

Cours wxWidgets 1ère partie

Introduction

Vous vous apprêtez à suivre un tutoriel sur l'utilisation de la bibliothèque wxWidgets (<http://www.wxwidgets.org>).

Il convient donc avant tout d'en faire une brève présentation.

wxWidgets (au départ, connue sous le nom wxWindows) est une bibliothèque de développement multi-plateforme.

Elle a à l'origine été créée pour permettre le développement d'applications en utilisant le même code sous Windows et Unix, et a ensuite été enrichie pour être compatible avec Linux, Mac, Windows CE, et bien d'autres systèmes d'exploitation.

Elle a l'avantage d'offrir une réduction des temps de programmation nécessaires pour qu'une application puisse être compilée et tourner sur tous ces systèmes d'exploitation.

Il est possible de l'utiliser en passant, bien sûr, par le C++, mais également Python, Perl, C#, ...

De plus, elle contient un grand nombre de classes simplifiant l'utilisation de nombreux contrôles.

Pour ce qui est de l'interface graphique, wxWidgets utilise, pour chaque plateforme, les API natives. Ce qui fait qu'une application aura, sous Windows, le look Windows, et sous Linux, le look Linux, etc...

Pour plus d'informations sur ce point, je vous invite à vous rendre sur le site officiel, où l'on peut trouver un bon nombre de captures d'écran d'applications tournant sous plusieurs systèmes d'exploitation.

Pour ce tutoriel, nous allons utiliser l'IDE Code::Blocks, qui d'ailleurs est basé sur wxWidgets, et qui de ce fait a l'avantage d'être disponible pour les principaux systèmes d'exploitations (Windows, Linux, Mac).

Il vous faut donc l'installer, ainsi que les fichiers de développement de wxWidgets.

Si le besoin s'en fait sentir, j'étofferai ce tutoriel afin d'expliquer la marche à suivre pour ces installations, mais ce n'en n'est pas le but premier, alors, nous verrons cela plus tard.

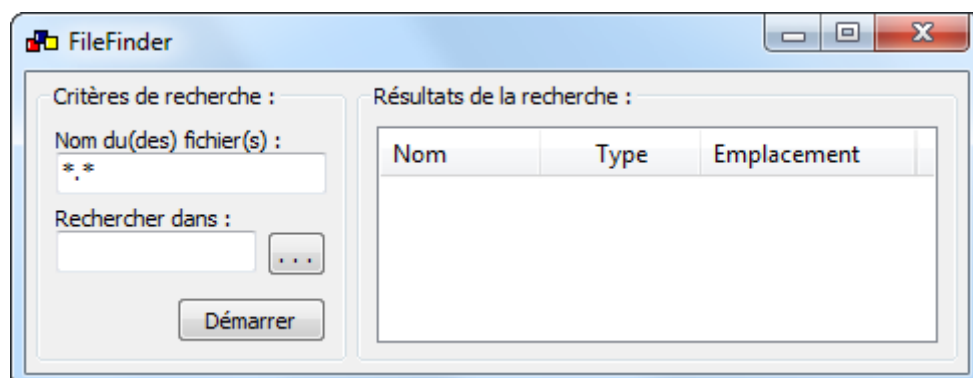
Qu'allons-nous faire au cours de cette première partie ?

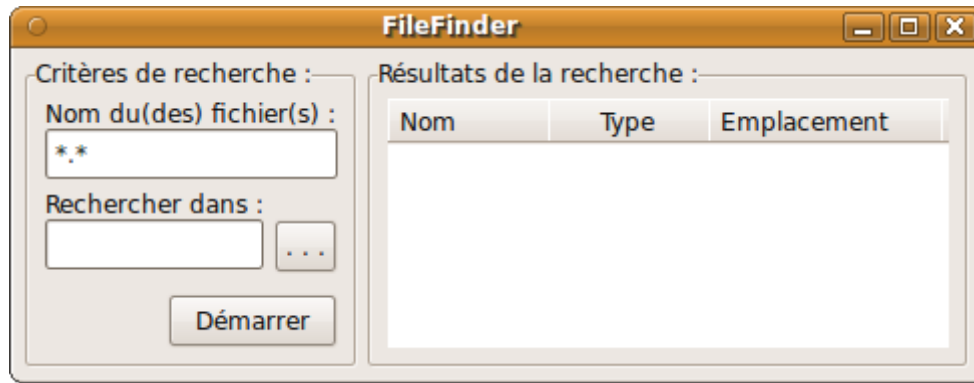
Afin de ne pas avoir un cours uniquement basé sur de la théorie (ce qui, à mon avis, serait sans doute plus rébarbatif qu'instructif) je vous propose d'utiliser comme fil conducteur la réalisation d'un petit utilitaire permettant de faire une recherche de fichiers.

Nous allons donc commencer par faire le minimum requis pour ce genre d'application : une simple recherche de fichiers depuis un répertoire donné, en utilisant des « jockers » (*.*, par exemple).

Puis, quand cette base sera fonctionnelle, nous étofferons un peu afin de lui ajouter quelques petites fonctionnalités (recherche d'un mot à l'intérieur des fichiers, options supplémentaires de recherche, ...). A ce sujet, si vous avez des suggestions d'améliorations, n'hésitez pas à m'en faire part.

Voici une capture, sous Windows et sous Ubuntu, de notre application telle qu'elle sera quand nous aurons terminé cette première partie:





Mise en place des éléments de base

C'est parti pour la réalisation de notre première application !

Tout d'abord, il faut que nous donnions un nom à notre utilitaire. Je vous propose « *FileFinder* » (je suis bien conscient que ce n'est pas très original, mais je préfère faire dans la simplicité, et vous pourrez de toute façon lui donner le nom que vous voudrez).

Création du projet

Démarrez Code::Blocks, créez un nouveau projet wxWidgets, donnez-lui le nom ci-dessus.

Pour les autres options du projet, voici ce qu'il faut mettre :

- Preferred GUI Builder : **None** : nous allons coder l'interface « à la main... ».
- Application Type : **Frame Based** : notre fenêtre principale sera dérivée de la classe **wxFrame**.
- wxWidget's Location : C'est vous qui voyez, en fonction de votre installation.
- Nous n'aurons pas besoin d'une configuration debug, mais vous pouvez en créer une quand même, si ça vous chante...
- Créez un projet vide.
- Pour les autres options, je vous laisse le choix (pour ma part, je choisis d'utiliser les DLL wxWidgets Unicode, mais pas en monolithique, sans les « En-têtes pré-compilées »).

Nous sommes maintenant en possession d'un joli projet entièrement vide.

Nous allons donc créer les deux classes principales de notre application :

- La classe **FileFinderApp**, dérivée de **wxApp**. Chaque application wxWidgets « classique » doit posséder une classe dérivée de **wxApp**, et contenant au minimum une redéfinition de la méthode **OnInit()**. Cette méthode sera appelée au lancement de l'application, et, en fonction de la valeur booléenne de retour, l'application s'arrêtera (valeur de retour = **false**) ou continuera (valeur de retour = **true**).
- La classe **MainFrame** dérivée de **wxFrame** et qui sera la fenêtre principale de l'application.

Nous allons dans un premier temps nous occuper de la fenêtre.

La classe de fenêtre

Tout d'abord, créez un nouveau fichier vide (menu *File* → *New* → *Empty file*), répondez Oui quand Code::Blocks vous demande si vous voulez ajouter ce fichier au projet courant, et enregistrez-le en le nommant *mainframe.h*.

Répétez la même opération pour un deuxième fichier que vous nommerez *mainframe.cpp*.

Ces deux fichiers doivent maintenant être ouverts dans l'éditeur de Code::Blocks. Revenez sur le premier fichier (*mainframe.h*) et entrez le code suivant (n'oubliez pas que le copier/coller est votre ami):

```
-----  
| #ifndef MAINFRAME_H_INCLUDED  
| #define MAINFRAME_H_INCLUDED  
|  
| // On ajoute les headers "basiques" de wxWidgets  
| #include <wx/wx.h>  
|  
|-----
```

```

| // Définition de la classe "MainFrame"
| class MainFrame : public wxFrame
| {
|     public:
|         // Le constructeur
|         MainFrame();
|         // Le destructeur
|         ~MainFrame();
| };
|
| #endif // MAINFRAME_H_INCLUDED

```

Il n'y a rien de particulier à dire sur ce fichier. Passons maintenant au contenu du fichier *mainframe.cpp* :

```

| // On récupère la définition de la classe MainFrame
| #include "mainframe.h"
|
| // Le constructeur
| MainFrame::MainFrame() : wxFrame(NULL, wxID_ANY, _T("FileFinder"))
| {
|     // Nous ajouterons ici la création des contrôles que contiendra notre fenêtre
| }
|
| // Le Destructeur
| MainFrame::~MainFrame()
| {
|     // Nous ajouterons ici les éventuelles libérations de mémoire nécessaires
|     // avant la fermeture de la fenêtre
| }

```

Ici non plus, rien de particulier, si ce n'est l'appel au constructeur de **wxFrame**.

Il contient 3 paramètres :

- **NULL** : correspond au pointeur vers l'objet parent (comme il s'agit de la fenêtre principale de notre application, elle n'en a pas, nous lui passons donc une valeur nulle).
- **wxID_ANY** : l'identifiant de notre fenêtre : dans notre cas, on laisse wxWidgets choisir la valeur. Nous verrons plus tard que ce n'est pas toujours le cas.
- **_T("FileFinder")** : il s'agit du texte qui apparaîtra dans la barre de titre de notre fenêtre. La macro **wxT("...")** permet de ne pas avoir à tenir compte du fait que notre application sera compilée en Ansi ou en Unicode.

La classe d'application

Nous allons maintenant nous occuper de la classe **FileFinderApp**, dont je vous ai parlé un peu plus haut.

Créez d'abord les deux fichiers (vides) qui vont recevoir le code de cette classe : *filefinderapp.h* et *filefinderapp.cpp*.

Pour le fichier *filefinderapp.h*, le code est :

```

| #ifndef FILEFINDERAPP_H_INCLUDED
| #define FILEFINDERAPP_H_INCLUDED
|
| // On ajoute les headers "basiques" de wxWidgets
| #include <wx/wx.h>
|
| // Définition de la classe "FileFinderApp"
| // Nous n'aurons pas besoin du constructeur et du destructeur
| // Mais seulement de la méthode "OnInit()"
| class FileFinderApp : public wxApp
| {
|     public:
|         virtual bool OnInit();
| };
|
| DECLARE_APP(FileFinderApp);

```

```
| #endif // FILEFINDERAPP_H_INCLUDED |
```

Vous noterez l'ajout de la macro **DECLARE_APP(FileFinderApp)** qui permet à wxWidgets de créer une fonction **wxGetApp()**. Cette fonction nous permettra plus tard d'obtenir une référence vers notre classe d'application.

Elle n'est pas obligatoire, mais peut être utile dans bien des cas.

Ensuite, le contenu du fichier *filefinderapp.cpp* :

```
| // On récupère la définition de la classe FileFinderApp |
| #include "filefinderapp.h" |
| // On récupère la définition de la classe MainFrame |
| #include "mainframe.h" |
| |
| // La méthode "OnInit()" |
| bool FileFinderApp::OnInit() |
| { |
|     // On crée un objet "MainFrame" |
|     MainFrame *frame=new MainFrame(); |
|     // On l'affiche |
|     frame->Show(); |
|     // On indique que l'application peut continuer |
|     return true; |
| } |
| IMPLEMENT_APP(FileFinderApp); |
```

Encore une fois, je pense que le code est suffisamment commenté.

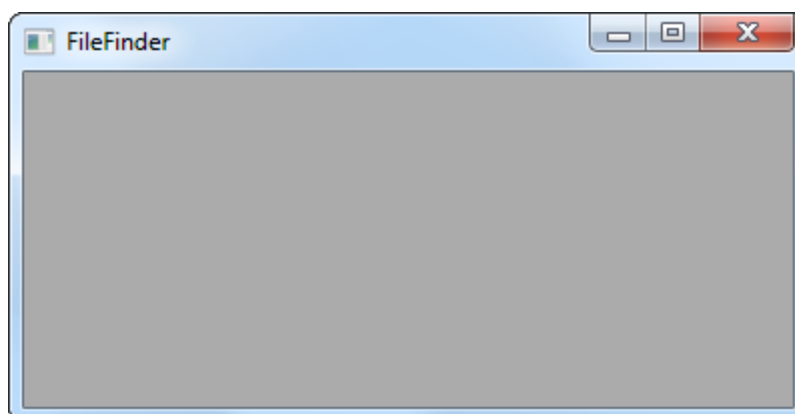
Il ne reste qu'une petite chose à expliquer : la macro **IMPLEMENT_APP(FileFinderApp)**.

