



# Programmation en langage C

## Travaux dirigés et pratiques guidés

3 mai 2017

Sébastien Aupetit

[aupetit@univ-tours.fr](mailto:aupetit@univ-tours.fr) - Bureau 209 du Département Informatique

Ce document est en cours de rédaction, signalez moi toutes erreurs ou omissions qui permettraient son amélioration pour les années à venir.

Ce document a été réalisé uniquement comme une aide pour les cours, TDs et TPs. Il ne peut se substituer à votre présence pendant les cours, TDs et TPs.

**Il ne doit pas être diffusé sous quelque forme que ce soit sans autorisation préalable.**



# 1

## Organisation des séances

Les séances de TDs et TP's vont vous permettre de progressivement écrire une application de facturation ultra-simplifiée. Pour cela, les 14h TDs et 12h TP's 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 chaînes 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 formatage 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 chaînée 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 <i>a priori</i> , gestion d'un dictionnaire de valeurs avec des unions, formatage 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

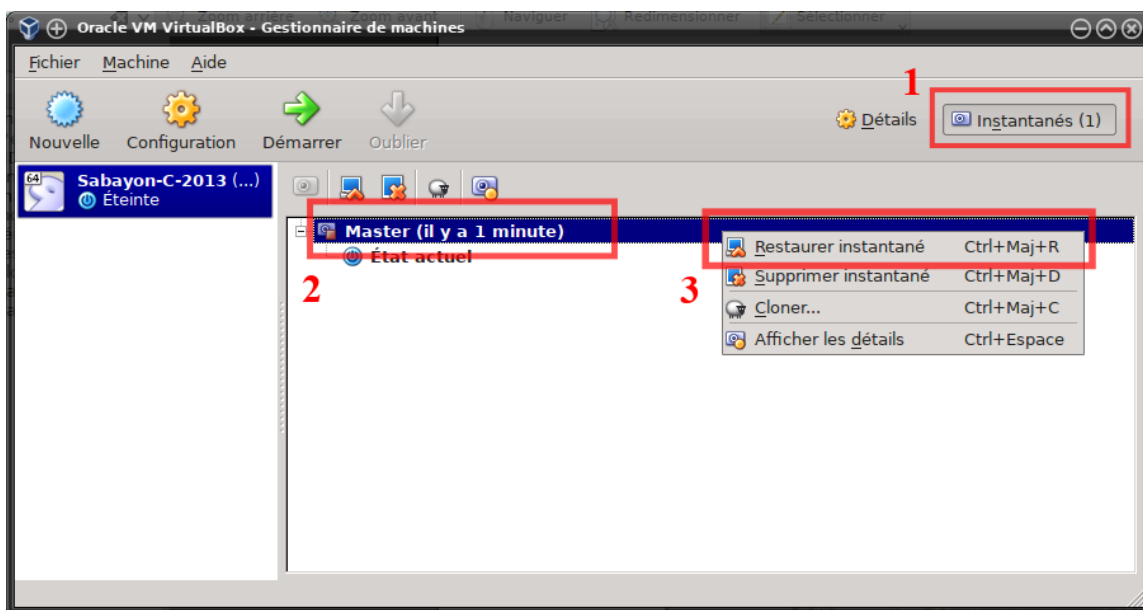
L'ensemble des séances de travaux pratiques se déroulent sous un environnement GNU/Linux. Pour cela, une machine virtuelle préconfigurée est mise à votre disposition. Cette machine virtuelle a été configurée pour qu'elle vous soit facile d'accès et de compréhension. Le nom de la machine virtuelle est « Sabayon-C 2013 ».

### 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 « Master ». Vous devez le restaurer afin d'obtenir une machine virtuelle propre :

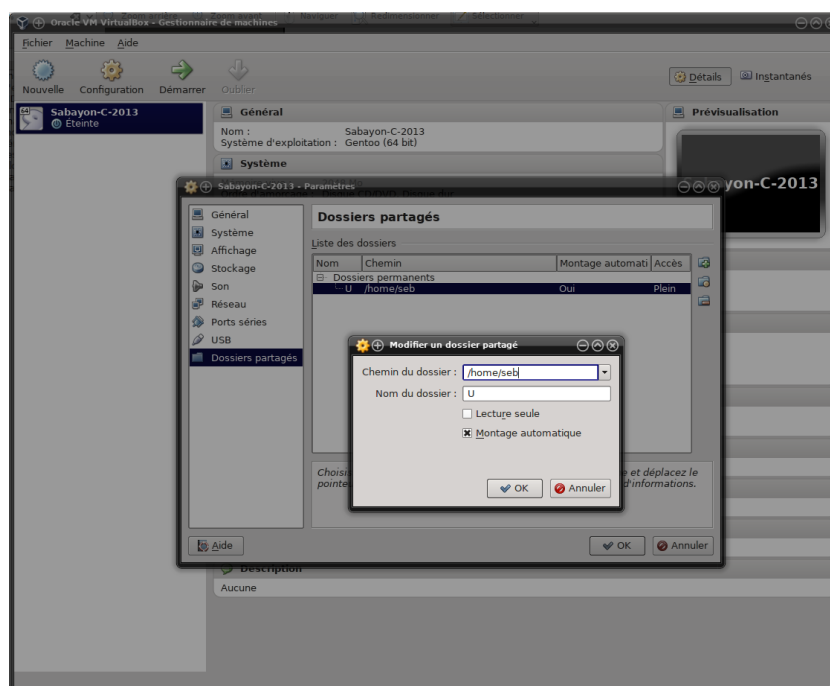
1. Cliquez sur « Instantané » ou « Snapshot »
2. Cliquez sur « Master »
3. Cliquez avec le bouton de droite et choisissez « Restaurer instantané » ou « Restore snapshot ».



#### 1.2 Le dossier partagé

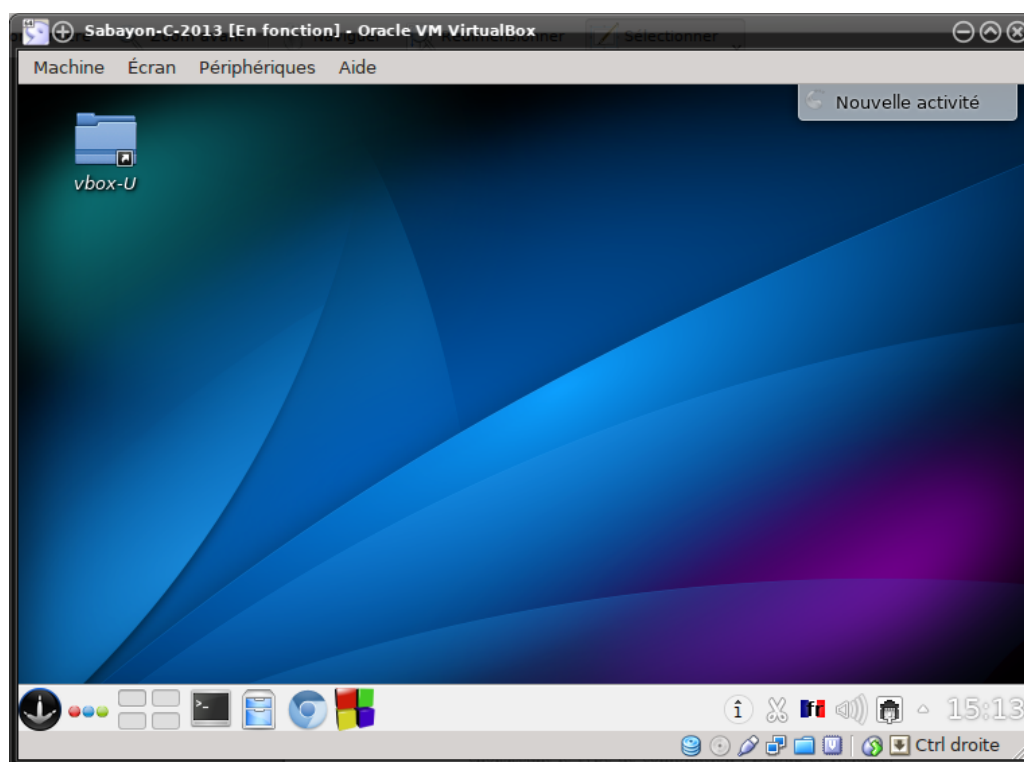
Avant l'exécution de la machine virtuelle, vous devez vous assurer que vos fichiers sont accessibles depuis le système GNU/Linux.

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 accessible 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 accéder 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 icône permettant de lancer une console (ici il s'agit d'une console **konsole**), une icône pour lancer l'explorateur de fichiers, une icône pour lancer le navigateur **chromium** (identique à Google Chrome) et une icône 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 accéder aux consoles textuelle du système GNU/Linux
- **Ctrl droite+F7** pour accéder à l'interface graphique

### ? Trucs et astuces

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'icône 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.

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

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

### 2.3.2 Code::Blocks ne répond plus

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

```
1 killall codeblocks
```

et si cela ne suffit pas, vous utilisez

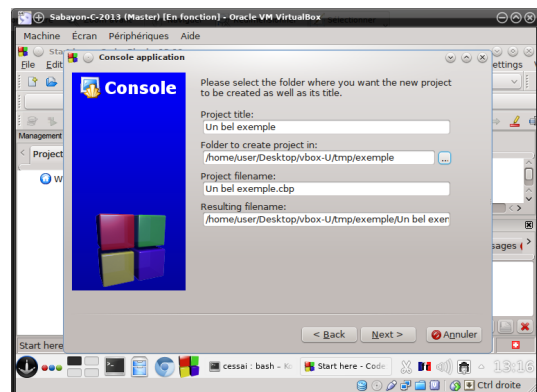
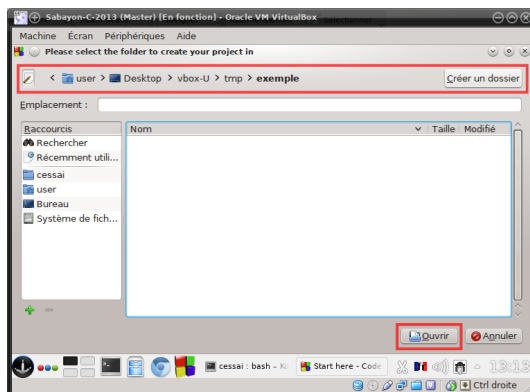
```
1 killall -s9 codeblocks
```

## 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 maîtriser votre environnement de développement.

### 3.1 Création d'un projet Code::Blocks

1. Créez un projet du type console<sup>1</sup>.
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) sélectionnez le.



5. Dans l'écran suivant, vous pouvez voir que deux cibles différentes sont créées : 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 TP, 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'exécutable.
- **Run** : Exécute le programme.



#### Avertissement

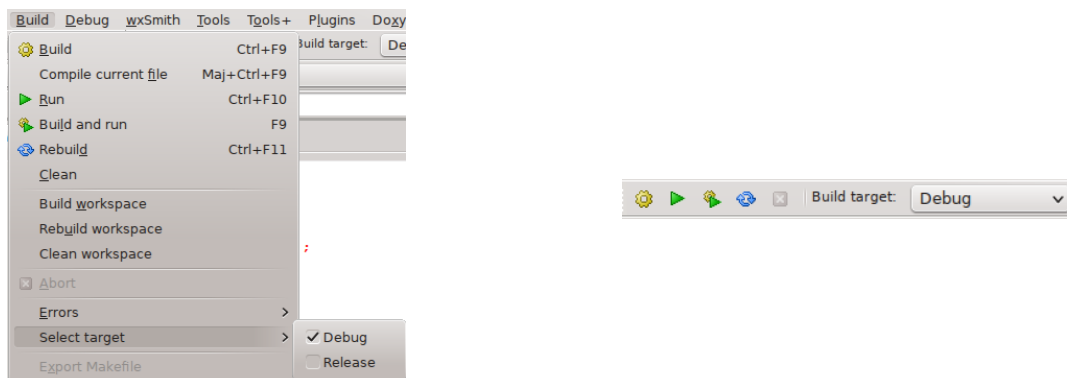
Le programme n'est pas compilé. Le programme exécuté correspond au dernier exécutable créé même si vous avez modifié le code depuis.

