Démarrer avec les MFC

Par Farscape

Version pour Visual 6.0

Introduction :

Pré requis :

Le niveau requis en C++ pour démarrer est plutôt moyen, il faut connaître les bases du langage.

Les MFC proposent un ensemble de classes couvrant l'ensemble des besoins pour développer une application Windows.

Elles fournissent des modèles d'architecture pour l'application sur lesquels celle-ci va être construite, à savoir le modèle document - vue décliné en trois options : applications **SDI** (single document interface : une seule fenêtre), **MDI** (multiple document interface : plusieurs fenêtres), et une application de type boite de dialogue.

L'ensemble des classes fenêtres est une encapsulation des contrôles Windows de base.
Les MFC utilisent l'héritage simple (pas d'héritage multiple), l'ensemble des classes forme une hiérarchie.

Ce graphique (sur le site de MSDN) donne une idée des classes disponibles :
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_mfc_hierarchy_chart.asp

Il montre bien les différents sujets couverts par les MFC:

Un ensemble de classes pour :

- ✓ L'architecture de l'application.
- ✓ La gestion des fenêtres
- ✓ La gestion des exceptions.
- ✓ La gestion des fichiers
- ✓ La gestion des dessins
- ✓ La gestion des menus
- ✓ Le support des bases de données.
- ✓ Les objets de synchronisation
- ✓ Les Sockets
- ✓ Les objets de stockage.
- ✓ Les services Internet.

Toutes les classes héritent de la classe **CObject** sauf les classes liées aux services Internet et quelques classes utilitaires.

Toutes les classes fenêtres ou contrôles héritent de la classe **CWnd** qui est la classe de base de toutes les fenêtres et comprend tous les traitements de base effectués sur une fenêtre : changement de titre, déplacement de la fenêtre etc..
Les noms des méthodes restent très proches de leur équivalent win32.

Microsoft Foundation Class Library Version 7.0

Conventions d'écritures pour les classes et variables :

Pour bien comprendre les classes MFC, il faut comprendre la manière dont les méthodes et variables sont nommées.
Les classes MFC utilisent la notation **Hongroise**.
C'est une convention qui s'applique à la création des noms des variables et des fonctions. Elle est utilisée par la majorité des programmeurs **Windows**, bien utilisée elle facilite la relecture et la maintenance des programmes. Cette notation est adoptée par Microsoft.

Un nom de variable est composé d'un court préfixe significatif placé devant un nom plus long, de préférence explicite. Le préfixe décrit le type de donnée référencée par la variable.

Dans certains cas, ce préfixe peut aussi décrire la façon dont la variable est utilisée. Le préfixe peut lui-même servir de nom de variable.

Voici quelques exemples d'emploi de la notation Hongroise :

```
char ch ;              // char
char achFile[128]  ;// tableau de caractères
Int    cbName ;

LPCTSTR lpszString // long pointeur constante string :
Int ich ;                  // index un tableau de caractères
```

Préfixe	Type de données
a	tableau (type composé)
ch	Caractère [char]
cb	Comptage d'octets
dw	Non signé long unsigned long (DWORD)
h	Handle (identificateur)
hdc	handle pour un périphérique.
hWnd	handle pour une fenêtre
I	Index (type composé)
d	Double
f	Float
L	Entier long [long]
Lp	Pointeur long (far)
n	Entier [int]
np	pointeur court (near)
sz	chaîne de caractère terminée par un zéro binaire.
w	entier non signé [unsigned int] (WORD)

On utilise aussi la notation Hongroise pour les noms de fonctions. La méthode la plus courante combine un verbe et un nom pour décrire une fonction. Exemple, dans Windows, on trouve trois routines qui s'appellent : **CreateWindow, DrawText, LoadIcon.** On peut trouver aussi un nom tout seul : **DialogBox.** Les routines qui convertissent un type en un autre sont souvent de la forme **XtoY, c**omme pour la fonction **DptoLP** qui convertit les coordonnées physiques en coordonnées logiques.

Pour finir, les variables membres d'une classe sont préfixées de **m**_ voulant dire membre.

Interactions avec l'éditeur :

Visual C++ n'est pas un RAD (rapid aid development), mais il génère néanmoins le squelette de l'application suivant le modèle choisi.

Il permet entre autres la génération de code lié à une nouvelle fenêtre, aux notifications de messages issus d'un contrôle, d'un menu etc..
Le dessin d'une interface à base de contrôles
L'éditeur dispose aussi d'assistants pour générer du code :

> **AppWizard** pour la génération d'un projet.
> **ClassWizard** pour la génération de fonctions en réponse aux messages de l'interface, la génération de classes etc..
> Sous Visual.net **ClassWizard** disparaît et ses différentes tâches sont réparties dans des fenêtres différentes …

L'apprentissage :

Apprendre à programmer avec les MFC nécessite de comprendre l'architecture générée par l'outil Visual et d'appliquer des méthodes référencées pour résoudre un problème d'architecture, comme par exemple comment mettre en place un splitter, avoir deux vues sur le même document, mettre en place une barre d'outils (toolbar) etc..

C'est sans doute cet aspect qui est le plus difficile à appréhender pour un débutant, car ces méthodes ne sont pas fournies directement par l'environnement, il faut les implémenter soit même.

Cet apprentissage se décompose en plusieurs parties :

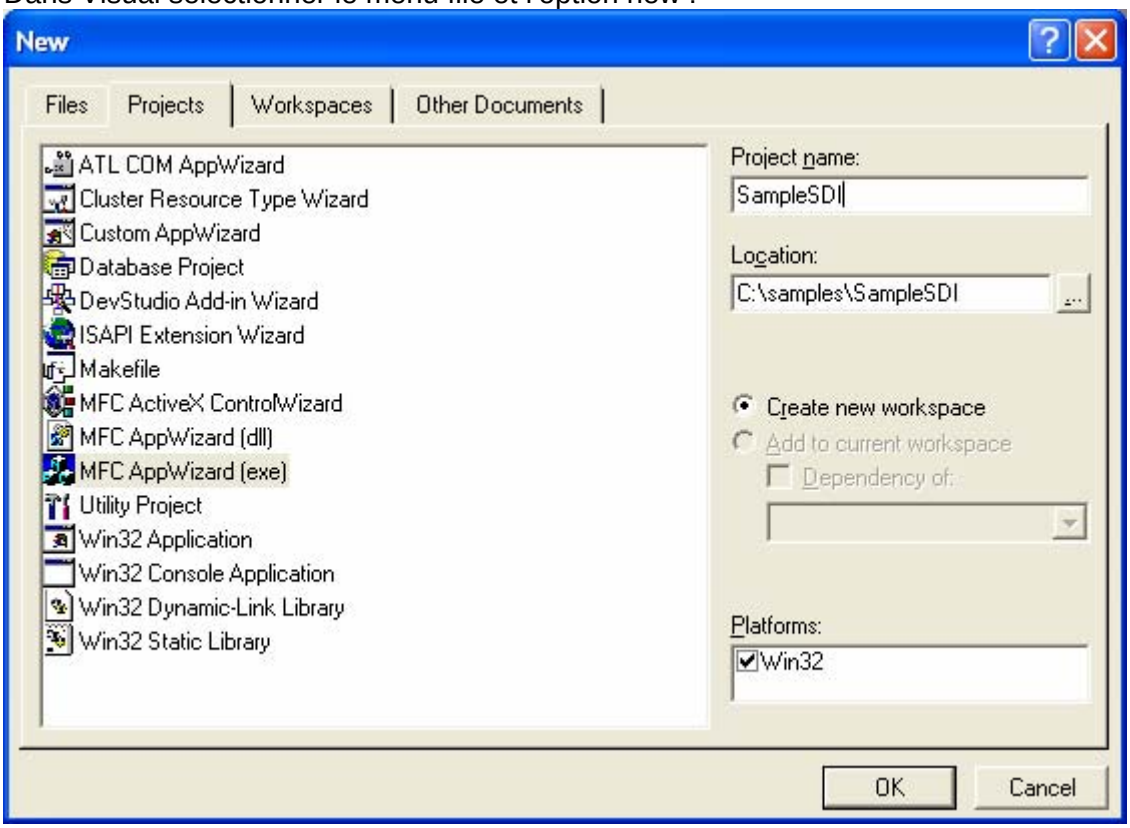
- Le FrameWork : tout ce qui touche à l'organisation des fenêtres.
- Les contrôles de bases edit ,static ,listbox etc.. , avec les différents traitements et personnalisations possibles.
- La persistance des données : gestion de fichier du plus simple à la base de données en passant par le mécanisme de sérialisation proposé par les MFC.
- Le GDI : les classes de dessin, bitmaps etc..

Nous allons effleurer l'essentiel pour démarrer une application.

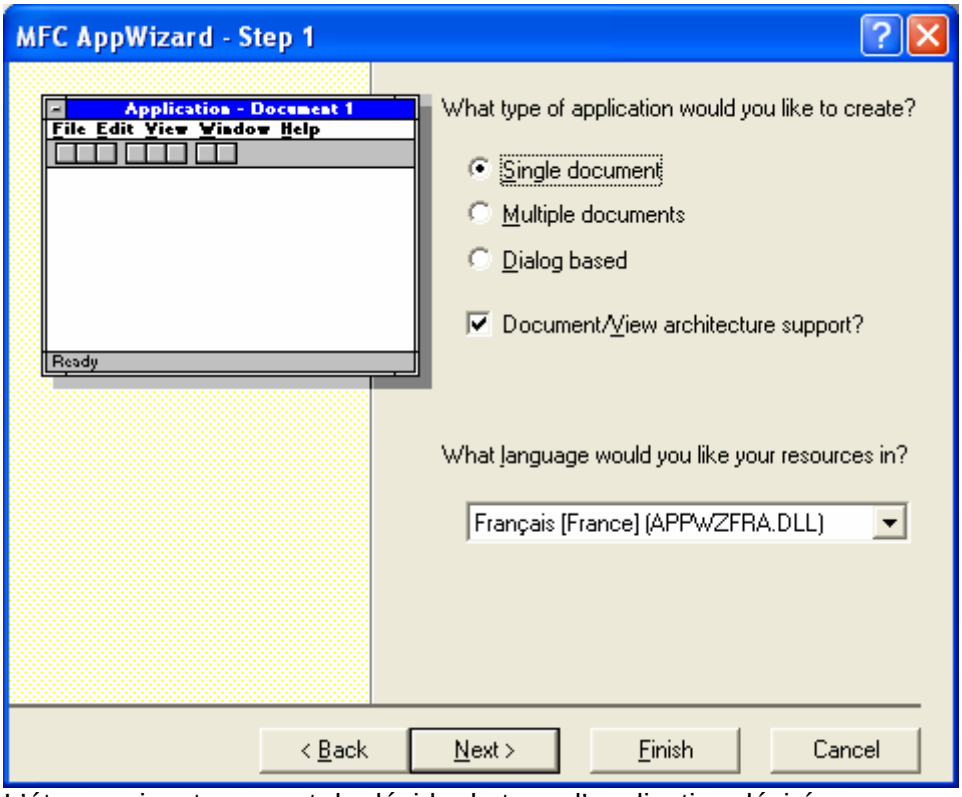
Le FrameWork première partie:

Génération d'un projet avec l'assistant création de projets :

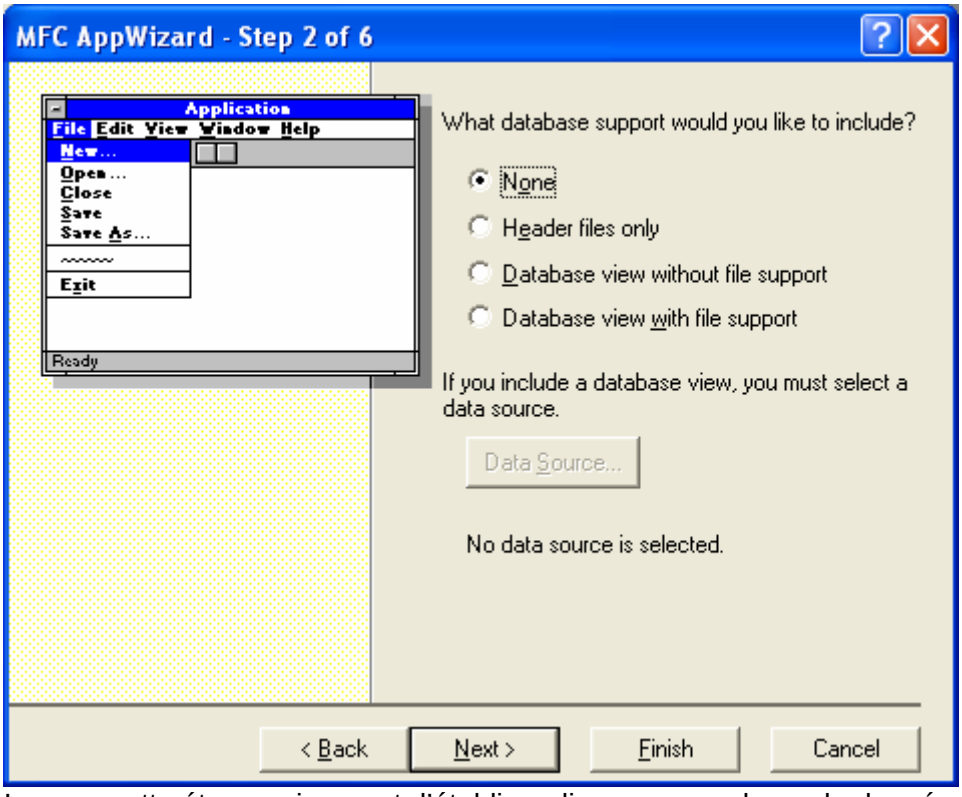
Pour commencer notre initiation nous allons générer une application de type SDI :
Dans Visual sélectionner le menu file et l'option new :



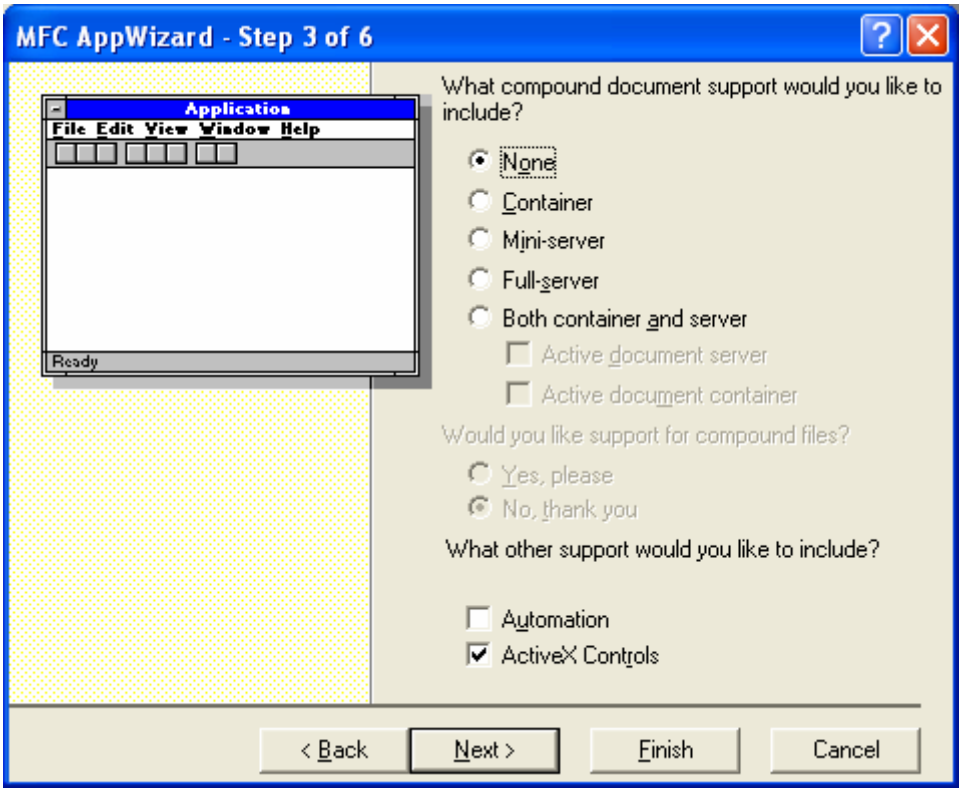
Sélectionnez la ligne **MFC AppWizad(exe)** et renseignez le nom du projet comme ci-dessus.



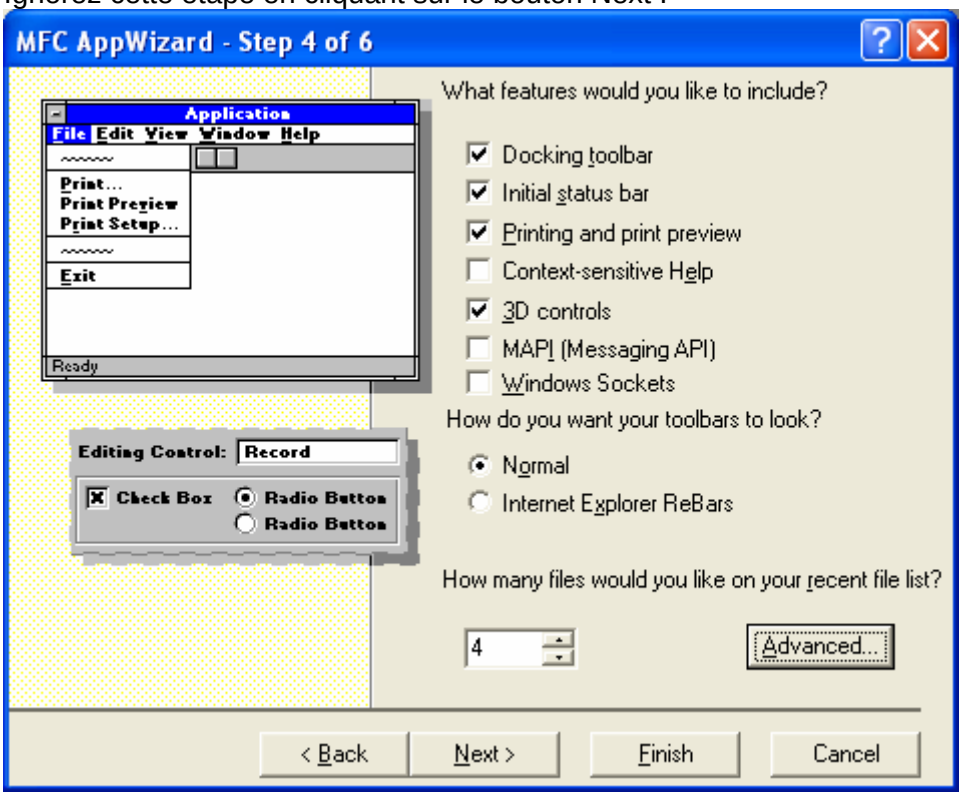
L'étape suivante permet de décider le type d'application désirée,
Sélectionnez l'option **single document** .
Puis cliquez sur le bouton Next.



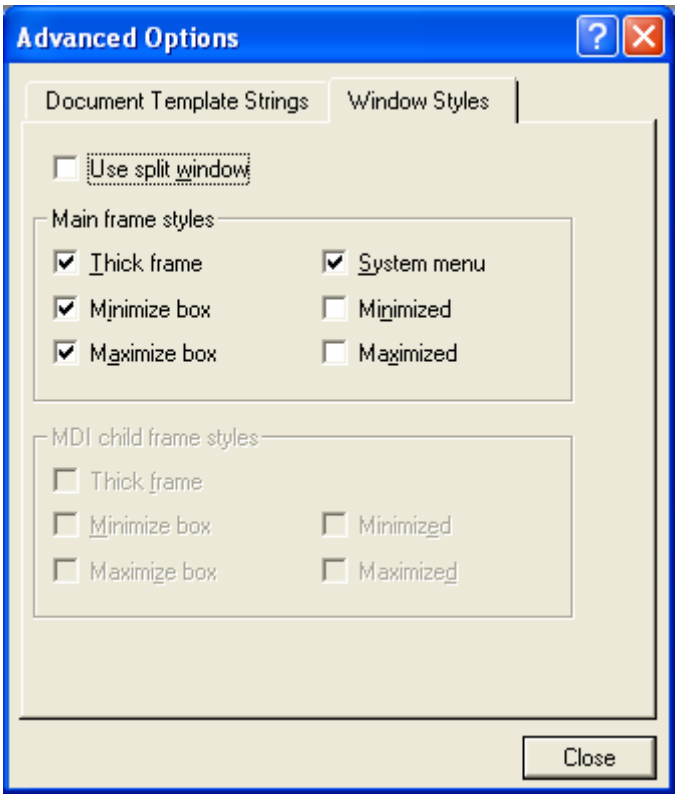
Ignorez cette étape qui permet d'établir un lien avec une base de données.
Pressez le bouton Next.



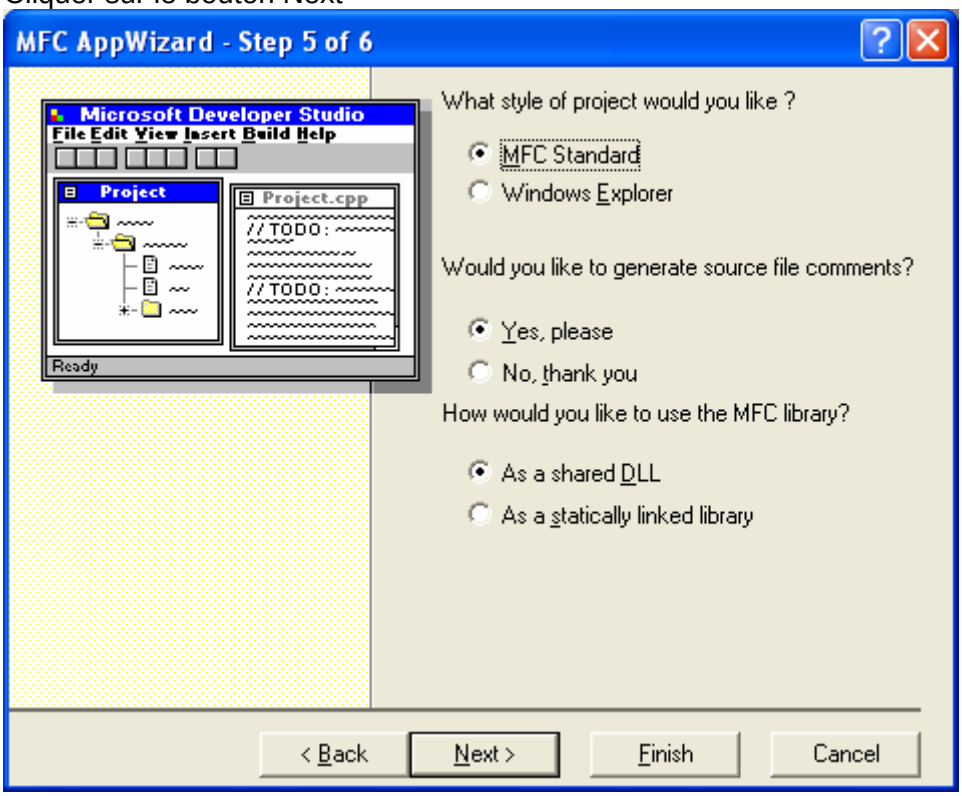
Cette étape concerne les liens avec l'interface OLE .
Ignorez cette étape en cliquant sur le bouton Next .



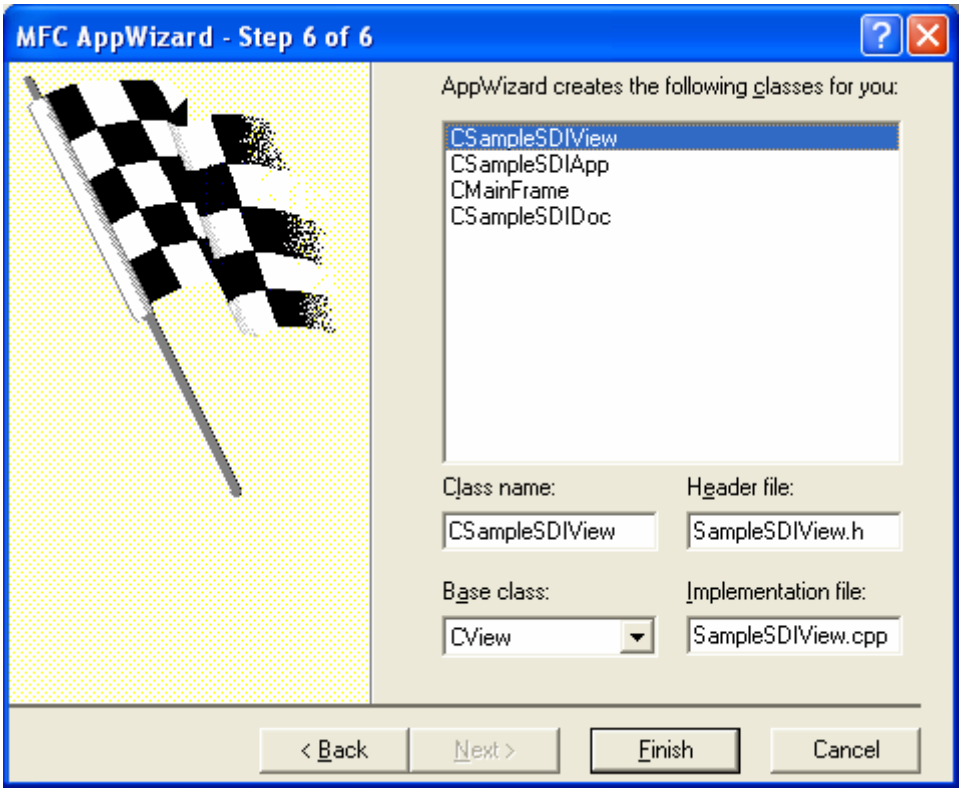
Cette étape permet de définir l'allure générale de l'application
Une option intéressante dans le bouton advanced



Dans l'onglet Windows Styles on peut spécifier que l'application dispose d'un splitter.
Pour l'instant laissez les options par défaut.
Cliquer sur le bouton Next

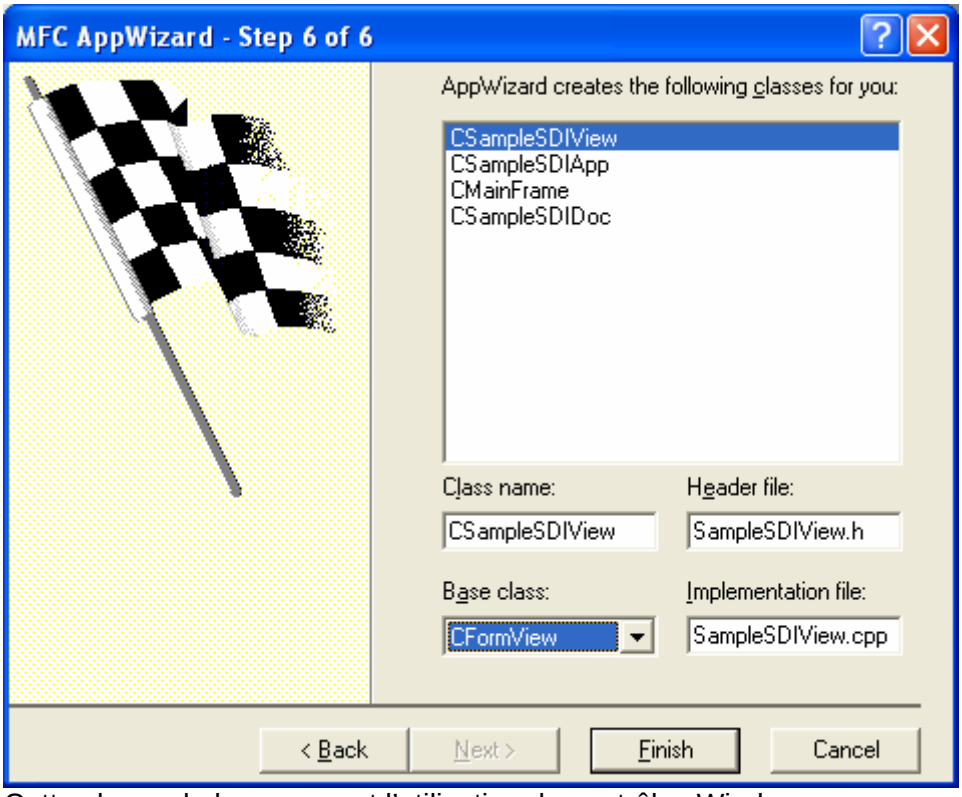


Toujours des paramétrages.
Laissez pour l'instant les paramètres par défaut
Cliquez sur le bouton Next.



Dernière étape :
Visual montre le nom des classes qui seront générées
Au total quatre classes :
Une classe fenêtre utilisateur **CSampleSDIView**
Une classe d'application **CSampleSDIApp**
Une classe fenêtre principale de l'application **CMainFrame**
Une classe Document **CSampleSDIDoc**

Vous remarquerez que à part la classe **CMainFrame** les classes principales utilisent le nom de l'application comme préfixe.
Revenons sur la classe **CSampleSDIView.** Nous allons sélectionner dans la boite de sélection (combobox) **Base class** la classe **CFormView** comme sur l'écran ci-dessous.

