Polytech Tours
Département Informatique

Programmation en langage C, travaux dirigés et pratiques guidés

17 octobre 2017

Sébastien Aupetit, Ronan Bocquillon
ronan.bocquillon@univ-tours.fr (DI, bureau 203)

1 Organisation des séances

Les séances de TDs et TPs vont vous permettre de progressivement écrire une application de facturation ultra-simplifiée. Pour cela, les 25h TPs sont réparties de la façon suivante :

TP 1.1, TP 1.2, TP 1.3 & TP 1.4	Prise en main d'un IDE, d'un compilateur et du projet
TD 2, TD 3 (1/2)	Allocation, caractères et chaines de caractères (strdup, strncpy, strcmp, strcasecmp, index, strstr, insertion, extraction)
TD 3 (2/2)	Algorithme de Vigenère, calcul sur des caractères
TP 4	Mise en oeuvre et finalisation des TD2 et TD3
TD 5	Conversion de base et formattage de dates
TD 6	Tableau dynamiques et E/S avec fichiers textes
TD 7, TP 8	Fonctions de validation de valeur, conversions simples, E/S sur des fichiers binaires avec des enregistrements de taille fixe, attributs textes stockés dans l'enregistrement
TP 9	Fonction de validation de valeur, conversions simples, E/S sur des fichiers binaires avec des enregistrements de taille fixe, attributs textes stockés sur le tas
TD 10, TP 11	Liste chainée simple, opérations sur la liste, E/S sur des fichiers binaires avec des enregistrements de taille variable
TD 12, TP 13	Lecture et analyse de fichiers textes avec longueur de lignes inconnues *a priori*, gestion d'un dictionnaire de valeurs avec des unions, formattage de texte à l'aide du dictionnaire

A la fin de ces séances vous devrez rendre le code que vous aurez produit afin qu'il soit noté. Cette note constituera votre note de contrôle continu. L'absentéisme fera également partie de la notation.

Avertissement

Pour que nous puissions étudier et corriger un maximum d'exercices lors des séances de travaux dirigés mais aussi pour les séances de travaux pratiques vous soient profitables, vous devez préparer les exercices avant les séances.

2 TP1.1 - Prise en main de l'environnement de travail

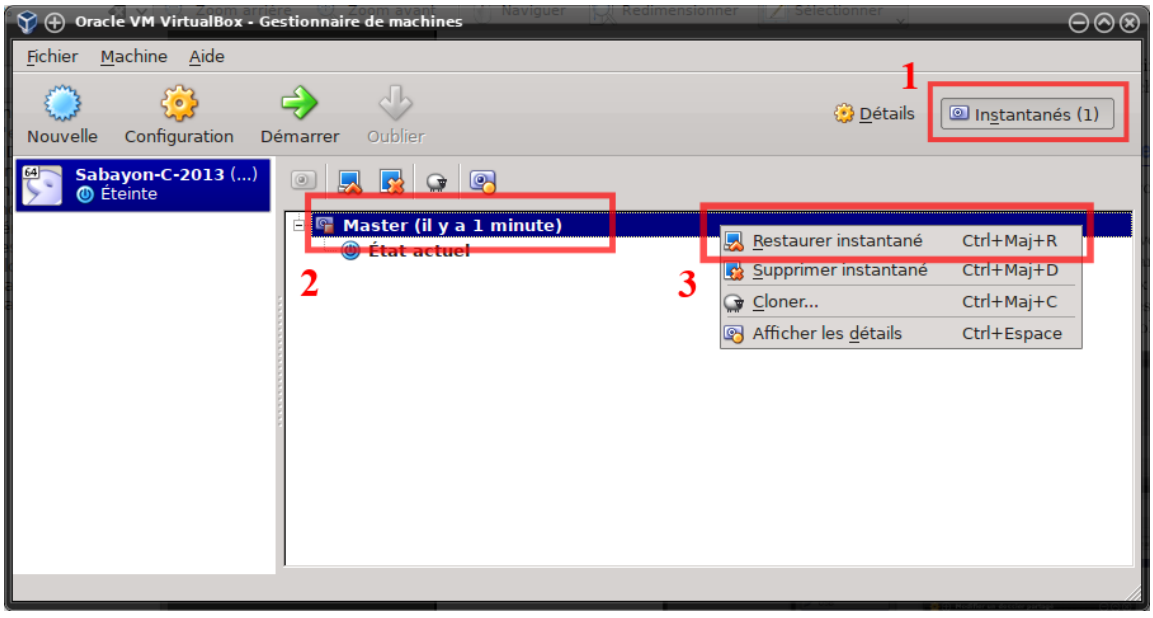
L'ensemble des séances de travaux pratiques se déroulement sous un environnement GNU/Linux. Pour cela, une machine virtuelle préconfigurée est mise à votre disposition. Cette machine virtuelle a été configurée pour qu'elle vous soit facile d'accès et de compréhension. Le nom de la machine virtuelle est « VMWARE Debian_x64 – PROJETS C ».

1 A chaque utilisation et avant d'exécuter la machine virtuelle

1.1 Remise en état original de la machine virtuelle

Si la machine virtuelle possède un « Instantané » ou « Snapshot » nommé « tools ». Vous devez le restaurer afin d'obtenir une machine virtuelle propre :

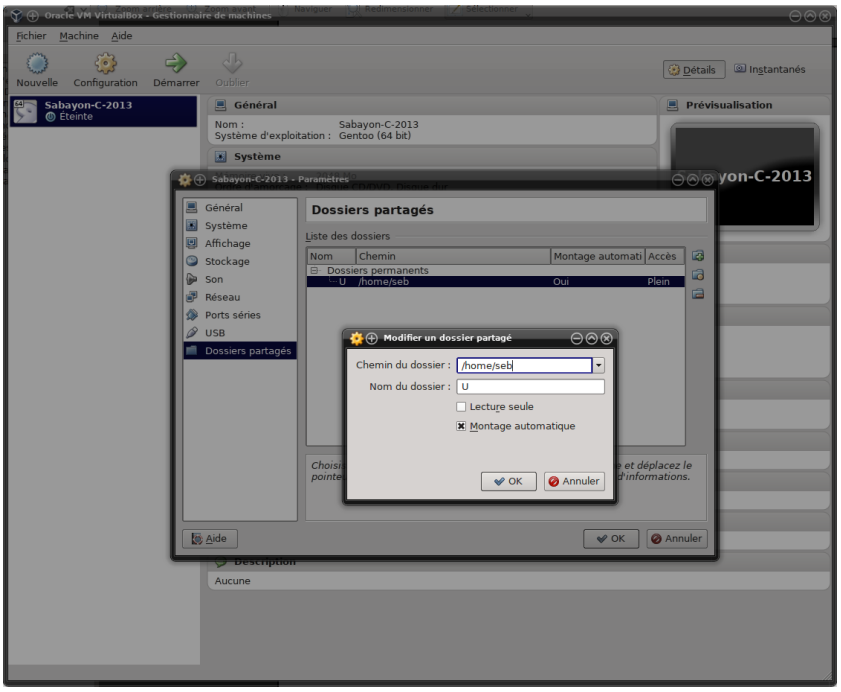
1. Cliquez sur « Instantané » ou « Snapshot »
2. Cliquez sur « tools »
3. Cliquez avec le bouton de droite et choisissez « Restaurer instantané » ou « Restore snapshot ».



1.2 Le dossier partagé

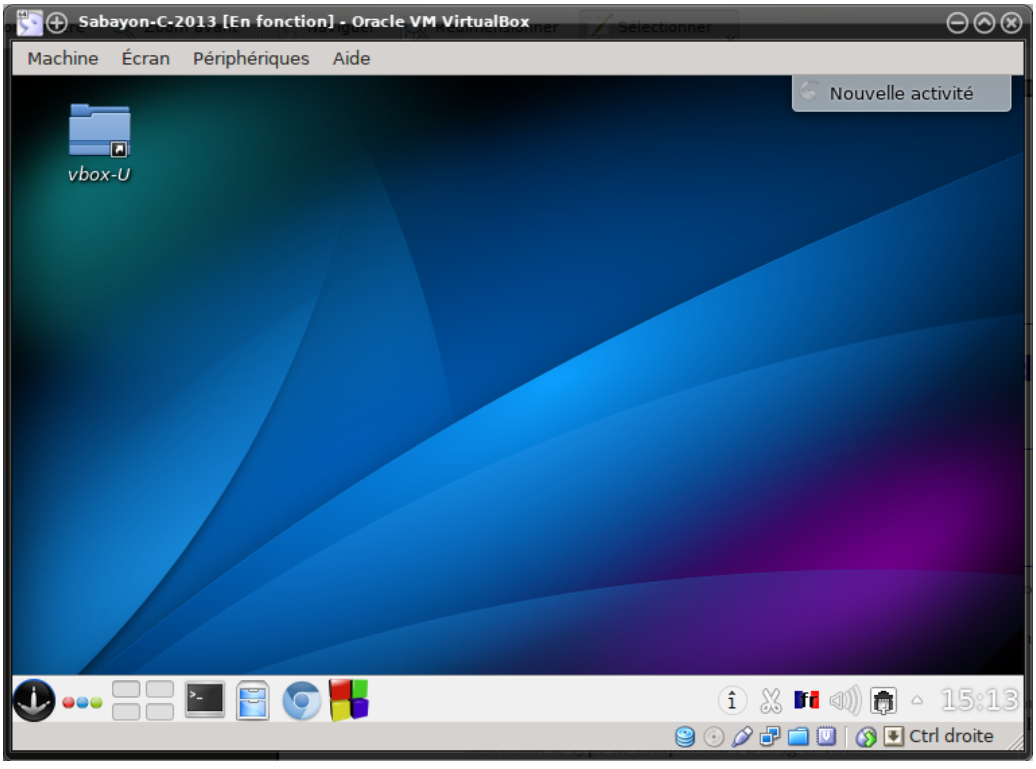
Avant l'exécution de la machine virtuelle, vous devez vous assurer que vos fichiers sont accessibles depuis le système GNU/Linux.

Pour cela, sous VirtualBox, vous sélectionnez la machine virtuelle puis vous cliquez sur configuration. Vous devez aller dans la liste gauche sur « Dossiers partagés » ou « Shared folders ». Un partage portant le nom « `U` » doit être défini. Le chemin que vous spécifiez ici sera accéssible depuis le système GNU/Linux et cela vous permettra de stocker vos fichiers en dehors de la machine virtuelle. Ainsi, aucun de vos camarades ne pourra accèder à votre code en utilisant la même machine virtuelle. En standard dans les salles TP de l'école, le chemin doit être « `U:\` ».



2 Démarrer la machine virtuelle

Démarrer la machine virtuelle en cliquant sur « Démarrer » ou « Start ». Après quelque seconde, vous devriez aboutir à un bureau KDE comme ci-dessous.



L'organisation est très similaire à ce que l'on peut trouver sur d'autres systèmes d'exploitation telle que Windows. La partie principale contient un répertoire « `vbox-U` qui correspond au partage mis en place plus haut. En bas de l'écran, on trouve respectivement de gauche à droite : le menu démarrer, le menu des activités (non utile ici), le menu des bureaux (non utile ici), une icone permettant de lancer une console (ici il s'agit d'une console `konsole`), une icone pour lancer l'explorateur de fichiers, une icone pour lancer le navigateur `chromium` (identique à Google Chrome) et une icone pour lancer l'éditeur `Code::Blocks`. Pour effectuer les travaux pratiques, vous utiliserez principalement ces quatre programmes.

2.1 Raccourcis clavier de VirtualBox

Par défaut sous VirtualBox, la touche `Ctrl` située à droite du clavier est utilisée comme touche spéciale pour VirtualBox. Les principales combinaisons de touches qui peuvent vous être utiles sont :

- `Ctrl droite+F` pour passer en mode plein écran ou en sortir
- `Ctrl droite+F1`, `Ctrl droite+F2`, `Ctrl droite+F3`, `Ctrl droite+F4`, `Ctrl droite+F5`, `Ctrl droite+F6` pour accèder aux consoles textuelle du système GNU/Linux
- `Ctrl droite+F7` pour accèder à l'interface graphique

Trucs et actuces

Il est recommandé de travailler en mettant la machine virtuelle en plein écran pour plus de confort.

2.2 La console

Dans le cadre de ces travaux pratiques, vous serez amené à utiliser la console pour saisir des commandes. Il existe de nombreux programmes réalisant une console. Avec cette machine virtuelle, nous utiliserons le programme « konsole ». Vous pouvez le démarrer en cliquant sur l'icone associée en bas de l'écran.

Une console permet d'avoir accès à un invité de commande ou « shell » pour effectuer des actions. Les principales commandes qui vous seront utiles sont données ci-dessous. Pour chacune de ces commandes, si `X` contient des espaces, vous devez utiliser `"X"` à la place. Sans espace, les guillemets ne sont pas nécessaires.