- **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'exécution sont plus long mais vous pouvez effectuer du débogage de code. A l'opposé, la version Release est plus rapide mais elle ne permet pas de faire du débogage. 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 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 apparaît 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 debugger
- **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'instruction 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

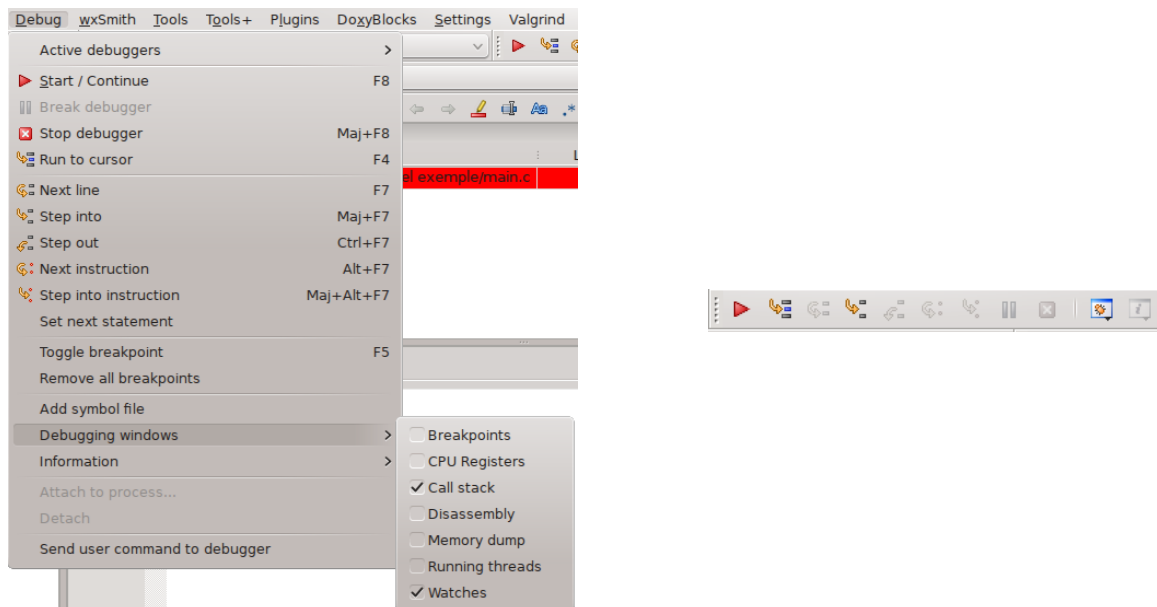
- **Debugging windows** : Permet d'afficher des fenêtres d'information spéciales (ex : la pile d'appel, les valeurs des variables...)

## Trucs et actuces

Assurez vous d'avoir en permanence les fenêtres d'information `Call stack` et `Watches` ouvertes lors du débogage de vos programmes. Dans la fenêtre `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 instruction` 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 Debug 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 <sup>1</sup>. Exécuter votre programme en mode Debug et en mode Release. Que constate-t-on ?
3. Avoir débogué 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

(**6**, 2, 5, 3, 9) → (2, 6, 5, 3, 9)  
(2, **6**, 5, 3, 9) → (2, 5, 6, 3, 9)  
(2, 5, **6**, 3, 9) → (2, 5, 3, 6, 9)  
(2, 5, 3, **6**, 9) → (2, 5, 3, 6, 9)

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

##### Etape 2

(**2**, 5, 3, 6, 9) → (2, 5, 3, 6, 9)  
(2, **5**, 3, 6, 9) → (2, 3, 5, 6, 9)  
(2, 3, **5**, 6, 9) → (2, 3, 5, 6, 9)

##### Etape 3

(**2**, 3, 5, 6, 9) → (2, 3, 5, 6, 9)  
(2, **3**, 5, 6, 9) → (2, 3, 5, 6, 9)

Commentaire : aucune permutation ; fin du tri.

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



# 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 :

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

Placez vous dans le répertoire ainsi créé :

```
1 cd ProjetC-JeanNoelDeLaMottePicquet
```

ou

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

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

```
1 wget -q -O - http://deposit.projectsforge.org/update/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 :

```
1 ./update.sh
```

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



### Avertissement

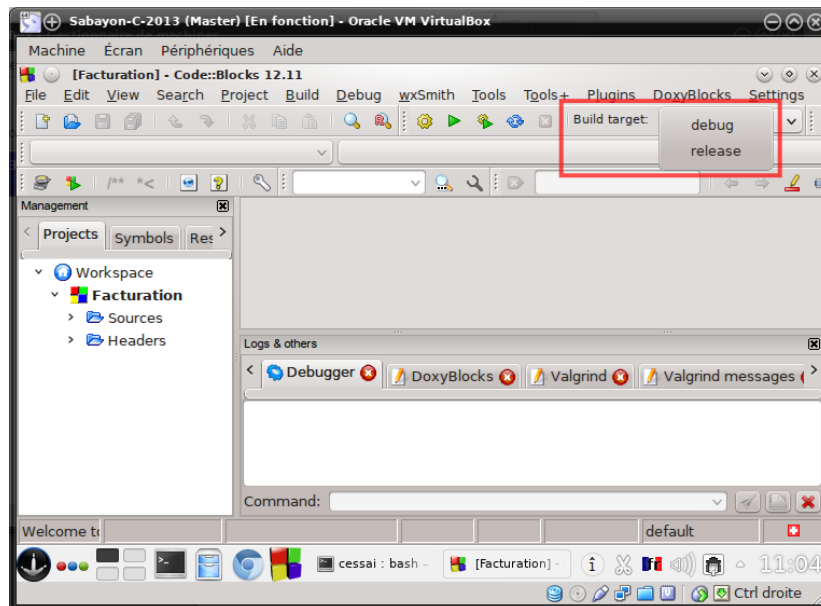
A chaque exécution du script, une copie de sauvegarde de vos fichiers sera faite avec la création d'un répertoire parallèle daté (ex : `~/Desktop/vbox-U/ProjetC-JeanNoelDeLaMottePicquet-2013-09-06#09:54:21`).

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

- `printfomat` 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'exécution sont plus long mais vous pouvez effectuer du débogage de code. A l'opposé, la version Release est plus rapide mais elle ne permet pas de faire du débogage. 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 choix 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 astuces

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

```
1 debug/facturation option1 option2
2 release/facturation option1 option2
```

## ? Trucs et astuces

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

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

Les principales options utiles sont :

<b>verbose-unittests</b>	Par défaut, les tests unitaires sont tous exécutés et l’affichage est compacte. En utilisant ce paramètre, vous demandez l’affichage de chaque <b>fonction des tests unitaires</b> .
<b>disable-unit-MyString</b>	Desactive l’exécution de l’ensemble des tests unitaires associés au module <b>MyString</b> . Il existe un paramètre similaire pour chacun des modules de tests unitaires. La liste s’affiche par défaut lors de l’exécution du programme.
<b>disable-unit-test_toLowerChar</b>	Desactive l’exécution de la fonction <b>test_toLowerChar</b> du test unitaire <b>MyString</b> . Il existe un paramètre similaire pour chacune des fonctions composant un test unitaire. La liste peut être obtenue lorsque <b>verbose-unittests</b> est spécifié.
<b>silent-tests</b>	Desactive l’affichage de la progression des tests unitaires
<b>reduce-dump-usage</b>	Par défaut, la liste des fonctions non implémentées est affichée à la fin du programme. Ce paramètre permet de désactiver cet affichage.
<b>disable-dump-usage</b>	Par défaut, la liste des fonctions non implémentées et la liste des modules ayant encore des fonctions non implémentées sont affichés à la fin du programme. Cette option permet de désactiver ces deux affichages.
<b>disable-gui</b>	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 implémentées.
- 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 lisible et commenté si nécessaire.
- Vous avez suivi toutes les séances de TDs et de TP.





# 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 vôtres. 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 concrètement 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).

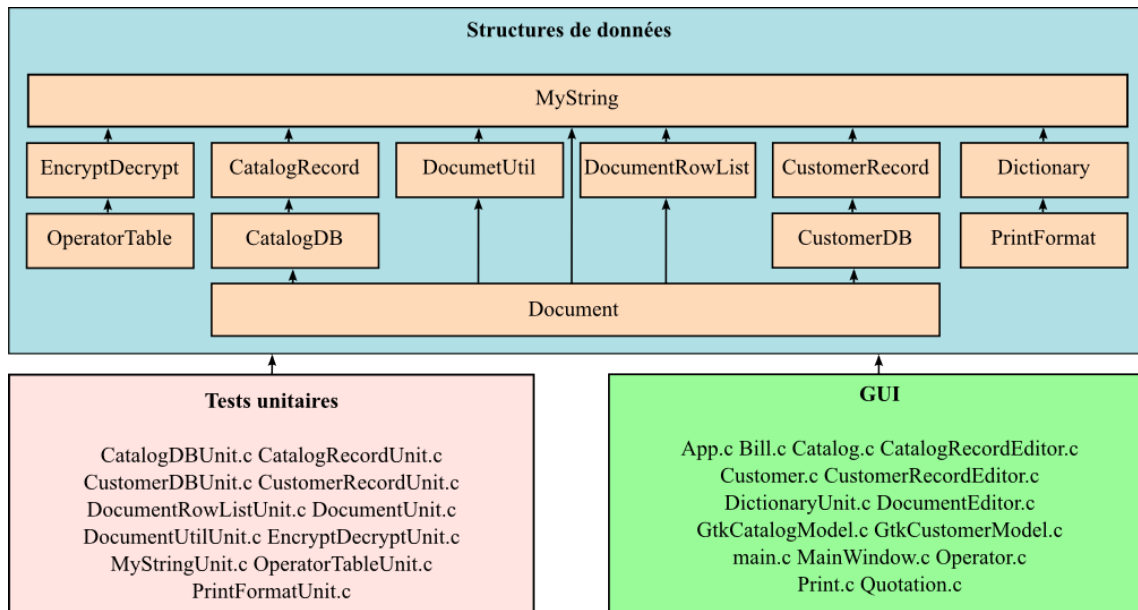


#### Avertissement

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	<b>Manipulation de chaînes de caractères :</b> <ul style="list-style-type: none"> <li>— Remplacement des fonctions standards de manipulation de chaînes de caractères</li> <li>— Ajout de fonctions de manipulation de chaînes de caractères</li> </ul>	MyStringUnit.h, MyStringUnit.c
OperatorTable.h, OperatorTable.c	<b>Gestion des opérateurs :</b> <ul style="list-style-type: none"> <li>— chargement et sauvegarde de la liste des opérateurs et de leurs mots de passe</li> <li>— ajout, suppression et modification d'un opérateur</li> </ul>	OperatorTableUnit.h, OperatorTableUnit.c
EncryptDecrypt.h, EncryptDecrypt.c	<b>Cryptage et décryptage du fichier des opérateurs :</b> <ul style="list-style-type: none"> <li>— cryptage et décryptage en fonction d'une clé de cryptage</li> </ul>	EncryptDecryptUnit.h, EncryptDecrypt.c
CatalogRecord.h, CatalogRecord.c, CatalogDB.h, CatalogDB.c, GtkCatalogModel.h, GtkCatalogModel.c, Catalog.h, Catalog.c	<b>Gestion du catalogue des produits :</b> <ul style="list-style-type: none"> <li>— Manipulation et édition d'un produit</li> <li>— Manipulation du catalogue des produits</li> </ul>	CatalogRecordUnit.h, CatalogRecordUnit.c, CatalogDBUnit.h, CatalogDBUnit.c
CustomerRecord.h, CustomerRecord.c, CustomerDB.h, CustomerDB.c, GtkCustomerModel.h, GtkCustomerModel.c, Customer.h, Customer.c	<b>Gestion du fichier client :</b> <ul style="list-style-type: none"> <li>— Manipulation et édition d'un client</li> <li>— Manipulation du fichier client</li> </ul>	CustomerRecordUnit.h, CustomerRecordUnit.c, CustomerDBUnit.h, CustomerDBUnit.c
DocumentUtil.h, DocumentUtil.c	<b>Fonctions utilitaires pour la manipulation de documents :</b> <ul style="list-style-type: none"> <li>— Calcule du numéro de document</li> <li>— Formattage de la date</li> <li>— Lecture et écriture de chaînes de caractères de taille variable dans un fichier binaire</li> </ul>	DocumentUtilUnit.h, DocumentUtilUnit.c
DocumentRowList.h, DocumentRowList.c, Document.h, Document.c, Quotation.h, Bill.h, Quotation.c, Bill.c, DocumentEditor.h, DocumentEditor.c	<b>Gestion des documents :</b> <ul style="list-style-type: none"> <li>— Manipulation d'un devis ou d'une facture</li> <li>— Manipulation du contenu d'un document</li> <li>— Edition et visualisation du contenu d'un document</li> </ul>	DocumentRowListUnit.h, DocumentRowListUnit.c, DocumentUnit.h, DocumentUnit.c
Print.h, Print.c, PrintFormat.h, PrintFormat.c, Dictionary.h, Dictionary.c	<b>Gestion d'aperçu avant impression et formattage de la mise en forme d'un document :</b> <ul style="list-style-type: none"> <li>— Visualisation de l'aperçu</li> <li>— Chargement d'un modèle de format</li> <li>— Formattage d'un document selon un modèle</li> </ul>	DictionaryUnit.h, DictionaryUnit.c, PrintFormat.h, PrintFormat.c
main.c, App.h, App.c, Config.h, MainWindow.h, MainWindow.c, Registry.h, UnitTest.h	<b>Gestion de l'application, de la configuration et des tests unitaires</b>	

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



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

Chaque module possédant 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 appelées 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 conséquent, à chaque fois que la fonction `stringLength` est appelée, 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 debugger 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 astuces

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 débogage 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 parties :

- 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.

