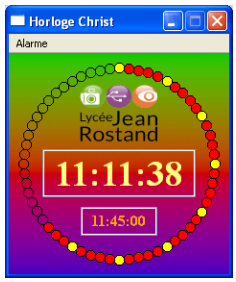
Durée: 4h **TP logiciel 12 : Développement graphique sous QT**

Horloge



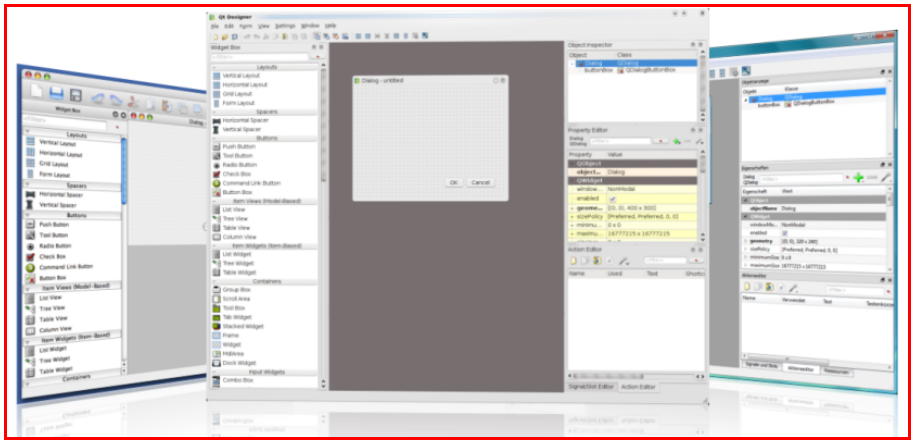
Objectifs:

- Utiliser Qt Creator avec le designer
- S'initier au développement des fenêtres pour windows.
- Mise en oeuvre de quelques objets widgets pour créer une application simple.

But: Développer l'affichage d'une horloge avec alarme.

Remarque: Commentez au maximum votre fichier pour une meilleure compréhension et servez-vous des aides (Qt et pages web) pour utiliser les nouvelles fonctions.

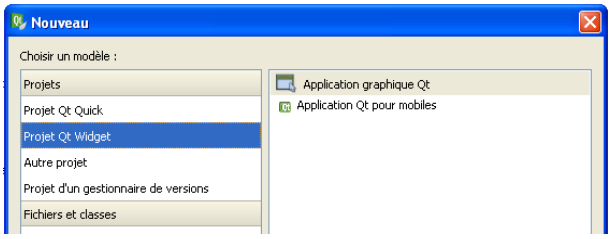
Présentation de l'environnement de développement Qt Designer :



Qt Designer est un outil de Qt pour la conception et la construction d'interfaces utilisateur graphiques (GUI) à partir de composants Qt. Vous pouvez composer et personnaliser vos widgets ou vos boîtes de dialogue dans un environnement what-you-see-is-what-you-get (WYSIWYG) de manière à les tester dans les différentes possibilités de styles et de résolution.

Les **widgets** et les **formes** créées avec Qt Designer sont intégrés de façon transparente avec le code programmé. Toutes les propriétés peuvent être modifiés dynamiquement dans le code.

Pour démarrer un projet, le plus simple est de créer une application graphique QT dans un Projet **QT Widgets**.

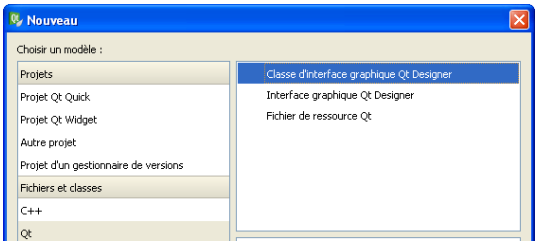


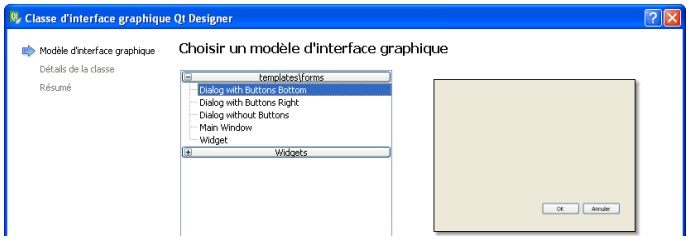
- Indiquer le chemin et le nom du projet, ainsi que le nom de la classe du modèle graphique qui correspond dans ce cas à une **QMainWindow** qui permet la gestion des menus et des barres d'outils.

