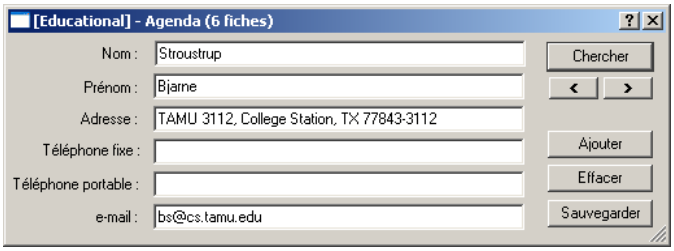
Centre Informatique pour les Lettres et les Sciences Humaines

TD 11 : Agenda

Le programme que nous allons écrire offre une gestion rudimentaire d'un agenda électronique. Outre les possibilités de création et de suppression de fiches descriptives, le programme permet une recherche basée sur la saisie d'une fiche incomplète (uniquement un prénom, par exemple). A la suite de cette recherche, il est possible de consulter *toutes* les fiches répondant au(x) critère(s) fixé(s).



L'interface utilisateur du programme réalisé au cours du TD 11

1 - Analyse préalable

Il semble évident que ce programme repose sur la constitution d'une collection de données, et que ces données seront d'un type créé spécialement pour l'occasion.

Du point de vue de l'utilisateur, l'accès aux données ne se fera en général pas séquentiellement (la seule exception correspond au cas d'un utilisateur parcourant l'ensemble des fiches, peut-être parce qu'il n'arrive à se souvenir d'aucun élément d'information lui permettant de lancer une recherche...). Cette imprévisibilité de l'ordre d'affichage des fiches semble plaider contre l'adoption d'une simple liste, mais, si l'on examine la question du point de vue du programme, c'est pourtant ce type de collection qui s'impose.

Les opérations de recherche peuvent, en effet, reposer sur n'importe lequel des champs proposés dans une fiche, et il n'est donc pas très simple de proposer un accès direct à la fiche recherchée, à partir d'une clé : il faudrait créer autant de tables qu'il y a de champs, et nous devrions mettre en place des mécanismes de prise en charge des "collisions" (deux fiches différentes peuvent avoir la même valeur dans un champ donné) et des éventuelles recherches "multicritères" (la multiplication des systèmes clé/valeur ne permet pas de faire un accès direct à une valeur correspondant, dans deux systèmes différents, à deux clés données).

Nous devons donc nous résoudre à utiliser une liste et à accepter l'idée d'une recherche basée sur un simple parcours exhaustif avec examen de toutes les valeurs rencontrées. Nous nous consolerons en remarquant que les opérations de gestion (les suppressions, notamment) seront, pour leur part, favorisée par ce choix.

> Il est clair que, dans le contexte du pseudo-agenda électronique qui nous occupe ici, la discussion des mérites respectifs d'une structure de données par rapport à une autre peut sembler oiseuse : même avec quelques milliers de fiches, aucun utilisateur ne remarquera jamais le moindre délai de réponse lors d'une recherche, et il est peu vraisemblable que qui que ce soit se donne un jour la peine de créer plus que les cinq où six fiches suffisant à vérifier le bon fonctionnement du programme. Les techniques que nous allons utiliser peuvent cependant être facilement employées dans d'autres contextes, qui peuvent donner lieu à des milliers de requêtes lancées automatiquement et portant sur des collections considérables. Il est donc préférable d'avoir conscience des raisons qui motivent ici le choix d'une simple liste.

Le choix de cette structure de donnée épuise les questions d'analyse préalable : l'aspect de l'interface suggérée ci-dessus et le principe du moindre effort dictent ensuite quasiment tous les détails du projet.

2 - Création du projet et dessin de l'interface

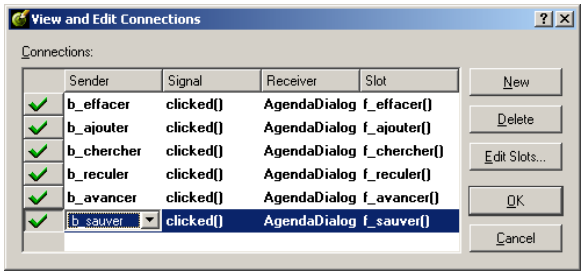
En vous inspirant de la procédure décrite dans l'aide mémoire, créez un projet nommé TD11 ☐.