```
1 char toLowerChar(char c) {
2     if ('A' <= c && c <= 'Z')
3         return (char) (c - 'A' + 'a');
4     else
5         return c;
6 }
```

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

```
1 char toUpperChar(char c) {
2     if ('a' <= c && c <= 'z')
3         return (char) (c - 'a' + 'A');
4     else
5         return c;
6 }
```

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

```
1 void makeLowerCaseString(char * str) {
2     while (*str != '\0') {
3         *str = toLowerChar(*str);
4         str++;
5     }
6 }
```

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

```
1 void makeUpperCaseString(char * str) {
2     while (*str != '\0') {
3         *str = toUpperChar(*str);
4         str++;
5     }
6 }
```

## 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 size_t stringLength(const char * str) {
2     size_t count = 0;
3     while (*str != '\0') {
4         count++;
5         str++;
6     }
7     return count;
8 }
```

## 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 int compareString(const char * str1, const char * str2) {
2     while (*str1 != '\0' && *str2 != '\0' && *str1 == *str2) {
3         str1++;
4         str2++;
5     }
6     if (*str1 == '\0' && *str2 == '\0')
7         return 0;
8     else {
9         return *str1 - *str2;
10    }
11 }
```

## 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 majuscule/minuscule.

```
1 int icaseCompareString(const char * str1, const char * str2) {
2     while (*str1 != '\0' && *str2 != '\0' && toLowerChar(*str1) == toLowerChar(*str2)) {
3         str1++;
4         str2++;
5     }
6     if (*str1 == '\0' && *str2 == '\0')
7         return 0;
8     else {
9         return toLowerChar(*str1) - toLowerChar(*str2);
10    }
11 }
```

## 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 occurrence du caractère `c` dans la chaîne ou `NULL` si la chaîne ne contient pas le caractère recherché.

```
1 const char * indexOfChar(const char *s, char c) {
2     while (*s != '\0' && *s != c)
3         s++;
4     if (*s == c)
5         return (char*) s;
6     else
7         return NULL;
8 }
```

## 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 occurrence de `aiguille` dans `meule_de_foin` ou `NULL` si la chaîne `meule_de_foin` ne contient pas la chaîne recherchée.

```
1 char * indexOfString(const char *meule_de_foin, const char *aiguille) {
2     while (*meule_de_foin != '\0') {
3         char * pmeule = (char*) meule_de_foin;
4         char * paiguille = (char*) aiguille;
5         while (*pmeule != '\0' && *paiguille != '\0' && *pmeule == *paiguille) {
6             pmeule++;
7             paiguille++;
8         }
9         if (*paiguille == '\0') {
10            return (char*) meule_de_foin;
11        }
12        meule_de_foin++;
13    }
14    return NULL;
15 }
```

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

Ecrire la fonction `int icenseStartWith(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 int icenseStartWith(const char *start, const char *str) {
2     while (*start != '\0' && *str != '\0' && toLowerChar(*start) == toLowerChar(*str)) {
3         start++;
4         str++;
5     }
6     return (*start == '\0');
7 }
```

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

Ecrire la fonction `int icenseEndWith(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 int icenseEndWith(const char *end, const char *str) {
2     size_t lenStr;
3     size_t lenEnd;
4
5     lenStr = stringLength(str);
6     lenEnd = stringLength(end);
7
8     if (lenStr < lenEnd)
9         return 0;
10    str += lenStr - lenEnd;
11    while (*end != '\0' && *str != '\0' && toLowerChar(*end) == toLowerChar(*str)) {
12        end++;
13        str++;
14    }
15    return (*end == '\0');
16 }
```

## 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.

```
1 void copyStringWithLength(char *dest, const char *src, size_t destsize) {
2     size_t len;
3     size_t i;
4
5     if (destsize == 0)
6         fatalError("destSize must not be null");
7
8     len = stringLength(src);
9
10    if (len + 1 > destsize)
11        len = destsize - 1;
12    for (i = 0; i < len; ++i)
13        dest[i] = src[i];
14    dest[len] = '\0';
15 }
```

## 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`.

```
1 char * duplicateString(const char * str) {
2     char * result;
3     char * p;
4
5     result = malloc((size_t) strlen(str) + 1);
6     if (result == NULL)
7         fatalError("Echec d'allocation");
8     p = result;
9
10    while (*str != '\0')
11        *p++ = *str++;
12    *p = *str;
13    return result;
14 }
```

### 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`.

```
1 char * concatenateString(const char * src1, const char * src2) {
2     char * result;
3     char * p;
4
5     result = (char*) malloc(strlen(src1) + strlen(src2) + 1);
6     if (result == NULL)
7         fatalError("Echec d'allocation");
8     p = result;
9
10    while (*src1 != '\0') {
11        *p = *src1;
12        p++;
13        src1++;
14    }
15
16    while (*src2 != '\0') {
17        *p = *src2;
18        p++;
19        src2++;
20    }
21    *p = '\0';
22    return result;
23 }
```

### 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"

```
1 char * subString(const char * start, const char * end) {
2     char * result;
3     char * p;
4
5     result = (char*) malloc((size_t) (end - start + 1));
6     p = result;
7
8     while (*start != *end) {
```



```

9     *p = *start;
10    start++;
11    p++;
12 }
13 *p = '\0';
14 return result;
15 }

```

## 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" */

```

```

1 char * insertString(const char * src, int insertPosition, const char * toBeInserted,
2 int insertLength) {
3     char * result;
4     int i;
5
6     result = (char*) malloc(stringLength(src) + (size_t) insertLength + 1);
7
8     for (i = 0; i < insertPosition; ++i)
9         result[i] = src[i];
10    for (i = 0; i < insertLength; ++i)
11        result[insertPosition + i] = toBeInserted[i];
12    for (i = insertPosition; src[i] != '\0'; ++i)
13        result[i + insertLength] = src[i];
14    result[i + insertLength] = '\0';
15    return result;
16 }

```

## 3 EncryptDecrypt.c : chiffage 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 chiffage. Elle ne contient que des lettres en majuscule ou en minuscule. Lors du processus de chiffage/déchiffage, la casse de la clé sera ignorée (ex : les clés `ab` et `AB` doivent conduire au même résultats).

```

1 void encrypt(const char * key, char * str) {
2     const char * p = key;
3
4     while (*str != '\0') {
5         if ('a' <= *str && *str <= 'z')
6             *str = (char) ((*str - 'a' + toLowerChar(*p) - 'a') % 26 + 'a');
7         else if ('A' <= *str && *str <= 'Z')
8             *str = (char) ((*str - 'A' + toLowerChar(*p) - 'a') % 26 + 'A');
9         str++;
10        p++;
11        if (*p == '\0')
12            p = key;
13    }
14 }
15
16 void decrypt(const char * key, char * str) {
17     const char * p = key;
18
19     while (*str != '\0') {
20         if ('a' <= *str && *str <= 'z')
21             *str = (char) ((26 + *str - 'a' - (toLowerChar(*p) - 'a')) % 26 + 'a');

```

```

22     else if ( 'A' <= *str && *str <= 'Z' )
23         *str = (char) ((26 + *str - 'A' - (toLowerCase(*p) - 'a')) % 26 + 'A');
24     str++;
25     p++;
26     if (*p == '\0')
27         p = key;
28 }
29 }

```

### 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 chiffrement 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 restent 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 suffisamment 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 chiffrement.
4. Écrire la fonction réalisant le déchiffrement.
5. Écrire une fonction *main* qui assure le chiffrement et le déchiffrement d'un message. Le message initial et le message résultant seront affichés à l'écran.
6. Ré-écrire les fonctions de chiffrement et de déchiffrement 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 char * computeDocumentNumber(long id) {
2     char buf[1024];
3     int pos = 0;
4     int len = 0;
5     long temp = id;
6     long val;
7
8     while (temp > 0) {
9         len++;
10        temp = temp / 36;
11    }
12
13    while (id > 0) {
14        val = id % 36;
15        if (val < 10)
16            buf[len - 1 - pos] = (char) ('0' + val);
17        else
18            buf[len - 1 - pos] = (char) ('A' + val - 10);
19        pos++;
20        id = id / 36;
21    }
22    buf[pos] = '\0';
23    return bridge_duplicateString(buf);
24 }
25 char * formatDate(int day, int month, int year) {
26     char buf[1024];
27
28     sprintf(buf, "%02d/%02d/%4d", day, month, year);
29     return duplicateString(buf);
30 }

```

### 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  $b_0, \dots, b_{N-1}$  ( $b_i \in \{0, 1, 2, \dots, B-1\}$ ).

$b_{N-1}$	$\dots$	$b_5$	$b_4$	$b_3$	$b_2$	$b_1$	$b_0$
-----------	---------	-------	-------	-------	-------	-------	-------

On a :

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

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

Exemple :  $0101 \Rightarrow 1+2(0+2(1+2(0)))=4+1=5$



## 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.

La convention de représentation retenue est la suivante :

Indice du tableau C	N-1	...	5	4	3	2	1	0
	$b_{N-1}$	...	$b_5$	$b_4$	$b_3$	$b_2$	$b_1$	$b_0$

Remarque : ce n'est pas la seule convention qui peut être retenue.

