Programmation
TP n° 3 : Nano Nano

I) Introduction

Nano est un éditeur de texte en ligne de commande très populaire. Bien qu'il soit bien plus simple que des outils modernes tels que **Notepad++** ou **Visual Studio Code**, la mécanique qui se cache derrière son fonctionnement reste particulièrement intéressante à comprendre.

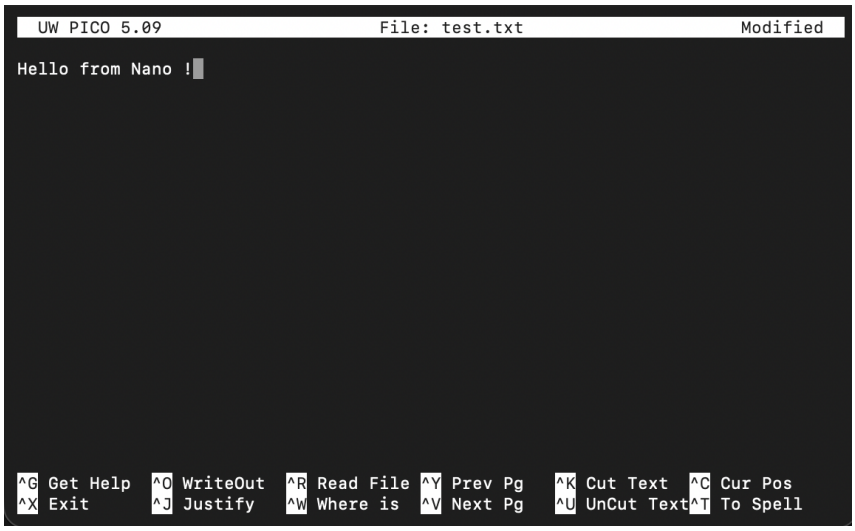

FIGURE 1 – L'éditeur de texte Nano.

Dans ce TP, nous allons recréer un **mini-éditeur de texte inspiré de Nano**, doté d'un ensemble restreint mais essentiel de fonctionnalités :

- **Lecture** du contenu d'un fichier texte dans une fenêtre dédiée ;
- **Modification** du texte affiché ;
- **Enregistrement** des changements sur le disque ;
- **Fermeture** propre de la fenêtre de l'éditeur.

Bien que ces fonctionnalités puissent sembler limitées, leur implémentation représente déjà une charge de travail conséquente. Imaginez la complexité supplémentaire qu'impliquerait l'ajout de fonctionnalités comme la **recherche de texte**, le **copier/coller** ou encore la **coloration syntaxique** !
Pour vous accompagner, un **squelette de code** est fourni, structuré pour vous guider pas à pas dans la réalisation.
Ce TP vous permettra de mobiliser et d'approfondir plusieurs compétences :

- Lecture et compréhension de code ;
- Manipulation de fichiers (lecture/écriture sur le disque) ;
- Manipuation des chaines de caractères ;
- Conception modulaire d'un programme en langage C ;
- Gestion de la mémoire dynamique ;
- Découverte de la bibliothèque **ncurses**, utilisée pour créer des interfaces en mode texte.

II) Lecture de code

Commencez par télécharger le squelette du code sur le dépôt GitHub suivant : https://github.com/augustin-laouar/teaching/tree/main/ENS_AGREG_PROG/TP/nano-nano

Exercice 1 : Compréhension de la structure du programme

L'objectif de cet exercice est d'acquérir une vision d'ensemble du fonctionnement du code et des principales structures de données utilisées. Pour vous guider, les fichiers d'en-tête ont été documentés selon le format **Doxygen**. Si vous le souhaitez, vous pouvez générer la documentation pour faciliter votre lecture.

1. Ouvrez le fichier `struct.h` et analysez les différentes structures de données qui y sont déclarées. Ces structures serviront à représenter le contenu du fichier texte en mémoire, ainsi que l'état du curseur et de l'éditeur. Essayez de réaliser un schéma illustrant :
   - la manière dont un fichier texte est représenté en mémoire ;
   - la façon dont la position du curseur est gérée dans ce modèle.
2. Parcourez ensuite les autres fichiers d'en-tête et identifiez le rôle de chacun d'eux dans l'organisation du code. Vous pouvez, si vous le souhaitez, réaliser un second schéma permettant de visualiser la structure globale du code et les relations entre les différents modules.

III) Lecture d'un fichier

Nous allons implémenter les fonctions de bases dont nous avons besoin pour lire un fichier. On ne s'interesse pas à la modification ni à l'enregistrement pour le moment.