`~`	Pour un chemin, cela représente le répertoire de l'utilisateur c'est-à-dire `/home/user/`.
`ls`	Liste le contenu du répertoire courant
`ls -al`	Liste le contenu du répertoire courant
`ls X`	Liste le contenu du répertoire `X`
`ls -al X`	Liste le contenu du répertoire `X` avec plus de détails
`pwd`	Affiche le chemin complet du répertoire courant
`top`	Affiche l'utilisation du CPU par les processus (`q` pour quitter, `k` pour tuer un processus)
`killall X`	Demande à un programme `X` de s'arrêter
`killall -s9 X`	Tue un programme `X` qui ne veut pas s'arrêter tout seul
`cd`	Le répertoire courant devient le répertoire de l'utilisateur (`/home/user`)
`cd X`	Le répertoire `X` devient le répertoire courant
`cd ..`	Le répertoire parent devient le répertoire courant
`cmd | less`	Affiche progressivement page par page ce qu'affiche le programme `cmd` (utilisez les flèches et la touche espace pour vous déplacer dans l'affichage, utilisez `q` pour quitter)
`rm X`	Efface le fichier `X`
`rm -rf X`	Efface le fichier `X` ou le répertoire `X` et son contenu de façon récursive sans demander confirmation
`mkdir X`	Crée le répertoire `X`
`man X`	Permet d'obtenir la page de manuelle de la commande `X`
`man 3 X`	Permet d'obtenir la page de manuelle de la fonction `X` du langage C

2.3 Parce que parfois `Code::Blocks` bogue

Il peut arriver que `Code::Blocks` bug et ne veuille plus démarrer ou tout simplement qu'il se fige et ne reponde plus. Il est nécessaire de distinguer les deux cas.

2.3.1 `Code::Blocks` ne démarre plus

Ce bogue se produit lorsque `Code::Blocks` a corrompu ses fichiers de configurations et n'arrive plus à les relire. Il vous suffit d'utiliser la commande suivante dans une console

```
killall -s9 codeblocks
rm -rf ~/.codeblocks
```

2.3.2 Code::Blocks ne répond plus

Il vous suffit d'utiliser la commande suivante dans une console

```
killall codeblocks
```

et si cela ne suffit pas, vous utilisez

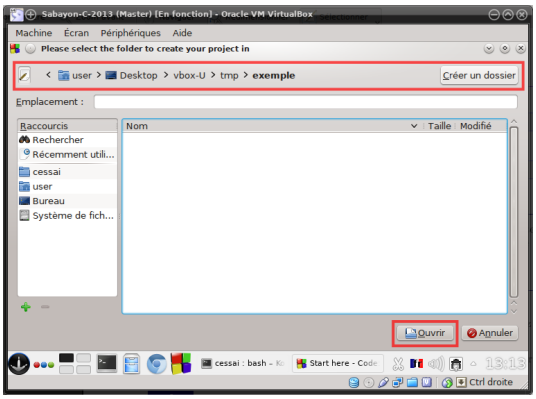
```
killall -s9 codeblocks
```

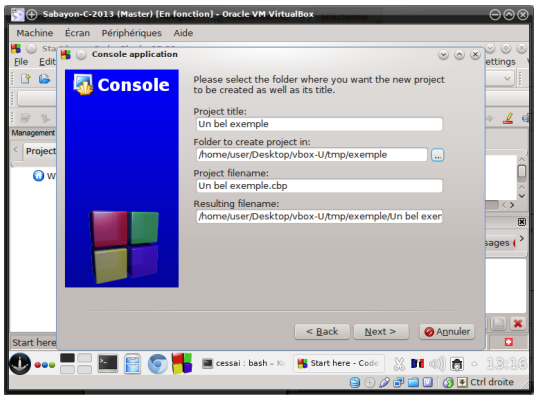
3 Utilisation de Code::Blocks

Dans le cadre de ces TDs/TPs, vous aller travailler sur une application complète déjà configurée pour `Code::Blocks` dans laquelle vous aurez à remplacer des portions de code par votre propre code. Avant d'aborder cela, vous devez d'abord maitriser votre environnement de développement.

3.1 Création d'un projet Code::Blocks

1. Créez un projet du type console [1].
2. On vous demande dans quel langage vous voulez le créer. Sélectionnez « C ».
3. Choisissez un titre pour votre projet.
4. Choisissez le répertoire où créer le projet. Utilisez le sélecteur de fichier pour (1) créer un répertoire dans `~/Desktop/vbox-U/` et (2) selectionnez le.





5. Dans l'écran suivant, vous pouvez voir que deux cibles différentes sont crées : une version Debug et une version Release.
6. Vous avez accès à vos fichiers sur la gauche et à un éditeur sur la droite.

3.2 Paramétrage de la construction du programme

Le projet peut être paramétré via le menu `Project`. Vous pouvez entre autres :

— Ajouter ou retirer des fichiers du projet.

Erreurs fréquentes

Lorsque vous ajoutez des fichiers sources, vous devez les ajouter à l'ensemble des « Target » (c'est-à-dire Debug et Release) sinon ils ne seront pris en compte que pour les « Target » qui les incluent.

— `Set programs' arguments` permet de spécifier les paramètres à transmettre au programme lors de son exécution. Ces paramètres sont récupérés par le programme via les variables `argc` et `argv` de la fonction `main`.

— `Build options` permet de définir les paramètres de compilation du programme.

1. `Code::Blocks` est capable de gérer de nombreux types de projet. Dans le cadre de ces TPs, nous nous limiterons au cas des projets du type console ou « Console Application ».

3.3 Construction et exécution du programme

Pour construire/compiler votre programme, vous pouvez utiliser soit le menu `Build`, soit les raccourcis claviers (recommandé), soit la barre d'outil.

- `Build` : Compile uniquement les fichiers dépendant des fichiers modifiés depuis la dernière compilation et crée l'executable.
- `Run` : Execute le programme.

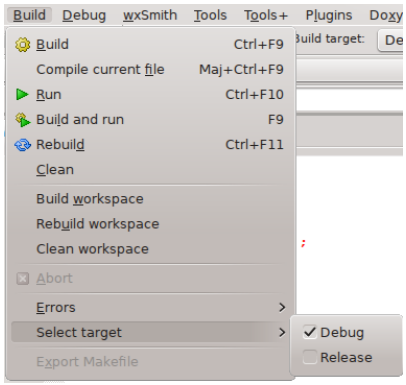
Avertissement

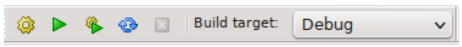
Le programme n'est pas compilé. Le programme exécuté correspond au dernier exécutable crée même si vous avez modifié le code depuis.

- `Build and run` : Compile et exécute votre programme
- `Rebuild` : Efface les fichiers déjà compilés et recompile tout le projet
- `Clean` : Efface les fichiers déjà compilés.
- `Select target` : Définit quelle est la version du programme concernée.

Erreurs fréquentes

N'utilisez pas la commande `Build and run` ! Lors de la compilation, le compilateur peut vous donner des avertissements. Avec cette commande, vous ne vous en apercevez pas car immédiatement ces messages sont remplacés par la sortie d'affichage de votre programme. **Il est très fortement recommandé d'utiliser les commandes `Build` puis `Run` à la place.**





3.4 Débuggage d'un programme

En général, chaque programme que vous développerez existera en deux versions : une version Debug et une version Release. Dans la version Debug, les temps d'éxécution sont plus long mais vous pouvez effectuer du déboggage de code. A l'opposé, la version Release est plus rapide mais elle ne permet pas de faire du deboggage. Le choix de l'une ou l'autre des versions s'effectue sous `Code::blocks` en choisissant le bouton `Build target` dans la barre d'outil ou via le menu `Build->Select target`.

Lorsque vous êtes en version Debug, vous pouvez tracer instruction par instruction ce que que fait réellement votre programme. Pour cela, il suffit de définir des points d'arrêt par un clic droit dans la marge et en choisissant « Add breakpoint » : un point rouge apparait indiquant la présence d'un point d'arrêt. Vous pouvez ajouter autant de point d'arrêt que vous le souhaitez.

Pour exécuter votre programme en mode Debug, vous devez utiliser le menu `Debug`, les raccourcis claviers (recommandé) ou la barre d'outils. Les commandes disponibles sont :

- `Start/Continue` : Débute l'exécution en mode Debug ou continue l'exécution du programme jusqu'au prochain point d'arrêt.
- `Stop debugger` : Arrête l'exécution du programme et du débugger
- `Run to cursor` : Débute l'exécution en mode Debug ou continue l'exécution du programme jusqu'à ce que la ligne courante dans l'éditeur de code soit atteinte
- `Next line` : Exécute la ligne d'instuction suivante
- `Step into` : Entre dans la fonction et attend d'autres ordres
- `Step out` : Continue l'exécution de la fonction jusqu'à ce qu'elle se termine

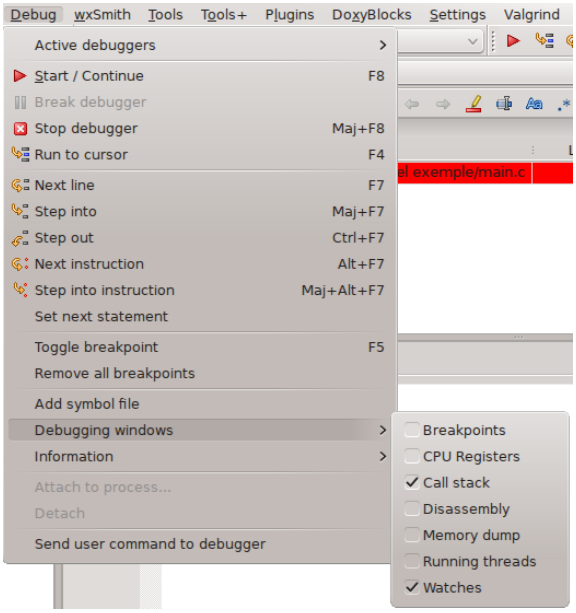
— `Debugging windows` : Permet d'afficher des fenêtres d'information spéciales (ex : la pile d'appel, les valeurs des variables...)

Trucs et actuces

Assurez vous d'avoir en permanence les fenêtres d'information `Call stack` et `Watches` ouvertes lors du déboggage de vos programmes. Dans la fenetre `Watches` vous pouvez ajouter des expressions supplémentaires qui seront calculées par le debugger (ex : `tab[10][i+2]`, `*record`, `record->attribute`).

Erreurs fréquentes

N'utilisez pas les commandes `New instruction` et `Step into instuction` car vous vous retrouverez dans le code assembleur du programme.



3 TP1.2 - Un peu de pratique pour s'exercer

A la fin de ce TP, vous devrez avoir effectué les actions suivantes sur votre programme :

1. Avoir créé un programme du type « Console Application »
2. Avoir configuré votre projet tel que :
   — En mode Débug et Release : les options de compilation suivantes soient activées : −−std=c89, −Wall et −Wstrict−prototypes
   — Définir le symbole NDEBUG pour le mode Release. Vérifier la prise en compte correcte de ce symbole en ajoutant une assertion toujours fausse dans votre programme [1]. Exécuter votre programme en mode Debug et en mode Release. Que constate-t-on ?
3. Avoir déboggé votre programme et avoir effectué un pas à pas, une inspection des variables et de la pile d'appel.

1 Le tri à bulles

La méthode du tri à bulles est une méthode basique de tri. Si N désigne le nombre d'éléments à trier, sa complexité est en $O(N^2)$, ce qui classe cette méthode parmi les moins performantes.

1.1 Principes

Soit N valeurs scalaires (entières ou réelles) à trier par valeur croissante. Ces N valeurs forment une liste. Le principe de tri consiste à parcourir la liste et à comparer deux éléments successifs au sein de la liste. Les deux éléments adjacents sont permutés lorsque l'ordre croissant des valeurs n'est pas respecté. Ainsi, les éléments de plus faible valeur remontent en début de liste. Si lors d'un parcours de la liste, aucune permutation n'est réalisée, cela signifie que la liste est totalement triée.

1.2 Exemple

Soit la liste d'entiers $(6, 2, 5, 3, 9)$ que l'on souhaite trier par valeur croissante. Les différentes étapes de la méthode sont décrites ci-dessous :

Etape 1

$$(\mathbf{6}, \mathbf{2}, 5, 3, 9) \rightarrow (2, 6, 5, 3, 9)$$
$$(2, \mathbf{6}, \mathbf{5}, 3, 9) \rightarrow (2, 5, 6, 3, 9)$$
$$(2, 5, \mathbf{6}, \mathbf{3}, 9) \rightarrow (2, 5, 3, 6, 9)$$
$$(2, 5, 3, \mathbf{6}, \mathbf{9}) \rightarrow (2, 5, 3, 6, 9)$$

Commentaire : les deux derniers éléments de la liste sont triés.

