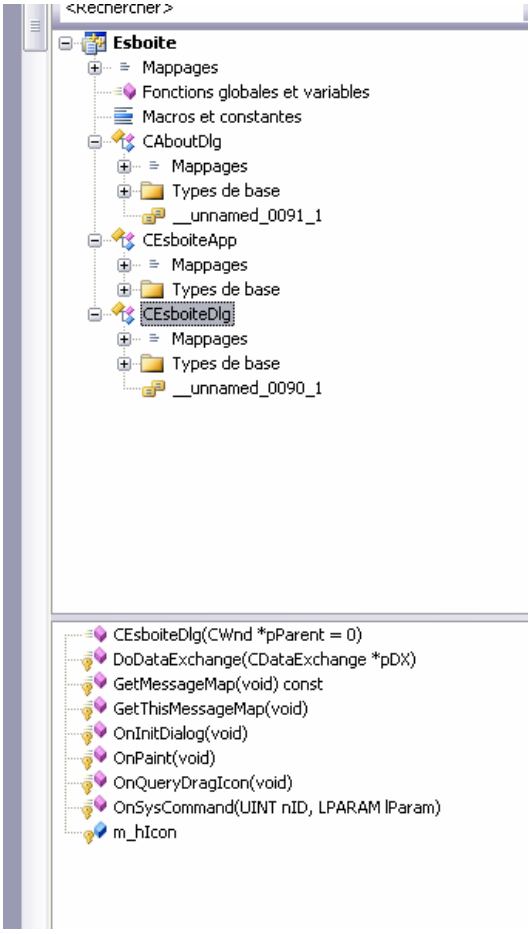
APPLICATION BASEE SUR UNE BOITE DE DIALOGUE

1. Création de l'application

Pour créer ce type d'application, on utilise l'assistant générateur de solution, qui va faire tout le travail d'initialisation pour nous.
Dans le menu principal du visual, on choisit *Fichier->Nouveau->Projet*, puis on choisit le bon répertoire dans « Emplacement" (\users\nom) et on donne un nom au projet (Esboîte dans l'exemple ci-dessous), puis on sélectionne *MCF ->Application MFC*.

Dans l'assistant (après un clic sur *Suivant*), on sélectionne comme type d'application, le bouton "***Basée sur des boites de dialogue***", puis on clique sur *Suivant* jusqu'à la fin (pour garder toutes les options par défaut). Après création du projet, la partie de gauche de l'écran ressemble à l'image ci dessous (si l'on clique sur les + pour développer les branches de l'arbre ainsi représenté)

Les onglets (partie gauche de l'écran) permettent de visualiser soit les classes présentes dans le projet, soit les ressources (menus, boîtes de dialogues dessinées…), soit les fichiers inclus dans le projet, soit encore le gestionnaire de propriétés. L'affichage des classes est pratique, car il permet de visualiser immédiatement les fonctions membres (en rose) et variables membres (en bleu) d'une classe, ainsi que leur degré de protection (public, …). Il est très facile de rajouter ainsi une variable ou fonction membre, en sélectionnant la classe, puis en cliquant sur le bouton de droite de sa souris. Les modifications portent automatiquement sur le fichier ".h" et sur le fichier ".cpp" correspondants à la classe choisie.



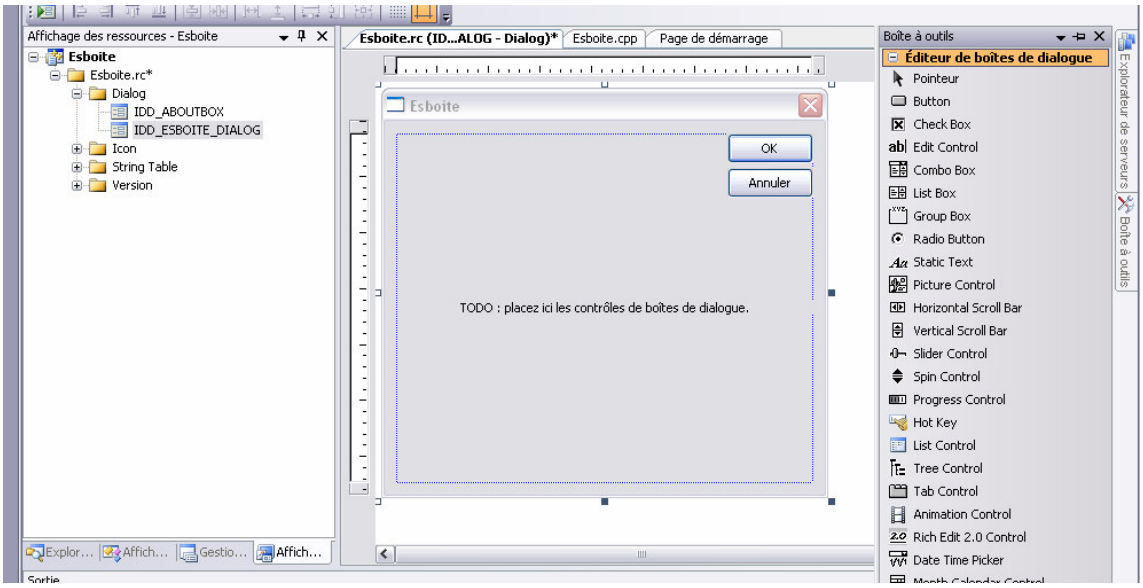
On remarque un objet déclaré de manière globale (voir Fonctions globales et variables) : TheApp. C'est lui qui initialise le programme. C'est un objet de classe CEsboîteApp, créé automatiquement par l'assistant. Cette classe est dérivée de CwinApp, et la première fonction

appelée après le constructeur est InitInstance. C'est une fonction virtuelle : on peut donc redéfinir son contenu même s'il en existe un dans la classe de base, CwinApp. Les lignes qui nous intéressent dans cette fonction sont les suivantes :

```
CEsboîteDlg dlg;     // Création d'un objet de classe CEsboîte Dlg, c'est à dire une boîte de dialogue
m_pMainWnd = &dlg;
int nResponse = dlg.DoModal();  // Appel de la fonction membre DoModal, qui affiche la boîte de dialogue et la gère jusqu'à un clic sur annuler ou OK.
if (nResponse == IDOK)
{
      // TODO : placez ici le code définissant le comportement
      lorsque la boîte de dialogue est
      //  fermée avec OK
}
else if (nResponse == IDCANCEL)
{
      // TODO : placez ici le code définissant le comportement
      lorsque la boîte de dialogue est
      //  fermée avec Annuler
}
// Lorsque la boîte de dialogue est fermée, retourner FALSE afin de
quitter
//   l'application, plutôt que de démarrer la pompe de messages de
l'application.
return FALSE;
```