Vous aurez peut-être remarqué qu'à aucun moment on ne crée d'élément de type **FileFinderApp**.

C'est en fait le framework wxWidgets qui va s'en charger par l'intermédiaire de cette macro qui sert à lui indiquer que cette classe est la classe principale de l'application, et que c'est en fonction de la valeur de retour de sa méthode **OnInit()** qu'il faudra ou non continuer à faire tourner l'exécutable.

Voilà, nous avons la base de notre projet. Nous allons maintenant pouvoir compiler tout ça, et l'exécuter.

Normalement, si vous n'avez pas fait d'erreur lors de la recopie du code source, vous devriez obtenir une fenêtre comme celle-ci :



J'en conviens, elle n'est pas très belle, mais c'est déjà un bon début.

Ajout d'une icône à la fenêtre

Nous allons maintenant lui affecter une icône.

Comme certains d'entre vous sont sous Linux ou Mac, nous n'allons pas utiliser un fichier ".ico" (qui ne pourrait pas être utilisable directement sous ces O.S. car il s'agit d'un format propre à Windows), mais une image X-Pixmap qui a l'avantage de fonctionner sous tous les O.S. (<http://fr.wikipedia.org/wiki/XPM>).

Vous trouverez celle que nous allons utiliser dans le sous-dossier *art* des sources wxWidgets) : le fichier se nomme

wxwidgets16x16.xpm. Faites-en une copie dans le répertoire du projet.

Pour utiliser un fichier XPM tel que celui-ci, il suffit de l'inclure dans le code source avec la directive **#include**.

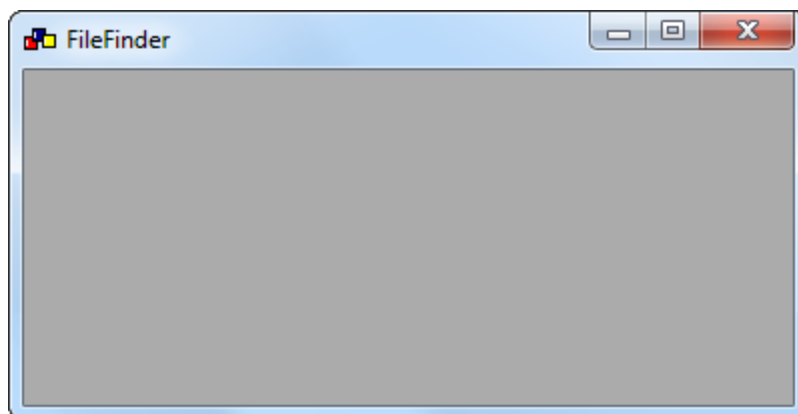
Ensuite, dans le constructeur de la fenêtre, nous allons affecter cette icône à la fenêtre avec la méthode **SetIcon(.....)**. Notre fichier *mainframe.cpp* devient donc :

```
// On récupère la définition de la classe MainFrame
#include "mainframe.h"
// On ajoute le fichier XPM de l'icône
#include "wxwidgets16x16.xpm"

// Le constructeur
MainFrame::MainFrame() : wxFrame(NULL, wxID_ANY, _T("FileFinder"))
{
    // On affecte l'icône à la fenêtre
    SetIcon(wxwidgets16x16_xpm);
    // Nous ajouterons ici la création des contrôles que contiendra notre fenêtre
}

// Le Destructeur
MainFrame::~MainFrame()
{
    // Nous ajouterons ici les éventuelles libérations de mémoire nécessaires
    // avant la fermeture de la fenêtre
}
```

Voilà, c'est tout : vous pouvez compiler et exécuter, vous devriez obtenir la même fenêtre que précédemment, mais avec une icône personnalisée :



Mise en place des contrôles

Maintenant que nous possédons une fenêtre, nous allons pouvoir y ajouter des contrôles, afin de la rendre un peu plus utile.

Pour cela, nous allons avoir besoin d'une partie un peu obscure de wxWidgets pour qui n'y est pas habitué : les **wxSizer**, et leurs classes dérivées.

Mais tout d'abord, qu'est-ce qu'un sizer ?

Il s'agit en fait d'un contrôle "virtuel" servant à la mise en place et au redimensionnement automatique des contrôles qui lui sont affectés.

Par exemple, sans utiliser les sizers, si je veux placer sur ma fenêtre un label (un **wxStaticText**), et sur la même ligne, une zone de texte (un **wxTextCtrl**) qui occupera le reste de la fenêtre en largeur, avec un espace de 5 pixels entre les deux, il va falloir que je procède de la façon suivante :

- Récupérer la largeur de la fenêtre (de sa zone cliente, en fait).
- Créer le contrôle **wxStaticText** à la position x0/y0.
- Mesurer sa largeur et y ajouter 5 pixels pour obtenir la position du contrôle suivant.
- Créer le contrôle **wxTextCtrl** à la bonne position, et de la bonne taille.
- Intercepter l'événement "**OnSize**" de la fenêtre.

- Refaire les calculs et repositionner les contrôles à chaque redimensionnement de la fenêtre

Avec les sizers, tout cela se fait automatiquement :

- On crée le sizer en lui indiquant dans quel sens il faudra qu'il intervienne sur les contrôles (horizontal ou vertical). Ainsi, pour l'exemple ci-dessus, il s'agirait d'un sizer horizontal.
- Créer le **wxStaticText** (sans se soucier de sa taille et de sa position), et l'ajouter au sizer en lui indiquant que ce contrôle a une taille fixe, et qu'il ne doit pas être redimensionné.
- Créer le **wxTextCtrl** et l'ajouter au sizer en indiquant qu'il doit être redimensionné pour occuper le reste de la largeur de la fenêtre.
- Il est également possible de donner une valeur aux « marges » à laisser autour de chaque contrôle
- Il suffit d'affecter le sizer à la fenêtre, et c'est lui qui va se charger de gérer le redimensionnement.
- Un sizer est également capable de calculer la taille minimale que devra avoir une fenêtre en fonction des contrôles qu'elle contient.

Et si l'on a plusieurs lignes à gérer, il suffit de créer plusieurs sizers horizontaux, et de les ajouter à un sizer vertical. De plus, il existe plusieurs sortes de sizers qui permettent d'obtenir des résultats différents :

- **wxBoxSizer** : c'est le sizer classique (en fait, la classe **wxSizer** n'est pas utilisable telle-quelle, elle sert uniquement de base aux autres). Ce sizer crée en quelque sorte une « boîte » virtuelle dans laquelle seront placés et organisés les contrôles en fonction des paramètres passés lors de leur ajout. C'est le premier sizer que nous utiliserons, pour mettre en place correctement les parties gauche (critères de recherche) et droite (résultats de la recherche) de notre interface.
- **wxStaticBoxSizer** : ce sizer a un comportement identique au précédent, mais il est associé à un contrôle « cadre » (**wxStaticBox**). C'est ce sizer que nous utiliserons pour organiser les contrôles à l'intérieur des deux zones (gauche et droite) de notre interface.
- **wxGridSizer** : comme son nom l'indique, il permet d'organiser les contrôles sur une grille virtuelle, dont les « cases » auront toutes la même taille.
- **wxFlexGridSizer** : comme le précédent, il va organiser les contrôles sur une grille, mais en offrant la possibilité d'avoir des lignes et des colonnes de taille différentes.
- **wxGridBagSizer** : il s'agit d'un **wxFlexGridSizer** offrant la possibilité de placer un contrôle « à cheval » sur plusieurs cellules, ou à une position donnée (par exemple, dans la 3ème cellule de la 2ème ligne, même si les cellules précédentes ne sont pas encore affectées).

Comme je viens de vous le dire, nous n'utiliserons que les deux premiers types de sizer. Nous allons maintenant en voir le fonctionnement.

Le wxBoxSizer

Lors de sa construction, on lui donne en paramètre la direction dans laquelle ce sizer doit agir. Deux constantes **wxWidgets** sont disponibles pour cela : **wxHORIZONTAL** et **wxVERTICAL** (je suppose que vous avez deviné leur signification).

Donc, si on veut créer un **wxBoxSizer** horizontal, il suffit d'écrire :

```
wxBoxSizer *hsizer=new wxBoxSizer(wxHORIZONTAL);
```

Pour lui ajouter un contrôle, il suffit d'utiliser sa méthode **Add(...)**.

Nous lui passerons en premier paramètre un pointeur vers le contrôle (ou vers un autre sizer) à gérer.

Le deuxième paramètre concerne la proportion que le sizer doit donner au contrôle : une valeur « 0 » indique que le contrôle ne doit pas être redimensionné, sinon, il suffit de mettre une valeur entière. Par exemple, si on ajoute un bouton avec la proportion « 1 », et une zone de texte avec la proportion « 4 », la zone de texte aura 4 fois la taille du bouton.

Viens ensuite un paramètre « flag », qui est directement associé au 4ème paramètre : la valeur des marges. Ce 3ème paramètre est une combinaison des valeurs suivantes :

- **wxALL** : la valeur des marges est à appliquer sur les 4 bords du contrôle.
- **wxLEFT**, **wxRIGHT**, **wxTOP**, **wxBOTTOM** : pour n'appliquer la valeur des marges que sur un ou plusieurs bords. Il faut en effet garder à l'esprit que si l'on ajoute un premier contrôle avec une marge de 5 pixels tout autour, puis un deuxième contrôle avec la même marge, il y aura un espace de 10 pixels entre les deux. Dans ce cas, il suffit de n'appliquer la marge que sur les bords "haut", "bas", "droite" du deuxième contrôle pour respecter les 5 pixels.

- **wxEXPAND** : le contrôle sera étendu dans la direction complémentaire à celle du sizer, pour en occuper tout

l'espace. Par exemple, avec un sizer horizontal, si l'on insère une zone de texte multi-lignes, puis un simple bouton, le bouton sera forcément moins « haut » que la zone de texte. Le flag **wxEXPAND** va étendre le bouton pour qu'il occupe toute la hauteur. Pour les autres valeurs que peut prendre ce paramètre, je vous laisse consulter la documentation officielle concernant les sizers : http://www.wxwidgets.org/manuals/stable/wx_wxsizer.html (en anglais).

Le wxStaticBoxSizer

Son fonctionnement est identique à celui du **wxBoxSizer** que l'on vient de voir, mais il permet d'afficher en plus une **wxStaticBox**, c'est à dire, un cadre avec une légende en haut à gauche.

Il suffit de lui ajouter un pointeur vers le contrôle parent de la **wxStaticBox**, et un **wxString** contenant le texte de légende à afficher, et le tour est joué :

```
wxStaticBoxSizer *hsizer=new wxStaticBoxSizer(wxHORIZONTAL, maFrame, _T("Texte à afficher :"));
```

Voilà pour ce qui est des explications de base sur les sizers. Nous allons maintenant voir comment nous allons les utiliser pour notre interface.

La mise en place des contrôles

Il va d'abord falloir que l'on gère les deux zones « gauche » et « droite ». Nous allons donc créer un premier **wxBoxSizer** horizontal.

Pour chacune de ces zones, nous utiliserons un **wxStaticBoxSizer** :

- pour la partie de gauche, ce sizer va devoir gérer plusieurs lignes de contrôles; il sera donc vertical. Ensuite, pour chaque ligne, il faudra ajouter un **wxBoxSizer** horizontal dès qu'il y aura plus d'un contrôle sur la ligne concernée.
- pour la partie de droite, le sizer n'ayant qu'un seul contrôle à gérer (la liste des résultats), il pourra être indifféremment horizontal ou vertical.