Etape 2

$$(\mathbf{2}, \mathbf{5}, 3, 6, 9) \rightarrow (2, 5, 3, 6, 9)$$
$$(2, \mathbf{5}, \mathbf{3}, 6, 9) \rightarrow (2, 3, 5, 6, 9)$$
$$(2, 3, \mathbf{5}, \mathbf{6}, 9) \rightarrow (2, 3, 5, 6, 9)$$

Etape 3

$$(\mathbf{2}, \mathbf{3}, 5, 6, 9) \rightarrow (2, 3, 5, 6, 9)$$
$$(2, \mathbf{3}, \mathbf{5}, 6, 9) \rightarrow (2, 3, 5, 6, 9)$$

Commentaire : aucune permutation ; fin du tri.

1. La fonction `assert(cond)` de `assert.h` est desactivée grâce au symbole préprocesseur `NDEBUG`.

1.3 Algorithme

On suppose que la liste de nombres à trier est représentée sous la forme d'un tableau de MAX entiers. La valeur de MAX est connue *a priori*. La méthode de tri est implémentée sous la forme de la fonction tri_bulle.

Entrée : tab : le tableau de nombres
MAX : type entier ; `/* nombre d'elements du tableau */`
Précondition : MAX ≥ 0
Sortie : rien
Postcondition : le tableau est trie en ordre croissant
Variable locale : i, j : type entier ; indice de parcours des elements du tableau.
tmp : type identique aux elements du tableau ; variable de stockage temporaire.
non_trie : type booleen ; `/* booleen permettant de savoir si le tableau */`
`/* est trie ou non (non trie = vrai si le tableau n'est pas trie). */`

Début

```
non_trie ← Vrai ; /* le tableau n'est pas trie                    */
i ← 0;
while i ≤ MAX-1 et non_trie est Vrai do
    non_trie ← Faux;
    for j = 1 to i = MAX-i do
        if tab[j] < tab[j-1] then
            tmp ← tab[j-1];
            tab[j-1] ← tab[j];
            tab[j] ← tmp;
            non_trie ← Vrai;
```

1.4 Mise en oeuvre : Version 1

Dans une première version, le tableau à trier sera considéré sous la forme d'une variable globale et les éléments du tableau seront de type entier. Pour réaliser cette mise en oeuvre, vous devez créer un projet et le configurer comme indiqué plus haut. Vous devez réaliser vos tests d'exécutions en mode Débug et en mode Release.

1.4.1 Création d'un tableau de MAX entiers et affectation des éléments

Ecrire la fonction init_tab qui permet d'affecter aux éléments du tableau tab_int une valeur entière pseudo aléatoire [2].

1.4.2 Affichage de la valeur des éléments du tableau tab_int

Ecrire la fonction affiche_contenu_tab_int qui permet d'afficher à l'écran le contenu de chaque élément du tableau. L'affichage doit être formaté selon l'exemple ci-dessous :

```
....
Tab_int[2] = 25
Tab_int[3] = 12
....
```

1.4.3 Ecriture de la fonction main

Ecrire la fonction main qui contiendra les appels aux 2 fonctions précédentes. Vérifier que le comportement de votre programme est celui attendu.

1.4.4 Fonction tri_bulle

Ecrire le code de la fonction tri_bulle. Insérer l'appel de cette fonction dans la fonction main. Vérifier que le tableau est bien trié par valeur croissante.

Erreurs fréquentes

Vous constaterez que le programme ne tri pas correctement les nombres du tableaux. Détectez et corrigez l'(es) erreur(s) en utilisant le débuggage pour tracer pas à pas ce que fait votre programme.

1.5 Mise en oeuvre : Version 2

Modifiez la version 1 de manière à ne plus utiliser de variables globales.

2. Cf. chapitre du support de cours

4 TP1.3 - Récupération du projet

Dans le répertoire `~/Desktop/vbox-U/`, créer un répertoire spécifique pour le projet tel que par exemple `ProjetC-JeanNoelDeLaMottePicquet`. Ce nom de répertoire ne doit contenir que des lettres non accentuées et/ou des chiffres.

Dans une console, vous pouvez utiliser les commandes suivantes :

```
cd ~/Desktop/vbox-U
mkdir ProjetC-JeanNoelDeLaMottePicquet
```

Placez vous dans le répertoire ainsi créé :

```
cd ProjetC-JeanNoelDeLaMottePicquet
```

ou

```
cd ~/Desktop/vbox-U/ProjetC-JeanNoelDeLaMottePicquet
```

Récupérez le projet et sa documentation en saisissant la commande suivante (en étant situé dans le répertoire) :

```
wget -q -O - https://secure.bocqfamily.fr/enseignement/c/update.sh | bash
```

Cette commande télécharge un script et l'exécute. Ce script télécharge à son tour l'ensemble des fichiers dont vous avez besoin.

Trucs et actuces

En cas de mise à jours du code source par l'enseignant, il vous suffit de saisir la commande précédente à nouveau ou la commande suivante pour en bénéficier :

```
./update.sh
```

L'ensemble des fichiers que vous ne devez pas modifier seront à nouveaux téléchargés et écrasés.

Avertissement

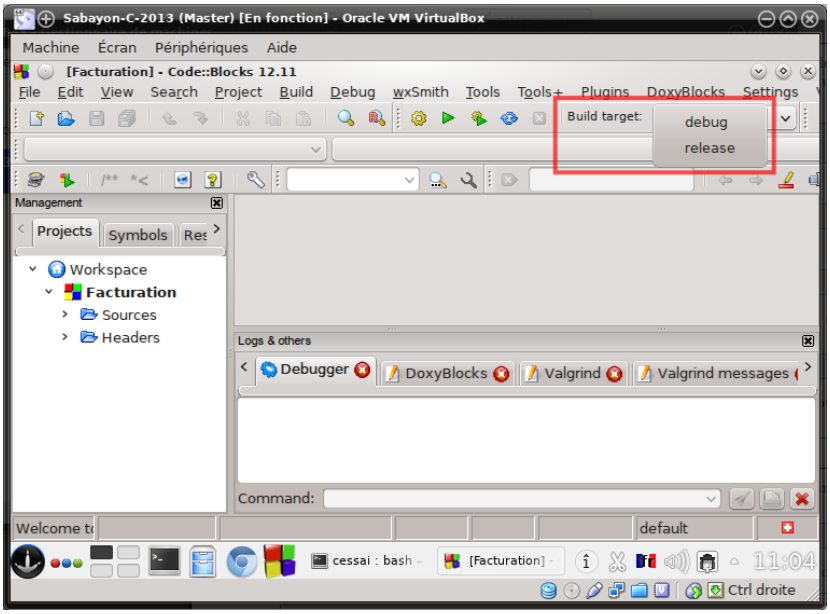
A chaque éxécution du script, une copie de sauvegarde de vos fichiers sera faite avec la création d'un répertoire parallèle daté (ex : `~/Desktop/vbox-U/ProjetC-JeanNoelDeLaMottePicquet-2013-09-06#09:54:21`).

L'ensemble des fichiers téléchargés s'organise de la façon suivante :

- `doc`contient la documentation Doxygen du projet. Elle peut être consultée avec un navigateur web tel que chromium en ouvrant le fichier `doc/index.html`.
- `subject` contient cette documentation
- `courses` contient le cours au format PDF imprimable
- `moniteur` contient les coordonnées des moniteurs
- `include` contient les fichiers d'entête (`*.h`) du projet
- `src` contient les fichiers de code source (`*.c`) du projet
- `make` contient des scripts pour la gestion du projet

— `printformat` contient les formats modèles d'impression pour le dernier chapitre
— `provided` contient le code objet des fonctions déjà implémentées (fonctions `provided_*`) sous la forme de bibliothèques dynamiques.
— `Makefile` contient le script Makefile permettant de construire le projet
— `project.cbp` est le fichier projet `Code::Blocks`
— `update.sh` est le script de mise à jours utilisé précédemment.

Le programme que vous allez développer existe en deux versions : une version Debug et une version Release. Dans la version Debug, les temps d'éxécution sont plus long mais vous pouvez effectuer du déboggage de code. A l'opposé, la version Release est plus rapide mais elle ne permet pas de faire du deboggage. Le choix de l'une ou l'autre des versions s'effectue sous `Code::blocks` en choisissant le `Build target` dans la barre d'outil ou via le menu `Build->Select target`. Ce chois peut aussi être effectué en ligne de commande via la commande `make` décrite ci-après.



Dans la console, un certain nombre de commandes sont disponibles :

— `make` :Identique à `make all` ou à `make debug/facturation release/facturation`. Construit les deux versions du programme
— `make debug/facturation` : Construit la version Debug du programme.
— `make release/facturation` : Construit la version Release du programme.
— `make clean` : Supprime tous les fichiers de code objet et les exécutables.
— `make valgrind-debug` : Exécute la version Debug du programme avec Valgrind.
— `make valgrind-release` : Exécute la version Release du programme avec Valgrind.
— `make valgrind-debug-nogui` : Exécute la version Debug du programme avec Valgrind. L'interface graphique n'est pas exécutée.
— `make valgrind-release-nogui` : Exécute la version Release du programme avec Valgrind. L'interface graphique n'est pas exécutée.
— `VALGRIND="option1 options2" make valgrind-debug` : Exécute la version Debug du programme avec Valgrind et transmet les `options1` et `option2` en paramètre au programme.
— `VALGRIND="option1 options2" make valgrind-debug | less` : Exécute la version Debug du programme avec Valgrind et transmet les `options1` et `option2` en paramètre au programme. Le texte affiché est envoyé au programme `less` qui effectue alors l'affichage écran par écran. Cette commande vous sera utile si l'affichage est trop important.

1 Options d'exécution du programme

Lors de l'exécution de votre programme, vous pouvez indiquer des paramètres supplémentaires changeant le comportement par défaut du programme.

Trucs et actuces

Sous `Code::Blocks`, vous les précisez via le menu `Project->Set programs' arguments`.

Trucs et actuces

Dans la console, vous les précisez après le nom du programme.

```
debug/facturation option1 option2
release/facturation option1 option2
```

Trucs et actuces

Dans le cas des tests valgrind, vous les précisez via la variable VALGRIND.

```
VALGRIND="option1 option2" make valgrind-debug
```

Les principales options utiles sont :

`verbose-unittests`	Par défaut, les tests unitaires sont tous exécutés et l'affichage est compacte. En utilisant ce paramètre, vous demandé l'affichage de chaque fonction des tests unitaires.
`disable-unit-MyString`	Desactive l'exécution de l'ensemble des tests unitaires associés au module `MyString`. Il existe un paramètre similaire pour chacun des modules de tests unitaires. La liste s'affiche par défaut lors de l'exécution du programme.
`disable-unit-test_toLowerChar`	Desactive l'exécution de la fonction `test_toLowerChar` du test unitaire `MyString`. Il existe un paramètre similaire pour chacune des fonctions composant un test unitaire. La liste peut être obtenue lorsque `verbose-unittests` est spécifié.
`silent-tests`	Desactive l'affichage de la progression des tests unitaires
`reduce-dump-usage`	Par défaut, la liste des fonctions non implémentées est affichée à la fin du programme. Ce paramètre permet de desactiver cet affichage.
`disable-dump-usage`	Par défaut, la liste des fonctions non implémentées et la liste des modules ayant encore des fonctions non implementées sont affichés à la fin du programme. Cette option permet de desactiver ces deux affichages.
`disable-gui`	Desactive l'interface graphique. Seule les tests unitaires sont exécutés.

2 Evaluation du projet

Pour l'évaluation du projet, les critères suivants (liste non exhaustive) sont pris en compte :

- Votre programme compile en mode Débug **et** en mode Release.
- Les fonctions sont implémentés.
- Les fonctions s'exécutent sans erreurs.
- Votre programme s'exécute en mode Debug sans aucune erreur jusqu'au bout.
- Votre programme s'exécute en mode Release sans aucune erreur jusqu'au bout.
- Votre programme s'exécute avec Valgrind sans avertissement Valgrind sur votre code.
- Vous respectez des conventions de nommage et de mise en forme du code tout au long de votre projet.
- Votre code est « propre », facilement lisibles et commenté si nécessaire.
- Vous avez suivi toutes les séances de TDs et de TPs.

5 TP1.4 - Organisation générale du projet

1 Objectifs du projet

Dans le cadre des TDs/TPs de langage C, nous vous proposons d'utiliser un programme complet et fonctionnel comme base de travail. Au fur et à mesure des séances, vous allez devoir remplacer des portions de code par les votres. Cette approche vous permet :

- d'avoir une vue globale de l'application ;
- de prendre conscience qu'un programme est constitué de « petits » codes et qu'il suffit souvent de les traiter les uns après les autres ;
- de comprendre ce qu'est un test unitaire (Le projet intègre une version simplifiée de tests unitaires) ;
- de voir comment une approche modulaire permet de séparer facilement le développement et la mise au point du code ;
- de lire et comprendre du code existant ;
- d'entrevoir concrêtement la mise en oeuvre d'une application GTK+.

