La programmation de l'interface utilisateur

par Jessee Edouard (Accueil)

Date de publication : 22 mai 2008

Dernière mise à jour : 23 mars 2009

Dans ce tutoriel, nous allons nous intéresser à création et à l'utilisation des contrôles (bouton, zone de texte, zone de liste, etc.), des boîtes de dialogue et enfin des menus et des accélérateurs claviers ("raccourcis claviers").
Commentez cet article :

I - Les contrôles

I-A - Vue d'ensemble

I-A-1 - Généralités

Nous avons déjà vu que la création d'une fenêtre se fait toujours à partir d'une classe de fenêtre existante. Sous Windows, le concept de fenêtre est assez large. En effet, une fenêtre peut désigner aussi bien une fenêtre (telle que nous l'entendons le plus souvent) qu'un contrôle (bouton, zone de texte, etc.) ou même tout et rien ! Il y a deux grandes catégories de contrôles : les **contrôles standard**, inhérents à Windows et les **contrôles communs**, spécifiques de chaque version, implémentés respectivement dans **user32.dll** et **comctl32.dll**.
Pour un contrôle, le style **WS_CHILD** est donc obligatoire. La plupart du temps on spécifie également le style **WS_VISIBLE** pour que le contrôle soit initialement visible. Et enfin, un contrôle n'a pas de menu, à la place on spécifie un entier (un cast est donc nécessaire ...) qui nous permettra de le référencer. Cet entier est appelé l'**ID** (identifier) du contrôle. Chaque fois qu'un contrôle ou un menu est titillé par l'utilisateur, celui-ci envoie le message **WM_COMMAND** à sa fenêtre parent et place :

- son ID dans le mot de poids faible de wParam
- son handle dans lParam
- et l'événement qui s'est produit (**notification**) dans le mot de poids fort de wParam

En fait pour les contrôles communs, c'est le message **WM_NOTIFY** qui est envoyé à la place du message WM_COMMAND. La structure de ce message est la suivante :

- Dans wParam : l'ID du contrôle ayant envoyé le message.
- Dans lParam : l'adresse d'une structure de type **NMHDR** (pour **N**otification **M**essage **H**ea**d**e**r**) fournissant des informations supplémentaires sur le message.

```
typedef struct tagNMHDR {
    HWND hwndFrom; /* Handle du contrôle ayant envoyé le message */
    UINT idFrom; /* ID du contrôle */
    UINT code; /* Raison de ce message */
} NMHDR;
```

La communication avec les contrôles se fait largement via les messages. La fonction **SendMessage** permet d'envoyer un message à une fenêtre quelconque (donc y compris les contrôles ...) tandis que **SendDlgItemMessage** n'est utilisable que pour les contrôles.

```
LRESULT SendMessage(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
LRESULT SendDlgItemMessage(HWND hwndParent, int nCtrlID, UINT message, WPARAM wParam, LPARAM lParam);
```

Mais sachez qu'il existe des fonctions et/ou macros qui encapsulent un ou plusieurs SendMessage*s* dans le but d'améliorer la lisibilité du code. En général, les fonctions sont déclarées dans **winuser.h** (inclus dans windows.h) et les macros dans **windowsx.h** (qu'il faut inclure après windows.h).
Et pour terminer avec les généralités, sachez que connaissant l'ID d'un contrôle, on peut récupérer son handle et vice versa.

```
HWND GetDlgItem(HWND hwndParent, int nCtrlID);
int GetDlgCtrlID(HWND hCtrl);
```

I-A-2 - Les classes de fenêtre prédéfinies

Les classes de fenêtre suivantes sont définies pour toutes les applications fenêtrées et peuvent donc être utilisées pour créer un contrôle (elles sont enregistrées avant même que WinMain ne soit appelée).