Nous allons maintenant dresser la liste des contrôles dont nous aurons besoin, afin de déterminer quelles variables nous allons devoir créer pour leur utilisation :

Commençons par la zone de gauche :

- Nous n'aurons pas besoin de modifier le texte de la **wxStaticBox** principale. Il est donc inutile de créer une variable pour cela.
- Il en est de même pour le label « Nom du (des) fichier(s) : » (un **wxStaticText**).
- Par contre, pour la zone de texte servant à recevoir le nom ou les jockers de recherche (un **wxTextCtrl**), nous allons avoir besoin d'un pointeur qui nous permettra d'en récupérer la valeur.
- Le label « Rechercher dans : » est logé à la même enseigne que le précédent : nous n'aurons pas besoin de le modifier.
- Nous aurons besoin d'un pointeur vers la zone de texte servant à la saisie du dossier de démarrage, pour en récupérer le contenu, ou pour le modifier.
- Nous aurons également besoin d'un « identifiant » pour le bouton « parcourir » situé à droite de cette dernière zone de texte, ainsi que d'un pointeur nous permettant de l'activer ou de le désactiver. Cet identifiant est en fait une valeur entière unique (que nous demanderons à **wxWidgets** de créer pour nous) et qui, comme son nom le laisse supposer, sert à identifier un contrôle, et à le connecter à la table des événements (nous verrons cela plus loin, rassurez-vous).
- Comme pour le bouton précédent, il nous faudra un identifiant et un pointeur pour le bouton « Démarrer ».

Pour la zone de droite :

- Il serait souhaitable de garder un pointeur vers la **wxStaticBox** principale, pour pouvoir afficher le nombre de résultats trouvés.
- Il nous faut également un pointeur pour le contrôle **wxListCtrl** servant à l'affichage des résultats.

Nous allons, dans un premier temps, ajouter les variables et les constantes (pour les identifiants des boutons) à notre classe **MainFrame** (fichier *mainframe.h*):

```
#ifndef MAINFRAME_H_INCLUDED
#define MAINFRAME_H_INCLUDED
```

```

|
| // On ajoute les headers "basiques" de wxWidgets
| #include <wx/wx.h>
| // On ajoute le header du contrôle wxListCtrl, qui n'est pas inclus
| // Dans les headers "basiques"
| #include <wx/listctrl.h>
|
| // Définition de la classe "MainFrame"
| class MainFrame : public wxFrame
| {
|     public:
|         // Le constructeur
|         MainFrame();
|         // Le destructeur
|         ~MainFrame();
|     private:
|         // Les pointeurs vers les wxTextCtrl pour le nom/les jockers et le dossier de démarrage
|         wxTextCtrl *m_txtName, *m_txtPath;
|         // Les pointeur vers les boutons "Parcourir" et "Démarrer"
|         wxButton *m_btnBrowse, *m_btnStart;
|         // Les identifiants pour ces même boutons
|         static const int ID_BTN_BRWSE, ID_BTN_START;
|         // Le pointeur vers la wxStaticBox de droite pour l'affichage du nombre de résultats trouvés
|         wxStaticBox *m_stbNumRes;
|         // Le pointeur vers le wxListCtrl pour l'affichage des résultats.
|         wxListCtrl *m_lstResults;
| };
|
| #endif // MAINFRAME_H_INCLUDED

```

Vous remarquerez, au passage, qu'il a fallu inclure un header spécifique pour le wxListCtrl, car il ne fait pas partie des contrôles de base de wxWidgets dont les en-têtes sont automatiquement incluses par le fichier *wx/wx.h*.

Nous allons passer à la création des contrôles. Tout se fait dans le constructeur de la fenêtre principale.

Il va également falloir que l'on ajoute la définition des deux constantes relatives aux boutons (les identifiants).

Comme je vous l'ai dit précédemment, ces constantes doivent avoir une valeur unique. En effet, si deux boutons ont le même identifiant, on ne pourra pas savoir sur lequel l'utilisateur a cliqué.

Pour cette valeur unique, il est possible d'utiliser des valeurs prédéfinies par wxWidgets (http://www.wxwidgets.org/manuals/stable/wx_eventhandlingoverview.html#windowids) pour les actions les plus courantes telles que « ouvrir », « fermer », « aide »,...

Nous pouvons également définir ces valeurs nous même, en prenant garde de ne pas utiliser une valeur présente dans la liste dont je viens de vous donner le lien.

La dernière possibilité est de demander à wxWidgets de nous créer une nouvelle valeur unique, par l'intermédiaire de la fonction **wxNewId()**.

Voici donc ce qu'il suffira de mettre avant le constructeur de la fenêtre principale :

```

| // On affecte les identifiants des deux boutons
| const int MainFrame::ID_BTN_BRWSE = wxNewId();
| const int MainFrame::ID_BTN_START = wxNewId();

```

Maintenant, avant la création des contrôles, il nous reste un petit détail à régler.

Je ne sais pas si vous avez remarqué, mais le fond de la fenêtre est d'une couleur lamentable.

Nous allons remédier à cela en modifiant cette couleur. Nous allons lui affecter une couleur « système » qui est la couleur par défaut pour la « face des boutons », et dont nous pouvons récupérer la valeur grâce à **wxSystemSettings::GetColour(wxSYS_COLOUR_BTNFACE)** :

```

| // On change la couleur du fond de la fenêtre
| SetBackgroundColour(wxSystemSettings::GetColour(wxSYS_COLOUR_BTNFACE));

```

On peut maintenant démarrer la création des contrôles. En premier lieu, le **wxBoxSizer** principal (il est horizontal, et va recevoir les deux zones "gauche" et "droite").

```
// On crée d'abord, le wxBoxSizer principal
wxBoxSizer *mainsizer=new wxBoxSizer(wxHORIZONTAL);
```

Ensuite, vient le premier **wxStaticBoxSizer** (vertical) qui va contenir les contrôles pour les critères de recherche :

```
// On crée maintenant le wxStaticBoxSizer pour la partie de gauche
wxStaticBoxSizer *left_vsizer=new wxStaticBoxSizer( wxVERTICAL, this,
                                                    _T("Critères de recherche :"));
```

Puis le premier contrôle de la partie de gauche : un **wxStaticText**.

Normalement, lors de la création de contrôles avec wxWidgets, il faut donner un identifiant à chaque fois. Cela peut vite devenir pénible, d'autant plus que l'on n'en a pas tout le temps besoin, comme maintenant.

Il existe pour cela un identifiant « passe partout » dont la constante est **wxID_ANY** (et dont la valeur est -1) pour indiquer à wxWidgets de se débrouiller tout seul :

```
// On crée le premier wxStaticText
wxStaticText *label=new wxStaticText(this, wxID_ANY, _T("Nom du(des) fichier(s) :"));
```

Et on l'ajoute au sizer vertical. Comme ce contrôle n'a pas besoin d'être redimensionné, on lui met la proportion « 0 ».

On va mettre une marge de 5 pixels partout, sauf en dessous (il sera collé au contrôle suivant). Ce qui donne :

```
// On l'ajoute au wxStaticBoxSizer
left_vsizer->Add(label, 0, wxLEFT|wxRIGHT|wxTOP, 5);
```

Maintenant, c'est au tour du premier **wxTextCtrl**, dont on a déclaré le pointeur comme variable membre privée de la classe **MainFrame** (son nom est **m_txtName**) :

```
// On crée ensuite le premier wxTextCtrl :
m_txtName=new wxTextCtrl(this, wxID_ANY, _T("*."));
```

Et on l'ajoute au sizer, sachant que cette fois-ci, il devra être étendu pour tenir sur toute la largeur (flag **wxEXPAND**), avoir une marge à gauche, à droite, et en bas (pas en haut, pour être collé au contrôle précédent), et ne pas être redimensionné en hauteur (proportion « 0 ») :

```
// On l'ajoute au sizer
left_vsizer->Add(m_txtName, 0, wxLEFT|wxRIGHT|wxBOTTOM|wxEXPAND, 5);
```

On répète l'opération pour le deuxième **wxStaticText**, sachant que l'on peut réutiliser la variable du premier, qu'il ne faut pas lui mettre de marge supérieure, car le contrôle précédent en a déjà une inférieure :

```
// On crée le deuxième wxStaticText
label=new wxStaticText(this, wxID_ANY, _T("Rechercher dans :"));
// On l'ajoute au sizer
left_vsizer->Add(label, 0, wxLEFT|wxRIGHT, 5);
```

Bien, maintenant, on va corser un peu tout ça. En effet, pour la ligne suivante, nous allons devoir ajouter un **wxBoxSizer** horizontal, pour placer sur la même ligne un **wxTextCtrl** et un **wxButton** :

```
// Le wxBoxSizer horizontal pour la 2ème zone de texte et le bouton "Parcourir"
wxBoxSizer *h_sizer1=new wxBoxSizer(wxHORIZONTAL);
```

Maintenant, on peut créer le **wxTextCtrl** et l'ajouter à ce dernier sizer (pas de marge car on en mettra une au sizer plus tard, proportion = « 1 » car redimensionnable dans le sens du sizer) :

```
// La deuxième zone de texte
m_txtPath=new wxTextCtrl(this, wxID_ANY, _T(""));
```

```

| // On l'ajoute au sizer horizontal
| h_sizer1->Add(m_txtPath, 1, wxALL, 0);

```

On fait de même pour le bouton.

La différence avec le contrôle précédent est que nous allons lui affecter l'identifiant créé un plus haut (**ID_BTN_BRWSE**), et que nous allons lui spécifier une largeur fixe. En effet, si on ne met rien pour la taille du contrôle, wxWidgets va utiliser une taille standard, qui sera trop large pour un bouton comme le nôtre (jetez un coup d'œil aux premières captures d'écran pour vous en rendre compte).

Pour pouvoir accéder au paramètre « taille », il faut d'abord que l'on fournisse le paramètre « position ». Or, c'est le sizer qui va décider de la position du bouton. Nous allons juste lui passer une macro : **wxDefaultPosition**. Cette macro veut en fait dire : **wxPoint(-1, -1)**, c'est-à-dire, une valeur **wxPoint(x,y)** dont on ne spécifie pas les coordonnées.

Ensuite, pour le paramètre « taille », nous allons utiliser une valeur **wxSize(largeur, hauteur)**. Comme nous ne sommes intéressés que par la largeur, nous mettrons la hauteur à « -1 » pour utiliser la valeur par défaut du système.

Notre bouton ne devra être redimensionnable dans aucune direction (proportion « 0 », et pas de flag **wxEXPAND**), et nous allons lui donner une marge à gauche, pour qu'il ne soit pas collé au **wxTextCtrl** précédent :

```

| // Le bouton "Parcourir"
| m_btnBrowse=new wxButton(this, ID_BTN_BRWSE, _T(". . ."), wxDefaultPosition, wxSize(30,-1));
| // On l'ajoute au sizer horizontal
| h_sizer1->Add(m_btnBrowse, 0, wxLEFT, 5);

```

Voilà, notre sizer horizontal est prêt. Nous pouvons l'ajouter au sizer vertical comme un contrôle normal. Il sera non redimensionnable verticalement (proportion « 0 »), extensible horizontalement (flag **wxEXPAND**), et n'aura pas de marges supérieure pour être collé au **wxStaticText**:

```

| // On ajoute le sizer horizontal au sizer vertical
| left_vsizer->Add(h_sizer1, 0, wxLEFT|wxRIGHT|wxBOTTOM|wxEXPAND, 5);

```

Il nous reste maintenant à créer le bouton "Démarrer", et à l'ajouter au sizer vertical, en lui donnant le flag **wxALIGN_RIGHT** pour qu'il soit placé à droite de la zone (ça fait plus joli, je trouve) :

```

| // Le bouton "Démarrer"
| m_btnStart=new wxButton(this, ID_BTN_START, _T("Démarrer"));
| // On l'ajoute au sizer vertical
| left_vsizer->Add(m_btnStart, 0, wxALL|wxALIGN_RIGHT, 5);

```

Nous avons maintenant terminé la création des contrôles de la partie gauche.

Il faut encore que l'on ajoute le **wxStaticBoxSizer** (vertical) au **wxBoxSizer** principal (horizontal) :

```

| // On ajoute le premier wxStaticBoxSizer au wxBoxSizer principal
| mainsizer->Add(left_vsizer, 0, wxALL|wxEXPAND, 5);

```

Nous allons maintenant nous occuper de la partie droite.

Le début est le même que pour la partie de gauche (création du **wxStaticBoxSizer** vertical), mais il faut en plus que l'on récupère le pointeur vers la **wxStaticBox** qu'il crée, pour pouvoir en changer le texte plus tard.

Autre petite chose : il faut que l'on donne une largeur minimale à ce sizer (300 pixels), pour que le **wxListCtrl** que nous allons lui ajouter ensuite soit de taille correcte :

```

| // On crée le deuxième wxStaticBoxSizer
| wxStaticBoxSizer *right_vsizer=new wxStaticBoxSizer(wxVERTICAL, this,
|                                     _T("Résultats de la recherche :"));
| // On récupère le pointeur vers la wxStaticBox
| m_stbNumRes=right_vsizer->GetStaticBox();
| // On définit la largeur minimale pour le deuxième wxStaticBoxSizer
| right_vsizer->SetMinSize(300, -1);

```

Maintenant, on va s'occuper du dernier contrôle: le **wxListCtrl**.

