Programmez avec le langage C++

50 heures Difficile

Mis à jour le 25/03/2019

TP : zNavigo, le navigateur web des Zéros !

Depuis le temps que vous pratiquez Qt, vous avez acquis sans vraiment le savoir les capacités de base pour réaliser des programmes complexes. Le but d'un TP comme celui-ci, c'est de vous montrer justement que vous êtes capables de mener à bien des projets qui auraient pu vous sembler complètement fous il y a quelque temps.

Vous ne rêvez pas : le but de ce TP sera de... réaliser un navigateur web ! Et vous allez y arriver, c'est à votre portée !

Nous allons commencer par découvrir la notion de moteur web, pour bien comprendre comment fonctionnent les autres navigateurs. Puis, nous mettrons en place le plan du développement de notre programme afin de nous assurer que nous partons dans la bonne direction et que nous n'oublions rien.

Les navigateurs et les moteurs web

Comme toujours, il faut d'abord prendre le temps de réfléchir à son programme avant de foncer le coder tête baissée. C'est ce qu'on appelle la phase de *conception*.

Je sais, je me répète à chaque fois mais c'est vraiment parce que c'est très important. Si je vous dis « faites-moi un navigateur web » et que vous créez de suite un nouveau projet en vous demandant ce que vous allez bien pouvoir mettre dans le `main` ... c'est l'échec assuré.

Pour moi, la conception est l'étape la plus difficile du projet. Plus difficile même que le codage. En effet, si vous concevez bien votre programme, si vous réfléchissez bien à la façon dont il doit fonctionner, vous aurez simplifié à l'avance votre projet et vous n'aurez pas à écrire inutilement des lignes de code difficiles.

Dans un premier temps, je vais vous expliquer comment fonctionne un navigateur web. Un peu de culture générale à ce sujet vous permettra de mieux comprendre ce que vous avez à faire (et ce que vous n'avez pas à faire).
Je vous donnerai ensuite quelques conseils pour organiser votre code : quelles classes créer, par quoi commencer, etc.

Les principaux navigateurs

Commençons par le commencement : vous savez ce qu'est un navigateur web ?
Bon, je ne me moque pas de vous mais il vaut mieux être sûr de ne perdre personne.

Un navigateur web est un programme qui permet de consulter des sites web.
Parmi les plus connus d'entre eux, citons Internet Explorer, Mozilla Firefox, Google Chrome ou encore Safari. Mais il y en a aussi beaucoup d'autres, certes moins utilisés, comme Opera, Konqueror, Epiphany, Maxthon, Lynx...

Je vous rassure, il n'est pas nécessaire de tous les connaître pour pouvoir prétendre en créer un.
Par contre, ce qu'il faut que vous sachiez, c'est que chacun de ces navigateurs est constitué de ce qu'on appelle un **moteur web**. Qu'est-ce que c'est que cette bête-là ?

Le moteur web

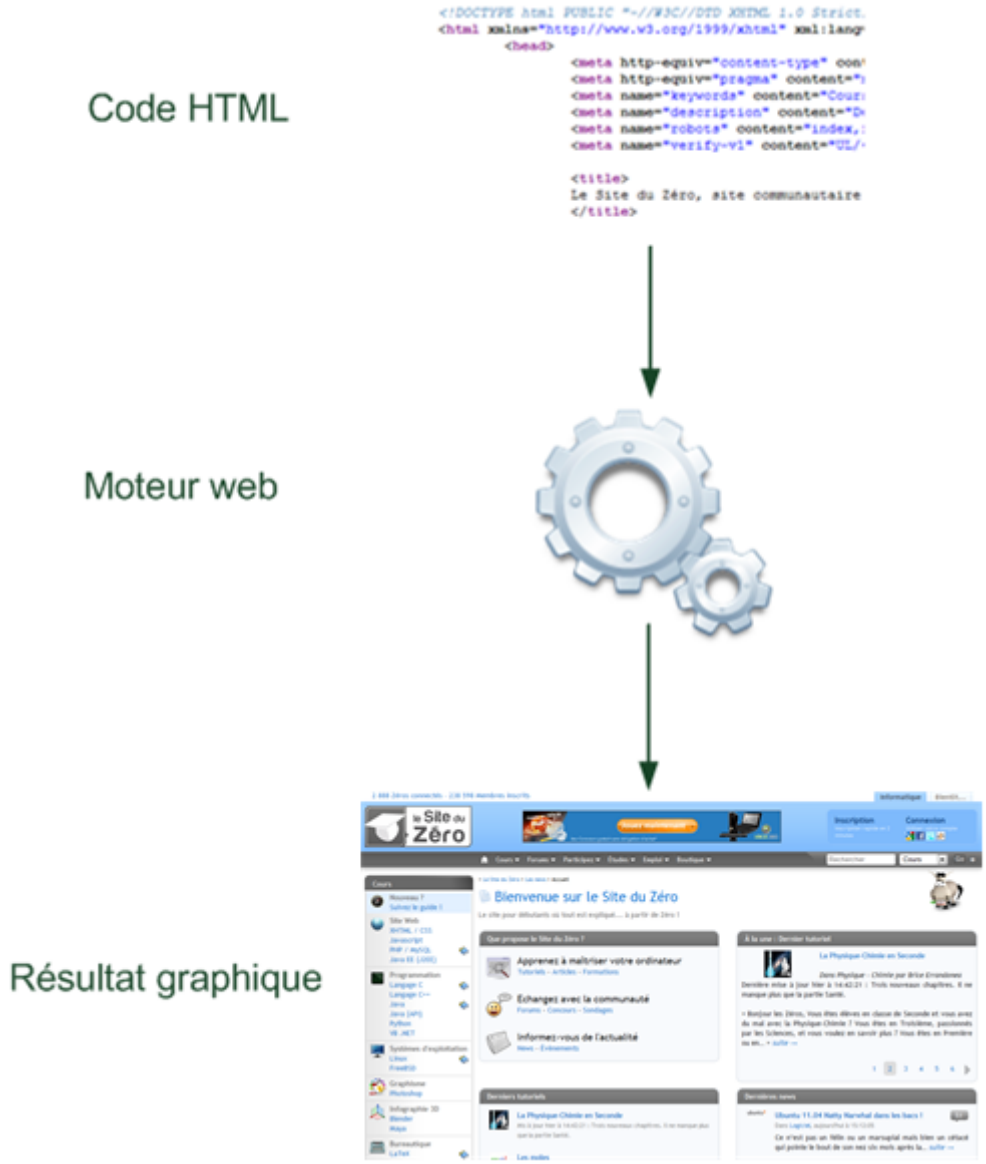
Tous les sites web sont écrits en langage HTML (ou XHTML). Voici un exemple de code HTML permettant de créer une page très simple :