```

1  #include <math.h>
2  #include <stdio.h>
3
4  long int BaseB2Dec( int B, int N, int b[] )
5  {
6      long int n;
7      int i;
8
9      n = b[N-1];
10     for( i = N-2; i >= 0; --i )
11         n = n*B + b[i];
12     return n;
13 }
```

2. Écrire la fonction **Bin2Dec**, utilisant **BaseB2Dec**, qui retourne la valeur décimale signée correspondant à la représentation binaire sur 16 bits fournie en paramètre.

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

```

1  long int Bin2Dec( int b[] )
2  {
3      long int n;
4
5      n = Base2Dec(2,15,b);
6      if (b[15] == 1)
7          n = (-1L<<15) + n;
8      return n;
9  }
```

3. Écrire la fonction **main**

```

1  int main(void)
2  {
3      long int n;
4      int b[16];
5      int i;
6
7      printf("Entrez les 16 bits du nombres :\n");
8      for( i = 15; i >= 0; --i )
9      {
10         printf("bits %d : ",i);
11         scanf("%d", &(b[i]));
12     }
13     n = Bin2Dec(b);
14     printf("\nLe nombre vaut %ld", n);
15     return 0;
16 }
```

## 2 Conversion décimale vers binaire

Soit  $n$  un entier décimal de type **short**. Soit  $b_0, \dots, b_{15}$  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 :

- $n=1 \Rightarrow 0000\ 0001$
- $n=-1 \Rightarrow 128-1=127 \Rightarrow 1111\ 1111$
- Soit *div* l'opérateur de division entière et *mod* l'opérateur reste de la division entière, alors pour tout entier  $n$ , on a :  $n = (n \text{ div } k) * k + (n \text{ mod } k)$ .
- Schéma de Horner sur un nombre non signé :

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

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

Exemple :  $n = 5 \Rightarrow 0101$  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.

La même convention de représentation que l'exercice précédent est utilisée.

Il est important de remarquer que :

$$\sum_{i=0}^{14} b_i 2^i = \begin{cases} n & \text{si } b_{15} = 0 \text{ i.e. } n \geq 0 \\ n + -2^{15} & \text{si } b_{15} = 1 \text{ i.e. } n < 0 \end{cases}$$

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 void Dec2Bin( short n, int b[] )
6 {
7     int i;
8
9     if (n < 0)
10     {
11         b[15] = 1;
12         n = (int) (pow(2, 15) + n);
13     }
14     else
15         b[15] = 0;
16
17     for( i = 0; i < 15; i++)
18     {
19         b[i] = n % 2;
20         n = (n - b[i]) / 2;
21     }
22 }
```

Autre solution :

```
1 void Dec2Bin( short n, int b[] )
2 {
3     int i;
4
5     for( i = 0; i < 16; ++i)
6         b[i] = (n & (1 << i)) > 0;
7 }
```

2. Écrire la fonction **main** qui demande la saisie d'un nombre entier de type short et qui affiche sa représentation binaire.

```
1 int main(void)
2 {
3     int b[16];
4     int n;
5     int i;
6
7     printf("Saisissez un nombre entier :");
8     scanf("%hd", &n);
9     Dec2Bin(n,b);
10    printf("\n");
11    for( i = 15; i >=0; --i)
12        printf("%hd", b[i]);
13    printf("\n");
14    return 0;
15 }
```

### 3 Conversion d'une chaîne de caractères hexadécimale 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*. Écrire la fonction **main** associée.

Le principe est similaire à celui de la conversion de base B vers décimale.

```
1 #include <stdio.h>
2
3 int hexaToInt( char * hexa )
4 {
5     int n = 0;
6     int v;
7     int i;
8
9     if (hexa[0] == '0')
10        if ((hexa[1] == 'x') || (hexa[1] == 'X'))
11        {
12            i = 2;
13            do
14            {
15                if (('0'<=hexa[i]) && (hexa[i]<='9'))
16                    v = hexa[i] - '0';
17                else
18                    if (('a'<=hexa[i]) && (hexa[i]<='z'))
19                        v = hexa[i] - 'a' + 10;
20                else
21                    if (('A'<=hexa[i]) && (hexa[i] <= 'Z'))
22                        v = hexa[i] - 'A' + 10;
23                else
24                    return n;
25                n = n*16+v;
26                ++i;
27            }
28            while (1);
29        }
30    return -1;
31 }
32
33 int main(void)
34 {
35     char * h = "0x14dE1";
36     char buf[200];
37     int n = hexaToInt(h);
38
39     printf("%d\n", n); // affiche 85473
40
41     gets(buf);
42     n = hexaToInt(buf);
43     printf("%d\n", n);
44     return 0;
45 }
```

### 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. Écrire 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.



Le principe est similaire à celui de la conversion d'un entier en base B sauf que l'on stocke des caractères plutôt que des nombres.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 void inverse( char * s );
6
7 char * intToString( int n, char * s )
8 {
9     char temp[2];
10    int negatif;
11
12    temp[1] = '\0';
13    s[0] = '\0';
14    if (n < 0)
15    {
16        negatif = 1;
17        n = -n;
18    }
19    else
20        negatif = 0;
21
22    do
23    {
24        temp[0] = '0' + (n % 10);
25        n /= 10;
26        strcat(s,temp);
27    }
28    while (n > 0);
29    if (negatif)
30        strcat(s, "-");
31
32    inverse(s);
33    return s;
34 }
35
36 void inverse( char * s )
37 {
38     int i, j;
39     int len = strlen(s);
40     char c;
41
42     for( i = 0, j = len-1; i<j; ++i, --j )
43     {
44         c = s[i];
45         s[i] = s[j];
46         s[j] = c;
47     }
48 }
49
50 int main(void)
51 {
52     char Result[20];
53
54     intToString(-102,Result);
55     strcat(Result, "\n");
56     printf(Result);
57     return 0;
58 }
```



# 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 :

```
1  /** The maximal length in characters of the name of an operator */
2  #define OPERATORTABLE_MAXNAMESIZE 20UL
3  /** The maximal length in characters of the password of an operator */
4  #define OPERATORTABLE_MAXPASSWORDSIZE 20UL
5
6  /** The dynamic table of operators */
7  typedef struct {
8      /** The number of operators in the table */
9      int recordCount;
10     /** The data about the operators. It's a 2D array of strings.
11      * Note: records[operatorId][0] is the name of the operatorId'th operator
12      * Note: records[operatorId][1] is the password of the operatorId'th operator
13      */
14     char *** records;
15 } 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  OperatorTable * OperatorTable_create(void) {
2      OperatorTable * table;
3
4      table = (OperatorTable*) malloc(sizeof(OperatorTable));
5      if (table != NULL) {
6          table->recordCount = 0;
7          table->records = NULL;
8      }
9      return table;
10 }
```

#### 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  int OperatorTable_getRecordCount(OperatorTable * table) {
2      return table->recordCount;
3  }
```

### 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 const char * OperatorTable_getName(OperatorTable * table, int recordNum) {
2     if (recordNum < 0 || recordNum > table->recordCount)
3         fatalError("OperatorTable_GetName");
4     return table->records[recordNum][0];
5 }
```

### 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 const char * OperatorTable_getPassword(OperatorTable * table, int recordNum) {
2     if (recordNum < 0 || recordNum > table->recordCount)
3         fatalError("OperatorTable_GetPassword");
4     return table->records[recordNum][1];
5 }
```

### 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 int OperatorTable_findOperator(OperatorTable * table, const char * name) {
2     int i;
3     for (i = 0; i < OperatorTable_getRecordCount(table); ++i)
4         if (icaseCompareString(OperatorTable_getName(table, i), name) == 0)
5             return i;
6     return -1;
7 }
```

### 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 retournée est la position de l'utilisateur dans la table des opérateurs.

```
1 int OperatorTable_setOperator(OperatorTable * table, const char * name,
2     const char * password) {
3     int i;
4
5     for (i = 0; i < OperatorTable_getRecordCount(table); ++i)
6         if (icaseCompareString(OperatorTable_getName(table, i), name) == 0) {
7             free(table->records[i][1]);
8             table->records[i][1] = duplicateString(password);
9             return i;
10        }
11
12    table->records = (char**) realloc(table->records, sizeof(char**)
13        * ((unsigned int) table->recordCount + 1));
14    if (table->records == NULL)
15        fatalError("Echec de réallocation");
16    table->records[table->recordCount] = (char**) malloc(sizeof(char*) * 2);
17    table->records[table->recordCount][0] = duplicateString(name);
18    table->records[table->recordCount][1] = duplicateString(password);
19    table->recordCount++;
20    return table->recordCount - 1;
21 }
```

### 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 void OperatorTable_removeRecord(OperatorTable * table, int recordIndex) {
2     int i;
3
4     free(table->records[recordIndex][0]);
5     free(table->records[recordIndex][1]);
6     free(table->records[recordIndex]);
7     for (i = recordIndex + 1; i < table->recordCount; ++i) {
8         table->records[i - 1] = table->records[i];
9     }
10    table->records = (char**) realloc(table->records, sizeof(char**)
11        * ((unsigned int) table->recordCount - 1));
12    table->recordCount--;
13 }

```

## 1.8 Destruction de liste

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

```

1 void OperatorTable_destroy(OperatorTable * table) {
2     int i;
3
4     if (table != NULL) {
5         for (i = 0; i < table->recordCount; ++i) {
6             free(table->records[i][0]);
7             free(table->records[i][1]);
8             free(table->records[i]);
9         }
10        free(table->records);
11        free(table);
12    }
13 }

```

## 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);

OperatorTable * OperatorTable_loadFromFile(const char * filename) {
2     char buf[128];
3     char namebuf[OPERATORTABLE_MAXNAMESIZE];
4     char passwordbuf[OPERATORTABLE_MAXPASSWORDSIZE];
5     OperatorTable * table;
6     FILE * fichier;
7
8     table = OperatorTable_create();
9     fichier = fopen(filename, "rt");
10    if (fichier != NULL) {
11        int i;
12        int nbEnr;
13        if (fgets(buf, 128, fichier) == NULL)
14            fatalError("OperatorDB_LoadFromFile: fgets");
15
16        nbEnr = atoi(buf);
17        for (i = 0; i < nbEnr; ++i) {
18            if (fgets(namebuf, OPERATORTABLE_MAXNAMESIZE, fichier) == NULL)
19                fatalError("OperatorDB_LoadFromFile: fgets");
20            if (fgets(passwordbuf, OPERATORTABLE_MAXPASSWORDSIZE, fichier) == NULL)
21                fatalError("OperatorDB_LoadFromFile: fgets");
22            /* suppression des sauts de lignes finaux */
23            if (namebuf[stringLength(namebuf) - 1] == '\n')
24                namebuf[stringLength(namebuf) - 1] = '\0';
25            if (passwordbuf[stringLength(passwordbuf) - 1] == '\n')
26                passwordbuf[stringLength(passwordbuf) - 1] = '\0';
27            decrypt(OperatorCryptKey, namebuf);
28            decrypt(OperatorCryptKey, passwordbuf);
29            OperatorTable_setOperator(table, namebuf, passwordbuf);
30        }
31        fclose(fichier);
32    }
33    return table;
34 }

```

## 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.

```
1 void OperatorTable_saveToFile(OperatorTable * table, const char * filename) {
2     char namebuf[OPERATORTABLE_MAXNAMESIZE];
3     char passwordbuf[OPERATORTABLE_MAXPASSWORDSIZE];
4     int i;
5     FILE * fichier;
6
7     fichier = fopen(filename, "wt");
8     if (fichier != NULL) {
9         fprintf(fichier, "%d\n", (int) table->recordCount);
10        for (i = 0; i < OperatorTable_getRecordCount(table); ++i) {
11            copyStringWithLength(namebuf, OperatorTable_getName(table, i),
12                                OPERATORTABLE_MAXNAMESIZE);
13            copyStringWithLength(passwordbuf, OperatorTable_getPassword(table, i),
14                                OPERATORTABLE_MAXPASSWORDSIZE);
15            encrypt(OperatorCryptKey, namebuf);
16            encrypt(OperatorCryptKey, passwordbuf);
17            fprintf(fichier, "%s\n%s\n", namebuf, passwordbuf);
18        }
19        fclose(fichier);
20    }
21 }
```

# 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.