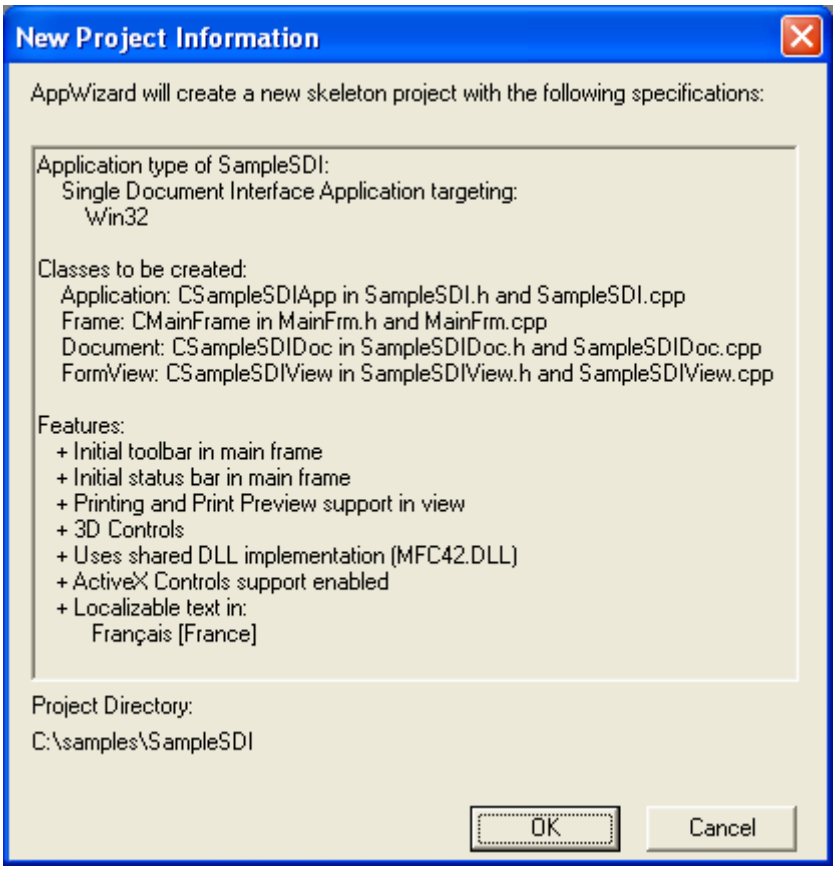


Cette classe de base permet l'utilisation de contrôles Windows sur son espace de travail. C'est donc à ce moment de la génération qu'il faudra choisir le type de fenêtre correspondant à l'interface souhaitée :

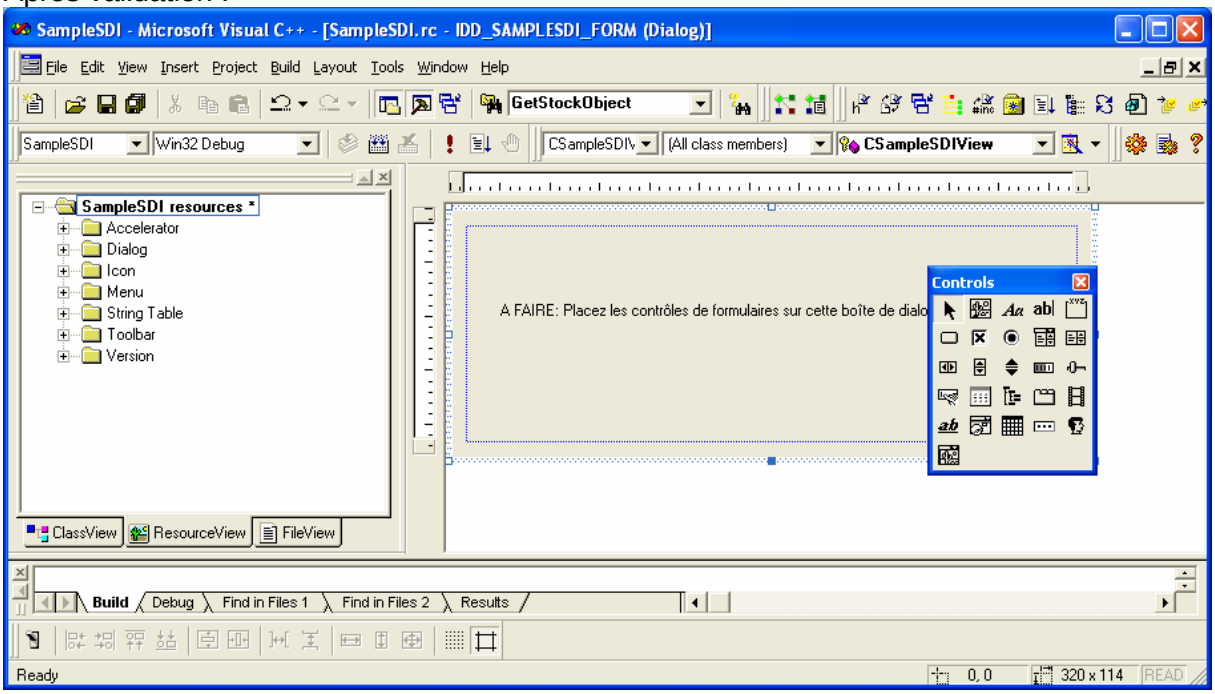
Classes Disponibles :

- ✓ **CEditView** : toute la fenêtre est une zone de saisie multi lignes.
- ✓ **CFormView** : fenêtre avec contrôles Windows.
- ✓ **CHtmlview** : fenêtre pour l'affichage de page html.
- ✓ **ClistView** : fenêtre sous forme de liste ; par exemple : l'explorateur Windows (partie droite de l'écran).
- ✓ **CRichEditView** : saisie de texte au format RTF comme wordpad.
- ✓ **CScrollView** : fenêtre de dessin avec une surface logique pouvant être supérieur à l'écran physique.
- ✓ **CTreeView** : fenêtre permettant d'avoir un arbre multi niveaux comme l'explorateur Windows (partie gauche de l'écran).
- ✓ **CView** : fenêtre de dessin simple.

Apres validation du choix Visual affiche un rapport de génération :

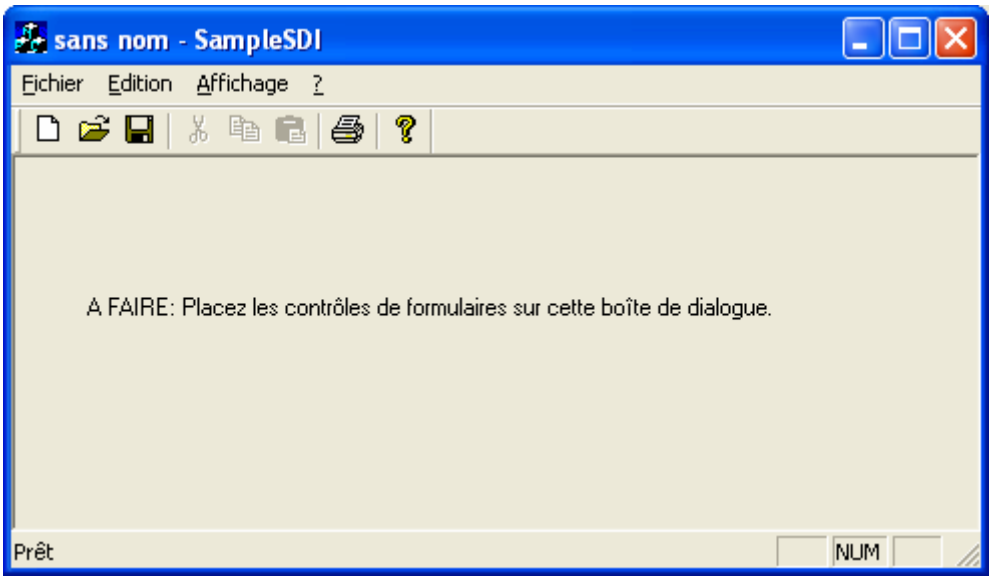


Après validation :



Visual se place directement sur le dessin de la fenêtre principale.
Avant de modifier et d'expliquer le contenu de ce qui a été généré, nous allons compiler et lier l'application. Pour cela utiliser le menu **build** et l'option **build sampleSDI.EXE** ou touche **F7 .**

Une fois ceci terminé, exécutez le programme menu **build execute** ou touche **CTRL+F5.**
Vous obtenez ceci :



L'application est composée des éléments suivants :

- Une barre de menu.
- Une barre d'outils ou Tools Bar (barre de boutons) : celle-ci peut être flottante
- Notre fenêtre de contrôles
- Une barre de statut qui affiche de manière standard l'état du clavier ainsi que les libellés d'aide lors de la sélection de menu ou lors du déplacement de la souris sur les boutons de la barre d'outils.

Etudions le code généré automatiquement :

La Classe d'application CWinApp :

Une application MFC comporte obligatoirement une classe d'application, dérivée de la classe de base **CWinApp**. Cette classe contrôle l'application, c'est elle qui va la démarrer avec la méthode **InitInstance**, mais aussi la terminer avec la méthode **ExitInstance**. Elle va mettre en place l'application grâce à l'unique objet instancié de cette classe. C'est aussi elle qui va router les différents messages aux différentes fenêtres de l'application. Elle offre en outre d'autres services comme la gestion des fichiers .ini.

Dans notre exemple les fichiers définissant cette classe se nomme **SampleSDI.H** et **SampleSDI.cpp** : le même nom que celui donné à l'application.

```
/////////////////////////////////////////////////////////////////////////////
// CSampleSDIApp:
// See SampleSDI.cpp for the implementation of this class
//

class CSampleSDIApp : public CWinApp
{
public:
	CSampleSDIApp();

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CSampleSDIApp)
	public:
	virtual BOOL InitInstance();
	//}}AFX_VIRTUAL

// Implementation
	//{{AFX_MSG(CSampleSDIApp)
	afx_msg void OnAppAbout();
		// NOTE - the ClassWizard will add and remove member functions here.
		//    DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

Ouvrez le source **CSampleSDiApp.cpp** :
Vous remarquerez le seul objet de la classe **CSampleSDiApp** qui va démarrer l'application :

```
// SampleSDI.cpp : Defines the class behaviors for the application.
//
#include "stdafx.h"
#include "SampleSDI.h"

#include "MainFrm.h"
#include "SampleSDIDoc.h"
#include "SampleSDIView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
```

```
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////////////
// CSampleSDIApp
BEGIN_MESSAGE_MAP(CSampleSDIApp, CWinApp)
	//{{AFX_MSG_MAP(CSampleSDIApp)
	ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
		// NOTE - the ClassWizard will add and remove mapping macros here.
		//    DO NOT EDIT what you see in these blocks of generated code!
	//}}AFX_MSG_MAP
	// Standard file based document commands
	ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
	ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
	// Standard print setup command
	ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CSampleSDIApp construction
CSampleSDIApp::CSampleSDIApp()
{
	// TODO: add construction code here,
	// Place all significant initialization in InitInstance
}
/////////////////////////////////////////////////////////////////////////////
// The one and only CSampleSDIApp object

CSampleSDIApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CSampleSDIApp initialization
BOOL CSampleSDIApp::InitInstance()
{
```

La fonction **InitInstance()** va initialiser l'application ,c'est donc la première fonction utile à l'application. On peut considérer en faisant le parallèle avec un programme DOS que c'est le main de l'application.

```
/////////////////////////////////////////////////////////////////////////////
// CSampleSDIApp initialization

BOOL CSampleSDIApp::InitInstance()
{
	AfxEnableControlContainer();

	// Standard initialization
	// If you are not using these features and wish to reduce the size
	//  of your final executable, you should remove from the following
	//  the specific initialization routines you do not need.
#ifdef _AFXDLL
	Enable3dControls();			// Call this when using MFC in a shared DLL
#else
	Enable3dControlsStatic();	// Call this when linking to MFC statically
#endif
	// Change the registry key under which our settings are stored.
	// TODO: You should modify this string to be something appropriate
	// such as the name of your company or organization.
	SetRegistryKey(_T("Local AppWizard-Generated Applications"));

	LoadStdProfileSettings();  // Load standard INI file options (including MRU)
```

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

	CSingleDocTemplate* pDocTemplate;
	pDocTemplate = new CSingleDocTemplate(
		IDR_MAINFRAME,
		RUNTIME_CLASS(CSampleSDIDoc),
		RUNTIME_CLASS(CMainFrame),	// main SDI frame window
		RUNTIME_CLASS(CSampleSDIView));
	AddDocTemplate(pDocTemplate);
	// Parse command line for standard shell commands, DDE, file open
	CCommandLineInfo cmdInfo;
	ParseCommandLine(cmdInfo);

	// Dispatch commands specified on the command line
	if (!ProcessShellCommand(cmdInfo))
		return FALSE;

	// The one and only window has been initialized, so show and update it.
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	return TRUE;
}
```

Une application Windows a la possibilité d'écrire des données dans un fichier ini qui porte son nom (héritage de Windows 3.xx). Depuis Windows 95 ces mêmes données peuvent être stockées dans la base de registre.

Pour utiliser ces fonctionnalités nous disposons de méthodes définies dans la classe d'application **CWinApp** permettant leurs écritures et lectures (par exemple **WriteProfileString** et **GetProfileString** pour une chaîne de caractères).

La ligne **SetRegistyKey** spécifie la clef qui sera définie dans la base de registre pour stocker ces données.
Si on veut stocker les données dans un fichier .ini plutôt de dans la base de registres, il suffira de supprimer cette ligne, rendant inopérant du même coup la ligne suivante qui permet la sauvegarde des derniers fichiers utilisés par l'application.

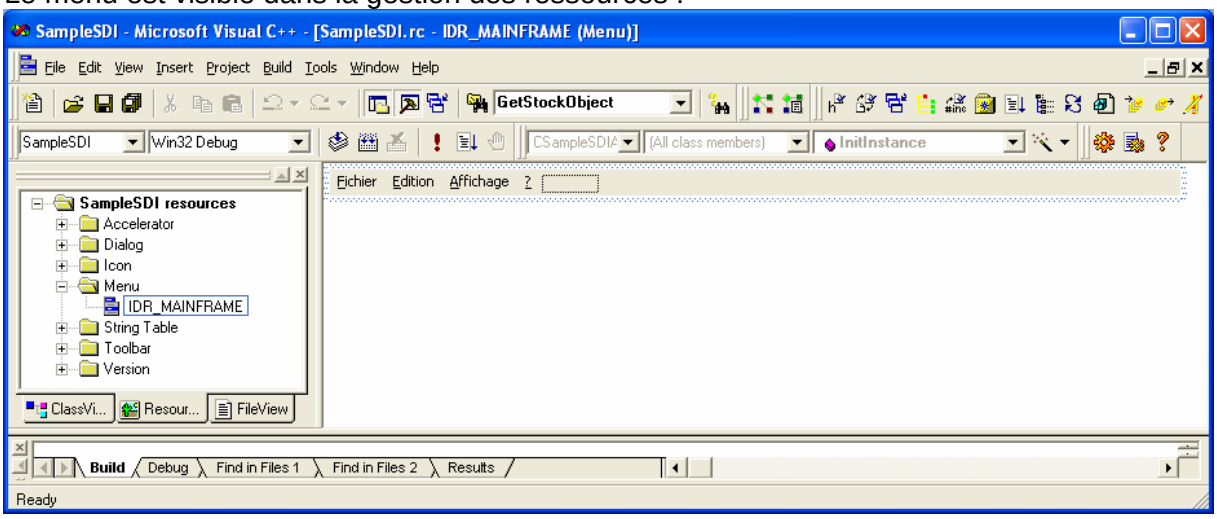
L'objet gestionnaire de fenêtre MFC :

Avec le bloc de lignes suivant on rentre dans le vif du sujet.
Il y est fait référence à l'initialisation d'un objet de la classe **CSingleDocTemplate** avec différentes classes entourées de la macro **RUNTIME_CLASS**.
En fait pour disposer d'une fenêtre de type Vue, il faut lui définir un environnement de travail qu'il faut renseigner au niveau de la classe d'application qui en sera le gestionnaire.

La définition d'une vue se décompose comme suit :

- Une fenêtre mère ici la classe : **CMainFrame** ,
- La classe fenêtre elle-même : **CSampleSDIView**
- D'une classe Document : **CSampleSDIDoc**
- D'un identifiant de menu : **IDR_MAINFRAME** correspondant au menu associé à la fenêtre.

Le menu est visible dans la gestion des ressources :

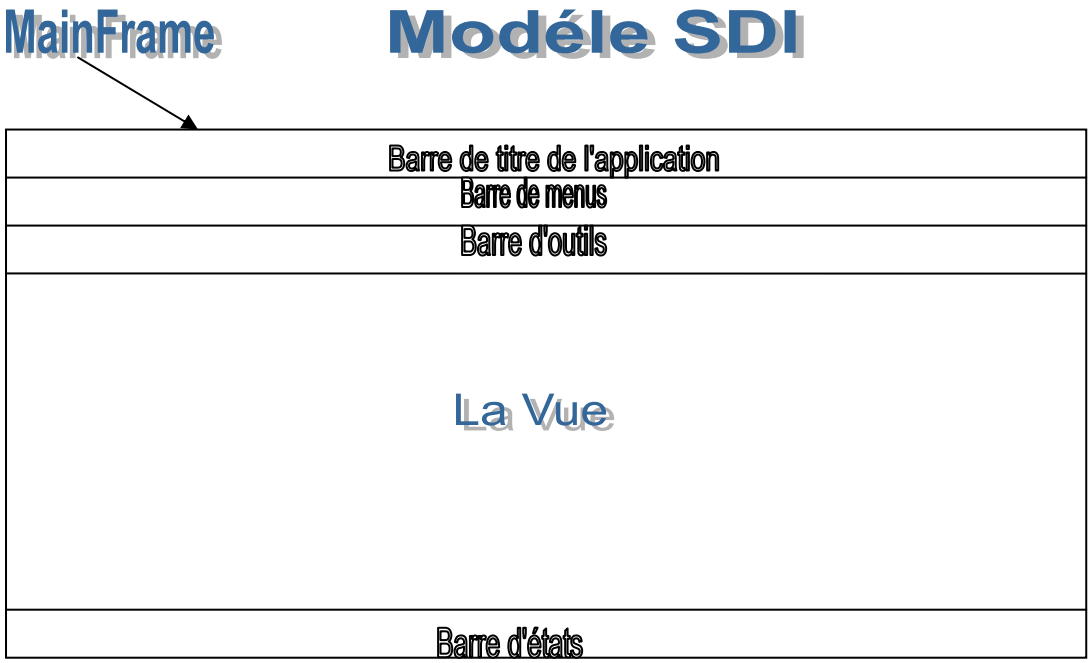


Une fois l'objet renseigné la fonction **AddocTemplate** est appelée avec l'objet décrivant l'unique fenêtre de notre application.

Revenons sur cette description, pourquoi une fenêtre mère ?

Les **MFC** décomposent la fenêtre en deux parties : le contexte de travail (fenêtre parent appelée la frame) qui comprendra les barres d'outils et la barre de status .

Dans le cas d'un projet SDI c'est la fenêtre principale **MainFrame** qui remplit ce rôle.
Dans le cas d'un projet MDI (que nous verrons plus tard) une autre classe apparaît la classe **CMDIChildWnd**, c'est elle qui jouera ce rôle, la deuxième partie étant la fenêtre elle-même.



La fenêtre vue est donc comprise dans la fenêtre principale établissant ainsi une hiérarchie parent / enfant.

L'architecture se charge de gérer les interactions entre la fenêtre principale et ses différentes fenêtres filles.

La classe Document :

Ce modèle de construction est appelé document-vue car pour chaque fenêtre vue on disposera d'un objet document.

Celui-ci n'a pas de partie graphique visible, il est destiné à enregistrer et lire les données relatives à la fenêtre.
Il est en relation directe avec la vue, celle-ci disposant d'un accès rapide pour y accéder la méthode **GetDocument()** .

La liaison document vue est effective dans les deux sens (vue document).
Bien que générée automatiquement il n'est pas indispensable de l'utiliser.
Pour ne pas trop compliquer tout de suite les choses je décrirai cette classe un peu plus loin dans la deuxième partie consacrée au **framework** pour le contexte **MDI** où elle prendra un peu plus d'importance pour des raisons d'architecture.

La macro RUNTIME_CLASS :

Autre point nébuleux et difficile à appréhender pour le débutant la macro **RUNTIME_CLASS**.
Cette macro permet d'obtenir la signature d'une classe.

En fait chaque classe dérivée de la classe **CObject** possède une structure **CRuntimeClass** qui peut être utilisée pour obtenir des informations en temps réel sur l'objet manipulé comme :

- Le nom de la classe contenue dans une chaîne de caractères.
- Une fonction **CreateObject( );** qui permet de créer un objet correspondant à la signature.

C'est justement cette fonctionnalité qui est utilisée en interne par l'architecture MFC pour créer les objets fenêtres. Vous remarquerez par la suite que pour les classes fenêtres générées par Visual le constructeur est protégé.

- Et pour finir la fonction **BOOL IsDerivedFrom(const CRuntimeClass* pBaseClass)** qui permet de savoir si la signature en cours hérite de la classe représentée par sa signature **pBaseClass.**
  Exemple d'écriture possible :

```
MyClass ::OneFunc(CWnd *pWnd)
{
   if(pWnd ->GetRuntimeClass()->IsDerivedFrom(RUNTIME_CLASS(CEdit)))
   {
     // cet objet est bien dérivé de la classe CEdit .
   }
}
```

Il est à noter que ce mécanisme est indépendant de celui du **RTTI du C++.**
Nous reviendrons plus loin sur ces fonctionnalités.

Nous avons vu les différentes initialisations dans la fonction **InitInstance** , mais rien qui montre comment est créée la fenêtre .

En fait c'est le groupe de ligne :

```
// Parse command line for standard shell commands, DDE, file open
        CCommandLineInfo cmdInfo;
        ParseCommandLine(cmdInfo);

        // Dispatch commands specified on the command line
        if (!ProcessShellCommand(cmdInfo))
                return FALSE;
```

qui s'en charge, et qui s'occupe de relayer les commandes par défaut du shell, permettant par exemple l'ouverture de l'application sur un double clic sur le fichier de données enregistré pour l'application.

Le nom sera utilisé par la classe document pour lire les informations.

Initialisation de la fenêtre principale de l'application CMainFrame :

Les deux lignes suivantes correspondent à la mise en place de la fenêtre principale. La première ligne rend visible la fenêtre et la deuxième permet son rafraîchissement.

```
        // The one and only window has been initialized, so show and update it.
        m_pMainWnd->ShowWindow(SW_SHOW);
        m_pMainWnd->UpdateWindow();
```

m_pMainWnd correspond donc à un pointeur sur la classe **CMainFrame** .

En fin de traitement la fonction **InitInstance** renvoie **true**, l'exécution de la boucle de messages au niveau de l'application est lancée par la fonction :

```
virtual int CWinApp::Run( );
```

Renvoyer FALSE abandonnera le lancement de l'application, faisant de cette fonction l'endroit idéal pour y gérer son accès ou son enregistrement.

Synthèse :

Une application **MFC** possède une classe d'application dérivée de la classe **CWinApp** .
Un objet unique de cette classe est instancié et permet par l'intermédiaire de sa méthode virtuelle **InitInstance** :

- d'initialiser l'application : couche réseau, objets olet (activex) etc..
- de définir les différentes vues disponibles dans l'application (une seule en SDI) .

La mise en place finale de la fenêtre principale correspondant à la classe **CMainFrame**.

Quelques données membres et fonctions utiles de la classe d'application :

::m_pszAppName	Contient le nom de l'application
::m_hInstance	Identifie l'instance de l'application
::m_lpCmdLine	Pointe vers une chaîne terminée par un '\0' qui contient la ligne de commande de l'application.
::m_nCmdShow	Spécifie comment la fenêtre va être initialisée.

Virtual int CWinApp::ExitInstance( )	Appelée lorsque l'application est fermée, à redéfinir pour faire un travail de libération mémoire par exemple.
void CWinApp::SetDialogBkColor(COLORREF clrCtlBk = RGB(192, 192, 192), COLORREF clrCtlText = RGB(0, 0, 0));	Permet de régler globalement pour les boîte de dialogue :la couleur de fond et/ou le texte dans les contrôles

Une fonction globale extrêmement utile :

```
CWinApp *AfxGetApp() ;
```

Cette fonction permet quelque soit l'endroit du programme d'obtenir un pointeur sur la classe d'application.

Exemple appliqué à notre projet d'exemple :

```
CSampleSDIApp *pTheApp=static_cast< CSampleSDIApp *>(AfxGetApp()) ;
```

Les fonctions de traitement des fichiers .ini :

```
CWinApp::GetProfileInt
UINT GetProfileInt( LPCTSTR lpszSection, LPCTSTR lpszEntry, int nDefault );

CWinApp::WriteProfileInt
BOOL WriteProfileInt( LPCTSTR lpszSection, LPCTSTR lpszEntry, int nValue );

CWinApp::GetProfileString
CString GetProfileString( LPCTSTR lpszSection, LPCTSTR lpszEntry, LPCTSTR
lpszDefault = NULL );

CWinApp::WriteProfileString
BOOL WriteProfileString( LPCTSTR lpszSection, LPCTSTR lpszEntry, LPCTSTR
lpszValue );

CWinApp::WriteProfileBinary
BOOL WriteProfileBinary(  LPCTSTR lpszSection,   LPCTSTR lpszEntry,   LPBYTE
pData,
   UINT nBytes );

CWinApp::GetProfileBinary
BOOL GetProfileBinary(LPCTSTR lpszSection,LPCTSTR lpszEntry,LPBYTE* ppData,
UINT* pBytes );
```

Ces fonctions permettent de stocker dans le .ini de l'application ou la base de registres des informations utilisateur.

Dans le cas d'un .ini celui-ci est stocké dans le répertoire Windows et porte le nom de l'application.

Les fonctions de lecture admettent un argument pour spécifier une valeur par défaut quand la clef n'existe pas dans le .ini.

Exemples :

```
// ecriture dans le fichier ini
AfxGetApp()->WriteProfileString("Parametres", "UserDefault", "Farscape");
// lecture dans le fichier ini
CString str = AfxGetApp()->GetProfileString("Parametres", "UserDefault") ;
AfxMessageBox(str) ;
```

Donnera le résultat suivant :

```
[Parametres]
UserDefault=Farscape
```

La classe Fenêtre Principale CMainFrame

Continuons notre étude de code avec la classe **CMainFrame** .

Rappels :
Visual C++ a généré pour notre exemple une classe d'application **CSampleSDIApp** dérivée de **CWinApp.**
Une classe fenêtre principale de l'application **CMainFrame** dérivée de la classe **CFrameWnd**
Voici sa définition :

```
class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
        CMainFrame();
        DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:
        // Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CMainFrame)
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        //}}AFX_VIRTUAL
// Implementation
public:
        virtual ~CMainFrame();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:  // control bar embedded members
        CStatusBar  m_wndStatusBar;
        CToolBar    m_wndToolBar;

// Generated message map functions
protected:
        //{{AFX_MSG(CMainFrame)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code!
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

Ce code assez court demande beaucoup d'explications…

Le rôle de cette classe c'est d'être responsable de la fenêtre principale de l'application. Elle va mettre en place lors de sa création deux éléments graphiques :

- La barre de status représenté par la classe **CStatusBar**.
- La barre d'outils représenté par la classe **CToolBar** .

Son rôle serra plus étoffé lorsqu'on parlera de projet **MDI**.

Comme c'est un élément important de l'application, elle dispose elle aussi d'une fonction globale permettant d'y accéder quelque soit l'endroit du programme

```
CWnd* AfxGetMainWnd( );
```

Exemple appliqué à notre projet d'exemple :

```
CMainFrame *pTheFrame=static_cast< CMainFrame *>( AfxGetMainWnd()) ;
```

Le traitement des Messages Windows :

Il nous faut maintenant parler de la gestion des messages et la manière dont ils sont implémentés dans le code.

Toutes les classes fenêtres héritent de la classe CWnd qui dispose d'une fonction particulière la « windows procedure » (**CWnd ::WindowProc**)

Pour chaque message Windows on aura une fonction commençant par **On…** générée.
Dans notre exemple le message **WM_CREATE** a généré la fonction :

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

Une boucle des messages (Message Map) implémentée sous forme de macro maintient la liste des fonctions en réponse aux messages de l'interface utilisateur définie, c'est-à-dire les commandes issues des menus, des boutons etc..

L'utilisateur pourra aussi disposer de messages privés (cette notion sera développée plus loin).

A la réception d'un message en destination de la fenêtre, la fonction correspondante est appelée (ou éventuellement celle de la classe de base si aucune définition n'est proposée).

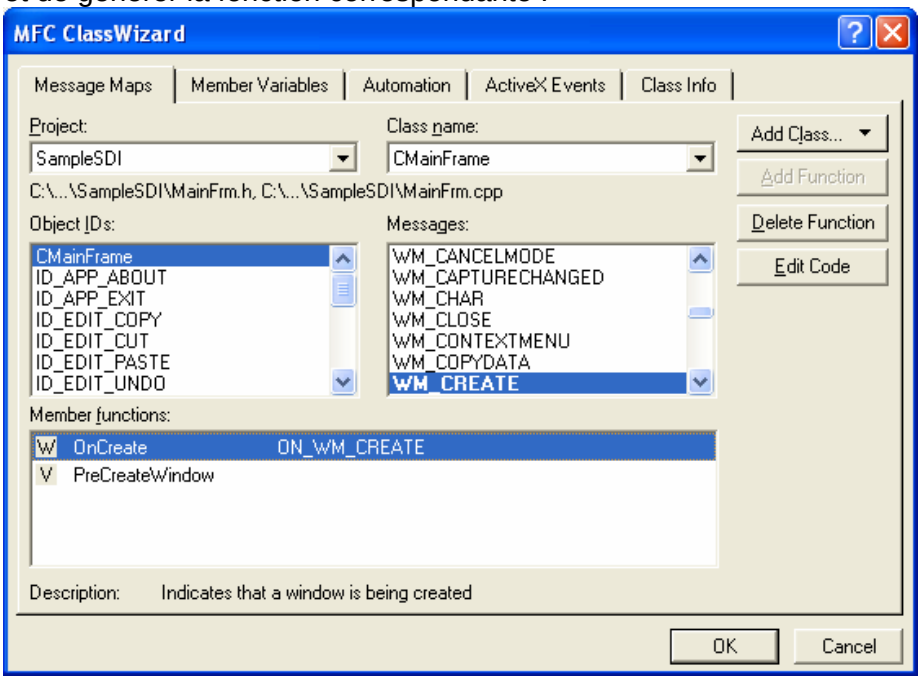
Une question classique :
Pourquoi ne pas avoir implémenté toutes les fonctions de réponses aux messages en méthodes virtuelles ?

Plusieurs raisons :
La liste des messages Windows n'est pas finie et l'utilisateur peut définir ses propres messages.

La deuxième, définir un nombre trop important de méthode virtuelle gaspillerait des ressources considérables, le compilateur devant maintenir dans une table (vtable) les pointeurs sur ces fonctions.

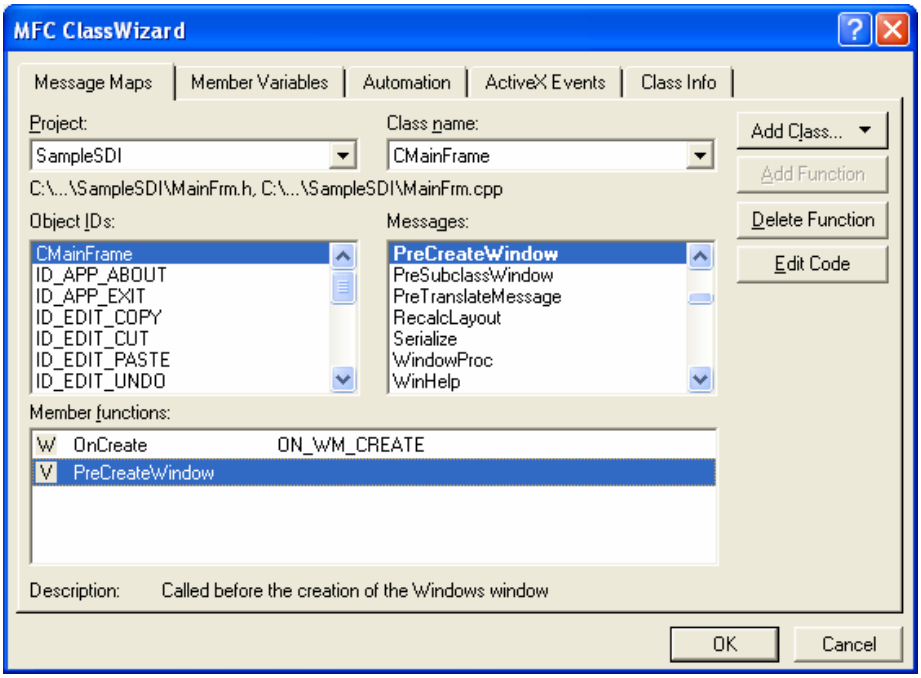
Ajouter un message avec l'environnement Visual :

Heureusement pour nous ce travail de génération de code est pris en charge par l'environnement qui permet pour une classe donnée de visualiser les messages disponibles et de générer la fonction correspondante :



Dans la partie gauche on voit bien la classe sélectionnée et dans la partie droite les différents messages commençant par **WM_** (Windows message)
Il suffira donc de sélectionner le message dans la liste et de cliquer sur le bouton **Add Function** pour générer le code de la fonction de réponse au message.

ClassWizard permet aussi de redéfinir des fonctions virtuelles définies dans la classe de base :



Une autre question :
Comment Visual s'y retrouve avec ces fonctions générées dans le code ?

Il insère tout simplement des commentaires permettant de repérer les blocs en question. On aura un bloc correspondant aux fonctions virtuelles redéfinies par l'utilisateur avec le commentaire **AFX_VIRTUAL** :

```
// Operations
public:
        // Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CMainFrame)
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        //}}AFX_VIRTUAL
```

Pour les messages on aura le commentaire **AFX_MSG**

```
// Generated message map functions
protected:
        //{{AFX_MSG(CMainFrame)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code!
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
```

La présence de la macro **DECLARE_MESSAGE_MAP()** est obligatoire pour la déclaration interne de cette table de message.

Toutes les fonctions de réponses aux messages sont préfixées du mot réservé **«afx_msg»**

Il est fortement recommandé de ne pas toucher aux commentaires ajoutés par Visual, sinon vous risquerez de ne plus pouvoir utiliser **ClassWizard** sur cette classe.

Dans le cas d'un ajout manuel d'une fonction ou message il est plus prudent de la rajouter après le bloc de commentaire.

Passons maintenant à la partie implémentation de ces fonctions :

```
/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
	//{{AFX_MSG_MAP(CMainFrame)
		// NOTE - the ClassWizard will add and remove mapping macros here.
		//    DO NOT EDIT what you see in these blocks of generated code !
	ON_WM_CREATE()
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Dans l'implémentation de la classe Visual a généré le bloc de réponse aux messages (message map).