cpp

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
  strict.dtd">
2 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
3    <head>
4       <title>Bienvenue sur mon site !</title>
5       <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
6    </head>
7    <body>
8    </body>
9 </html>
```

C'est bien joli tout ce code, mais cela ne ressemble pas au résultat visuel qu'on a l'habitude de voir lorsqu'on navigue sur le Web.

L'objectif est justement de transformer ce code en un résultat visuel : le site web. C'est le rôle du moteur web.
La figure suivante présente son fonctionnement, résumé dans un schéma très simple.



Le rôle du moteur web

Cela n'a l'air de rien mais c'est un travail difficile : réaliser un moteur web est très délicat. C'est généralement le fruit des efforts de nombreux programmeurs experts (et encore, ils avouent parfois avoir du mal). Certains moteurs sont meilleurs que d'autres mais aucun n'est parfait ni complet. Comme le Web est en perpétuelle évolution, il est peu probable qu'un moteur parfait sorte un jour.

Quand on programme un navigateur, on utilise généralement le moteur web sous forme de bibliothèque.
Le moteur web n'est donc pas un programme mais il est utilisé par des programmes.

Ce sera peut-être plus clair avec un schéma. Regardons comment est constitué Firefox par exemple :

Schéma du navigateur web

On voit que le navigateur (en vert) « contient » le moteur web (en jaune au centre).

> La partie en vert est habituellement appelée le « chrome », pour désigner l'interface.

> Mais c'est nul ! Alors le navigateur web c'est juste les 2-3 boutons en haut et c'est tout ?

Oh non ! Loin de là.
Le navigateur ne se contente pas de gérer les boutons « Page Précédente », « Page Suivante », « Actualiser », etc. C'est aussi lui qui gère les marque-pages (favoris), le système d'onglets, les options d'affichage, la barre de recherche, etc.

Tout cela représente déjà un énorme travail ! En fait, les développeurs de Firefox ne sont pas les mêmes que ceux qui développent son moteur web. Il y a des équipes séparées, tellement chacun de ces éléments représente du travail.

Un grand nombre de navigateurs ne s'occupent d'ailleurs pas du moteur web. Ils en utilisent un « tout prêt ».

De nombreux navigateurs sont basés sur le même moteur web. L'un des plus célèbres s'appelle WebKit : il est utilisé par Safari, Google Chrome et Konqueror, notamment.

Créer un moteur web n'est pas de votre niveau (ni du mien). Comme de nombreux navigateurs, nous en utiliserons un pré-existant.

Lequel ? Eh bien il se trouve que Qt vous propose depuis d'utiliser le moteur WebKit dans vos programmes. C'est donc ce moteur-là que nous allons utiliser pour créer notre navigateur.

Configurez son projet pour utiliser WebKit

WebKit est un des nombreux modules de Qt. Il ne fait pas partie du module « GUI », dédié à la création de fenêtres, il s'agit d'un module à part.

Pour pouvoir l'utiliser, il faudra modifier le fichier `.pro` du projet pour que Qt sache qu'il a besoin de charger WebKit.
Voici un exemple de fichier `.pro` qui indique que le projet utilise WebKit :

text