La boîte est donc créée grâce à la déclaration de l'objet (l'appel au constructeur est fait de manière interne), puis visualisée et gérée par l'appel de la fonction **DoModal**. Cette fonction existe pour notre objet, car la classe CEsboîteDlg a été dérivée de la classe **CDialog**. Or la classe CDialog comporte comme fonction membre DoModal, qui gère tout de qui se passe dans la fenêtre jusqu'au prochain OK ou Cancel. Les objets des classes dérivées peuvent donc y faire appel.

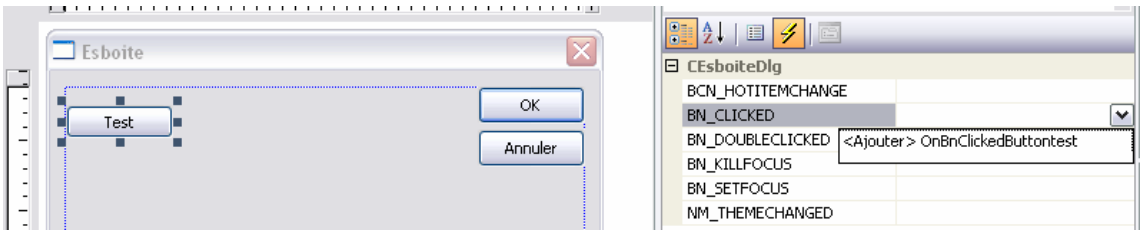
Pour savoir à quoi ressemble la boîte qui sera dessinée à l'écran, il faut éditer les Ressources du projet (voir ci-dessous)



Dans l'onglet Ressources, on trouve une arborescence qui décrit les éléments graphiques ou textuels contenus dans le projet. Les boîtes de dialogues en font partie. Tout élément a un identificateur (ID, c'est une valeur numérique constante, on utilisera donc toujours des majuscules). Celui de la boîte principale de notre application est IDD_ESBOITE_DIALOG. Il a été choisi par l'assistant, mais est modifiable à volonté. En double cliquant sur cet identificateur, on fait apparaître l'élément dans la fenêtre d'édition. On peut alors le modifier, l'agrandir, déplacer les boutons, en ajouter, c'est à dire définir l'interface homme machine. Si l'on compile et que l'on exécute le projet à ce stade, la boîte apparaîtra à l'écran, et le programme se terminera si l'on clique sur OK ou Cancel.

2. Ajout d'un bouton et interfaçage avec le programme source

Pour ajouter un bouton dans la boîte de dialogue, il suffit de choisir l'icône bouton dans la boîte d'outils située traditionnellement à droite de l'écran, puis de dessiner le bouton dans la boîte de dialogue (ou d'utiliser un glisser-déplacer).
Le bouton apparaît avec une légende par défaut "button1", que l'on modifie dans la propriété « Caption » (liste à droite). On peut le remplacer par exemple par "**test**". Il est préférable de modifier l'ID en conséquence, c'est à dire de mettre ID_BUTTONTEST au lieu de ID_BUTTON1 (en bas de liste). C'est l'ID qui va nous permettre de définir comment la boîte de dialogue doit réagir au clic sur le bouton, grâce au gestionnaire d'événements (anciennement ClassWizard).
Pour ajouter une fonction réagissant à un clic sur le bouton « Test », il suffit de double-cliquer sur le dessin du bouton en question. Ceci ajoutera à la classe EsboiteDlg une fonction membre dont le nom sera **OnBnClickedButtonTest**. Il est aussi possible d'ajouter la même fonction en sélectionnant le bouton, puis en cliquant sur l'icône « éclair », qui fait apparaître la liste des messages susceptibles d'arriver sur ce bouton. Le simple clic correspond au message BN_CLICKED.



Il est également possible de le faire en cliquant sur le bouton de droite pour faire apparaître dans le menu contextuel : « Ajouter un gestionnaire d'événement »

Le prototype de la fonction est automatiquement ajouté au fichier .h et au fichier CPP (elle apparaît dans le navigateur de classes). Pour éditer la fonction, il suffit alors de double cliquer sur le nom de la fonction dans cette zone : l'éditeur ouvre alors automatiquement l'éditeur au bon endroit dans le fichier EsboîteDlg.cpp.
Le code apparaît de la manière suivante :

```
void CEsboîteDlg::OnTest()
{
// TODO : ajoutez ici le code de votre gestionnaire de notification
de contrôle

}
```

Si l'on ajoute un message dans cette fonction :
MessageBox(_T("Bonjour"));
Le message "bonjour" apparaîtra lorsque l'on cliquera sur le bouton "test".
Pour obtenir ce résultat, le système utilise une table de message, mise automatiquement à jour par l'environnement de Visual. Cette table est visible au début du fichier EsboîteDlg.CPP

```
BEGIN_MESSAGE_MAP(CEsboiteDlg, CDialog)
	ON_WM_SYSCOMMAND()
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	//}}AFX_MSG_MAP
	ON_BN_CLICKED(IDC_BUTTONTEST, &CEsboiteDlg::OnBnClickedButtontest)
END_MESSAGE_MAP()
```