Le projet proposé est une gestion de produits/devis/factures. Le projet n'est bien sûr pas complet et évite volontairement la mise en oeuvre de bases de données ou la mise en oeuvre d'un vrai système de tests unitaires tels que CUnit.

2 Les concepts

- Opérateur : individu authentifié pouvant manipuler l'application
- Produit : élément vendu par l'entreprise
- Catalogue des produits : ensemble des produits vendus par l'entreprise
- Fichier client : ensemble des clients répertoriés de l'entreprise
- Document : devis ou facture crée par l'entreprise
- Devis : évaluation des montants associés à un certain nombre de produits
- Facture : montants à payer par le client pour un certain nombre de produits

3 Les modules

Le projet est structuré en différents modules (ou unités fonctionnelles).

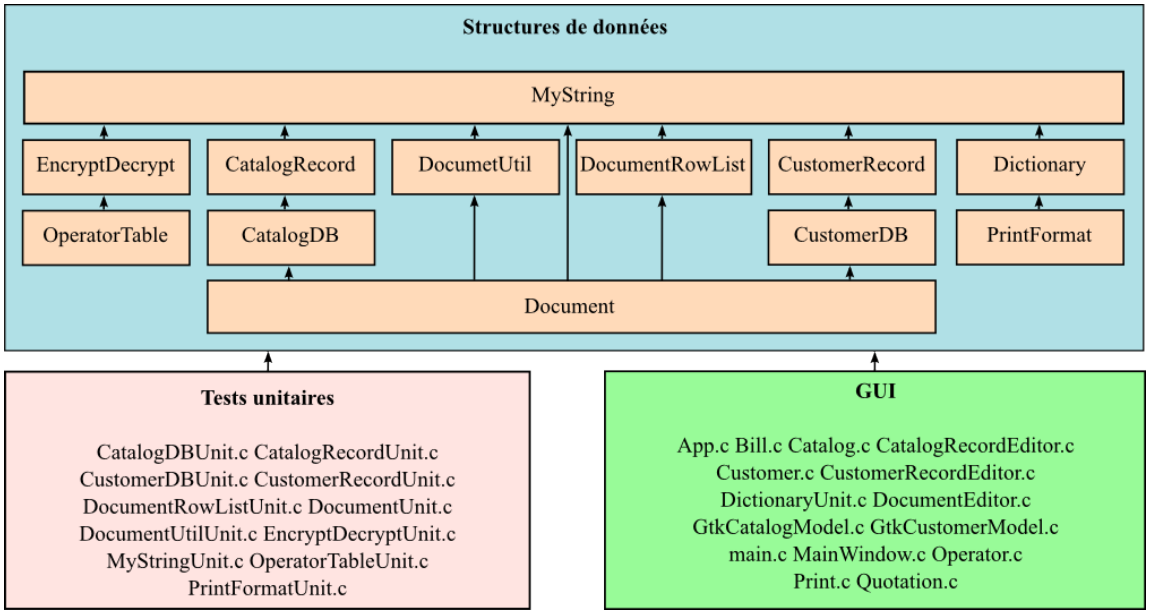
Avertissement

Chaque fois que vous remplacerez une fonction celle-ci sera utilisée par votre code mais également par le code déjà implémenté.

Fichiers	Role	Tests unitaires
MyString.h, MyString.c	**Manipulation de chaînes de caractères :** — Remplacement des fonctions standards de manipulation de chaînes de caractères — Ajout de fonctions de manipulation de chaînes de caractères	MyStringUnit.h, MyStringUnit.c
OperatorTable.h, OperatorTable.c	**Gestion des opérateurs :** — chargement et sauvegarde de la liste des opérateurs et de leurs mots de passe — ajout, suppression et modification d'un opérateur	OperatorTableUnit.h, OperatorTableUnit.c
EncryptDecrypt.h, EncryptDecrypt.c	**Cryptage et décryptage du fichier des opérateurs :** — cryptage et décryptage en fonction d'une clé de cryptage	EncryptDecryptUnit.h, EncryptDecrypt.c
CatalogRecord.h, CatalogRecord.c, CatalogDB.h, CatalogDB.c, GtkCatalogModel.h, GtkCatalogModel.c, Catalog.h, Catalog.c	**Gestion du catalogue des produits :** — Manipulation et édition d'un produit — Manipulation du catalogue des produits	CatalogRecordUnit.h, CatalogRecordUnit.c, CatalogDBUnit.h, CatalogDBUnit.c
CustomerRecord.h, CustomerRecord.c, CustomerDB.h, CustomerDB.c, GtkCustomerModel.h, GtkCustomerModel.c, Customer.h, Customer.c	**Gestion du fichier client :** — Manipulation et édition d'un client — Manipulation du fichier client	CustomerRecordUnit.h, CustomerRecordUnit.c, CustomerDBUnit.h, CustomerDBUnit.c
DocumentUtil.h, DocumentUtil.c	**Fonctions utilitaires pour la manipulation de documents :** — Calcule du numéro de document — Formattage de la date — Lecture et écriture de chaînes de caractères de taille variable dans un fichier binaire	DocumentUtilUnit.h, DocumentUtilUnit.c
DocumentRowList.h, DocumentRowList.c, Document.h, Document.c, Quotation.h, Bill .h, Quotation.c, Bill .c, DocumentEditor.h, DocumentEditor.c	**Gestion des documents :** — Manipulation d'un devis ou d'une facture — Manipulation du contenu d'un document — Edition et visualisation du contenu d'un document	DocumentRowListUnit.h, DocumentRowListUnit.c, DocumentUnit.h, DocumentUnit.c
Print.h, Print.c, PrintFormat.h, PrintFormat.c, Dictionary.h, Dictionary.c	**Gestion d'aperçu avant impression et formattage de la mise en forme d'un document :** — Visualisation de l'aperçu — Chargement d'un modèle de format — Formattage d'un document selon un modèle	DictionaryUnit.h, DictionaryUnit.c, PrintFormat.h, PrintFormat.c
main.c, App.h, App.c, Config.h,MainWindow.h, MainWindow.c, Registry.h,UnitTest.h	**Gestion de l'application, de la configuration et des tests unitaires**	

.

Les dépendences entre les modules sont données par le schéma suivant :



4 Le projet est fonctionnel : qu'est-ce que j'ai à faire ?

Chaque module possèdant des fonctions à réécrire est associé à un autre module fourni sans le code source implémentant de façon correcte une solution. L'application est donc parfaitement fonctionnelle. Les fonctions fournies à réécrire commencent toutes par le préfix provided_ et sont appellées dans les fonctions correspondantes sans préfix.

Lorsque vous réécrivez une fonction, votre implémentation est utilisée dans votre programme mais aussi lorsque les fonctions fournies utilisent aussi la fonction (ex : votre fonction stringLength sera utilisée par toutes les fonctions provided_* nécessitant stringLength).

Pour permettre ce mécanisme, des macros ont été utilisées. Pour la fonction stringLength, la déclaration du fichier MyString.h se présente ainsi :

```
1 OVERRIDABLE_PREFIX size_t OVERRIDABLE(stringLength)(const char * str);
```

Vous pouvez considérer que les macros n'ont aucun effet et que donc cela correspond à la déclaration :

```
1 size_t stringLength(const char * str);
```

Dans le fichier MyString.h, l'implémentation se présente sous la forme :

```
1 size_t IMPLEMENT(stringLength)(const char * str) {
2     return provided_stringLength(str);
3 }
```

La macro IMPLEMENT est utile au mécanisme de remplacement mais vous pouvez considérer ce code comme équivalent à

```
1 size_t stringLength(const char * str) {
2     return provided_stringLength(str);
3 }
```

Par conséquence, à chaque fois que la fonction stringLength est appellée, c'est cette fonction qui est appelée. A son tour, elle appelle la fonction provided_stringLength.

Avertissement

Le projet est configuré de manière à imposer un respect strict de la norme C89 et des bonnes pratiques en C. Ainsi, tout avertissement sur des pratiques dangereuses est considéré comme une erreur de compilation. Vous devez vous assurer de ne rien laisser au hasard y compris les conversions. En mode strict, les commentaires courts // sont interdits.

A chaque exécution de l'application, le programme commence par exécuter l'ensemble des tests unitaires sur les fonctions du programme. En cas d'erreur d'implémentation, le programme est arrêté. Si le programme est exécuté avec un debugger, le débugger est arrêté au lieu de terminer le programme afin de vous permettre d'inspecter l'état du programme ayant conduit à une erreur.

Trucs et actuces

Le code fourni définit une fonction nommée fatalError.

```
1 void fatalError(const char * message);
```

Cette fonction a pour particularité d'afficher un message et de termine le programme avec le code d'erreur 1. Elle est donc équivalente au code suivant lorsque line et file contiennent la ligne et le nom du fichier source où s'est produit l'erreur.

```
1 fprintf(stderr, "Fatal error: \"%s\" at line %ld in source file %s\n", message, line, file);
2 exit(1);
```

Cependant, en mode Debug, cette fonction a la particularité de **stopper le debugger avant de quitter le programme**. Cela vous permet donc d'effectuer un deboggage plus facilement et notamment de chercher la cause de cette erreur.

Avertissement

La réussite des tests unitaires est une étape nécessaire de validation de votre implémentation mais elle ne garantit pas que votre code est correct : il semble juste correct par rapport à ce qui a été testé.

Pour tester votre implémentation, il est fortement recommandé d'utiliser au fur et à mesure le programme valgrind sur votre programme en version Debug. Les commandes permettant de la faire vous ont été données précédemment.

5 Convention de nommage et documentation Doxygen

L'ensemble des fonctions du programme sont nommées selon l'organisation suivante en trois partie :

- Un prefix (optionnel) : jouant un rôle de qualificateur tels que provided pour les fonctions fournies ou test pour les tests unitaires.
- Un nom de module ou de structure de données (optionel) : chaque mot commence par une majuscule tels que MainWindow ou CatalogRecord.
- Un nom de fonction : chaque mot commence par une majuscule sauf le premier mot qui est en minuscule tels que removeRecord.

Les trois parties du nom sont séparées par des _. On a par exemple provided_CustomerRecord_read.

L'ensemble des fonctions du programme sont documentées à l'aide de commentaires Doxygen. La documentation obtenue se trouve dans le répertoire doc. Il vous est très fortement recommandé de la parcourir intégralement dès le début pour mieux comprendre l'organisation du projet.

6 Structuration des fichiers d'entêtes

Les fichiers d'entêtes sont structurés selon différents paquets :

- provided/*.h : ces fichiers déclares les prototypes des fonctions pré-implémentées ;
- user/*.h : à usage interne ;
- *.h : les fichiers d'entête de l'application. En générale ils incluent les fichiers d'entête définissant les structures de données et définissent les prototypes des fonctions de l'application (y compris celles que vous allez implémenter).

6 TD2&3 et TP4 : Les chaînes de caractères et leur manipulation

Dans l'ensemble de ces TDs/TPs, il vous est interdit d'utiliser le fichier d'entête string.h. Vous allez au fur et à mesure réécrire toutes les fonctions dont vous allez avoir besoin.

Le fichier d'entête base/MyString.h définit un certain nombre de macros vous permettant d'utiliser au choix, soit le nom standard de la commande, soit le nom spécifique de la commande. Les fonctions qui sont dans ce cas sont : strcmp, strlen, strcpy, strncpy, strdup, strcasecmp, tolower, toupper, strcat, strncat, index et strstr.

1 MyString.c : niveau débutant

1.1 Conversion minuscule/majuscule

Ecrire la fonction **char** toLowerChar(**char** c) qui retourne le caractère fournit en paramètre en minuscule s'il s'agit d'une lettre.

Ecrire la fonction **char** toUpperChar(**char** c) qui retourne le caractère fournit en paramètre en majuscule s'il s'agit d'une lettre.

Ecrire la fonction **void** makeLowerCaseString(**char** * str) qui transforme en minuscule la chaîne de caractères fournie en paramètre.

Ecrire la fonction **void** makeUpperCaseString(**char** * str) qui transforme en majuscule la chaîne de caractères fournie en paramètre.

1.2 Longueur d'une chaîne de caractères

Ecrire la fonction size_t stringLength(**const char** * str) qui retourne le nombre de caractères d'une chaîne. Donner au moins deux solutions utilisant des mécanismes différents.