```
TEMPLATE = app
QT += widgets webkitwidgets
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += FenPrincipale.h
SOURCES += FenPrincipale.cpp main.cpp
```

D'autre part, vous devrez rajouter l'`include` suivant dans les fichiers de votre code source faisant appel à WebKit :

cpp

```
#include <QtWebKitWidgets>
```

Enfin, il faudra certainement joindre de nouveaux DLL à votre programme pour qu'il fonctionne.

Ouf, tout est prêt.

Organisation du projet

Objectif

Avant d'aller plus loin, il est conseillé d'avoir en tête le programme que l'on cherche à créer. Reportez-vous à la figure présentée en introduction.

Parmi les fonctionnalités de ce super navigateur, affectueusement nommé « zNavigo », on compte :

- accéder aux pages précédentes et suivantes ;
- arrêter le chargement de la page ;

- actualiser la page ;
- revenir à la page d'accueil ;
- saisir une adresse ;
- naviguer par onglets ;
- afficher le pourcentage de chargement dans la barre d'état.

Le menu `Fichier` permet d'ouvrir et de fermer un onglet, ainsi que de quitter le programme.
Le menu `Navigation` reprend le contenu de la barre d'outils (ce qui est très facile à faire grâce aux `QAction`, je vous le rappelle).
Le menu `?` (aide) propose d'afficher les fenêtres `À propos...` et ... `À propos de Qt...` qui donnent des informations respectivement sur notre programme et sur Qt.

Cela n'a l'air de rien comme cela, mais cela représente déjà un sacré boulot !
Si vous avez du mal dans un premier temps, vous pouvez vous épargner la gestion des onglets... mais moi j'ai trouvé que c'était un peu trop simple sans les onglets alors j'ai choisi de vous faire jouer avec, histoire de corser le tout. ;-)

Les fichiers du projet

J'ai l'habitude de faire une classe par fenêtre. Comme notre projet ne sera constitué (au moins dans un premier temps) que d'une seule fenêtre, nous aurons donc les fichiers suivants :

- ...main.cpp ;
- ...FenPrincipale.h ;
- ...FenPrincipale.cpp.

Si vous voulez utiliser les mêmes icônes que moi, utilisez le code web qui suit pour les télécharger (toutes ces icônes sont sous licence LGPL et proviennent du site `everaldo.com`).

Télécharger les icônes

Utiliser `QWebView` pour afficher une page web

`QWebView` est le principal nouveau widget que vous aurez besoin d'utiliser dans ce chapitre. Il permet d'afficher une page web. C'est lui le moteur web.

Vous ne savez pas vous en servir mais vous savez maintenant lire la documentation. Vous allez voir, ce n'est pas bien difficile !

Regardez en particulier les signaux et slots proposés par le `QWebView`. Il y a tout ce qu'il faut savoir pour, par exemple, connaître le pourcentage de chargement de la page pour le répercuter sur la barre de progression de la barre d'état (signal `loadProgress(int)`).

Comme l'indique la documentation, pour créer le widget et charger une page, c'est très simple :

cpp