Nom	Classe
"BUTTON"	Bouton
"COMBOBOX"	Zone de liste combinée
"EDIT"	Contrôle permettant à l'utilisateur de saisir du texte
"LISTBOX"	Zone de liste
"RichEdit"	Contrôle permettant à l'utilisateur de saisir du texte enrichi
RICHEDIT_CLASS	Amélioration de la classe RichEdit (plus sophistiqué ...)
"SCROLLBAR"	Barre de défilement
"STATIC"	Etiquette

Ce sont les **contrôles standard** (il ne s'agit pas d'une liste exhaustive).
Par exemple :

```
CreateWindow( "BUTTON",                 /* Classe : bouton           */
              "OK",                     /* Texte du bouton           */
              WS_CHILD | WS_VISIBLE,    /* Styles                    */
              0,                        /* x                         */
              0,                        /* y                         */
              100,                      /* Largeur                   */
              20,                       /* Hauteur                   */
              hWnd,                     /* Fenêtre parent            */
              (HMENU)1,                 /* ID du bouton : 1          */
              hInstance,                /* Instance de l'application */
              NULL                      /* Paramètres additionnels   */
            );
```

Bien que l'on puisse créer le contrôle à n'importe quel moment, le moment idéal pour le faire est pendant la création de sa fenêtre parent c'est-à-dire pendant le traitement du message **WM_CREATE**. Autre remarque : il est rarement nécessaire de récupérer le handle du contrôle (retourné par CreateWindow) car en général, on travaille avec l'ID du contrôle plutôt qu'avec son handle (parce que c'est plus facile : ça évite de déclarer 50 variables pour 50 contrôles ...) et de toute façon, au cas où on aurait besoin de son handle, il y a toujours la fonction **GetDlgItem**. N'oubliez pas non plus que **DestroyWindow** détruit non seulement une fenêtre mais également toutes ses fenêtres enfants. Il n'est donc pas nécessaire de détruire manuellement les contrôles d'une fenêtre ni avant ni après que celle-ci ait été détruite, puisque cela se fait automatiquement.
Voici un exemple qui montre comment profiter du message WM_CREATE pour créer notre contrôle :

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HINSTANCE hInstance;

    switch (message)
    {
    case WM_CREATE:
        hInstance = ((LPCREATESTRUCT)lParam)->hInstance;

        CreateWindow("BUTTON", "OK", WS_CHILD | WS_VISIBLE, 0, 0, 100, 24,
        hwnd, (HMENU)1, hInstance, NULL);

        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

Maintenant on va tester si notre bouton fonctionne bien. Pour cela on va émettre un beep en réponse à chaque clic. La fonction **Beep** reçoit en arguments la fréquence en Hz et la durée en ms du son à émettre. Elle très pratique pour déboguer.

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HINSTANCE hInstance;

    switch (message)
    {
    case WM_CREATE:
        hInstance = ((LPCREATESTRUCT)lParam)->hInstance;

        CreateWindow("BUTTON", "OK", WS_CHILD | WS_VISIBLE, 0, 0, 100, 24,
        hwnd, (HMENU)1, hInstance, NULL);

        break;

    case WM_COMMAND:
        /*************************************************\
        * LOWORD(wParam) = ID du contrôle ou du menu      *
        * HIWORD(wParam) = Raison du message (notification) *
        \*************************************************/

        switch(LOWORD(wParam))
        {
        case 1:
            switch(HIWORD(wParam))
            {
            case BN_CLICKED:
                Beep(1000, 100);
                break;

            default:
                break;
            }

            break;