1.3 Comparaison de chaînes de caractères

Ecrire la fonction **int** compareString(**const char** * str1, **const char** * str2) qui compare deux chaînes de caractères selon l'ordre lexicographique. La fonction retourne, respectivement, un nombre négatif, 0 ou un nombre positif, si la première chaîne est, respectivement, avant, égale ou après la seconde chaîne dans l'ordre lexicographique.

L'ordre lexicographique ordonne les chaînes suivantes dans l'ordre ci-dessous :

- "" la chaîne vide
- "a"
- "aa"
- "aaz"
- "ab"
- "b"
- "qslkddflk"
- "z"

L'ordre lexicographique respecte l'ordre naturel des caractères de la table ASCII. Ainsi, "A" est avant "a".

1.4 Comparaison de chaînes de caractères insensible à la casse

Ecrire la fonction **int** icaseCompareString(**const char** * str1, **const char** * str2) qui compare deux chaînes de caractères comme compareString mais sans tenir compte de la différence majsucule/minuscule.

1.5 Recherche de caractères dans une chaîne de caractères

Ecrire la fonction **const char** * indexOfChar(**const char** *str, **char** c) qui retourne un pointeur sur la première occurence du caractère c dans la chaîne ou NULL si la chaîne ne contient pas le caractère recherché.

1.6 Recherche d'une chaîne de caractères dans une chaîne de caractères

Ecrire la fonction **char** *indexOfString(**const char** *meule_de_foin, **const char** *aiguille) qui retourne un pointeur sur la première occurence de aiguille dans meule_de_foin ou NULL si la chaîne meule_de_foin ne contient pas la chaîne recherchée.

1.7 Est-ce que la chaîne de caractères commence par . . . ?

Ecrire la fonction **int** icaseStartWith(**const char** * start, **const char** * str) qui retourne vrai si la chaîne de caractères str débute par la chaîne de caractères start et faux sinon. La comparaison est insensible à la casse.

1.8 Est-ce que la chaîne de caractères se termine par . . . ?

Ecrire la fonction **int** icaseEndWith(**const char** * end, **const char** * str) qui retourne vrai si la chaîne de caractères str se termine par la chaîne de caractères end et faux sinon. La comparaison est insensible à la casse.

1.9 Copie d'une chaîne de caractères dans une autre

Ecrire la fonction **void** copyStringWithLength(**char** * dest, **const char** * src, size_t destSize) qui copie les caractères de src dans la chaîne dest. Cette opération copie au maximum destSize caractères dans la chaîne. Il peut donc y avoir troncature lors de la copie. Dans tous les cas, dest est une chaîne de caractères valide au sens des convention du C après l'opération.

2 MyString.c : niveau intermédiaire

2.1 Duplication d'une chaîne sur le tas

Ecrire la fonction **char** * duplicateString(**const char** * str) qui retourne une nouvelle chaîne allouée sur le tas contenant une copie des caractères de str.

2.2 Concaténation sur le tas

Ecrire la fonction **char** * concatenateString(**const char** * str1, **const char** * str2) qui retourne une nouvelle chaîne allouée sur le tas. La nouvelle chaîne est le résultat de la concaténation des deux chaînes str1 et str2.

2.3 Extraction d'une sous chaîne de caractères

Ecrire la fonction **char** * subString(**const char** * start, **const char** * end) qui retourne une nouvelle chaîne allouée sur le tas. La nouvelle chaîne contient les caractères commençant à start (inclu) et se terminant à end (exclu). La nouvelle chaîne est une chaîne de caractères valide au sens des conventions du C après l'opération. On suppose que start et end pointe sur des caractères valides d'une chaîne de caractères.

```
1 char * str = "abcdef";
2 char * s1 = subString(str,str);
3 char * s2 = subString(str,str+strlen(str));
4 char * s3 = subString(str+1,str+2);
```

L'extrait de code précédent produit les chaînes de caractères suivantes :

s1 ""

s2 "abcdef"

s3 "b"

2.4 Insertion dans une chaîne de caractères

Ecrire la fonction insertString qui retourne une nouvelle chaîne allouée sur le tas. La nouvelle chaîne est obtenue par insertion de insertLength caractères de toBeInserted dans la chaîne src à la position insertPosition. On suppose que toBeInserted contient au moins insertLength caractères autre que le marqueur de fin et que insertPosition est une position valide dans la chaîne src.

```
1 char * insertString(const char * src, int insertPosition, const char * toBeInserted, int ✂
      ↪ insertLength);
```

```
1  const char * src = "abcghi";
2  const char * toBeInserted = "def";
3
4  temp1 = insertString(src, 3, toBeInserted, 3);
5  /* temp1 doit correspondre a "abcdefghi" */
6
7  temp2 = insertString(src, 3, toBeInserted, 2);
8  /* temp2 doit correspondre a "abcdeghi" */
9
10 temp3 = insertString(src, 0, toBeInserted, 2);
11 /* temp3 doit correspondre a "deabcghi" */
12
13 temp4 = insertString(src, 6, toBeInserted, 2);
14 /* temp4 doit correspondre a "abcghide" */
```

3 EncryptDecrypt.c : chiffrage avec la méthode de Vigenère

Nous allons écrire les fonctions suivantes :

- **void** encrypt(**const char** * key, **char** * str)
- **void** decrypt(**const char** * key, **char** * str)

key est la clé de chiffrage. Elle ne contient que des lettres en majuscule ou en minuscule. Lors du processus de chiffrage/déchiffrage, la casse de la clé sera ignorée (ex : les clés ab et AB doivent conduire au même résultats).

3.1 La méthode

Le principe de la méthode de Vigenère repose sur l'emploi d'une table de correspondance (dite « table de Vigenère », voir plus loin) et d'un mot clef dont la longueur est choisie arbitrairement. Le chiffrage d'un texte est effectué de la manière suivante :

Mot clef : `PERDU`

Texte : `L'ESCARGOT SE PROMENE AVAC SA MAISON`

On affecte, de gauche à droite, une lettre du mot-clef pour chaque lettre du texte. Le mot-clef est répété autant de fois que nécessaire (les caractères non chiffrés ont été supprimés) :

```
L E S C A R G O T S E P R O M E N E A V A C S A M A I S O N
P E R D U P E R D U P E R D U P E R D U P E R D U P E R D U
```

La première lettre du texte à chiffrer est un 'L'. Elle se trouve au dessus de la lettre 'P' du mot-clef. En se reportant au tableau de Vigenère, on voit que l'intersection de la ligne 'L' avec la colonne 'P' est la lettre 'A'. Le 'A' devient la première lettre du message chiffré. On répète cette opération pour toutes les lettres du texte qui se trouve ainsi chiffré (les caractères non chiffrés restend inchangés) :

```
A'IJFUGKFW MT TIRGTRV DPTG JD GPMJRH
```

Pour déchiffrer le message, on procède par la méthode inverse :

```
P E R D U P E R D U P E R D U P E R D U P E R D U P E R D U
A I J F U G K F W M T T I R G T R V D P T G J D G P M J R H
```

Pour la première lettre du message, on prend la colonne 'P' du tableau et on recherche la lettre 'A' en descendant la colonne. La ligne correspondant à la lettre 'A' donne la lettre encodée : 'L'. On déchiffre le reste du message en répétant cette opération pour toutes les lettres du message. La clé est réutilisée de façon cyclique afin d'obtenir une clé suffisament longue.

3.2 Table de Vigenère

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
B	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a
C	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b
D	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c
E	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d
F	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e
G	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f
H	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g
I	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h
J	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i
K	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j
L	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k
M	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l
N	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m
O	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n
P	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
Q	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
R	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
S	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
T	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
U	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
V	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
W	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
X	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
Y	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
Z	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y

3.3 Travail demandé

Pour toutes les questions, veiller à traiter convenablement les erreurs potentielles.

1. Écrire une fonction qui permet la saisie du message à traiter, du mot-clef et d'un indicateur précisant le type d'opération à effectuer.
2. Écrire une fonction permettant d'initialiser la table de correspondance.
3. Écrire la fonction réalisant le chiffrage.
4. Écrire la fonction réalisant le déchiffrage.
5. Écrire une fonction *main* qui assure le chiffrage et le déchiffrage d'un message. Le message initial et le message résultant seront affichés à l'écran.
6. Ré-écrire les fonctions de chiffrage et de déchiffrage sans utiliser la table de Vigenère. La correspondance des caractères sera réalisée par l'intermédiaire d'une fonction mathématique.

7 TD5 : conversion de nombres en chaînes de caractères

Cette série d'exercices devrait vous permettre de compléter par vous même les fonctions suivantes du fichier DocumentUtil.c :

- **char** ∗ computeDocumentNumber(**long** id) retourne une chaîne de caractère allouée sur le tas contenant la représentation en base 36 (0-9A-Z) du nombre paramètre ;
- **char** ∗ formatDate(**int** day, **int** month, **int** year) retourne une chaîne de caractère allouée sur le tas contenant la représentation de la date au format JJ/MM/AAAA (donc en base 10).

1 Conversion base B vers décimale

Soit n un entier positif ou nul qui s'exprime en base B ($B \geq 2$) sous la forme de N chiffres b_0, ..., b_{N-1} ($b_i \in \{0, 1, 2, \ldots, B-1\}$).

b_{N-1}	...	b_5	b_4	b_3	b_2	b_1	b_0

On a :

- $n = \sum_{i=0}^{N-1} b_i B^i$
- Schéma de Horner :

$$n = \sum_{i=0}^{N-1} b_i B^i = b_0 + B[b_1 + B[b_2 + \ldots + B[b_{N-2} + Bb_{N-1}]]]$$

Exemple : 0101 => 1+2(0+2(1+2(0)))=4+1=5

Avertissement

Vous ne devez jamais calculer B^i dans le schéma de Horner.

1. Écrire la fonction BaseB2Dec qui admet trois arguments (la base B, le nombre de chiffres N du tableau, et un tableau de N chiffres) et qui retourne l'entier long n associé. On prendra soin de préciser la convention de représentation utilisée.
2. Écrire la fonction Bin2Dec, utilisant BaseB2Dec, qui retourne la valeur décimale signée correspondant à la représentation binaire sur 16 bits fournie en paramètre.

$$n = \begin{cases} \sum_{i=0}^{14} b_i * 2^i & \text{Si } n \geq 0 \text{ (on a } b_{15} = 0) \\ -2^{15} + \sum_{i=0}^{14} b_i * 2^i & \text{Si } n < 0 \text{ (on a } b_{15} = 1) \end{cases} \tag{7.1}$$

3. Écrire la fonction main

2 Conversion décimale vers binaire

Soit n un entier décimal de type **short**. Soit $b_0, \ldots, b_{15}$ sa forme binaire ($b_i \in \{0,1\}$).

b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

On a :

$$n = \begin{cases} \sum_{i=0}^{14} b_i 2^i & \text{si } n \geq 0 \\ -2^{15} + \sum_{i=0}^{14} b_i 2^i & \text{si } n < 0 \end{cases}$$

Remarques :

- n=1 => **0**000 0001
- n=-1 => 128-1=127 => **1**111 1111
- Soit *div* l'opérateur de division entière et *mod* l'opérateur reste de la division entière, alors pour tout entier n, on a : $n = (n \ div \ k) * k \ + \ (n \ mod \ k)$.
- Schéma de Horner sur un nombre non signé :

$$n = \sum_{i=0}^{N-1} b_i B^i = b_0 + B[b_1 + B[b_2 + \ldots + B[b_{N-2} + Bb_{N-1}]]]$$

1. $b_0 = n \mod B$
2. $b_1 = (n \div B) \mod B$
3. $b_2 = ((n \div B) \div B) \mod B$
4. ...

Exemple : $n = 5 \Longrightarrow$ 0101 en base 2.

Avertissement

Vous ne devez jamais calculer B^i dans le schéma de Horner.

1. Écrire la fonction Dec2Bin qui admet deux paramètres (un nombre entier de type **short** et un tableau de 16 entiers) et qui stocke dans ce tableau la représentation binaire signée de l'entier.
2. Écrire la fonction main qui demande la saisie d'un nombre entier de type short et qui affiche sa représentation binaire.

3 Conversion d'une chaîne de caractères héxadécimale en un entier

On considère la chaîne de caractères s qui représente un entier non signé sous forme hexadécimale. La chaîne s se présente sous une des formes suivantes : "0X1AB97", "0x1AB97", "0x1aB97" ou "0X1Ab97". Écrire la fonction admettant comme argument un pointeur sur la chaîne s et qui renvoie l'entier représenté par la chaîne s. Ecrire la fonction main associée.

4 Conversion d'un entier en une chaîne de caractères

Écrire une fonction qui admet deux paramètres (un entier n et un tableau s de caractères suffisamment grand), et qui écrit dans s la représentation de n sous forme d'une chaîne de caractères. Ecrire la fonction main associée. Lorsque le nombre n est négatif, le signe '-' doit précéder les nombres dans la chaîne s.