```
1 QWebView *pageWeb = new QWebView;
2 pageWeb->load(QUrl("http://www.siteduzero.com/"));
```

Voilà c'est tout ce que je vous expliquerai sur `QWebView`, pour le reste lisez la documentation. :-)

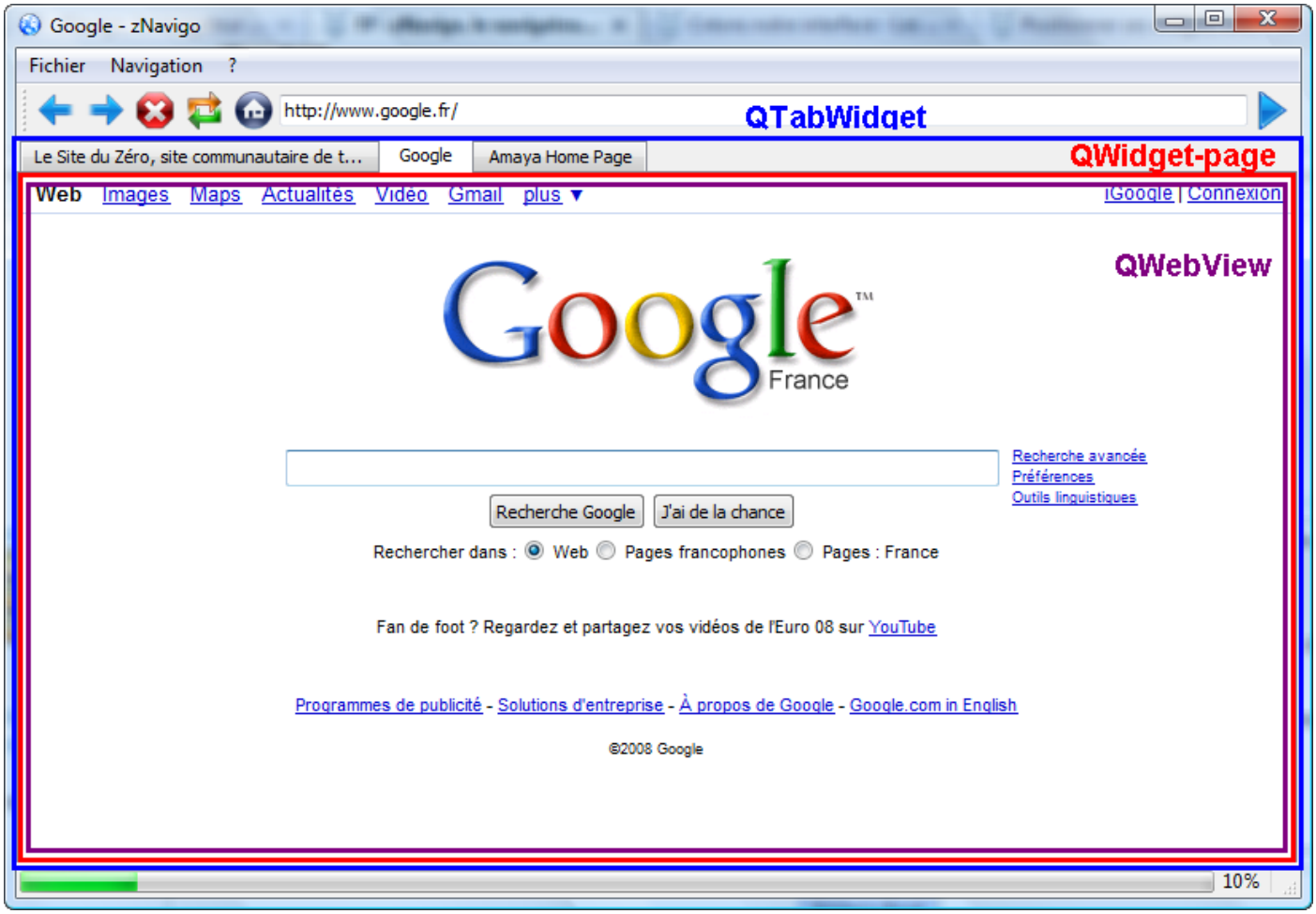
La navigation par onglets

Le problème de `QWebView`, c'est qu'il ne permet d'afficher qu'une seule page web à la fois. Il ne gère pas la navigation par onglets. Il va falloir implémenter le système d'onglets nous-mêmes.

Vous n'avez jamais entendu parler de `QTabWidget` ? Si si, souvenez-vous, nous l'avons découvert dans un des chapitres précédents. Ce widget-conteneur est capable d'accueillir n'importe quel widget... comme un `QWebView` !
En combinant un `QTabWidget` et des `QWebView` (un par onglet), vous pourrez reconstituer un véritable navigateur par onglets !

Une petite astuce toutefois, qui pourra vous être bien utile : savoir retrouver un widget contenu dans un widget parent.
Comme vous le savez, le `QTabWidget` utilise des sous-widgets pour gérer chacune des pages. Ces sous-widgets sont généralement des `QWidget` génériques (invisibles), qui servent à contenir d'autres widgets.

Dans notre cas : `QTabWidget` *contient* des `QWidget` (pages d'onglets) qui eux-mêmes *contiennent* chacun un `QWebView` (la page web). Voyez la figure suivante.



Structure de zNavigo

La méthode `findChild` (définie dans `QObject`) permet de retrouver le widget enfant contenu dans le widget parent.

Par exemple, si je connais le `QWidget` `pageOnglet`, je peux retrouver le `QWebView` qu'il contient comme ceci :

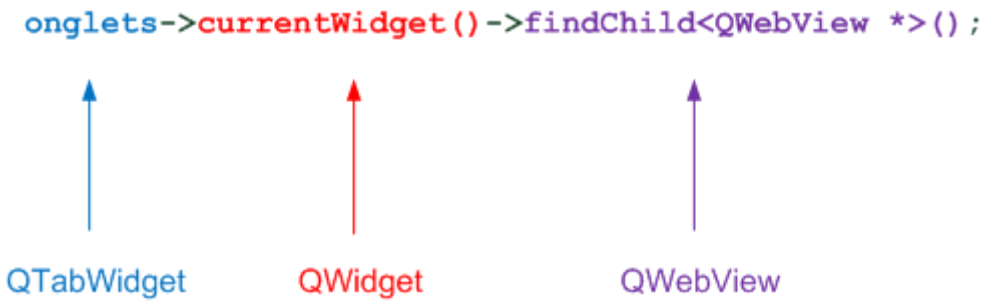
cpp

```cpp
QWebView *pageWeb = pageOnglet->findChild<QWebView *>();
```

Mieux encore, je vous donne la méthode toute faite qui permet de retrouver le `QWebView` actuellement visualisé par l'utilisateur (figure suivante) :

cpp