Remarque :

Pour ajouter une fenêtre de Qt Designer dans un projet qui existe déjà, il faut aller dans le menu Fichier > Nouveau fichier ou projet puis sélectionnez Qt > Classe d'interface graphique Qt Designer

Vous devez choisir le type de fenêtre à créer.





Les 3 premiers choix correspondent à des **QDialog**.

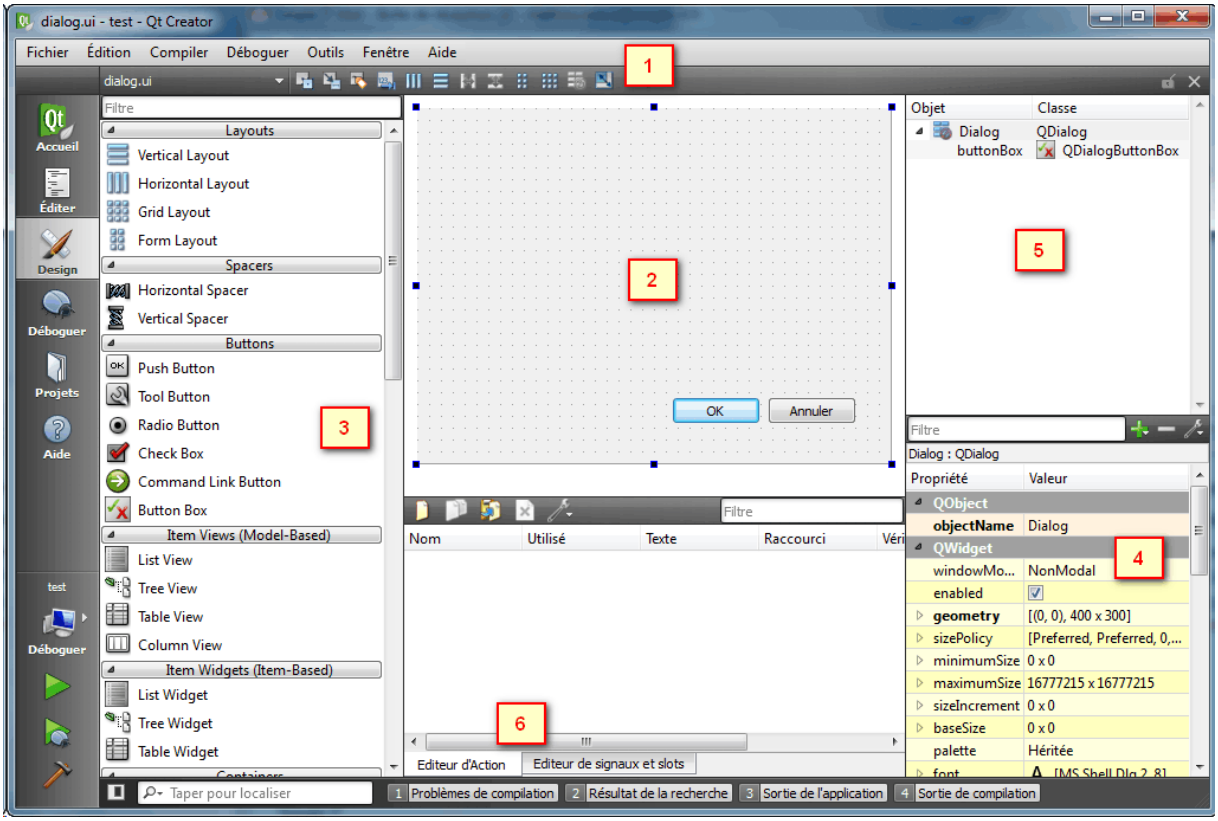
Le 4ième choix est un **QMainWindow** qui permet la gestion des menus et des barres d'outils.

