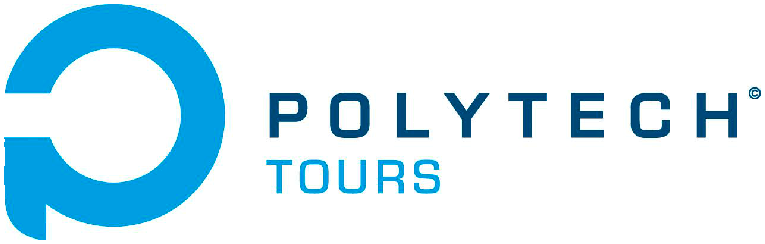
Polytech'Tours
Département Informatique
64 Avenue Jean Portalis
37200 Tours



Programmation en langage C
TDs et TPs guidés

Sébastien Aupetit
sebastien.aupetit@univ-tours.fr

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 14h TDs et 12h TPs sont réparties de la façon suivante :

TP 1, TP 1 bis	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.

2 TP1 : Utilisation d'un IDE et d'un compilateur

1 Code::Blocks

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

1. Créer un programme
   - Créer un projet par programme. Le programme que vous réalisez est une application console et le compilateur est GCC.
   - Saisir le code source du programme avec indentation et mise en forme correcte du code (cf. cours). Chercher à identifier la convention de nommage et d'indentation qui vous convient le mieux.
   - Compiler le programme avec le compilateur, exécuter le programme. Effectuer ces deux étapes en choisissant le type de compilation : Debug et Release.
2. Configugation
   - Configurer le projet de manière à activer les options **−−std=c89**, **−Wall** et **−Wstrict−prototypes**.
   - Comment peut-on connaître la liste des options du compilateur permettant d'activer les avertissements ?
   - 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. Exécuter votre programme en mode Debug et en mode Release. Que constate-t-on ?
3. Déboggage
   - Démarrer une session de débuggage
   - Effectuer un pas à pas dans le programme (entrer dans une fonction, en sortir, executer une fonction sans entrer dedans...)
   - Inspecter des variables et leurs contenus
   - Inspecter la pile d'appel
   - Définir et utiliser des points d'arrêt pour le débuggage
4. Inspecter les différents menus du compilateur et notamment la configuration du projet.

L'environnement de développment intégré (IDE) **Code::Blocks** est disponible gratuitement sur les systèmes d'exploitation MS-Windows, GNU/Linux et Mac OS X. Il s'agit donc de l'IDE obligatoire pour les TPs de langage C.

2 Exercice de 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.

2.1 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.

2.2 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.

2.3 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**.

```
Entree :
  tab : le tableau de nombres
  MAX : type entier ; nombre d'elements du tableau
Preconditions :
  MAX ≥ 0
Sortie :
  rien
Postconditions :
  le tableau est trie en ordre croissant

Variables locales :
    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).

Corps :
  non_trie ← Vrai /* le tableau n'est pas trie */
  i = 0;
  Tant que (i ≤ MAX-1) et (non_trie est Vrai) Faire
  Debut
    non_trie ← Faux
    Pour j = 1 a i = MAX-i Faire
    Debut
      Si (tab[j] < tab[j-1]) Alors
      Debut
        tmp ← tab[j-1]
        tab[j-1] ← tab[j]
        tab[j] ← tmp
        non_trie ← Vrai
      Fin
    Fin
  Fin
```

2.4 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 au début du sujet. Vous devez réaliser vos tests d'exécutions en mode Débug et en mode Release.

2.4.1 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.