```
1  /** The size of the name field */
2  #define CUSTOMERRECORD_NAME_SIZE 70UL
3  /** The size of the address field */
4  #define CUSTOMERRECORD_ADDRESS_SIZE 130UL
5  /** The size of the postalCode field */
6  #define CUSTOMERRECORD_POSTALCODE_SIZE 20UL
7  /** The size of the town field */
8  #define CUSTOMERRECORD_TOWN_SIZE 90UL
9
10 /** The size in bytes of all the packed fields of a CustomerRecord */
11 #define CUSTOMERRECORD_SIZE ( CUSTOMERRECORD_NAME_SIZE + \
12                                CUSTOMERRECORD_ADDRESS_SIZE + \
13                                CUSTOMERRECORD_POSTALCODE_SIZE + \
14                                CUSTOMERRECORD_TOWN_SIZE )
15
16 /** A customer record */
17 typedef struct {
18     /** The name */
19     char name[CUSTOMERRECORD_NAME_SIZE];
20     /** The address */
21     char address[CUSTOMERRECORD_ADDRESS_SIZE];
22     /** The postal code */
23     char postalCode[CUSTOMERRECORD_POSTALCODE_SIZE];
24     /** The Town */
25     char town[CUSTOMERRECORD_TOWN_SIZE];
26 } 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)`

```
1 void CustomerRecord_setValue_name(CustomerRecord * record, const char * value) {
2     copyStringWithLength(record->name, value, CUSTOMERRECORD_NAME_SIZE);
3 }
```

— **void** `CustomerRecord_setValue_address(CustomerRecord * record, const char * value)`

```
1 void CustomerRecord_setValue_address(CustomerRecord * record, const char * value) {
2     copyStringWithLength(record->address, value, CUSTOMERRECORD_ADDRESS_SIZE);
3 }
```

— **void** `CustomerRecord_setValue_postalCode(CustomerRecord * record, const char * value)`

```
1 void CustomerRecord_setValue_postalCode(CustomerRecord * record, const char * value) {
2     copyStringWithLength(record->postalCode, value, CUSTOMERRECORD_POSTALCODE_SIZE);
3 }
```

— **void** `CustomerRecord_setValue_town(CustomerRecord * record, const char * value)`

```
1 void CustomerRecord_setValue_town(CustomerRecord * record, const char * value) {
2     copyStringWithLength(record->town, value, CUSTOMERRECORD_TOWN_SIZE);
3 }
```

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)

```
1 char * CustomerRecord_getValue_name(CustomerRecord * record) {  
2     return duplicateString(record->name);  
3 }
```

— **char \* CustomerRecord\_getValue\_address**(CustomerRecord \* record)

```
1 char * CustomerRecord_getValue_address(CustomerRecord * record) {  
2     return duplicateString(record->address);  
3 }
```

— **char \* CustomerRecord\_getValue\_postalCode**(CustomerRecord \* record)

```
1 char * CustomerRecord_getValue_postalCode(CustomerRecord * record) {  
2     return duplicateString(record->postalCode);  
3 }
```

— **char \* CustomerRecord\_getValue\_town**(CustomerRecord \* record)

```
1 char * CustomerRecord_getValue_town(CustomerRecord * record) {  
2     return duplicateString(record->town);  
3 }
```

## 1.2 Initialisation

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

```
1 static void zeroTab(char * tab, int size) {  
2     while (size > 0) {  
3         *tab = '\0';  
4         tab++;  
5         size--;  
6     }  
7 }  
8 void CustomerRecord_init(CustomerRecord * record) {  
9     zeroTab(record->name, CUSTOMERRECORD_NAME_SIZE);  
10    zeroTab(record->address, CUSTOMERRECORD_ADDRESS_SIZE);  
11    zeroTab(record->postalCode, CUSTOMERRECORD_POSTALCODE_SIZE);  
12    zeroTab(record->town, CUSTOMERRECORD_TOWN_SIZE);  
13 }
```

## 1.3 Finalisation

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

```
1 void CustomerRecord_finalize(CustomerRecord * UNUSED( record)) {  
2 }
```

## 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 void CustomerRecord_read(CustomerRecord * record, FILE * file) {  
2     if (fread(record->name, CUSTOMERRECORD_NAME_SIZE, 1, file) != 1)  
3         fatalError("CustomerRecord_Read: fread");  
4  
5     if (fread(record->address, CUSTOMERRECORD_ADDRESS_SIZE, 1, file) != 1)  
6         fatalError("CustomerRecord_Read: fread");  
7  
8     if (fread(record->postalCode, CUSTOMERRECORD_POSTALCODE_SIZE, 1, file) != 1)  
9         fatalError("CustomerRecord_Read: fread");  
10  
11    if (fread(record->town, CUSTOMERRECORD_TOWN_SIZE, 1, file) != 1)  
12        fatalError("CustomerRecord_Read: fread");  
13 }
```



## 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.

```
1 void CustomerRecord_write(CustomerRecord * record, FILE * file) {
2     if (fwrite(record->name, CUSTOMERRECORD_NAME_SIZE, 1, file) != 1)
3         fatalError("CustomerRecord_Write: fwrite");
4
5     if (fwrite(record->address, CUSTOMERRECORD_ADDRESS_SIZE, 1, file) != 1)
6         fatalError("CustomerRecord_Write: fwrite");
7
8     if (fwrite(record->postalCode, CUSTOMERRECORD_POSTALCODE_SIZE, 1, file) != 1)
9         fatalError("CustomerRecord_Write: fwrite");
10
11    if (fwrite(record->town, CUSTOMERRECORD_TOWN_SIZE, 1, file) != 1)
12        fatalError("CustomerRecord_Write: fwrite");
13 }
```

## 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'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 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.

```
1 CustomerDB * CustomerDB_create(const char * filename) {
2     CustomerDB * customerDB;
3     customerDB = (CustomerDB*) malloc(sizeof(CustomerDB));
4     if (customerDB == NULL)
5         fatalError("CustomerDB_Create: malloc failed");
6
7     customerDB->file = fopen(filename, "w+b");
8     if (customerDB->file == NULL) {
9         free(customerDB);
10        return NULL;
11    }
12
13    customerDB->recordCount = 0;
14    fseek(customerDB->file, 0, SEEK_SET);
15    if (fwrite(&customerDB->recordCount, sizeof(customerDB->recordCount), 1, >
16        ↪ customerDB->file) != 1) {
17        fclose(customerDB->file);
18        free(customerDB);
19        fatalError("CustomerDB_Create: fwrite failed");
20    }
21    return customerDB;
22 }
```

## 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.

```
1 CustomerDB * CustomerDB_open(const char * filename) {
2     CustomerDB * customerDB;
3     customerDB = (CustomerDB*) malloc(sizeof(CustomerDB));
4     if (customerDB == NULL)
5         fatalError("CustomerDB_Open: malloc failed");
6
7     customerDB->file = fopen(filename, "r+b");
8     if (customerDB->file == NULL) {
9         free(customerDB);
10        return NULL;
11    }
12
13    fseek(customerDB->file, 0, SEEK_SET);
14    if (fread(&customerDB->recordCount, sizeof(customerDB->recordCount), 1, customerDB->file) <
15        ↪ != 1) {
16        fclose(customerDB->file);
17        free(customerDB);
18        fatalError("CustomerDB_Open: fread failed");
19    }
20    return customerDB;
}
```

## 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.

```
1 CustomerDB * CustomerDB_openOrCreate(const char * filename) {
2     CustomerDB * customerDB;
3     customerDB = CustomerDB_open(filename);
4     if (customerDB == NULL)
5         customerDB = CustomerDB_create(filename);
6     return customerDB;
7 }
```

## 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.

```
1 void CustomerDB_close(CustomerDB * customerDB) {
2     if (customerDB != NULL) {
3         fseek(customerDB->file, 0, SEEK_SET);
4         if (fwrite(&customerDB->recordCount, sizeof(customerDB->recordCount), 1,
5             ↪ customerDB->file)
6             != 1)
7             fatalError("CustomerDB_Close: fwrite failed");
8         fclose(customerDB->file);
9         free(customerDB);
10    }
}
```

## 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.

```
1 int CustomerDB_getRecordCount(CustomerDB * customerDB) {
2     if (customerDB != NULL)
3         return customerDB->recordCount;
4     else
5         return 0;
6 }
```

## 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**.

```
1 void CustomerDB_readRecord(CustomerDB * customerDB, int recordIndex, CustomerRecord * >
    ↪ record) {
2     if (customerDB != NULL) {
3         if (recordIndex < 0 || recordIndex >= customerDB->recordCount)
4             fatalError("CustomerDB_GetRecord: invalid index");
5         fseek(customerDB->file, (long) sizeof(customerDB->recordCount) + CUSTOMERRECORD_SIZE >
    ↪ * recordIndex, SEEK_SET);
6         CustomerRecord_read(record, customerDB->file);
7     }
8 }
```

## 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.

```
1 void CustomerDB_writeRecord(CustomerDB * customerDB, int recordIndex, CustomerRecord * >
    ↪ record) {
2     if (customerDB != NULL) {
3         if (recordIndex < 0 || recordIndex > customerDB->recordCount)
4             fatalError("CustomerDB_SetRecord: invalid index");
5         fseek(customerDB->file, (long) sizeof(customerDB->recordCount) + CUSTOMERRECORD_SIZE >
    ↪ * recordIndex, SEEK_SET);
6         CustomerRecord_write(record, customerDB->file);
7         if (recordIndex == customerDB->recordCount)
8             customerDB->recordCount++;
9     }
10 }
```

## 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**.

```
1 void CustomerDB_insertRecord(CustomerDB * customerDB, int recordIndex, CustomerRecord * >
    ↪ record) {
2     if (customerDB != NULL) {
3         int i;
4         CustomerRecord temp;
5         CustomerRecord_init(&temp);
6         for (i = CustomerDB_getRecordCount(customerDB); i > recordIndex; --i) {
7             CustomerDB_readRecord(customerDB, i - 1, &temp);
8             CustomerDB_writeRecord(customerDB, i, &temp);
9         }
10        CustomerDB_writeRecord(customerDB, recordIndex, record);
11        CustomerRecord_finalize(&temp);
12    }
13 }
```

## 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.

```
1 void CustomerDB_appendRecord(CustomerDB * customerDB, CustomerRecord * record) {
2     CustomerDB_insertRecord(customerDB, CustomerDB_getRecordCount(customerDB), record);
3 }
```

## 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.

```
1 void CustomerDB_removeRecord(CustomerDB * customerDB, int recordIndex) {
2     if (customerDB != NULL) {
3         CustomerRecord temp;
4         int i;
5         CustomerRecord_init(&temp);
6         for (i = recordIndex; i < CustomerDB_getRecordCount(customerDB) - 1; ++i) {
7             CustomerDB_readRecord(customerDB, i + 1, &temp);
8             CustomerDB_writeRecord(customerDB, i, &temp);
9         }
10        customerDB->recordCount--;
11        CustomerRecord_finalize(&temp);
12    }
13 }
```

# 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.

```

1  /** The size in bytes of the code field of a CatalogRecord */
2  #define CATALOGRECORD_CODE_SIZE 16UL
3  /** The size in bytes of the designation field of a CatalogRecord */
4  #define CATALOGRECORD_DESIGNATION_SIZE 128UL
5  /** The size in bytes of the unity field of a CatalogRecord */
6  #define CATALOGRECORD_UNITY_SIZE 20UL
7  /** The size in bytes of the basePrice field of a CatalogRecord */
8  #define CATALOGRECORD_BASEPRICE_SIZE ((unsigned long) sizeof(double))
9  /** The size in bytes of the sellingPrice field of a CatalogRecord */
10 #define CATALOGRECORD_SELLINGPRICE_SIZE ((unsigned long) sizeof(double))
11 /** The size in bytes of the rateOfVAT field of a CatalogRecord */
12 #define CATALOGRECORD_RATEOFVAT_SIZE ((unsigned long) sizeof(double))
13
14 /** The size in bytes of all the packed fields of a CatalogRecord */
15 #define CATALOGRECORD_SIZE (CATALOGRECORD_CODE_SIZE + \
16                             CATALOGRECORD_DESIGNATION_SIZE + \
17                             CATALOGRECORD_UNITY_SIZE + \
18                             CATALOGRECORD_BASEPRICE_SIZE + \
19                             CATALOGRECORD_SELLINGPRICE_SIZE + \
20                             CATALOGRECORD_RATEOFVAT_SIZE)
21
22 /** The maximal length in characters of the string fields of a CatalogRecord */
23 #define CATALOGRECORD_MAXSTRING_SIZE ( \
24     MAXVALUE(CATALOGRECORD_CODE_SIZE, \
25     MAXVALUE(CATALOGRECORD_DESIGNATION_SIZE, CATALOGRECORD_UNITY_SIZE)))
26
27 /** A catalog record
28  */
29 typedef struct {
30     char * code /** The code of the product */;
31     char * designation /** The designation of the product */;
32     char * unity /** The unity of the product */;
33     double basePrice /** The base price of the product (the product should not be sold at a >=
34         ↳ lower price) */;
35     double sellingPrice /** The selling price of the product */;
36     double rateOfVAT /** The rate of the VAT of the product */;
37 } 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.

```

1  int CatalogRecord_isValueValid_code(const char * value) {
2      const char * p;
3      p = value;
4      while (*p != '\0') {
5          if (!(( '0' <= *p && *p <= '9' ) || ( 'a' <= *p && *p <= 'z' ) || ( 'A' <= *p && *p <= 'Z' )))
6              return 0;
7          p++;
8      }
9      return p != value;
10 }