Enfin, le dernier choix correspond à une simple fenêtre de type **QWidget**.

Cliquer sur “suivant” et indiquer le nom de la classe de votre nouvelle fenêtre. Trois fichiers seront créés :

- **dialog.ui** : c'est le fichier qui contiendra l'interface graphique (de type XML). C'est le fichier que l'on modifie avec l'éditeur Qt Designer ;
- **dialog.h** : permet de charger le fichier .ui dans votre projet C++ (fichier en-tête de classe) ;
- **dialog.cpp** : permet de charger le fichier .ui dans votre projet C++ (fichier code source de classe).

Analyse de la fenêtre de Qt Designer



1 - Sur la barre d'outils de Qt Designer, quatre boutons permettent de passer d'un mode d'édition à un autre. 

- **Éditer les widgets** : le mode par défaut, Il permet d'insérer des widgets sur la fenêtre et de modifier leurs propriétés.
- **Éditer signaux/slots** : permet de créer des connexions entre les signaux et les slots de vos widgets.
- **Éditer les copains** : permet d'associer des **QLabel** avec leurs champs respectifs. Lorsque vous faites un layout de type **QFormLayout**, ces associations sont automatiquement créées.
- **Éditer l'ordre des onglets** : permet de modifier l'ordre de tabulation entre les champs de la fenêtre, pour ceux qui naviguent au clavier et passent d'un champ à l'autre en appuyant sur la touche Tab.

2 - Au centre de Qt Designer, correspond la fenêtre à éditer. Pour le moment, celle-ci est vide.

- Si l'on crée une QMainWindow, on a, en plus, une barre de menus et une barre d'outils. Leur édition se fait à la souris.
- Si l'on crée une QDialog, on a des boutons « OK » et « Annuler » déjà disposés.

3 - Widget Box : Ce dock donne la possibilité de sélectionner un widget à placer sur la fenêtre. Ceux-ci sont organisés par groupes. Pour placer un de ces widgets sur la fenêtre, il suffit de faire un glisser-déplacer.

4 - Property Editor : Lorsqu'un widget est sélectionné sur la fenêtre principale, on peut éditer ses propriétés. Les widgets possèdent en général beaucoup de propriétés, celles-ci sont organisées en fonction de la classe dans laquelle elles ont été définies. On peut ainsi modifier toutes les propriétés dont un widget hérite, en plus des propriétés qui lui sont propres.

Si aucun widget n'est sélectionné, ce sont les propriétés de la fenêtre que l'on édite. On peut par exemple modifier son titre avec la propriété windowTitle, son icône avec windowIcon, etc.

5 - Object Inspector : affiche la liste des widgets placés sur la fenêtre, en fonction de leur relation de parenté, sous forme d'arbre.

6 - Éditeur de signaux/slots et éditeur d'action : ils sont séparés par des onglets. L'éditeur de signaux/slots est utile si on associe des signaux et des slots, les connexions du widget sélectionné apparaissant ici.

L'éditeur d'actions permet de créer des QAction. C'est donc utile lorsque l'on vous crée une QMainWindow avec des menus et une barre d'outils.