Disposez sur votre dialogue six lineEdit respectivement nommés `nom`, `prenom`, `adresse`, `fixe`, `portable` et `mail` ☐.

Ajoutez les textLabel indiquant la nature de l'information affichée par chacun des six textEdit ☐.

Ajoutez six pushButton respectivement nommés `b_chercher`, `b_reculer`, `b_avancer`, `b_ajouter`, `b_effacer`, et `b_sauver` ☐.

Créez les slots et les connections suggérées par l'image ci-contre ☐.



3 - La classe dialogue

Le seul aspect important de notre programme qui n'a pas encore été évoqué est la "persistance" des données : les fiches créées lors d'une session de travail doivent évidemment pouvoir être utilisées lors d'une session ultérieure, ce qui impose le recours à un fichier de données. Plutôt que de demander à l'utilisateur de désigner ce fichier (ce qui serait sans doute d'un coût trop important étant donné le faible intérêt du programme), nous allons faire en sorte que le lancement du programme s'accompagne automatiquement de la lecture des données contenues dans un fichier nommé agenda.txt et situé dans le même dossier que le programme lui-même. Le code prenant cette opération en charge est évidemment placé dans

Le constructeur

A la fin de l'exécution du programme, il faudra signaler une éventuelle modification des données qui n'aurait pas été suivie d'une sauvegarde. Cette contrainte impose de recourir à deux variables membre : l'une pour stocker le chemin d'accès au fichier de données, l'autre pour indiquer si les données ont été modifiées depuis la dernière sauvegarde. Une autre variable membre est évidemment nécessaire : la collection dont l'exploitation est la raison d'être du programme.

```
1  TD11DialogImpl::TD11DialogImpl( QWidget* parent, const char* name, bool modal, WFlags f )
                                                  : TD11Dialog( parent, name, modal, f )
2  {
3  QFileInfo info(*qApp->argv());
4  m_chemin = info.dirPath() + "/agenda.txt";
5  QFile fich(m_chemin);
6  if(fich.open(IO_ReadOnly | IO_Translate))
7     {
8     QTextStream fichier(&fich);
9     CFiche lue;
10    while (!fich.atEnd())
11       {
12       fichier >> lue;
13       m_agenda.append(lue);
14       }
15    }
16 m_fichierAJour = true;
17 }
```

1

La technique utilisée *(3-4)* pour créer le chemin d'accès au fichier de données est celle présentée dans l'aide mémoire "Accessoires divers".

Plutôt que de chercher à exploiter lui-même le contenu du fichier, le constructeur de la classe de dialogue s'appuie sur la classe que nous créerons pour représenter nos données et suppose que :

1 : La classe `CFiche` dispose d'un opérateur d'extraction qui permet de transférer directement une valeur d'un `QTextStream` vers une instance.

Dans le fichier TD11DialogImpl.cpp, ajoutez les lignes *3-16* au constructeur de la classe .

La fonction `f_sauver()`

Pour que le constructeur puisse lire des données dans le fichier, il faut les y écrire...

```
1  void TD11DialogImpl::f_sauver()
2  {
3  QFile fich(m_chemin);
4  if(fich.open(IO_WriteOnly | IO_Translate))
5     {
6     QTextStream fichier(&fich);
7     QValueList<CFiche>::Iterator it;
8     for(it = m_agenda.begin() ; it != m_agenda.end() ; ++it)
9        fichier << *it;
10    }
11 m_fichierAJour = true;
12 }
```

2

2 : La classe `CFiche` dispose d'un opérateur d'insertion qui permet de transférer directement la valeur d'une instance dans un `QTextStream`.

Ajoutez la fonction `f_sauver()` à votre classe `TD011DialogImpl` ☐ et donnez-lui le contenu suggéré ci-dessus ☐.