Il est délimité par les macros **BEGIN_MESSAGE_MAP** et **END_MESSAGE_MAP.**

La première ligne **IMPLEMENT_DYNCREATE** avec le nom de la classe et le nom de la classe de base est nécessaire pour la création dynamique d'objet par les MFC et son système de reconnaissance de signature (sujet évoqué plus haut).

Dans la macro **BEGIN_MESSAGE_MAP** il est précisé le nom de la classe où s'appliquent les messages et le nom de la classe parent.

Attention une erreur classique : en cas d'héritage successif d'une classe fenêtre il faut bien notifier le bon parent pour chaque classe dérivée, sinon les messages destinés à la classe parent ne seront pas traités.

Un message rajouté manuellement (message privé) doit être placé juste au dessus de **END_MESSAGE_MAP().**

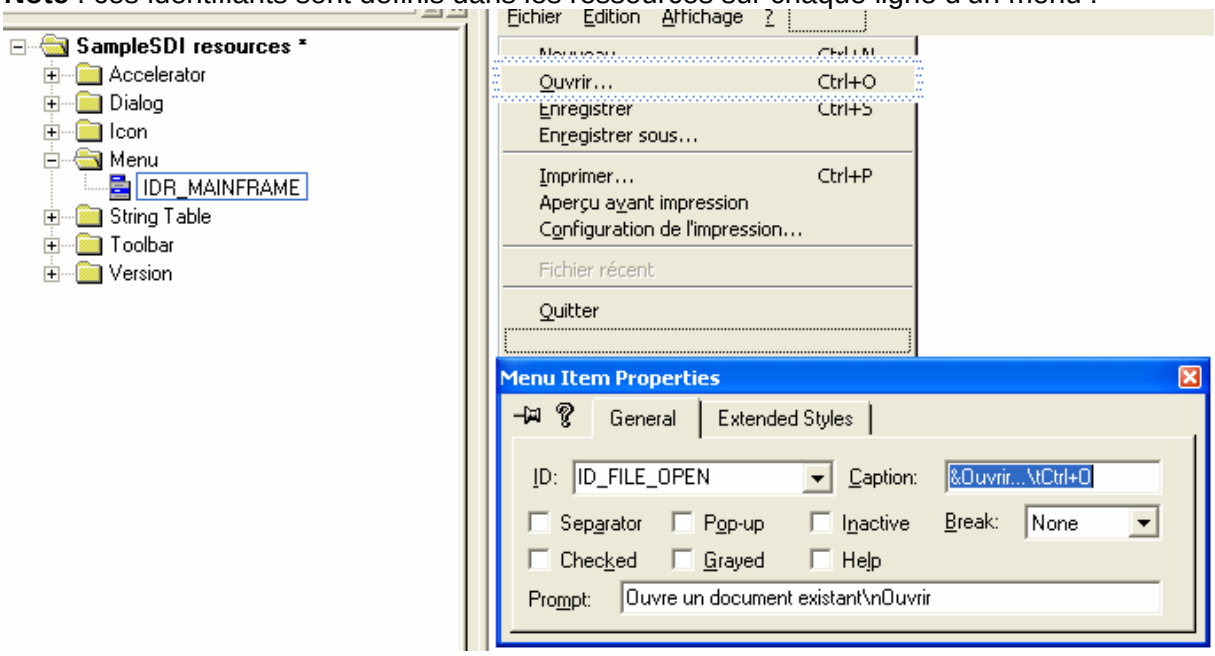
Les différentes catégories de messages :

- Les messages Windows :
  L'identifiant commence par **ON_WM_**
  Voir documentation MSDN pour une liste exhaustive.

- Les Commandes du menu ou de barre d'outils (ToolBar):
  L'identifiant commence par **ON_COMMAND(Identifiant,NomFonction)**
  Exemple :

```
////////////////////////////////////////////////////////////////////////////
// CSampleSDIApp

BEGIN_MESSAGE_MAP(CSampleSDIApp, CWinApp)
	//{{AFX_MSG_MAP(CSampleSDIApp)
	ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
		// NOTE - the ClassWizard will add and remove mapping macros here.
		//    DO NOT EDIT what you see in these blocks of generated code!
	//}}AFX_MSG_MAP
	// Standard file based document commands
	ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
	ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
	// Standard print setup command
	ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

Note : ces identifiants sont définis dans les ressources sur chaque ligne d'un menu :



- Les Messages privés :
  Ce sont des messages que vous définissez pour vos propres besoins,
  ils sont envoyés par l'intermédiaire des fonctions :

```
CWnd::PostMessage

BOOL PostMessage(
UINT message,
WPARAM wParam = 0,
LPARAM lParam = 0 );
```

qui place le message dans la file d'attente en cours pour la fenêtre concernée. Non bloquante, la fonction rend immédiatement la main.

```
CWnd::SendMessage
LRESULT SendMessage( UINT message, WPARAM wParam = 0, LPARAM lParam = 0 );
```

Envoie immédiat avec attente de réponse, donc bloquant

Dans le message map on aura **ON_MESSAGE(WM_USER+100 ,NomFonction)**
WM_USER étant le define de départ pour les messages utilisateurs.
La fonction de réponse aura le prototype suivant :

```
LPARAM CMyWnd ::NomFonction(WPARAM wp,LPARAM lP)
{
  return 0L;
}
```

- Les messages émanant des contrôles:
  Il est possible d'intercepter au niveau de la classe fenêtre contenant les contrôles certaines notifications.
  Nous verrons cette possibilité sur le chapitre consacré aux contrôles.

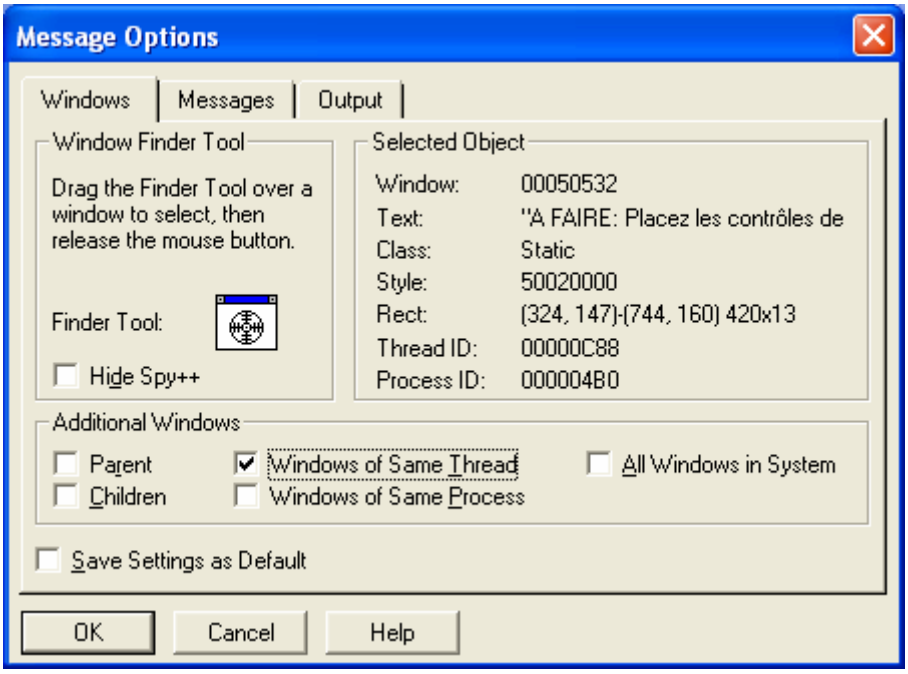
Les Classes de fenêtres :

Chaque fenêtre sait se redessiner, mais qu'est ce qui différencie un contrôle de type edit et un contrôle de type static ?
Réponse : sa classe de fenêtre, chaque fenêtre possède une classe permettant de définir son affichage et son comportement.
Exemples : un contrôle Edit aura pour classe … « Edit » un contrôle static « static »

Spy l'espion qui vous veut du bien :

Il existe un utilitaire qui permet de visualiser et de se rendre compte des propriétés des fenêtres, des échanges etc. ..

C'est **Spy** ,il est livré avec **visual**



Après lancement du programme sélectionnez l'option du menu **spy** : **messages**.
Placez la souris sur la zone en face du libellé Finder Tool, puis maintenez le bouton gauche dessus et déplacez la souris sur les différentes parties de notre programme de test, par exemple sur le libellé **static** de notre fenêtre, et lâchez le clic gauche dessus. Vous obtenez le résultat visible sur la capture ci-dessus.

On retrouve bien tout ce qui été décrit a savoir le handle de fenêtre, le texte, la classe de fenêtre, son style sa positon etc .

Après validation de cette boîte de dialogue, **spy** ouvre une fenêtre qui trace tous les messages reçus par la fenêtre sélectionnée.

Exercez-vous avec SPY qui est un bon outil pour comprendre comment se déroule les choses, les messages envoyés suite à une action etc …
En gros espionner l'application …

Les messages explications complémentaires:

Le fonctionnement des fenêtres est basé sur l'échange de messages,

- Ainsi nous aurons des messages en provenance de la fenêtre pour nous avertir d'un changement.
  Quelques messages usuels et importants :

  - **WM_PAINT** : il est envoyé chaque fois qu'une fenêtre a besoin d'être redessinée suite à un recouvrement par une autre fenêtre par exemple ou par une action volontaire,
  - **WM_SIZE :** il est envoyé quand la fenêtre change de taille par une demande manuelle de l'utilisateur ou par demande spécifique.
  - **WM_CREATE :** il signale que la fenêtre va être créée
  - **WM_CLOSE :** il signale que la fenêtre va être fermée.

  Il m'est impossible de décrire la liste des messages, elle est trop importante.
  Il suffit de savoir que le système envoie des messages concernant la fenêtre, la gestion du clavier, la gestion de la souris etc..

  Pour le reste c'est un apprentissage progressif en fonction de ses besoins et on se reportera à la documentation MSDN pour le détail de fonctionnement.

- Nous enverrons des messages spécifiques pour changer les propriétés d'une fenêtre.

- Ou encore nous interrogerons une fenêtre pour récupérer un état ou un status.

  Les deux dernières possibilités sont encapsulées par les classes MFC :

  Exemple :
  Communication avec le contrôle de la classe **CListCtrl** :

  Affectation d'une valeur :

```
_AFXCMN_INLINE void CListCtrl::SetWorkAreas(int nWorkAreas, LPRECT
lpRect)
        { ASSERT(::IsWindow(m_hWnd)); ::SendMessage(m_hWnd,
LVM_SETWORKAREAS, nWorkAreas, (LPARAM) lpRect); }
```

  Récupération d'une valeur :

```
_AFXCMN_INLINE DWORD CListCtrl::GetExtendedStyle()
        { ASSERT(::IsWindow(m_hWnd)); return (DWORD)
::SendMessage(m_hWnd, LVM_GETEXTENDEDLISTVIEWSTYLE, 0, 0); }La
classe vue de l'application  CSampleSDIView
```

  Vous remarquerez dans les deux cas de l'utilisation de la fonction **SendMessage**.
  De la variable **m_hWnd** qui est une donnée membre de la classe **CWnd** et qui représente son handle (identifiant visible avec **spy**) de fenêtre.

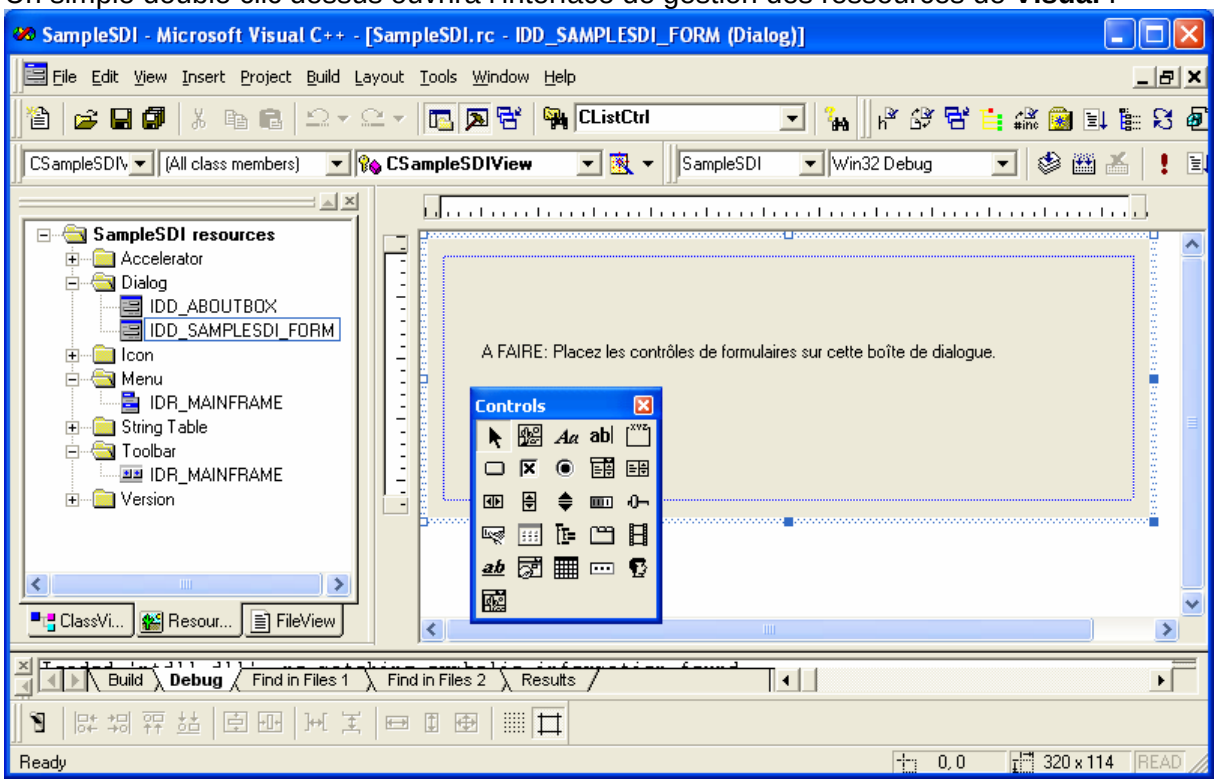
La fenêtre vue de notre application CSampleSDIView :

Volontairement j'ai choisi comme fenêtre de traitement une fenêtre de la classe de base **CFormView** .
Ce qui va nous permettre de travailler avec des contrôles et de mettre enfin en pratique les notions développées précédemment.

Notre fenêtre vue dispose elle aussi d'une boucle des messages permettant d'intercepter des messages de la barre d'outils ou des contrôles que nous allons placer maintenant :

Le Fichier des ressources graphiques de l'application (SampleSDI.rc)

L'ensemble des ressources graphiques de l'application est regroupé dans un fichier qui porte le nom de l'application plus l'extension **.RC**
Ce fichier apparaît dans votre projet, c'est un fichier au format texte éditable avec **notepad**.
Un simple double clic dessus ouvrira l'interface de gestion des ressources de **Visual** .



Dans le dossier dialogue nous voyons apparaître deux identifiants de fenêtres, le premier **IDD_ABOUTBOX** correspond à la boîte de dialogue « a propos de » de notre application.
Nous reviendrons plus tard sur les boîtes de dialogue.
Le deuxième **IDD_SAMPLESDI_FORM** correspond à notre fenêtre vue .

Comment Visual fait il le lien entre la fenêtre dessinée a l'écran et la classe associée **CSampleSDIView ?**

Par son identifiant qui apparaît dans la définition de la classe vue :

```
class CSampleSDIView : public CFormView
{
protected: // create from serialization only
	CSampleSDIView();
	DECLARE_DYNCREATE(CSampleSDIView)

public:
	//{{AFX_DATA(CSampleSDIView)
	enum{ IDD = IDD_SAMPLESDI_FORM };
		// NOTE: the ClassWizard will add data members here
	//}}AFX_DATA
```

Ce même identifiant est utilisé sur le constructeur de la classe pour indiquer quelle ressource utiliser lors de la construction de la fenêtre :

```
 ////////////////////////////////////////////////////////////////////////////
// CSampleSDIView construction/destruction

CSampleSDIView::CSampleSDIView()
	: CFormView(CSampleSDIView::IDD)
{
	//{{AFX_DATA_INIT(CSampleSDIView)
		// NOTE: the ClassWizard will add member initialization here
	//}}AFX_DATA_INIT
	// TODO: add construction code here

}
```

L'éditeur de ressources et les contrôles disponibles :

Dans l'éditeur de ressources on trouve une barre d'outils avec la liste des contrôles disponibles.
Cette liste correspond aux contrôles de base de Windows, elle est extensible avec des contrôles de type ActiveX.

Les contrôles disponibles :

Nom du Contrôle	Classe MFC associée	Description
Picture	**CPicture**	Affichage d'une image.
Static Text	**CStatic**	Affichage d'un libellé fixe
Edit Box	**CEdit**	Saisie d'une zone
Group Box		Permet de définir un groupe de contrôles.
Button	**CButton**	Gestion d'un bouton.
Check Box	**CButton**	Gestion d'une case a cochée.
Radio button	**CButton**	Gestion d'un bouton radio (un choix parmi d'autres).
Combo Box	**CComboBox**	Boite de sélection pouvant combiner un mode Edit et une liste déroulante.
List Box	**CListBox**	Liste de sélection mono colonne
Horizontal scroll bar	**CScrollBar**	Gestion d'un ascenseur horizontal
Vertical scroll bar	**CScrollBar**	Gestion d'un ascenseur vertical
Spin	**CSpinButtonCtrl**	Contrôle d'incrémentation d'un édit.
Progress	**CProgressCtrl**	Barre de progression
Slider	**CSliderCtrl**	Gestion d'un curseur analogique
Hot Key	**CHotKeyCtrl**	
ListControl	**CListCtrl**	Liste multi colonnes comme l'explorateur windows.
Tree Control	**CTreeCtrl**	Gestion d'un arbre multi branches
Tab Control	**CTabCtrl**	Gestion des onglets.
Animate		
Rich Edit	**CRichEdit**	Saisie d'une zone de texte au format **RTF** (**wordpad**)
Date Time Picker	**CDateTimeCtrl**	Sélection et saisie d'une date avec calendrier ou de l'heure.
Month calendar	**CMonthCalCtrl**	Gestion d'un calendrier
IP adress	**CIPAddressCtrl**	Saisie d'une adresse IP
Custom Control		Permet de faire un contrôle personnalisé
Extanded combo box	**CComboBoxEx**	Extension d'une **CComboBox** pour le support d'images par item (3 possibles /ligne)

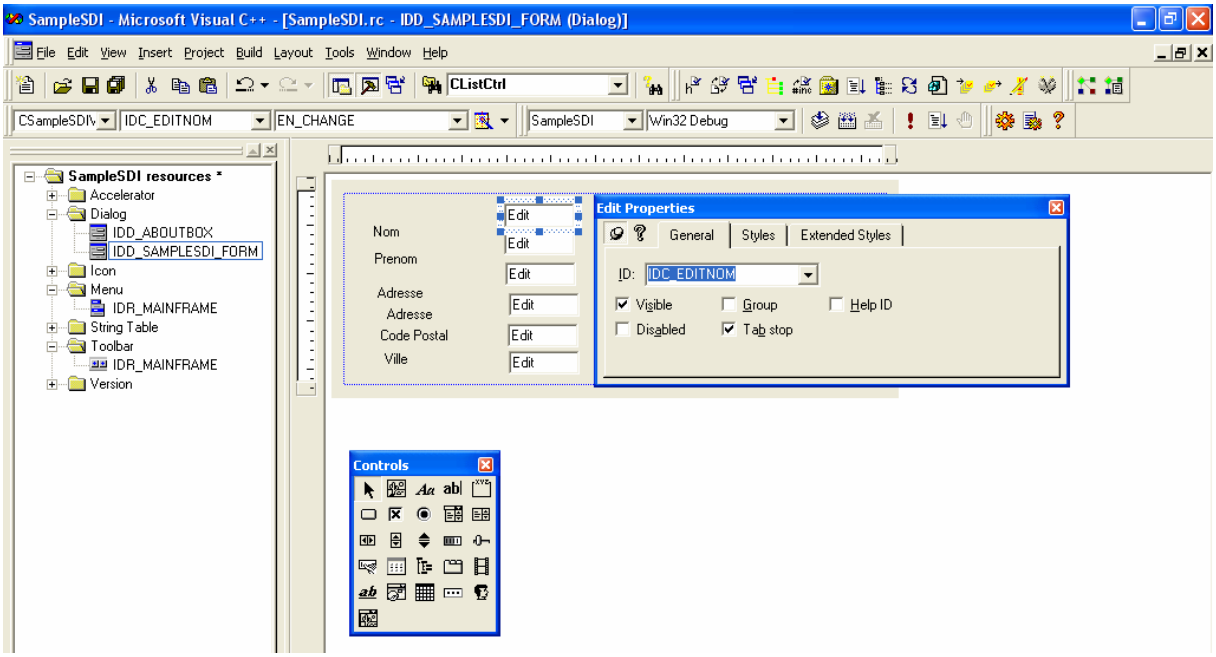
Mise en place des contrôles :

Nous allons placer sur notre fenêtre les éléments suivants :

Des zones d'éditions, des libellés et deux boutons.
Pour cela il suffit de cliquer sur la barre d'outils sur le contrôle en question, de positionner la souris sur la fenêtre et de cliquer a nouveau pour faire apparaître le contrôle.

Pour notre exemple nous allons mettre en place une fiche permettant de saisir :
Un nom
Un Prénom
Une adresse sur deux lignes.
Un code Postal
Une Ville.

Placer les contrôles sur la fenêtre, vous pouvez les dupliquer par les touches ou boutons habituels de copier/coller Windows.
Chaque contrôle dispose d'un identifiant généré automatiquement et modifiable.



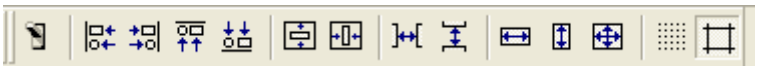
Renseignez les identifiant suivants :

Le nom : IDC_EDITNOM
Prénom : IDC_EDITPRENOM
Adresse : IDC_EDITADRESSE
Adresse : IDC_EDITADRESSE1
Code Postal : IDC_EDITCDP
La ville. : IDC_EDITVILLE
Un bouton enregistrer: IDC_BUTTONOK
Un bouton RAZ : IDC_BUTTONRAZ

Ajustez la longueur des différentes zones d'éditions en étirant les contrôles avec la souris.

Gestion de l'alignement des contrôles :

En bas de la zone d'édition de la fenêtre vous disposez d'une barre d'outils pour gérer l'alignement des contrôles :



Description de gauche à droite des boutons :

1. permet de simuler le fonctionnement de la dialogue en cours
2. alignement sur la gauche des contrôles sur le contrôle dominant (voir plus bas explications)
3. alignement sur la droite des contrôles sur le contrôle dominant.
4. alignement sur le bord supérieur des contrôles sur le contrôle dominant.
5. alignement sur le bord inférieur des contrôles sur le contrôle dominant.
6. centrage des contrôles verticalement sur la boîte de dialogue.
7. centrage des contrôles horizontalement sur la boîte de dialogue.
8. règle les espaces horizontalement entre les contrôles sélectionnés.
9. règle les espaces verticalement entre les contrôles sélectionnés.
10. donne la même largeur aux contrôles sélectionnés que le contrôle dominant.
11. donne la même hauteur aux contrôles sélectionnés que le contrôle dominant.
12. donne la même taille aux contrôles sélectionnés que le contrôle dominant.
13. affichage d'une grille dans la zone d'édition. (les guides disparaissent)
14. enlève la grille

La notion de contrôle dominant :

Nous allons aligner nos contrôles, pour cela réglez correctement la position du premier Edit le nom.

Ensuite sélectionnez avec la souris Clic gauche + touche Ctrl les autres contrôles Edit et en dernier le contrôle Edit Nom, qui sera notre contrôle dominant.
Sélectionnez ensuite le bouton d'alignement vers la gauche (2).
Le même principe de sélection est applicable pour les différents outils de la barre d'alignement.

Entraînez vous !
Agrandissez la zone adresse puis par copie affecter la même taille sur la zone la deuxième zone d'adresse. Alignez ensuite vos static.

Autres possibilités :

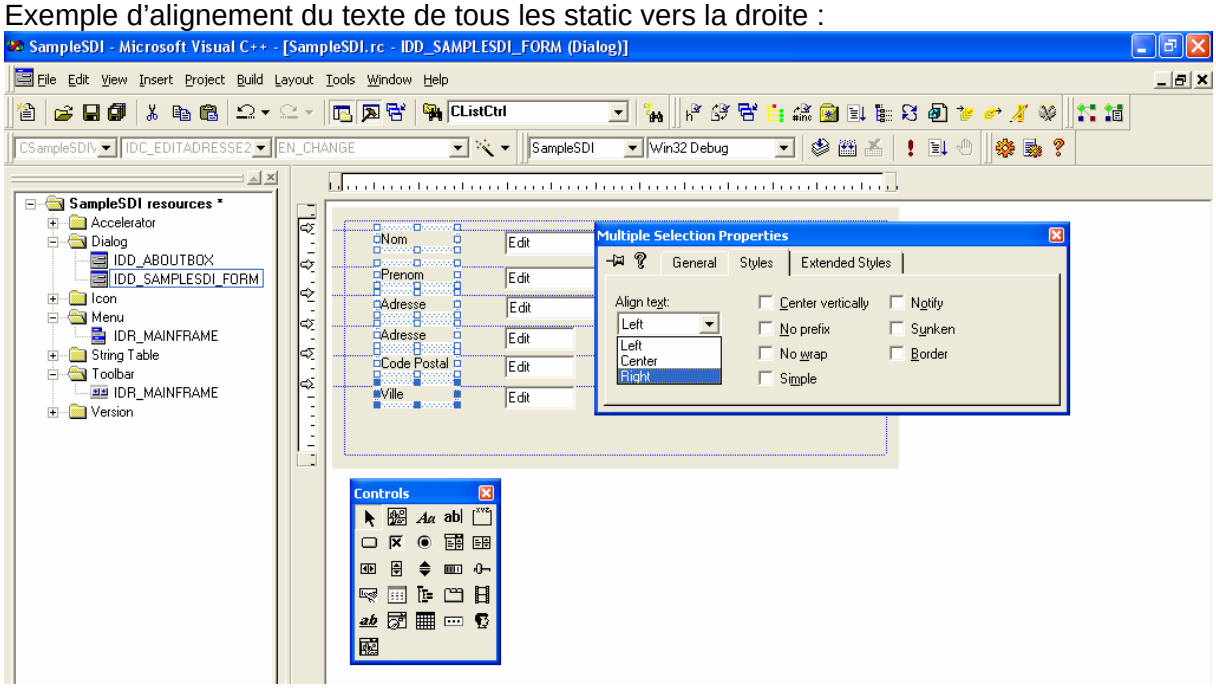
- L'ajout de lignes de guide sur lequel les contrôles sont placés pour faciliter le déplacement.
  Pour cela cliquez dans les règles sur les côtés de la fenêtres (voir ci-après).

- Le réglage des propriétés de plusieurs contrôles identiques en même temps :
  Il suffit de sélectionner les contrôles en question et de faire clic droit propriétés.

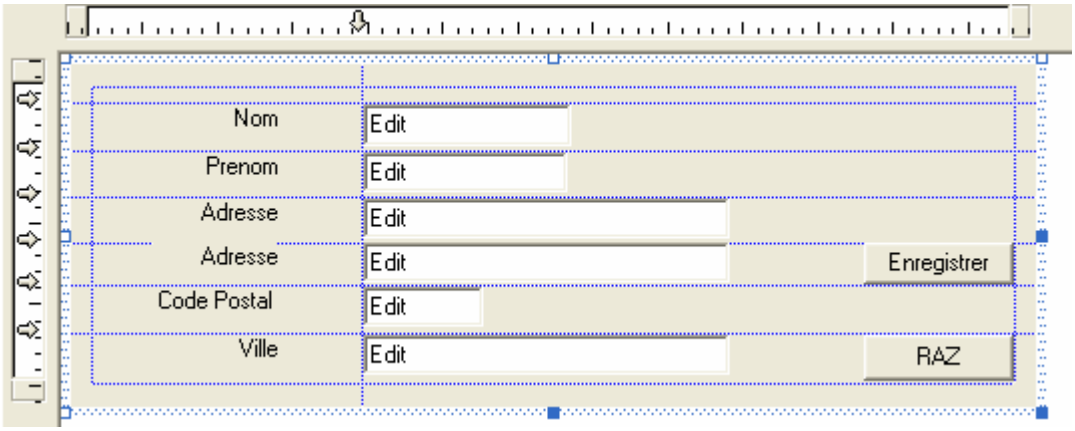
- La capture d'un groupe de contrôles :
  Il suffit de faire clic sur une zone libre de la fenêtre et de déplacer la souris pour délimiter une surface de sélection des contrôles.

- Le déplacement d'un groupe de contrôles :
  Un groupe de contrôle sélectionné peut être déplacé simplement avec la souris

Exemple d'alignement du texte de tous les static vers la droite :

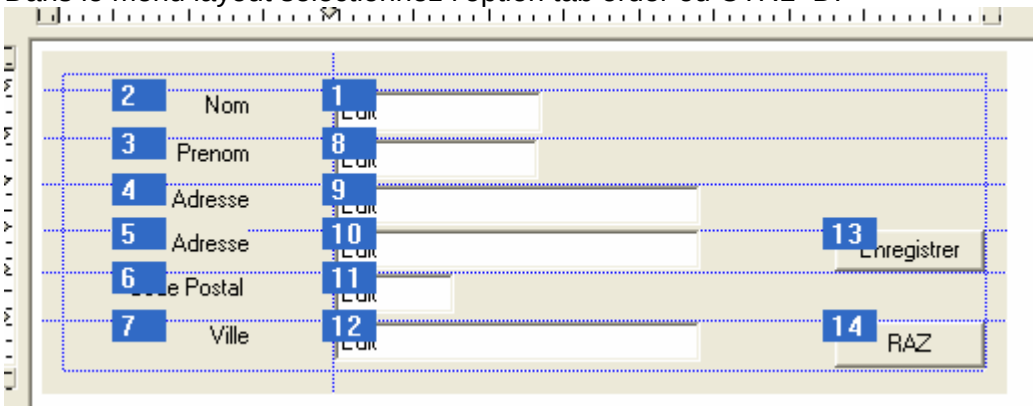


Notre fenêtre finale :

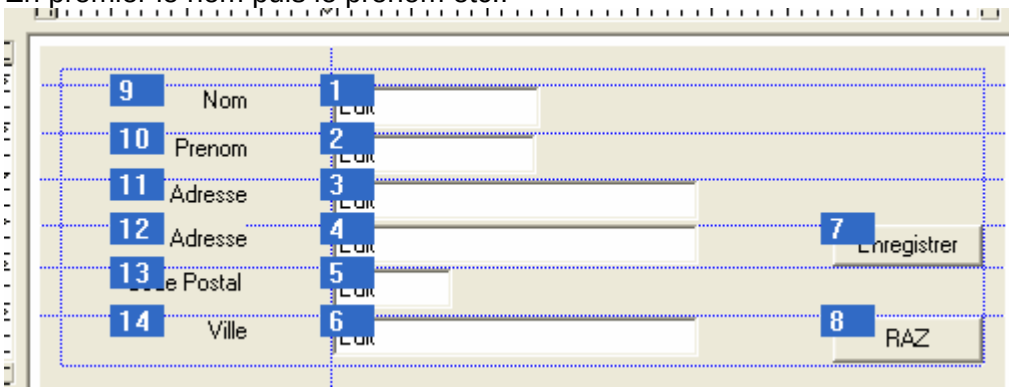


Gestion de l'ordre de saisie des contrôles :

Une fois les contrôles placés il faut régler l'ordre de saisie, qui correspond au déplacement avec la touche tabulation.
Dans le menu layout sélectionnez l'option tab order ou CTRL+D.

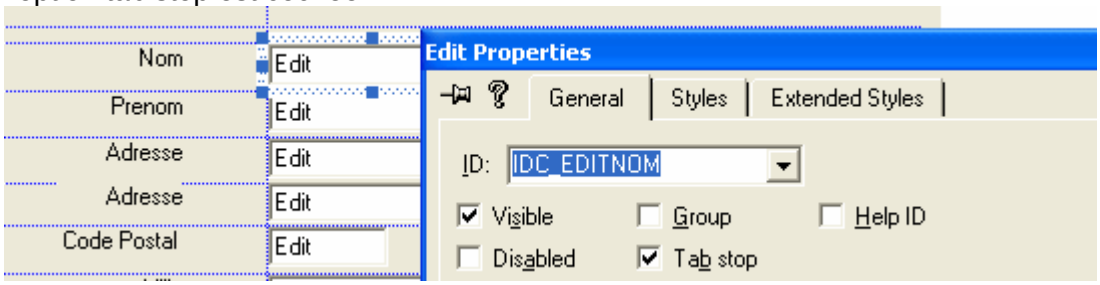


Cliquez sur les édits dans l'ordre naturel de saisie :
En premier le nom puis le prénom etc..

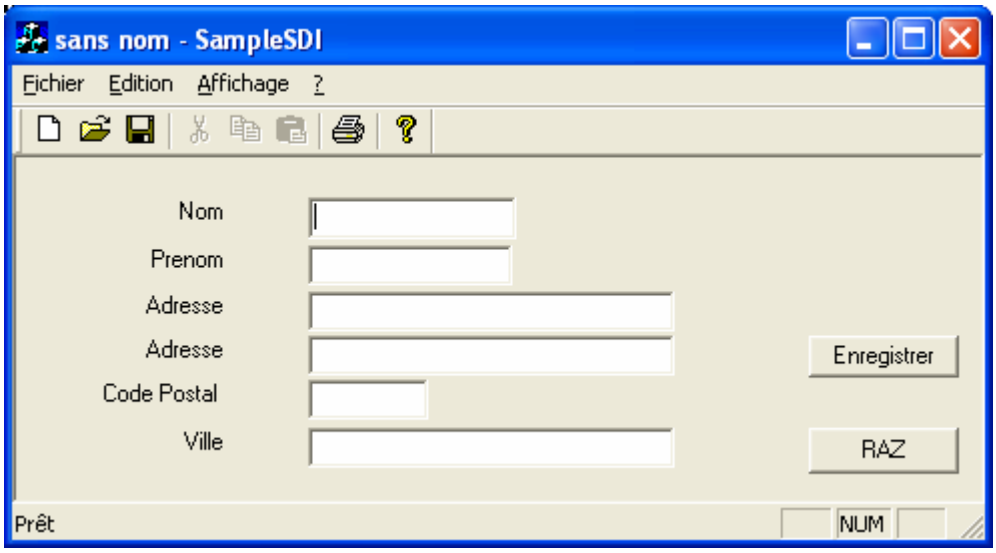


Pour finir le réglage de la tabulation cliquer sur une zone vierge de la fenêtre.

Une petite précision :
Un contrôle supporte la navigation par la touche tabulation si dans ses propriétés l'option tab stop est cochée.



Il ne reste plus qu'à compiler et lier notre programme pour apprécier le résultat :



Une question se pose maintenant :
Comment communiquer avec les contrôles, c'est-à-dire comme récupérer les valeurs saisies, ou comment placer une valeur dedans ? C'est l'objet du chapitre suivant.

Comment travailler avec les contrôles ?

Il y a deux manières de travailler avec des contrôles placés sur une fenêtre.

La première méthode consiste à récupérer un pointeur sur la fenêtre du contrôle.

Exemple :