```cpp
QWebView *FenPrincipale::pageActuelle()
{
    return onglets->currentWidget()->findChild<QWebView *>();
}
```



... onglets correspond au `QTabWidget`.
Sa méthode `currentWidget()` permet d'obtenir un pointeur vers le `QWidget` qui sert de page pour la page actuellement affichée.
On demande ensuite à retrouver le `QWebView` que le `QWidget` contient à l'aide de la méthode `findChild()`. Cette méthode utilise les templates C++ avec `<QWebView *>` (nous découvrirons en profondeur leur fonctionnement dans un prochain chapitre). Cela permet de faire en sorte que la méthode renvoie bien un `QWebView *` (sinon elle n'aurait pas su quoi renvoyer).

J'admets, c'est un petit peu compliqué, mais au moins cela pourra vous aider.

Let's go !

Voilà, vous savez déjà tout ce qu'il faut pour vous en sortir.

Notez que ce TP fait la part belle à la `QMainWindow`, n'hésitez donc pas à relire ce chapitre dans un premier temps pour bien vous remémorer son fonctionnement.
Pour ma part, j'ai choisi de coder la fenêtre « à la main » (pas de Qt Designer donc) car celle-ci est un peu complexe.

Comme il y a beaucoup d'initialisations à faire dans le constructeur, je vous conseille de les placer dans des méthodes que vous appellerez depuis le constructeur pour améliorer la lisibilité globale :

cpp
```cpp
FenPrincipale::FenPrincipale()
{
    creerActions();
    creerMenus();
    creerBarresOutils();

    /* Autres initialisations */

}
```

Bon courage !

Génération de la fenêtre principale

Je ne vous présente pas ici le fichier ...main.cpp, il est simple et classique. Intéressons-nous aux fichiers de la fenêtre.

...FenPrincipale.h (première version)

Dans un premier temps, je ne crée que le squelette de la classe et ses premières méthodes, j'en rajouterai d'autres au fur et à mesure si besoin est.

cpp

```cpp
#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtWidgets>
#include <QtWebKitWidgets>

class FenPrincipale : public QMainWindow
{
    Q_OBJECT

    public:
        FenPrincipale();

    private:
        void creerActions();
        void creerMenus();
        void creerBarresOutils();
        void creerBarreEtat();

    private slots:

    private:
        QTabWidget *onglets;

        QAction *actionNouvelOnglet;
        QAction *actionFermerOnglet;
        QAction *actionQuitter;
        QAction *actionAPropos;
        QAction *actionAProposQt;
        QAction *actionPrecedente;
        QAction *actionSuivante;
        QAction *actionStop;
        QAction *actionActualiser;
        QAction *actionAccueil;
        QAction *actionGo;

        QLineEdit *champAdresse;
        QProgressBar *progression;
};

#endif
```

La classe hérite de `QMainWindow` comme prévu. J'ai inclus ... `<QtWidgets>` et ... `<QtWebKitWidgets>` pour pouvoir utiliser le module Widgets et le module WebKit (moteur web).

Mon idée c'est, comme je vous l'avais dit, de couper le constructeur en plusieurs sous-méthodes qui s'occupent chacune de créer une section différente de la `QMainWindow` : actions, menus, barre d'outils, barre d'état...

J'ai prévu une section pour les slots personnalisés mais je n'ai encore rien mis, je verrai au fur et à mesure.

Enfin, j'ai préparé les principaux attributs de la classe. En fin de compte, à part de nombreuses `QAction`, il n'y en a pas beaucoup. Je n'ai même pas eu besoin de mettre des objets de type `QWebView` : ceux-ci seront créés à la volée au cours du programme et on pourra les retrouver grâce à la méthode `pageActuelle()` que je vous ai donnée un peu plus tôt.

Voyons voir l'implémentation du constructeur et de ses sous-méthodes qui génèrent le contenu de la fenêtre.

Construction de la fenêtre

Direction ...FenPrincipale.cpp, on commence par le constructeur :

cpp

```cpp
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    // Génération des widgets de la fenêtre principale
    creerActions();
    creerMenus();
    creerBarresOutils();
    creerBarreEtat();

    // Génération des onglets et chargement de la page d'accueil
    onglets = new QTabWidget;
    onglets->addTab(creerOngletPageWeb(tr("http://www.siteduzero.com")), tr("(Nouvelle page)"));
    connect(onglets, SIGNAL(currentChanged(int)), this, SLOT(changementOnglet(int)));
    setCentralWidget(onglets);