Cette table spécifie donc que le message BN_CLICKED sur l'élément dont l'ID est IDC_BUTTONTEST, envoyé lors du clic sur le bouton test, va déclencher l'appel à la fonction **`OnBnClickedButtontest`**. On y remarque d'autres messages ajoutés par défaut lors de la création du projet.
Dans le fichier EsboîteDlg.h, on remarque les lignes suivantes :

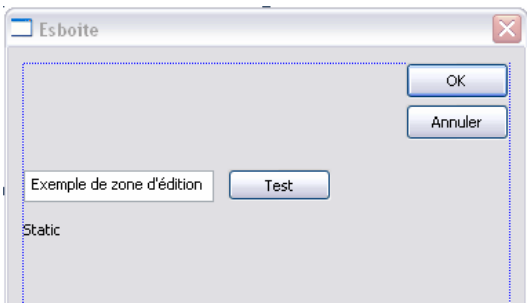
```
virtual BOOL OnInitDialog();
afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
afx_msg void OnBnClickedButtontest ();
DECLARE_EVENTSINK_MAP()
DECLARE_MESSAGE_MAP()
```

Il s'agit des déclarations des fonctions réagissant aux messages systèmes et aux messages utilisateur. La table des messages est déclarée par la macro : DECLARE_MESSAGE_MAP()

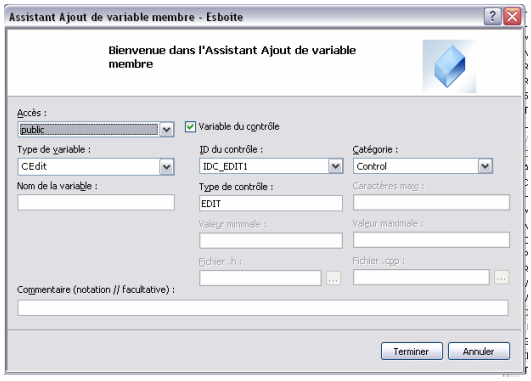
Attention : les commentaires mis par défaut et suivit de {{ ou }} dans le projet sont utilisés par Visual. Ne pas les enlever !!! Toute modification de ces lignes entraîne une impossibilité d'utiliser toutes les fonctions automatiques de Visual.

3. Ajout de contrôles et interfaçage avec le programme source

Dans l'éditeur de ressource, on peut ajouter un certain nombre de contrôle dans la boîte de dialogue. Si l'on ajoute une boîte d'édition (case permettant à l'utilisateur d'entrer un texte de manière interactive lors de l'exécution), il est très facile de récupérer le contenu de la case pour utiliser le texte entré par l'utilisateur. On ajoute donc d'abord la boîte d'édition en choisissant "edit control" dans la palette d'outil, puis un simple texte statique (static text) qui servira de contrôle.
La boîte de dialogue ressemble alors à la figure ci-contre.



Si l'on regarde les propriétés de chaque élément (bouton de droite, properties), on voit ICD_EDIT1 pour la case d'édition et IDC_STATIC pour le texte. Remplaçons IDC_STATIC par IDC_TEXTE pour la suite. Il est possible, grâce à l'assistant menu contextuel->A jouter une variable, d'ajouter des variables membres à la classe CEsboitDlg directement liées aux éléments dessinés..



Pour chaque élément d'une boite de dialogue, il est possible d'ajouter deux types de variables : soit un objet permettant de contrôler l'élément (changer sa forme, son style, récupérer des infos, etc) soit une variable permettant de fixer ou récupérer le contenu de l'élément en question.

Ici, pour la case d'édition, on souhaite une variable reflétant le contenu de la case, c'est-à-dire ce que l'utilisateur du programme aura tapé sur son clavier. Il faut donc dans la catégorie choisir « Value » et non « Control » pour ensuite choisir le type de variable. Ici on souhaite récupérer un texte, donc on utilisera « CString ». Si on veut récupérer directement un nombre entier sur 16 bits, par exemple, on choisira « short », etc.
On renseigne alors le nom de la variable : **m_Chaine**, puis on clique sur Terminer.

On peut vérifier que la variable membre a bien été ajoutée à la classe (en bleu dans l'onglet des classes).
On constate également en grisé l'ajout des lignes suivantes dans EsboîteDlg.cpp

```
void CEsboîteDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        DDX_Text(pDX, IDC_EDIT1, m_Chaine);
}
```

Cette fonction **DoDataExchange** sera automatiquement appelée à l'initialisation de la boîte de dialogue, et à chaque appel de la fonction membre **UpdateData.** La variable membre m_Chaine prendra alors la valeur de la chaîne éditée par l'utilisateur. Pour le vérifier, on peut appeler la fonction **UpdateData** dans la fonction OnButtonTest, et recopier m_Chaine dans le texte statique, en ajoutant une variable membre associée à IDC_TEXTE : m_Texte.
La fonction OnTest devient :

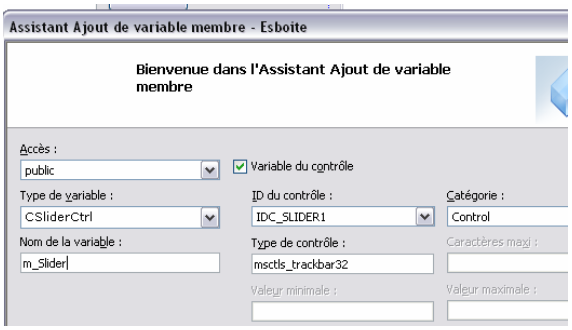
```
void CEsboîteDlg:: OnBnClickedButtontest ()
{
        UpdateData(TRUE); // recopie du texte de la boîte d'edition dans m_Chaine
        m_Texte=m_Chaine; // modification du texte satique
        UpdateData(FALSE); // actualisation de l'affichage de la boite de dialogue

}
```

La boîte d'édition est un élément sur lequel nous n'avons pas de contrôle. L'objet est créé de manière interne. Si lon veut le contrôler de manière plus poussée, on peut ajouter à l'aide de