Placer des widgets sur la fenêtre:

Il y a deux manières différentes de placer des widgets sur la fenêtre:

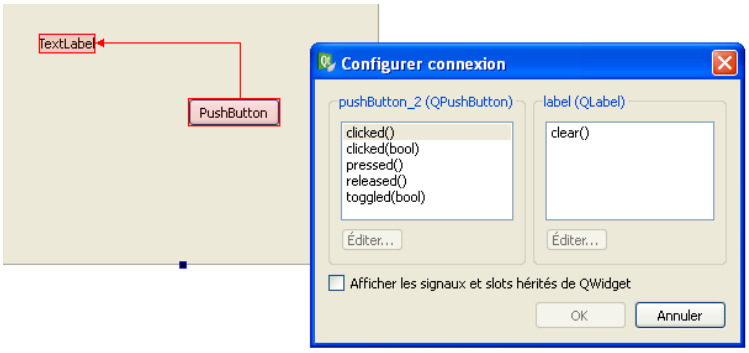
De manière absolue : vos widgets seront disposés au pixel près sur la fenêtre. C'est la méthode par défaut, la plus précise, mais la moins flexible aussi.

Avec des layouts (recommandé pour les fenêtres complexes) : On peut utiliser les layouts tel que verticaux, horizontaux, en grille, en formulaire… Grâce à cette technique, les widgets s'adapteront automatiquement à la taille de votre fenêtre.

Configurer les signaux et les slots 

Pour passer en mode « **Edit Signals/Slots** », il suffit de cliquer sur le second bouton de la barre d'outils.

Dans ce mode, on ne peut pas ajouter, modifier, supprimer, ni déplacer de widgets. Par contre, si l'on pointe sur les widgets de la fenêtre, vous doit les voir s'encadrer de rouge et l'on peut, de manière très intuitive, associer les widgets entre eux pour créer des connexions simples entre leurs signaux et slots.



Remarque:

Qt Designer est incapable de faire des connexions complexes ou des slots plus personnalisés, Il nous faudra développer dans le code source (en modifiant les fichiers **.h** et **.cpp** qui ont été créés en même temps que le **.ui**).

Utiliser la fenêtre dans votre application

Comme Qt Creator a créé un fichier .ui mais aussi des fichiers **.cpp** et **.h** de la classe, ces fichiers vont être appelés pour l'exécution de la fenêtre créée.

Si l'on crée une fenêtre Form1, la classe Form1 qui a été conçue automatiquement par Qt Creator (fichiers **Form1.h** et **Form1.cpp**) permettra de générer une nouvelle instance de cette classe.

Un fichier automatiquement généré par Qt Designer a été automatiquement inclus dans le .cpp : **#include "ui_form1.h"**.

Par ailleurs le constructeur charge l'interface définie dans ce fichier auto-généré grâce à **ui->setupUi(this);**. C'est cette ligne qui lance la construction de la fenêtre.

Utilisez la classe Form1 pour compléter ses fonctionnalités et la rendre intelligente.

Exemple, modifier la valeur d'un label, code: **ui->label1->setText("Bonjour");**
ou effectuer une connexion, code:

connect(ui->bouton1, SIGNAL(clicked()), this, SLOT(modifier_Text( )));

Ce code permet de faire en sorte que le slot **modifier_Text( )** de la fenêtre soit appelé à chaque fois que l'on clique sur le bouton. Bien sûr, il faut développer le code du slot.

Il ne reste plus qu'à adapter le main pour appeler la fenêtre comme une fenêtre classique:

```
#include <QtGui/QApplication>
#include "form1.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Form1 w;
    w.show();
    return a.exec();
}
```

Personnaliser le code et utiliser les Auto-Connect

Les fenêtres créées avec Qt Designer bénéficient du système « Auto-Connect » de Qt. C'est un système qui crée les connexions tout seul.

Il suffit de créer des slots en leur donnant un nom qui respecte une convention.

Exemple entre un widget bouton1 et son signal clicked( ).

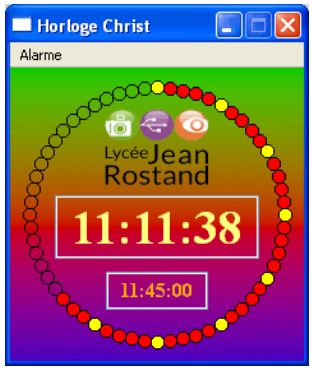
Si l'on crée un slot appelé **on_bouton1_clicked( )** dans votre fenêtre, ce slot sera automatiquement appelé lors d'un clic sur le bouton.