La fonction `f_ajouter()`

Cette fonction doit transférer le contenu des lineEdit dans une instance de la classe `CFiche` *(3)*, puis ajouter cette valeur à la collection :

```
1  void TD11DialogImpl::f_ajouter()
2  {
3  CFiche aAjouter(nom->text(), prenom->text(), adresse->text(),
                                    fixe->text(), portable->text(), mail->text());
4  if(m_agenda.contains(aAjouter))
5     QMessageBox::warning(this,"Agenda","Il existe déjà une fiche identique !");
6  else
7     {
8     m_agenda.append(aAjouter);
9     m_fichierAJour = false; //puisqu'on vient de modifier la collection
10    }
11 }
```

3 4

Le code proposé ci-dessus impose deux nouvelles contraintes à la classe `CFiche` :

3 : La classe `CFiche` doit disposer d'un constructeur acceptant six paramètres de type `QString` indiquant le contenu des rubriques.

> La mise en place de ce constructeur ne devra bien sûr pas priver la classe du constructeur par défaut rendu obligatoire par l'utilisation d'une `QValueList` de `CFiche`.

4 : La classe `CFiche` doit disposer d'un opérateur == permettant de comparer deux instances.

> La fonction `contains()des QValueList` est l'une de celles qui ont cette exigence (cf. Leçon 11).

Ajoutez la fonction `f_ajouter()` à votre classe `TD011DialogImpl` ☐ et donnez-lui le contenu suggéré ci-dessus ☐.

La fonction `f_chercher()`

Les `QValueList` disposent d'une fonction `find()`, qui repose sur l'utilisation de l'opérateur == de la classe de données. Puisque nous venons de voir que cet opérateur sera nécessaire, rien ne s'oppose à ce que nous utilisions cette fonction.

Le problème principal que pose cette fonction est qu'elle est susceptible de découvrir plusieurs fiches correspondant aux critères fixés, alors que l'interface utilisateur ne permet d'en afficher qu'une à la fois. Il nous faut donc mettre en place un dispositif annexe permettant à l'utilisateur de "naviguer" dans l'ensemble des fiches trouvées. Ce dispositif repose sur deux nouvelles variables membre : l'une est une `QMap` servant à stocker une copie des fiches intéressantes, alors que l'autre sert à indiquer laquelle d'entre-elles doit être présentée à l'écran.

```
1  void TD11DialogImpl::f_chercher()
2  {
3  m_trouvees.clear();
4  CFiche aChercher(nom->text(), prenom->text(), adresse->text(),
                                    fixe->text(), portable->text(), mail->text());
5  QValueList<CFiche>::Iterator it = m_agenda.find(aChercher);
6  while(it != m_agenda.end())
7     {
8     m_trouvees[m_trouvees.count()] = *it;
9     it = m_agenda.find(++it, aChercher);
10    }
11 m_aAfficher = 0;
12 afficheResultat();
13 }
```

Le transfert à l'écran des informations contenues dans une fiche est sous-traité *(12)* à la fonction `afficheResultat()`, ce qui simplifiera la navigation à l'aide des boutons [<] et [>] associés aux fonctions `f_reculer()` et `f_avancer()`.

Ajoutez la fonction `f_chercher()` à votre classe `TD011DialogImpl` ☐ et donnez-lui le contenu suggéré ci-dessus ☐.

La fonction `afficheResultat()`

Cette fonction doit utiliser le contenu de la variable `m_aAfficher` pour choisir, dans la collection `m_trouvees`, la fiche qui doit apparaître.

```
1  void TD11DialogImpl::afficheResultat()
2  {
3  if(m_trouvees.count() == 0)
4    return;
5  nom->setText(m_trouvees[m_aAfficher].nom());
6  prenom->setText(m_trouvees[m_aAfficher].prenom());
7  adresse->setText(m_trouvees[m_aAfficher].adresse());
8  fixe->setText(m_trouvees[m_aAfficher].fixe());
9  portable->setText(m_trouvees[m_aAfficher].portable());
10 mail->setText(m_trouvees[m_aAfficher].mail());
   //désactive un bouton lorsqu'on arrive en bout de collection
11 b_reculer->setEnabled(m_aAfficher > 0);
12 b_avancer->setEnabled(m_trouvees.count() > m_aAfficher+1);
13 }
```

5

5 : La classe `CFiche` doit offrir des fonctions `nom()`, `prenom()`, `adresse()`, `fixe()`, `portable()` et `mail()` renvoyant les valeurs contenues dans les champs correspondants.