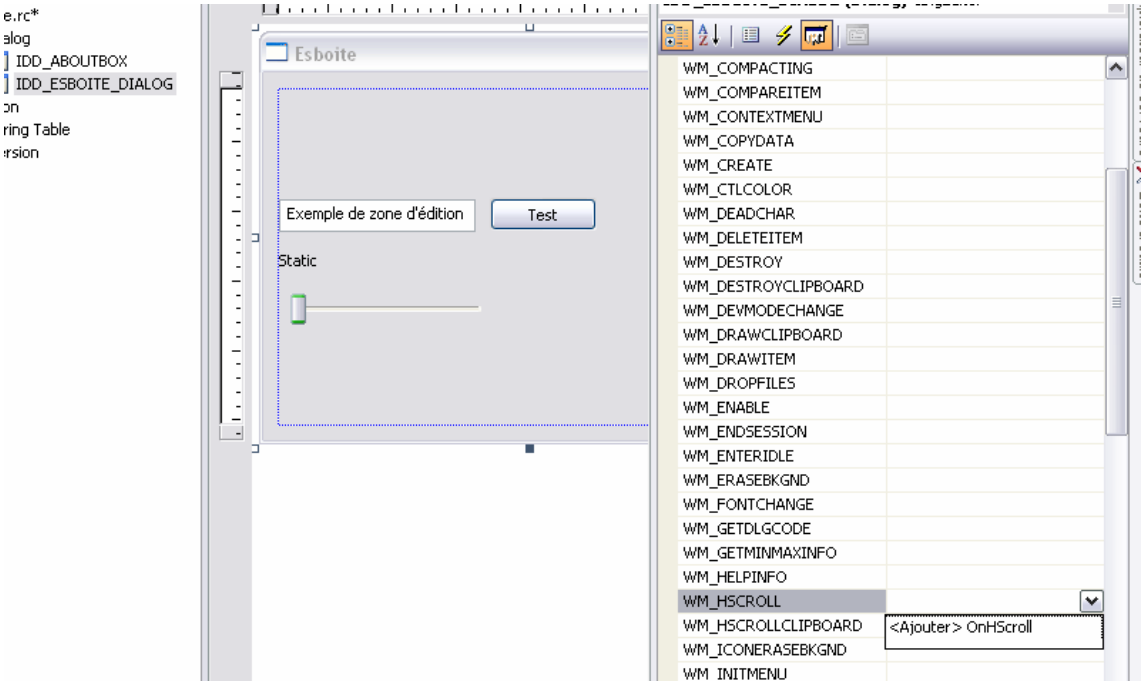
l'assistant une variable membre à EsboîteDlg qui sera non plus une valeur mais un contrôle. On pourra alors faire appel à toutes les fonctions membres de ce contrôle pour agir sur l'objet. Dans le cas de la boîte d'édition, on ajouterait un contrôle de type CEdit. Toutes les fonctions membres de la classe CEdit seraient alors à notre disposition.

Illustration avec un autre type de contrôle, le "Slider".

Si l'on ajoute un élément de type Slider et que l'on souhaite afficher dans la boîte de dialogue la valeur correspondant au curseur, on procède de la manière suivante.
On ajoute l'élément Slider, référencé par IDC_SLIDER.
A l'aide de ClasseWizard, on ajoute une variable membre de CSliderCtrl, appelée m_Slider.



L'objet est donc m_Slider, il est de classe CSliderCtrl et est automatiquement lié à l'élément dessiné, grâce à la ligne
DDX_Control(pDX, IDC_SLIDER, m_Slider);
dans la fonction DoDataExchange
L'objet m_Slider va nous permettre de récupérer à tout instant la position du curseur, grâce à la fonction GetPos, qui est membre de la classe CEditCtrl. En fait on ne va récupérer cette position que lorsque l'utilisateur va la modifier, c'est à dire lorqu'il va faire glisser le curseur. A chaque mouvement, un message est envoyé au système. Ce message est WM_HSCROLL (scrolling hoirizontal). Il faut donc que la boîte de dialogue réagisse à ce message. Pour cela, à l'aide de l'onglet Message (icône à droite de l'éclair, dans les propriétés), on crée une fonction OnHScroll (voir figure ci-dessous).