        default:
            break;
        }

        break; /* case WM_COMMAND */

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

I-A-3 - La boucle des messages revisitée

Un contrôle peut être atteint par l'utilisation de la touche TAB s'il possède le style **WS_TABSTOP**. Ce comportement (de même que la notion de bouton par défaut) est en fait typique des boîtes de dialogue (qui sont bien entendu également des fenêtres ...). Pour autoriser un tel comportement dans une fenêtre « normale », il suffit de passer tous les messages à **IsDialogMessage** qui effectue un traitement adéquat si le message est un *« Dialog Message »*. Si la fonction retourne TRUE, ce qui signifie que le message était bien un Dialog Message (et donc déjà été convenablement traité), il ne faut plus faire un nouveau traitement. La boucle des messages devient donc :

```
while (GetMessage(&msg, NULL, 0, 0))
```

```
{
    if (!IsDialogMessage(hWnd, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

I-B - Les contrôles standard

I-B-1 - Le contrôle Bouton

I-B-1-a - Les styles

Dans la colonne de gauche, la liste des constantes permettant de spécifier le type du bouton et dans celle de droite, la liste des constantes permettant de chosir la disposition texte du bouton.

Type du bouton	Disposition
BS_PUSHBUTTON	**Position du texte par rapport au bouton**
BS_DEFPUSHBUTTON	BS_LEFTTEXT (BS_RIGHTBUTTON)
BS_CHECKBOX	
BS_AUTOCHECKBOX	**Alignement du texte**
BS_RADIOBUTTON	BS_LEFT
BS_AUTORADIOBUTTON	BS_RIGHT
BS_3STATE	BS_TOP
BS_AUTO3STATE	BS_BOTTOM
BS_PUSHLIKE	BS_CENTER
BS_GROUPBOX	BS_VCENTER
BS_OWNERDRAW	

BS_PUSHBUTTON (**bouton poussoir** ou de commande) est le type par défaut. Le type BS_DEFPUSHBUTTON est en principe réservé au **bouton par défaut** (celui qui est activé lorsqu'on appuie sur ENTREE) d'une boîte de dialogue. Par défaut, ce bouton s'il est présent doit avoir l'ID **IDOK**. Mais on peut changer cela en envoyant à la fenêtre le message **DM_SETDEFID** qui permet d'utiliser une autre valeur (que l'on placera dans wParam) à la place de IDOK. Un **bouton automatique** (BS_AUTO*XXX*) est en gros un bouton dont l'état change automatiquement chaque fois qu'on clique dessus.
Un BS_CHECKBOX et un BS_3STATE ont la même apparence (**case à cocher**) sauf qu'un BS_3STATE, en plus de pouvoir être coché (BST_CHECKED) ou décoché (BST_UNCHECKED), peut aussi être dans un état non assigné (BST_INDETERMINATE). Les cases à cocher sont généralement utilisées pour donner à l'utilisateur la possibilité de sélectionner une ou plusieurs options parmi un groupe d'options. A l'inverse les **boutons radio** sont généralement utilisés pour demander à l'utilisateur de faire un choix parmi un groupe d'options. Si cette convention est conforme à vos attentes, vous n'avez aucune raison d'utiliser les boutons de type BS_CHECKBOX, BS_3STATE ou BS_RADIOBUTTON à la place de BS_AUTOCHECKBOX, BS_AUTO3STATE ou BS_AUTORADIOBUTTON.
Pour être encore plus rationnel (et cela est même requis pour les auto radio buttons), il ne faut pas se contenter de placer les contrôles les uns à côté des autres à l'intérieur d'un rectangle assez net pour former un « vrai » groupe de contrôles. En plus de cela, le premier contrôle du groupe doit avoir le style **WS_GROUP**. Autrement dit, un contrôle possédant ce style commence un nouveau groupe. Généralement, on donne aussi le style WS_TABSTOP au premier contrôle de chaque groupe pour permettre à l'utilisateur de naviguer facilement entre les différents groupes à l'aide de la touche TAB et de passer d'un contrôle à un autre du même groupe en utilisant les touches fléchées du clavier. Du point de vue esthétique, il est coutume de placer un groupe d'options à l'intérieur d'un **group box** (BS_GROUPBOX) mais bien entendu, cela n'est point une règle.
Et enfin, BS_PUSHLIKE est un style qu'on peut donner aux *« state-buttons »* (BS_AUTOCHECKBOX, BS_AUTORADIOBUTTON, etc.) pour leur donner une apparence de push button.

I-B-1-b - Les fonctions

CheckDlgButton :