```
CListBox *pListBox =static_cast<CListBox *>(GetDlgItem(IDC_LISTBOX)) ;
pListBox->AddString("coucou ") ;

CEdit *pEdit=static_cast<CEdit *>(GetDlgItem(IDC_EDITNOM)) ;
pEdit->SetWindowText("Robert") ;

// etc …
```

On l'aura compris la méthode **GetDlgItem** ici appliquée à la classe fenêtre **CSampleSDIView** permet de récupérer un pointeur de type **CWnd *** sur l'identifiant du contrôle passé en argument de la fonction.

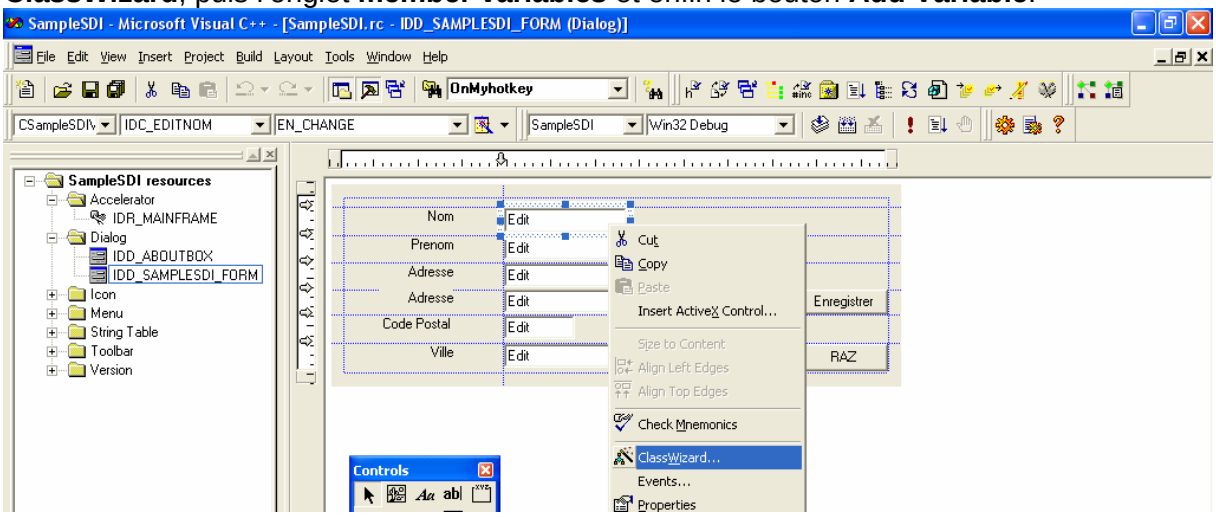
A partir de ce moment toutes les fonctionnalités de la classe **CWnd** sont accessibles comme par exemple affecter une nouvelle valeur à l'édit par la méthode **SetWindowText.**

La deuxième consiste à associer une variable au contrôle, à ce niveau on dispose encore de deux possibilités : la variable associée peut être le contrôle lui-même un **CEdit** une **CListBox** etc, ou une variable pour manipuler le contenu du contrôle.

Exemple : dans le cas d'un contrôle **CEdit** il sera pratique de travailler avec une variable **CString** pour changer ou récupérer le contenu du **CEdit** .
La classe **CString** est une classe utilitaire permettant de travailler avec les chaînes de caractères et ressemble sur certains points à la classe **string** des **STL**.

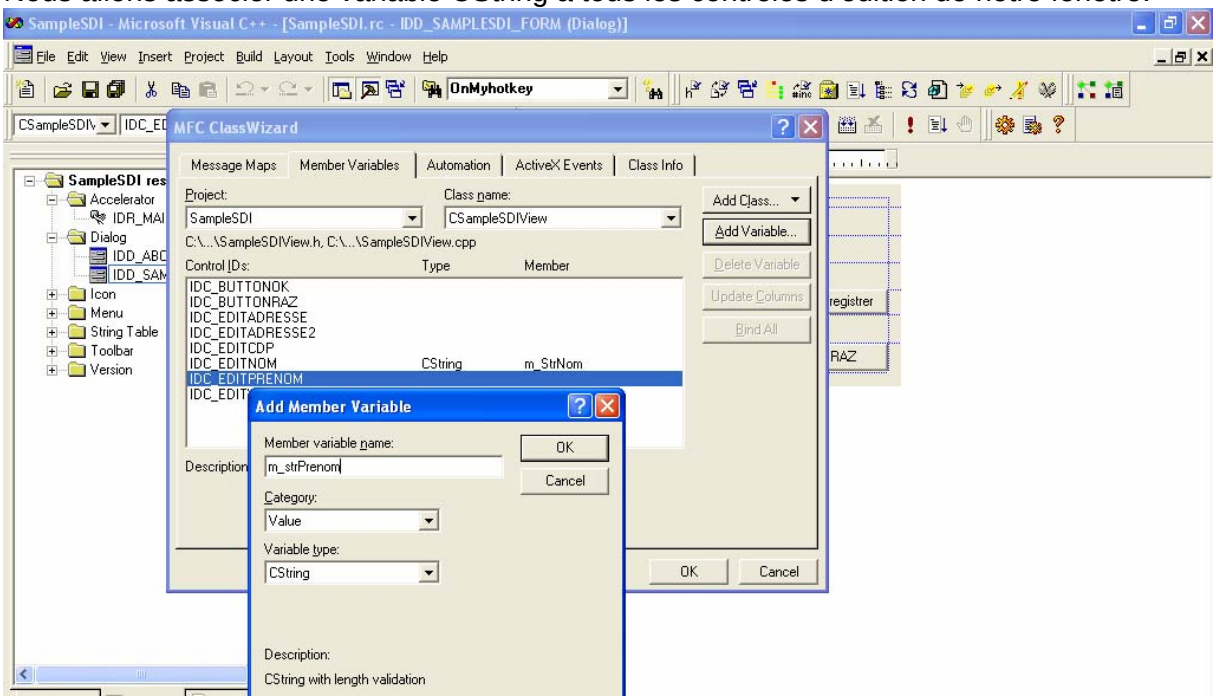
Cette association peut se faire directement à partir de l'éditeur de ressources :

Dans l'éditeur de ressources sur le contrôle en question faire clic droit. Sélectionnez l'option **ClassWizard**, puis l'onglet **member variables** et enfin le bouton **Add Variable**.

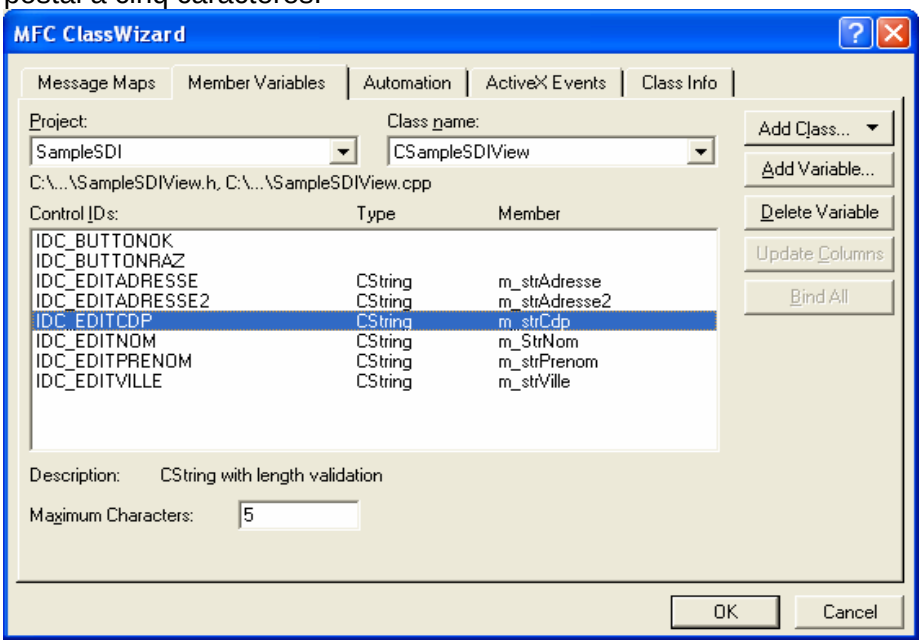


Il ne reste plus qu'à renseigner le nom de la variable et indiquer le type de variable : contrôle ou valeur. Dans le cas d'une valeur le type de variable **CString ,int ,long** c'est suivant le type de contrôle.

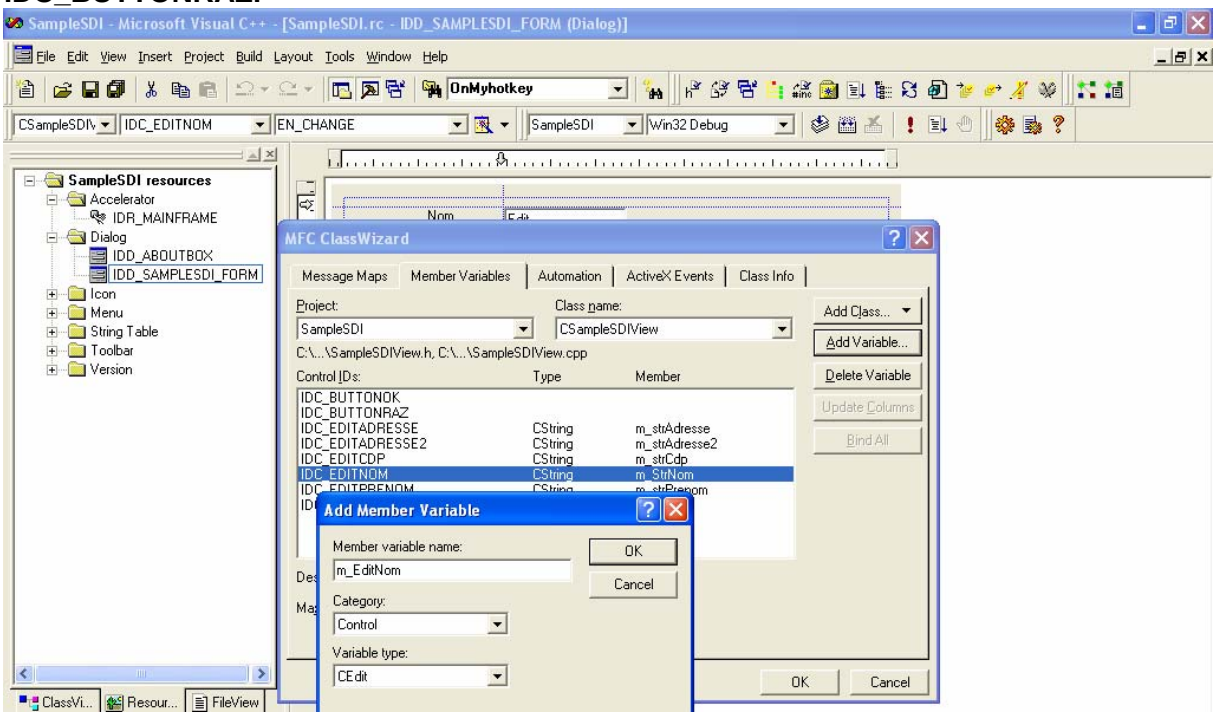
Exemple appliqué à notre projet de test :
Nous allons associer une variable CString à tous les contrôles d'édition de notre fenêtre.

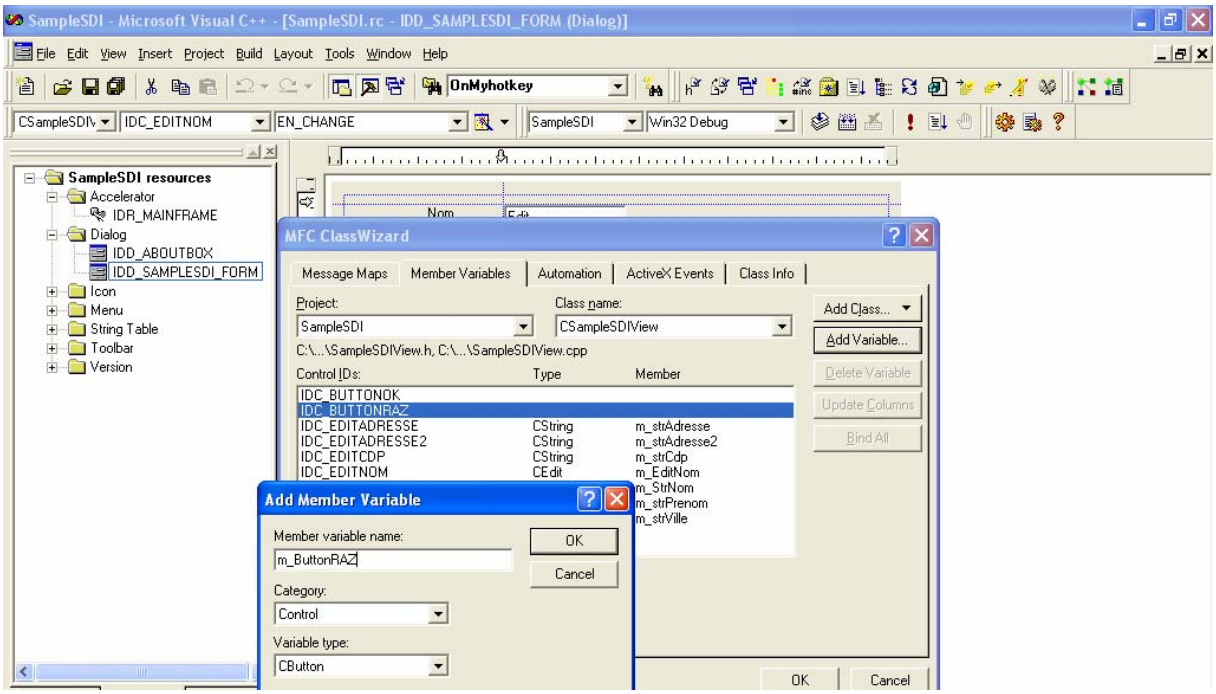


Dans le cas d'une **CString** sur un contrôle d'édition (**CEdit**) on pourra limiter la zone de saisie à un certain nombre de caractères. Exemple, ci-dessous je limite la saisie du code postal à cinq caractères.

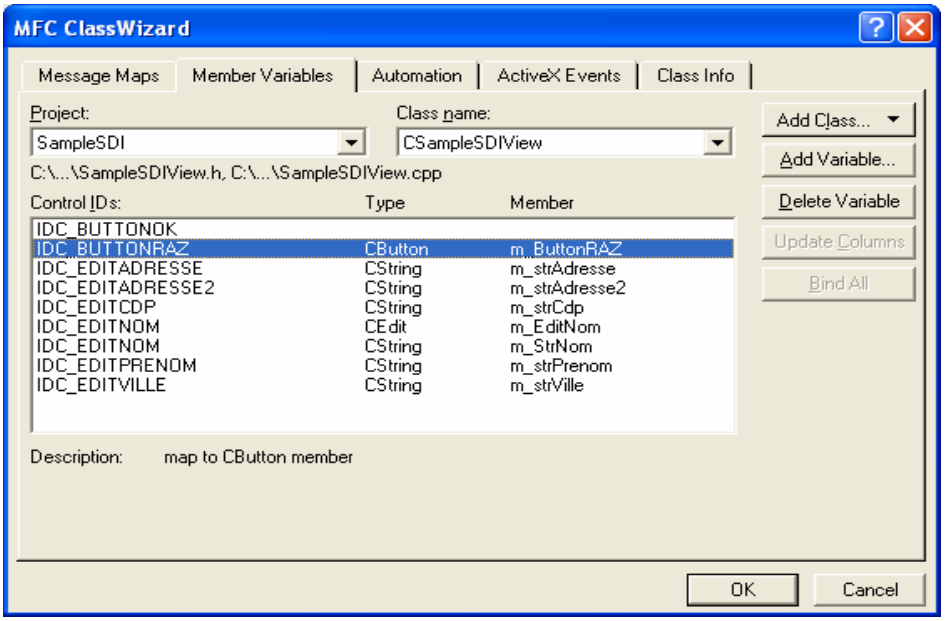


Pour bien comprendre le mécanisme d'association de variables, nous allons associer une variable de classe **CEdit** sur le nom, et une variable de classe **CButton** sur l'identifiant **IDC_BUTTONRAZ.**





Vous devez avoir au final :



Examinons le code généré par Visual :
Une variable est rajoutée dans la classe fenêtre où est situé le contrôle. Dans le .h de la classe on trouvera:

```
class CSampleSDIView : public CFormView
{
protected: // create from serialization only
	CSampleSDIView();
	DECLARE_DYNCREATE(CSampleSDIView)

public:
	//{{AFX_DATA(CSampleSDIView)
	enum { IDD = IDD_SAMPLESDI_FORM };
	CButton	m_ButtonRAZ;
	CEdit	m_EditNom;
	CString	m_StrNom;
	CString	m_strPrenom;
	CString	m_strVille;
	CString	m_strCdp;
	CString	m_strAdresse2;
	CString	m_strAdresse;
	//}}AFX_DATA
```

Dans le code :

```
CSampleSDIView::CSampleSDIView()
	: CFormView(CSampleSDIView::IDD)
{
	//{{AFX_DATA_INIT(CSampleSDIView)
	m_StrNom = _T("");
	m_strPrenom = _T("");
	m_strVille = _T("");
	m_strCdp = _T("");
	m_strAdresse2 = _T("");
	m_strAdresse = _T("");
	//}}AFX_DATA_INIT
	// TODO: add construction code here
}
```

La variable **CString** est initialisée dans le constructeur.

```
void CSampleSDIView::DoDataExchange(CDataExchange* pDX)
{
	CFormView::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CSampleSDIView)
	DDX_Control(pDX, IDC_BUTTONRAZ, m_ButtonRAZ);
	DDX_Control(pDX, IDC_EDITNOM, m_EditNom);
	DDX_Text(pDX, IDC_EDITNOM, m_StrNom);
	DDX_Text(pDX, IDC_EDITPRENOM, m_strPrenom);
	DDX_Text(pDX, IDC_EDITVILLE, m_strVille);
	DDV_MaxChars(pDX, m_strVille, 25);
	DDX_Text(pDX, IDC_EDITCDP, m_strCdp);
	DDV_MaxChars(pDX, m_strCdp, 5);
	DDX_Text(pDX, IDC_EDITADRESSE2, m_strAdresse2);
	DDX_Text(pDX, IDC_EDITADRESSE, m_strAdresse);
	//}}AFX_DATA_MAP
}
```

C'est la fonction **DodaExchange** qui établit le lien entre le contrôle Windows et les variables. Vous noterez les différentes macros utilisées en fonction des associations que nous avons effectuées.

DDX_Control : Permet l'association du contrôle à l'identifiant désigné.
En gros c'est ce contrôle qui va se substituer au fonctionnement par défaut du contrôle Windows, on parle de **subclassing**.

DDX_Text : établit une relation d'échange dans les deux sens entre une variable de classe **CString** et un contrôle.

DDV_MaxChars : Elle permet de préciser le nombre maxi de caractères **saisissables** pour la variable associée.

Cette fonction est appelée par la fonction **UpdateData** et le premier appel initialise les liens notamment pour la variable contrôle (**m_EditNom)**) qui va **subclasser** le contrôle windows.

Note : j'insiste, le contrôle sera graphiquement prêt (donnée membre **m_hWnd** différente de NULL) après le premier **UpdateData(FALSE)** et pas avant, celui étant fait par les MFC dans la fonction **CFormView::OnInitialUpdate** pour une **CFormView** ou **CDialog::OnInitDialog** pour une **CDialog** (nous verrons ça plus loin).

```
void CFormView::OnInitialUpdate()
{
  ASSERT_VALID(this);
  if (!UpdateData(FALSE))
    TRACE(traceAppMsg, 0, "UpdateData failed during formview initial update.\n");
  CScrollView::OnInitialUpdate();
}
BOOL CDialog::OnInitDialog()
{
  //……..
  // transfer data into the dialog from member variables
  if (!UpdateData(FALSE))
  {
    TRACE(traceAppMsg, 0, "Warning: UpdateData failed during dialog init.\n");
    EndDialog(-1);
    return FALSE;
  }
//……..
}
```

Conséquences :
Toute tentative d'utilisation d'un contrôle avant l'exécution de ces fonctions se solderont par une assertion d'erreur.
Exemple : faire **m_EditNom.*SetWindowText("Farscape") ;*** dans le constructeur.

Liste des erreurs communes en relation avec la fonction DodataExchange :

- Utiliser une variable dont le lien n'existe pas dans la fonction provoquera l'erreur.

```
ASSERT( IsWindow(m_hWnd) );
```

Ceci signifie que le **handle** de fenêtre n'est pas initialisé.

- Changer par mégarde l'identifiant d'un contrôle sans faire de même dans la fonction provoquera aussi une erreur.
- Supprimer un contrôle dans les ressources et garder le lien dans la fonction provoquera aussi une erreur.

Une fois ces initialisations faites on peut utiliser les deux formes de mise à jour et récupération de valeurs dans un contrôle.
Récupération d'une valeur sur un **CEdit** :

Directement avec la fonction **GetWindowText** .

```
CString str ;
m_EditNom.GetWindowText(str) ;
// ou
GetDlgItem(IDC_EDITNOM)->GetWindowText(str) ;
```

Par la variable :

```
UpdateData(TRUE) ; // mise a jour des variables associées aux contrôles
```

Affectation d'une valeur à un CEdit :
Directement avec la fonction **SetWindowText** :

```
CString str = "Farscape"  ;
m_EditNom.SetWindowText(str) ;
// ou
GetDlgItem(IDC_EDITNOM)->SetWindowText(str) ;
```

Par la variable :

```
m_strNom="Farscape" ;
UpdateData(FALSE) ; // mise a jour des contrôles à partir des variables associées.
```

Les lignes précédentes ont mis en lumière une nouvelle fonction **UpdateData.** Rentrons un peu dans le détail de cette fonction:

Définition :

```
CWnd::UpdateData
BOOL UpdateData( BOOL bSaveAndValidate = TRUE );
```

Cette fonction comme son nom l'indique permet :

- La mise à jour des variables attachées aux contrôles lorsque **bSaveAndValidate** est égal à **TRUE** .
- La mise à jour des contrôles depuis les variables lorsque **bSaveAndValidate** est égal à **FALSE**.

Jusque là rien d'extraordinaire. Il faut quand même savoir que ce mécanisme de mise à jour dans les deux sens est assujetti à une fonction essentielle dans la classe fenêtre où sont situés les contrôles :

```
CWnd::DoDataExchange
virtual void DoDataExchange( CDataExchange* pDX );
```

Cette fonction virtuelle sera générée automatiquement par Visual lors de la génération d'une classe Fenêtre. Le code de conversion approprié sera placé automatiquement en fonction des types de variables que l'utilisateur aura sélectionné ; les fichiers de définitions et sources étant bien sûr mis à jour automatiquement.

Il existe d'autres formes d'échanges : contrôle vers entiers, string etc..

DoDataExchange sera appelée chaque fois que **UpdateData** sera invoquée,
L'échange des données se fera par la fonction **DDX_xxxx** qui établira le lien entre le numéro d'identité du contrôle (IDC_) et sa variable.

Dans le cas de la fonction **DDX_Control** d'autres mécanismes sont mis en jeu.

