Développer un IDE en C++

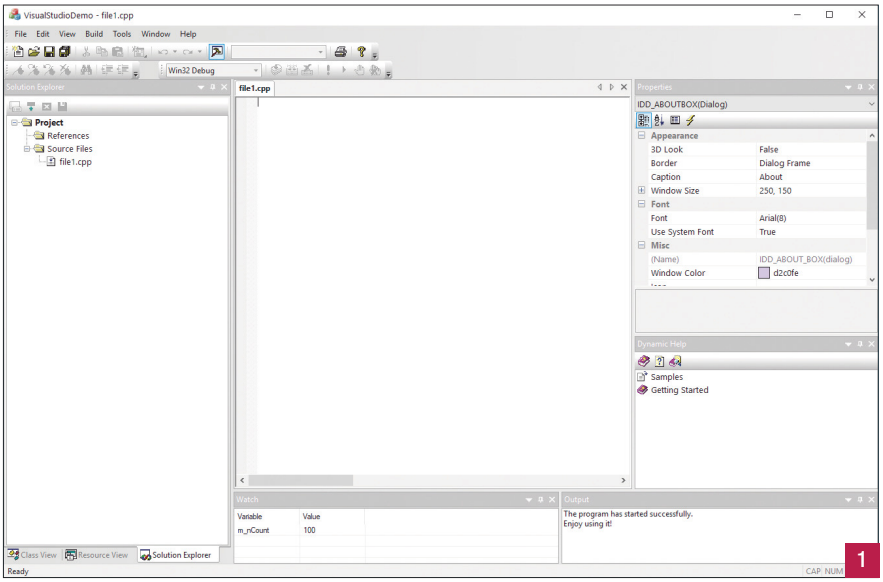
Partie 1

Christophe PICHAUD
Consultant sur les technologies Microsoft
christophepichaud@hotmail.com
www.windowscpp.net

NeOS-SDI makes IT work

Il n'y a pas si longtemps, la compilation se faisait uniquement en ligne de commandes. Et on lançait un (n)make makefile pour compiler un projet. On éditait le code source dans un éditeur rustique d'apparence mais très riche en fonctionnalités et c'était le standard. Tout était léger et vous allez me dire que les fonctionnalités étaient limitées… La vérité est entre les deux.

De nos jours, rien que pour prendre l'exemple de mon IDE préféré qu'est Visual Studio, l'installation offline pèse 30 Go… Le monde est fou. Vous voulez un petit challenge : codez vous-même un IDE et revenons aux fondamentaux. Moi, je me suis fixé un objectif : apprendre à ma fille à développer. Il y a quelques années, je lui avais fait un outil de dessin pour maîtriser la souris, maintenant, on passe au cran supérieur, faire des petits programmes pour apprendre.



1

Les modules importants de l'IDE

Avant tout, il faut un éditeur de texte qui colorie la syntaxe. On ne va pas se mettre dans le contexte de notepad ou tout est noir ou blanc ! On va prendre le cœur de Notepad++ qui se nomme Scintilla. C'est une librairie open-source très puissante. De plus, Scintilla sait parser tous les langages ou presque.

Compilation de la librairie Scintilla

La librairie Scintilla est disponible ici : http://www.scintilla.org. La version est 4.0.2. Le package zip (scintilla402.zip) à télécharger fait 1.6 MB. L'extraction des fichiers fait apparaître un dossier Win32 qui contient un makefile. On va donc compiler en ligne de commandes. Ouvrez un « Developer Command Prompt for VS 201x ». Se placer dans le dossier Win32 et lancez la commande nmake –f scintilla.mak.

```
D:\Dev\scintilla402\scintilla\win32>nmake -f scintilla.mak
```

Le résultat de la compilation est dans le répertoire bin. On y trouve SciLexer.dll et Scintilla.dll.

Intégration de Scintilla avec les MFC

Il existe une version « MFC » de Scintilla qui permet de gérer la zone de l'éditeur de texte comme un contrôle Windows et aussi la vue et le document qui sont des concepts purs MFC. C'est un MVP Visual C++ qui met cela à disposition sur http://www.naughter.com/scintilla.html. On y trouve des classes qui wrappent la librairie pour un emploi direct avec les MFC.