Exercice 2 : Buffer

1. Implémentez les fonctions permettant d'initialisé les objets `buffer` et `line`.
2. Implémentez les fontions permettant de libérer la mémoire de ces objets.
3. Implémentez la fonction `line_reserve` qui garantit que la ligne dispose d'au moins `need` octets de mémoire allouée. Si la capacité actuelle est insuffisante, la fonction devra agrandir dynamiquement le tableau data (en doublant sa taille ou, si ce n'est pas assez, en atteignant directement la taille demandée) à l'aide de `realloc()`.
4. Implémentez la fonction `buffer_reserve` qui garantit que le tableau de lignes du buffer dispose d'une capacité suffisante pour contenir au moins `need` lines. Si la capacité actuelle est insuffisante, la fonction devra agrandir dynamiquement le tableau (en doublant sa taille ou en atteignant directement la taille demandée) à l'aide de `realloc()`, puis initialiser les nouvelles lignes créées avec `line_init`.
5. Implémentez la fonction `buffer_insert_line` qui insère une nouvelle ligne dans le buffer à la position `idx`.

   Si la capacité actuelle est insuffisante, commencez par appeler `buffer_reserve` afin d'agrandir dynamiquement le tableau de lignes.

   Les lignes existantes à partir de `idx` devront être décalées d'une case vers le bas pour libérer la place de la nouvelle ligne.

   Attention : pour cette opération, utilisez `memmove()` (et non `memcpy()`), car les zones mémoire se chevauchent : le tableau `B->lines` est déplacé vers lui-même. `memmove()` garantit une copie sûre même en cas de recouvrement, alors que `memcpy()` provoquerait un comportement indéfini.

   Ensuite, initialisez la nouvelle ligne avec `line_init`. Si une chaîne `s` est fournie, allouez la mémoire nécessaire avec `line_reserve` puis copiez son contenu dans la nouvelle ligne à l'aide de `memcpy` (cette fois sans risque, car les zones mémoire sont distinctes).

   Enfin, incrémentez le compteur `count` du buffer.

À propos du caractère nul (`\0`) : en C, une chaîne de caractères est un tableau de `char` terminé par un octet nul, qui marque la fin de la chaîne. Ce caractère de terminaison **n'est pas compté** dans la longueur logique de la chaîne : si une ligne contient 5 caractères visibles, sa longueur `len` vaut 5, mais il faut tout de même prévoir un sixième octet pour stocker le marqueur `\0`.
Les fonctions de la bibliothèque standard (p. ex. `strlen`, `printf`, `strcmp`) s'arrêtent lorsqu'elles rencontrent cet octet. Vous devez donc *toujours* réserver `len + 1` octets pour stocker une chaîne de longueur `len`, et écrire explicitement le terminateur nul à la fin : `line.data[L] = '\0';`.
Sans ce marqueur, les fonctions liront au-delà de la fin du texte, provoquant un affichage erroné ou un comportement indéfini. Dans votre implémentation, terminez toujours la chaîne par `'\0'`.

Exercice 3 : Chargement d'un fichier

Implémentez la fonction `file_load` dans `io.h` qui charge le contenu d'un fichier texte ligne par ligne dans le buffer.

1. Avant toute lecture, veillez à **vider et réinitialiser le buffer** à l'aide de `buffer_free` puis `buffer_init`, afin d'éviter toute fuite mémoire ou contenu résiduel d'un fichier précédent.
2. Utilisez les fonctions de la bibliothèque standard `fopen()` et `getline()` pour ouvrir et lire le fichier ligne par ligne. Si le fichier n'a pas pu être ouvert (par exemple s'il n'existe pas), la fonction doit **retourner** `-1`.
3. Pour chaque ligne lue, retirez les caractères de fin de ligne `'\n'` et `'\r'` afin de ne conserver que le texte brut. Après cette suppression, repositionnez correctement le caractère de fin de chaîne `'\0'`.
4. Insérez ensuite la ligne obtenue dans le buffer à l'aide de `buffer_insert_line`. Si aucune ligne n'a été lue (par exemple si le fichier est vide), insérez manuellement une ligne vide afin de garantir qu'au moins une ligne existe dans le buffer.
5. Enfin, fermez le fichier et **retournez** `0` en cas de succès.

Exercice 4 : Testez !

Des fonctions utilitaires sont fournies dans le fichier `test_utils.h` pour vous aider à valider votre implémentation. Utilisez ces fonctions afin de tester votre code, par exemple en lançant votre programme avec un fichier en entrée et en affichant le contenu lu directement dans le terminal.
Modifiez votre `main.c` pour tester le programme.