    // Définition de quelques propriétés de la fenêtre
    setMinimumSize(500, 350);
    setWindowIcon(QIcon("images/znavigo.png"));
    setWindowTitle(tr("zNavigo"));
}
```

Les méthodes `creerActions()`, `creerMenus()`, `creerBarresOutils()` et `creerBarreEtat()` ne seront pas présentées dans le livre car elles sont longues et répétitives (mais vous pourrez les retrouver en entier en téléchargeant le code web présenté à la fin de ce chapitre). Elles consistent simplement à mettre en place les éléments de la fenêtre principale comme nous avons appris à le faire.

Par contre, ce qui est intéressant ensuite dans le constructeur, c'est que l'on crée le `QTabWidget` et on lui ajoute un premier onglet. Pour la création d'un onglet, on va faire appel à une méthode « maison » `creerOngletPageWeb()` qui se charge de créer le QWidget-page de l'onglet, ainsi que de créer un `QWebView` et de lui faire charger la page web envoyée en paramètre (`http://www.siteduzero.com` sera donc la page d'accueil par défaut, mais je vous promets que j'ai choisi ce site au hasard ;-)).

Vous noterez que l'on utilise la fonction de `tr()` partout où on écrit du texte susceptible d'être affiché. Cela permet de faciliter la traduction ultérieure du programme dans d'autres langues, si on

le souhaite. Son utilisation ne coûte rien et ne complexifie pas vraiment le programme, donc pourquoi s'en priver ?

On connecte enfin et surtout le signal `currentChanged()` du `QTabWidget` à un slot personnalisé `changementOnglet()` que l'on va devoir écrire. Ce slot sera appelé à chaque fois que l'utilisateur change d'onglet, pour, par exemple, mettre à jour l'URL dans la barre d'adresse ainsi que le titre de la page affiché en haut de la fenêtre.

Voyons maintenant quelques méthodes qui s'occupent de gérer les onglets...

Méthodes de gestion des onglets

En fait, il n'y a que 2 méthodes dans cette catégorie :

- `creerOngletPageWeb()` : je vous en ai parlé dans le constructeur, elle se charge de créer un QWidget-page ainsi qu'un `QWebView` à l'intérieur, et de renvoyer ce QWidget-page à l'appelant pour qu'il puisse créer le nouvel onglet.
- `pageActuelle()` : une méthode bien pratique que je vous ai donnée un peu plus tôt, qui permet à tout moment d'obtenir un pointeur vers le `QWebView` de l'onglet actuellement sélectionné.

Voici ces méthodes :

cpp

```cpp
QWidget *FenPrincipale::creerOngletPageWeb(QString url)
{
    QWidget *pageOnglet = new QWidget;
    QWebView *pageWeb = new QWebView;

    QVBoxLayout *layout = new QVBoxLayout;
    layout->setContentsMargins(0,0,0,0);
    layout->addWidget(pageWeb);
    pageOnglet->setLayout(layout);

    if (url.isEmpty())
    {
        pageWeb->load(QUrl(tr("html/page_blanche.html")));
    }
    else
    {
        if (url.left(7) != "http://")
        {
            url = "http://" + url;
        }
        pageWeb->load(QUrl(url));
    }

    // Gestion des signaux envoyés par la page web
    connect(pageWeb, SIGNAL(titleChanged(QString)), this, SLOT(changementTitre(QString)));
    connect(pageWeb, SIGNAL(urlChanged(QUrl)), this, SLOT(changementUrl(QUrl)));
    connect(pageWeb, SIGNAL(loadStarted()), this, SLOT(chargementDebut()));
    connect(pageWeb, SIGNAL(loadProgress(int)), this, SLOT(chargementEnCours(int)));
```

```
29      connect(pageWeb, SIGNAL(loadFinished(bool)), this, SLOT(chargementTermine(bool)));
30
31      return pageOnglet;
32  }
33
34  QWebView *FenPrincipale::pageActuelle()
35  {
36      return onglets->currentWidget()->findChild<QWebView *>();
37  }
```

Je ne commente pas `pageActuelle()`, je l'ai déjà fait auparavant.

Pour ce qui est de `creerOngletPageWeb()`, elle crée comme prévu un `QWidget` et elle place un nouveau `QWebView` à l'intérieur. La page web charge l'URL indiquée en paramètre et rajoute le préfixe ... http:// si celui-ci a été oublié.
Si aucune URL n'a été spécifiée, on charge une page blanche. J'ai pour l'occasion créé un fichier HTML vide, placé dans un sous-dossier ... html du programme.

On connecte plusieurs signaux intéressants envoyés par le `QWebView` qui, à mon avis, parlent d'eux-mêmes : « Le titre a changé », « L'URL a changé », « Début du chargement », « Chargement en cours », « Chargement terminé ».

Bref, rien de sorcier, mais cela fait encore tout plein de slots personnalisés à écrire ! ;-)

Les slots personnalisés

Bon, il y a de quoi faire. Il faut compléter `FenPrincipale.h` pour lui ajouter tous les slots que l'on a l'intention d'écrire : `precedente()`, `suivante()`, etc. Je vous laisse le soin de le faire, ce n'est pas difficile. Ce qui est intéressant, c'est de voir leur implémentation dans le fichier `.cpp`.

Implémentation des slots

Slots appelés par les actions de la barre d'outils

Commençons par les actions de la barre d'outils :

cpp