Obtenir un compilateur C#

Ensuite il faut un compilateur et là c'est très simple car le Framework .NET distribue en son sein le compilateur C# csc.exe. Attention, ce compilateur ne compile que jusqu'au C# 5. Il existe aussi Roslyn… Disponible en tant que package Nuget ici : https://dotnet.myget.org/feed/roslyn/package/nuget/Microsoft.Net.Compilers/3.0.0-dev-61717-03 ; il suffit d'extraire tout son contenu via un outil comme 7zip. A partir de là, on dispose de notre compilateur C# dernière version.

Le squelette de l'IDE

En cherchant un bout de code sur le Ribbon MFC, je suis tombé sur un exemple de code qui est bluffant ; il s'agit de Visual Studio Demo. C'est un sample MFC 2008 SP1 qui fournit un clone de Visual Studio version 2005/2008. Il y a les panneaux Classes, Ressources, Solution, Propriétés, etc. Regardons : [1]

Cette application est parfaite : on va juste supprimer quelques éléments visuels et le tour est joué. On va juste garder le panneau de gauche Solution Explorer et le panneau Output du bas.

Implémentation de fonctionnalités

La première chose à faire est d'insérer l'éditeur de code Scintilla dans l'application. On va donc faire les #include nécessaires dans le fichier d'entêtes précompilées qu'est stdafx.h :

```
#define USE_SCINTILLA
#define SCI_NAMESPACE
#ifdef USE_SCINTILLA
#include <platform.h>
#include <scintilla.h>
#include <SciLexer.h>
#endif
```

Maintenant, il faut charger la DLL au lancement de l'application :

```
BOOL CVisualStudioDemoApp::InitInstance()
{
#ifdef USE_SCINTILLA
  //Load the scintilla dll
  m_hSciDLL = LoadLibrary(_T("SciLexer.dll"));
  if (m_hSciDLL == NULL)
  {
      AfxMessageBox(_T("Scintilla DLL is not installed!"));
      return FALSE;
  }
#endif

  // ...
```

Maintenant, il faut créer la View sous forme de vue Scintilla. L'application est composée d'un seule type de fenêtre : la vue Scintilla ; donc dans la méthode InitInstance() de l'App, on y fait figurer cela :

```
  // Register the application's document templates.  Document templates
  //  serve as the connection between documents, frame windows and views.

  m_pDocTemplateCpp = new CMultiDocTemplate(IDR_DEVTYPE_CPP,
            RUNTIME_CLASS(CScintillaDemoDoc),
            RUNTIME_CLASS(CChildFrame),
               RUNTIME_CLASS(CScintillaDemoView));

  AddDocTemplate(m_pDocTemplateCpp);
```

Cette déclaration enregistre le couple MDI Doc/Vue. A chaque fois que l'on sélectionnera « New » dans le menu fichier, on aura un couple Doc/Vue. Le document contient les data de l'application. Présentée comme ça, l'affaire semble simple. Il faut cependant plonger dans les classes CScintillaDemoDoc et CScintillaDemoView pour bien comprendre ce qui se passe en arrière-plan.

La classe CScintillaDemoDoc et plus…

Cette classe hérite de CScintillaDoc. Dans cette classe, la méthode qui importe est celle qui fournit la sérialisation des données alias le Load & Save d'un fichier. C'est la méthode Serialize() qui fournit ce service.

```
void CScintillaDemoDoc::Serialize(CArchive& ar)
{
   CScintillaDoc::Serialize(ar);
}
```

La méthode délègue le service à la classe CScintillaDoc qui elle-même le délègue à la classe CScintillaView :

```
void CScintillaDoc::Serialize(CArchive& ar)
{
  CScintillaView* pView = GetView();
  AFXASSUME(pView != nullptr);

  pView->Serialize(ar);
}
```

La classe CScintillaView encapsule tout ça. Regardons la définition de la fonction Serialize :

```
void CScintillaView::Serialize(CArchive& ar)
{
  //Validate our parameters
  ASSERT_VALID(this);

  CScintillaCtrl& rCtrl = GetCtrl();
```