Les fonctions de MyString.h peuvent être utilisées.

8 TD6 : Gestion des opérateurs

Cette série d'exercices devrait vous permettre de compléter par vous même les fonctions de OperatorTable.c. Ce module a pour objectif de gérer la liste des utilisateurs et de leurs mots de passe pour le logiciel. Cette liste est stockée dans un fichier texte crypté selon la méthode de Vigenère. En mémoire, la liste des utilisateurs est stockée sous la forme d'un tableau dynamique. Chaque élément du tableau est lui même un tableau de deux chaînes de caractères de taille variable : la première chaîne est le nom de l'utilisateur, la deuxième chaîne est le mot de passe de l'utilisateur. La structure de données associées est la suivante :

```
/** The maximal length in characters of the name of an operator */
#define OPERATORTABLE_MAXNAMESIZE 20UL
/** The maximal length in characters of the password of an operator */
#define OPERATORTABLE_MAXPASSWORDSIZE 20UL

/** The dynamic table of operators */
typedef struct {
    /** The number of operators in the table */
    int recordCount;
    /** The data about the operators. It's a 2D array of strings.
     * Note: records[operatorId][0] is the name of the operatorId'th operator
     * Note: records[operatorId][1] is the password of the operatorId'th operator
     */
    char *** records;
} OperatorTable;
```

Pour simplifier la gestion des E/S sur les fichiers, les noms et mots de passe sont respectivement limités à OPERATORTABLE_MAXNAMESIZE et OPERATORTABLE_MAXPASSWORDSIZE caractères (y compris le marqueur de fin de chaîne).

1 Liste des opérateurs en mémoire

1.1 Création de liste

Ecrire la fonction OperatorTable * OperatorTable_create(**void**) qui retourne un pointeur sur une structure OperatorTable allouée sur le tas et correctement initialisée de manière à représenter une liste vide.

1.2 Obtenir le nombre d'opérateurs

Ecrire la fonction **int** OperatorTable_getRecordCount(OperatorTable * table) qui retourne le nombre d'opérateurs de la liste.

1.3 Obtenir le nom d'un opérateur à partir de la position dans la liste

Ecrire la fonction **const char** * OperatorTable_getName(OperatorTable * table, **int** recordNum) qui retourne le nom du recordNum-ième opérateur de la liste.

1.4 Obtenir le mot de passe d'un opérateur à partir de la position dans la liste

Ecrire la fonction **const char** * OperatorTable_getPassword(OperatorTable * table, **int** recordNum) qui retourne le mot de passe du recordNum-ième opérateur de la liste.

1.5 Recherche d'un opérateur dans la liste

Ecrire la fonction **int** OperatorTable_findOperator(OperatorTable * table, **const char** * name) qui retourne la position de l'opérateur dans la liste associé au nom name ou -1 si l'opérateur n'a pas pu être trouvé. Les comparaisons de noms d'opérateurs sont insensibles à la casse.

1.6 Définir ou modifier le mot de passe d'un opérateur

Ecrire la fonction **int** OperatorTable_setOperator(OperatorTable * table, **const char** * name, **const char** * password) qui permet de définir un opérateur et son mot de passe s'il n'est pas déjà dans la liste ou qui modifie le mot de passe d'un opérateur déjà présent dans la liste. Les comparaisons de noms d'opérateurs sont insensibles à la casse. La valeur retourné est la position de l'utilisateur dans la table des opérateurs.

1.7 Suppression d'un opérateur de la liste

Ecrire la fonction **void** OperatorTable_removeRecord(OperatorTable * table, **int** recordIndex) qui supprime l'opérateur à la position recordIndex de la liste.

1.8 Destruction de liste

Ecrire la fonction **void** OperatorTable_destroy(OperatorTable * table) qui desalloue la mémoire associée à une liste d'opérateurs.

2 Liste des opérateurs dans un fichier texte

2.1 Lecture de la liste des opérateurs

Ecrire la fonction OperatorTable * OperatorTable_loadFromFile(**const char** * filename) qui charge la liste des opérateurs à partir du fichier texte crypté par la méthode de Vigenère.

Pour rappel : fgets () lit au plus size −1 caractères depuis stream et les place dans le tampon pointé par s. La lecture s'arrête après EOF ou un retour-chariot. Si un retour-chariot (newline) est lu, il est placé dans le tampon. Un octet nul 0 est placé à la fin de la ligne.

```
1 char * fgets (char * s, int size, FILE * stream);
```

2.2 Ecriture de la liste des opérateurs

Ecrire la fonction **void** OperatorTable_saveToFile(OperatorTable * table, **const char** * filename) qui écrit la liste des opérateurs dans un fichier texte crypté par la méthode de Vigenère.

9 TD7&TP8 : Gestion des clients

1 Manipulation des enregistrements client

Les exercices suivants doivent vous permettre de remplir le fichier CustomerRecord.c.

Soit les définitions suivantes permettant de déclarer les constantes et types de données associés à un enregistrement client.

```
/** The size of the name field */
#define CUSTOMERRECORD_NAME_SIZE 70UL
/** The size of the address field */
#define CUSTOMERRECORD_ADDRESS_SIZE 130UL
/** The size of the portalCode field */
#define CUSTOMERRECORD_POSTALCODE_SIZE 20UL
/** The size of the town field */
#define CUSTOMERRECORD_TOWN_SIZE 90UL

/** The size in bytes of all the packed fields of a CustomerRecord */
#define CUSTOMERRECORD_SIZE (     CUSTOMERRECORD_NAME_SIZE + \
                                  CUSTOMERRECORD_ADDRESS_SIZE + \
                                  CUSTOMERRECORD_POSTALCODE_SIZE + \
                                  CUSTOMERRECORD_TOWN_SIZE)

/** A customer record */
typedef struct {
    /** The name */
    char name[CUSTOMERRECORD_NAME_SIZE];
    /** The address */
    char address[CUSTOMERRECORD_ADDRESS_SIZE];
    /** The postal code */
    char postalCode[CUSTOMERRECORD_POSTALCODE_SIZE];
    /** The Town */
    char town[CUSTOMERRECORD_TOWN_SIZE];
} CustomerRecord;
```

1.1 Accesseurs

Ecrire les fonctions suivantes qui permettent de modifier le contenu d'un champ d'un enregistrement.

- **void** CustomerRecord_setValue_name(CustomerRecord * record, **const char** * value)
- **void** CustomerRecord_setValue_address(CustomerRecord * record, **const char** * value)
- **void** CustomerRecord_setValue_postalCode(CustomerRecord * record, **const char** * value)
- **void** CustomerRecord_setValue_town(CustomerRecord * record, **const char** * value)

Ecrire les fonctions suivantes qui permettent d'obtenir une copie sur le tas de la valeur d'un champ d'un enregistrement.

- **char** * CustomerRecord_getValue_name(CustomerRecord * record)
- **char** * CustomerRecord_getValue_address(CustomerRecord * record)
- **char** * CustomerRecord_getValue_postalCode(CustomerRecord * record)
- **char** * CustomerRecord_getValue_town(CustomerRecord * record)

1.2 Initialisation

Ecrire la fonction **void** CustomerRecord_init(CustomerRecord * record) qui initialise l'enregistrement fourni en paramètre.

1.3 Finalisation

Ecrire la fonction **void** CustomerRecord_finalize(CustomerRecord * record) qui finalise un enregistrement (libère les ressources qui ne sont plus nécessaires).

1.4 Lecture à partir d'un fichier binaire

Ecrire la fonction **void** CustomerRecord_read(CustomerRecord * record, FILE * file) qui permet de lire le contenu d'un enregistrement à partir du fichier binaire et qui stocke les informations dans l'enregistrement. Le fichier binaire contient que des enregistrements de taille fixe de CUSTOMERRECORD_SIZE octets. On prendra soin de ne pas écrire d'octets de padding dans la fichier.

1.5 Ecriture dans un fichier binaire

Ecrire la fonction **void** CustomerRecord_write(CustomerRecord * record, FILE * file) qui permet d'écrire le contenu d'un enregistrement dans un fichier binaire. Cette fonction est la réciproque de la précédente.

2 Manipulation d'une base de clients

Les exercices suivants doivent vous permettre de remplir le fichier CustomerDB.c.

Soit la structure de données suivante.

```
1 /** The structure which represents an opened customer database */
2 typedef struct {
3     FILE * file; /**< The FILE pointer for the associated file */
4     int recordCount; /**< The number of record in the database */
5 } CustomerDB;
```

Cette structure de données joue un rôle similaire à FILE pour les fonctions d'E/S standard. Chaque fichier client ne peut être manipulé qu'en ayant une structure de ce type à disposition. La structure stocke :

- un attribut file : c'est le lien vers le fichier ouvert ;
- un attribut recordCount : c'est le nombre d'enegistrement stocké dans le fichier client. Il est modifié au fur et à mesure des opérations sur le fichier client mais n'est écrit qu'à la fermeture du fichier.

Un fichier client est un fichier binaire dont le contenu est structuré de la façon suivante :

- le fichier débute par un entier de type **int** stockant le nombre d'enregistrements valides du fichier client ;
- le reste du fichier est constitué des données des enregistrements clients mis bout à bout. Chaque enregistrement client à une taille fixe de CUSTOMERRECORD_SIZE octets.

2.1 Création d'un fichier client

Ecrire la fonction CustomerDB * CustomerDB_create(**const char** * filename) qui crée et ouvre en lecture/écriture un fichier client et retourne une structure permettant de manipuler le contenu du fichier. En cas d'échec d'ouverture du fichier, la fonction renvoie NULL (le fonctionnement est similaire à la fonction fopen). Dans les autres cas d'erreurs, la programme se termine.

2.2 Ouverture d'un fichier client existant

Ecrire la fonction CustomerDB * CustomerDB_open(**const char** * filename) qui ouvre en lecture/écriture un fichier client existant et retourne une structure permettant de manipuler le contenu du fichier. En cas d'échec d'ouverture du fichier, la fonction renvoie NULL (le fonctionnement est similaire à la fonction fopen). Dans les autres cas d'erreurs, la programme se termine.

2.3 Ouverture ou, à défaut, création d'un fichier client

Ecrire la fonction CustomerDB * CustomerDB_openOrCreate(**const char** * filename) qui ouvre en lecture/écriture un fichier client existant ou, si le fichier n'existe pas, crée et ouvre le fichier. La fonction retourne une structure permettant de manipuler le contenu du fichier. Cette fonction est une combinaison des deux fonctions précédentes.

2.4 Fermeture d'un fichier client

Ecrire la fonction **void** CustomerDB_close(CustomerDB * customerDB) qui ferme le fichier client et met à jours le nombre d'enregistrements de la base.

2.5 Obtenir le nombre d'enregistrements clients

Ecrire la fonction **int** CustomerDB_getRecordCount(CustomerDB * customerDB) qui permet d'obtenir le nombre d'enregistrements clients d'un fichier client ouvert.

2.6 Lecture d'un enregistrement

Ecrire la fonction **void** CustomerDB_readRecord(CustomerDB * customerDB, **int** recordIndex, CustomerRecord * record) qui permet de lire le recordIndex-ième enregistrement du fichier client et qui stocke les informations lues dans record.

2.7 Ecriture d'un enregistrement

Ecrire la fonction **void** CustomerDB_writeRecord(CustomerDB * customerDB, **int** recordIndex, CustomerRecord * record) qui permet d'écrire le recordIndex-ième enregistrement du fichier client. Le nombre d'enregistrements du fichier doit être mis à jours si cette écriture conduit à ajouter un enregistrement à la fin du fichier.

2.8 Insertion d'un enregistrement au sein du fichier

Ecrire la fonction **void** CustomerDB_insertRecord(CustomerDB * customerDB, **int** recordIndex, CustomerRecord * record) qui insère le contenu d'un enregistrement à la position recordIndex.

2.9 Ajout d'un enregistrement à la fin

Ecrire la fonction **void** CustomerDB_appendRecord(CustomerDB * customerDB, CustomerRecord * record) qui ajoute un enregistrement à la fin du fichier clients.

2.10 Suppression d'un enregistrement

Ecrire la fonction **void** CustomerDB_removeRecord(CustomerDB * customerDB, **int** recordIndex) qui permet de supprimer l'enregistrement se trouvant à la position recordIndex du fichier client. On ne cherchera pas à redimensionner le fichier.

10 TP9 : Gestion du catalogue des produits

1 Manipulation des produits

Les exercices suivants doivent vous permettre de remplir le fichier CatalogRecord.c.