La convention à respecter est représentée par: **on_bouton1_clicked( )**

Nom du widget — **signal envoyé par le widget**

Travail demandé:

1) Créer un nouveau projet "Application graphique Qt" dans un répertoire **votre_nom_Tp_Horloge** et ayant comme Nom de fichier : **votre nom_Horloge**. Modifier le nom de classe **WindowsForm** par **Form1**.



2) A l'aide du designer, ajouter dans l'ordre
   - un premier **label** renommer en **labelFond**
   - un deuxième **label** renommer en **labelHorloge**
   - un troisième **label** renommer en **labelAlarme**
   - un quatrième **label** renommer en **labelLogo**

3) Supprimer la **barre de status** et la barre d'outils '**mainToolBar**' de l'objet **Form1**. Puis modifier la barre de menu en remplaçant l'onglet '**Tapez ici**' par '**Menu**' ajouter un onglet en dessous '**Réglage Alarme**', un séparateur et un dernier onglet '**Quitter**'

4) Créer, dans votre répertoire de fichiers sources Qt (**votre_nom_Tp_Horloge)** , un répertoire **Images** dans lequel vous copiez vos fichiers images utiles à votre projet.

Ajouter à votre projet un fichier ressource Qt (faire: Ajouter nouveau ... Qt fichier de ressource Qt puis choisir) donner un nom et l'emplacement (Horloge_Ressouces à stocker dans votre répertoire sources.

Modifier, sous Qt, votre fichier **Horloge_Ressouces.qrc**, pour ajouter les fichiers images nécessaires au projet.

5) Dans le designer de Qt, modifier le **labelFond** pour insérer votre image de fond en modifiant la propriété **pixmap** et en choisissant un fichier ressouce (Choisir ressouce ...). Valider la propriété **scaledContents** qui permet d'ajuster l'image à la taille de votre label.

Ajouter, dans le constructeur '**Form1**' du '**fichier form1.cpp**' le code :

```
this->setFixedSize(230,250); // Fixe la taille de la fenêtre principale
ui->labelFond->setGeometry(this->geometry());
// place et dimensionne le labelFond à la taille de l'objet centralWidget du Form1
```

6) Modifier les propriétés de l'objet **labelHorloge** tel que:

- **geometry** : x = 35, y = 100, Largeur = 160, hauteur = 50
- **font** : Times New Roman taille 28 gras
- **frameShape** : StyledPanel
- **alignment Horizontal** : AlignementCenterH
- **palette** : **WindowText** : choisir une couleur

Modifier les propriétés de l'objet **labelAlarme** tel que:

- **geometry** : x = 75, y = 160, Largeur = 80, hauteur = 30
- **font** : Times New Roman taille 12 gras
- **frameShape** : StyledPanel
- **alignment Horizontal** : AlignementCenterH
- **palette** : **WindowText** : choisir une couleur
- appeller, dans le constructeur de Form1, la méthode **hide( )** pour cacher l'objet.

Modifier les propriétés de l'objet **labelLogo** tel que:

- **geometry** : x = 70, y = 30, Largeur = 90, hauteur = 65
- **pixmax** : choisir un fichier resource *.pgn pour le logo (pour la transparence)
- **scaledContents** : validé

7) Accéder à la fenêtre **Design** de Qt et sélectionner dans l'éditeur d'action '**actionQuitter**' bouton droit souris et aller au slot **triggered()** pour coder:

```
this->close( ); // fermeture fenêtre programme
```

- Tester votre programme et corriger les éventuelles erreurs.

8) Ajouter, dans le fichier **form1.h**, un pointeur **timer1** sur la classe **QTimer** et instancier l'objet dans le constructeur de **Form1** puis ajouter le code qui permet d'accéder à la méthode **AfficheHeure ( )** toutes les 500 ms :

```
connect(timer1,SIGNAL(timeout()),this,SLOT(AfficheHeure()));
timer1->start(500);
```

Remarque : N'oublier pas de déclarer la fonction en **private slots: void AfficheHeure( )**dans le fichier **form1.h** et d'inclure la classe **qtimer.h**.