```
BOOL CheckDlgButton(HWND hwndParent, int nIDButton, UINT uCheck);
```

Cette fonction permet de modifier l'état d'une case à cocher ou d'un bouton radio. Les valeurs utilisables sont : BST_CHECKED (coché), BST_UNCHECKED (décoché) et BST_INDETERMINATE (non assigné).

CheckRadioButton :

```
BOOL CheckRadioButton(HWND hwndParent, int nIDFirstButton, int nIDLastButton, int nIDCheckButton);
```

Permet de sélectionner (cocher) un bouton radio tout en s'assurant qu'un et seul seulement parmi un groupe ne soit sélectionné (les autres seront tous décochés).

IsDlgButtonChecked :

```
UINT IsDlgButtonChecked(HWND hwndParent, int nIDButton);
```

Cette fonction ne retourne pas une valeur booléenne (!) mais l'état du bouton (qui doit être une case à cocher ou un bouton radio) c'est-à-dire coché, décoché ou non assigné.

I-B-1-c - Les messages

BM_CLICK : Ce message peut être envoyé à un bouton pour simuler un clic de la part de l'utilisateur. Il ne nécessite aucun paramètre (c'est-à-dire que vous laissez tout simplement wParam et lParam à zéro).
BM_SETCHECK : Modifie l'état d'une case à cocher ou d'un bouton radio. Utilise wParam comme unique paramètre, qui peut être BST_CHECKED, BST_UNCHECKED ou BST_INDETERMINATE (pour les boutons à 3 états uniquement). Macro équivalente :

```
void Button_SetCheck(HWND hwndCtl, int check);
```

BM_GETCHECK : Retourne l'état d'une case à cocher ou d'un bouton radio. Ne nécessite aucun paramètre. Macro équivalente :

```
int Button_GetCheck(HWND hwndCtl);
```

BM_SETSTATE : Modifie l'état d'un bouton poussoir. Utilise wParam comme unique paramètre. Mettez TRUE pour rendre le bouton enfoncé et FALSE pour le relâcher. Macro équivalente :

```
UINT Button_SetState(HWND hwndCtl, int state);
```

BM_GETSTATE : Retourne l'état d'un bouton. Attention ! Ce n'est pas « l'inverse » du message BM_SETSTATE. BM_GETSTATE peut être utilisé avec n'importe quel bouton. Il ne nécessite aucun paramètre. Et enfin, il retourne une combinaison des valeurs suivantes :

- BST_CHECKED : le bouton est coché
- BST_UNCHECKED : le bouton est décoché
- BST_INDETERMINATE : le bouton est non assigné
- BST_FOCUS : le bouton a le focus clavier
- BST_PUSHED : le bouton est maintenu enfoncé

Ces valeurs sont toutes indépendantes donc on peut utiliser l'opérateur & pour extraire les informations utiles. Par exemple :

```
int state = (int)SendDlgItemMessage(hWnd, MYBUTTON, BM_GETSTATE, 0, 0);
if (state & BST_CHECKED) {
    ...
}
```

La Macro équivalente est :