Pour l'affichage de la liste des résultats, on va lui donner le style « **wxLC_REPORT** », qui correspond à la vue « détails » des gestionnaires de fichiers.

Le paramètre « style » se trouvant après la position et la taille, nous allons utiliser les macros « **wxDefaultPosition** » et « **wxDefaultSize** ».

Nous allons ensuite lui créer 3 colonnes : une pour le nom du fichier/dossier, une pour le type (fichier ou dossier) et une pour l'emplacement.

Nous utiliserons pour cela la méthode **wxListCtrl::InsertColumn()**, avec comme paramètres : le numéro de la colonne, le texte de l'en-tête, l'alignement et la largeur.

Nous l'ajouterons ensuite au deuxième sizer vertical, avec une marge de 5 pixels tout autour, une proportion de « **1** » car il devra occuper toute la hauteur de la zone, et le flag **wxEXPAND** pour qu'il en occupe toute la largeur.

Ce qui nous donne :

```
// On crée le wxListCtrl
m_lstResults=new wxListCtrl(this, wxID_ANY, wxDefaultPosition, wxDefaultSize, wxLC_REPORT);
// On ajoute les colonnes
m_lstResults->InsertColumn(0, _T("Nom"), wxLIST_FORMAT_LEFT);
m_lstResults->InsertColumn(1, _T("Type"), wxLIST_FORMAT_CENTER);
m_lstResults->InsertColumn(2, _T("Emplacement"), wxLIST_FORMAT_LEFT);
// On ajoute le wxListCtrl au wxStaticBoxSizer
right_vsizer->Add(m_lstResults, 1, wxALL|wxEXPAND, 5);
```

La zone de droite est maintenant terminée. On peut ajouter le sizer vertical au sizer principal :

```
// On ajoute le deuxième sizer vertical au sizer principal
mainsizer->Add(right_vsizer, 1, wxTOP|wxBOTTOM|wxRIGHT|wxEXPAND, 5);
```

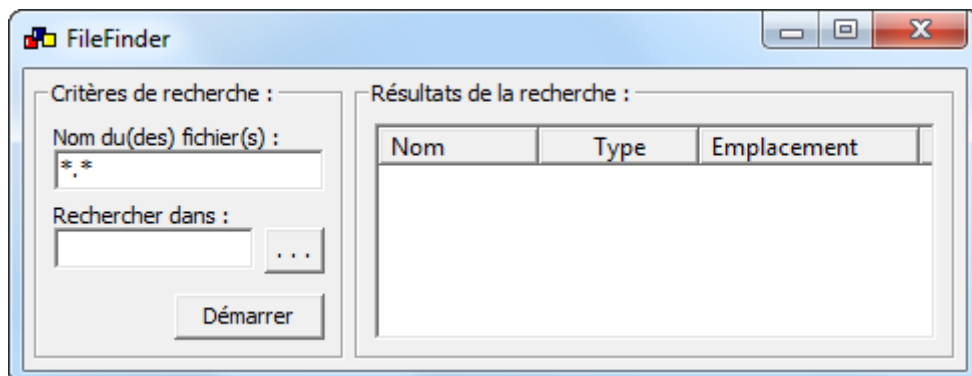
Il reste maintenant à affecter le sizer principal à la fenêtre :

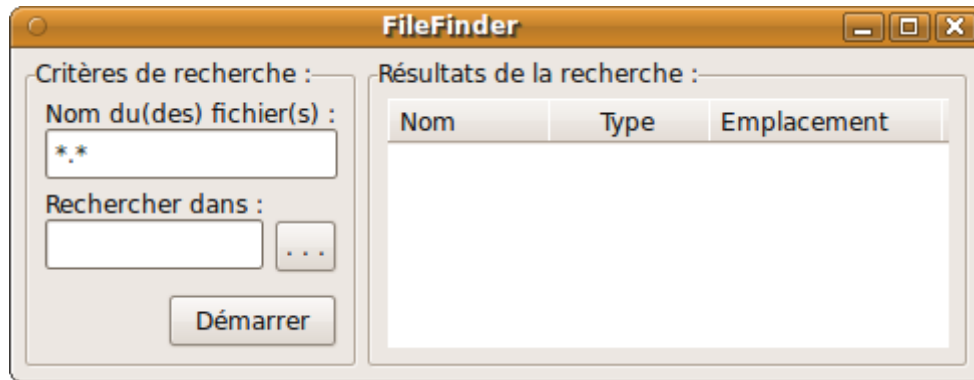
```
// On affecte le wxBoxSizer principal à la fenêtre
SetSizer(mainsizer);
```

On demande le calcul de la taille minimale de la fenêtre en fonction de celle des contrôles :

```
// On calcule la taille minimale de la fenêtre
mainsizer->SetSizeHints(this);
```

Voilà, c'est terminé. Notre interface est prête. Vous pouvez compiler et exécuter, pour voir ce que ça donne :





Voici maintenant un récapitulatif des différentes portions de code que nous avons inséré :

Fichier *mainframe.cpp* :

```
// On récupère la définition de la classe MainFrame
#include "mainframe.h"
// On ajoute le fichier XPM de l'icône
#include "wxwidgets16x16.xpm"

// On affecte les identifiants des deux boutons
const int MainFrame::ID_BTN_BRWSE = wxNewId();
const int MainFrame::ID_BTN_START = wxNewId();

// Le constructeur
MainFrame::MainFrame() : wxFrame(NULL, wxID_ANY, _T("FileFinder"))
{
    // On affecte l'icône à la fenêtre
    SetIcon(wxwidgets16x16_xpm);
    // On change la couleur du fond de la fenêtre
    SetBackgroundColour(wxSystemSettings::GetColour(wxSYS_COLOUR_BTNFACE));
    // On crée d'abord, le wxBoxSizer principal
    wxBoxSizer *mainsizer=new wxBoxSizer(wxHORIZONTAL);
    // On crée maintenant le wxStaticBoxSizer pour la partie de gauche
    wxStaticBoxSizer *left_vsizer=new wxStaticBoxSizer(wxVERTICAL, this,
        _T("Critères de recherche :"));

    // On crée le premier wxStaticText
    wxStaticText *label=new wxStaticText(this, wxID_ANY, _T("Nom du(des) fichier(s) :"));
    // On l'ajoute au wxStaticBoxSizer
    left_vsizer->Add(label, 0, wxLEFT|wxRIGHT|wxTOP, 5);
    // On crée ensuite le premier wxTextCtrl :
    m_txtName=new wxTextCtrl(this, wxID_ANY, _T("*. *"));
    // On l'ajoute au sizer
    left_vsizer->Add(m_txtName, 0, wxLEFT|wxRIGHT|wxBOTTOM|wxEXPAND, 5);
    // On crée le deuxième wxStaticText
    label=new wxStaticText(this, wxID_ANY, _T("Rechercher dans :"));
    // On l'ajoute au sizer
    left_vsizer->Add(label, 0, wxLEFT|wxRIGHT, 5);
    // Le wxBoxSizer horizontal pour la 2ème zone de texte et le bouton "Parcourir"
    wxBoxSizer *h_sizer1=new wxBoxSizer(wxHORIZONTAL);
    // La deuxième zone de texte
    m_txtPath=new wxTextCtrl(this, wxID_ANY, _T(""));
    // On l'ajoute au sizer horizontal
    h_sizer1->Add(m_txtPath, 1, wxALL, 0);
    // Le bouton "Parcourir"
    m_btnBrowse=new wxButton(this, ID_BTN_BRWSE, _T(". . ."),
        wxDefaultPosition, wxSize(30, -1));
    // On l'ajoute au sizer horizontal
    h_sizer1->Add(m_btnBrowse, 0, wxLEFT, 5);
    // On ajoute le sizer horizontal au sizer vertical
    left_vsizer->Add(h_sizer1, 0, wxLEFT|wxRIGHT|wxBOTTOM|wxEXPAND, 5);
    // Le bouton "Démarrer"
    m_btnStart=new wxButton(this, ID_BTN_START, _T("Démarrer"));
    // On l'ajoute au sizer vertical
    left_vsizer->Add(m_btnStart, 0, wxALL|wxALIGN_RIGHT, 5);
    // On ajoute le premier wxStaticBoxSizer au wxBoxSizer principal
    mainsizer->Add(left_vsizer, 0, wxALL|wxEXPAND, 5);
}
```

```

|
|         // On crée le deuxième wxStaticBoxSizer
|         wxStaticBoxSizer *right_vsizer=new wxStaticBoxSizer(wxVERTICAL, this,
|
|         // On récupère le pointeur vers la wxStaticBox
|         m_stbNumRes=right_vsizer->GetStaticBox();
|         // On définit la largeur minimale pour le deuxième wxStaticBoxSizer
|         right_vsizer->SetMinSize(300, -1);
|         // On crée le wxListCtrl
|         m_lstResults=new wxListCtrl(this, wxID_ANY, wxDefaultPosition, wxDefaultSize,
| wxLC_REPORT);
|
|         // On ajoute les colonnes
|         m_lstResults->InsertColumn(0, _T("Nom"), wxLIST_FORMAT_LEFT);
|         m_lstResults->InsertColumn(1, _T("Type"), wxLIST_FORMAT_CENTER);
|         m_lstResults->InsertColumn(2, _T("Emplacement"), wxLIST_FORMAT_LEFT);
|         // On ajoute le wxListCtrl au wxStaticBoxSizer
|         right_vsizer->Add(m_lstResults, 1, wxALL|wxEXPAND, 5);
|         // On ajoute le deuxième sizer vertical au sizer principal
|         mainsizer->Add(right_vsizer, 1, wxTOP|wxBOTTOM|wxRIGHT|wxEXPAND, 5);
|         // On affecte le wxBoxSizer principal à la fenêtre
|         SetSizer(mainsizer);
|         // On calcule la taille minimale de la fenêtre
|         mainsizer->SetSizeHints(this);
|         // Et voilà, c'est terminé
|     }
|
| // Le Destructeur
| MainFrame::~MainFrame()
| {
|     // Nous ajouterons ici les éventuelles libérations de mémoire nécessaires
|     // avant la fermeture de la fenêtre
| }

```

Allez, puisque vous avez été très attentifs, je vais vous donner un petit bonus (il ne concerne que les utilisateurs de Windows).

Vous avez sans-doute remarqué deux choses :

- Premièrement, l'exécutable qui a été généré ne possède pas d'icône.
- Deuxièmement, les contrôles ne prennent pas en compte le thème de Windows.

Nous allons remédier à cela par l'intermédiaire d'un fichier « ressources ».

En effet, sous Windows, il est possible d'ajouter de nombreuses données dans l'exécutable (images, sons, ...).

Pour cela, nous allons d'abord créer le fichier ressources.

Sous Code::Blocks, créez un nouveau fichier vide (menu *File* → *New* → *Empty file*), répondez « Oui » lorsqu'on vous demande si vous voulez ajouter ce fichier au projet courant, et enregistrez-le sous le nom « *resources.rc* ».

Nous allons tout d'abord ajouter l'icône de l'exécutable. Vous pouvez utiliser n'importe quel fichier « .ico » valide.

De mon côté, je vais utiliser le fichier icône « standard » wxWidgets. Il se trouve normalement dans l'arborescence de votre installation des libs wxWidgets, dans le dossier « *include\wx\msw* » et se nomme « *std.ico* ».

Nous allons donc indiquer au compilateur qu'il doit l'intégrer à l'exécutable.

Dans le fichier « *resources.rc* » que vous venez de créer, ajouter la ligne :

```

| appIcon ICON "wx/msw/std.ico"
|

```

A noter que le nom « **appIcon** » n'a pas été donné au hasard. Windows classe les icônes par ordre alphabétique, et prend le premier pour l'affichage dans l'explorateur. Avec un tel nom, il y a de fortes chances que l'icône « *std.ico* » soit placée en tête de liste, et soit utilisée pour représenter l'exécutable dans l'explorateur Windows.

Nous allons également intégrer le fichier *Manifest* à cet exécutable, pour que les contrôles aient le look Windows.

