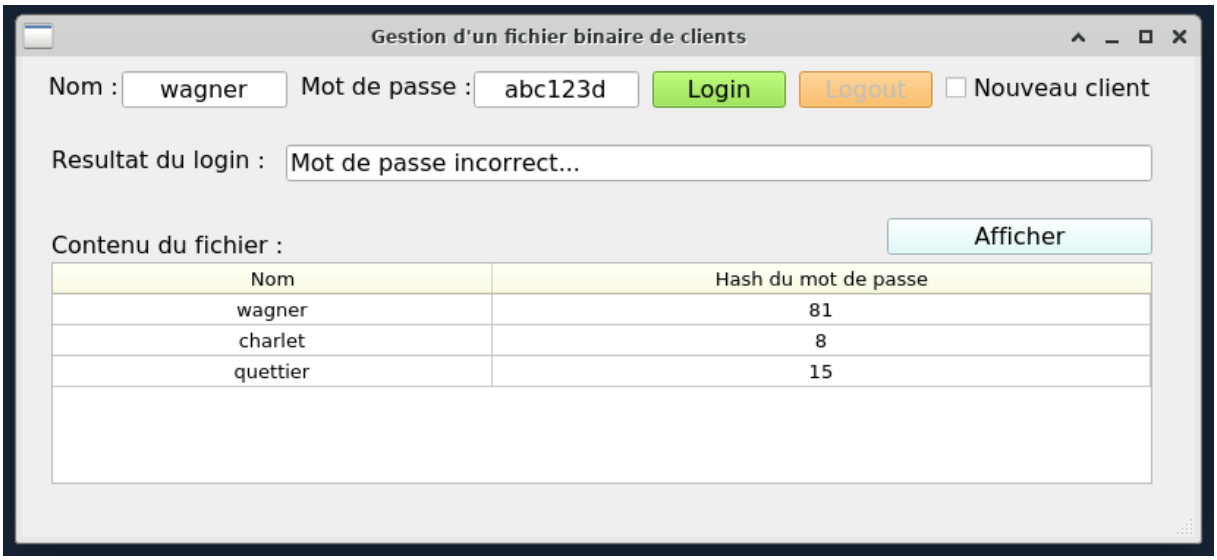
Exercice n°2 : le makefile et les fichiers de bas niveau

Objectifs :

- Se familiariser avec l'utilitaire **make** et la création d'un fichier **makefile**
- Se familiariser avec l'utilisation de la librairie graphique Qt
- Effectuer ses premiers accès **fichiers** de bas niveau (**open**, **close**, **read**, **write** et **lseek**)

Description générale :

L'application ressemble visuellement à



Elle permet de « simuler » l'entrée en session d'un client dans une application de vente. Pour cela, un client possède un nom et un mot de passe. L'application devra ici simplement gérer le fichier des mots de passe (plutôt leur « hash » ; voir plus loin) des clients.

La fenêtre graphique a été créée pour vous et vous est fournie sous la forme d'un ensemble de fichiers sources.

Lien GitHub : https://github.com/hepl-dsoo/LaboUnix2022_Exercice2

Etape 1 : Création d'un fichier makefile

On vous fournit les fichiers suivants :

- **main.cpp** : le main de votre application
- **mywindow.cpp** : code source de la fenêtre de l'application, fichier que vous devez modifier.
- **mywindows.h** : fichier header correspondant au fichier mywindow.cpp.
- **moc_mywindow.cpp** et **ui_mywindow.h** : fichiers sources propres à Qt nécessaires à la fenêtre graphique.
- **FichierClient.cpp** et **FichierClient.h** : fichiers sources (que vous devez modifier) contenant les fonctions de gestion du fichier des clients et de leurs mots de passe.
- **Compile.sh** : fichier texte (on appelle ça un script) contenant les lignes de compilation nécessaires pour la création du fichier exécutable « UNIX_Exercice2 ».

Il est déjà possible de compiler l'application en utilisant le fichier **Compile.sh** :

```
# sh Compile.sh
# UNIX_Exercice2
```

Le but est donc de créer un fichier **makefile** permettant de réaliser la compilation de tous les fichiers nécessaires à la création de l'exécutable UNIX_Exercice2.

Etape 2 : Programmation des fonctions d'accès fichier

Pour des raisons de sécurité, il est rare que l'on mémorise les mots de passe des utilisateurs en clair dans un fichier. Une solution consiste à stocker à la place le « **hash** » du mot de passe. Pour calculer un hash, on utilise une **fonction de hashage** (comme MD5 ou SHA1). Pour cet exercice, nous nous contenterons d'une fonction de hashage très simple : elle consiste à calculer la somme (modulo 97) pondérée (par leur position) des codes ASCII des caractères constituant le mot de passe. Illustrons cela sur un exemple :

Soit le mode passe « abc123 » pour lequel nous pouvons construire la table

Position dans le mot de passe	Caractère	Code ASCII
1	a	97
2	b	98
3	c	99
4	1	49
5	2	50
6	3	51

Le hash est alors calculé par : (1 * 97 + 2 * 98 + 3 * 99 + 4 * 49 + 5 * 50 + 6 * 51) % 97 = 1342 % 97 = 81. Il s'agit du mot de passe correct de « wagner » dans la capture d'écran ci-dessus. Le hash de tout mot de passe est donc un nombre entier compris entre 0 et 96.