```
// header afxdd_.h
// for getting access to the actual controls
void AFXAPI DDX_Control(CDataExchange* pDX, int nIDC, CWnd& rControl);
```

Explications complémentaires :
Au premier **UpdateData** le contrôle est « subclassé » ,c'est-à-dire qu'on va intercepter les messages Windows à destination du contrôle (tous les processus par défaut de Windows) pour lui donner un autre « moteur » de gestion des messages ,celui de l'objet mentionné dans la fonction **DDX_Control** .
Ce travail est fait par la fonction **CWnd ::SubclassWindow(HWND hWnd)** qui fait appel elle-même à

```
LONG SetWindowLong(
HWND hWnd,
int nIndex,
LONG dwNewLong);
```

C'est à la fin de ce traitement dont je viens de résumer les grandes lignes que la donnée membre **m_hWnd** de la variable contrôle sera affectée.

Conclusions :
Toute variable contrôle déclarée dans une fenêtre dialogue (**CDialog** ou **CFormView** etc ..) qui ne fera pas partie de la fonction **DoDataExchange** ne sera pas « subclassée » son **handle** de fenêtre sera égal à **NULL** et son utilisation provoquera une assertion d'erreur.

- Toute variable contrôle non présente dans la fonction **DoDataExchange** ne sera pas affectée par l'action de **UpdateData**.
- Pour subclasser un contrôle manuellement on pourra utiliser la fonction **SubClassDlgItem** :

```
BOOL CAboutDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  // IDC_BUTTON1 is the ID for a button on the
  // dialog template used for CAboutDlg.
  m_myButton.SubclassDlgItem(IDC_BUTTON1, this);
 //……….
```

Pour placer dans la fonction **DoDataExchange** une variable dynamique utilisez la technique suivante :

```
void CBnqView::DoDataExchange(CDataExchange* pDX)
{
   CMyFormView::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CBnqView)
   DDX_Control(pDX, IDC_CEDITCDEBNQ, m_EditCdeBnq);
   // m_pEditDynamique mis à null dans le constructeur et initialisé dans OnInitialUpdate.
   if(m_pEditDynamique)
   {
      DDX_Control(pDX, IDC_CEDITDYN, *m_pEditDynamique);
   }
   DDX_Control(pDX, IDC_CEDITCDEGUICHET, m_EditCdeGuichet);
   DDX_Control(pDX, IDC_CEDITDOM, m_EditDom);
  //……..
  //}}AFX_DATA_MAP
}
```

L'appel à la fonction **DDX_Control** se fera uniquement si le pointeur est différent de NULL.

La fonction d'initialisation de la Vue OninitialUpdate :

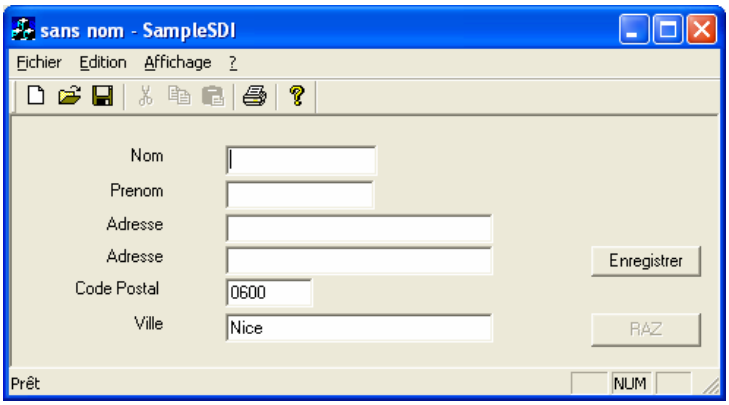
```
void CSampleSDIView::OnInitialUpdate()
{
        CFormView::OnInitialUpdate();
        GetParentFrame()->RecalcLayout();
        ResizeParentToFit();

}
```

Dans le cas d'une vue c'est la fonction virtuelle **OnInitialUpdate** qui est appelée pour procéder aux différentes mises à jour utilisateur.
La première ligne appelle la méthode de la classe de base et procède donc aux initialisations des contrôles (voir explications précédentes).
Les deux lignes qui suivent permettent d'ajuster la fenêtre au contenu.

Pour mettre en pratique tout ce que nous avons dit, nous allons donner une valeur par défaut à certains contrôles :

```
void CSampleSDIView::OnInitialUpdate()
{
        CFormView::OnInitialUpdate();
        GetParentFrame()->RecalcLayout();
        ResizeParentToFit();

   // initialisations manuelles:
   m_strVille="Nice";
   m_strCdp="0600";

   UpdateData(FALSE); // -> mise a jour des contrôles depuis les variables.
   m_ButtonRAZ.EnableWindow(FALSE);// désactiver le bouton

}
```

Résultat après construction du programme :

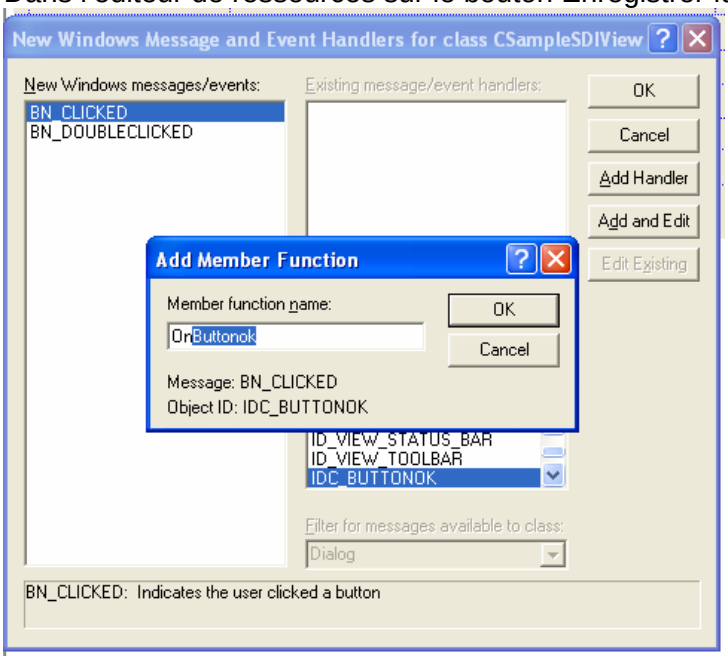


Je vais maintenant rajouter le code suivant :

- Permettre l'enregistrement si tous les contrôles sont renseignés.
- Réactivez le bouton RAZ si un des quatre premiers contrôles est rempli
- Implémenter le traitement de la remise à zéro sur le bouton RAZ.

Intercepter le clic sur un bouton :

Dans l'éditeur de ressources sur le bouton Enregistrer faire clic droit option **events**



A gauche nous avons le type d'événement à droite les événements existants.
Validez le choix de la fonction proposé. Visual a généré automatiquement la fonction de réponse dans la source :

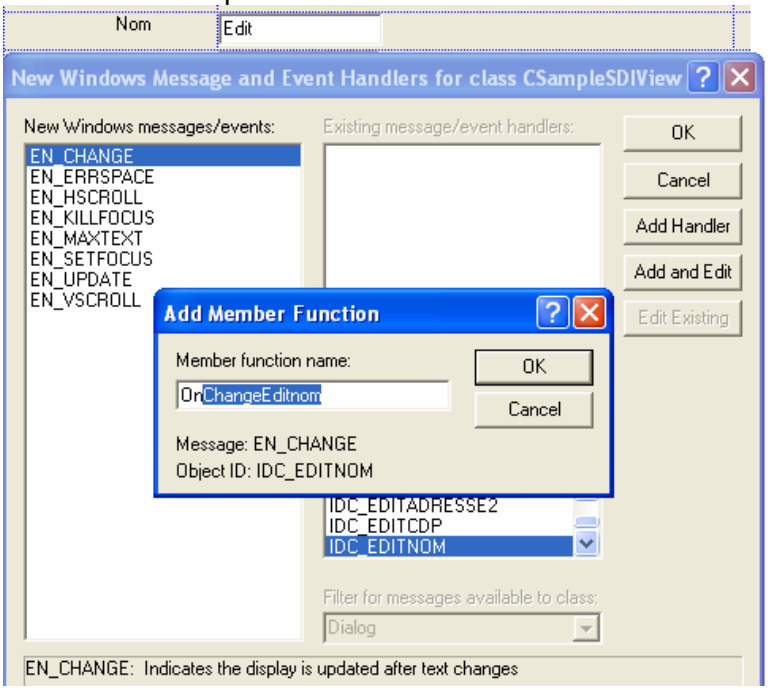
```
BEGIN_MESSAGE_MAP(CSampleSDIView, CFormView)
//{{AFX_MSG_MAP(CSampleSDIView)
ON_BN_CLICKED(IDC_BUTTONOK, OnButtonok)
//}}AFX_MSG_MAP
```

```
void CSampleSDIView::OnButtonok()
{
   // TODO: Add your control notification handler code here
   UpdateData(TRUE); // mise a jour des données.
   // tableau des CString associées aux contrôles
   CString *parString[]={&m_StrNom,&m_strPrenom, &m_strAdresse2,
                         &m_strAdresse, &m_strVille,&m_strCdp   };
   // tableau des identifiants correspondant
   UINT arnId[]={IDC_EDITNOM,IDC_EDITPRENOM, IDC_EDITADRESSE,
            IDC_EDITADRESSE2,IDC_EDITCDP,IDC_EDITVILLE };

   for(int i=0;i<sizeof(parString)/sizeof(CString *);i++)
   {
      // si la chaine est vide on redonne la main en saisie au contrôle en question.
      if(parString[i]->IsEmpty())
      {
         GetDlgItem(arnId[i])->SetFocus();
         return;
      }
   }
   if(AfxMessageBox("confirmez l'enregistrement",MB_YESNO|MB_ICONQUESTION)==IDYES)
   {
      // sauvegarde des données.
   }
}
```

Reportez vous à la documentation MSDN pour les classes et méthodes utilisées:

Intercepter un message sur un contrôle :

Nous allons maintenant intercepter un message généré par le contrôle dans la vue :
Toujours dans les ressources, placez vous sur le contrôle **CEdit** de saisie du nom et Faites clic droit option **events.**



Nous allons intercepter le message qui indique qu'un changement est arrivé dans le contrôle **IDC_EDITNOM** ,validez le choix et procédez de même pour les contrôles :
IDC_EDITPRENOM,IDC_EDITADRESSE, IDC_EDITADRESSE1

Visual va générer une fonction pour chaque contrôle, cette notification je vais la rediriger sur une seule fonction de traitement **OnMajRAZButton :**

```
void CSampleSDIView::OnMajRAZButton()
{
   UINT arnId[]={IDC_EDITNOM,IDC_EDITPRENOM,
            IDC_EDITADRESSE,IDC_EDITADRESSE2};

   int nFull=0;
   CString str;
   for(int i=0;i<sizeof(arnId)/sizeof(UINT);i++)
   {
      GetDlgItem(arnId[i])->GetWindowText(str); // récupération du texte dans le contrôle
      nFull+=(!str.IsEmpty());
   }
  // active ou désactive le contrôle si tous les contrôles sont remplis ou vides.
   m_ButtonRAZ.EnableWindow((nFull==sizeof(arnId)/sizeof(UINT)));
}
void CSampleSDIView::OnChangeEditnom()
{
  // TODO: Add your control notification handler code here
   OnMajRAZButton();
}
// etc ...
```

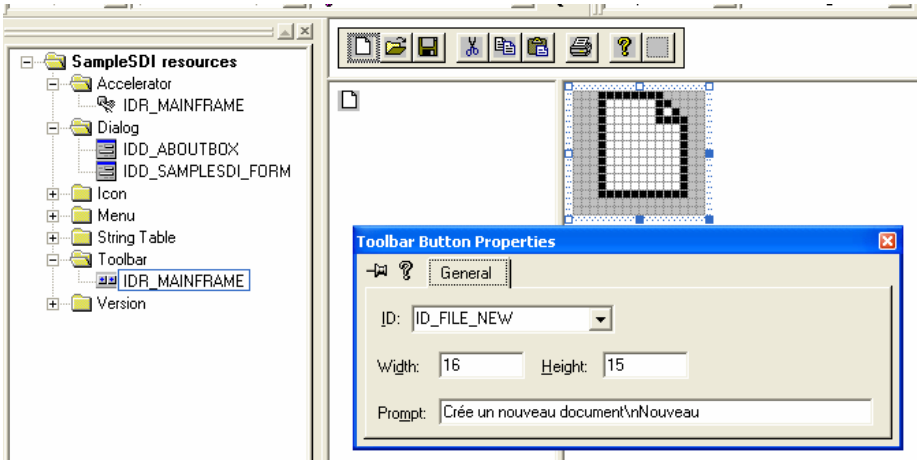
Une question se pose, pourquoi ne pas avoir fait **UpdateData(TRUE)** ?

UpdateData met à jour tous les contrôles et dans le contexte présent, je suis dans le cas où pour chaque caractère saisi dans les édits concernés un message va être envoyé.
On utilisera ce style d'écriture chaque fois que l'on désire de ne pas perturber les variables associées avec des valeurs intermédiaires.
Continuons l'exemple en implémentant la remise à zéro des contrôles. Pour cela interceptez le message de clic sur le bouton RAZ

```
void CSampleSDIView::OnButtonraz()
{
        // TODO: Add your control notification handler code here
CString *parString[]={&m_StrNom,&m_strPrenom,
                &m_strAdresse2,&m_strAdresse,
                &m_strVille,&m_strCdp};

   for(int i=0;i<sizeof(parString)/sizeof(CString *);i++)
      parString[i]->Empty();
   UpdateData(FALSE); // mise a jour des contrôles.
}
```

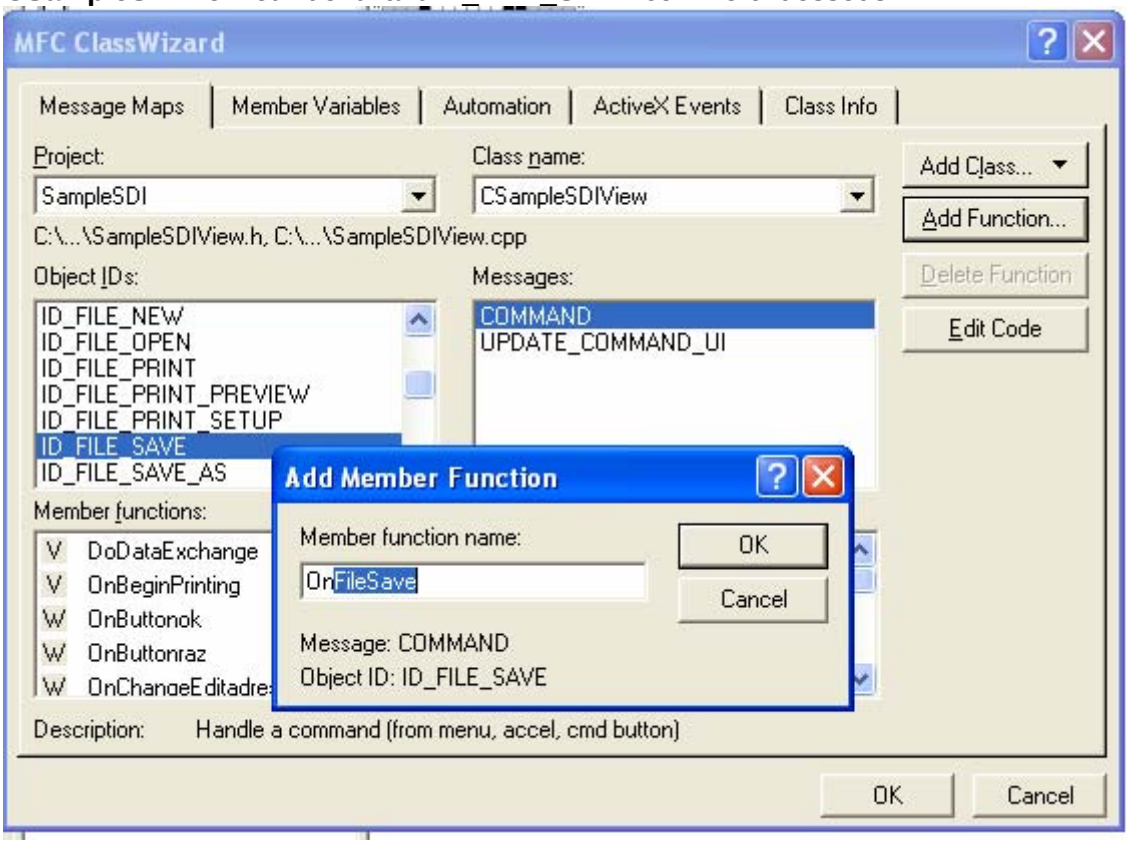
Implémenter une commande d'une barre d'outils

Pour continuer notre apprentissage sur la mise en place des traitements en réponse aux messages, nous allons voir comment traiter un clic sur un bouton de la barre d'outils.



Celle-ci est définie dans l'éditeur de ressources dans la section **Toolbar**.
Chaque bouton dispose d'un identifiant. Double cliquez sur un des boutons pour voir le paramétrage. La ligne « **prompt »** correspond aux libellés affichés respectivement sur la barre d'états (statusbar) et sur la bulle du bouton (tooltip). Les deux libellés sont séparés par un **'\n'**.

Mise en place du message :
Appelez classwizard (CTRL+W) ,dans la zone **« classe name »** sélectionnez la vue **CSampleSDIView** et l'identifiant ID_FILE_SAVE comme ci-dessous :



Dans la partie messages deux messages :
COMMAND **:** traitement du clic
UPDATE_COMMAND_UI **:** possibilité d'autoriser ou d'interdire le traitement du message COMMANDE
Interceptez les deux messages.
Je vais modifier mon code pour prendre en compte le traitement déjà existant sur le bouton Enregistrer :

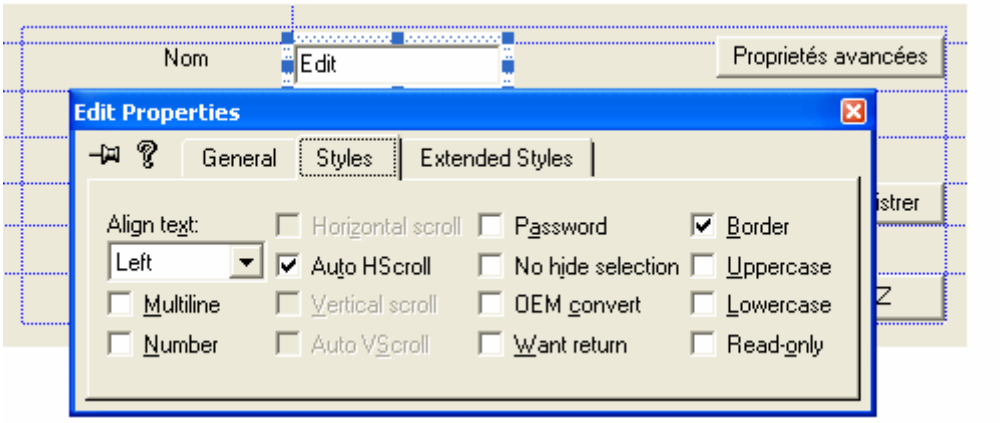
```
bool CSampleSDIView::CanSaveData(bool bSetFocus/*=false*/)
{
   UpdateData(TRUE); // mise a jour des données.
   // tableau des CString associées aux contrôles
   CString *parString[]={&m_StrNom,
                         &m_strPrenom,&m_strAdresse2,
                         &m_strAdresse, &m_strVille,&m_strCdp
                        };
   // tableau des identifiants correspondant
   UINT arnId[]={IDC_EDITNOM,IDC_EDITPRENOM,
                 IDC_EDITADRESSE, IDC_EDITADRESSE2,
                 IDC_EDITCDP,IDC_EDITVILLE
                };

   for(int i=0;i<sizeof(parString)/sizeof(CString *);i++)
   {
      // si la chaine est vide
      if(parString[i]->IsEmpty())
      {
         // on redonne la main en saisie au contrôle en question.
         if(bSetFocus) GetDlgItem(arnId[i])->SetFocus();
         return false;
      }
   }
   return true;
}
void CSampleSDIView::OnButtonok()
{
// TODO: Add your control notification handler code here
   if(CanSaveData(true) && AfxMessageBox("confirmez l'enregistrement",MB_YESNO
                                         |MB_ICONQUESTION)==IDYES)
   {
      // sauvegarde des données.
   }
}
void CSampleSDIView::OnFileSave()
{
   // TODO: Add your command handler code here
   if(AfxMessageBox("confirmez l'enregistrement",MB_YESNO |MB_ICONQUESTION)==IDYES)
   {
      // sauvegarde des données.
   }
}
void CSampleSDIView::OnUpdateFileSave(CCmdUI* pCmdUI)
{
   // TODO: Add your command update UI handler code here
   pCmdUI->Enable(CanSaveData());
}
```

pCmdUi->Enable permet d'activer ou de désactiver le bouton en fonction du résultat de la Méthode **CanSaveData().**
Ce code montre bien la convergence de traitements en fonction d'éléments d'interface différents mais représentant le même traitement. Ici le bouton sur ma vue et le bouton sur la barre d'outils. Il est à noter que la ligne **enregistrer** du menu **fichier** est aussi traitée, car partageant le même Identifiant que le bouton de la barre d'outils

Compléments d'informations sur les contrôles

Les Edits :

Un contrôle **CEdit** peut être multi lignes, il suffit pour cela de cocher dans les propriétés du contrôle l'option **« multiline »** et **« want return »** pour la gestion du retour chariot.
Les principales propriétés d'un édit sont résumées ci-dessous :



On retrouve la notion d'alignement, ainsi que :

- Le mode multi ligne.
- La saisie exclusive de nombre
- La gestion du scroll horizontal et vertical du texte.
- La saisie en mode mot de passe (avec des *).
- La mise en majuscule / minuscule.
- La possibilité de mettre l'édit en lecture seule.

Les boutons Radios :

On utilisera les radios chaque fois que l'on doit faire un choix unique parmi des options.
Exemple : imaginons le traitement du régime TVA d'un client on aura par exemple **France / Corse / Export / CEE .**

1. **étape :**
   Dans l'éditeur de ressources sur la boîte de dialogue concernée on placera le premier radio en spécifiant son nom :

   Dans mon exemple je le nommerai **IDC_RADIOTVA** son label France
   Ensuite ne pas oublier de cocher l'option **Tab Stop** et surtout l'option **Group** permettant de spécifier que l'on commence un nouveau groupe et donc que les contrôles suivants de même nature en font partie.

   On place ensuite les autres radios et on laissera l'éditeur donner l'ID du radio qu'il incrémentera de manière automatique par rapport au premier radio.

   Si je dois placer un autre groupe de radios derrière il faudra juste spécifier sur le premier radio l'option groupe pour distinguer les deux groupes.

2. **étape :**
   Pour connaître la valeur de la sélection il faut attacher une variable uniquement sur le premier bouton radio du groupe.

Pour cela il faut lancer « ClassWizard » ,onglet « member variables » ,sélectionner dans la liste l'id du bouton radio ici **IDC_RADIOTVA** .

Cliquez sur le bouton « Add Variable » et spécifiez le nom de la variable de type « int » Exemple « m_nRegimeTVA ».
Et validez le choix par le bouton « Ok »

« ClassWizard » :
- génère automatiquement le code de déclaration dans la classe
- initialise la valeur dans le constructeur – 1
- place le code d'échange des informations entre la variable et le contrôle dans la fonction membre de la classe dialogue : **DoDataExchange(CDataExchange* pDX).**

3. étape :
Pour donner une valeur de départ à l'affichage de la fenêtre il suffira d'initialiser la variable dans la fonction **InitDialog** pour une boite de dialogue modale (**CDialog**) et **OnInitialUpdate** pour une classe **CFormView** :

m_nRegimeTVA=1 ;

Ceci sélectionnera le deuxième radio de la liste, l'indexation commençant à zéro et finissant au nombre de bouton radio du groupe – 1 dans mon exemple : 3.
La valeur à « -1 » voulant dire aucune valeur sélectionnée.
Pour signifier sa valeur au contrôle on utilisera la fonction

UpdateData(FALSE) ;

Pour la mise à jour du contrôle et

UpdateData(TRUE) ;

Pour récupérer la valeur dans la variable.

Note : **UpdateData** s'applique bien sur à toutes les variables présentes dans la fonction **DoDataExchange.**

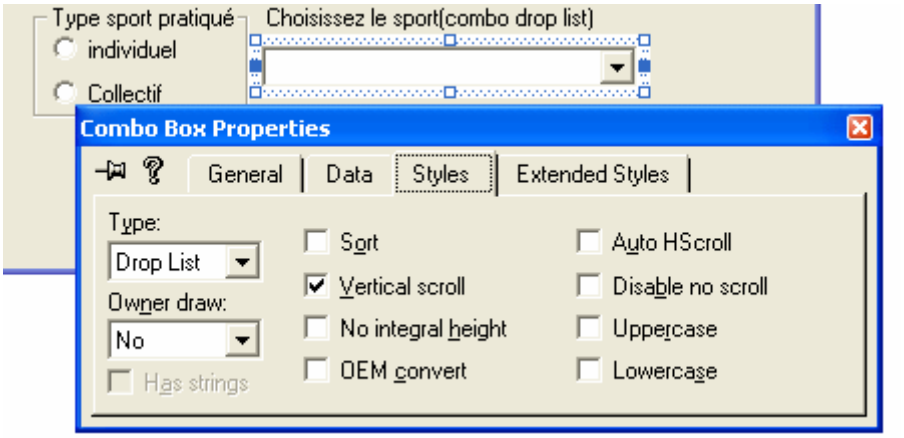
Autre cas: si la variable ne suffit pas et que l'on désire par exemple récupérer le label du radio sélectionné on peut procéder comme suit.
Toujours par rapport à mon exemple :

```
CString CMyDialog::GetRadioLabelForValue(UINT nId,int &rnValue)
{
     CWnd *pRadio=NULL; UpdateData(TRUE);
      if(rnValue >=0)
      {
        pRadio=GetDlgItem(nId);
        for(int n=0;pRadio && n< rnValue;n++)
         pRadio=pRadio->GetWindow (GW_HWNDNEXT);
       }
       CString str;
       if(pRadio) pRadio->GetWindowText(str); return str;
}
CString str=GetRadioLabelForValue(IDC_RADIOTVA , m_nRegimeTVA);
```

Les ComboBox :

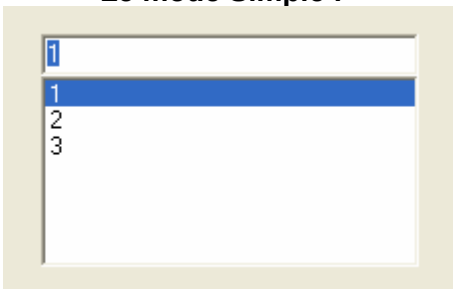
Leur utilisation est assez simple

- **Mode de fonctionnement :**



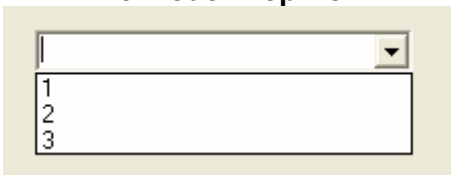
Elle supporte plusieurs modes de fonctionnement conditionnés à la valeur de la zone Type :

- ✓ **Le mode Simple :**



Une zone d'édition et une zone fixe de choix réglable.

- ✓ **Le mode Drop Down :**



Une zone d'édition et une liste déroulante popup pour les options.

- ✓ **Le mode Drop List :**

Une liste déroulante pour les options celles-ci n'étant pas modifiables.

- **Le réglage de la liste déroulante :**

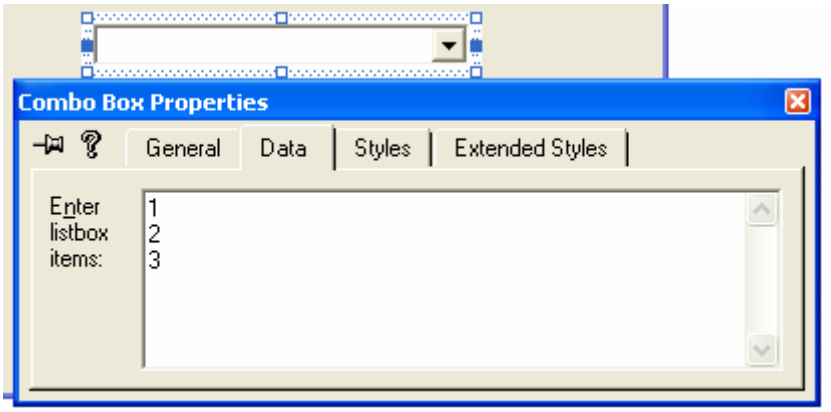
  La taille par défaut de la liste déroulante correspond à celle d'un élément, donnant ainsi l'impression qu'elle ne s'ouvre pas.
  Son réglage se fait dans l'éditeur de ressources :
  Sélectionner le contrôle **ComboBox** et cliquer sur la flèche de la liste, un rectangle montrant la hauteur de la liste apparaît, il suffit alors de lui régler sa taille avec la souris.

- **Le remplissage d'une CCombobox** :

  Il peut se faire de deux manières :

  - ✓ **Les données sont fixes :**

  On peut les rentrer directement au niveau des ressources :



Note : faire **CTRL+entrée** pour rentrer plusieurs lignes.

- ✓ **Le remplissage est dynamique :**

Il suffira de remplir la **CComboBox** par sa méthode **AddString** . :

Si elle n'a pas le style **CBS_SORT** la chaîne est insérée en fin de liste.
Cette fonction renvoie le numéro d'item dans la liste box la première démarrant à zéro.

```
CComboBox::AddString
int AddString( LPCTSTR lpszString );

CString str;
int nIndex;
for (int i=0;i < 20;i++)
{
 str.Format("Ma chaîne %d"), i);
 nIndex=MyComboBox.AddString( str );
}
```

Autre fonction utilisable:
CComboBox::InsertString int InsertString(int nIndex, LPCTSTR lpszString);

Même chose que **AddString** mais ici on précise l'emplacement avec l'argument nIndex. Si nIndex == -1 l'insertion se fait en bout de liste.

- **La sélection d'une ligne :**

CComboBox::SetCurSel int SetCurSel(int nSelect);

La valeur retournée est l'indice de l'élément sélectionné si la commande réussit, Sinon elle renvoie **CB_ERR** pour indiquer une erreur. Si nSelect est égal à -1 la sélection en cours est supprimée et le retour sera aussi **CB_ERR**.
Exemple sélection du dernier élément de la combobox.

```
int nCount = MyComboBox.GetCount();
if (nCount > 0) MyComboBox.SetCurSel(nCount-1);
```

Pour la sélection en fonction d'une chaîne de caractère utilisez **SelectString** qui est la combinaison de **FindString** et **SetCurSel**.

- **Récupération de la sélection en cours :**

  - ✓ Récupération de l'indice de l'item en sélection :

CComboBox::GetCurSel int GetCurSel( ) const;

  - ✓ Récupération d'une chaîne pour un numéro d'indice :

CComboBox::GetLBText int GetLBText(int nIndex, LPTSTR lpszText) const;
void GetLBText(int nIndex, CString& rString) const;

```
CString str ;
 int nIndex = MyComboBox.GetCurSel();
 if(nIndex!=LB_ERR)
            MyComboBox.GetLBText(nIndex,str);
 AfxMessageBox(str);
```

- **Suppression d'une ligne :**

CComboBox::DeleteString int DeleteString(UINT nIndex);

```
for (int i=0;i < MyComboBox.GetCount();i++)
 {
 MyComboBox.DeleteString( i );
 }
 // Ce même code peut être écrit de la manière suivante : MyComboBox.ResetContent()
;
```

CComboBox::ResetContent void ResetContent( );

Supprime tous les éléments de la **combobox**.

Les Listbox

Il est relativement aisé de travailler avec une **CListbox** :

Le plus simple étant de déclarer une variable de la classe **CListBox** attachée au contrôle et d'utiliser directement les méthodes de cette classe.

Traitements courants :

- **Insérer des éléments :**

  CListBox::AddString int AddString(LPCTSTR lpszItem);

```
CString str;
 for (int i=0;i < 10;i++)
{
        str.Format(_T("item string %d"), i);
        MyListBox.AddString( str );
 }
```

- **Récupérer le texte d'une ligne :**

  On utilisera la méthode **GetText :**

  CListBox::GetText

  int GetText(int nIndex, LPTSTR lpszBuffer) const;
  void GetText(int nIndex, CString& rString) const;

  Exemple : récupérer le texte de la ligne sélectionnée:

```
CStrng str;
int nIndex = myListBox.GetCurSel();
if((nIndex != LB_ERR)) myListBox.GetText( nindex,str );
AfxMessageBox(str);  // affichage dans une boîte de dialogue
```

- **Supprimer une ligne :**

  Si on ne dispose pas du numéro index de la ligne il faudra la rechercher en utilisant la fonction **FindString** :

  CListBox::FindString int FindString(int nStartAfter, LPCTSTR lpszItem) const;

  Exemple:

```
nIndex = myListBox.FindString(0,"coucou");
if((nIndex != LB_ERR)) myListBox.DeleteString( nindex );
```

  Autre exemple supprimer la ligne en cours de sélection :

  En utilisant la fonction **GetCurSel** qui renvoie l'indice courant sélectionné.