IV) Lire un fichier dans l'éditeur

Exercice 5 : Editor

Nous allons commencer par implémenter les fonctions permettant d'utiliser la structure `Editor`. Les fonctions `editor_init` et `editor_free` sont partiellement implémentées. Pour le moment, seule la gestion de **ncurses**, la bibliothèque utilisée pour gérer une fenêtre indépendante dans le terminal, est réalisée.
La fonction `editor_set_status` est déjà implémentée. Elle permet de mettre à jour le message d'état courant de l'éditeur, qui sera ensuite affiché par `ncurses` dans la barre d'état.

Commençons par la fonction `editor_init` :

1. Avant toute utilisation, nettoyez la mémoire du buffer à l'aide d'un remplissage par des zéros. Cette étape garantit que toutes les variables de la structure `Editor` sont correctement initialisées.
2. Initialisez les différents éléments de l'éditeur : `buffer`, `filename` et `dirty`. Si aucun nom de fichier n'est fourni, `filename` doit être initialisé avec une chaîne arbitraire (par exemple `"NoName"`).
3. Initialisez ensuite les variables de l'objet `Cursor` associé à l'éditeur.
4. Chargez le fichier si un nom a été fourni. Si aucun fichier n'est spécifié, insérez tout de même une ligne vide dans le `buffer` pour permettre l'édition.
5. Enfin, mettez à jour le message de statut à l'aide de `editor_set_status`. Si un fichier a été ouvert, indiquez que l'éditeur est en train de l'éditer ; sinon, affichez qu'il s'agit d'un nouveau fichier.

Attention : la fonction `editor_set_status` accepte un nombre variable d'arguments, comme `printf`. Cela signifie que vous pouvez lui passer une chaîne de format contenant des spécificateurs (`%s`, `%d`, etc.) ainsi que les variables à substituer dans cette chaîne. Utilisez-la de manière appropriée pour afficher le message de statut.

6. Libérez correctement la mémoire dans la fonction `editor_free`.

Exercice 6 : Affichage

Intéressons nous maintenant à **render.c**. Ce fichier implémente toute la logique d'affichage de notre objet `Editor`. Cette partie du code est composé majoritairement d'appels à `ncurses`. L'utilisation de cette librairie pouvant être très fastidieuse et n'étant pas l'objet principal de ce TP, la majorité du code est déjà disponible pour vous (de rien...).

1. Lisez le code et appuyez vous sur la documentation (**man ncurses** ou bien https://invisible-island.net/ncurses/man/ncurses.3x.html) pour le comprendre. Ne perdez pas trop de temps sur cette étape.
2. Dans les fonctions `ensure_cursor_visible` `draw_text`, initialisez correctement la variable `text_rows` afin qu'elle représente le nombre de lignes de texte affichables dans la fenêtre. Pour cela, souvenez-vous que la dernière ligne de la fenêtre est réservée à la **barre d'état** (affichée par la fonction `draw_status`). Le nombre de ligne visible dans le terminal est stockée dans `E->screen_rows`, initialisée par la fonction `getmaxyx(stdscr, ...)` lors de l'initialisation de l'objet `Editor`.
3. Toujours dans `draw_text`, initialisez les variables `vx` et `vy` pour représenter la position visuelle du curseur à l'écran. Cette position dépend de la ligne et de la colonne actuelles du curseur dans le texte, mais aussi du décalage de défilement de la fenêtre (si une partie du texte n'est plus visible à l'écran).

Exercice 7 : Testez !

À ce stade, le programme devrait pouvoir afficher un fichier dans une fenêtre dédié. Modifiez `main.c` pour tester le programme.

V) Gestion des entrées clavier

Exercice 8 : Déplacements du curseur

Le curseur doit pouvoir se déplacer dans le texte à l'aide des flèches directionnelles du clavier. Les fonctions `move_left`, `move_right`, `move_up` et `move_down` de `input.c` gèrent ces mouvements en mettant à jour la position du curseur dans la structure *Editor*.

1. **Déplacements horizontaux** Implémentez les fonctions `move_left` et `move_right`. Le curseur doit se déplacer d'un caractère à la fois sur la ligne courante. Si le curseur atteint le début ou la fin de la ligne, il doit passer à la ligne précédente ou suivante.
2. **Déplacements verticaux** Implémentez les fonctions `move_up` et `move_down`. Le curseur doit se déplacer d'une ligne vers le haut ou vers le bas. Lorsque vous changez de ligne, pensez à ajuster la position horizontale (`col`) si la nouvelle ligne est plus courte que la précédente, pour éviter de dépasser la longueur du texte sur cette ligne.