La fonction est créée en double-cliquant message (attention à ce que la classe active soit bien CEsboîteDlg)
On edite la fonction en double cliquant ensuite sur son nom, dans la zone du bas.
Dans cette fonction, l'appel
m_Slider.GetPos();
permettra de récupérer sous forme d'entier la position du curseur (comprise par défaut entre 0 et 100). Pour l'afficher dans la boîte de dialogue il estr nécessaire de convertir l'entier sous forme de chaîne. D'où la fonction OnHScroll :

```
void CEsboîteDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
	int pos;
	pos=m_Slider.GetPos(); // récupération de la position
	m_Texte.Format("%d",pos);
	UpdateData(FALSE); // actualisation de la boite (donc de l'affichage de IDC_TEXTE)
	CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

4. Insertion d'une autre boîte de dialogue dans le projet

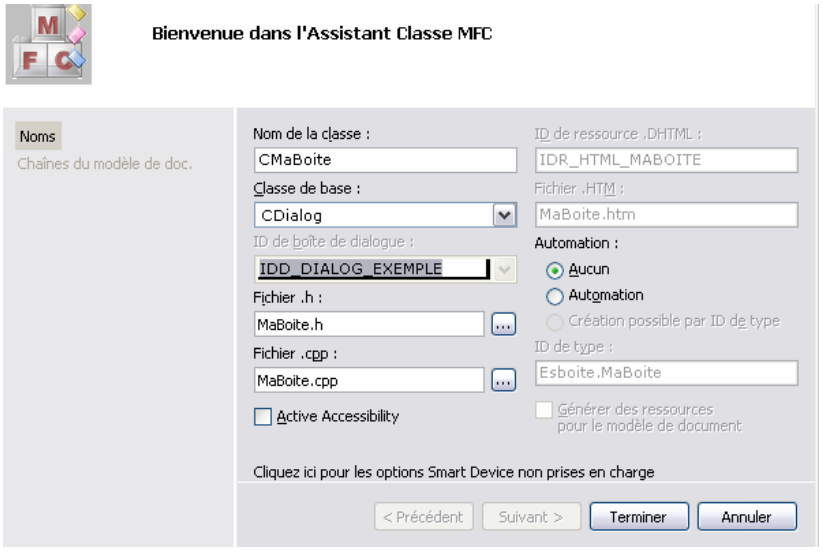
Création de la boite

Dans l'éditeur de ressources, on peut ajouter des menus, boites de dialogue, etc.
Pour cela, il suffit de cliquer sur l'icône "Dialog" avec le bouton de droite, et de choisir *Inséser Dialog*.

La boite créée s'appelle automatiquement IDD_DIALOGX, ce qui n'est pas très parlant. On commence par modifier son ID dans ses propriétés
On peut alors changer le nom et la légende (titre dans la barre bleue supérieure : case Caption). Dans l'onglet style, il est nécessaire de cocher la case *Visible*.

Création de la classe associée

Avant d'utiliser cette boite, il faut maintenant créer une classe C++ associée au dessin. L'assistant s'en charge pour nous. Il est accessible par menu contextuel si l'on clique sur bouton de droite, n'importe où dans la fenêtre d'édition de la boite. Un message s'affiche : choisissez de créer une nouvelle classe. Donnez un nom à la classe, commençant par un C (CMaBoite par exemple).
On voit ci-contre que, par défaut, la classe dérivera d'une classe CDialog, et sera liée au dessin par l'ID, c'est à dire IDD_DIALOG_EXEMPLE dans ce cas.



La classe est alors automatiquement insérée dans le projet. Pour l'utiliser un objet de cette classe, il faut prévoir un bouton dans la première boite de dialogue (l'application principale), dont l'utilisation déclenchera l'ouverture de cette nouvelle boite.
Ajoutez donc un bouton "Ouvrir la boite" dans la boite principale, et ajoutez la gestion du clic. Dans le OnOuvrirBoite que l'on crée, membre de CEsBoiteDlg, on peut ajouter le code suivant :

CMaBoite MaBoite ; // déclaration d'un objet de classe CMaboite
MaBoite.DoModal(); // Affichage et gestion de la fenêtre

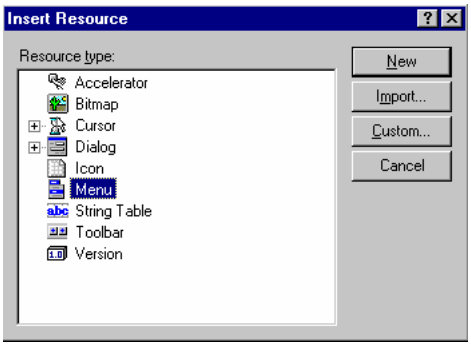
Il suffit alors d'inclure MaBoite.h au début du fichier dans lequel on utilise cet objet, et de compiler.

5. Ajout d'un menu dans une boite de dialogue

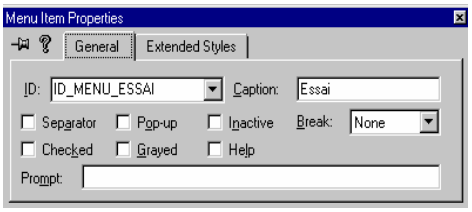
Bien que ce ne soit pas courant, il est possible d'ajouter un menu dans une boite de dialogue. La gestion de ces menus sera exactement la même lorsqu'il s'agira des menus d'une application complète.

Création du menu

Dans la partie ressources, en cliquant sur le bouton de droite sur Esboite ressources, on peut choisir *insert*, puis *menu*, et *new*.
Il suffit alors de double-cliquer sur les cases vides du menu pour ajouter des entrées. Ajoutez deux entrées principales (ligne du haut), et une entrée secondaire pour chaque entrée principale.



Pour chaque entrée secondaire, qui générera l'envoi d'un message, et donc l'appel d'une fonction, il faut un identificateur. Choisissez ID_MENU_XXX, si XXX est votre légende.



Dans les propriétés de la boite de dialogue, on peut dans la case Menu, choisir l'ID du menu que l'on vient de définir.

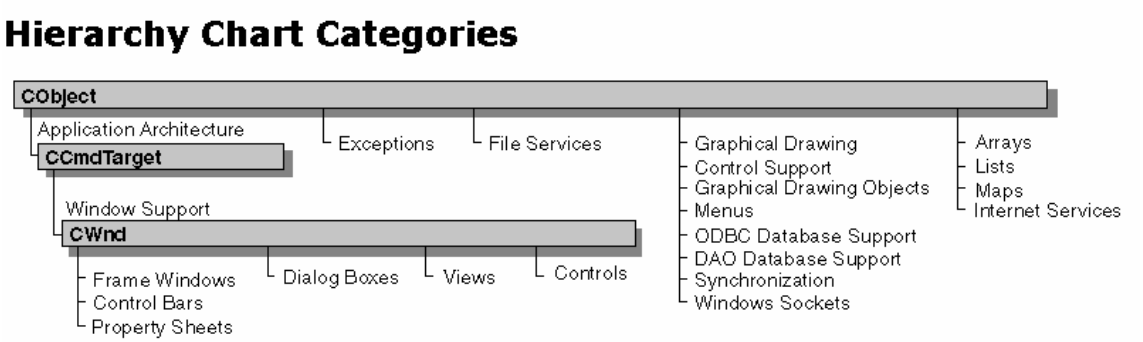
Association des entrées (item) aux clics

La procédure est similaire à celle des boutons, à l'aide du gestionnaire d'événements. Activez le menu dans l'éditeur de ressource. Cliquez sur le bouton de droite . Ajouter un gestionnaire d'événement. Sélectionnez les ID des points de menu qui vous intéressent, et faites les réagir au message COMMAND. Attention à la classe active au moment où vous ajoutez les fonctions membres en question.
Dans chacune des fonctions membres créées, utilisez

MessageBox(L"message"); // pour vérifier

LES CLASSES PERMETTANT D'OUVRIR DES FENETRES

Nous avons vu jusqu'à maintenant un type particulier de fenêtre, les boîtes de dialogue. Il en existe beaucoup d'autres, qui permettent de répondre aux différents besoins de l'interface.



Pour utiliser ces types de fenêtre, des classes ont été créées dans les MFC (Microsoft Fundation Classes). Leur hiérarchie est présentée (partiellement) ci dessus par catégorie. La plupart des classes dérivent ainsi d'une classe CObject, et celles permettant de gérer des fenêtres au sens large dérivent de la classe CWnd. C'est donc souvent sur cette classe qu'il faudra aller chercher des informations.

Les fenêtres CWnd

Comme pour les boîtes de dialogue, on n'utilisera jamais directement un objet de classe CWnd. On dérivera toujours une classe de CWnd ou de ses dérivées, afin de pouvoir rajouter les fonctionnalités qui nous intéressent (dessin, menu, etc).
La méthode de création d'une fenêtre de type CWnd sera donc la suivante.

1- Création d'une classe CWexemple dérivant de CWnd
2- Création d'un objet de type CWexemple par le constructeur par défaut :

CWexemple Ofenetre; // l'objet est créé, mais par encore la fenêtre

3- Appel de la fonction membre Create (membre de CWnd donc de CWexemple).
Cette fonction admet un grand nombre de paramètres, puisque le prototype est le suivant :