2.4.2 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
....
```

2.4.3 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.

2.4.4 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.

2.5 Mise en oeuvre : Version 2

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

3 Autres compilateurs

Sur votre temps libre, vous pouvez réaliser le même travail :
- avec Visual Studio (6 ou 2003/2005/2008) sous MS-Windows
- avec Kdevelop sous GNU/Linux :
  - Pour pouvoir déboguer des programmes effectuant des saisies au clavier (scanf, gets, ...), il est nécessaire d'activer l'option suivante : **Option du projet > Débogueur > Enable separate terminal for application IO**.
  - Pour pouvoir utiliser des fonctions mathématiques (pow, sqrt, ...) dans votre programme, il est nécessaire d'ajouter à l'option **Option du projet > Option de configure > Drapeaux de l'éditeur de liens (LDFLAGS)** le texte **−lm**
- avec Eclipse sous GNU/Linux
- avec Anjuta sout GNU/Linux
- ...

3 TP1 bis : Prise en main du projet des TDs/TPs

1 Objectifs du projet

Dans le cadre des TDs/TPs de langage C, on vous propose 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 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).

3.1 Manipulation de chaînes de caractères

Fichiers : **MyString.h** et **MyString.c**

Rôles :

- Remplacement des fonctions standards de manipulation de chaînes de caractères
- Ajout de fonctions de manipulation de chaînes de caractères

Tests unitaires : **MyStringUnit.h** et **MyStringUnit.c**

3.2 Gestion des opérateurs

Fichiers : **OperatorTable.h** et **OperatorTable.c**

Rôles :

- chargement et sauvegarde de la liste des opérateurs et de leurs mots de passe
- ajout, suppression et modification d'un opérateur

Tests unitaires : **OperatorTableUnit.h** et **OperatorTableUnit.c**

3.3 Cryptage et décryptage du fichier des opérateurs

Fichiers : **EncryptDecrypt.h** et **EncryptDecrypt.c**

Rôles :

- cryptage et décryptage en fonction d'une clé de cryptage

Tests unitaires : **EncryptDecryptUnit.h** et **EncryptDecrypt.c**

3.4 Gestion du catalogue des produits

Fichiers : **CatalogRecord.h**, **CatalogRecord.c**, **CatalogDB.h**, **CatalogDB.c**, **GtkCatalogModel.h**, **GtkCatalogModel.c**, **Catalog.h** et **Catalog.c**

Rôles :

- Manipulation et édition d'un produit
- Manipulation du catalogue des produits

Tests unitaires : **CatalogRecordUnit.h**, **CatalogRecordUnit.c**, **CatalogDBUnit.h** et **CatalogDBUnit.c**

3.5 Gestion du fichier client

Fichiers : **CustomerRecord.h**, **CustomerRecord.c**, **CustomerDB.h**, **CustomerDB.c**, **GtkCustomerModel.h**, **GtkCustomerModel.c,** **lstinline**Customer.h| et **Customer.c**

Rôles :

- Manipulation et édition d'un client
- Manipulation du fichier client

Tests unitaires : **CustomerRecordUnit.h**, **CustomerRecordUnit.c**, **CustomerDBUnit.h** et **CustomerDBUnit.c**

3.6 Fonctions utilitaires pour la manipulation de documents

Fichiers : **DocumentUtil.h** et **DocumentUtil.c**

Rôles :

- Calcule du numéro de document
- Formattage de la date
- Lecture et écriture de chaînes de caractères de taille variable dans un fichier binaire

Tests unitaires : **DocumentUtilUnit.h** et **DocumentUtilUnit.c**

3.7 Gestion des documents

Fichiers : **DocumentRowList.h**, **DocumentRowList.c**, **Document.h**, **Document.c**, **Quotation.h**, **Quotation.c**, **Bill.h**, **Bill.c**, **DocumentEditor.h**, **DocumentEditor.c**

Rôles :

- Manipulation d'un devis ou d'une facture
- Manipulation du contenu d'un document
- Edition et visualisation du contenu d'un document

Tests unitaires : **DocumentRowListUnit.h**, **DocumentRowListUnit.c**, **DocumentUnit.h** et **DocumentUnit.c**

3.8 Gestion d'aperçu avant impression et formattage de la mise en forme d'un document

Fichiers : **Print.h**, **Print.c**, **PrintFormat.h**, **PrintFormat.c**, **Dictionary.h** et **Dictionary.c**

Rôles :

- Visualisation de l'aperçu
- Chargement d'un modèle de format
- Formattage d'un document selon un modèle

Tests unitaires : **DictionaryUnit.h**, **DictionaryUnit.c**, **PrintFormat.h** et **PrintFormat.c**

3.9 Gestion de l'application

Fichiers : **main.c**, **App.h**, **App.c**, **MainWindow.h**, **MainWindow.c**

3.10 Fonctions pour les tests unitaires