Ajoutez la fonction `afficheResultat()` à votre classe `TD011DialogImpl` ☐ et donnez-lui le contenu suggéré ci-dessus ☐.

Les fonctions `f_avancer()` et `f_reculer()`

Comme la désactivation des boutons qui les appellent interdit tout dépassement des limites de la collection `m_trouvees`, ces deux fonctions peuvent se contenter d'ajuster la valeur de `m_aAfficher` et d'appeler `afficheResultat()`.

```
1  void TD11DialogImpl::f_avancer()
2  {
3  ++ m_aAfficher;
4  afficheResultat();
5  }
```

```
1  void TD11DialogImpl::f_reculer()
2  {
3  -- m_aAfficher;
4  afficheResultat();
5  }
```

Ajoutez les fonctions `f_reculer()` et `f_avancer()` à votre classe `TD011DialogImpl` ☐ et donnez-leur le contenu suggéré ci-dessus ☐.

La fonction `f_effacer()`

Le code de cette fonction est essentiellement composé de lignes *(4-11)* sécurisant l'opération : étant donné que les champs vides sont considérés comme non-spécifiés (et, donc, rendent acceptable n'importe quelle valeur observée dans une fiche), il est très facile de lancer par inadvertance l'effacement d'un grand nombre de fiches.

Cliquer sur [Effacer] alors que la fiche présente à l'écran est vierge conduit, par exemple, à effacer TOUTES les fiches de la collection.

Toute opération qui se solderait par l'effacement de plusieurs fiches fera donc l'objet d'une demande de confirmation mentionnant ce nombre :

```
1  void TD11DialogImpl::f_effacer()
2  {
3  CFiche aEffacer(nom->text(), prenom->text(), adresse->text(),
                                    fixe->text(), portable->text(), mail->text());
4  int nb = m_agenda.contains(aEffacer); //y a-t-il plusieurs victimes ?
5  QString message = "Etes-vous certains de vouloir détruire ces %1 fiches ?";
6  int reponse = QMessageBox::Yes;
7  if (nb > 0)
8     reponse = QMessageBox::warning(this,"Agenda",message.arg(nb),
                                                  QMessageBox::Yes, QMessageBox::No);
9  if (reponse != QMessageBox::Yes)
10    return;
11 QValueList<CFiche>::Iterator victime = m_agenda.find(aEffacer);
12 while(victime != m_agenda.end())
13    victime = m_agenda.find(m_agenda.remove(victime), aEffacer);
14 m_fichierAJour = false;
15 }
```

Ajoutez la fonction `f_effacer()` à votre classe `TD011DialogImpl` ☐ et donnez-lui le contenu suggéré ci-dessus ☐.

La fonction `done()`

La fermeture d'un `QDialog` est provoquée par l'exécution d'une fonction nommée `done()`. Si notre classe de dialogue définit cette fonction, nous pouvons y spécifier des traitements qui seront exécutés lors d'une demande de fermeture du dialogue.

Dans le cas présent, il s'agit simplement d'obtenir une confirmation dans le cas où les données ont été modifiées depuis la dernière sauvegarde :

```
1  void TD11DialogImpl::done(int r)
2  {
3  int reponse = QMessageBox::Yes;
4  if(!m_fichierAJour)
5     reponse = QMessageBox::warning(this,"Agenda", "Quitter sans enregistrer les "
                              "modifications ?", QMessageBox::Yes, QMessageBox::No);
6  if(reponse == QMessageBox::Yes)
7     QDialog::done(r);
8  }
```

> Remarquez que, si le fichier est à jour ou si l'utilisateur confirme l'abandon des modifications qu'il a faites, la fonction `TD11DialogImpl::done()` doit appeler la fonction `QDialog::done()` pour que le dialogue se ferme effectivement *(7)*. Les rapports entre ces deux fonctions vous apparaîtront plus clairement dès la Leçon 13.

Ajoutez la fonction `done()` à votre classe `TD011DialogImpl` ☐ et donnez-lui le contenu suggéré ci-dessus ☐.