```
virtual BOOL Create( LPCTSTR lpszClassName, LPCTSTR lpszWindowName,
DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID,
CCreateContext* pContext = NULL);
```

Un exemple d'appel sera :

Ofenetre.Create(NULL,"Nom de la fenetre",WS_OVERLAPPEDWINDOW, rect, this,0,NULL);

rect devra être un objet de classe CRect, défini par exemple de la manière suivante :

CRect rect(0,0,300,400); // défini la taille initiale de la fenêtre, avant Create !

this est un opérateur spécifique du C++. Il représente le pointeur vers l'objet courant. En effet, la fonction Create sera appelée depuis la fonction membre d'un objet d'une classe donnée. La fenêtre sera alors une fenêtre fille de cet objet.

4-Il faut ensuite rendre la fenêtre visible, par l'appel de la fonction membre ShowWindow :

Ofenetre.ShowWindow(TRUE);

En procédant de cette manière, la fenêtre est créée, mais elle disparaîtra lorsque l'on quittera la fonction dans laquelle on a déclaré l'objet. Il peut donc être nécessaire de définir une fenêtre par l'intermédiaire d'un pointeur.
On écrira alors:

```
void Classe :: fonction()
{
   CRect rect(0,0,300,400);
   CWexemple *POfenetre= new CWexemple(); // déclaration d'un pointeur et
   allocation pour l'objet.
   POfenetre->Create(NULL,"Nom de la fenetre", WS_OVERLAPPEDWINDOW,
   rect, this,0,NULL);
   POfenetre->ShowWindow(TRUE);
}
```

Comme aucune libération de mémoire n'est faite, l'objet n'est pas détruit à la sortie de la fonction. C'est l'utilisateur qui détruira l'objet en fermant la fenêtre lui-même.

Remarque : l'opérateur **new** remplace malloc en programmation objet. Si l'on souhaite libérer la mémoire manuellement et ainsi détruire l'objet, il est nécessaire d'utiliser la fonction membre DestroyWindow, qui appelle de manière interne l'opérateur **delete**.

```
POfenetre->DestroyWindow(); // destruction de l'objet et libération mémoire
```

Si l'on crée directement une fenêtre comme défini ci-dessus, alors celle ci sera "prisonnière" à l'intérieur de la boite de dialogue d'où elle a été créée, car le 5ème paramètre passé à la fonction est le pointeur this. La fenêtre est donc en quelque sorte une fenêtre fille de la classe en cours. Si l'on souhaite une fenêtre indépendante, c'est une fenêtre de type WS_POPUP qu'il faut utiliser, mais la fonction Create ne le permet pas. Il faut utiliser la fonction CreateEx de la manière suivante :

```
void Classe :: fonction()
{
   CRect rect(0,0,640,480);
   CMaFen *PMaFen=new CMaFen();
   PMaFen->CreateEx(0,AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW |
   CS_BYTEALIGNWINDOW | CS_SAVEBITS,NULL,
   (HBRUSH)(COLOR_WINDOW+1)), "titre",
   WS_POPUP|WS_OVERLAPPEDWINDOW , rect,GetParent() , 0 , NULL);
   PMaFen->ShowWindow(TRUE);
}
```

L'appel à AfxRegisterWndClass permet de configurer complètement la fenêtre (la manière dont elle est redessinée à chaque mouvement ou occultation, le curseur de la souris, sa couleur de fond, etc.
WS_OVERLAPPEDWINDOW défini une fenêtre avec légende, bordure, menu système, cases d'agrandissement, de minimisation et de fermeture.

Les fenêtres CFrameWnd

Pour éviter d'avoir à redéfinir ces paramètres systématiquement, des classes MFC ont été dérivées de CWnd et sont plus simples à utiliser. C'est le cas des CFrameWnd, qui donnent exactement le même résultat que ci-dessus, à l'aide des appels suivant :

```
CAffiche *PFen=new CAffiche(); // la classe CAffiche est dérivée de CFrameWnd
PFen->Create(NULL, "Visualisation"); // création simplifiée
PFen->ShowWindow(TRUE);
```

Ces types de fenêtres sont utilisées pour les applications dites SDI (Single Document Interface), c'est à dire ne nécessitant pas l'ouvertures de plusieurs fenêtres dans sa zone client. Les application dites MDI permettent d'ouvrir plusieurs sous fenêtres ou fenêtres filles.
Pour définir rapidement une application SDI, il suffit donc dans un projet de définir une classe dérivant de CFrameWnd et de créer un objet de cette classe dans la fonction InitInstance de l'objet principal (qui lui dérive de CWinApp).
Afin de faciliter encore le développement, il est possible d'attacher un menu directement à la création de la fenêtre. Ceci se fait en utilisant la fonction LoadFrame au lieu de Create de la manière suivante :

