquer une même fonction à chaque élément de la liste. Prototype :
```
void map(PListe pliste, void (*fonction)(void *data, void *oa), void *optarg);
```
`pliste` est la liste à considérer, `fonction` est la fonction à appliquer et `optarg` est un argument supplémentaire qui peut être utilisé pour transmettre des informations supplémentaires ou récupérer un résultat de traitement.
Nous allons à présent gérer la lecture et l'écriture dans des fichiers. Pour pouvoir lire une liste, il faudra, à un moment ou à un autre, connaître le type des données qu'elle contient. Nous pouvons soit définir un code associé à chaque donnée, mais cela nécessite de disposer d'une table de tous les codes possibles, ce qui n'est pas facile à maintenir. L'alternative que nous choisirons ici s'appuie sur l'utilisateur et sur le fait qu'il connait à l'avance le type des données qu'il va lire. Lors de l'écriture, il ne sera donc pas nécessaire d'ajouter de code indiquant le type des données, par contre, lors de la lecture, il faudra transmettre une liste qui ne contiendra aucun élément, mais permettra de trouver les fonctions appropriées pour manipuler les données.
9. Ecrivez la fonction d'écriture de la liste. Prototype :
```
int ecrire_liste(PListe pliste, const char *nom_fichier);
```
La fonction renverra 0 s'il y a une erreur ou 1 si tout s'est bien passé.
10. Ecrivez la fonction de lecture de la liste. Prototype :
```
int lire_liste(PListe pliste, const char * nom_fichier);
```
La fonction renverra 0 s'il y a une erreur ou 1 si tout s'est bien passé.

Exercice 3 *(obligatoire)* – Fonctions de manipulation d'entiers

1. Ecrivez les fonctions de manipulations d'entiers. Prototypes :
```
void *dupliquer_int(const void *src);
void copier_int(const void *src, void *dst);
void detruire_int(void *data);
void afficher_int(const void *data);
int comparer_int(const void *a, const void *b);
int ecrire_int(const void *data, FILE *f);
void * lire_int(FILE *);
```

TME : Utilisation de la bibliothèque

Récupérez les fichiers fournis pour cette séance. Ils contiennent les implémentations des fonctions vues en TD.

Exercice 4 *(obligatoire)* – Test de la bibliothèque avec des entiers

1. Ecrivez une fonction main (fichier `ex_liste_entiers.c`) pour tester la bibliothèque de liste avec des données entières. Vous prendrez soin d'utiliser chacune des fonctions de manipulation de la liste et de faire en sorte de vérifier le résultat de l'appel.

Exercice 5 *(obligatoire)* – Listes de chaînes de caractères

1. Ecrivez les fonctions de manipulations de chaines de caractères (fichier `fonctions_string.c`). Prototypes :

```
void *dupliquer_str(const void *src);
void copier_str(const void *src, void *dst);
void detruire_str(void *data);
void afficher_str(const void *data);
int comparer_str(const void *a, const void *b);
int ecrire_str(const void *data, FILE *f);
void * lire_str(FILE *);
```

2. Ecrivez une fonction main pour tester la bibliothèque de liste avec des données de type chaînes de caractères (fichier `ex_liste_string.c`). Vous prendrez soin d'utiliser chacune des fonctions de manipulation de la liste et de faire en sorte de vérifier le résultat de l'appel.

Exercice 6 *(entraînement)* – Manipulation d'un dictionnaire

Nous allons maintenant écrire une fonction permettant de mesurer la longueur des mots contenus dans un dictionnaire. Pour cela, nous allons écrire les fonctions de manipulation d'une donnée composée de deux entiers. Nous nous en servirons pour stocker le résultat de notre comptage : le premier entier sera la longueur du mot et le second le nombre de mots de cette longueur dans le dictionnaire.

1. Ecrivez les fonctions de manipulations de cette donnée avec 2 entiers (fichier `fonctions_2entiers.h`). Structure et prototypes :

```
typedef struct double_int {
  int a;
  int b;
} Double_int;


void *dupliquer_2int(const void *src);
void copier_2int(const void *src, void *dst);
void detruire_2int(void *data);
void afficher_2int(const void *data);
int comparer_2int(const void *a, const void *b);
int ecrire_2int(const void *data, FILE *f);
void * lire_2int(FILE *);
```

La donnée sera de type `Double_int`.

L'affichage se fera sous la forme suivante :

```
a=12 b=42
```

La comparaison ne s'appuiera que sur le champ `a`.

2. Pour compter le nombre de mots, nous allons nous appuyer sur la fonction `map`. Nous devons donc écrire la fonction de traitement d'un mot de la liste chaînée. Cette fonction, qui gèrera le comptage, va recevoir en argument un `char *` sous la forme d'un pointeur générique `void *`. L'argument optionnel va nous servir à stocker le résultat sous la forme d'une liste de double entiers (la longueur et le nombre de mots de cette longueur). Cette liste sera créée avant de faire appel à la fonction `map`. La fonction de traitement d'un mot mesurera la longueur du mot transmis et cherchera si cette longueur fait partie de la liste de résultats actuels, si oui, le nombre d'éléments de cette taille sera incrémenté, sinon, on ajoutera cette longueur avec la valeur 1 dans la liste (elle sera ajoutée en place).

   Ecrivez la fonction de comptage et la fonction main permettant de charger un dictionnaire (fichier `french_za` fourni), de calculer le nombre de mots avec leur longueur, d'afficher le résultat et de libérer la mémoire (fichier `compte_mots.c`).