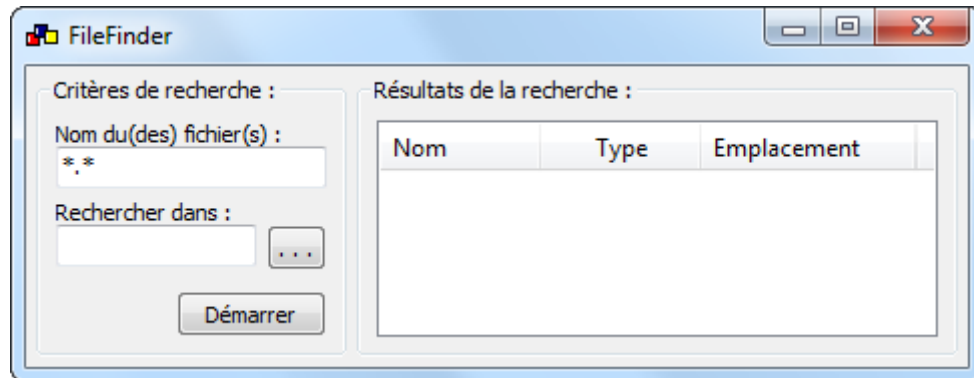
Ce fichier manifest se trouve normalement au même endroit que l'icône utilisée précédemment.

Ajoutez la ligne suivante au fichier « *resources.rc* » :

```
1 24 "wx/msw/wx.manifest"
```

Le « 1 » correspond à l'identifiant du « Manifest », et le « 24 » à son type. Ces deux valeurs doivent toujours être celles-ci pour un fichier manifest.

Vous pouvez maintenant compiler. Votre fichier « *FileFinder.exe* » doit maintenant être doté d'une icône identique à celle que nous avons affectée à la fenêtre. De plus, si vous exécutez une dernière fois l'application, les contrôles ont maintenant le « look Windows » (enfin, pour ceux qui sont sous Windows XP, Vista ou Seven).



Les méthodes événementielles

Nous avons créé une belle fenêtre, mais elle ne fait pour l'instant pas grand chose.

Il faut que l'on indique à notre application de réagir en fonction des actions de l'utilisateur :

- Lorsqu'il clique sur le bouton « Parcourir », il faut ouvrir une boîte de dialogue de sélection de répertoire pour lui permettre de sélectionner le dossier dans lequel va débiter la recherche.
- Lorsqu'il clique sur le bouton « Démarrer », il faut mettre en route la recherche des fichiers avec les critères qu'il a fournis.

Nous allons donc dans un premier temps modifier notre classe **MainFrame** pour lui ajouter les méthodes qui vont être appelées lors des clics sur les boutons « Parcourir » et « Démarrer ».

Ensuite, nous allons connecter ces méthodes à la table des événements, de façon à indiquer à l'OS où se trouve le code à exécuter lorsque l'utilisateur clique sur un bouton. Comme nous avons deux événements à gérer, nous allons pouvoir mettre en place deux méthodes différentes pour cette « connexion ».

Nous allons donc ajouter deux méthodes à la classe **MainFrame**.

La première, que nous appellerons « **OnButtonBrowseClicked** » sera appelée chaque fois que l'utilisateur cliquera sur le bouton « Parcourir ». La deuxième, que nous appellerons « **OnButtonStartClicked** » sera appelée lors du clic sur le bouton « Démarrer ».

Nous ne nous occuperons pas pour l'instant du contenu de ces méthodes. Nous y placerons simplement un appel à la fonction **wxMessageBox(...)** afin de vérifier qu'elles sont bien appelées.

Si l'on jette un coup d'œil à la documentation officielle concernant le **wxButton** (http://www.wxwidgets.org/manuals/stable/wx_wxbutton.html), à la rubrique « Event handling » (gestion des événements), il est dit que ce contrôle réagit à un « **wxEVT_COMMAND_BUTTON_CLICKED** ».

Le début de cette constante nous permet de savoir que les méthodes que nous allons avoir à créer devront posséder un paramètre de type **wxCommandEvent**, afin que l'on puisse récupérer quelques informations sur le contrôle qui a généré cet événement.

Ce type d'événement est celui que vous aurez le plus souvent à utiliser dans vos programmes wxWidgets (pour les menus, les contrôles basiques tels que les boutons radio, les cases à cocher, ...).

Il y a malgré tout plusieurs types d'événements, et tous n'utilisent pas le **wxCommandEvent**. Ainsi, le **wxListCtrl** utilise un

wxCommandEvent, qui contiendra des informations spécifiques au **wxListCtrl**.

Pour plus d'infos sur la gestion des événements sous wxWidgets, je vous recommande vivement de lire l'article qui y est consacré dans la documentation officielle (il est malheureusement en anglais, mais il permet d'en comprendre les grands principes) : http://www.wxwidgets.org/manuals/stable/wx_eventhandlingoverview.html.

Une dernière petite chose : chaque événement est transmis à la méthode qui lui est associée par sa référence.

Pour ajouter ces deux méthodes à la classe **MainFrame**, il suffit d'ajouter leur déclaration dans le header cette classe, et leur définition dans le fichier source, mais ça, vous le saviez déjà, en tant que bon programmeur C++.

Ce qui donne, pour le fichier *mainframe.h* :

```
-----  
#ifndef MAINFRAME_H_INCLUDED  
#define MAINFRAME_H_INCLUDED  
  
// On ajoute les headers "basiques" de wxWidgets  
#include <wx/wx.h>  
// On ajoute le header du contrôle wxListCtrl, qui n'est pas inclus  
// Dans les headers "basiques"  
#include <wx/listctrl.h>  
  
// Définition de la classe "MainFrame"  
class MainFrame : public wxFrame  
{  
public:  
    // Le constructeur  
    MainFrame();  
    // Le destructeur  
    ~MainFrame();  
private:  
    // Déclaration des méthodes événementielles  
    void OnButtonBrowseClicked(wxCommandEvent &event);  
    void OnButtonStartClicked(wxCommandEvent &event);  
    // Les pointeurs vers les wxTextCtrl pour le nom/les jockers et le dossier de démarrage  
    wxTextCtrl *m_txtName, *m_txtPath;  
    // Les pointeur vers les boutons "Parcourir" et "Démarrer"  
    wxButton *m_btnBrowse, *m_btnStart;  
    // Les identifiants pour ces même boutons  
    static const int ID_BTN_BRWSE, ID_BTN_START;  
    // Le pointeur vers la wxStaticBox de droite pour l'affichage du nombre de résultats trouvés  
    wxStaticBox *m_stbNumRes;  
    // Le pointeur vers le wxListCtrl pour l'affichage des résultats.  
    wxListCtrl *m_lstResults;  
};  
#endif // MAINFRAME_H_INCLUDED  
-----
```

Et, pour le fichier *mainframe.cpp* (je ne mets ici que les lignes à ajouter à la fin du fichier, pour ne pas surcharger le cours) :

```
-----  
void MainFrame::OnButtonBrowseClicked(wxCommandEvent &event)  
{  
    wxMessageBox(_T("Appel de la méthode 'OnButtonBrowseClicked'"));  
}  
  
void MainFrame::OnButtonStartClicked(wxCommandEvent &event)  
{  
    wxMessageBox(_T("Appel de la méthode 'OnButtonStartClicked'"));  
}  
-----
```

C'est tout. Vous pouvez compiler si ça vous chante, mais de toute façon, il ne va rien se passer, car nous n'avons pas connecté les clics sur les boutons à leur méthodes respectives.

C'est ce que nous allons voir maintenant.

Pour que notre fenêtre puisse rediriger les événements vers les méthodes que l'on vient de créer, il faut impérativement déclarer (dans le header) et définir (dans le fichier cpp) une « table des événements ».

- Pour la déclaration, il suffit d'ajouter la macro **DECLARE_EVENT_TABLE()** dans le header de notre classe **MainFrame** (fichier *mainframe.h*).

```

#ifndef MAINFRAME_H_INCLUDED
#define MAINFRAME_H_INCLUDED

// On ajoute les headers "basiques" de wxWidgets
#include <wx/wx.h>
// On ajoute le header du contrôle wxListCtrl, qui n'est pas inclus
// Dans les headers "basiques"
#include <wx/listctrl.h>

// Définition de la classe "MainFrame"
class MainFrame : public wxFrame
{
public:
    // Le constructeur
    MainFrame();
    // Le destructeur
    ~MainFrame();
private:
    // Déclaration des méthodes événementielles
    void OnButtonBrowseClicked(wxCommandEvent &event);
    void OnButtonStartClicked(wxCommandEvent &event);
    // Les pointeurs vers les wxTextCtrl pour le nom/les jockers et le dossier de démarrage
    wxTextCtrl *m_txtName, *m_txtPath;
    // Les pointeur vers les boutons "Parcourir" et "Démarrer"
    wxButton *m_btnBrowse, *m_btnStart;
    // Les identifiants pour ces même boutons
    static const int ID_BTN_BRWSE, ID_BTN_START;
    // Le pointeur vers la wxStaticBox de droite pour l'affichage du nombre de résultats trouvés
    wxStaticBox *m_stbNumRes;
    // Le pointeur vers le wxListCtrl pour l'affichage des résultats.
    wxListCtrl *m_lstResults;
    // Déclaration de la table des événements
    DECLARE_EVENT_TABLE();
};
#endif // MAINFRAME_H_INCLUDED

```

- Pour la définition, il faut utiliser deux macros : une première pour marquer le « début » de la table des événements (**BEGIN_EVENT_TABLE(..., ...)**), et qui prend en paramètres le nom de notre classe, ainsi que celui de celle dont elle dérive, ainsi qu'une autre pour « fermer » cette table (**END_EVENT_TABLE**).

Comme je vous l'ai dit précédemment, nous allons utiliser deux techniques différentes pour « connecter » les boutons aux méthodes de la fenêtre.

- La première consiste à ajouter une macro dans la définition de la table des événements. Cette macro, par son nom, définira le type d'événement à gérer (clic sur un bouton dans notre cas), et prendra en paramètre l'identifiant du bouton (que nous avons défini dans le chapitre 2), et le nom de la méthode à associer.
- La deuxième consistera à connecter le bouton au moment de sa création, par l'intermédiaire de la méthode **wxEventHandler::Connect(...)** (notre fenêtre est dérivée de **wxFrame**, dérivée de **wxTopLevelWindow**, dérivée de **wxWindow**, dérivée de **wxEventHandler**). Nous allons donc utiliser la première solution (qui est d'ailleurs la plus souvent utilisée), pour connecter le bouton « parcourir » à la méthode **MainFrame::OnButtonBrowseClicked**.

Nous voulons gérer l'événement « clic sur un bouton ». En regardant la documentation officielle, on peut voir que cela se fait avec la macro **EVT_BUTTON(identifiant,méthode_associée)**.

Voici ce que ça donne (fichier *mainframe.cpp*):

```

// On affecte les identifiants des deux boutons
const int MainFrame::ID_BTN_BRWSE = wxNewId();
const int MainFrame::ID_BTN_START = wxNewId();

// Définition de la table des événements
BEGIN_EVENT_TABLE(MainFrame,wxFrame)
    // Connection du bouton "Parcourir" à la méthode MainFrame::OnButtonBrowseClicked

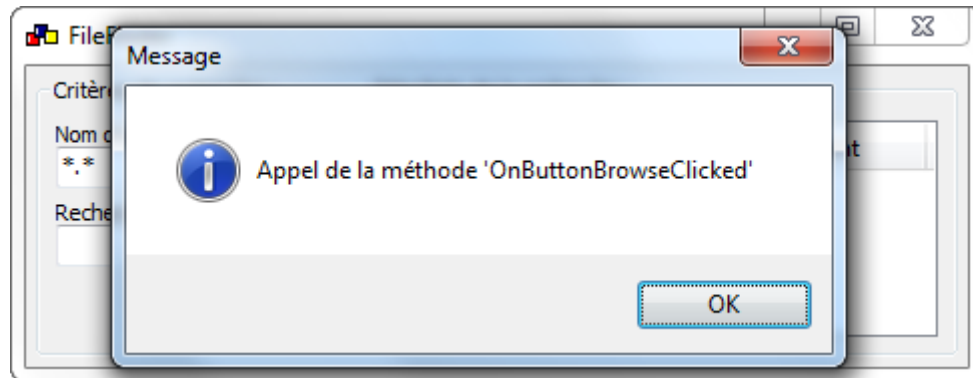
```

```

|   EVT_BUTTON(ID_BTN_BRWSE, MainFrame::OnButtonBrowseClicked)
| END_EVENT_TABLE()
|
| // Le constructeur
| MainFrame::MainFrame() : wxFrame(NULL, wxID_ANY, _T("FileFinder"))
| {
|     // .....
| }

```

Vous pouvez désormais compiler, et tester le bouton parcourir : vous devriez obtenir un petit message vous indiquant que l'appel de la méthode **MainFrame::OnButtonBrowseClicked** s'est bien effectué correctement :



Comme vous pouvez le voir, c'était tout simple, et ça marche à merveille.