Soit les définitions suivantes permettant de déclarer les constantes et types de données associés à un produit.

```
/** The size in bytes of the code field of a CatalogRecord */
#define CATALOGRECORD_CODE_SIZE 16UL
/** The size in bytes of the designation field of a CatalogRecord */
#define CATALOGRECORD_DESIGNATION_SIZE 128UL
/** The size in bytes of the unity field of a CatalogRecord */
#define CATALOGRECORD_UNITY_SIZE 20UL
/** The size in bytes of the basePrice field of a CatalogRecord */
#define CATALOGRECORD_BASEPRICE_SIZE ((unsigned long) sizeof(double))
/** The size in bytes of the sellingPrice field of a CatalogRecord */
#define CATALOGRECORD_SELLINGPRICE_SIZE ((unsigned long) sizeof(double))
/** The size in bytes of the rateOfVAT field of a CatalogRecord */
#define CATALOGRECORD_RATEOFVAT_SIZE ((unsigned long) sizeof(double))

/** The size in bytes of all the packed fields of a CatalogRecord */
#define CATALOGRECORD_SIZE (CATALOGRECORD_CODE_SIZE + \
                            CATALOGRECORD_DESIGNATION_SIZE + \
                            CATALOGRECORD_UNITY_SIZE + \
                            CATALOGRECORD_BASEPRICE_SIZE + \
                            CATALOGRECORD_SELLINGPRICE_SIZE + \
                            CATALOGRECORD_RATEOFVAT_SIZE)

/** The maximal length in characters of the string fields of a CatalogRecord */
#define CATALOGRECORD_MAXSTRING_SIZE ( \
      MAXVALUE(CATALOGRECORD_CODE_SIZE, \
      MAXVALUE(CATALOGRECORD_DESIGNATION_SIZE,CATALOGRECORD_UNITY_SIZE)))

/** A catalog record
 */
typedef struct {
    char * code /** The code of the product */;
    char * designation /** The designation of the product */;
    char * unity /** The unity of the product */;
    double basePrice /** The base price of the product (the product should not be sold at a
        ↪ lower price) */;
    double sellingPrice /** The selling price of the product */;
    double rateOfVAT /** The rate of the VAT of the product */;
} CatalogRecord;
```

1.1 Les vérificateurs

Ecrire la fonction **int** CatalogRecord_isValueValid_code(**const char** * value) qui retourne vrai si et seulement si le paramètre ne contient que des chiffres et des lettres.

Ecrire la fonction **int** CatalogRecord_isValueValid_positiveNumber(**const char** * value) qui retourne vrai si et seulement si le paramètre est dans sa totalité un nombre positif. Pour faciliter le travail, on pourra utiliser la fonction strtod.

```
double strtod (const char *nptr, char **endptr);
```

Cette fonction renvoie la valeur convertie si c'est possible. Si endptr n'est pas NULL, un pointeur sur le caractère suivant le dernier caractère converti y est stocké. Si aucune conversion n'est possible, la fonction renvoie zéro, et la valeur de nptr est stockée dans endptr. La documentation complète de cette fonction peut être obtenue en ligne de commande avec :

```
man strtod
```

1.2 Les accesseurs

Ecrire les fonctions suivantes qui permettent de modifier le contenu d'un champ d'un enregistrement.

- **void** CatalogRecord_setValue_code(CatalogRecord * record, **const char** * value)
- **void** CatalogRecord_setValue_designation(CatalogRecord * record, **const char** * value)
- **void** CatalogRecord_setValue_unity(CatalogRecord * record, **const char** * value
- **void** CatalogRecord_setValue_basePrice(CatalogRecord * record, **const char** * value)
- **void** CatalogRecord_setValue_sellingPrice(CatalogRecord * record, **const char** * value)
- **void** CatalogRecord_setValue_rateOfVAT(CatalogRecord * record, **const char** * value)

Ecrire les fonctions suivantes qui permettent d'obtenir une copie sur le tas de la valeur d'un champ d'un enregistrement sous forme d'une chaîne de caractères.

- **char** * CatalogRecord_getValue_code(CatalogRecord * record)
- **char** * CatalogRecord_getValue_designation(CatalogRecord * record)
- **char** * CatalogRecord_getValue_unity(CatalogRecord * record)
- **char** * CatalogRecord_getValue_basePrice(CatalogRecord * record)
- **char** * CatalogRecord_getValue_sellingPrice(CatalogRecord * record)
- **char** * CatalogRecord_getValue_rateOfVAT(CatalogRecord * record)

1.3 Initialisation

Ecrire la fonction **void** CatalogRecord_init(CatalogRecord * record) qui initialise l'enregistrement fourni en paramètre.

1.4 Finalisation

Ecrire la fonction **void** CatalogRecord_finalize(CatalogRecord * record) qui finalise un enregistrement (libère les ressources qui ne sont plus nécessaires).

1.5 Lecture à partir d'un fichier binaire

Ecrire la fonction **void** CatalogRecord_read(CatalogRecord * record, FILE * file) qui permet de lire le contenu d'un enregistrement à partir du fichier binaire et qui stocke les informations dans l'enregistrement. Le fichier binaire contient que des enregistrements de taille fixe de CATALOGRECORD_SIZE octets. On prendra soin de ne pas écrire d'octets de padding dans la fichier.

1.6 Ecriture dans un fichier binaire

Ecrire la fonction **void** CatalogRecord_write(CatalogRecord * record, FILE * file) qui permet d'écrire le contenu d'un enregistrement dans un fichier binaire. Cette fonction est la réciproque de la précédente.

2 Manipulation d'un catalogue de produits

Les exercices suivants doivent vous permettre de remplir le fichier CatalogDB.c.

Soit la structure de données suivante.

```
1 /** The structure which represents an opened catalog database */
2 typedef struct _CatalogDB {
3     FILE * file; /**< The FILE pointer for the associated file */
4     int recordCount; /**< The number of record in the database */
5 } CatalogDB;
```

Cette structure de données joue un rôle similaire à FILE pour les fonctions d'E/S standards. Chaque fichier catalogue ne peut être manipulé qu'en ayant une structure de ce type à disposition. La structure stocke :

- un attribut file : c'est le lien vers le fichier ouvert ;
- un attribut recordCount : c'est le nombre d'enregistrement stocké dans le fichier client. Il est modifié au fur et à mesure des opérations sur le fichier client mais n'est écrit qu'à la fermeture du fichier.

Un fichier catalogue est un fichier binaire dont le contenu est structuré de la façon suivante :

- le fichier début par un entier de type **int** stockant le nombre d'enregistrement valide du fichier catalogue ;
- le reste du fichier est constitué des données des enregistrements des produits mis bout à bout. Chaque enregistrement à une taille fixe de CATALOGRECORD_SIZE octets.

2.1 Création d'un fichier catalogue

Ecrire la fonction CatalogDB * CatalogDB_create(**const char** * filename) qui crée et ouvre en lecture/écriture un fichier catalogue et retourne une structure permettant de manipuler le contenu du fichier. En cas d'échec d'ouverture du fichier, la fonction renvoie NULL (le fonctionnement est similaire à la fonction fopen). Dans les autres cas d'erreurs, la programme se termine.

2.2 Ouverture d'un fichier catalogue existant

Ecrire la fonction CatalogDB * CatalogDB_open(**const char** * filename) qui ouvre en lecture/écriture un fichier catalogue existant et retourne une structure permettant de manipuler le contenu du fichier. En cas d'échec d'ouverture du fichier, la fonction renvoie NULL (le fonctionnement est similaire à la fonction fopen). Dans les autres cas d'erreurs, la programme se termine.

2.3 Ouverture ou, à défaut, création d'un fichier catalogue

Ecrire la fonction CatalogDB * CatalogDB_openOrCreate(**const char** * filename) qui ouvre en lecture/écriture un fichier catalogue existant ou, si le fichier n'existe pas, crée et ouvre le fichier. La fonction retourne une structure permettant de manipuler le contenu du fichier. La fonction retourne une structure permettant de manipuler le contenu du fichier. Cette fonction est une combinaison des deux fonctions précédentes.

2.4 Fermeture d'un fichier catalogue

Ecrire la fonction **void** CatalogDB_close(CatalogDB * catalogDB) qui ferme le fichier catalogue.

2.5 Obtenir le nombre d'enregistrements du catalogue

Ecrire la fonction **int** CatalogDB_getRecordCount(CatalogDB * catalogDB) qui permet d'obtenir le nombre d'enregistrements d'un fichier catalogue ouvert.

2.6 Lecture d'un enregistrement

Ecrire la fonction **void** CatalogDB_readRecord(CatalogDB * catalogDB, **int** recordIndex, CatalogRecord * ↪ record) qui permet de lire le recordIndex-ième enregistrement du fichier catalogue et qui stocke les informations lues dans record.

2.7 Ecriture d'un enregistrement

Ecrire la fonction **void** CatalogDB_writeRecord(CatalogDB* catalogDB, **int** recordIndex, CatalogRecord * ↪ record) qui permet d'écrire le recordIndex-ième enregistrement du fichier catalogue. Le nombre d'enregistrements du fichier doit être mis à jours si cette écriture conduit à ajouter un enregistrement à la fin du fichier.

2.8 Insertion d'un enregistrement au sein du fichier

Ecrire la fonction **void** CatalogDB_insertRecord(CatalogDB* catalogDB,**int** recordIndex, CatalogRecord * ↪ record) qui insère le contenu d'un enregistrement à la position recordIndex.

2.9 Ajout d'un enregistrement à la fin

Ecrire la fonction **void** CatalogDB_appendRecord(CatalogDB * catalogDB, CatalogRecord *record) qui ajoute un enregistrement à la fin du fichier clients.

2.10 Suppression d'un enregistrement

Ecrire la fonction **void** CatalogDB_removeRecord(CatalogDB * catalogDB, **int** recordIndex) qui permet de supprimer l'enregistrement se trouvant à la position recordIndex du fichier catalogue. On ne cherchera pas à redimensionner le fichier.

11 TD10&TP11 : Manipulation d'un document

Cette série d'exercices devrait vous permettre de compléter par vous même les fonctions de DocumentUtil.c, DocumentRowList.c et Document.c.

1 Fonctions génériques

Pour les deux fonctions suivantes, on considère des fichiers binaires.

Ecrire la fonction **void** writeString(**const char** * str, FILE * file) qui écrit dans un fichier binaire la chaîne de caractères fournie en paramètre. La longueur de la chaîne est quelconque.

Ecrire la fonction **char** * readString(FILE * file) qui lit dans un fichier binaire une chaîne de caractères précédemment écrite via la fonction writeString. La chaîne de caractères retournée est allouée sur le tas.

2 La liste des produits du document

La liste des produits d'un document est stockée sous la forme d'une liste chainée simple. Pour cela, on vous fournit la structure de données suivante :

```
/** Structure representing a row in a document (as a cell in a simple linked list) */
typedef struct _DocumentRow {
    char * code /** The code */;
    char * designation /** The designation */;
    double quantity /** The quantity */;
    char * unity /** The unity */;
    double basePrice /** The base price */;
    double sellingPrice /** The selling price */;
    double discount /** The discount */;
    double rateOfVAT /** The rate of VAT */;
    struct _DocumentRow * next /** The pointer to the next row */;
} DocumentRow;
```

Lorsque plusieurs cellules sont chainées en mémoire, on obtient une organisation telle que :

2.1 Manipulation des cellules

2.1.1 Initialisation d'une cellule de la liste

Ecrire la fonction **void** DocumentRow_init(DocumentRow * row) qui initialise tous les champs d'une cellule à des valeurs « raisonnable » de manière à représenter une ligne de produit vide.

2.1.2 Finalisation d'une cellule

Ecrire la fonction **void** DocumentRow_finalize(DocumentRow * row) qui finalise une cellule en libérant toute la mémoire non nécessaire à la cellule.

2.1.3 Allocation sur le tas et initialisation d'une cellule

Ecrire la fonction DocumentRow * DocumentRow_create(**void**) qui alloue une cellule sur le tas, l'initialise et la retourne.

2.1.4 Destruction d'une cellule allouée sur le tas

Ecrire la fonction **void** DocumentRow_destroy(DocumentRow * row) qui desalloue une cellule précédement allouée par DocumentRow_create.

2.1.5 Ecriture d'une cellule dans un fichier binaire

En utilisant la fonction writeString, écrire la fonction **void** DocumentRow_writeRow(DocumentRow * row, FILE * file) qui écrit le contenu d'une cellule dans un fichier binaire.

2.1.6 Lecture d'une cellule à partir d'un fichier binaire

En utilisant la fonction readString, écrire la fonction DocumentRow * DocumentRow_readRow(FILE * fichier) qui retourne une cellule allouée sur le tas dont le contenu est lu à partir du fichier binaire.

2.2 Manipulation de la liste

2.2.1 Initialisation d'une liste

Une liste chainée est assimilable à une tête de liste et aux cellules composants la liste. Ecrire la fonction **void** DocumentRowList_init(DocumentRow ** list) qui initialise une liste comme étant vide.