9) Développer la méthode **Form1::AfficheHeure( )** qui met à jour l'objet **labelHorloge**

```
QString hms ; // objet hms sur la classe QString
hms = QTime::currentTime( ).toString( ); // récupération heure courante dans hms
ui->labelHorloge->setText(hms); // mise à jour du label lableHorloge
```

Remarque : N'oublier pas d'inclure la classe **QTime**.
- Tester votre programme et corriger les éventuelles erreurs.

10) On souhaite effectuer l'affichage des secondes de façon graphique dans des cercles colorés. Pour cela, il faut spécialiser un objet **widget** qui dessine un cercle dans une méthode **paintEvent( )** et permet de le colorer en fonction de nos besoins. Il faut donc une nouvelle classe dérivant d'un **QWidget**.

- Ajouter à votre projet une nouvelle classe C++ "**MonCercle**" dont la **classe Parent** est un **QWidget** et déclarer un attribut **couleur** de **type entier public**.

- Ajouter une méthode **public slots: void paintEvent(QPaintEvent * e)** qui permettra de dessiner votre cercle.

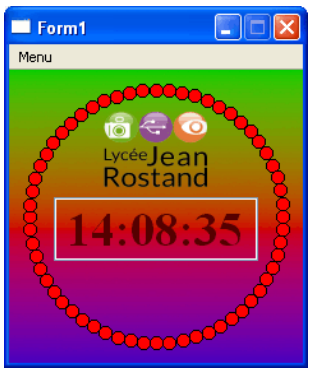
- Initialiser à '1' l'attribut **couleur** dans le constructeur de la classe,

- Dans la méthode **void MonCercle:: paintEvent(QPaintEvent * e)** , déclarer un objet **cercle** de type **QPainter** avec l'argument d'entrée '**this**' et utiliser les méthodes **setPen( )**, **setBrush( )** et **drawEllipse ( )** de la classe **Qpainter** pour le personnaliser de façon à obtenir un cercle de diamètre 10 coloré différemment en fonction de la variable couleur.

si **couleur** = '1' , le cercle est rouge
si **couleur** = '2' , le cercle est jaune
si **couleur** = '3' , le cercle est noir
si **couleur** = '4' , le cercle est blanc

Remarque : N'oubliez pas d'inclure la classe **QPainter** et servez-vous de l'aide de Qt pour l'utilisation des méthodes associées à l'objet (voir notamment l'aide de **QPen**).

11) Déclarer dans **.h** de **Form1**, un tableau dynamique de 60 objets de type **MonCercle** et instancier les 60 objets avec l'argument d'entrée '**this**' dans le constructeur de Form1. Utiliser la propriété **setGeometry(...)** pour les placer comme dans le dessin ci-contre.



Remarque: Pour placer les 60 objets dans un cercle, il suffit de calculer les coordonnées x_{nb} et y_{nb} pour chaque objet ($0 \leq nb < 60$) correspondant à l'équation d'un cercle tel que:

$$x_{nb} = rayon * \cos (2 * PI * nb / 60)$$
$$y_{nb} = rayon * \sin (2 * PI * nb / 60)$$

- Tester votre programme et corriger les éventuelles erreurs.

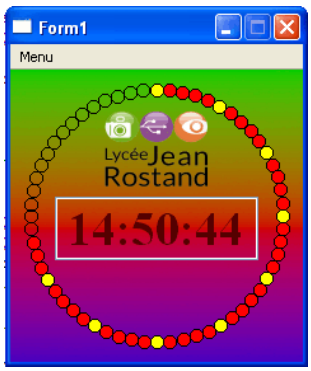
12) Pour animer la coloration des cercles formant l'horloge, il suffit de modifier la couleur des 60 objets en fonction du temps écouler .