Les variables membre

L'ajout des déclarations des variables membre que nous avons supposées présentes ☐ devrait vous conduire à un fichier td11dialogimpl.h ayant le contenu suivant :

```
1  #include "td11dialog.h"
2  #include "fiche.h"
3  class TD11DialogImpl : public TD11Dialog
4  {
5     Q_OBJECT
6  public:
7     TD11DialogImpl(QWidget* parent=0, const char* name=0, bool modal=FALSE, WFlags f=0 );
8     void f_effacer();
9     void f_reculer();
10    void f_avancer();
11    void afficheResultat();
12    void f_chercher();
13    void f_sauver();
```

```
14    void f_ajouter();
15    void done(int r);
16  protected:
17    QValueList <CFiche> m_agenda;
18    QMap <int, CFiche> m_trouvees;
19    QString m_chemin;
20    bool m_fichierAJour;
21    int m_aAfficher;
22  };
```

3 - La classe CFiche

Les "promesses" que l'écriture des fonctions membre de la classe de dialogue nous a conduit à faire sont finalement peu nombreuses :

	Origine	Nature de la promesse
1	`TD11DialogImpl()`	`QTextStream & operator >> (QTextStream & flux, CFiche & aLire)`
2	`f_sauver()`	`QTextStream & operator << (QTextStream & flux, CFiche & aEcrire)`
3	`f_ajouter()`	`CFiche(QString, QString, QString, QString, QString, QString)`
4	`f_ajouter()`	`bool CFiche::operator == (const CFiche & autre) const`
5	`afficheResultat()`	6 fonctions membre renvoyant chacune la valeur d'un champ

Ajouter (en vous aidant, au besoin, de l'aide mémoire "Créer une classe") une classe nommée `CFiche` à votre projet ☐.

Ajouter à cette classe des variables membre de type `QString` nommées `m_nom`, `m_prenom`, `m_adresse`, `m_fixe`, `m_portable` et `m_mail` ☐.

Le constructeur à arguments multiples

Ajouter à votre classe la fonction suivante ☐.

```
1  CFiche::CFiche(QString n,QString p,QString a,QString tf,QString tp,QString m)
             : m_nom(n),m_prenom(p),m_adresse(a),m_fixe(tf),m_portable(tp),m_mail(m)
2  {
   //les initialisations sont faites, le corps reste vide
3  }
```

Si vous utilisez la commande "Add member function", n'oubliez pas que les constructeurs sont dépourvus de type : le champ "Function type" du dialogue proposé par Visual C++ doit donc rester vide. La liste d'initialisation doit, bien entendu, être ensuite ajoutée "à la main" dans le fichier fiche.cpp.

> Plutôt que de créer un second constructeur, il aurait aussi été possible d'ajouter des paramètres au constructeur prévu par Microsoft, en veillant à ce que chacun d'entre eux dispose d'une valeur par défaut (car la création d'une `QValueList` de `CFiche` impose qu'on puisse instancier cette classe sans fournir de valeurs d'initialisation).

La fonction `operator == ()`

Cette fonction est au cœur du programme : la valeur qu'elle renvoie va déterminer si les deux instances qu'elle compare doivent être considérées comme "identiques", ce qui, dans le contexte de notre agenda, signifie que l'une correspond aux critères de recherche spécifiés par l'autre.

Pour que ce soit le cas, il faut que les valeurs observées dans chacun des six champs (nom, prénom, etc.) soient "compatibles". Nous écrirons donc :

```
1  bool CFiche::operator ==(const CFiche &autre) const
2  {
3  if(!compatibles(m_nom, autre.m_nom))
4    return false;
5  if(!compatibles(m_prenom, autre.m_prenom))
6    return false;
7  if(!compatibles(m_adresse, autre.m_adresse))
8    return false;
```

```
9   if(!compatibles(m_fixe, autre.m_fixe))
10    return false;
11  if(!compatibles(m_portable, autre.m_portable))
12    return false;
13  if(!compatibles(m_mail, autre.m_mail))
14    return false;
15  return true;
16  }
```

Deux valeurs sont compatibles si elles sont identiques ou si l'une d'entre elles est "non spécifiée", c'est à dire correspond à une chaîne vide :

```
1   bool CFiche::compatibles(const QString & s1, const QString & s2) const
2   {
3   if(s1.isEmpty() || s2.isEmpty())
4     return true;
5   return s1 == s2;
6   }
```

Ajoutez les fonctions `operator ==()` et `compatibles()` à votre classe `CFiche` ☐ et donnez-leur le contenu suggéré ci-dessus ☐.