```
int nIndex = myListBox.GetCurSel();
if((nIndex != LB_ERR))  myListBox.DeleteString( nindex );
```

- **Sélectionner le dernier élément :**

```
// Nombre d'éléments dans la listbox
int nCount = MyListBox.GetCount(); // si pas d'erreur listbox non vide ,
sélection sur l'item nombre d'éléments -1
 // Puisque les indices commencent à zéro
 if (nCount > 0)  MyListBox.SetCurSel(nCount-1);
```

- **Détruire tous les éléments :**

  On utilisera la méthode **ResetContent( )**

  CListBox::ResetContent void ResetContent( );

```
// Détruire tous les éléments d'une ClistBox
MyListBox.ResetContent();
ASSERT(MyListBox.GetCount() == 0);
```

Voila pour les principales fonctionnalités d'une **CListBox**

Le Traitement des couleurs

Les contrôles MFC ne gèrent pas les couleurs nativement dans l'éditeur de ressources comme ça se fait en **Visual Basic** par exemple (qui utilise des Activex).

Ça ne veut pas dire pour autant que le traitement de la couleur est impossible, ça nécessitera d'intégrer du code en interceptant un message spécifique.

Dans les MFC chaque contrôle envoie un message au parent pour demander sa couleur. De ce fait si on veut gérer les couleurs des contrôles il suffit d'intercepter le message **WM_CTLCOLOR** au niveau de la fenêtre parent, ce qui n'est pas forcement pratique, puisque ça oblige à définir les couleurs au niveau de la fenêtre parent pour tous les contrôles présents. Avant les MFC 4.0 on n'avait pas le choix.

Pour améliorer le traitement des couleurs Microsoft a ajouté avec les MFC 4.0 la notion de message **reflected** qui permet de définir la couleur au niveau du contrôle, nécessitant par la même de faire une classe dérivée du contrôle de base pour traiter le message.

Cette technique permet la combinaison des deux méthodes, celle de la personnalisation au niveau du parent avec **WM_CTLCOLOR** et celle du message **reflected** au niveau du contrôle, le message au niveau du contrôle étant préfixé d'un **=**

Pour la couleur on aura **= WM_CTLCOLOR** dans **ClassWizard** sur une classe de type contrôle

Note : Au final il sera donc normal que le message **WM_CTLCOLOR (**simple sans le =**)** ne fonctionne pas au niveau du contrôle.

La fonction de réponse pour ce message sera :

HBRUSH CtlColor(CDC* pDC, UINT nCtlColor)

Je propose ci-dessous un traitement plus évolué car il permet grâce à la définition d'une classe template de gérer les couleurs pour les contrôles en une seule fois :

Note : sa compréhension dépasse largement le niveau de connaissances acquises jusqu'ici, il ne faut donc pas s'affoler ….

```
// include
//////////////////////////////////////////////////////////////////////
// Classe Template attributs couleurs
template <class GENERIC_CTRLCOLOR>
class CTplCtrl : public GENERIC_CTRLCOLOR
{
// Construction
public:
   CTplCtrl()
   {
      m_arClrCtlText[0]=::GetSysColor(COLOR_WINDOWTEXT);
      m_arClrCtlText[1]=RGB(0  ,0  ,255); // LtBlue
      m_arClrCtlText[2]=RGB(128,0,0);    // Red.
      m_arClrCtlBkText[0]=::GetSysColor(COLOR_WINDOW);
      m_arClrCtlBkText[1]=::GetSysColor(COLOR_WINDOW);
      m_arClrCtlBkText[2]=::GetSysColor(COLOR_WINDOW);
      for(int i=0;i<3;i++)
      m_arHbrClrCtlBk[i]=::CreateSolidBrush(m_arClrCtlBkText[i]);
   }

  enum ModeColor
  {
   Normal,
   Disable,
   ReadOnly
  };

   void SetBkColor(COLORREF clrCtlBk = RGB(192, 192, 192), // couleur de fond
                   COLORREF clrCtlText = RGB(0, 0, 0),                 // couleur d'écriture.
                   ModeColor eMode=Normal)                             // mode actif/Inactif/lecture seule.
   {
      m_arClrCtlText[eMode]=clrCtlText;
      m_arClrCtlBkText[eMode]=clrCtlBk;
      if(m_arHbrClrCtlBk[eMode])
         ::DeleteObject(m_arHbrClrCtlBk[eMode]);
      m_arHbrClrCtlBk[eMode] = ::CreateSolidBrush(clrCtlBk);
      if(m_hWnd) Invalidate();
   }

// Attributes
public:

   HBRUSH    m_arHbrClrCtlBk[3]; // brush de fond
   COLORREF  m_arClrCtlBkText[3];// couleur du fond.
   COLORREF  m_arClrCtlText[3];  // couleurs d'ecriture.

// Operations
public:

   virtual ~CTplCtrl()
   {
      for(int i=0;i<3;i++)
      if(m_arHbrClrCtlBk[i]) ::DeleteObject(m_arHbrClrCtlBk[i]);
```

```
    };

    afx_msg HBRUSH CtlColor(CDC* pDC, UINT nCtlColor)
    {
        bool bCEdit=IsKindOf(RUNTIME_CLASS(CEdit));
        HBRUSH hbr=NULL;
        ModeColor eMode=Normal;
        if(GetStyle() & ES_READONLY) eMode=ReadOnly;
        if(!IsWindowEnabled()) eMode=Disable;

        // TODO: Change any attributes of the DC here
        pDC->SetTextColor(m_arClrCtlText[eMode]);

        // Fixe le fond en transparent  pour le texte
        if(!bCEdit) pDC->SetBkMode(TRANSPARENT);
        else pDC->SetBkColor(m_arClrCtlBkText[eMode]);

        // retourne le handle de la brush pour le fond si il existe.
        if(m_arHbrClrCtlBk[eMode]) hbr = m_arHbrClrCtlBk[eMode];

        // TODO: Return a different brush if the default is not desired
        return hbr;
    }

    virtual BOOL OnChildNotify( UINT message, WPARAM wParam, LPARAM lParam, LRESULT*
pLResult )
    {
        // interception du message reflect
        if(message >= WM_CTLCOLORMSGBOX && message <= WM_CTLCOLORSTATIC)
        {
            UINT nCtlType = message - WM_CTLCOLORMSGBOX;
            ASSERT(nCtlType >= CTLCOLOR_MSGBOX);
            ASSERT(nCtlType <= CTLCOLOR_STATIC);

            CDC dcTemp; dcTemp.m_hDC = (HDC)wParam;

            HBRUSH hbr = CtlColor(&dcTemp, nCtlType);
            // fast detach of temporary objects
            dcTemp.m_hDC = NULL;
            *pLResult = (LRESULT)hbr;
           return TRUE;
        }
        return GENERIC_CTRLCOLOR::OnChildNotify( message,wParam, lParam,pLResult );
    }
};
```

Utilisation :

```
class CSampleSDIView : public CFormView
{
protected: // create from serialization only
	CSampleSDIView();
	DECLARE_DYNCREATE(CSampleSDIView)

public:
	//{{AFX_DATA(CSampleSDIView)
	enum { IDD = IDD_SAMPLESDI_FORM };
	CButton	m_ButtonRAZ;
	CTplCtrl<CEdit>	m_EditNom;
```

Classwizard n'appréciant pas ce genre de déclaration on pourra écrire la chose suivante :

```
Typedef CTplCtrl<CEdit> CEditEx;
```

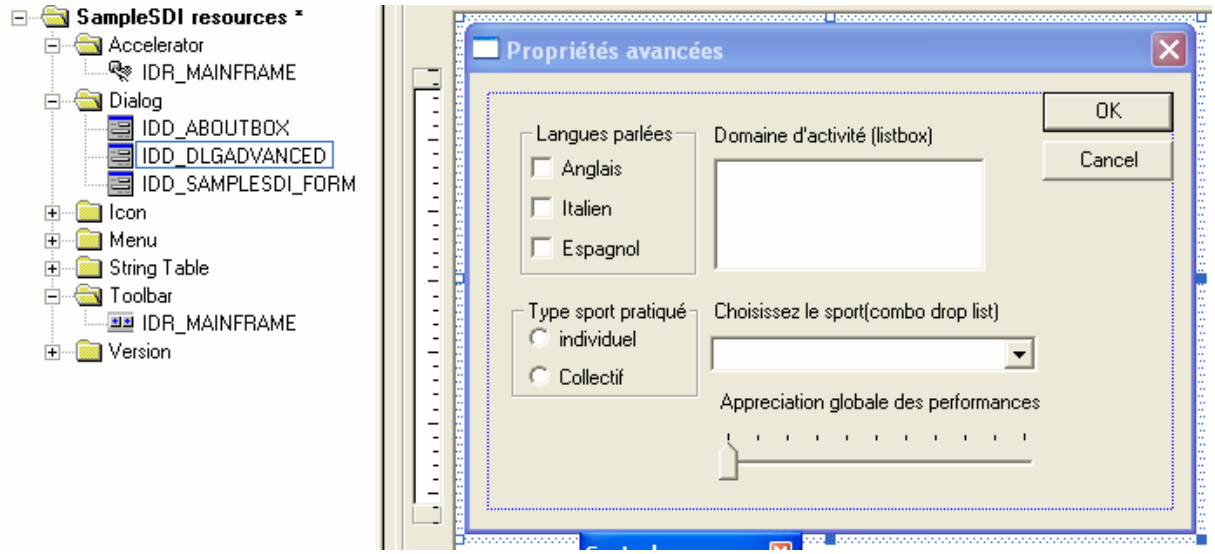
Et utiliser **CEditEx** comme nouvelle classe à la place de **CEdit**. On pourra faire de même pour un **Cstatic.** Pour changer la couleur il suffira d'appeler la fonction **SetBkColor** dans OnInitialUpdate, par exemple.

Les Boîtes de dialogue

Définition :
Toutes les applications Windows utilisent des boites de dialogues comme par exemple à chaque fois que vous voulez sauvegarder, imprimer un document dans Word, une boîte de dialogue apparaît.
Il existe deux types de boîte de dialogue : les dialogues modales ou non modales.
Une boite de dialogue modale empêche l'utilisateur de sortir ou d'aller sur une autre fenêtre tant qu'il n'a pas validé ou abandonné le traitement. Au contraire, une boite de dialogue non modale n'est pas bloquante.

Application :

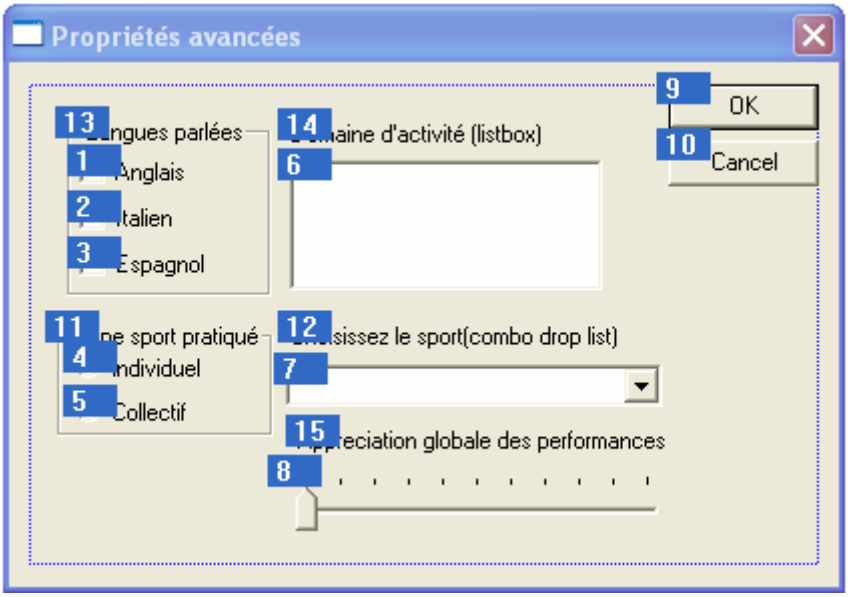
Nous allons définir une boite de dialogue modale avec un panel de contrôles supplémentaire. Pour cela allez dans l'éditeur de ressources section « **dialogs »** et faire clic droit **insert dialog**.
Je l'ai nommée **IDD_DLGADVANCED**



- **Les contrôles utilisés pour l'exemple sont :**

Trois cases à cocher (Check Box)
Deux boutons radio (radio button)
Une Liste de sélection (ListBox)
Une boite de sélection (Combo Box)
Un curseur (Slider Ctrl)
Les cases à cocher et les boutons radios sont entourés d'un contrôle groupe (group box)

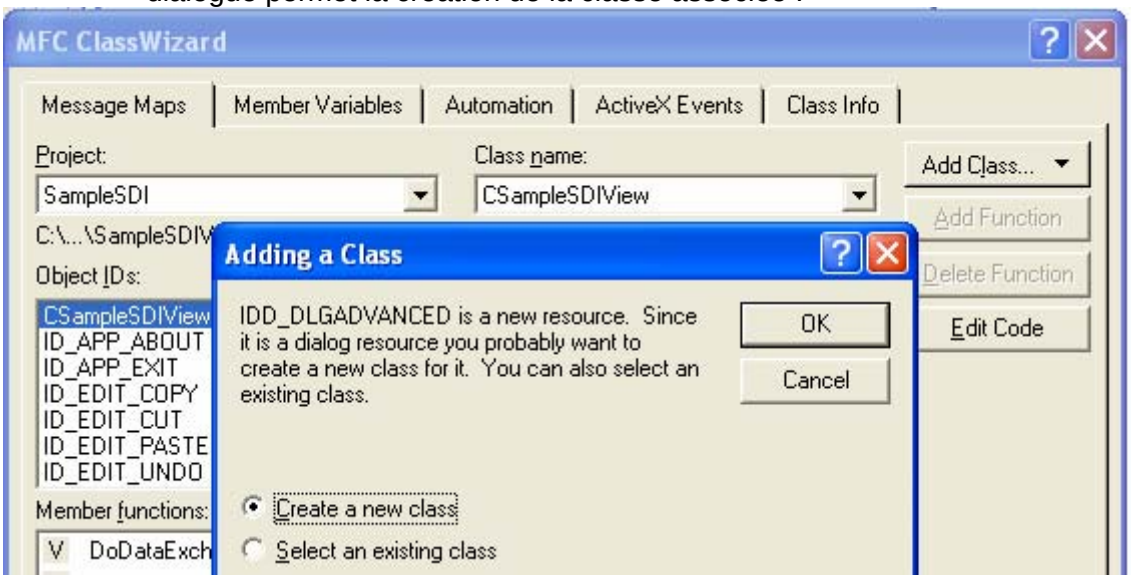
➢ **Réglage de l'ordre de tabulation :**

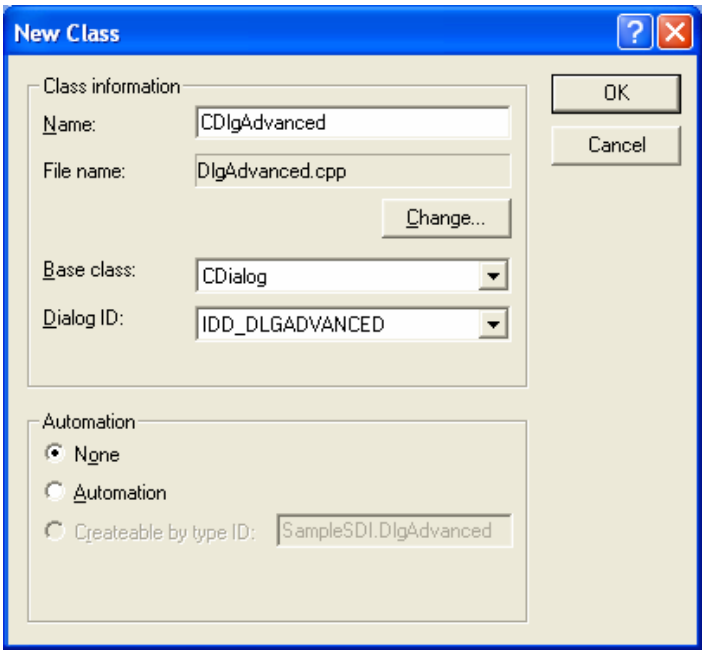


Note : mettre l'option groupe sur le bouton radio **« individuel »**.

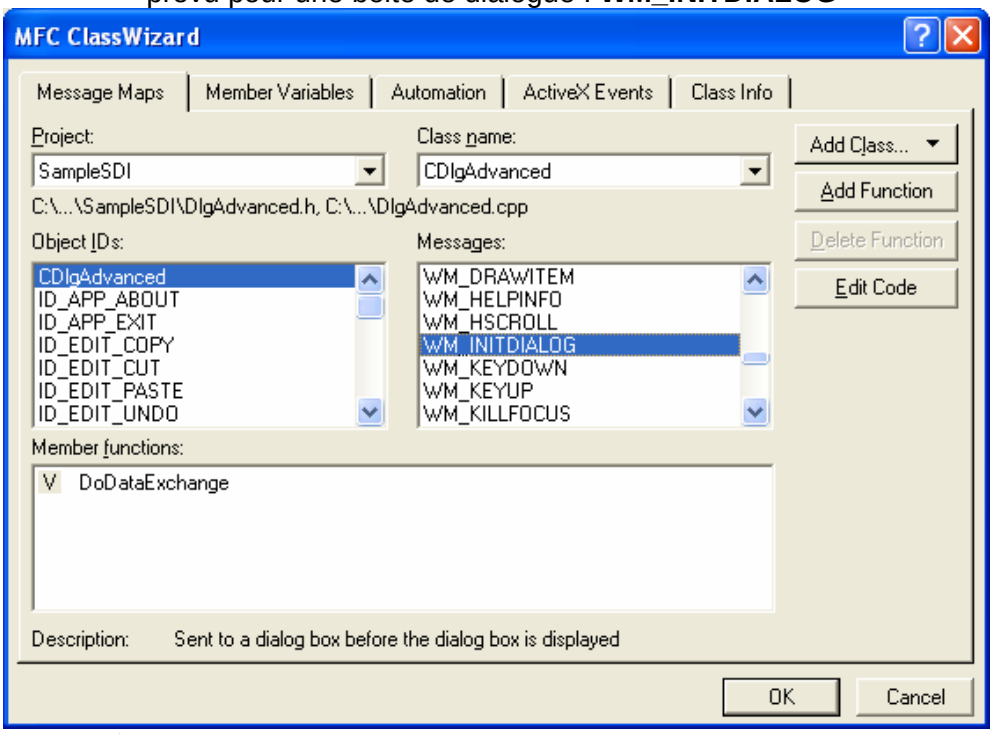
➢ **Création de la classe associée a la boîte de dialogue :**

✓ L'appel de **classwizard** par un double clic sur une zone libre sur la boîte de dialogue permet la création de la classe associée :



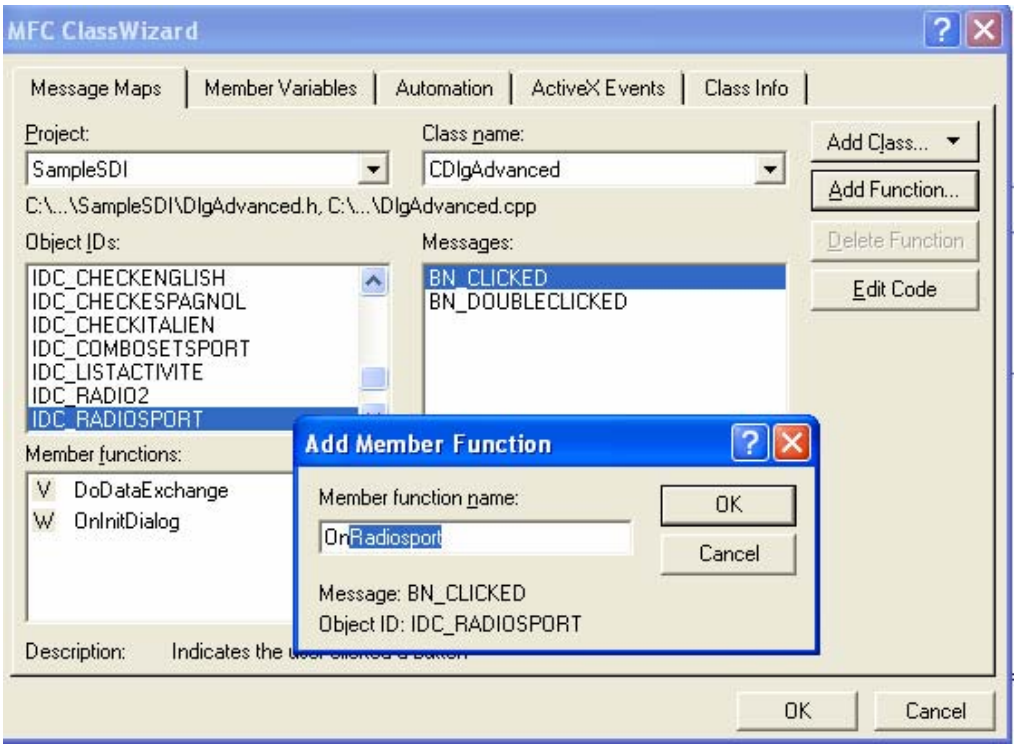


- ✓ Saisissez le nom de la classe : **CDlgAdvanced** puis valider sa création par la touche OK.
- ✓ Continuons le paramétrage de notre classe en générant le message d'initialisation prévu pour une boite de dialogue : **WM_INITDIALOG**



- ✓ Rajoutez la fonction avec le bouton **« add Function »**
- ✓ Interceptez le clic sur le bouton IDOK

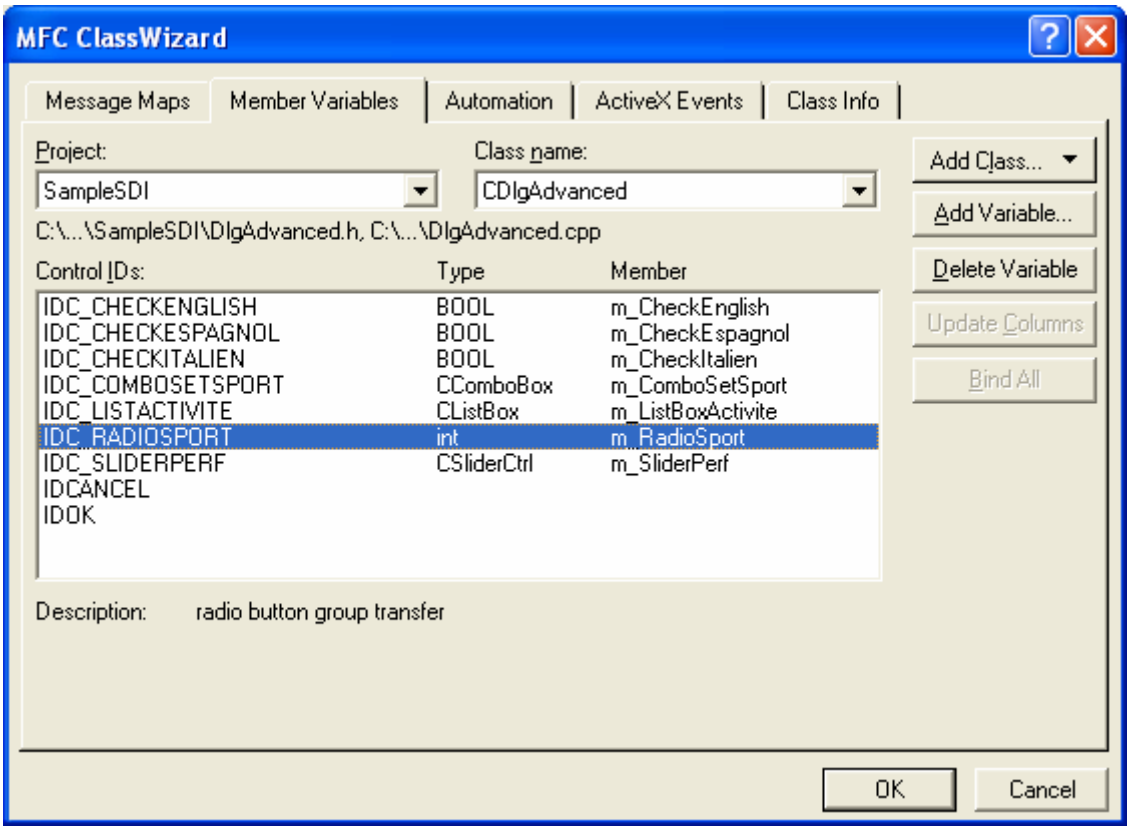
- ✓ Interceptez le message **BN_CLICKED** sur le bouton radio :



Faites de même avec le deuxième radio en donnant la même fonction de réponse **OnRadiosport**.

Création des variables attachées aux contrôles :

- ✓ Associez les variables suivantes aux contrôles



- ✓ Rajoutez dans le .h de la classe les variables :

```
CString    m_strSport;    // pour le sport pratique
CString    m_strActivite; // pour l'activité
int        m_nPerf;       // note globale de 0 a 20
```

```
class CDlgAdvanced : public CDialog
{
// Construction
public:
        CDlgAdvanced(CWnd* pParent = NULL);   // standard constructor
// Dialog Data
        //{{AFX_DATA(CDlgAdvanced)
        enum { IDD = IDD_DLGADVANCED };
        CSliderCtrl     m_SliderPerf;
        CListBox        m_ListBoxActivite;
        CComboBox       m_ComboSetSport;
        BOOL    m_CheckEnglish;
        BOOL    m_CheckEspagnol;
        BOOL    m_CheckItalien;
        int             m_RadioSport;
        //}}AFX_DATA
        void    InitComboSport(); // initialisation de la combobox
        void    SetComboSport(const char *szSport); // sélection de la combobox
public:
   CString    m_strSport;    // pour le sport pratique
   CString    m_strActivite; // pour l'activité
   int        m_nPerf;       // note globale de 0 a 20
```

- ➢ **L'initialisation des contrôles :**

```
BOOL CDlgAdvanced::OnInitDialog()
{
        CDialog::OnInitDialog();

    // TODO: Add extra initialization here
   char *aszActivite[]={"lycéen","Etudiant","chômeur","Ouvrier","Cadre","indépendant"};

   for(int i=0;i<sizeof(aszActivite)/sizeof(char *);i++)
      m_ListBoxActivite.AddString(aszActivite[i]);

   int nIndex=m_ListBoxActivite.FindStringExact(m_strActivite);
   if(nIndex!=LB_ERR) m_ListBoxActivite.SetCurSel(nIndex);

   InitComboSport();
   SetComboSport(m_strSport);

        return TRUE;  // return TRUE unless you set the focus to a control
                      // EXCEPTION: OCX Property Pages should return FALSE
}
void CDlgAdvanced::SetComboSport(const char *szSport)
{
   int nIndex=m_ComboSetSport.FindStringExact(0,szSport); // recherche le libelle szSport
   if(nIndex!=LB_ERR) m_ComboSetSport.SetCurSel(nIndex);  // sélectionne la ligne dans la
combobox
}
void CDlgAdvanced::InitComboSport()
{
   UpdateData(TRUE); // mise a jour des datas

   m_ComboSetSport.ResetContent(); // efface le contenu de la combobox

   if(m_RadioSport<=0) // chargement de la liste en fonction du radio.
   {
      char *aszSport[]={"Escrime","Natation","Course a pied","Fitness","Autre"};

      for(int i=0;i<sizeof(aszSport)/sizeof(char *);i++)
         m_ComboSetSport.AddString(aszSport[i]);
   }
   else
   { // sports collectifs ==1
      char *aszSport[]={"Football","HandBall","VoleyBall","Fitness","Autre"};

      for(int i=0;i<sizeof(aszSport)/sizeof(char *);i++)
         m_ComboSetSport.AddString(aszSport[i]);
   }
}

void CDlgAdvanced::OnRadiosport()
{
  // TODO: Add your control notification handler code here
   InitComboSport();
   m_strSport="";
   m_ComboSetSport.SetCurSel(0);
}
```

Dans la fonction **InitDialog** j'initialise :

- ✓ La **listbox** en insérant les libellés et en sélectionnant la ligne avec le libellé affecté à **m_strActivite.**
- ✓ La **combobox** avec la fonction **InitComboSport,** puis la sélection de la ligne est faite en fonction du contenu de la variable **m_strSport** avec la fonction **SetComboSport**

Le fait de cliquer sur un des boutons radios permet le changement dynamique de la **combobox** avec la fonction **InitComboSport() (**voir fonction **OnRadiosport).**

- ➢ **Traitement de l'acceptation de la Boîte de dialogue :**

  Le traitement est effectué dans la fonction **OnOK** :
  Il suffira d'appeler **UpdateData(TRUE)** pour affecter les variables avec les différentes valeurs choisies.

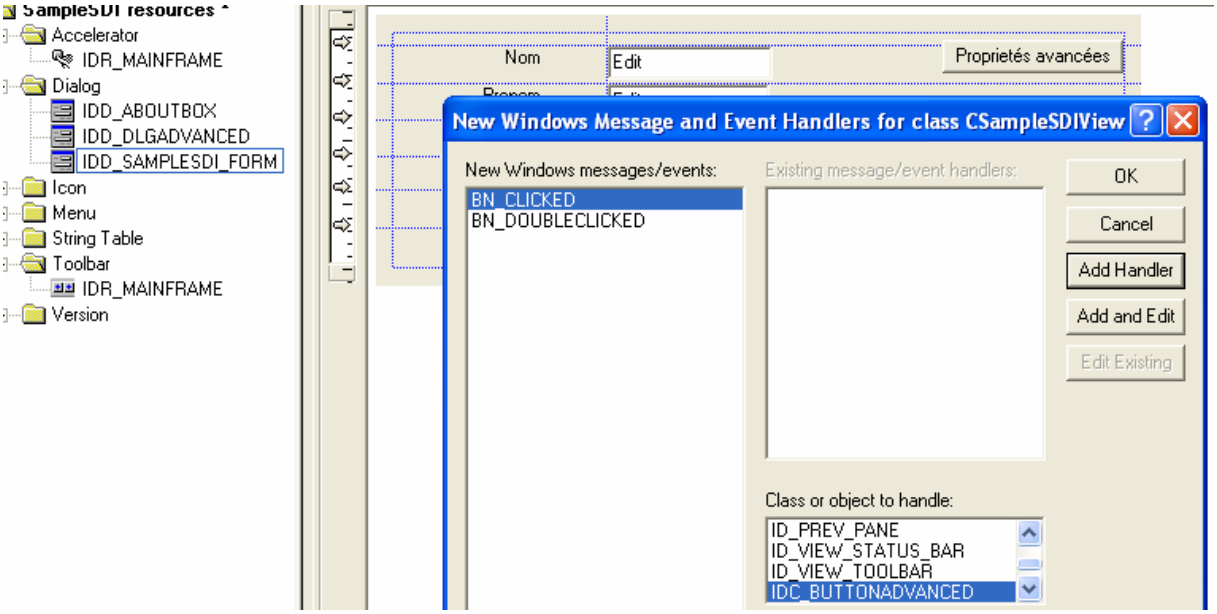
```
void CDlgAdvanced::OnOK()
{
   // TODO: Add extra validation here
   UpdateData(TRUE);
   int nIndexCombo=m_ComboSetSport.GetCurSel();
   if(nIndexCombo==LB_ERR) return ; // pas de selection == pas bon

   int nIndexList =m_ListBoxActivite.GetCurSel();
   if(nIndexList==LB_ERR) return ; // pas de selection == pas bon