```

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`.

```
1 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 :

```
1 man strtod
```

```
1 int CatalogRecord_isValueValid_positiveNumber(const char * value) {
2     char * end;
3     double val;
4     end = NULL;
5     val = strtod(value, &end);
6     return (end != value && *end == '\0' && val >= 0);
7 }
```

## 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)**

```
1 void CatalogRecord_setValue_code(CatalogRecord * record, const char * value) {
2     free(record->code);
3     record->code = duplicateString(value);
4 }
```

— **void CatalogRecord\_setValue\_designation(CatalogRecord \* record, const char \* value)**

```
1 void CatalogRecord_setValue_designation(CatalogRecord * record, const char * value) {
2     free(record->designation);
3     record->designation = duplicateString(value);
4 }
```

— **void CatalogRecord\_setValue\_unity(CatalogRecord \* record, const char \* value)**

```
1 void CatalogRecord_setValue_unity(CatalogRecord * record, const char * value) {
2     free(record->unity);
3     record->unity = duplicateString(value);
4 }
```

— **void CatalogRecord\_setValue\_basePrice(CatalogRecord \* record, const char \* value)**

```
1 void CatalogRecord_setValue_basePrice(CatalogRecord * record, const char * value) {
2     record->basePrice = atof(value);
3 }
```

— **void CatalogRecord\_setValue\_sellingPrice(CatalogRecord \* record, const char \* value)**

```
1 void CatalogRecord_setValue_sellingPrice(CatalogRecord * record, const char * value) {
2     record->sellingPrice = atof(value);
3 }
```

— **void CatalogRecord\_setValue\_rateOfVAT(CatalogRecord \* record, const char \* value)**

```
1 void CatalogRecord_setValue_rateOfVAT(CatalogRecord * record, const char * value) {
2     record->rateOfVAT = atof(value);
3 }
```

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)**

```
1 char * CatalogRecord_getValue_code(CatalogRecord * record) {
2     return duplicateString(record->code);
3 }
```

— **char \* CatalogRecord\_getValue\_designation(CatalogRecord \* record)**

```
1 char * CatalogRecord_getValue_designation(CatalogRecord * record) {
2     return duplicateString(record->designation);
3 }
```

— **char \* CatalogRecord\_getValue\_unity(CatalogRecord \* record)**

```
1 char * CatalogRecord_getValue_unity(CatalogRecord * record) {
2     return duplicateString(record->unity);
3 }
```

— **char \* CatalogRecord\_getValue\_basePrice(CatalogRecord \* record)**

```
1 char * CatalogRecord_getValue_basePrice(CatalogRecord * record) {
2     char buf[1024];
3     sprintf(buf, "%.2f", record->basePrice);
4     return duplicateString(buf);
5 }
```

— **char \* CatalogRecord\_getValue\_sellingPrice(CatalogRecord \* record)**

```
1 char * CatalogRecord_getValue_sellingPrice(CatalogRecord * record) {
2     char buf[1024];
3     sprintf(buf, "%.2f", record->sellingPrice);
4     return duplicateString(buf);
5 }
```

— **char \* CatalogRecord\_getValue\_rateOfVAT(CatalogRecord \* record)**

```
1 char * CatalogRecord_getValue_rateOfVAT(CatalogRecord * record) {
2     char buf[1024];
3     sprintf(buf, "%.2f", record->rateOfVAT);
4     return duplicateString(buf);
5 }
```

### 1.3 Initialisation

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

```
1 void CatalogRecord_init(CatalogRecord * record) {
2     record->code = duplicateString("");
3     record->designation = duplicateString("");
4     record->unity = duplicateString("");
5     record->basePrice = 0;
6     record->sellingPrice = 0;
7     record->rateOfVAT = 0;
8 }
```

### 1.4 Finalisation

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

```
1 void CatalogRecord_finalize(CatalogRecord * record) {
2     if (record->code != NULL)
3         free(record->code);
4     if (record->designation != NULL)
5         free(record->designation);
6     if (record->unity != NULL)
7         free(record->unity);
8 }
```

### 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 void CatalogRecord_read(CatalogRecord * record, FILE * file) {
2     char buffer[CATALOGRECORD_MAXSTRING_SIZE];
3
4     if (fread(buffer, CATALOGRECORD_CODE_SIZE, 1, file) != 1)
5         fatalError("CatalogRecord_Read: fread");
6     if (record->code != NULL)
7         free(record->code);
8     record->code = duplicateString(buffer);
9
10    if (fread(buffer, CATALOGRECORD_DESIGNATION_SIZE, 1, file) != 1)
11        fatalError("CatalogRecord_Read: fread");
12    if (record->designation != NULL)
13        free(record->designation);
14    record->designation = duplicateString(buffer);
15
16    if (fread(buffer, CATALOGRECORD_UNITY_SIZE, 1, file) != 1)
17        fatalError("CatalogRecord_Read: fread");
18    if (record->unity != NULL)
19        free(record->unity);
20 }
```

```

20 record->unity = duplicateString(buffer);
21
22 if (fread(&record->basePrice, sizeof(record->basePrice), 1, file) != 1)
23     fatalError("CatalogRecord_Read: fread");
24
25 if (fread(&record->sellingPrice, sizeof(record->sellingPrice), 1, file) != 1)
26     fatalError("CatalogRecord_Read: fread");
27
28 if (fread(&record->rateOfVAT, sizeof(record->rateOfVAT), 1, file) != 1)
29     fatalError("CatalogRecord_Read: fread");
30 }

```

## 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.

```

1 static void zeroTab(char * tab, int size) {
2     while (size > 0) {
3         *tab = '\0';
4         tab++;
5         size--;
6     }
7 }
8
9 void CatalogRecord_write(CatalogRecord * record, FILE * file) {
10     char buffer[CATALOGRECORD_MAXSTRING_SIZE];
11
12     zeroTab(buffer, CATALOGRECORD_MAXSTRING_SIZE);
13
14     copyStringWithLength(buffer, record->code, CATALOGRECORD_CODE_SIZE);
15     if (fwrite(buffer, CATALOGRECORD_CODE_SIZE, 1, file) != 1)
16         fatalError("CatalogRecord_Write: fwrite");
17
18     copyStringWithLength(buffer, record->designation, CATALOGRECORD_DESIGNATION_SIZE);
19     if (fwrite(buffer, CATALOGRECORD_DESIGNATION_SIZE, 1, file) != 1)
20         fatalError("CatalogRecord_Write: fwrite");
21
22     copyStringWithLength(buffer, record->unity, CATALOGRECORD_UNITY_SIZE);
23     if (fwrite(buffer, CATALOGRECORD_UNITY_SIZE, 1, file) != 1)
24         fatalError("CatalogRecord_Write: fwrite");
25
26     if (fwrite(&record->basePrice, sizeof(record->basePrice), 1, file) != 1)
27         fatalError("CatalogRecord_Write: fwrite");
28
29     if (fwrite(&record->sellingPrice, sizeof(record->sellingPrice), 1, file) != 1)
30         fatalError("CatalogRecord_Write: fwrite");
31
32     if (fwrite(&record->rateOfVAT, sizeof(record->rateOfVAT), 1, file) != 1)
33         fatalError("CatalogRecord_Write: fwrite");
34 }

```

## 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.

```
1 CatalogDB * CatalogDB_create(const char * filename) {
2     CatalogDB * catalogDB;
3
4     catalogDB = (CatalogDB*) malloc(sizeof(CatalogDB));
5     if (catalogDB == NULL)
6         fatalError("CatalogDB_Create: malloc failed");
7
8     catalogDB->file = fopen(filename, "w+b");
9     if (catalogDB->file == NULL) {
10         free(catalogDB);
11         return NULL;
12     }
13
14     catalogDB->recordCount = 0;
15     fseek(catalogDB->file, 0, SEEK_SET);
16     if (fwrite(&catalogDB->recordCount, sizeof(catalogDB->recordCount), 1, catalogDB->file) >=
17         ↪ != 1) {
18         fclose(catalogDB->file);
19         free(catalogDB);
20         fatalError("CatalogDB_Create: fwrite failed");
21     }
22     return catalogDB;
23 }
```

## 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.

```
1 CatalogDB * CatalogDB_open(const char * filename) {
2     CatalogDB * catalogDB;
3     catalogDB = (CatalogDB*) malloc(sizeof(CatalogDB));
4     if (catalogDB == NULL)
5         fatalError("CatalogDB_Open: malloc failed");
6
7     catalogDB->file = fopen(filename, "r+b");
8     if (catalogDB->file == NULL) {
9         free(catalogDB);
10        return NULL;
11    }
12
13    fseek(catalogDB->file, 0, SEEK_SET);
14    if (fread(&catalogDB->recordCount, sizeof(catalogDB->recordCount), 1, catalogDB->file) != >=
15        ↪ 1) {
16        fclose(catalogDB->file);
17        free(catalogDB);
18        fatalError("CatalogDB_Open: fread failed");
19    }
20    return catalogDB;
21 }
```

## 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. Cette fonction est une combinaison des deux fonctions précédentes.

```
1 CatalogDB * CatalogDB_openOrCreate(const char * filename) {
2     CatalogDB * catalogDB;
3
4     catalogDB = CatalogDB_open(filename);
5     if (catalogDB == NULL)
6         catalogDB = CatalogDB_create(filename);
7     return catalogDB;
8 }
```

## 2.4 Fermeture d'un fichier catalogue

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

```
1 void CatalogDB_close(CatalogDB * catalogDB) {
2     if (catalogDB != NULL) {
3         fseek(catalogDB->file, 0, SEEK_SET);
4         if (fwrite(&catalogDB->recordCount, sizeof(catalogDB->recordCount), 1, catalogDB->file)
5             != 1)
6             fatalError("CatalogDB_Close: fwrite failed");
7         fclose(catalogDB->file);
8         free(catalogDB);
9     }
10 }
```

## 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.

```
1 int CatalogDB_getRecordCount(CatalogDB * catalogDB) {
2     if (catalogDB != NULL)
3         return catalogDB->recordCount;
4     else
5         return 0;
6 }
```

## 2.6 Lecture d'un enregistrement

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

```
1 void CatalogDB_readRecord(CatalogDB * catalogDB, int recordIndex, CatalogRecord * record) {
2     if (catalogDB != NULL) {
3         if (recordIndex < 0 || recordIndex >= catalogDB->recordCount)
4             fatalError("CatalogDB_GetRecord: invalid index");
5         fseek(catalogDB->file, ((long) sizeof(catalogDB->recordCount) + CATALOGRECORD_SIZE
6             * recordIndex, SEEK_SET);
7         CatalogRecord_read(record, catalogDB->file);
8     }
9 }
```

## 2.7 Ecriture d'un enregistrement

Ecrire la fonction **void CatalogDB\_writeRecord(CatalogDB \* catalogDB, int recordIndex, CatalogRecord \*  $\times \rightarrow$  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.

```
1 void CatalogDB_writeRecord(CatalogDB * catalogDB, int recordIndex, CatalogRecord * record) {
2     if (catalogDB != NULL) {
3         if (recordIndex < 0 || recordIndex > catalogDB->recordCount)
4             fatalError("CatalogDB_SetRecord: invalid index");
5         fseek(catalogDB->file, ((long) sizeof(catalogDB->recordCount) + CATALOGRECORD_SIZE
6             * recordIndex, SEEK_SET);
7         CatalogRecord_write(record, catalogDB->file);
8         if (recordIndex == catalogDB->recordCount)
9             catalogDB->recordCount++;
10     }
11 }
```

## 2.8 Insertion d'un enregistrement au sein du fichier

Ecrire la fonction **void CatalogDB\_insertRecord(CatalogDB \* catalogDB, int recordIndex, CatalogRecord \*  $\times \rightarrow$  record)** qui insère le contenu d'un enregistrement à la position **recordIndex**.

```
1 void CatalogDB_insertRecord(CatalogDB * catalogDB, int recordIndex, CatalogRecord * record) {
2     if (catalogDB != NULL) {
3         int i;
4         CatalogRecord temp;
5         CatalogRecord_init(&temp);
6         for (i = CatalogDB_getRecordCount(catalogDB); i > recordIndex; --i) {
7             CatalogDB_readRecord(catalogDB, i - 1, &temp);
8             CatalogDB_writeRecord(catalogDB, i, &temp);
9         }
10         CatalogDB_writeRecord(catalogDB, recordIndex, record);
11         CatalogRecord_finalize(&temp);
12     }
13 }
```

## 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.

```
1 void CatalogDB_appendRecord(CatalogDB * catalogDB, CatalogRecord *record) {
2     CatalogDB_insertRecord(catalogDB, CatalogDB_getRecordCount(catalogDB), record);
3 }
```

## 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.

```
1 void CatalogDB_removeRecord(CatalogDB * catalogDB, int recordIndex) {
2     if (catalogDB != NULL) {
3         CatalogRecord temp;
4         int i;
5         CatalogRecord_init(&temp);
6         for (i = recordIndex; i < CatalogDB_getRecordCount(catalogDB) - 1; ++i) {
7             CatalogDB_readRecord(catalogDB, i + 1, &temp);
8             CatalogDB_writeRecord(catalogDB, i, &temp);
9         }
10        catalogDB->recordCount--;
11        CatalogRecord_finalize(&temp);
12    }
13 }
```



# 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.

```
1 void writeString(const char * str, FILE * file) {
2     size_t len;
3     len = strlen(str);
4     if (fwrite(&len, sizeof(len), 1, file) != 1)
5         fatalError("writeString");
6     if (len > 0)
7         if (fwrite(str, len, 1, file) != 1)
8             fatalError("writeString");
9 }
```

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.