Exercice 9 : Modification du texte

Dans cet exercice, nous allons implémenter les fonctions permettant de modifier le contenu du texte. Pour cela, vous aurez besoin d'appeler certaines fonctions définies dans `buffer.c`, qui ne sont pas encore implémentées. Pas de panique : leurs signatures sont déjà connues, vous pouvez donc écrire votre code comme si elles existaient. Nous reviendrons plus tard sur leur implémentation.

1. Implémentez la fonction `insert_char_at_cursor`. Pensez à mettre à jour **toutes** les variables impactés.
2. Implémentez ensuite `insert_newline_at_cursor`. Utilisez la fonction `buffer_split_line` pour diviser la ligne en deux à la position du curseur. Vous appellerez cette fonction même si le curseur se trouve déjà en fin de ligne, afin de simplifier le code et d'éviter de multiplier les cas particuliers.
3. Implémentez enfin `backspace_at_cursor`. Attention au cas particulier où le curseur se trouve au **début d'une ligne** : dans ce cas, il faut fusionner la ligne courante avec la précédente à l'aide de `buffer_join_with_previous`.

Exercice 10 : Capture de l'entrée clavier

Pour finaliser cette partie, nous allons implémenter la fonction `handle_key`. Cette fonction doit réagir aux différentes entrées clavier capturées par *ncurses* :

- **Ctrl + Q** : quitter l'application ;
- **Ctrl + S** : sauvegarder le fichier ;
- Les **flèches directionnelles** : capturées via les constantes `KEY_LEFT`, `KEY_RIGHT`, `KEY_UP` et `KEY_DOWN` de la bibliothèque `ncurses` ;
- Le **retour arrière** (`KEY_BACKSPACE`) ;
- La **touche Entrée** (ou retour chariot) ;
- Enfin, les **caractères à insérer** dans le texte. Pour vérifier si un caractère est insérable, utilisez la fonction `isprint(int _c)` de la bibliothèque standard.

Les **caractères imprimables** sont ceux qui possèdent une représentation visuelle (lettres, chiffres, symboles, espace, etc.) ; ils sont détectables avec `isprint()`. À l'inverse, les **caractères de contrôle** (comme `\n` pour la nouvelle ligne ou `\t` pour la tabulation) ont bien un code ASCII, mais *ne produisent pas de glyphe visible* : ils déclenchent une action (saut de ligne, tabulation, etc.).
L'utilisation de *ncurses* permet de gérer proprement ces entrées, notamment les **touches spéciales** (flèches, F1–F12, combinaisons Ctrl, etc.) via des constantes `KEY_*` et `keypad(stdscr, TRUE)`. Une lecture naïve avec `getchar()` ne distinguerait pas correctement ces touches.
N'oubliez pas de modifier le `status` de l'`Editor` quand nécessaire !

Exercice 11 : Main

Nous avons maintenant tous les éléments nécessaires pour implémenter la fonction `main`, qui sera finalement assez simple à écrire !

1. Implémentez l'initialisation de la structure `Editor`.
2. Implémentez ensuite la boucle principale qui capture les entrées clavier et appelle la fonction `handle_key`. Pour cela, utilisez la fonction `getch()` de la bibliothèque `ncurses`, qui permet de récupérer une touche pressée par l'utilisateur.

VI) Dernières fonctions

Notre éditeur de texte est presque prêt ! Il manque juste l'implémentation des fonctions que l'on avait remis à plus tard.

Exercice 12 : Enregistrement du fichier

Pour le moment, lorsque l'on appuie sur `Ctrl + S` dans l'éditeur, l'entrée clavier est bien capturée et la fonction `file_save` est appelée. Cependant, celle-ci ne fait encore rien !

1. Implémentez la fonction `file_save`. Elle devra écrire chaque ligne du `Buffer` dans le fichier cible en gérant correctement les erreurs d'écriture. Assurez-vous également que le fichier est fermé proprement en cas d'erreur.

Exercice 13 : Modifications du Buffer

On arrive au bout ! Il ne reste qu'à implémenter les fonctions que nous avions laissé de côté tout à l'heure. Vous devez être maintenant habituer à la structure du code. Ce dernier exercice ne sera donc pas guidé. Bonne chance !

1. Implémentez `line_insert_char` et `line_delete_char`.
2. Implémentez `buffer_delete_line`, `buffer_split_line` et `buffer_split_line`.

Félicitation, vous êtes arrivé au bout de ce TP !