   // Récupération des libellés sélectionnés.
   m_ComboSetSport.GetLBText(nIndexCombo,m_strSport);
   m_ListBoxActivite.GetText(nIndexList,m_strActivite);

   CDialog::OnOK();
}
```

On notera l'utilisation des méthodes **GetLBText et GetText** pour la **CComboBox** et la **CListBox** afin de récupérer le libellé en cours de sélection.

Si une des deux sélections n'est pas valide le traitement d'acceptation de la boîte de dialogue est refusé, empêchant sa fermeture.

➢ **Implémentation de l'appel de la boîte de dialogue dans la vue :**

✓ Rajouter un bouton « propriétés avancées » dans la vue et interceptez le message **BN_CLICKED** :



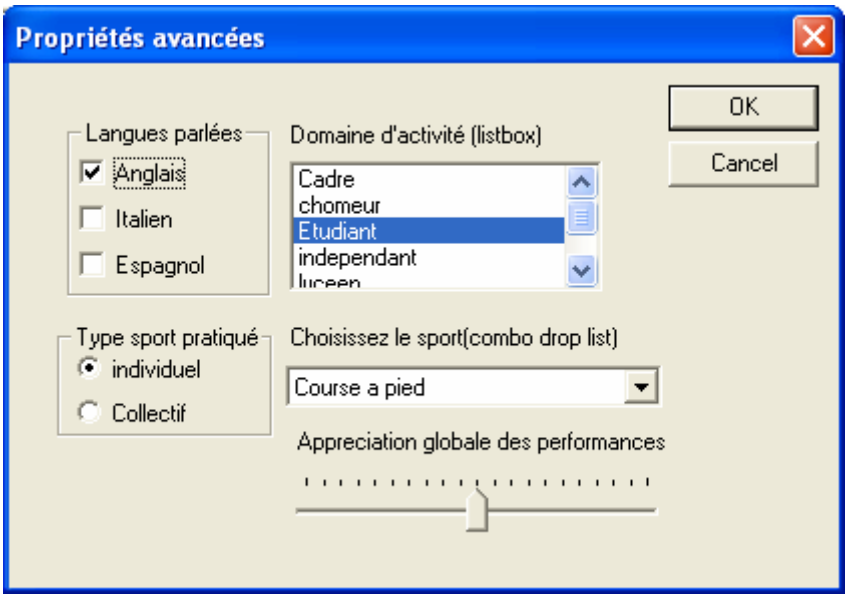
✓ Rajoutez : **#include "DlgAdvanced.h"** dans le source de la classe **CSampleSDIView.**

➢ **Appel de la boite de dialogue :**

Il ne reste plus qu'à implémenter l'appel de notre boîte de dialogue dans la vue:

```
void CSampleSDIView::OnButtonadvanced()
{
  // TODO: Add your control notification handler code here
   CDlgAdvanced Dlg;

   Dlg.m_RadioSport=0;
   Dlg.m_strActivite="Etudiant";
   Dlg.m_strSport="Course a pied";
   Dlg.m_nPerf=10; // 10/20
   Dlg.m_CheckEnglish=1;

   if(Dlg.DoModal()==IDOK)
   {
      // sauvegarde des éléments (à faire) .
      afxDump << "Sport " << Dlg.m_strSport <<"\n";
      afxDump << "Activite " << Dlg.m_strActivite << "\n";
      afxDump << "anglais  " << Dlg.m_CheckEnglish <<"\n";
      afxDump << "italien  " << Dlg.m_CheckItalien <<"\n";
      afxDump << "Espagnol " << Dlg.m_CheckEspagnol <<"\n";
      afxDump << "Performance " << Dlg.m_nPerf <<"\n";
   }
}
```

- **Le résultat :**



La boîte de dialogue est déclarée localement à la méthode **OnButtonadvanced** .
Les différentes valeurs sont initialisées avant l'appel de la méthode **DoModal** qui lance la boîte de dialogue.

Cette méthode renverra en sortie de traitement soit la valeur **IDOK** ou **IDCANCEL**.
En cas de succès il ne reste plus qu'à sauvegarder les valeurs récupérées grâce à la fonction **OnOk**.

Généralités sur les boîtes de dialogues

Une boite de dialogue modale pourra être utilisée chaque fois que l'on désire que le traitement proposé soit exclusif c'est-à-dire que l'utilisateur ne puisse faire autre chose dans le programme à cet instant .

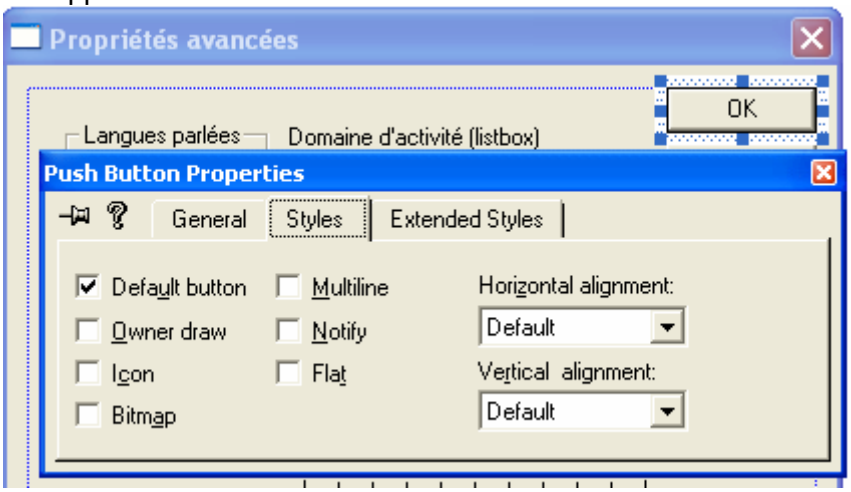
Les premières opérations sur les contrôles doivent être faites dans la fonction **OnInitDialog** après la fonction qui appelle la méthode de la classe de base .**CDialog :: OnInitDialog** qui les initialisera graphiquement

Toutes tentatives d'accès avant cette initialisation se solderont par une assertion d'erreur. Après **DoModal** les contrôles n'existent plus, une tentative d'accès sur les contrôles provoquera aussi une assertion d'erreur.

La sauvegarde des informations dans des variables doit être faite avant destruction de la boîte de dialogue, la fonction virtuelle **OnOk** est l'endroit approprié pour ce traitement.

Les boites de dialogues supportent mal la réutilisation du même objet, il est préférable d'utiliser un objet unique comme dans notre exemple où l'objet était local a la fonction.

Aspect qui peut être gênant : l'utilisation de la touche entrée ou échappement provoque la fermeture de la boîte de dialogue et la valeur IDOK ou IDCANCEL est envoyée.
Lors de l'appui sur la touche entrée le bouton disposant de la propriété : bouton par défaut est appelé :



Pour intercepter la touche entrée ou échappement on peut procéder comme suit : on surchargera la méthode **OnCommand** de la classe **CWnd**:

CWnd::OnCommand virtual BOOL OnCommand(WPARAM wParam,LPARAM lParam);

```
BOOL CDlgAdvanced::OnCommand(WPARAM wParam, LPARAM lParam)
{
            // TODO: Add your specialized code here and/or call the base class
           CWnd *pWnd = GetFocus();
            switch(wParam)
            {
               case IDOK: if(pWnd!=GetDlgItem(IDOK))
                        {
                                return FALSE;
                        }
                        break;
                case IDCANCEL:if(pWnd!=GetDlgItem(IDCANCEL))
                        {
                                return FALSE;
                        }
                        break;
            }
            return CDialog::OnCommand(wParam, lParam); }
}
```

Le code ci-dessus permet d'interdire la fermeture de la fenêtre par la touche entrée si le contrôle possédant le « focus » n'est pas le bouton **IDOK.**

Même chose avec la touche Echappement, la fenêtre ne pourra pas se fermer si le bouton possédant le « focus » n'est pas le bouton **IDCANCEL**.

Une petite précision : lors de la fermeture de la fenêtre par la croix le message **IDCANCEL** sera généré.

La gestion des couleurs :

Même remarque que pour les contrôles, la classe dialogue ne gère par la couleur de fond. Pour remédier à ce problème il faut intercepter le message **WM_ERASEBKGND**

```
BOOL CDlgAdvanced::OnEraseBkgnd(CDC* pDC)
{
  // TODO: Add your message handler code here and/or call default

  CBrush backBrush(m_crBackColor);//COLORREF
  CBrush *pOldBrush=pDC->SelectObject(&backBrush);
  CRect rect;
  pDC->GetClipBox(&rect);
  pDC->PatBlt(rect.left,rect.top,rect.Width(),rect.Height(),PATCOPY);
  pDC->SelectObject(pOldBrush);

  return TRUE;
}
```

La variable **m_crBackColor** doit être déclarée dans la classe avec le type **COLORREF**.
On initialisera sa valeur en utilisant la macro **RBG** dans la méthode **OnInitDialog :**

Exemple :
m_crBackColor=RGB(187,221,255);

RGB voulant dire rouge, vert, bleu, un moyen pour visualiser les couleurs : utiliser dans MS-Paint et la fonction modification des couleurs.

Le traitement de la couleur des contrôles :

Il est tout a fait possible au niveau de la boîte de dialogue de gérer en un seule fois la couleur des contrôles présent. Il existe une fonction au niveau de la classe d'application permettant de gérer pour toute l'application la couleur de fond et la couleur d'écriture du texte des contrôles :

CWinApp::SetDialogBkColor void SetDialogBkColor(COLORREF clrCtlBk = RGB(192, 192, 192), COLORREF clrCtlText = RGB(0, 0, 0));

Mais cette méthode ne permet pas un changement dynamique par boîte de dialogue, donc voici une autre méthode.

Il faut implémenter le message **WM_CTLCOLOR** sur la **CDialog**
Appliquer à notre exemple ça donne le code suivant :

```
class CDlgAdvanced : public CDialog
{
// Construction
public:
        CDlgAdvanced(CWnd* pParent = NULL);   // standard constructor
   ~CDlgAdvanced();
// Dialog Data
        //{{AFX_DATA(CDlgAdvanced)
        enum { IDD = IDD_DLGADVANCED };
        CSliderCtrl     m_SliderPerf;
        CListBox        m_ListBoxActivite;
        CComboBox   m_ComboSetSport;
        BOOL  m_CheckEnglish;
        BOOL  m_CheckEspagnol;
        BOOL  m_CheckItalien;
        int             m_RadioSport;
        //}}AFX_DATA

        void    InitComboSport();
        void    SetComboSport(const char *szSport);
        void    SetDialogBkColor(COLORREF clrCtlBk=RGB(192, 192, 192),
                  COLORREF clrCtlText=RGB(0, 0, 0));

public:
   CString    m_strSport;    // pour le sport pratique
   CString    m_strActivite;  // pour l'activite
   int       m_nPerf;       // note globale de 0 a 20
   COLORREF    m_crBackColor;
   HBRUSH     m_HbrClrCtlBk;
   COLORREF    m_ClrCtlText;
//.....
```

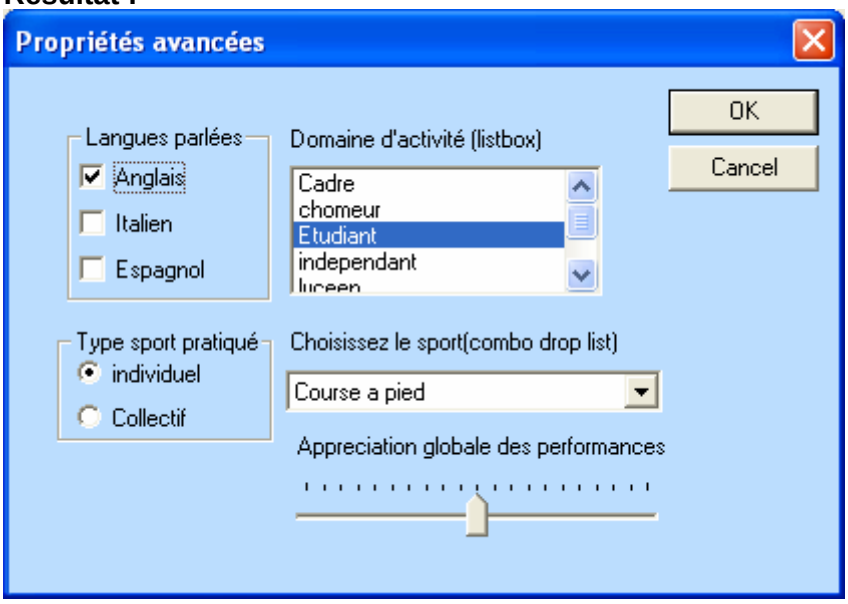
```
CDlgAdvanced::CDlgAdvanced(CWnd* pParent /*=NULL*/)
	: CDialog(CDlgAdvanced::IDD, pParent)
{
	//{{AFX_DATA_INIT(CDlgAdvanced)
	m_CheckEnglish = FALSE;
	m_CheckEspagnol = FALSE;
	m_CheckItalien = FALSE;
	m_RadioSport = -1;
	//}}AFX_DATA_INIT
  /* HBRUSH */    m_HbrClrCtlBk=NULL ;
  /* COLORREF */  m_ClrCtlText= RGB(0, 0, 0) ;

}
CDlgAdvanced::~CDlgAdvanced()
{
	// Destruct
    if(m_HbrClrCtlBk) ::DeleteObject(m_HbrClrCtlBk);
}
void CDlgAdvanced::SetDialogBkColor(COLORREF clrCtlBk /*= RGB(192, 192, 192)*/,
                                    COLORREF clrCtlText /*= RGB(0, 0, 0) */)
{
   //m_HbrClrCtlBk est à null dans le constructeur
   if(m_HbrClrCtlBk) ::DeleteObject(m_HbrClrCtlBk);
   m_HbrClrCtlBk   = ::CreateSolidBrush(clrCtlBk);
   m_ClrCtlText    = clrCtlText;
   m_crBackColor   = clrCtlBk;
   if(m_hWnd) Invalidate();
 }
//-----------------------------------------------------
BOOL CDlgAdvanced::OnEraseBkgnd(CDC* pDC)
{
	// TODO: Add your message handler code here and/or call default
  CBrush backBrush(m_crBackColor);//COLORREF
  CBrush *pOldBrush=pDC->SelectObject(&backBrush);
  CRect rect;
  pDC->GetClipBox(&rect);
  pDC->PatBlt(rect.left,rect.top,rect.Width(),rect.Height(),PATCOPY);
  pDC->SelectObject(pOldBrush);

  return TRUE;
}
//-----------------------------------------------------
HBRUSH CDlgAdvanced::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
	HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);

   /*  CTLCOLOR_BTN button control
      CTLCOLOR_DLG dialog box
      CTLCOLOR_EDIT edit control
      CTLCOLOR_LISTBOX list box
      CTLCOLOR_MSGBOX message box
      CTLCOLOR_SCROLLBAR scroll bar
      CTLCOLOR_STATIC static text, frame, or rectangle
   */
   // TODO: Change any attributes of the DC here
   // par exemple en fonction de nCtlColor voir doc.

   switch(nCtlColor)
   {
```

```
    // Intercepte le message pour es statics
    //case CTLCOLOR_DLG: deja traite dans OnEraseBkgnd
    case CTLCOLOR_STATIC : // Fixe la couleur d'ecriture du texte
        pDC->SetTextColor(m_ClrCtlText);
        // enventuellement suivant les cas
        // pDC->pDC->SetBkColor(m_ClrCtlBk);
        // Fixe le fond en transparent pour le texte
        // à ne pas faire pour un edit.
        pDC->SetBkMode(TRANSPARENT);
        // retourne le handle de la brush pour le fond si il existe.
        if(m_HbrClrCtlBk ) hbr = m_HbrClrCtlBk;
        break;
    }
    // TODO: Return a different brush if the default is not desired
    return hbr;
}
```

Dans la méthode **OnInitDialog** on mettra la ligne suivante :
SetDialogBkColor(RGB(187,221,255));

Le même code fonctionne avec une **CFormView**

Résultat :



La sérialisation des données

Dernière étape de notre projet : sauvegarder les données en utilisant le modèle de lecture/sauvegarde proposé par visual .

La lecture des données :

Lors de la génération du projet visual a généré automatiquement la commande d'ouverture de fichier avec la fonction **CWinApp::OnFileOpen** :

```
/////////////////////////////////////////////////////////////////////////////
// CSampleSDIApp
BEGIN_MESSAGE_MAP(CSampleSDIApp, CWinApp)
        //{{AFX_MSG_MAP(CSampleSDIApp)
        ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
                // NOTE - the ClassWizard will add and remove mapping macros here.
                //    DO NOT EDIT what you see in these blocks of generated code!
        //}}AFX_MSG_MAP
        // Standard file based document commands
        ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
        ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
        // Standard print setup command
        ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

- ✓ Cette fonction permet la sélection d'un fichier
- ✓ Elle appelle la fonction **CDocument ::OnOpenDocument** pour l'objet document en cours .

  La fonction virtuelle **CDocument ::DeleteContents** est appelée pour la libération d'objets du document en cours.

  Si vous devez libérer des objets existants dans la vue ou le document c'est dans cette fonction qu'il faudra le faire et non dans le destructeur de la vue.

  Celui-ci sera réservé pour la destruction d'objets non concernés par l'ouverture d'un fichier de données.

  Un objet de sérialisation **CArchive** est crée en mode lecture et la fonction **serialize** du document est appelée.

- ✓ La fonction **OnInitialUpdade** de la vue est appelée.

La sauvegarde des données:

Il faudra intercepter la commande **OnFileSave** de la classe Document ,qui appelle **OnSaveDocument** de **CDocumen**t qui appel à son tour la fonction **serialize** du document avec un objet **CArchive** initialisé pour la sauvegarde.

La notification de modification de données :

La classe document dispose d'une variable indiquant si les données utilisateur ont été modifiées.
Lorsque l'utilisateur ferme l'application une boîte de dialogue demande si on désire sauvegarder les données.
Cette variable est gérée par les fonctions :

CDocument::SetModifiedFlag
void SetModifiedFlag(BOOL bModified = TRUE);

et la fonction :

CDocument::IsModified
BOOL IsModified( );

SetModifiedFlag permet de spécifier que les données sont modifiées ou non .
IsModified permet de connaître l'état des données.

Avant de mettre ces notions en application dans notre projet quelques explications s'imposent :

Les MFC fournissent des classes de traitement de fichier :

- La classe **CFile** qui utilise directement les apis 32 Windows et qui fournit les opérations de base pour la gestion d'un fichier.
- La classe **CStdioFile** qui correspond à la fonction du C **fopen** .
- Et enfin pour gérer la sérialisation de données nous avons la classe **CArchive** qui utilisée avec un objet **CFile** permet l'archivage de données et d'objets.

La notion d'archivage d'objets est très puissante et correspond à la notion de **marshalling** de données, c'est-à-dire la faculté de transformer ce qui n'est pas directement enregistrable dans un fichier en une suite d'octets, le mécanisme de lecture devant être capable du processus inverse c'est-à-dire la reconstruction de l'objet.

Pour bien comprendre la puissance de ce mécanisme je vous propose un petit détour pour illustrer ce concept :

Sérialisation de collection d'objets en mémoire :

Les classes MFC proposent des objets permettant de maintenir des collections d'objets en mémoire. Classiquement on retrouvera des objets gérant des tableaux, des listes, et des maps. On trouve dans les MFC deux générations de ces objets :

- **CObArray , CObList, CMapPtrToWord**
  Déclinés avec des string, word, int, etc..

- **CArray, CList ,CMap** (avec des classes templates)

  il est préférable d'utiliser ces classes templates en remplacement des précédentes.

L'exemple qui suit :
- ✓ Déclare un objet à sérialiser
- ✓ Définit un tableau de cet objet.
- ✓ Effectue son remplissage, sa sauvegarde et sa lecture avec la classe **CArchive**.

```
// header .....

// la classe qui contient les données
class CItem : public CObject
{

public:
   DECLARE_SERIAL( CItem )

   CItem(){Clear();}
   //----------------------------
   ~CItem(){}
   //----------------------------
    // constructeur de copie
   CItem(const CItem &rItem)
   {
      CopyFrom(rItem);
   }
   //----------------------------
   // operateur d'affectation
   const CItem& operator=(const CItem& Src)
   {
      CopyFrom(Src);
      return *this;
   }
   //----------------------------
   // effacer les données.
   void Clear()
   {
      m_strNom="";
      m_strPrenom="";
      m_strCdp="";
      m_strVille="";
   }
   //----------------------------
   // la methode de serialisation.
```

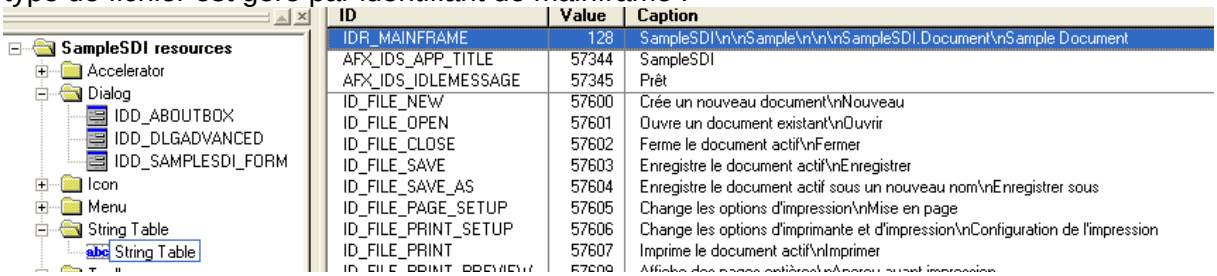
```
    void Serialize(CArchive& ar)
    {
        if(ar.IsStoring())
            ar << m_strNom << m_strPrenom << m_strCdp <<m_strVille;
        else
            ar >> m_strNom >> m_strPrenom >> m_strCdp >> m_strVille;
    }
    //-----------------------------
    // copier les données d'une source.
    void CopyFrom(const CItem & Src )
    {
        if(this==&Src) return;

        Clear(); // clear eventuel.

        m_strNom    =Src.m_strNom;
        m_strPrenom =Src.m_strPrenom;
        m_strCdp    =Src.m_strCdp;
        m_strVille  =Src.m_strVille;
   }
    //-----------------------------
    // dump pour test
    CString GetStrDump()
    {
        return ( m_strNom + "/" +m_strPrenom + "/" + m_strCdp +"/" + m_strVille);
    }

    // les données.
    CString m_strNom;
    CString m_strPrenom;
    CString m_strCdp;
    CString m_strVille;
};

// la gestion d'un tableau de cette classe .
typedef CArray<CItem,CItem&> CArrayItem;
```

```
// Source.....................
//---------------------------------------------------------------------------
// définition de la méthode de sérialisation de l'objet CItem pour le template CArray.
template <> static void AFXAPI SerializeElements <CItem> ( CArchive& ar,
    CItem* pItem, int nCount )
{
    for ( int i = 0; i < nCount; i++, pItem++ )
        pItem->Serialize( ar );
}

IMPLEMENT_SERIAL( CItem, CObject, 0)

void Test()
{
// un élément
   CItem item;

    item.m_strCdp="06800";
```

```
    item.m_strNom="farscape";
    item.m_strPrenom="???";
    item.m_strVille="Nice";

   // un tableau de l'élément
    CArrayItem arItem;

     // remplissage.
    arItem.Add(item);
    for(int i=0;i<10;i++)
    {
       item.m_strNom.SetAt(0,'A'+i);
       arItem.Add(item);
    }
     // archivage.
    {
       CFile File;
       if(File.Open("MyArchive.arc", CFile::modeCreate | CFile::modeWrite ))
       {
          CArchive ar( &File, CArchive::store);
          arItem.Serialize(ar);
       }
    }
    // Lecture de l'archive.
    arItem.RemoveAll();
    CFile File;
    if(File.Open("MyArchive.arc", CFile::modeRead ))
    {
       CArchive ar( &File, CArchive::load);
       arItem.Serialize(ar);
    }
    // verification du contenu.
    for(i=0;i<arItem.GetSize();i++)
    {
       AfxMessageBox(arItem[i].GetStrDump());
    }

}
```

Nous avons vu que les MFC fournissent une méthode de lecture/écriture des données grâce au modèle document vue.

Ce modèle va jusqu'à la description de l'extension du fichier géré par l'application. En fait, le type de fichier est géré par identifiant de mainframe :



En fait le paramétrage du type de fichier est à renseigner au moment de la génération du programme dans les propriétés avancées.

A chaque nIDResource associée à un document template on trouve dans l'éditeur de ressources dans la « stringtable » la chaîne d'ouverture pour le document.

Exemple:
Pour l'id IDR_MAINFRAME nous avons :

SampleSDI\n\nSample\n\n\nSampleSDI.Document\nSample Document

Description:

IDR_MAINFRAME <windowTitle>\n<docName>\n<fileNewName>\n
<filterName>\n <filterExt>\n<regFileTypeID>\n
<regFileTypeName>\n <filterMacExt(filterWinExt)>\n
<filterMacName(filterWinName)>

Que nous allons remplacer par :
IDR_MAINFRAME SampleSDI \n\nSample\nFichiers (*.SDIS)\n.SDI\n
SampleSDI.Document\nSample Document

A l'ouverture j'aurai bien : **Fichiers *.SDIS**

Enregistrement de l'extension de fichier dans la base de registre :

Comment faire pour que lorsque l'on double-clique sur le document géré par notre application, celle-ci s'exécute ? Il faut enregistrer l'extension du fichier dans la base de registre. Pour un programme MFC il y a une fonction qui fait ce travail :

void RegisterShellFileTypes(BOOL bCompat = FALSE);

La valeur bCompat a TRUE permet l'insertion des entrées impression et impression sur dans la base de registre ainsi que le sélectionner glisser lâcher du document sur une imprimante.

Cette fonction est à insérer derrière le dernier AddocTemplate dans la fonction InitInstance de la classe d'application.

La fonction **EnableShellOpen** doit être appelée.

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CSampleSDIDoc),
        RUNTIME_CLASS(CMainFrame),      // main SDI frame window
        RUNTIME_CLASS(CSampleSDIView));
AddDocTemplate(pDocTemplate);
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);

 EnableShellOpen();
 RegisterShellFileTypes();
```

Mise en place du code de lecture ecriture de nos données :

Nous allons commencer par mettre en place des données supplémentaires pour sauvegarder la valeur des contrôles de notre boîte de dialogue :

```
public:
        //{{AFX_DATA(CSampleSDIView)
        enum { IDD = IDD_SAMPLESDI_FORM };
        CButton        m_ButtonRAZ;
        CTplCtrl<CEdit>      m_EditNom;
        CString        m_StrNom;
        CString        m_strPrenom;
        CString        m_strVille;
        CString        m_strCdp;
        CString        m_strAdresse2;
        CString        m_strAdresse;
        //}}AFX_DATA

    CString m_strSport;        // sport
    CString m_strActivite;    //Activite
    int     m_CheckEnglish;   //anglais
    int     m_CheckItalien ;    //italien
    int     m_CheckEspagnol;//Espagnol
    int     m_nPerf;                //Performance
```

Ajouter une methode de sauvargarde dans notre vue :

```
void CSampleSDIView::Serialize(CArchive& ar)
{
    // sauvegarde
    if (ar.IsStoring())
    {
      ar << m_StrNom << m_strPrenom << m_strVille << m_strCdp << m_strAdresse
<<m_strAdresse2;
      ar << m_strSport << m_strActivite << m_CheckEnglish << m_CheckEspagnol <<
m_nPerf;
    }
    else
    {
      // lecture
      ar >> m_StrNom >> m_strPrenom >> m_strVille >> m_strCdp >> m_strAdresse >>
m_strAdresse2;
      ar >> m_strSport >> m_strActivite >> m_CheckEnglish >> m_CheckEspagnol >>
m_nPerf;
    }
}
```

Modifier le code d'appel de notre boîte de dialogue :

```
void CSampleSDIView::OnButtonadvanced()
{
        // TODO: Add your control notification handler code here
   CDlgAdvanced Dlg;

   /*
   Dlg.m_RadioSport=0;
   Dlg.m_strActivite="Etudiant";
   Dlg.m_strSport="Course a pied";
   Dlg.m_nPerf=10; // 10/20
   Dlg.m_CheckEnglish=1;
   */

   Dlg.m_strSport     =m_strSport;
   Dlg.m_strActivite   =m_strActivite;
   Dlg.m_CheckEnglish  =m_CheckEnglish;
   Dlg.m_CheckEspagnol =m_CheckEspagnol;
   Dlg.m_nPerf        =m_nPerf;

   if(Dlg.DoModal()==IDOK)
   {
      // sauvegarde des elements.
#ifdef _DEBUG
      afxDump << "Sport " << Dlg.m_strSport <<"\n";
      afxDump << "Activite " << Dlg.m_strActivite << "\n";
      afxDump << "anglais  " << Dlg.m_CheckEnglish <<"\n";
      afxDump << "italien  " << Dlg.m_CheckItalien <<"\n";
      afxDump << "Espagnol " << Dlg.m_CheckEspagnol <<"\n";
      afxDump << "Performance " << Dlg.m_nPerf <<"\n";
#endif
      m_strSport     =Dlg.m_strSport;
      m_strActivite   =Dlg.m_strActivite;
      m_CheckEnglish  =Dlg.m_CheckEnglish;
      m_CheckEspagnol =Dlg.m_CheckEspagnol;
      m_nPerf        =Dlg.m_nPerf;
   }
}
```

Mettre entre commentaire le message ID_FILE_SAVE que nous avons intercepté au niveau de la vue :

```
BEGIN_MESSAGE_MAP(CSampleSDIView, CFormView)
	//{{AFX_MSG_MAP(CSampleSDIView)
	ON_BN_CLICKED(IDC_BUTTONOK, OnButtonok)
	ON_EN_CHANGE(IDC_EDITNOM, OnChangeEditnom)
	ON_EN_CHANGE(IDC_EDITPRENOM, OnChangeEditprenom)
	ON_EN_CHANGE(IDC_EDITADRESSE2, OnChangeEditadresse2)
	ON_EN_CHANGE(IDC_EDITADRESSE, OnChangeEditadresse)
	ON_BN_CLICKED(IDC_BUTTONRAZ, OnButtonraz)
	//ON_COMMAND(ID_FILE_SAVE, OnFileSave) laisser faire le document
	ON_UPDATE_COMMAND_UI(ID_FILE_SAVE, OnUpdateFileSave)
	ON_BN_CLICKED(IDC_BUTTONADVANCED, OnButtonadvanced)
	//}}AFX_MSG_MAP
	// Standard printing commands
	ON_COMMAND(ID_FILE_PRINT, CFormView::OnFilePrint)
	ON_COMMAND(ID_FILE_PRINT_DIRECT, CFormView::OnFilePrint)
	ON_COMMAND(ID_FILE_PRINT_PREVIEW, CFormView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

Et enfin modifier la fonction de serialisation de la classe Document :

```
/////////////////////////////////////////////////////////////////////////////
// CSampleSDIDoc serialization

void CSampleSDIDoc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
		// TODO: add storing code here
	}
	else
	{
		// TODO: add loading code here
	}
   POSITION pos = GetFirstViewPosition();
   CSampleSDIView* pView = static_cast<CSampleSDIView *>(GetNextView(pos));
   pView->Serialize(ar);

}
```

J'ai redirigé la serialisation du document sur la fonction définie dans notre vue.