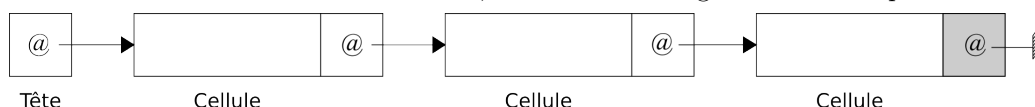
```
1 char * readString(FILE * file) {
2     size_t len;
3     char * result;
4
5     if (fread(&len, sizeof(len), 1, file) != 1)
6         fatalError("readString");
7     result = (char*) malloc(len + 1);
8     if (len > 0)
9         if (fread(result, len, 1, file) != 1)
10             fatalError("readString");
11     result[len] = '\0';
12     return result;
13 }
```

### 2 La liste des produits du document

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

```
1 /** Structure representing a row in a document (as a cell in a simple linked list) */
2 typedef struct _DocumentRow {
3     char * code /** The code */;
4     char * designation /** The designation */;
5     double quantity /** The quantity */;
6     char * unity /** The unity */;
7     double basePrice /** The base price */;
8     double sellingPrice /** The selling price */;
9     double discount /** The discount */;
10    double rateOfVAT /** The rate of VAT */;
11    struct _DocumentRow * next /** The pointer to the next row */;
12 } DocumentRow;
```

Lorsque plusieurs cellules sont chaînées 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.

```
1 void DocumentRow_init(DocumentRow * row) {
2     row->code = duplicateString("");
3     row->designation = duplicateString("");
4     row->quantity = 0;
5     row->unity = duplicateString("");
6     row->basePrice = 0;
7     row->sellingPrice = 0;
8     row->discount = 0;
9     row->rateOfVAT = 0;
10    row->next = NULL;
11 }
```

### 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.

```
1 void DocumentRow_finalize(DocumentRow * row) {
2     free(row->code);
3     free(row->designation);
4     free(row->unity);
5 }
```

### 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.

```
1 DocumentRow * DocumentRow_create(void) {
2     DocumentRow * row;
3
4     row = (DocumentRow *) malloc(sizeof(DocumentRow));
5     if (row == NULL)
6         fatalError("DocumentRow_Create");
7     DocumentRow_init(row);
8     return row;
9 }
```

### 2.1.4 Destruction d'une cellule allouée sur le tas

Ecrire la fonction **void DocumentRow\_destroy(DocumentRow \* row)** qui desalloue une cellule précédemment allouée par **DocumentRow\_create**.

```
1 void DocumentRow_destroy(DocumentRow * row) {
2     DocumentRow_finalize(row);
3     free(row);
4 }
```

### 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.

```
1 void DocumentRow_writeRow(DocumentRow * row, FILE * file) {
2     writeString(row->code, file);
3     writeString(row->designation, file);
4     if (fwrite(&row->quantity, sizeof(row->quantity), 1, file) != 1)
5         fatalError("writeRow");
6     writeString(row->unity, file);
7     if (fwrite(&row->basePrice, sizeof(row->basePrice), 1, file) != 1)
8         fatalError("writeRow");
9     if (fwrite(&row->sellingPrice, sizeof(row->sellingPrice), 1, file) != 1)
10        fatalError("writeRow");
11    if (fwrite(&row->discount, sizeof(row->discount), 1, file) != 1)
12        fatalError("writeRow");
13    if (fwrite(&row->rateOfVAT, sizeof(row->rateOfVAT), 1, file) != 1)
14        fatalError("writeRow");
15 }
```

### 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.

```
1 DocumentRow * DocumentRow_readRow(FILE * fichier) {
2     DocumentRow * row;
3
4     row = DocumentRow_create();
5     free(row->code);
6     row->code = readString(fichier);
7     free(row->designation);
8     row->designation = readString(fichier);
9     if (fread(&row->quantity, sizeof(row->quantity), 1, fichier) != 1)
10        fatalError("readRow");
11    free(row->unity);
12    row->unity = readString(fichier);
13    if (fread(&row->basePrice, sizeof(row->basePrice), 1, fichier) != 1)
14        fatalError("readRow");
15    if (fread(&row->sellingPrice, sizeof(row->sellingPrice), 1, fichier) != 1)
16        fatalError("readRow");
17    if (fread(&row->discount, sizeof(row->discount), 1, fichier) != 1)
18        fatalError("readRow");
19    if (fread(&row->rateOfVAT, sizeof(row->rateOfVAT), 1, fichier) != 1)
20        fatalError("readRow");
21    return row;
22 }
```

## 2.2 Manipulation de la liste

### 2.2.1 Initialisation d'une liste

Une liste chaînée 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.

```
1 /** Initialize a list of rows
2  * @param list the address of the pointer on the first cell of the list
3  */
4 void DocumentRowList_init(DocumentRow ** list) {
5     *list = NULL;
6 }
```

### 2.2.2 Destruction de liste

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

```
1 void DocumentRowList_finalize(DocumentRow ** list) {
2     while (*list != NULL) {
3         DocumentRowList_removeRow(list, *list);
4     }
5 }
```

### 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`.

```
1 DocumentRow * DocumentRowList_get(DocumentRow * list, int rowIndex) {
2     DocumentRow * cur;
3     int i;
4
5     cur = list;
6     i = 0;
7
8     if (rowIndex < 0)
9         return NULL;
10
11    while (cur != NULL && i != rowIndex) {
12        cur = cur->next;
13        i++;
14    }
15    return cur;
16 }
```

### 2.2.4 Nombre d'éléments d'une liste

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

```
1 int DocumentRowList_getRowCount(DocumentRow * list) {
2     int count = 0;
3
4     while (list != NULL) {
5         count++;
6         list = list->next;
7     }
8     return count;
9 }
```

### 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.

```
1 void DocumentRowList_pushBack(DocumentRow ** list, DocumentRow * row) {
2     if (*list == NULL)
3         *list = row;
4     else {
5         DocumentRow * cur = *list;
6         while (cur->next != NULL)
7             cur = cur->next;
8         cur->next = row;
9     }
10 }
```

### 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
    ↪ row);
```

```
1 void DocumentRowList_insertAfter(DocumentRow ** UNUSED(list), DocumentRow * position,
2     DocumentRow * row) {
3     row->next = position->next;
4     position->next = row;
5 }
```

### 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);
```

```
1 void DocumentRowList_insertBefore(DocumentRow ** list, DocumentRow * position, DocumentRow * row
    ↪ row) {
2     if (position != NULL) {
3         if (position == *list) {
4             row->next = position;
5             *list = row;
6         } else {
7             DocumentRow * previous = *list;
8             while (previous != NULL && previous->next != position)
9                 previous = previous->next;
10            if (previous != NULL) {
11                row->next = previous->next;
12                previous->next = row;
13            }
14        }
15    }
16 }
```



### 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.

```
1 void DocumentRowList_removeRow(DocumentRow ** list, DocumentRow * position) {
2     if (*list == position) {
3         *list = position->next;
4         DocumentRow_destroy(position);
5     } else {
6         DocumentRow * previous = *list;
7         while (previous != NULL && previous->next != position)
8             previous = previous->next;
9         if (previous != NULL)
10            previous->next = position->next;
11        DocumentRow_destroy(position);
12    }
13 }
```

## 3 Le document

Un document est défini grâce aux structures de données suivantes :

```
1 /** Enumeration defining the type of a document */
2 typedef enum {
3     QUOTATION /**< It's a quotation */ ,
4     BILL /**< It's a bill */
5 } TypeDocument;
6
7 /** Structure representing a document */
8 typedef struct {
9     CustomerRecord customer /** The customer */;
10    char * editDate /** The last edit data */;
11    char * expiryDate /** The peremption date */;
12    char * docNumber /** The document number */;
13    char * object /** The object of the document */;
14    char * operator /** The last operator */;
15    DocumentRow * rows /** The rows */;
16    TypeDocument typeDocument /** The type of document */;
17 } Document;
```

### 3.1 Initialisation

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

```
1 void Document_init(Document * document) {
2     CustomerRecord_init(&document->customer);
3     document->editDate = duplicateString("");
4     document->expiryDate = duplicateString("");
5     document->docNumber = duplicateString("");
6     document->object = duplicateString("");
7     document->operator = duplicateString("");
8     DocumentRowList_init(&document->rows);
9     document->typeDocument = QUOTATION;
10 }
```

### 3.2 Finalisation

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

```
1 void Document_finalize(Document * document) {
2     CustomerRecord_finalize(&document->customer);
3     free(document->editDate);
4     free(document->expiryDate);
5     free(document->docNumber);
6     free(document->object);
7     free(document->operator);
8     DocumentRowList_finalize(&document->rows);
9 }
```

### 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.

```
1 void Document_saveToFile(Document * document, const char * filename) {
2     DocumentRow * cur;
3     int count;
4     FILE * fichier;
5
6     count = DocumentRowList_getRowCount(document->rows);
7     fichier = fopen(filename, "wb");
8
9     CustomerRecord_write(&document->customer, fichier);
10    writeString(document->docNumber, fichier);
11    writeString(document->editDate, fichier);
12    writeString(document->object, fichier);
13    writeString(document->operator, fichier);
14    writeString(document->expiryDate, fichier);
15
16    if (fwrite(&document->typeDocument, sizeof(document->typeDocument), 1, fichier) != 1)
17        fatalError("Document_SaveToFile");
18
19    if (fwrite(&count, sizeof(count), 1, fichier) != 1)
20        fatalError("Document_SaveToFile");
21
22    cur = document->rows;
23    while (cur != NULL) {
24        DocumentRow_writeRow(cur, fichier);
25        cur = cur->next;
26    }
27
28    fclose(fichier);
29 }
```

### 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**.

```
1 void Document_loadFromFile(Document * document, const char * filename) {
2     int count, i;
3     FILE * fichier;
4
5     fichier = fopen(filename, "rb");
6     if (fopen == NULL)
7         fatalError("Echec ouverture fichier");
8
9     CustomerRecord_read(&document->customer, fichier);
10    free(document->docNumber);
11    document->docNumber = readString(fichier);
12    free(document->editDate);
13    document->editDate = readString(fichier);
14    free(document->object);
15    document->object = readString(fichier);
16    free(document->operator);
17    document->operator = readString(fichier);
18    free(document->expiryDate);
19    document->expiryDate = readString(fichier);
20
21    if (fread(&document->typeDocument, sizeof(document->typeDocument), 1, fichier) != 1)
22        fatalError("Document_LoadFromFile");
23
24    if (fread(&count, sizeof(count), 1, fichier) != 1)
25        fatalError("Document_LoadFromFile");
26
27    DocumentRowList_finalize(&document->rows);
28    for (i = 0; i < count; ++i)
29        DocumentRowList_pushBack(&document->rows, DocumentRow_readRow(fichier));
30
31    fclose(fichier);
32 }
```

# 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 délimitées 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 longueur 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 longueur 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 formater 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 Structures de données

Un dictionnaire est défini par les structures suivantes :

```

1
2 /** Enumeration defining the type of the doctionary entries */
3 typedef enum {
4     UNDEFINED_ENTRY /* It's undefined */,
5     NUMBER_ENTRY /* It's a number */,
6     STRING_ENTRY /* It's a string */
7 } DictionaryEntryType;
8
9 /** Structure representing an entry in the dictionary */
10 typedef struct {
11     /** The type of entry */
12     DictionaryEntryType type;
13     /** The name of the entry */
14     char * name;
15     /** The union which store the value of the entry */
16     union {
17         /** The value of the entry when it's a string */
18         char * stringValue;
19         /** The value of the entry when it's a real number */
20         double numberValue;
21     } value;
22 } DictionaryEntry;
23
24 /** Structure representing a dictionary */
25 typedef struct _Dictionary {
26     /** The number of entries of the dictionary */
27     int count;
28     /** The table of entries */
29     DictionaryEntry * entries;
30 } Dictionary;

```

Un dictionnaire est donc assimilable à un tableau dynamique.