Des mots de passe différents peuvent générer des hash identiques mais il est quasiment impossible retrouver un mot de passe à partir de son hash.

Vous trouverez dans le fichier **FichierClient.h** :

```
typedef struct
{
  char  nom[20];
  int   hash;
} CLIENT;
```

ainsi que le prototype des fonctions de gestion du fichier. Le fichier **FichierClient.cpp** contient déjà les définitions des fonctions que vous devez implémenter. Le fichier « des mots de passe » est un fichier binaire où différentes structures du type CLIENT sont enregistrées séquentiellement.

Vous devez donc implémenter les différentes fonctions de telle sorte que

- La fonction **estPresent**
  - Tente d'ouvrir en lecture seule le fichier dont le nom se trouve dans la macro **FICHIER_CLIENTS** définie dans **FichierClient.h**. Si le fichier n'existe pas, la fonction retourne -1. Sinon,
  - Lit séquentiellement le fichier structure par structure jusqu'au moment où elle trouve le client dont le nom est passé en paramètre à la fonction. Si le client est trouvé, la fonction retourne la position dans le fichier (la position : 1,2,3, … et non l'indice !). Si le client n'est pas trouvé, la fonction retourne 0.

- La fonction **hash**
  - Reçoit en paramètre un mot de passe
  - Retourne le hash de ce mot de passe construit en suivant l'algorithme décrit plus haut.

- La fonction **ajouteClient**
  - Tente d'ouvrir le fichier en (écriture seule + écriture en fin de fichier). Si le fichier n'existe pas, elle le crée.
  - Enregistre en fin de fichier une structure CLIENT contenant le nom et le hash du mot de passe reçus en paramètre par la fonction.

- La fonction **verifieMotDePasse**
  - Tente d'ouvrir le fichier en lecture seule. Si le fichier n'existe pas, elle retourne -1. Sinon,
  - Elle lit dans le fichier la structure située à la position reçue en paramètre par la fonction.
  - Elle compare le hash du mot de passe reçu en paramètre par la fonction et le hash contenu dans la structure lue sur disque. S'ils sont identiques, on considère que c'est correct et la fonction retourne 1. Sinon, la procédure de login échoue et la fonction retourne 0.

- La fonction **listeClients**
  - Reçoit en paramètre l'adresse d'un vecteur de clients suffisamment grand pour contenir tous les clients contenus dans le fichier.
  - Lit l'ensemble des clients du fichier et les place dans le vecteur.
  - Retourne le nombre de clients lus.

IMPORTANT : Pour implémenter ces fonctions, vous devez utiliser les appels système **open**, **read**, **write**, **lseek** et **close**.

Etape 3 : Manipulation du module « FichierClient » par l'interface graphique

A) Un clic sur le **bouton « Login »** doit récupérer le nom, le mot de passe et un entier (valant 1 ou 0 selon que l'on désire créer un nouveau client ou non). Deux cas de figure :

- Soit l'utilisateur veut créer un nouveau client :
  - Si le nom encodé correspond à un client déjà présent dans le fichier, l' « entrée en session » échoue et on affiche « Client déjà existant ! » dans la zone « Résultat du login : » (utilisation de la méthode **setResultat**).
  - Si le nom encodé ne correspond à aucun client existant, on ajoute un nouveau client dans le fichier. L'entrée en session est un succès et on affiche le message « Nouveau client créé : bienvenue ! »
- Soit l'utilisateur ne veut pas créer un nouveau client :
  - Si le nom encodé correspond à un client présent dans le fichier, le mot de passe encodé par l'utilisateur est vérifié sur base du hash qui se trouve dans le fichier. Si le mot de passe est correct, on affiche le message « Re-bonjour cher client ! ». Si le mot de passe est incorrect, on affiche le message « Mot de passe incorrect… ».
  - Si le nom encodé ne correspond à aucun client existant, on affiche le message « Client inconnu… »

Pour cela, vous devez modifier la fonction

void MyWindow::on_pushButtonLogin_clicked()

et utiliser les fonctions du module **FichierClient** que vous avez implémentées à l'étape 2.

Pour récupérer le nom, le mot de passe encodé par l'utilisateur et savoir s'il a coché la checkbox « Nouveau client », vous disposez des méthodes

- **const char * MyWindow::getNom();**
- **const char * MyWindow::getMotDePasse();**
- **int MyWindow::isNouveauClientChecked();**

tandis que pour afficher le résultat du login, vous disposez de la méthode

- **void MyWindow::setResultat(const char *Text);**

B) Un clic sur le **bouton « Afficher »** affiche le contenu du fichier dans la table située dans le bas de la fenêtre.

Pour cela, vous devez modifier la fonction

void MyWindow::on_pushButtonAfficheFichier_clicked()

et utiliser la fonction **listeClients** du module **FichierClient** que vous avez implémentée à l'étape 2.

Pour manipuler la table, vous disposez des méthodes

- **void MyWindow::ajouteTupleTableClients(const char *nom, int hash);** qui ajoute une ligne à la table avec les éléments passés en paramètres.
- **void MyWindow::videTableClients();** qui vide complètement la table.