```
CAffiche *PFen=new CAffiche(); // la classe CAffiche est dérivée de CFrameWnd
PFen->LoadFrame(IDR_MAINFRAME); // création simplifiée, chargeant le menu
PFen->ShowWindow(TRUE);
```

Le menu ayant comme identificateur IDR_MAINFRAME sera automatiquement chargé et affiché à la création de la fenêtre.
Si une icône et un curseur de souris ont été définis avec le même identificateur, ils seront automatiquement chargés également.

En résumé, pour créer une application à base d'une simple fenêtre, il faut :

- dériver une classe CMonApp de la classe CWinApp et déclarer un objet de ce type de manière globale.
- dériver une classe CMaFen de la classe CFrameWnd
- définir un menu dans l'éditeur de ressources
- Surcharger la fonction InitInstance de la classe CMonApp, pour y déclarer un objet de classe CMaFen :

```
CMaFen *PMaFen=new CMaFen();
PMaFen->LoadFrame(IDR_MAINFRAME);
m_pMainWnd=PMaFen; // recopie dans l'objet principal
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();
return TRUE;
```

- Générer les fonctions membres de la classe CMaFen à l'aide des entrées du menu.

Ceci est théorique, car en pratique, il est relativement difficile de partir de zéro (application Win32) dans l'environnement du Visual. Il faut en effet configurer toutes les options de compilation, et ajouter un grand nombre de détails sans lesquels cette application minimum ne peut tourner. **En pratique, il est plus simple de démarrer, à l'aide de AppWizard, une application basée sur une boite de dialogue, rajouter un menu et une classe dérivant de CFrameWnd, puis modifier l'InitInstance comme ci-dessus et ne pas appeler la boite de dialogue par défaut.**

Dessiner dans une fenêtre (lignes, points, textes, figures élémentaires)

Une fenêtre possède une zone dans laquelle on peut écrire du texte, dessiner des lignes ou encore afficher des images. A chaque fois que Windows doit dessiner la fenêtre, un message WM_PAINT est envoyé à cette fenêtre. C'est donc ce message qu'il faut intercepter, et c'est dans la fonction associée, OnPaint(), que l'on va placer les instructions de dessin.

Pour cela, après avoir défini une application minimale comme ci-dessus (avec une classe de fenêtre principale dérivant d'une CFrameWnd), il suffit d'utiliser ClassWizard pour que la fenêtre réagisse au message WM_PAINT.
Le fonction membre créée, Visual ajoute automatiquement

CPaintDC dc(this); // device context for painting

Il s'agit de la déclaration d'un objet DC de type CPaintDC, où DC signifie Device Context. La fenêtre est considérée comme un périphérique, exactement comme une imprimante. Pour sortir des données sur un tel périphérique, les MFC proposent une certain nombre de classes, dont la base est CPaintDC ou CClientDC qui permettent ces dessins.
A la construction de CPaintDC, la fonction CWnd::BeginPaint() est appelée, alors que la fonction CWnd::EndPaint() est appelée à la destruction.
Ces deux classes dérivent de la classe CDC, qui comportent un très grand nombre d'outils nécessaires au dessin. Les énumérer tous remplirait un livre. Voici quelques exemples :
LineTo, MoveTo, PolyLine, PolyBezier, pour les lignes
FillRect, FillSolidRect, Rectangle, pour les rectangles
Ellipse, Polygon, pour les formes fermées,
GetPixel, SetPixel, BitBlt, pour les points ou les images
TextOut pour le texte
Ces fonctions doivent être utilisées après définition d'un pinceau ou crayon virtuel avec lequel on va effectivement faire le dessin. Ce pinceau permet de définir la couleur, le type de ligne (hachuré, plein, …) ou encore l'épaisseur de la ligne.
Les outils disponibles sont principalement:
CPen pour le crayon,
CBrush pour la brosse,
CFont pour le texte,
CBitmap pour les images.

On définira par exemple un stylo rouge dessinant une ligne pleine, d'épaisseur 1, de la manière suivante :

CPen stylo(PS_SOLID,1,RGB(255,0,0));

Pour être utilisé dans le périphérique, le stylo doit être sélectionné :

DC.SelectObject(&stylo); // où DC est l'objet de classe CPaintDC

Pour positionner le stylo :

DC.MoveTo(x0,y0);

Pour dessiner une ligne de (x0,y0) à (x1,y1):

DC.LineTo(x1,y1);

Pour dessiner une ellipse (ou un cercle):

DC.Ellipse(50,50,200,200);

Pour écrire un texte (la police sera celle par défaut) en (300,100)

DC.TextOut(300,100,"HELLO");

A la fin de la fonction OnPaint, il est nécessaire de "relâcher" ou libérer le contexte de périphérique, par l'instruction

ReleaseDC(&DC);

Remarque : l'origine des coordonnées est, dans ce cas, en **HAUT** à **GAUCHE**.

Les boites de dialogue non modales

Les boites de dialogues peuvent être utilisées grâce à leur fonction membre DoModal, qui s'occupe de la gestion de toute la boite jusqu'à sa fermeture par OK ou Annuler. Le reste de l'application est bloquée pendant ce temps. De plus, la communication de données entre la boite et la classe qui, dans une de ses fonctions membres, a crée cette boite est impossible (sauf en passant le pointeur au constructeur, ou en prenant en compte les données après fermeture de la boite).

Pour éviter ces problèmes, il peut donc être nécessaire de créer une boite de dialogue comme une fenêtre standard, sans qu'elle soit bloquante. Il suffit de déclarer un pointeur vers un objet de type boite de dialogue, puis d'appeler la fonction membre Create au lieu de DoModal.
Si CMaBoite est une boite de dialogue dont l'ID est IDD_MABOITE, on peut écrire :