Un modèle est défini par la structure suivante :

```

1
2 /** Structure holding the three format strings defining a model */
3 typedef struct _PrintFormat {
4     /** The name of the model */
5     char * name;
6     /** The header format */
7     char * header;
8     /** The row format */
9     char * row;
10    /** The footer format */
11    char * footer;
12 } PrintFormat;

```

Chaque modèle est décomposé en trois parties :

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

```

1 Dictionary * Dictionary_create(void) {
2     Dictionary * dictionary;
3
4     REGISTRY_USINGFUNCTION;
5
6     dictionary = (Dictionary*) malloc(sizeof(Dictionary));
7     if (dictionary == NULL)
8         fatalError("Dictionary_Create");
9     dictionary->count = 0;
10    dictionary->entries = NULL;
11    return dictionary;
12 }

```

## 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);

```

```

1 DictionaryEntry * Dictionary_getEntry(Dictionary * dictionary, const char * name) {
2     int i;
3
4     REGISTRY_USINGFUNCTION;
5
6     for (i = 0; i < dictionary->count; ++i)
7         if (icaseCompareString(dictionary->entries[i].name, name) == 0)
8             return &dictionary->entries[i];
9     return NULL;
10 }

```

## 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);

```

```

1 void Dictionary_setStringEntry(Dictionary * dictionary, const char * name,
2     const char * value) {
3     DictionaryEntry * entry;
4
5     REGISTRY_USINGFUNCTION;
6
7     entry = Dictionary_getEntry(dictionary, name);
8     if (entry == NULL) {
9         dictionary->entries = (DictionaryEntry*) realloc(dictionary->entries,
10             (size_t) (dictionary->count + 1) * sizeof(DictionaryEntry));
11         entry = &dictionary->entries[dictionary->count];
12         entry->name = duplicateString(name);
13         entry->type = UNDEFINED_ENTRY;
14         dictionary->count++;
15     }
16     if (entry->type == STRING_ENTRY)
17         free(entry->value.stringValue);
18     entry->type = STRING_ENTRY;
19     entry->value.stringValue = duplicateString(value);
20 }

```

## 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);

```

```

1 void Dictionary_setNumberEntry(Dictionary * dictionary, const char * name, double value) {
2     DictionaryEntry * entry;
3
4     REGISTRY_USINGFUNCTION;
5
6     entry = Dictionary_getEntry(dictionary, name);
7     if (entry == NULL) {
8         dictionary->entries = (DictionaryEntry*) realloc(dictionary->entries,

```

```

9         (size_t) (dictionary->count + 1) * sizeof(DictionaryEntry));
10     entry = &dictionary->entries[dictionary->count];
11     entry->name = duplicateString(name);
12     entry->type = UNDEFINED_ENTRY;
13     dictionary->count++;
14 }
15 if (entry->type == STRING_ENTRY)
16     free(entry->value.stringValue);
17 entry->type = NUMBER_ENTRY;
18 entry->value.numberValue = value;
19 }

```

## 2.5 Destruction d'un dictionnaire

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

```

1 void Dictionary_destroy(Dictionary * dictionary) {
2     int i;
3
4     REGISTRY_USINGFUNCTION;
5
6     for (i = 0; i < dictionary->count; ++i) {
7         free(dictionary->entries[i].name);
8         if (dictionary->entries[i].type == STRING_ENTRY)
9             free(dictionary->entries[i].value.stringValue);
10    }
11    free(dictionary->entries);
12    free(dictionary);
13 }

```

## 3 Manipulation du modèle

### 3.1 Création d'un modèle

Ecrire la fonction **void PrintFormat\_init(PrintFormat \* format)** qui permet de créer un modèle vide sur le tas.

```

1 void PrintFormat_init(PrintFormat * format) {
2     REGISTRY_USINGFUNCTION;
3
4     format->name = duplicateString("");
5     format->header = duplicateString("");
6     format->row = duplicateString("");
7     format->footer = duplicateString("");
8 }

```

### 3.2 Destruction d'un modèle

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

```

1 void PrintFormat_finalize(PrintFormat * format) {
2     REGISTRY_USINGFUNCTION;
3
4     free(format->name);
5     free(format->header);
6     free(format->row);
7     free(format->footer);
8 }

```

### 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 une 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.

```
1 static char * readLine(FILE * fichier) {
2     char buf[1024];
3     char * result = duplicateString("");
4     size_t len;
5
6     while (fgets(buf, 1024, fichier) != NULL) {
7         char * temp = concatenateString(result, buf);
8         free(result);
9         result = temp;
10
11         len = stringLength(buf);
12         if (len > 0)
13             if (buf[len - 1] == '\n') {
14                 return result;
15             }
16     }
17     return result;
18 }
19
20 void PrintFormat_loadFromFile(PrintFormat * format, const char * filename) {
21     char * line;
22     FILE * fichier;
23
24     REGISTRY_USINGFUNCTION;
25
26     fichier = fopen(filename, "rt");
27     if (fichier == NULL)
28         fatalError("PrintFormat_LoadFromFile");
29
30     line = readLine(fichier);
31     if (icaseStartWith(".NAME", line)) {
32         free(format->name);
33         line[stringLength(line) - 1] = '\0'; /* suppression du saut de ligne */
34         format->name = duplicateString(line + 6);
35     }
36     free(line);
37
38     free(format->header);
39     format->header = duplicateString("");
40     free(format->row);
41     format->row = duplicateString("");
42     free(format->footer);
43     format->footer = duplicateString("");
44
45     line = readLine(fichier);
46     if (icaseStartWith(".HEADER", line)) {
47         free(line);
48         line = readLine(fichier);
49
50         while (!icaseStartWith(".ROW", line)) {
51             char * temp = concatenateString(format->header, line);
52             free(format->header);
53             format->header = temp;
54             free(line);
55             line = readLine(fichier);
56         }
57         format->header[stringLength(format->header) - 1] = '\0'; /* suppression du saut de
↪ ligne */
58     }
59
60     free(line);
61     line = readLine(fichier);
62     while (!icaseStartWith(".FOOTER", line)) {
63         char * temp = concatenateString(format->row, line);
64         free(format->row);
65         format->row = temp;
66         free(line);
67         line = readLine(fichier);
```

```

68     }
69     format->row[stringLength(format->row) - 1] = '\0'; /* suppression du saut de ligne */
70
71     free(line);
72     line = readLine(fichier);
73     while (!strcasecmp(format->footer, line)) {
74         char * temp = concatenateString(format->footer, line);
75         free(format->footer);
76         format->footer = temp;
77         free(line);
78         line = readLine(fichier);
79     }
80     format->footer[stringLength(format->footer) - 1] = '\0'; /* suppression du saut de ligne */
81     free(line);
82
83     fclose(fichier);
84 }

```

## 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.

```

1  char * Dictionary_format(Dictionary * dictionary, const char * format) {
2      char * result, *temp;
3      char * start, *end;
4      char * name, *parameters;
5      DictionaryEntry * entry;
6
7      REGISTRY_USINGFUNCTION;
8
9      start = indexOfChar(format, '%');
10     if (start == NULL) {
11         result = duplicateString(format);
12     } else {
13         result = subString(format, start);
14
15         while (start != NULL) {
16             end = indexOfChar(start + 1, '%');
17             if (end == NULL) {
18                 fprintf(stderr, "Fermeture de balise manquante\n");
19                 end = start + stringLength(start);
20             }
21
22             if (start + 1 == end) {
23                 temp = concatenateString(result, "%");
24                 free(result);
25                 result = temp;
26             } else {
27                 char * lbrace = indexOfChar(start + 1, '{');
28                 if (lbrace == NULL) {
29                     name = subString(start + 1, end);
30                     parameters = duplicateString("");
31                 } else if (lbrace > end) {
32                     name = subString(start + 1, end);
33                     parameters = duplicateString("");
34                 } else {
35                     char * rbrace = indexOfChar(lbrace + 1, '}');
36                     if (rbrace == NULL || rbrace > end) {
37                         fprintf(stderr, "Accolade fermante manquante dans une balise\n");

```



```

38         rbrace = end - 1;
39     }
40     name = subString(start + 1, lbrace);
41     parameters = subString(lbrace + 1, rbrace);
42 }
43
44 entry = Dictionary_getEntry(dictionary, name);
45 if (entry == NULL) {
46     fprintf(stderr, "Variable %s non definie donc ignoree\n", name);
47 } else {
48     char * value;
49     if (entry->type == STRING_ENTRY) {
50         char * caseIndex = indexOfString(parameters, "case=");
51         char * minIndex = indexOfString(parameters, "min=");
52         char * maxIndex = indexOfString(parameters, "max=");
53
54         value = duplicateString(entry->value.stringValue);
55         if (caseIndex != NULL) {
56             if (toLowerCaseChar(*(caseIndex + 5)) == 'u')
57                 makeUpperCaseString(value);
58             else
59                 makeLowerCaseString(value);
60         }
61
62         if (minIndex != NULL) {
63             size_t minLength = (size_t) atoi(minIndex + 4);
64             size_t len = stringLength(value);
65             if (len < minLength) {
66                 size_t i;
67                 temp = (char*) malloc(minLength + 1);
68                 for (i = 0; i < len; ++i)
69                     temp[i] = value[i];
70                 for (i = len; i < minLength; ++i)
71                     temp[i] = ' ';
72                 temp[minLength] = '\0';
73                 free(value);
74                 value = temp;
75             }
76         }
77
78         if (maxIndex != NULL) {
79             size_t maxLength = (size_t) atoi(maxIndex + 4);
80             size_t len = stringLength(value);
81             if (len > maxLength)
82                 value[maxLength] = '\0';
83         }
84     } else {
85         char * precisionIndex = indexOfString(parameters, "precision=");
86         char * minIndex = indexOfString(parameters, "min=");
87         int precision, min;
88         char buf[1024];
89
90         if (precisionIndex != NULL)
91             precision = atoi(precisionIndex + 10);
92         else
93             precision = -1;
94
95         if (minIndex != NULL)
96             min = atoi(minIndex + 4);
97         else
98             min = 0;
99
100         if (precision < 0)
101             sprintf(buf, "%*f", min, entry->value.numberValue);
102         else
103             sprintf(buf, "%*.*f", min, precision, entry->value.numberValue);
104         value = duplicateString(buf);
105     }
106
107     temp = concatenateString(result, value);
108     free(result);
109     free(value);
110     result = temp;
111 }
112 free(name);
113 free(parameters);
114 }
115 start = indexOfChar(end + 1, '%');
116
117 if (start != NULL) {
118     char * sub = subString(end + 1, start);
119     temp = concatenateString(result, sub);
120     free(sub);
121     free(result);
122     result = temp;
123 }
124 }
125
126 temp = concatenateString(result, end + 1);

```

```
127     free(result);  
128     result = temp;  
129 }  
130  
131 return result;  
132 }
```