2.2.2 Destruction de liste

Ecrire la fonction **void** DocumentRowList_finalize(DocumentRow ** list) qui détuit une liste en détruisant les cellules de la liste. A la fin de la fonction, la liste doit être une liste valide mais vide.

2.2.3 Accès au n-ième élément de la liste

Ecrire la fonction DocumentRow * DocumentRowList_get(DocumentRow * list, **int** rowIndex) qui retourne un pointeur sur le rowIndex-ième élément de la liste. Si cet élément n'existe pas, la liste renvoie NULL.

2.2.4 Nombre d'éléments d'une liste

Ecrire la fonction **int** DocumentRowList_getRowCount(DocumentRow * list) qui retoune le nombre d'éléments d'une liste.

2.2.5 Ajout en fin de liste

Ecrire la fonction **void** DocumentRowList_pushBack(DocumentRow ** list, DocumentRow * row) qui ajoute une cellule à la fin de la liste.

2.2.6 Insertion après une cellule

Ecrire la fonction DocumentRowList_insertAfter qui insère une cellule après la position indiquée dans une liste.

```
1 void DocumentRowList_insertAfter(DocumentRow ** list, DocumentRow * position, DocumentRow *
      ↪ row);
```

2.2.7 Insertion avant une cellule

Ecrire la fonction DocumentRowList_insertBefore qui insère une cellule avant la position indiquée dans une liste.

```
1 void DocumentRowList_insertBefore(DocumentRow ** list, DocumentRow* position, DocumentRow* row);
```

2.2.8 Suppression d'une cellule

Ecrire la fonction **void** DocumentRowList_removeRow(DocumentRow ** list, DocumentRow * position) qui supprime la cellule indiquée d'une liste.

3 Le document

Un document est défini grâce aux structures de données suivantes :

```
/** Enumeration defining the type of a document */
typedef enum {
    QUOTATION /**< It's a quotation */,
    BILL /**< It's a bill */
} TypeDocument;

/** Structure representing a document */
typedef struct {
    CustomerRecord customer /** The customer */;
    char * editDate /** The last edit data */;
    char * expiryDate /** The peremption date */;
    char * docNumber /** The document number */;
    char * object /** The object of the document */;
    char * operator /** The last operator */;
    DocumentRow * rows /** The rows */;
    TypeDocument typeDocument /** The type of document */;
} Document;
```

3.1 Initialisation

Ecrire la fonction **void** Document_init(Document * document) qui initialise un document de manière à ce qu'il soit vierge.

3.2 Finalisation

Ecrire la fonction **void** Document_finalize(Document * document) qui finalise un document en libérant les éventuelles zones mémoires allouées.

3.3 Ecriture dans un fichier

Ecrire la fonction **void** Document_saveToFile(Document * document, **const char** * filename) qui écrit le contenu d'un document dans un fichier binaire. Vous devez au maximum utiliser les fonctions définies précédement.

3.4 Lecture à partir d'un fichier binaire

Ecrire la fonction **void** Document_loadFromFile(Document * document, **const char** * filename) qui lit le contenu d'un document à partir d'un fichier. Le document, fournis en paramètre, à remplir a été précédemment initialisé par la fonction Document_init.

12 TD12&TP13 : Aperçu avant impression

Cette série d'exercices devrait vous permettre de compléter par vous même les fonctions de Dictionary.c et PrintFormat.c.

1 Principes

1.1 Objectif

L'objectif de ces exercices est de mettre en place un système de mise en forme de documents pour l'impression (non implémentée). Le système consiste, à partir d'un modèle de document et d'informations, à générer un document texte. Un modèle est donné au format texte et contient des balises permettant d'indiquer les informations à insérer et les éventuels traitements à effectuer dessus. Le modèle sera stocké dans une structure de données PrintFormat tandis que les informations seront stockées dans des dictionnaires de données représentés par la structure Dictionary.

1.2 Dictionnaire et modèles

Un dictionnaire contient un ensemble de couple nom de variable/valeur. Les valeurs peuvent être des chaînes de caractères ou des réels de type **double**.

Les balises dans un modèle sont délimités par le caractère %. A l'intérieur d'une balise, on trouve obligatoirement un nom de variable (insensible à la casse) et d'éventuelles modifications à apporter à la valeur substituée entre accolades. Les modifications prévues sont limitées et dépendent du type de la variable :

- Si la variable est une chaîne de caractères, les modificateurs possibles sont :
  - **case** pour modifier la casse : U ou u pour transformer en majuscule, une autre valeur pour transformer en minuscule ;
  - min pour spécifier la longeur minimale de la chaîne de substitution. Si la chaîne à substituer est trop courte, des espaces sont ajoutés à la fin ;
  - max pour spécifier la longuer maximale de la chaîne de substitution. Si la chaîne à substituer est trop longue, elle est tronquée.
- Si la variable est un nombre réel, les modificateurs possibles sont :
  - precision pour spécifier la précision du nombre réel. Il s'agit du nombre de chiffre après la virgule. Si la précision est nulle, le séparateur de décimale ne doit pas faire partie de la chaîne de substitution.
  - min pour spécifier la largeur minimale de la chaîne de substitution. Si la chaîne à substituer est trop courte, des espaces sont ajoutés au début.

Si plusieurs modificateurs identiques sont présents pour une substitution, on supposera que seul la première occurrence du modificateur est prise en compte. Si deux caractères % se suivent dans le texte alors ils sont remplacés par un seul % (séquence d'échappement).

1.3 Exemple

Supposons que le dictionnaire suivant soit utilisée pour formatter un modèle :

Nom	Type	Valeur
var1	nombre	10.2
var2	chaîne	"abcDef"

Après formattage, on doit obtenir les résultats suivants :

```
Chaine de format                                Chaine de substitution
"%%"                                              "%"
"%VAR1{precision=0}%"                             "10"
"%VAR1{precision=0}% %VAR1{precision=0}%"         "10 10"
"%VAR1{precision=2}%"                             "10,20"
"%VAR1{precision=2,min=10}%"                      "      10,20"
"%VAR2%"                                          "abcDef"
"%VAR2{max=3}%"                                   "abc"
"%VAR2{max=10}%"                                  "abcDef"
"%VAR2{min=8}%"                                   "abcDef  "
"%VAR2{case=U}%"                                  "ABCDEF"
"%VAR2{case=l}%"                                  "abcdef"
"%VAR2{case=U,max=4}%"                            "ABCD"
```

1.4 Strutures de données

Un dictionnaire est défini par les structures suivantes :

```
/** Enumeration defining the type of the doctionary entries */
typedef enum {
    UNDEFINED_ENTRY /* It's undefined */,
    NUMBER_ENTRY /** It's a number */,
    STRING_ENTRY /** It's a string */
} DictionaryEntryType;

/** Structure representing an entry in the dictionary */
typedef struct {
    /** The type of entry */
    DictionaryEntryType type;
    /** The name of the entry */
    char * name;
    /** The union which store the value of the entry */
    union {
        /** The value of the entry when it's a string */
        char * stringValue;
        /** The value of the entry when it's a real number */
        double numberValue;
    } value;
} DictionaryEntry;

/** Structure representing a dictionary */
typedef struct _Dictionary {
    /** The number of entries of the dictionary */
    int count;
    /** The table of entries */
    DictionaryEntry * entries;
} Dictionary;
```

Un dictionnaire est donc assimilable à un tableau dynamique.

Un modèle est défini par la structure suivante :

```
/** Structure holding the three format strings defining a model */
typedef struct _PrintFormat {
    /** The name of the model */
    char * name;
    /** The header format */
    char * header;
    /** The row format */
    char * row;
    /** The footer format */
    char * footer;
} PrintFormat;
```

Chaque modèle est décomposé en trois parties :

- le modèle de l'entête : utilisée pour formatter le début du document ;
- le modèle de ligne : utilisée pour formatter une ligne produit du document ;
- le modèle de pied de page : utilisée pour formatter la fin du document.

Ce modèle est simplifié et ne gère donc pas la notion délicate de découpage en pages.

2 Manipulation du dictionnaire

2.1 Création d'un dictionnaire

Ecrire la fonction Dictionary * Dictionary_create(**void**) qui crée sur le tas un nouveau dictionnaire.

2.2 Recherche de variables

Ecrire la fonction Dictionary_getEntry qui retourne un pointeur sur l'entrée du dictionnaire correspondant à la variable indiquée. La mise en correspondance entre les noms est insensible à la casse.

```
1 DictionaryEntry * Dictionary_getEntry(Dictionary * dictionary, const char * name);
```

2.3 Définition d'une variable du type chaîne de caractères

Ecrire la fonction Dictionary_setStringEntry qui permet de définir ou modifier une variable du type chaîne de caractères. On prendra soin de libérer les éventuelles zones mémoires inutiles. Attention, lors d'une modification, une variable peut changer de type.

```
1 void Dictionary_setStringEntry(Dictionary * dictionary, const char * name, const char * value);
```

2.4 Définition d'une variable du type nombre réel

Ecrire la fonction Dictionary_setNumberEntry qui permet de définir ou modifier une variable du type nombre réel. On prendra soin de libérer les éventuelles zones mémoires inutiles. Attention, lors d'une modification, une variable peut changer de type.

```
1 void Dictionary_setNumberEntry(Dictionary * dictionary, const char * name, double value);
```

2.5 Destruction d'un dictionnaire

Ecrire la fonction **void** Dictionary_destroy(Dictionary * dictionary) qui détruit et desalloue un dictionnaire et son contenu.

3 Manipulation du modèle

3.1 Création d'un modèle

Ecrire la fonction **void** PrintFormat_init(PrintFormat * format) qui permet de crée un modèle vide sur le tas.

3.2 Destruction d'un modèle

Ecrire la fonction **void** PrintFormat_finalize(PrintFormat * format) qui permet de détruire un modèle ayant été crée sur le tas en libérant les éventuelles zones mémoires inutiles.

3.3 Chargement d'un modèle

Tous les modèles se présentent sous la forme d'un fichier texte dont le contenu est structurée de la façon suivante :

```
1  .NAME Ici il y a le nom du modele
2  .HEADER
3  Ici commence l'entete
4
5  Il peut y avoir plusieurs lignes
6       et meme des blancs
7  Ici se termine l'entete
8  .ROW
9  Ici commence le modele d'une ligne en generale sur une seule ligne
10 mais rien ne l'oblige
11 .FOOTER
12 Ici commence le pied de page
13 qui peut aussi comporter plusieurs lignes
14 .END
15 Il peut y avoir du texte ici mais on l'ignore
```

L'intégralité du texte, y compris les sauts de lignes, entre deux marqueurs fait partie du modèle. Ainsi, le modèle pour une ligne de produits comporte forcément un saut de ligne à la fin. Une ligne de texte d'un modèle peut être de longueur quelconque.

Ecrire la fonction **void** PrintFormat_loadFromFile(PrintFormat* format, **const char*** filename) qui charge un modèle à partir d'un fichier texte. Le modèle fournit en paramètre a déjà été crée sur le tas. Pour cela, on vous conseille d'écrire d'abord la fonction **static char** * readLine(FILE * fichier) qui lit un ligne entière dans le fichier, quelque soit sa longueur, et qui retourne la ligne comme une chaîne de caractères allouées sur le tas.

```
1 char * fgets (char * s, int size, FILE * stream);
```

fgets() lit au plus size−1 caractères depuis stream et les place dans le tampon pointé par s. La lecture s'arrête après EOF ou un retour-chariot. Si un retour-chariot (newline) est lu, il est placé dans le tampon. Un octet nul 0 est placé à la fin de la ligne. fgets() renvoie le pointeur s si elle réussit, et NULL en cas d'erreur, ou si la fin de fichier est atteinte avant d'avoir pu lire au moins un caractère.

4 Formattage d'un modèle à partir d'un dictionnaire

La création d'un document à partir d'un modèle et d'un dictionnaire suit l'algorithme suivant :

while *on n'a pas parcouru tout le modèle* **do**
    Chercher le début d'une balise;
    Copier le texte précédant la balise;
    if *c'est un double '%'* **then**
        Ajouter un '%' dans le texte
    else
        Chercher la fin de la balise;
        Extraire le nom de la balise et ses parametres;
        Rechercher la valeur associee au nom de la balise;
        Formater la valeur trouvee selon les parametres de la balise;
        Ajouter la valeur formatee a la suite du texte;
Copier le texte jusqu'à la fin du modèle;

En suivant ou non cet algorithme, écrire la fonction **char** $*$ Dictionary_format(Dictionary $*$ dictionary, ✂ ↪ **const char** $*$ format) qui retourne une chaîne de caractères allouée sur le tas contenant le résultat de la mise en forme de la chaîne de format à partir du dictionnaire. Si une variable à substituer n'est pas présente dans le dictionnaire, on affichera un message d'avertissement sur la sortie d'erreur et la substitution de la balise se fera avec la chaîne vide sans aucun formattage supplémentaire.