Voici l'envers du décor : la classe CScintillaCtrl. Elle encapsule le contrôle CScintilla :

```
class SCINTILLACTRL_EXT_CLASS CScintillaCtrl : public CWnd
{
public:
//Constructors / Destructors
  CScintillaCtrl();
  virtual ~CScintillaCtrl();
```

Cette classe CScintillaCtrl, importée depuis la DLL Scintilla, hérite de la classe CWnd des MFC qui représente une fenêtre Windows. A partir de là, l'application gère les fichiers et l'édition de code. C'est built-in ! Il reste cependant, quelques fonctionnalités à implémenter : charger une solution et avoir des fichiers à compiler.

La solution

L'application va gérer une solution avec un projet unique (pour le moment) qui contient des fichiers et des paramètres. A chaque fois qu'un fichier sera chargé, il sera ajouté à la solution. La création d'un fichier affiche une boîte de dialogue pour que le fichier soit créé et sauvegardé directement et être inséré immédiatement à la solution : [2]

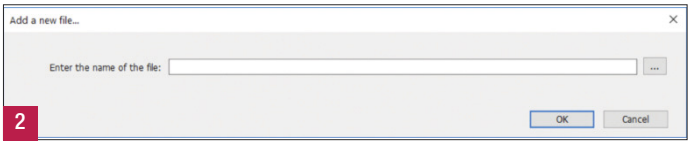
```
BOOL CScintillaDemoDoc::OnNewDocument()
{
   CFileNewDialog dlg;
   if (dlg.DoModal() == IDCANCEL)
      return FALSE;

   CString strFN = dlg.m_strFileName;

   LPTSTR lpszFileName;
   lpszFileName = PathFindFileName(strFN);

   CFile file((LPCTSTR)strFN, CFile::modeCreate);
   file.Close();

   if (PathFileExists((LPCTSTR)strFN) == FALSE)
   {
      AfxMessageBox(_T("The filename is not correct !"));
      return FALSE;
   }
```



2

```
    this->SetModifiedFlag(FALSE);

    std::shared_ptr<CCodeFile> cf = std::make_shared<CCodeFile>();
    cf->_name = lpszFileName;
    cf->_path = strFN;

    GetManager()->m_pSolution->AddFileToProject(cf);
    GetManager()->UpdateSolution(cf);

    if (!CDocument::OnNewDocument())
        return FALSE;

    return TRUE;x
}
```

Pour le moment, la solution est un simple fichier INI qui se présente sous cette forme :

```
[Solution]
Name=sol3
WorkingDir=d:\dev\net
CompilerPath=D:\Dev\UltraFluid Modeler\VisualStudioDemo\Bin\tools\CSC.exe
LastCompileCmd=D:\Dev\net\tools\CSC.exe /target:exe /out:d:\dev\net\Debug
\sol3.exe D:\Dev\net\main.cs D:\Dev\net\logger.cs
FileCount=2
File_1=D:\Dev\net\main.cs
File_2=D:\Dev\net\logger.cs
```

La gestion des paramètres se fait en utilisant le pop menu sur l'item Project dans le Solution Explorer : [3]

Une boîte de dialogue apparaît : [4]

Il est possible de changer les éléments minimums à la production d'une application. Pour le moment, les différentes options de compilations ne sont pas gérées : choisir Debug/Release, choisir Exe/Dll, etc. Patience, il y aura un deuxième article.

La compilation

La phase de compilation génère les actions suivantes :

- Créer la commande d'appel au compilateur Roslyn ;
- Faire un CreateProcess() pour lancer la commande et y rediriger les flux de sortie et les afficher dans le panneau Output Build.