Voici quelques petites précisions sur cette technique :

Il est facile, par un simple coup d'œil dans la table des événements, de retrouver à quelle méthode est connecté un bouton, ou n'importe quel contrôle (si bien sûr vous avez donnée un nom explicite à vos identifiants).

Il est possible de connecter plusieurs contrôles à la même méthode (dans notre exemple, si vous ajoutez **EVT_BUTTON(ID_BTN_START, MainFrame::OnButtonBrowseClicked)** dans la table des événements, un clic sur le bouton « Démarrer » fera apparaître le même message que pour le bouton « Parcourir »).

Le problème, c'est que tout est défini au moment de la compilation. Il n'est plus possible d'utiliser ces macros lors de l'exécution (ce qui peut être utile, par exemple pour des menus dont le contenu change en fonction de ce qui est sélectionné).

Nous allons maintenant nous attaquer à la deuxième technique pour connecter un contrôle (dans notre cas le bouton « Démarrer ») à une méthode.

Comme je vous l'ai dit plus haut, la connexion se fait au moment de la création du contrôle (en fait, elle peut se faire n'importe où et n'importe quand, mais j'ai pris l'habitude de la faire tout de suite après avoir créé le contrôle).

On utilise dans ce cas la méthode **wxEventHandler::Connect(...)**.

Le premier paramètre est l'identifiant du bouton, c'est à dire, dans notre cas, **ID_BTN_START**.

Le deuxième correspond au type d'événement que l'on veut gérer : pour un clic sur un bouton, c'est **wxEVT_COMMAND_BUTTON_CLICKED**. Vous pouvez retrouver le nom de cette macro dans la documentation officielle concernant le **wxButton**, rubrique « Event Handling » : http://www.wxwidgets.org/manuals/stable/wx_wxbutton.html

Pour le troisième, c'est un peu plus compliqué. Retenez simplement qu'il faut passer une **wxObjectEventFunction**.

Regardez la documentation officielle concernant la méthode **wxEventHandler::Connect()** (http://www.wxwidgets.org/manuals/stable/wx_wxevthandler.html#wxevthandlerconnect), il y a un petit exemple.

Ainsi, pour un événement de type **wxCommandEvent**, il faut donner ce paramètre grâce à **wxCommandEventHandler(Nom_de_la_méthode)**.

Pour un événement de type **wxSizeEvent**, on aurait mis **wxSizeEventHandler(MainFrame::OnSize)**, et ainsi de suite.

Ce qui donne (je vous laisse retrouver où ça va dans le fichier *mainframe.cpp*):

```

// Le bouton "Démarrer"
m_btnStart=new wxButton(this, ID_BTN_START, _T("Démarrer"));
Connect(ID_BTN_START, wxEVT_COMMAND_BUTTON_CLICKED,
        wxCommandEventHandler(MainFrame::OnButtonStartClicked));
// On l'ajoute au sizer vertical
left_vsizer->Add(m_btnStart,0,wxALL|wxALIGN_RIGHT,5);

```

Et voilà, c'est tout aussi simple (et aussi efficace) que la première solution.

L'avantage de celle-ci, c'est que l'on peut regrouper la création du contrôle, et sa connexion à une méthode.

Un autre avantage (et non des moindres) est que la connexion peut se faire en cours d'exécution du programme, ce que ne permet pas la technique précédente.

Il est également possible de « déconnecter » un contrôle d'une méthode grâce à la classe **wxEventHandler** dont est dérivée notre fenêtre.

Voilà, nous avons réussi à connecter nos deux boutons à leurs méthodes respectives.

Il ne nous reste plus qu'à placer le bon code dans ces méthodes, et notre petit utilitaire sera fonctionnel.

Le code pour sélectionner le dossier de démarrage

Notre interface est maintenant en place, et fonctionnelle (même si, pour l'instant, seules des petites boîtes de message s'affichent lorsque l'on clique sur les boutons).

Nous allons maintenant écrire le code correspondant à la méthode **MainFrame::OnButtonBrowseClicked()**.

Cette méthode est appelée lorsque l'utilisateur clique sur le bouton « Parcourir », et doit permettre la sélection du dossier dans lequel va commencer la recherche.

Cette sélection de dossier se fait par l'intermédiaire de la classe **wxDirDialog** qui a été créée spécialement pour cela (http://www.wxwidgets.org/manuals/stable/wx_wxdirdialog.html).

Si l'on jette un coup d'œil à la documentation officielle concernant le constructeur de cette classe, on peut voir qu'il lui faut comme paramètres :

- Un pointeur vers la fenêtre parente.
- Le message à afficher dans la boîte de dialogue de sélection.
- Le répertoire présélectionné.
- Le style de cette boîte (nous indiquerons le style **wxDD_DIR_MUST_EXIST**).
- Pour les autres paramètres, nous laisserons les valeurs par défaut.

Après avoir affiché cette boîte de dialogue, et si l'utilisateur n'a pas cliqué sur le bouton « Annuler » qu'elle contient, il nous suffira de récupérer le chemin complet du répertoire sélectionné, et de l'afficher dans le **wxTextCtrl** correspondant (dont nous avons stocké le pointeur dans la variable **m_txtPath**). Ce qui donne comme code (en remplacement du simple appel à la fonction **wxMessageBox** dans le fichier *mainframe.cpp*):

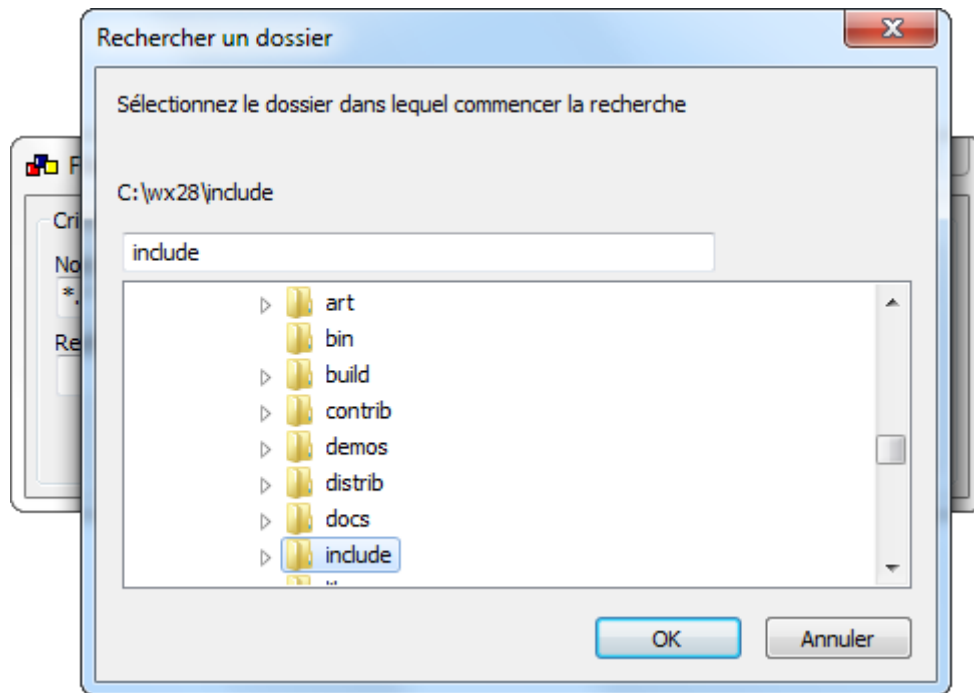
```

#include <wx/dirdlg.h>
void MainFrame::OnButtonBrowseClicked(wxCommandEvent &event)
{
    // On crée le wxDirDialog
    wxDirDialog dirdlg(this, // La fenêtre parente
        _T("Sélectionnez le dossier dans lequel commencer la recherche"), // Le message
        _T(""), // Le dossier par défaut
        wxDD_DIR_MUST_EXIST); // Le style de la boîte de dialogue
    // On l'affiche jusqu'à ce que l'utilisateur ait cliqué sur "Ok" ou "Annuler"
    int result=dirdlg.ShowModal();
    // Si le bouton "Annuler" a été pressé, alors, on sort
    if (result==wxID_CANCEL) return;
    // Sinon, on récupère le chemin complet du dossier sélectionné
    wxString sPath=dirdlg.GetPath();
    // Et on l'affiche dans le wxTextCtrl correspondant
    m_txtPath->SetValue(sPath);
}

```

| } _____ |

Voilà, ce n'est pas plus compliqué que cela. Vous pouvez compiler et tester, ça marche !



La recherche des fichiers et des dossiers

Il ne nous manque plus que l'essentiel : la recherche des fichiers proprement dite ; après cela, notre application sera fonctionnelle.

Il nous faut donc écrire le code de la méthode **MainFrame::OnButtonStartClicked()**.

Voici ce qu'elle devra effectuer :

- Lancer l'énumération des fichiers et sous/dossiers contenus dans le répertoire de départ.
- Ajouter les fichiers et les dossiers trouvés au **wxListCtrl** pour l'affichage des résultats de la recherche.

Pour obtenir la liste des fichiers/dossiers, nous disposons d'une classe spécialement étudiée pour gérer les actions relatives à un répertoire : **wxDir** (http://www.wxwidgets.org/manuals/stable/wx_wkdir.html).

Cette classe nous propose 3 possibilités pour obtenir le listing du contenu d'un dossier :

- La méthode **GetFirst() - GetNext()** : avec cette méthode, nous allons devoir nous débrouiller pour lister le contenu des éventuels sous-dossiers.
- La méthode **GetAllFiles()** : l'inconvénient de cette méthode est qu'elle ne nous rend pas la main avant d'avoir fini l'énumération du contenu. L'utilisateur ne pourrait pas annuler la recherche si celle-ci devient trop longue. De plus, le résultat de l'énumération est placé dans un **wxArrayString**, ce qui peut vite prendre beaucoup de mémoire inutilement.
- La méthode **Traverse()** : très simple d'utilisation, cette méthode nécessite par contre la création d'une classe supplémentaire : la classe **wxDirTraverser**. C'est la méthode que nous allons utiliser.

Pour utiliser cette dernière méthode, il nous faut d'abord créer une classe dérivée de **wxDirTraverser()** (http://www.wxwidgets.org/manuals/stable/wx_wkdirtraverser.html).

Cette classe contiendra les méthodes **OnFile()** et **OnDir()** qui seront appelées pour chaque fichier/dossier que **wxDir::Traverse()** trouvera.

Ensuite, lorsqu'on lance la recherche, on passe une instance de cette classe à la méthode **wxDir::Traverse()**.

Pour chaque fichier trouvé, un appel est fait à **wxDirTraverser::OnFile()**.

Pour chaque dossier (vous l'aurez deviné), un appel est fait à `wxDirTraverser::OnDir()`.

C'est donc à l'intérieur de ces deux fonctions que nous allons placer le code pour ajouter le résultat au `wxListCtrl`.

Nous y placerons également un bout de code pour vérifier si l'utilisateur a éventuellement demandé de stopper la recherche.

A ce sujet, nous devons proposer à l'utilisateur cette possibilité. Pour ce faire, nous allons simplement changer le texte du bouton « Démarrer » en « Arrêter ». Il suffira donc, dans l'appel de la méthode associée à ce bouton, de regarder quel est le texte du bouton, pour savoir ce que l'on doit faire :

- Si le texte est « Démarrer », alors, il faut démarrer la recherche.
- Si le texte est « Arrêter », alors il faut stopper tout.

Pour demander l'arrêt (ou au contraire, autoriser la continuité) de la recherche, tout se passe dans les méthodes `wxDirTraverser::OnFile()` et `wxDirTraverser::OnDir()`. En effet, ces deux méthodes doivent renvoyer une valeur de type `wxDirTraverseResult` qui peut prendre les valeurs suivantes:

- `wxDIR_IGNORE` : Ignorer le contenu du répertoire qui vient d'être trouvé.
- `wxDIR_STOP` : Arrêter l'énumération (intéressant, non ?).
- `wxDIR_CONTINUE` : Vous avez deviné : continuer l'énumération.