```
1  void FenPrincipale::precedente()
2  {
3      pageActuelle()->back();
4  }
5
6  void FenPrincipale::suivante()
7  {
8      pageActuelle()->forward();
9  }
10
11 void FenPrincipale::accueil()
12 {
13     pageActuelle()->load(QUrl(tr("http://www.siteduzero.com")));
14 }
15
```

```
16 void FenPrincipale::stop()
17 {
18     pageActuelle()->stop();
19 }
20
21 void FenPrincipale::actualiser()
22 {
23     pageActuelle()->reload();
24 }
```

On utilise la (très) pratique fonction `pageActuelle()` pour obtenir un pointeur vers le `QWebView` que l'utilisateur est en train de regarder (histoire d'affecter la page web de l'onglet en cours et non pas les autres).

Toutes ces méthodes, comme `back()` et `forward()`, sont des slots. On les appelle ici comme si c'étaient de simples méthodes (je vous rappelle que les slots sont en réalité des méthodes qui peuvent être connectées à des signaux).

> Pourquoi ne pas avoir connecté directement les signaux envoyés par les `QAction` aux slots du `QWebView` ?

On aurait pu, s'il n'y avait pas eu d'onglets. Le problème justement ici, c'est qu'on gère plusieurs onglets différents.

Par exemple, on ne pouvait pas connecter lors de sa création la `QAction` « actualiser » au `QWebView` … parce que le `QWebView` à actualiser dépend de l'onglet actuellement sélectionné !
Voilà donc pourquoi on passe par un petit slot maison qui va d'abord chercher à savoir quel est le `QWebView` que l'on est en train de visualiser pour être sûr qu'on recharge la bonne page.

Slots appelés par d'autres actions des menus

Voici les slots appelés par les actions des menus suivants :

- Nouvel onglet ;
- Fermer l'onglet ;
- À propos…

cpp

```
1 void FenPrincipale::aPropos()
2 {
3     QMessageBox::information(this, tr("A propos..."), tr("zNavigo est un projet réalisé pour
  illustrer les tutoriels C++ du <a href=\"http://www.siteduzero.com\">Site du Zéro</a>.<br />Les
  images de ce programme ont été créées par <a href=\"http://www.everaldo.com\">Everaldo
  Coelho</a>"));
4 }
5
6 void FenPrincipale::nouvelOnglet()
7 {
8     int indexNouvelOnglet = onglets->addTab(creerOngletPageWeb(), tr("(Nouvelle page)"));
```

```
    onglets->setCurrentIndex(indexNouvelOnglet);

    champAdresse->setText("");
    champAdresse->setFocus(Qt::OtherFocusReason);
}

void FenPrincipale::fermerOnglet()
{
    // On ne doit pas fermer le dernier onglet
    if (onglets->count() > 1)
    {
        onglets->removeTab(onglets->currentIndex());
    }
    else
    {
        QMessageBox::critical(this, tr("Erreur"), tr("Il faut au moins un onglet !"));
    }
}
```

Le slot `aPropos()` se contente d'afficher une boîte de dialogue.

`nouvelOnglet()` rajoute un nouvel onglet à l'aide de la méthode `addTab()` du `QTabWidget`, comme on l'avait fait dans le constructeur. Pour que le nouvel onglet s'affiche immédiatement, on force son affichage avec `setCurrentIndex()` qui se sert de l'index (numéro) de l'onglet que l'on vient de créer. On vide la barre d'adresse et on lui donne le focus, c'est-à-dire que le curseur est directement placé dedans pour que l'utilisateur puisse écrire une URL.

> L'action « Nouvel onglet » a comme raccourci `Ctrl + T`, ce qui permet d'ouvrir un onglet à tout moment à l'aide du raccourci clavier correspondant.
> Vous pouvez aussi ajouter un bouton dans la barre d'outils pour ouvrir un nouvel onglet ou, encore mieux, rajouter un mini-bouton dans un des coins du `QTabWidget`. Regardez du côté de la méthode `setCornerWidget()`.

`fermerOnglet()` supprime l'onglet actuellement sélectionné. Il vérifie au préalable que l'on n'est pas en train d'essayer de supprimer le dernier onglet, auquel cas le `QTabWidget` n'aurait plus lieu d'exister (un système à onglets sans onglets, cela fait désordre).

Slots de chargement d'une page et de changement d'onglet

Ces slots sont appelés respectivement lorsqu'on demande à charger une page (on appuie sur la touche `Entrée` après avoir écrit une URL ou on clique sur le bouton tout à droite de la barre d'outils) et lorsqu'on change d'onglet.

cpp

```
void FenPrincipale::chargerPage()
{
    QString url = champAdresse->text();