Déclarer, dans la méthode **void Form1::AfficheHeure( )** , une variable **nbs** et affecter la valeur correspondante à la seconde en cours. (Voir l'aide de **Qtime**)

Puis, mettre à jour l'attribut **couleur** des 60 objets de type **MonCercle** en fonction de la valeur **nbs** et effectuer un rafraîchissement de l'écran avec la commande.

this- update( ) ;

vous devez obtenir :



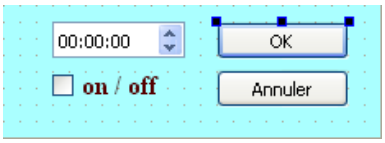
13) **Gestion affichage de l'alarme**.

a) Création de la fenêtre QDialog : FormAlarme.

- Ajouter à votre projet une nouvelle classe d'interface graphique QT Designer de type '**Dialog with Buttons Rigth**' ayant comme nom de classe '**FormAlarme**', et ajuster :
  - **geometry** : Largeur = 230, hauteur = 80
  - **windowsTitle** : Réglage Alarme

- Placer, à l'aide du designer, les objets **QtimeEdit** et **CheckBox** et régler les attributs pour obtenir la fenêtre :

nom de l'objet **QtimeEdit**: **alarmEdit**
nom de l'objet **QCheckBox** : **alarmCheck**



- Créer, dans le fichier **.h** de **FormAlarme**, un attribut **alarm_actif** de **type booléen** initialisé, dans le constructeur, à **false** .

- Cliquer sur l'objet **alarmCheck,** puis **bouton droit de la souris** pour sélectionner **aller au slot...** et choisir le slot **stateChanged(int)** qui permet d'exécuter un code en cas de changement de l'état du **checBox**. Coder la méthode pour mettre à jour la variable **alarm_actif** (**true ou false**) en fonction de l'état du **checkBox.**

- Créer, dans le fichier **.h** de **FormAlarme**, un objet **alarme** de type **QTime** .

- Cliquer sur l'objet **alarmEdit,** puis **bouton droit de la souris** pour sélectionner **aller au slot...** et choisir le slot **timeChanged(QTime)** qui permet d'exécuter un code en cas de changement de l'état du **QTimeEdit.** Coder la méthode qui met à jour l'objet **alarme** en fonction de l'état **hh.mm.ss** de l'bjet **QTimeEdit**.

Utiliser la méthode **setHMS( )** de l'objet alarme.

b) Exécution de la fenêtre : FormAlarme suite à un clique sur l'onglet **Réglage Alarme**

Il faut maintenant activer et rendre la fenêtre **Form_Alarm** visible dans le cas ou l'on clique sur l'onglet **Réglage Alarme** de la fenêtre principale de l'application.

- Ajouter, dans le fichier **.h** de **Form1**, un pointeur **reglage** de type **FormAlarm**. et instancier l'objet dans le constructeur de **Form1** avec l'argument d'entrée **'this'**.

- Accéder à la fenêtre **Design Form1.ui** de Qt et sélectionner dans l'éditeur d'action '**actionR_glage_alarme**' bouton droit souris et aller au slot **triggered()** pour coder la méthode :

reglage->exec ( ) // exécute la fenêtre de dialog et attend qu'elle soit
// fermée pour continuer l'exécution du code.

- Suivant le paramètre (true ou false) de l'attribut **alarm_actif** de **FormAlarm**, mettre à jour le texte de l'objet **labelAlarme** de **Form1** (texte issu de l'objet **alarme** de **FormAlarm)** et rendre visible ou non l'objet **labelAlarme** .

Utiliser les méthodes: **setText(...)** , **show( )** , **hide( )**.

c) <u>Activation visuelle de l'alarme.</u>

- Modifier le code dans la fonction **void Form1::AfficheHeure( )** pour faire flasher les 60 objets de type **MonCercle** en noir et blanc pendant 30 secondes si l'alarme correspond à l'heure courante.