Fichiers : **UnitTest.h**

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 fournis 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**).

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 une erreur de compilation. Vous devez vous assurer de ne rien laisser au hasard y compris les convertions. 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.

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 [1]. Pour cela, il vous suffit d'exécuter le script **./TestValgrind.sh** et d'étudier les éventuelles problèmes soulevés par **valgrind** (fuites mémoires...).

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**.

6 Structuration des fichiers d'entêtes

Les fichiers d'entêtes sont structurés selon différents paquets :
- **base/*.h** : ces fichiers définissent principalement les structures de données que l'on va manipuler ;
- **provided/*.h** : ces fichiers déclares les prototypes des fonctions pré-implémentées ;
- **bridge/*.h** : à usage interne, permettent aux fonctions pré-implémentées d'utiliser votre implémentation dès que vous avez implémenté quelque chose ;
- ***.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).

7 Configuration

7.1 Sous Linux en salle Unix

Pour utiliser le projet fourni sous Linux en salle Unix, vous n'avez rien de particulier à faire si ce n'est d'exécuter la commande **make** dans le répertoire racine du projet.

Si vous avez effectué correctement les démarches décrites ci-dessus, vous devez avoir deux nouveaux répertoires qui sont apparus : **debug−build** et **release−build**. Ces répertoires contiennent respectivement les fichiers projets pour la version Debug et la version Release de votre programme. Dans ces répertoires vous trouverez des fichiers projets pour **Code::Blocks**, Eclipse et un Makefile. **Dans le cadre de ces TPs vous devez utiliser Code::Blocks.**

1. Disponible uniquement sous Linux

Etant donné que les fichiers projets sont générés automatiquement, vous ne devez pas les modifier (pas d'ajout de fichiers, pas de changement de configuration du projet...). Pour réaliser les TPs, aucun de ces changements ne sont nécessaires.

Sous **Code::Blocks**, n'oubliez pas de sélectionner l'option *facturation* dans le menu *Build* puis *Select target* (ou via la barre d'icones *Build target*).

Modifiez le fichier **main.c** pour y mettre votre nom, votre groupe et l'année.

7.2 Autres cas

⚠ Mise en garde

- Aucune aide ne sera donnée sur les autres possibilités suivantes.
- Ces autres possibilités peuvent ne pas fonctionner, induire des bugs ou vous empêcher de réaliser toutes les démarches demandées pour vérifier votre code.
- Ces possibilités n'ont pas été testées intensivement.
- Si vous ne comprennez pas ce qui est décrit ci-dessous, utilisez la section précédente.

En fonction du système sous lequel vous utilisez le projet, les démarches a effectuer sont différentes :

7.2.1 Linux hors des salles unix

- Installer les programmes et bibliothèques suivantes :
  - **cmake** version 2.8.1 ou supérieur
  - **Code::Blocks**
  - la bibliothèque GTK+ version 2.20 ou plus récent avec les fichiers de développements
  - **valgrind**
- Suivre les étapes de *Linux des salles Unix*

7.2.2 Windows (XP, Vista, Seven)

- Le programme **valgrind** n'est disponible que sous Linux. Vous ne pourrez donc pas effectuer l'intégralité des vérifications.
- Installer les programmes suivant en suivants leurs modes d'emploi :
  - **cmake** version 2.8.1 ou supérieur en configurant la variable d'environnement PATH
  - **Code::Blocks** avec le compilteur **MinGW**
  - la bibliothèque GTK+ version 2.20 ou plus récent avec les fichiers de développements et configurer la variable d'environnement PATH
- Modifier au besoin la variable **MINGWPATH** du fichier **configure.bat** si nécessaire
- Executer le script **configure.bat** dans le répertoire racine du projet.
- Sous **Code::Blocks**, aller dans le menu *Settings*, puis *Compiler and debugger*. Dans l'onglet *Toolchain executables*, sur la ligne *Make program*, remplacer **make.exe** par **mingw32−make.exe**.

4 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"**
- **"b"**
- **"aa"**
- **"aaz"**
- **"ab"**
- **"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 **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.