Pour indiquer au `wxDirTraverser` qu'il doit stopper l'énumération, il suffit d'ajouter une variable statique de type booléen dans la méthode `MainFrame::OnButtonStartClicked()` et de lui en passer le pointeur en paramètre, et le tour est joué.

Voici, concrètement ce que ça va donner.

En premier lieu, nous devons créer la classe dérivée de `wxDirTraverser`. Nous l'appellerons `DirTraverser`, tout simplement.

Commencez donc par ajouter deux fichiers vides (menu *File* → *New* → *Empty file*, répondez « Oui » lorsqu'on vous demande si vous désirez les ajouter au projet courant, et nommez-les respectivement `dirtraverser.h` et `dirtraverser.cpp`.

Voici leur contenu initial (nous compléterons par la suite).

La déclaration de la classe (fichier `dirtraverser.h`) :

```
-----  
#ifndef DIRTRAVERSER_H_INCLUDED  
#define DIRTRAVERSER_H_INCLUDED  
  
#include <wx/dir.h>  
  
class DirTraverser : public wxDirTraverser  
{  
public:  
    // Le constructeur : on lui passe en paramètres le pointeur vers le wxListCtrl  
    // pour l'ajout des résultats, ainsi que le pointeur vers la variable de type  
    // booléen pour l'arrêt de l'énumération  
    DirTraverser(wxListCtrl *lstResults, bool *stopFlag);  
    // La méthode appelée lorsqu'un fichier est trouvé  
    virtual wxDirTraverseResult OnFile(const wxString &filename);  
    // La méthode appelée lorsqu'un dossier est trouvé  
    virtual wxDirTraverseResult OnDir(const wxString &dirname);  
private:  
    // Les variables pour stocker les pointeurs passés en paramètres au constructeur  
    wxListCtrl *m_lstResults;  
    bool *m_bStopFlag;  
};  
  
#endif // DIRTRAVERSER_H_INCLUDED  
-----
```

La définition de la classe (fichier `dirtraverser.cpp`) :

```
-----  
#include "dirtraverser.h"  
DirTraverser::DirTraverser(wxListCtrl *lstResults, bool *stopFlag)  
{  
    // On stocke les deux pointeurs passés en paramètres  
    m_lstResults=lstResults;  
}
```

```

|     m_bStopFlag=stopFlag;
| }
|
| wxDirTraverseResult DirTraverser::OnFile(const wxString &filename)
| {
|     // Cette méthode sera appelée pour chaque fichier trouvé
|     return wxDIR_CONTINUE;
| }
|
| wxDirTraverseResult DirTraverser::OnDir(const wxString &dirname)
| {
|     // Cette méthode sera appelée pour chaque dossier trouvé
|     return wxDIR_CONTINUE;
| }

```

Voilà : rien de bien particulier, car tout a été expliqué avant.

Nous allons maintenant écrire le code de la méthode **MainFrame::OnButtonStartClicked()**, et nous reviendrons plus tard pour finaliser la classe **DirTraverser**.

Tout d'abord, pour pouvoir utiliser le **DirTraverser** que nous venons de créer, il faut ajouter le header de cette classe au début du fichier *mainframe.cpp*.

```

// On récupère la définition de la classe DirTraverser
#include "dirtraverser.h"

```

Puis, dans la méthode **MainFrame::OnButtonStartClicked()**, nous allons placer la déclaration de la variable statique qui va nous servir pour demander l'arrêt de la recherche:

```

static bool bStopFlag;

```

Viendra ensuite la condition pour savoir si l'on veut démarrer la recherche, ou l'arrêter :

```

if(m_btnStart->GetLabel()==_T("Démarrer"))
{
    // Placer ici le code pour lancer la recherche
} else {
    // Placer ici le code pour stopper la recherche
}

```

Occupons nous dans un premier temps de la portion de code pour stopper la recherche : nous avons dit plus haut qu'il suffisait de modifier la valeur de la variable statique **bStopFlag**.

Ce qui nous donne :

```

bStopFlag=true;

```

C'est tout. Nous nous chargerons, à chaque appel de **DirTraverser::OnFile()** et **DirTraverser::OnDir()**, de regarder la valeur de cette variable puisque nous disposerons de son pointeur.

Maintenant, la portion de code pour lancer la recherche :

Il faut d'abord vérifier que le répertoire dans lequel nous allons démarrer la recherche soit valide :

```

// On récupère le répertoire dans lequel faire la recherche
wxString sFolder=m_txtPath->GetValue();
// On vérifie que ce répertoire existe bien, sinon, on sort
if (!wxDir::Exists(sFolder)) return;

```

Maintenant que l'on a vérifié la validité du répertoire, on peut lancer la recherche.

Il faut d'abord modifier le label du bouton « Démarrer » pour mettre « Arrêter » à la place.

```

m_btnStart->SetLabel(_T("Arrêter"));

```

Ensuite, on peut construire l'objet **wxDi**r :

```
wxDi dir(sFolder);
```

Puis finalement lancer la recherche en créant au passage un objet **DirTraverser** et en s'assurant que le booléen soit bien réinitialisé :

```
bStopFlag=false;
DirTraverser traverser(m_lstResults, &bStopFlags);
dir.Traverse(traverser, m_txtName->GetValue(), wxDIR_DIRS|wxDIR_FILES);
```

Vous aurez remarqué, au passage, le deuxième paramètre qui correspond au masque de recherche.

Lorsque la fonction **wxDi::Traverse()** nous rend la main, c'est que la recherche est terminée, ou que l'on a demandé son arrêt.

On peut donc remettre le label du bouton "Démarrer" à sa valeur d'origine :

```
m_btnStart->SetLabel(_T("Démarrer"));
```

Et c'est tout.

Voici donc un récapitulatif du code de la méthode **MainFrame::OnButtonStartClicked()** :

```
void MainFrame::OnButtonStartClicked(wxCommandEvent &event)
{
    static bool bStopFlag;
    if(m_btnStart->GetLabel()==_T("Démarrer"))
    {
        // On récupère le répertoire dans lequel faire la recherche
        wxString sFolder=m_txtPath->GetValue();
        // On vérifie que ce répertoire existe bien, sinon, on sort
        if (!wxDi::Exists(sFolder)) return;
        // On change le label du bouton "Démarrer" en "Arrêter"
        m_btnStart->SetLabel(_T("Arrêter"));
        // On construit l'objet wxDi
        wxDi dir(sFolder);
        // On construit l'objet DirTraverser
        DirTraverser traverser(m_lstResults, &bStopFlag);
        // On ré-initialise le booléen
        bStopFlag=false;
        // Et on lance la recherche
        dir.Traverse(traverser, m_txtName->GetValue(), wxDIR_DIRS);
        // Quand on arrive ici, c'est que la recherche est terminée
        // On peut donc remettre le label du bouton "Démarrer"
        m_btnStart->SetLabel(_T("Démarrer"));
    } else {
        bStopFlag=true;
    }
}
```

Bien, maintenant, il ne nous reste plus qu'à écrire le code des méthodes **OnFile()** et **OnDir()** de notre **DirTraverser**.

Elles doivent faire toutes les deux sensiblement la même chose :

- Ajouter le fichier/dossier qui leur est transmis en paramètre dans le **wxListCtrl**.
- Vérifier l'état du booléen pour éventuellement arrêter la recherche en cours

La seule différence est le texte qui doit être affiché dans la deuxième colonne du **wxListCtrl** : « Fichier » ou « Dossier ».

Pour ne pas avoir à retaper deux fois le même code, nous allons ajouter une méthode **AddItemToListCtrl()** au **DirTraverser**, et lui passer en paramètre le nom du fichier/dossier à ajouter, ainsi que le texte à mettre dans la deuxième colonne.

Cette méthode se chargera de séparer le nom du fichier/dossier de son emplacement, vérifiera l'état du booléen, et renverra la valeur que les fonctions **OnFile** et **OnDir** doivent retourner pour la suite ou l'arrêt de la recherche.

Commençons donc par ajouter la déclaration de cette nouvelle méthode (fichier *dirtraverser.h*):

```
#ifndef DIRTRAVERSER_H_INCLUDED
#define DIRTRAVERSER_H_INCLUDED

#include <wx/dir.h>
#include <wx/listctrl.h>
class DirTraverser : public wxDirTraverser
{
public:
    // Le constructeur : on lui passe en paramètres le pointeur vers le wxListCtrl
    // pour l'ajout des résultats, ainsi que le pointeur vers la variable de type
    // booléen pour l'arrêt de l'énumération
    DirTraverser(wxListCtrl *m_lstResults, bool *stopFlag);
    // La méthode appelée lorsqu'un fichier est trouvé
    virtual wxDirTraverseResult OnFile(const wxString &filename);
    // La méthode appelée lorsqu'un dossier est trouvé
    virtual wxDirTraverseResult OnDir(const wxString &dirname);
private:
    // La méthode qui va faire tout le travail pour les deux ci-dessus
    wxDirTraverseResult AddItemToListCtrl(const wxString &item, const wxString &type);
    // Les variables pour stocker les pointeurs passés en paramètres au constructeur
    wxListCtrl *m_lstResults;
    bool *m_bStopFlag;
};
#endif
```

Grâce à cette méthode, le code des deux fonction **OnFile()** et **OnDir()** est énormément simplifié :

```
wxDirTraverseResult DirTraverser::OnFile(const wxString &filename)
{
    return AddItemToListCtrl(filename, _T("Fichier"));
}

wxDirTraverseResult DirTraverser::OnDir(const wxString &dirname)
{
    return AddItemToListCtrl(dirname, _T("Dossier"));
}
```

Et comme je vous l'ai dit, c'est la méthode **AddItemToListCtrl()** qui va tout faire.

D'abord, séparer le nom du fichier de son emplacement. Pour cela, on recherche le dernier caractère correspondant au séparateur de dossiers dans un chemin, à savoir « \ » pour Windows, et « / » pour les autres OS.

Et comme wxWidgets nous propose quelque chose de tout prêt pour obtenir ce caractère, on ne va pas se priver de s'en servir. Il est récupérable grâce à **wxFileName::GetPathSeparator()**. Il faudra donc penser à ajouter le header de **wxFileName** (<wx/fileName.h>) au début du fichier *dirtraverser.cpp* pour y avoir accès.

```
int i=item.Find(wxFileName::GetPathSeparator(), true);
wxString sPath=item.Left(i-1);
wxString sName=item.Right(item.Length()-i);
```

Ensuite, il faut ajouter l'élément à la liste des résultats, grâce à **wxListCtrl::InsertItem()**.

```
long itemIndex=m_lstResults->InsertItem(0, sName);
```

Puis mettre le texte correspondant à la deuxième colonne :

```
m_lstResults->SetItem(itemIndex, 1, type);
```

Puis placer le chemin dans la troisième colonne :

```
m_lstResults->SetItem(itemIndex, 2, sPath);
```

Il ne reste plus qu'à vérifier l'état du booléen pour renvoyer la bonne valeur, mais avant, il faut que l'on insère un petit appel à la fonction `wxYield()` pour permettre à l'interface de se mettre à jour :

```
wxYield();  
return (*m_bStopFlag==true)?wxDIR_STOP:wxDIR_CONTINUE;
```

Voici donc un récapitulatif du fichier `dirtraverser.cpp` :

```
#include "dirtraverser.h"  
#include <wx/filename.h>  
  
DirTraverser::DirTraverser(wxListCtrl *lstResults, bool *stopFlag)  
{  
    // On stocke les deux pointeurs passés en paramètres  
    m_lstResults=lstResults;  
    m_bStopFlag=stopFlag;  
}  
  
wxDirTraverseResult DirTraverser::OnFile(const wxString &filename)  
{  
    return AddItemToListCtrl(filename, _T("Fichier"));  
}  
  
wxDirTraverseResult DirTraverser::OnDir(const wxString &dirname)  
{  
    return AddItemToListCtrl(dirname, _T("Dossier"));  
}  
  
wxDirTraverseResult DirTraverser::AddItemToListCtrl(const wxString &item, const wxString &type)  
{  
    // On récupère la position du dernier séparateur  
    int i=item.Find(wxFileName::GetPathSeparator(), true);  
    // On sépare le nom et le chemin  
    wxString sPath=item.Left(i);  
    wxString sName=item.Right(item.Length()-i-1);  
    // On ajoute tout ça à la liste des résultats  
    long itemIndex=m_lstResults->InsertItem(0, sName);  
    m_lstResults->SetItem(itemIndex, 1, type);  
    m_lstResults->SetItem(itemIndex, 2, sPath);  
    // Pour permettre à l'interface de se rafraîchir  
    wxYield();  
    // Et on retourne la bonne valeur, en fonction du booléen  
    return (*m_bStopFlag==true) ? wxDIR_STOP : wxDIR_CONTINUE;  
}
```

Voilà, c'est terminé.