    // On rajoute le "http://" s'il n'est pas déjà dans l'adresse
```

```
    if (url.left(7) != "http://")
    {
        url = "http://" + url;
        champAdresse->setText(url);
    }

    pageActuelle()->load(QUrl(url));
}

void FenPrincipale::changementOnglet(int index)
{
    changementTitre(pageActuelle()->title());
    changementUrl(pageActuelle()->url());
}
```

On vérifie au préalable que l'utilisateur a mis le préfixe http:// et, si ce n'est pas le cas on le rajoute (sinon l'adresse n'est pas valide).

Lorsque l'utilisateur change d'onglet, on met à jour deux éléments sur la fenêtre : le titre de la page, affiché tout en haut de la fenêtre et sur un onglet, et l'URL inscrite dans la barre d'adresse. `changementTitre()` et `changementUrl()` sont des slots personnalisés, que l'on se permet d'appeler comme n'importe quelle méthode. Ces slots sont aussi automatiquement appelés lorsque le `QWebView` envoie les signaux correspondants.

Voyons voir comment implémenter ces slots...

Slots appelés lorsqu'un signal est envoyé par le `QWebView`

Lorsque le `QWebView` s'active, il va envoyer des signaux. Ceux-ci sont connectés à des slots personnalisés de notre fenêtre. Les voici :

cpp

```
void FenPrincipale::changementTitre(const QString & titreComplet)
{
    QString titreCourt = titreComplet;

    // On tronque le titre pour éviter des onglets trop larges
    if (titreComplet.size() > 40)
    {
        titreCourt = titreComplet.left(40) + "...";
    }

    setWindowTitle(titreCourt + " - " + tr("zNavigo"));
    onglets->setTabText(onglets->currentIndex(), titreCourt);
}

void FenPrincipale::changementUrl(const QUrl & url)
{
    if (url.toString() != tr("html/page_blanche.html"))
    {
        champAdresse->setText(url.toString());
    }
}

```

```
23 void FenPrincipale::chargementDebut()
24 {
25     progression->setVisible(true);
26 }
27
28 void FenPrincipale::chargementEnCours(int pourcentage)
29 {
30     progression->setValue(pourcentage);
31 }
32
33 void FenPrincipale::chargementTermine(bool ok)
34 {
35     progression->setVisible(false);
36     statusBar()->showMessage(tr("Prêt"), 2000);
37 }
```

Ces slots ne sont pas très complexes. Ils mettent à jour la fenêtre (par exemple la barre de progression en bas) lorsqu'il y a lieu.

Certains sont très utiles, comme `changementUrl()`. En effet, lorsque l'utilisateur clique sur un lien dans la page, l'URL change et il faut par conséquent mettre à jour le champ d'adresse.

Conclusion et améliorations possibles

Téléchargez le code source et l'exécutable

Je vous propose de télécharger le code source ainsi que l'exécutable Windows du projet.

Télécharger le programme

Pensez à ajouter les DLL nécessaires dans le même dossier que l'exécutable, si vous voulez que celui-ci fonctionne.

Améliorations possibles

Améliorer le navigateur, c'est possible ?
Certainement ! Il fonctionne mais il est encore loin d'être parfait, et j'ai des tonnes d'idées pour l'améliorer. Bon, ces idées sont repompées des navigateurs qui existent déjà mais rien ne vous empêche d'en inventer de nouvelles, super-révolutionnaires bien sûr !

- **Afficher l'historique dans un menu** : il existe une classe `QWebHistory` qui permet de récupérer l'historique de toutes les pages visitées *via* un `QWebView`. Renseignez-vous ensuite sur la doc de `QWebHistory` pour essayer de trouver comment récupérer la liste des pages visitées.
- **Recherche dans la page** : rajoutez la possibilité de faire une recherche dans le texte de la page. Indice : `QWebView` dispose d'une méthode `findText()` !
- **Fenêtre d'options** : vous pourriez créer une nouvelle fenêtre d'options qui permet de définir la taille de police par défaut, l'URL de la page d'accueil, etc.

Pour modifier la taille de la police par défaut, regardez du côté de `QWebSettings` .Pour enregistrer les options, vous pouvez passer par la classe `QFile` pour écrire dans un fichier. Mais j'ai mieux : utilisez la classe `QSettings` qui est spécialement conçue pour enregistrer des options. En général, les options sont enregistrées dans un fichier (... `.ini` , ... `.conf` ...), mais on peut aussi enregistrer les options dans la base de registre sous Windows.

- **Gestion des marque-pages (favoris)** : voilà une fonctionnalité très répandue sur la plupart des navigateurs. L'utilisateur aime bien pouvoir enregistrer les adresses de ses sites web préférés.

Là encore, pour l'enregistrement, je vous recommande chaudement de passer par un `QSettings` .

Que pensez-vous de ce cours ?

INDIQUER QUE CE CHAPITRE N'EST PAS TERMINÉ

MODÉLISEZ SES FENÊTRES AVEC QT DESIGNER

DÉCOUVREZ L''ARCHITECTURE MVC AVEC LES WIDGETS COMPLEXES

Les professeurs

Mathieu Nebra

Entrepreneur à plein temps, auteur à plein temps et co-fondateur d'OpenClassrooms :o)

Matthieu Schaller

Chercheur en astrophysique et cosmologie. Spécialiste en simulations numériques de galaxies sur superordinateurs.

Découvrez aussi ce cours en...

Livre PDF