```
CMaBoite *PBoiteNonModale = new CMaBoite(this);
PBoiteNonModale->Create(IDD_MABOITE);
```

Remarque : A la place de IDD_MABOITE, on peut également utiliser CMaBoite::IDD, qui contient l'ID de la boite. L'avantage de cette notation est d'être effectivement indépendante du non de l'ID (cet ID peut être modifié dans l'éditeur de ressources, ce qui rendrait le code incorrect).

Le fait de passer le pointeur *this* à la boite permet à celle-ci de récupérer le pointeur de l'objet créateur ou parent, et ainsi de communiquer des données ou même d'appeler ses fonctions membres.
On déclare alors une variable membre Pparent dans la classe CMaBoite, et on récupère la valeur passée dans le constructeur.
Dans une fonction membre de CMaBoite, on peut alors accéder aux données ou fonctions membres de l'objet parent :
Pparent->Fonction() ou x=Pparent.u par exemple.

Il est toutefois nécessaire de prendre des précautions avec ce type de boite. En effet, l'objet créé n'est pas ici explicitement détruit. Il y aura donc des fuites de mémoire si l'on n'ajoute pas dans la classe CMaBoite une surcharge de la fonction PostNcDestroy, qui est une fonction appelée après destruction de la fenêtre associée à la classe CMaBoite. Le code y sera :

```
void CMaBoite::PostNcDestroy()
{
        delete this;
}
```

Dans le OnOk ou le OnCancel de la boite de dialogue, on pourra appeler la fonction DestroyWindow(), qui détruit la fenêtre associée à la classe, et déclenche l'appel à PostNcDestroy.

Communication entre objets

Passage de pointeur au constructeur :

Nous avons vu comment un exemple de communication : passage à un objet du pointeur de l'objet en cours. Ceci permet la récupération de valeurs pour un objet fils par rapport à un objet parent, sans qu'il y ait de lien d'héritage entre les classes.

```
class CBoite
{
        public:
        Cboite();
        ~Cboite();
        int Longueur, Largeur , Hauteur;
        void Fonction();
}

CBoite::Fonction()
{
        CSphere Sphere(this);
        Sphere.Creation();
}
```

```
class CSphere
{       public:
        CSphere();
        ~CSphere();
        int x,y,Rayon;
        void Creation();
        CBoite *Pparent;
}

CSphere::CSphere(CBoite *P)
{
        Pparent= P;
}

CSphere::Creation()
{
        x=Pparent->Largeur/2;
        y=Pparent->Hauteur/2;
        Rayon=Pparent->Hauteur/2;
}
```

Dans l'exemple ci-dessus, la classe CBoite possède trois variables membres, auxquelles le fonction Creation de la classe CSphere a accès, grâce au passage du pointeur *this* en paramètre au constructeur. Dans ce cas, on aurait pu aussi passer directement les valeurs des paramètres ou même leurs adresses à la fonction Création. Par contre, sous windows, il est parfois plus simple de passer le pointeur de l'objet complet, car les paramètres à passer seraient souvent trop nombreux !

Remarque : il existe également, pour les classes basées sur CWnd, la fonction GetParent qui permet de récupérer le pointeur si celui-ci n'a pas été passé en paramètre au constructeur.

Communication par message

Si l'on prend l'exemple d'une boite de dialogue non modale comportant un curseur, nous avons vu que le clic sur le curseur envoie un message WM_HSCROLL à la boite de dialogue. Par contre, l'objet parent, lui, ne reçoit rien. C'est le cas pour tous les messages Windows. Il peut être nécessaire d'envoyer un message à l'objet parent, exactement comme Windows le fait de manière interne. Pour cela, il faut commencer par définir un nouveau message. Ceux ci sont en fait de simples numéros, représentés par des noms comme WM_XXX. Le dernier utilisé par Windows est WM_USER+4. Nous pouvons donc définir nos messages de la manière suivante :

```
        enum
```

```
{
        WM_MONMESSAGE1=WM_USER+5,
        WM_MONMESSAGE2
};
```

Ainsi défini, WMMESSAGE1 prendra automatiquement la valeur WM_USER+5, et ainsi de suite pour les autres messages.
Cette définition doit être mise dans le fichier entête de la classe qui enverra le message, ou dans un fichier entête spécifique si nécessaire.

Dans une des fonctions membre de l'objet enfant, on peut appeler la fonction PostMessage :

```
m_Parent->PostMessage(WM_MONMESSAGE1,0,0);
```

La fonction PostMessage admet trois paramètres. Le premier est le message à envoyer. Le deuxième doit être un UINT, et le troisième un long. Ces paramètres pourront être utilisés pour récupérer des valeurs dans la fonction qui récupérera le message. Il peut s'agir d'un pointeur vers l'objet courant, par exemple.

Il faut ensuite ajouter, dans la carte des messages de la classe de l'objet parent, la macro nécessaire à l'aiguillage du système vers votre fonction en cas d'arrivée de votre message.

```
BEGIN_MESSAGE_MAP(CMainDlg, CDialog) // par exemple
        ON_MESSAGE(WM_MONMESSAGE1,OnMonMessage1)
END_MESSAGE_MAP()
```

On ajoute alors à la classe parent la fonction membre OnMonMessage1, admettant obligatoirement deux paramètres, un UINT et un long, et le tour est joué !

Exemple :
Dans l'objet enfant :

```
void CAdderDialog::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    m_pParent->PostMessage(WM_MONMESSAGE1,(WPARAM)nPos,0);
    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

Dans l'objet parent :

```
void CMainDlg::OnMonMessage1(UINT wParam, long lParam) // obligatoire
{
m_Slider.SetPos((int)wParam); // représente la position d'un curseur
}
```

Dans ce cas, un scroll horizontal dans l'objet parent sera répercuté dans l'objet parent.

Remarque : la fonction *PostMessage* envoie le message, mais n'est pas bloquante : les instructions continuent même si le parent n'a pas reçu le message. Au contraire, l'instruction *SendMessage* est bloquante.

En résumé, il faut donc:

1- Définir un message
2- Définir l'aiguillage dans la carte des messages
3-Définir la fonction de réception du message
4- envoyer le message !