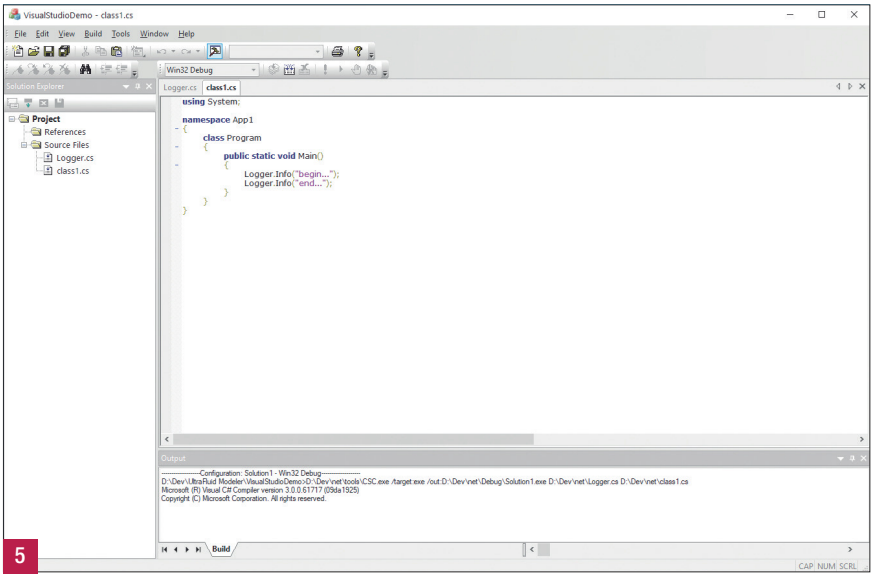
Voici la dernière ligne de commande qui a été générée :

```
D:\Dev\net\tools\CSC.exe /target:exe /out:D:\Dev\net\Debug\Solution1.exe
D:\Dev\net\Logger.cs D:\Dev\net\class1.cs.
```

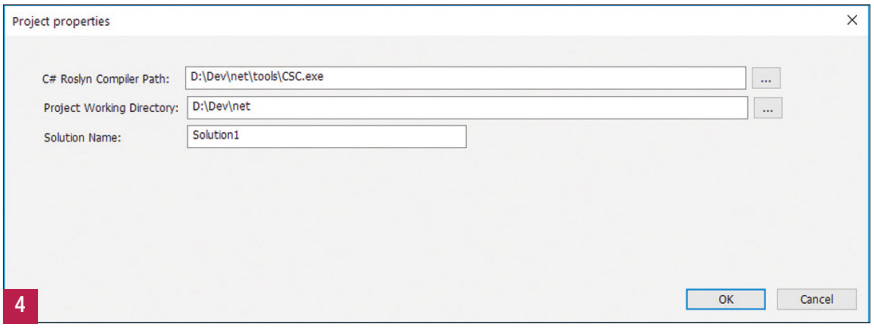
CSC.exe est le nom du compilateur Roslyn. Le flag target indique que l'on va créer un exe donc le nom est fourni au flag /out. Ensuite on trouve la liste des fichiers cs à compiler. [5]

CONCLUSION

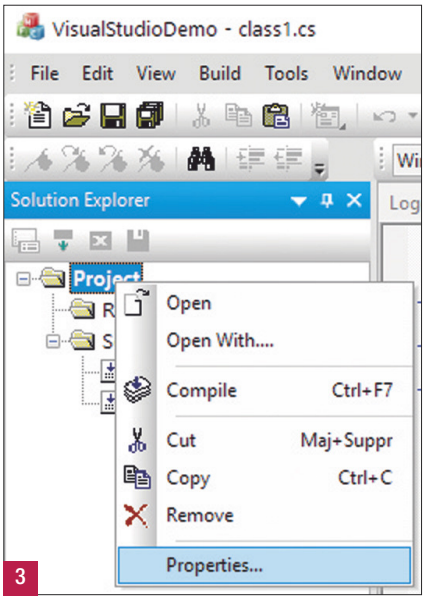
Dans cette première partie, on pose les bases d'un IDE, à savoir un éditeur à coloration syntaxique, un gestionnaire de fichiers et un compilateur. Toutes les briques sont en place pour y implémenter les fonctionnalités manquantes… Il va falloir gérer les références et les options du compilateur via le paramétrage du projet. Voilà, on se retrouve le mois prochain pour la suite de ce projet. •



5



4



3