```
char * str = "abcdef";
char * s1 = subString(str,str);
char * s2 = subString(str,str+strlen(str));
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 **char * insertString(const char * src, int insertPosition, const char * toBeInserted, int insertLength)** 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**.

```
const char * src = "abcghi";
const char * toBeInserted = "def";

temp1 = insertString(src, 3, toBeInserted, 3);
/* temp1 doit correspondre a "abcdefghi" */

temp2 = insertString(src, 3, toBeInserted, 2);
/* temp2 doit correspondre a "abcdeghi" */

temp3 = insertString(src, 0, toBeInserted, 2);
/* temp3 doit correspondre a "deabcghi" */

temp4 = insertString(src, 6, toBeInserted, 2);
/* 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.

5 TD5 : conversion de nombres en chaînes de caractères

Cette série d'exercices devrait vous permettre de compléter par vous même les fonctions suivantes de **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 $b_0, \ldots, b_{N-1}$ ($b_i \in \{0, 1, 2, \ldots, B-1\}$).

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

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.

$$n = \begin{cases} \sum_{i=0}^{14} b_i * 2^i & \text{Si } n \geq 0 \text{ (on a } b_{15} = 0) \\ -2^{15} + \sum_{i=0}^{14} b_i * 2^i & \text{Si } n < 0 \text{ (on a } b_{15} = 1) \end{cases} \tag{5.1}$$

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. $b_1 = (n \ div \ B) \ modB$
3. $b_2 = (n \ div \ B^2) \ modB$
4. ...

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

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.

6 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 20
/** The maximal length in characters of the password of an operator */
#define OPERATORTABLE_MAXPASSWORDSIZE 20

/** 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()** 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.

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.

7 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 70
/** The size of the address field */
#define CUSTOMERRECORD_ADDRESS_SIZE 130
/** The size of the portalCode field */
#define CUSTOMERRECORD_POSTALCODE_SIZE 20
/** The size of the town field */
#define CUSTOMERRECORD_TOWN_SIZE 90

/** The size in bytes of all the packed fields of a CustomerRecord */
#define CUSTOMERRECORD_SIZE ((long)(    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.

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.

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.

2.4 Fermeture d'un fichier client

Ecrire la fonction **void CustomerDB_close(CustomerDB * customerDB)** qui ferme le fichier client.

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.

8 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 16
/** The size in bytes of the designation field of a CatalogRecord */
#define CATALOGRECORD_DESIGNATION_SIZE 128
/** The size in bytes of the unity field of a CatalogRecord */
#define CATALOGRECORD_UNITY_SIZE 20
/** The size in bytes of the basePrice field of a CatalogRecord */
#define CATALOGRECORD_BASEPRICE_SIZE sizeof(double)
/** The size in bytes of the sellingPrice field of a CatalogRecord */
#define CATALOGRECORD_SELLINGPRICE_SIZE sizeof(double)
/** The size in bytes of the rateOfVAT field of a CatalogRecord */
#define CATALOGRECORD_RATEOFVAT_SIZE sizeof(double)

/** The size in bytes of all the packed fields of a CatalogRecord */
#define CATALOGRECORD_SIZE ((long)(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 ((long) \
      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**.

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.

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.

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.

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.

9 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.

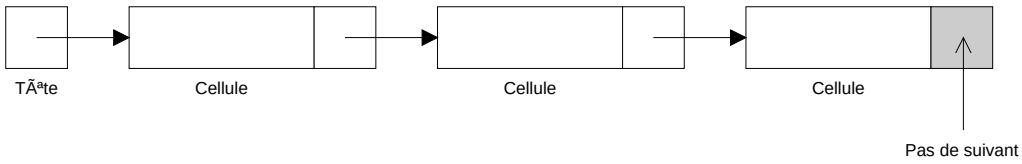
Ecrire la fonction **char * readString(FILE * file)** qui lit dans un fichier binaire une chaîne de caractères précédemment écrite via la fonction **writeString**. La chaîne de caractères retournée est allouée sur le tas.

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 **void DocumentRowList_insertAfter(DocumentRow ** list, DocumentRow * position, DocumentRow * row)** qui insère une cellule après la position indiquée dans une liste.

2.2.7 Insertion avant une cellule

Ecrire la fonction **void DocumentRowList_insertBefore(DocumentRow ** list, DocumentRow* position, DocumentRow* row)** qui insère une cellule avant la position indiquée dans une liste.

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 précédemment.

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**.

10 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 **DictionaryEntry * Dictionary_getEntry(Dictionary * dictionary, const char * name)** 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.

2.3 Définition d'une variable du type chaîne de caractères

Ecrire la fonction **void Dictionary_setStringEntry(Dictionary * dictionary, const char * name, const char * value)** 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.

2.4 Définition d'une variable du type nombre réel

Ecrire la fonction **void Dictionary_setNumberEntry(Dictionary * dictionary, const char * name, double value)** 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.

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 :

```
.NAME Ici il y a le nom du modele
.HEADER
Ici commence l'entete

Il peut y avoir plusieurs lignes
     et meme des blancs
Ici se termine l'entete
.ROW
Ici commence le modele d'une ligne en generale sur une seule ligne
mais rien ne l'oblige
.FOOTER
Ici commence le pied de page
qui peut aussi comporter plusieurs lignes
.END
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.

```
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 :

```
1  TantQue on n'a pas parcouru tout le modele Faire
2    chercher le debut d'une balise
3    copier le texte precedant la balise
4    Si c'est un double '%' Alors
5        ajouter un '%' dans le texte
6    Sinon
7      chercher la fin de la balise
8      extraire le nom de la balise et ses parametres
9      rechercher la valeur associee au nom de la balise
10     formater la valeur trouvee selon les parametres de la balise
11     ajouter la valeur formatee a la suite du texte
12   FinSi
13 FinTantQue
14 copier le texte jusqu'a la fin du modele
```

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.