Les fonctions `operator >> ()` et `operator << ()`

Comme exposé dans la Leçon 11, ces opérateurs doivent être pris en charge par des fonctions globales que la classe `CFiche` devra déclarer amies :

```
1   QTextStream & operator << (QTextStream & flux, const CFiche & f)
2   {
3   flux << f.m_nom << "\n" << f.m_prenom << "\n" << f.m_adresse << "\n";
4   flux << f.m_fixe << "\n" << f.m_portable << "\n" << f.m_mail << "\n";
5   return flux;
6   }
```

```
1   QTextStream & operator >> (QTextStream & flux, CFiche & f)
2   {
3   f.m_nom = flux.readLine();
4   f.m_prenom = flux.readLine();
5   f.m_adresse = flux.readLine();
6   f.m_fixe = flux.readLine();
7   f.m_portable = flux.readLine();
8   f.m_mail = flux.readLine();;
9   return flux;
10  }
```

Ajoutez le code ci-dessus à la fin de votre fichier fiche.cpp ☐.

> Bien que ces fonctions ne soient pas membre de `CFiche`, leur raison d'être et le lien d'amitié qui les lie à cette classe justifient, à mes yeux, que leur code figure dans le fichier a priori destiné à l'implémentation des fonctions membre de `CFiche`.

Le fichier fiche.h

Les fonctions `nom()`, `prenom()`, `adresse()`, `fixe()`, `portable()` et `mail()` peuvent être définies directement dans la définition de la classe, ce qui donne finalement au fichier fiche.h l'aspect suivant ☐ :

```
1   #include "qstring.h"
2   class CFiche
3   {
4   public:
5     CFiche();
6     virtual ~CFiche();
7     CFiche(QString n, QString p, QString a, QString tf, QString tp, QString m);
8     bool operator ==(const CFiche &autre) const;
9     QString nom() const {return m_nom;}
10    QString prenom() const {return m_prenom;}
```

```
11     QString adresse() const {return m_adresse;}
12     QString fixe() const {return m_fixe;}
13     QString portable() const {return m_portable;}
14     QString mail() const {return m_mail;}
15  protected:
16     QString m_nom;
17     QString m_prenom;
18     QString m_adresse;
19     QString m_fixe;
20     QString m_portable;
21     QString m_mail;
22     bool compatibles(const QString & s1, const QString & s2) const;
   //déclarations d'amitié
23     friend QTextStream & operator << (QTextStream & flux, const CFiche & f);
24     friend QTextStream & operator >> (QTextStream & flux, CFiche & f);
25  }; //fin de la définition de la classe CFiche
26  //déclaration des fonctions globales "annexes"
27  QTextStream & operator << (QTextStream & flux, const CFiche & f);
28  QTextStream & operator >> (QTextStream & flux, CFiche & f);
```

La fonction `compatibles()` ne fait pas partie de l'interface de `CFiche` (son utilité n'est apparue qu'au cours de la définition des fonctions membre de cette classe) et rejoint donc les variables membre dans la section `protected:`

Remarquez que les lignes *23* et *24* ne constituent que des déclarations d'amitié, et qu'il est par ailleurs *(27-28)* nécessaire de déclarer les fonctions concernées pour que leur usage soit accepté dans tous les fichiers qui, comme TD11DialogImpl.cpp, comportent une directive

```
#include "fiche.h"
```

Il serait bien curieux que ces fonctions globales puissent être déclarées *dans* la classe...

Si vous avez placé en tête du fichier TD11DialogImpl.cpp toutes les directives `#include` exigées par l'usage que les fonctions définies dans ce fichier font de la librairie Qt, le programme doit maintenant pouvoir être compilé ☐ et exécuté ☐.