Il ne reste plus qu'à essayer.
Autre test : placer une copie de notre programme dans le volume **c:\windows** et double cliquer sur un fichier généré par notre application à travers l'explorateur windows. Le programme doit s'ouvrir…

Le débuggage d'une application

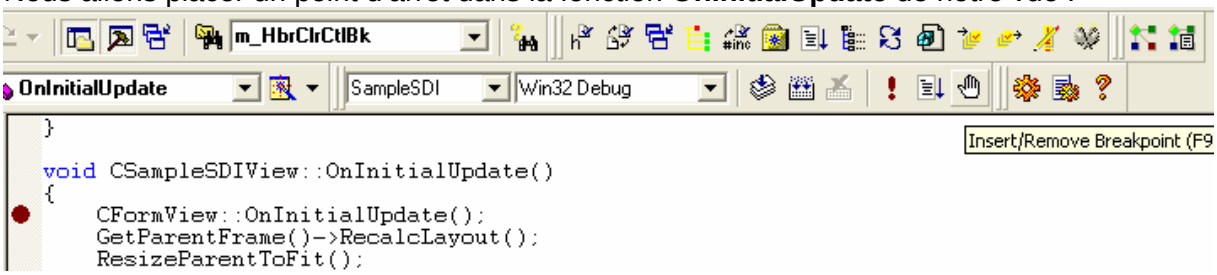
Visual dispose d'un debbuger intégré à l'environnement de développement,
Lorsque l'application est compilée en Debug nous pouvons tracer pas à pas le code de notre application et :

- Inspecter les différentes variables
- Modifier les variables.
- Modifier le code de l'application et continuer son exécution
- Mettre des points d'arrêts dans le code .

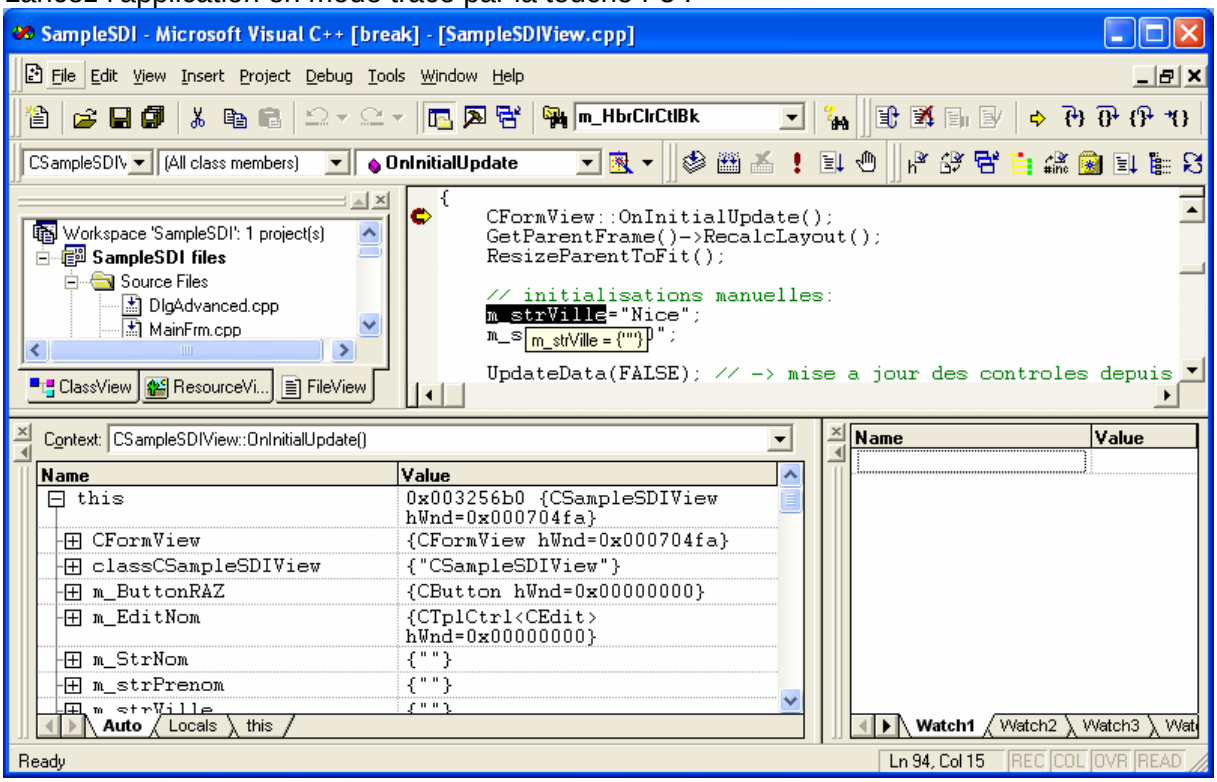
Tour d'horizon des fonctionnalités :

Compiler et linker notre projet d'exemple en mode Debug :

Nous allons placer un point d'arrêt dans la fonction **OnInitialUpdate** de notre vue :

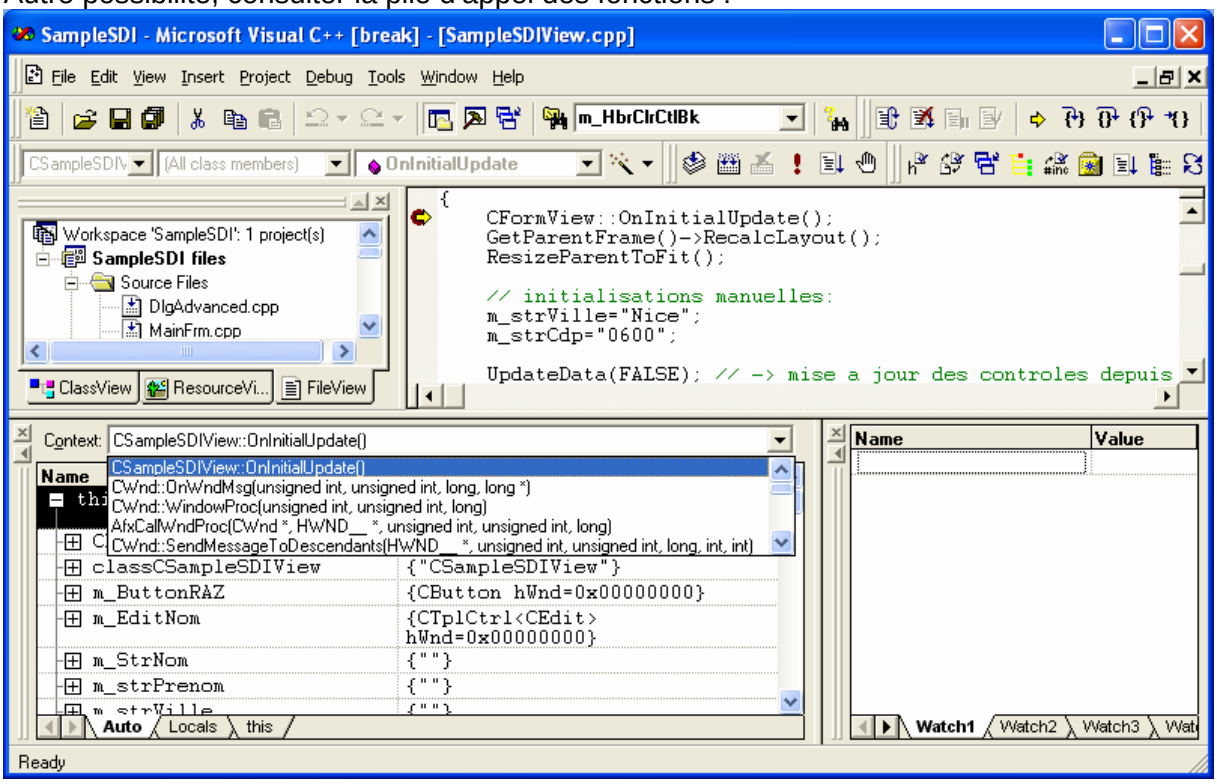


Placez le curseur sur la ligne **CFormView ::OnInitialUpdate** comme sur l'écran ci-dessus .
Cliquez sur l'icône représentant un main (à côté de la bulle Insert/remove breakpoint).
Lancez l'application en mode trace par la touche F5 .



Sur l'écran ci-dessus nous voyons les principales fonctionnalités du debugger :
L'inspection du contenu d'un variable sélectionnée ici **m_strVille**.

Le panneau contexte permet de voir le contenu de l'objet en cours **this** (notre vue).
Toutes les variables sont consultables.
Nous pouvons aussi par un glisser déplacer placer un variable sélectionnée dans le panneau de droite (watch) permettant ainsi de la surveiller tout au long de sa portée.
Autre possibilité, consulter la pile d'appel des fonctions :



Cette fonctionnalité est très importante et utile, car en cas d'erreur signalée par une assertion MFC, Il suffit de remonter la pile d'appels jusqu'à trouver une fonction correspondant à notre code pour trouver l'origine du problème.

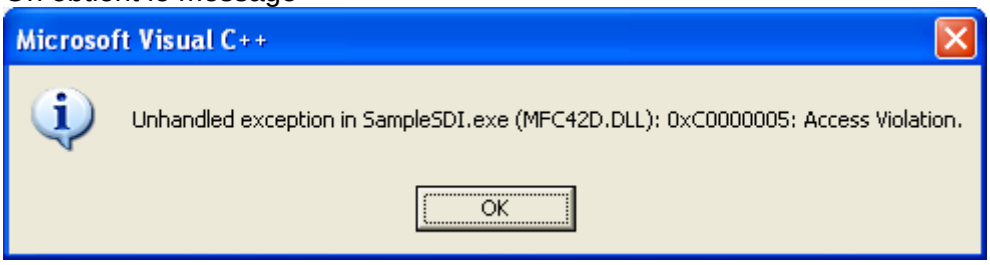
Exemples :

Nous allons provoquer deux erreurs courantes des débutants, une concernant l'allocation mémoire, l'autre concernant la gestion des contrôles.

Insérez le code suivant dans la fonction **OninitialUpdate** de notre vue :

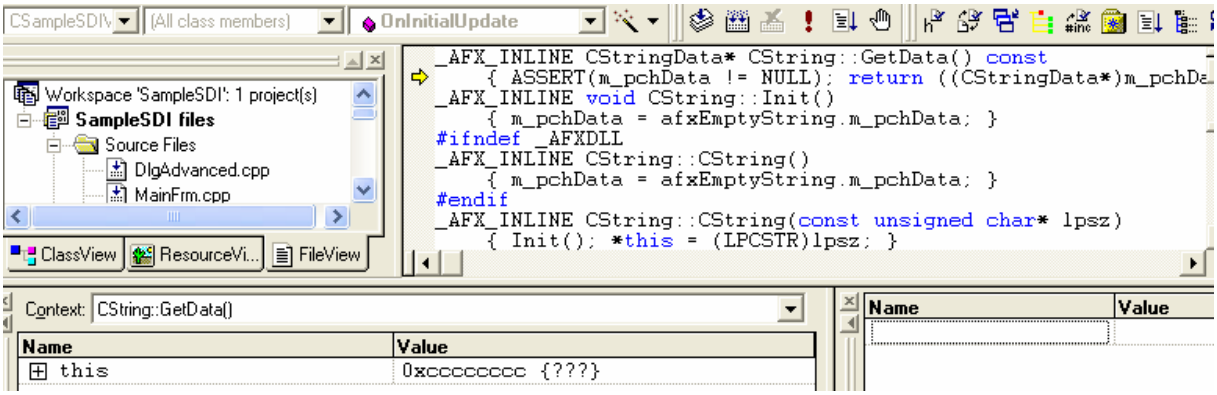
```
CString *pString ;
*pString="coucou" ;
```

Lancez l'exécution du programme par F5 .
On obtient le message

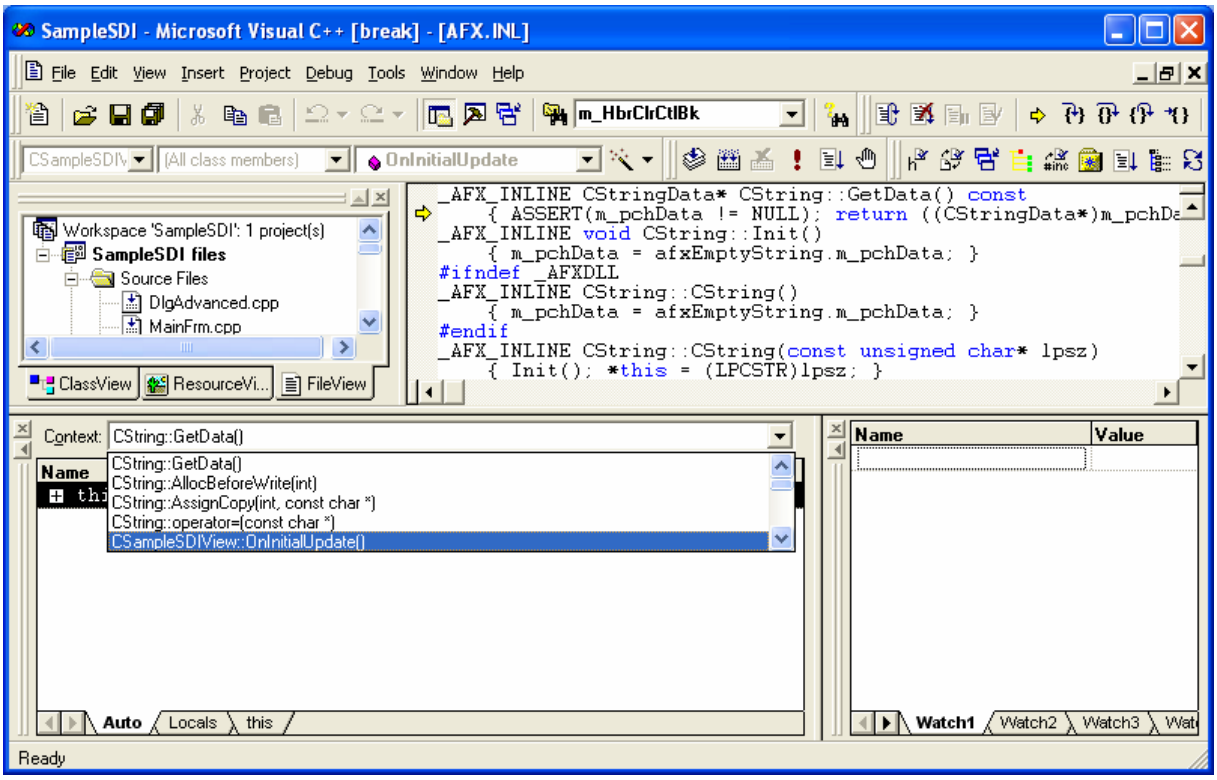


Pressez le bouton OK

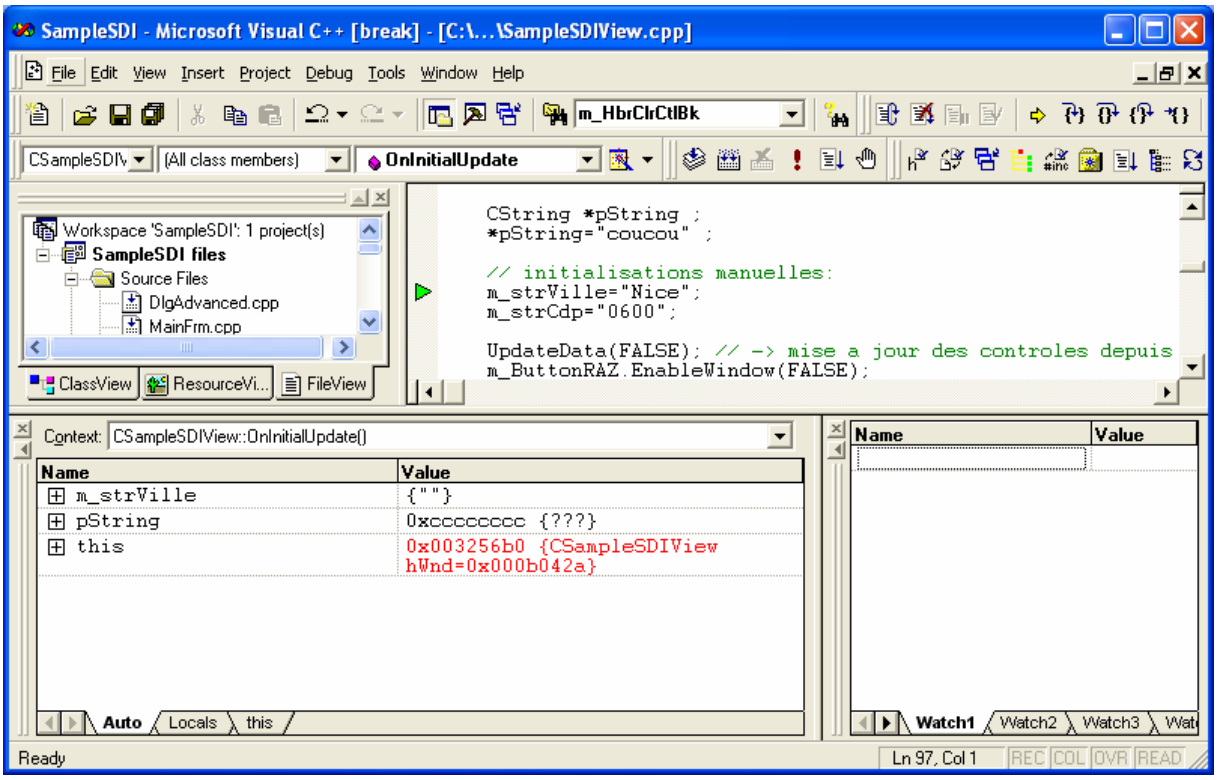
Vous obtenez :



Nous tombons donc directement dans du code MFC, comme trouver la portion de notre code causant le problème ? Il suffit de remonter la pile des appels :



pour arriver directement sur la première fonction issue de notre code, qu'il suffit de sélectionner :



Le curseur est placé sur la ligne immédiatement après l'erreur.
Un autre point significatif de l'erreur : la valeur du pointeur **pstring** dans la fenêtre **context** .

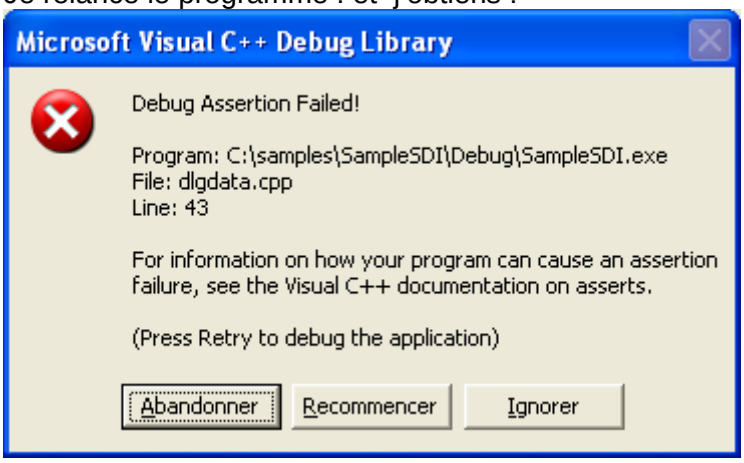
Explications :
Le compilateur suivant les cas à l'exécution peut affecter des valeurs spécifiques à une variable pointeur.

Connaître ces valeurs peut s'avérer fort utile en mode Debug :

- **0xFDFDFDFD :** indique une adresse en dehors de l'espace d'adressage du processus.
- **0xDDDDDDDD:** indique que la mémoire vient d'être libérée.
  Attention ça ne veut pas dire que l'instruction delete va mettre cette valeur dans un pointeur.
- **0xCDCDCDCD :** mémoire globale non initialisée par exemple un pointeur membre d'une classe.
- **0xCCCCCCCC :** mémoire locale (pile /stack) non initialisée par exemple un pointeur déclaré dans une fonction membre d'une classe.

On comprend l'utilité de toujours initialiser les pointeurs à **NULL**, et de leur affecter cette valeur après libération de la mémoire.
Dans notre exemple **pstring** contient la valeur reconnaissable **0xcccccccc** et le debugger met des « ??? » en face.

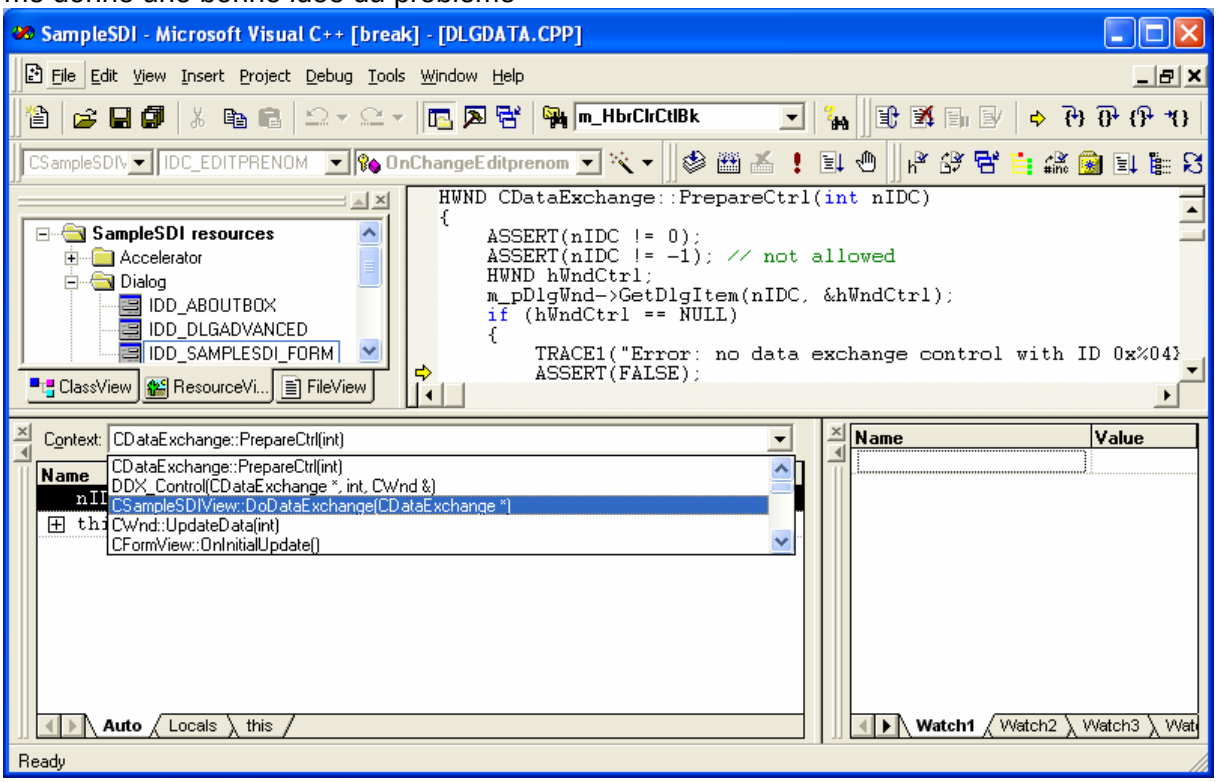
Autre exemple :
Je supprime un contrôle de notre vue, et ce contrôle dispose d'une variable membre associée.
Je relance le programme : et j'obtiens :



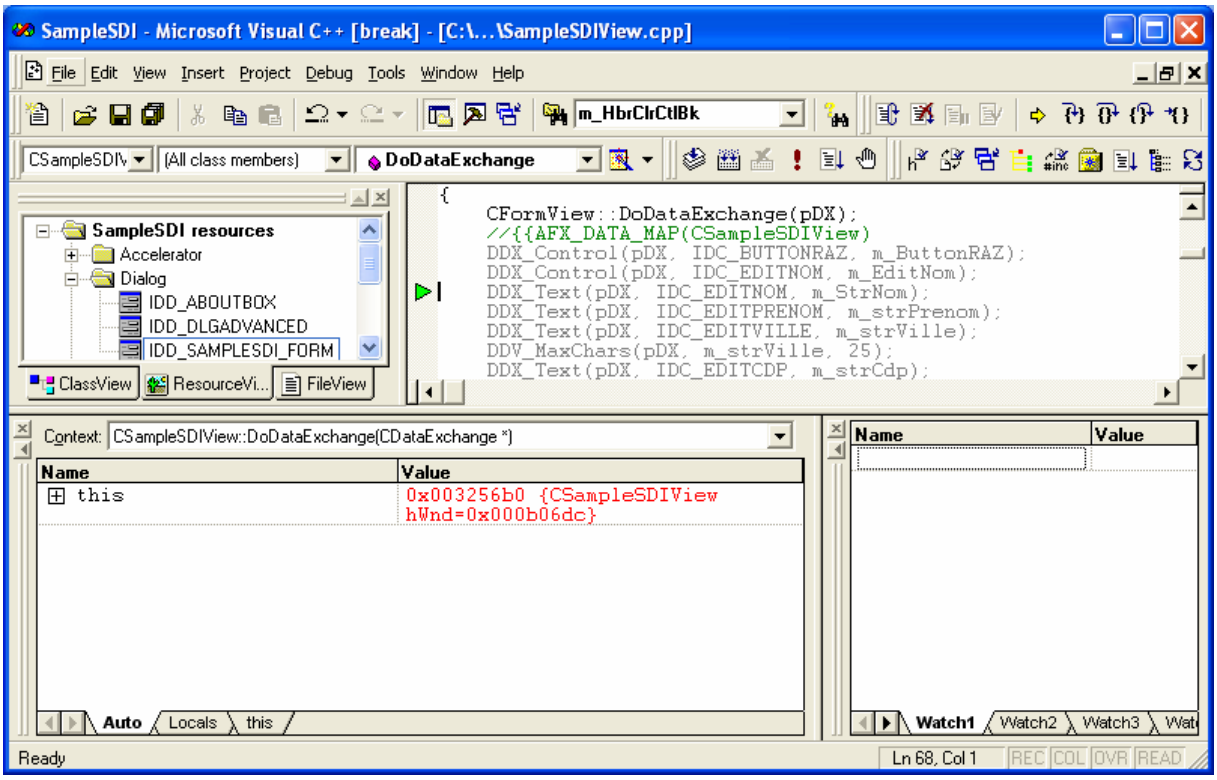
Je presse le bouton recommencer pour debugger :
Je tombe à nouveau sur du code MFC :

```
HWND CDataExchange::PrepareCtrl(int nIDC)
{
    ASSERT(nIDC != 0);
    ASSERT(nIDC != -1); // not allowed
    HWND hWndCtrl;
    m_pDlgWnd->GetDlgItem(nIDC, &hWndCtrl);
    if (hWndCtrl == NULL)
    {
        TRACE1("Error: no data exchange control with ID 0x%04X.\n", nIDC);
        ASSERT(FALSE);
        AfxThrowNotSupportedException();
    }
```

Je procède de la même manière en remontant la pile des appels, même si la ligne **TRACE1** me donne une bonne idée du problème



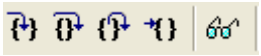
Cette fois si je tombe dans notre fonction **DodataExchange** :



La variable causant le problème est **m_EditNom**.

Note :
Si vous regardez bien la pile des fonctions dans la zone contexte, on retrouve l'enchaînement des fonctions qui est décrit pour l'explication de **UpdateData** (voir écran précédent)

Visual permet aussi de se déplacer pas à pas dans le code en cours de débuggage :



Description dans l'ordre des icônes :

- ✓ Entre dans le code de la fonction où est positionné le curseur.
- ✓ Exécute la fonction sans rentrer dans le code.
- ✓ Sort de la fonction en cours pour passer à la prochaine ligne qui suit.
- ✓ Exécute jusqu'à la position du curseur.
- ✓ Visualise le contenu d'une variable.

Autre possibilité intéressante : utiliser la macro TRACE.

TRACE fonctionne de la même manière que **printf** du C, les messages apparaissent dans l'onglet «debug» de visual .

Exemple tiré de MSDN:

```
int i = 1;
char sz[] = "one";
TRACE( "Integer = %d, String = %s\n", i, sz );
```

Il existe un autre méthode pour laisser des traces dans le panneau debug : **afxDump**

Exemple tiré de MSDN :

```
CPerson myPerson = new CPerson;
// set some fields of the CPerson object...
//..
// now dump the contents
#ifdef _DEBUG
afxDump << "Dumping myPerson:\n";
myPerson->Dump( afxDump );
afxDump << "\n";
#endif
```

afxDump est actif uniquement en mode debug.

Et enfin on peut aussi utiliser la fonction **AfxMessageBox** pour afficher des messages dans une boîte de dialogue.

C'est souvent cette méthode qui sera retenue en release pour traquer une erreur.
Il suffira de mettre cette fonction dans les endroits clefs du programme pour se rapprocher au plus près du code produisant l'erreur.

Exemple :

```
int i = 1;
char sz[] = "one";
CString str;
Str.Format( "Integer = %d, String = %s", i, sz );
AfxMessageBox(str);
```

Conclusions :

A travers cet exemple nous avons vu les bases d'une application MFC :

- L'architecture SDI d'un programme MFC
- Les messages Windows
- L'éditeur de ressources
- La notion de contrôles
- La communication avec les contrôles
- L'implémentation d'une boîte de dialogue modale
- La serialisation de données et la reconnaissance de l'extension fichier par Windows
- L'utilisation du debugger

Le lien sur le projet final : http://farscape.developpez.com/Samples/SampleSDI.zip

Remerciements :

Je remercie toute l'équipe du forum Visual C++ pour leur relecture attentive du document, notamment nico-pyright(c) et bigboomshakala.

Je remercie également Anomaly pour la correction orthographique.