Vous pouvez compiler, et tester l'application : elle est fonctionnelle. Il reste bien entendu quelques petites choses à améliorer, mais ça fonctionne !

Pour information, deux petits bugs se sont glissés dans le code (ils ne sont pas très méchants, mais ils font que les résultats obtenus ne sont pas forcément ceux recherchés).

Quelques optimisations

Correction du 1er bug

Tout d'abord, je ne sais pas si vous l'avez remarqué, mais si vous lancez deux recherches sans fermer l'application, les résultats s'ajoutent dans la liste.

C'est le premier petit bug dont je vous parlais à la fin du chapitre précédent.

Cela vient du fait que nous n'avons jamais donné d'instruction pour vider cette liste au début de la recherche.

Il est très facile de corriger cela, en ajoutant une instruction juste avant de lancer cette recherche (fichier `mainframe.cpp`) :

```

.....
// On ré-initialise le booléen
bStopFlag=false;
// On vide la liste des résultats
m_lstResults->DeleteAllItems();
// Et on lance la recherche;
.....

```

Affichage du nombre de résultats

Deuxième petite chose : souvenez-vous, lors de la création de l'interface, nous avons gardé un pointeur vers la **wxStaticBox** entourant la liste des résultats, afin de pouvoir en changer le texte, et ainsi informer l'utilisateur du nombre d'éléments trouvés.

Justement, **wxDir::Traverse()** (http://www.wxwidgets.org/manuals/stable/wx_wkdir.html#wkdirtraverse) va nous donner en retour le nombre dont on a besoin.

Il suffit donc de le récupérer, et de changer le texte de la **wxStaticBox** (fichier *mainframe.cpp*):

```

// Et on lance la recherche
size_t count=dir.Traverse(traverser, m_txtName->GetValue(), wxDIR_DIRS|wxDIR_FILES);
// Quand on arrive ici, c'est que la recherche est terminée
// On peut donc remettre le label du bouton "Démarrer"
m_btnStart->SetLabel(_T("Démarrer"));
// ainsi que le texte de la wxStaticBox entourant la liste des résultats
m_stbNumRes->SetLabel(wxString::Format(_T("Résultats de la recherche : %0ld éléments trouvés"),
count));

```

Correction du 2ème bug

Ensuite, nous allons résoudre le deuxième petit bug de cette application.

En effet, lorsqu'on lance une recherche, tous les sous-dossiers du répertoire de départ sont ajoutés aux résultats de la recherche, alors qu'il ne devrait y avoir que ceux dont le nom correspond au masque de recherche.

Cela est dû au fait que la méthode **DirTraverser::OnDir(...)** est appelée pour chaque sous-dossier rencontré, contrairement à la méthode **DirTraverser::OnFile(...)** qui est appelée uniquement quand le nom du fichier correspond au critères de la recherche.

Rassurez-vous, ce comportement est tout à fait normal.

En effet, nous voulons que l'énumération se fasse dans tous les sous-dossiers du répertoire de base, et non uniquement dans ceux dont le nom correspond au masque fourni.

Qu'à cela ne tienne, nous allons légèrement modifier notre classe **DirTraverser** afin qu'elle puisse tester les noms des dossiers avant de les ajouter à la liste des résultats.

Tout d'abord, il faut que cette classe ait accès au masque de recherche. Nous allons tout simplement le lui passer en paramètre de construction, et le stocker dans une variable **wxString** privée.

Au moment de la recherche, comme le nom du dossier à tester est passé à la méthode **OnDir(...)** grâce à un **wxString**, et que cette classe possède une méthode **Matches** permettant de savoir si le **wxString** correspond à un masque de recherche, notre petit bug sera vite corrigé (il faudra quand même que l'on fasse l'extraction du nom du dossier car c'est son chemin complet qui est passé à la méthode **OnDir(...)**).

Voici donc, dans un premier temps, le code de la classe **DirTraverser** modifié :

Le fichier *dirtraverser.h*

```

#ifndef DIRTRAVERSER_H_INCLUDED
#define DIRTRAVERSER_H_INCLUDED

#include <wx/dir.h>
#include <wx/listctrl.h>
class DirTraverser : public wxDirTraverser
{
public:

```

```

// Le constructeur : on lui passe en paramètres le pointeur vers le wxListCtrl
// pour l'ajout des résultats, ainsi que le pointeur vers la variable de type
// booléen pour l'arrêt de l'énumération
DirTraverser(wxListCtrl *lstResults, const wxString mask, bool *stopFlag);
// La méthode appelée lorsqu'un fichier est trouvé
virtual wxDirTraverseResult OnFile(const wxString &filename);
// La méthode appelée lorsqu'un dossier est trouvé
virtual wxDirTraverseResult OnDir(const wxString &dirname);
private:
// La méthode qui va faire tout le travail pour les deux ci-dessus
wxDirTraverseResult AddItemToListCtrl(const wxString &item, const wxString &type);
// Les variables pour stocker les pointeurs passés en paramètres au constructeur
wxListCtrl *m_lstResults;
bool *m_bStopFlag;
// La variable pour stocker le masque de recherche
// (correction du premier bug de notre application)
wxString m_mask;
};
#endif

```

Le fichier *dirtraverser.cpp*

```

#include "dirtraverser.h"
#include <wx/filename.h>

DirTraverser::DirTraverser(wxListCtrl *lstResults, const wxString mask, bool *stopFlag)
{
// On stocke les deux pointeurs passés en paramètres
m_lstResults=lstResults;
m_bStopFlag=stopFlag;
// On stocke également le masque de recherche
m_mask=mask;
}

wxDirTraverseResult DirTraverser::OnFile(const wxString &filename)
{
return AddItemToListCtrl(filename, _T("Fichier"));
}

wxDirTraverseResult DirTraverser::OnDir(const wxString &dirname)
{
// On récupère le nom du dossier
int i=dirname.Find(wxFileName::GetPathSeparator(), true);
wxString sName=dirname.Right(dirname.Length()-i-1);
// On teste s'il correspond au masque de recherche
if (sName.Matches(m_mask))
{
// Si c'est le cas, on l'ajoute à la liste des résultats
return AddItemToListCtrl(dirname, _T("Dossier"));
} else {
// Sinon, on passe éventuellement à la suite
return (*m_bStopFlag==true)?wxDIR_STOP:wxDIR_CONTINUE;
}
}

wxDirTraverseResult DirTraverser::AddItemToListCtrl(const wxString &item, const wxString &type)
{
// On récupère la position du dernier séparateur
int i=item.Find(wxFileName::GetPathSeparator(), true);
// On sépare le nom et le chemin
wxString sPath=item.Left(i);
wxString sName=item.Right(item.Length()-i-1);
// On ajoute tout ça à la liste des résultats
long itemIndex=m_lstResults->InsertItem(0, sName);
m_lstResults->SetItem(itemIndex, 1, type);
m_lstResults->SetItem(itemIndex, 2, sPath);
// Pour permettre à l'interface de se rafraîchir
wxYield();
// Et on retourne la bonne valeur, en fonction du booléen

```

```
| return (*m_bStopFlag==true)?wxDIR_STOP:wxDIR_CONTINUE; |  
| } |
```

Il reste maintenant à modifier le code créant une instance de cette classe, dans le fichier *mainframe.cpp*, en ajoutant le masque de recherche aux paramètres de construction du **DirTraverser**.

Cela se passe dans la méthode **MainFrame::OnButtonStartClicked(...)** : il faut remplacer

```
| DirTraverser traverser(m_lstResults, &bStopFlag); |
```

par

```
| DirTraverser traverser(m_lstResults, m_txtName->GetValue(), &bStopFlag); |
```

Une fonction dépréciée

Il reste une dernière petite chose à régler : nous avons utilisé la fonction **wxYield()** pour permettre à notre application de rafraîchir la fenêtre si le besoin s'en fait sentir.

Or, cette fonction est depuis quelques versions déclarée comme « dépréciée ».

Il faut la remplacer par la méthode **wxApp::Yield()**.

Si vous vous penchez une dernière fois sur la documentation officielle pour **wxApp**, et plus précisément sur la partie concernant **wxApp::Yield()**, vous vous apercevrez que cette méthode n'est pas statique.

Il faut donc appeler la méthode **Yield()** membre de notre instance de classe d'application. Or, je ne sais pas si vous vous en souvenez, mais nous n'avons jamais créé de variable de type **FileFinderApp**.

Nous avons seulement utilisé la macro **IMPLEMENT_APP(FileFinderApp)** qui s'est elle-même chargée d'en créer une instance.

Malgré tout, les concepteurs de wxWidgets ont tout prévu. Dans le fichier header de la classe **FileFinderApp**, juste après sa déclaration, nous avons ajouté une autre macro : **DECLARE_APP(FileFinderApp)**.

Cette macro, comme je vous l'ai vaguement dit au début de ce cours, va se charger de créer une fonction **wxGetApp()** qui va nous renvoyer une référence vers l'instance de notre classe d'application en cours d'utilisation.

Il nous suffit donc d'inclure le header contenant cette macro (le fichier *filefinderapp.h*) quand nous en aurons besoin, et nous pourrions obtenir une variable de type **FileFinderApp&**.

La commande **wxYield()** que nous devons remplacer se trouve dans le fichier *dirtraverser.cpp*. Il faut donc ajouter au début de ce fichier :

```
| #include "filefinderapp.h" |
```

Ensuite, il suffit de remplacer la commande **wxYield()** (dans la méthode **DirTraverser::AddItemToListCtrl(.....)**) par les lignes ci-dessous :

```
| FileFinderApp& myapp=wxGetApp(); |  
| Myapp.Yield(); |
```

Nous pouvons même faire plus simple, car il est inutile de créer une variable juste pour cela :

```
| wxGetApp().Yield(); |
```

Et voilà, c'était tout simple.

A noter que la commande **wxGetApp()** peut aussi vous servir à appeler une méthode spécifique à votre classe d'application. Si par exemple, nous avons créé une méthode publique **FileFinderApp::GetApplicationPath()** pour récupérer le chemin de l'application (ce n'est qu'un exemple, ne le faites pas), nous pourrions l'appeler par **wxGetApp().GetApplicationPath()**.

Une dernière optimisation

La modification dont je vais vous faire part maintenant n'affecte pas un problème de fonctionnement de notre application. Malgré tout, il peut s'agir d'un petit problème dans certaines conditions.

Souvenez-vous : lors de la mise en place des contrôles, afin de réduire la taille du bouton "parcourir", nous lui avons donné une largeur fixe (30 pixels).

Le problème est que cette taille risque de ne pas être suffisante si vous utilisez une "grande police" de caractères.

Il existe une méthode alternative permettant de créer un bouton avec une taille qui s'adapte au texte à afficher.

Il s'agit tout simplement de lui affecter le style `wxBU_EXACTFIT`.

Le code de création du bouton devient donc (fichier `mainframe.cpp`) :

```
// Le bouton "Parcourir"
m_btnBrowse=new wxButton(this, ID_BTN_BRWSE, _T(" . . ."), wxDefaultPosition, wxDefaultSize,
wxBU_EXACTFIT);
```

Conclusion

Voilà : cette première partie est maintenant terminée.

J'espère qu'elle vous aura donné l'envie d'user et d'abuser de wxWidgets.

Il est vrai qu'à l'origine, j'avais l'intention d'ajouter des fonctionnalités à notre petit utilitaire, comme la possibilité de spécifier une chaîne de caractères à rechercher dans les fichiers, et bien d'autres encore, mais je pense que nous verrons cela dans une prochaine mise à jour de ce cours.

Vous pourrez retrouver le code source complet dans la rubrique « projets » de www.wxdev.fr.

Bien entendu, j'attends avec impatience vos remarques, et critiques sur ce cours.

De même, s'il y a un point précis de wxWidgets que vous souhaiteriez voir abordé, n'hésitez pas à m'en faire part, afin que je puisse éventuellement l'intégrer dans les prochaines parties du cours.

En attendant, bonne prog, et @ + sur www.wxdev.fr.

Xav'