```
int Button_GetState(HWND hwndCtl);
```

BM_SETIMAGE : Associe une nouvelle image (une icône ou un bitmap) à un bouton. Le bouton doit avoir le style **BS_ICON** pour pouvoir afficher une icône et **BS_BITMAP** pour pouvoir afficher une image bitmap (sachez également que le style **BS_TEXT** permet à un bouton d'afficher du texte mais ce style est appliqué par défaut). Avec les boutons « normaux » on ne peut qu'afficher soit du texte soit une image. Pour pouvoir afficher à la fois du texte et une image (et faire d'autres trucs jusqu'ici impossibles), il faudra utiliser des **boutons personnalisés** (**Owner Draw Buttons**) que nous verrons encore plus loin. Ce message requiert en paramètres le type de l'image qu'on veut associer au bouton (wParam) qui peut prendre les valeurs **IMAGE_ICON** ou **IMAGE_BITMAP** et le handle de l'image en question (lParam). En retour, on a le handle de l'ancienne image associée au bouton (ou NULL).
BM_GETIMAGE : Retourne le handle de l'image associée au bouton ou NULL si aucune image ne lui a été associée. Nécessite en paramètre (wParam) le type de l'image (IMAGE_ICON ou IMAGE_BITMAP).

I-B-1-d - Les notifications

BN_CLICKED : Envoyé lorsque le bouton a été cliqué
BN_DBLCLK : Envoyé lorsque le bouton a été double cliqué. Seuls les boutons radio et les boutons personnalisés savent envoyer automatiquement cette notification. Pour les autres types de bouton, il faut qu'ils aient le style **BS_NOTIFY**.

I-B-2 - Le contrôle Edit

I-B-2-a - Les styles

Format

ES_LOWERCASE / ES_UPPERCASE : Convertit automatiquement en minuscules / majuscules tous les caractères entrés.
ES_NUMBER : Le texte tapé ne doit comporter que des chiffres.
ES_ PASSWORD : Affiche des * à la place des caractères entrés. Utilisé pour saisir un mot de passe.
ES_LEFT / ES_RIGHT / ES_TEXT : Permet de choisir l'alignement du texte.

Comportement

ES_MULTILINE : Spécifie un contrôle multi lignes. Par défaut un contrôle Edit n'est constitué que d'une seule ligne.
ES_WANTRETURN : Par défaut, l'appui sur la touche ENTREE provoque l'activation du bouton par défaut si celui-ci est bien sûr présent. Pour spécifier que l'appui sur la touche ENTREE doit provoquer le saut à la ligne suivante et non l'activation du bouton par défaut, il faut spécifier le style ES_WANTRETURN.
WS_HSCROLL / WS_VSCROLL : En fait, ces styles ne sont pas spécifiques des contrôles Edit mais communs à toutes les fenêtres. Ils permettent d'indiquer si la fenêtre comporte oui ou non une barre de défilement horizontale/verticale. Ces barres lorsqu'elles sont présentes sont automatiquement prises en charge par le contrôle Edit alors qu'il faut en général soi-même les gérer pour les autres classes de fenêtre.
ES_AUTOHSCROLL : Active la saisie « kilométrique » c'est-à-dire permet à l'utilisateur de continuer la saisie sur la même ligne même quand la largeur du texte dépasse la largeur du contrôle.

ES_AUTOVSCROLL : Fait défiler automatiquement le texte vers le haut quand l'utilisateur tape ENTREE à la dernière ligne.
ES_READONLY : Spécifie un contrôle en lecture seule, c'est-à-dire que le contrôle peut afficher du texte mais l'utilisateur ne pourra pas le modifier.

I-B-2-b - Les messages

WM_COPY : Copie la sélection courante vers le presse-papiers. Ne nécessite aucun paramètre.
WM_CUT : Coupe la sélection courante et place le texte ainsi coupé dans le presse-papiers. Ne nécessite aucun paramètre
WM_PASTE : Copie si possible le contenu du presse-papiers à partir de la position courante dans le contrôle. Ne nécessite aucun paramètre.
WM_CLEAR : Efface la sélection courante. Ne nécessite aucun paramètre.
EM_CANUNDO : Retourne l'état du flag « Can Undo » indiquant s'il est possible d'annuler la dernière action (TRUE) ou non (FALSE). Ce message ne nécessite aucun paramètre. Macro équivalente :

```
BOOL Edit_CanUndo(HWND hwndCtl);
```

EM_UNDO : Annule, si possible, la dernière action effectuée. Ne nécessite aucun paramètre. Macro équivalente :

```
BOOL Edit_Undo(HWND hwndCtl);
```

EM_LIMITTEXT : Ou EM_SETLIMITTEXT. Limite le nombre maximum de caractères que l'utilisateur peut entrer à un nombre passé en paramètre (wParam). On peut mettre 0 pour limiter ce nombre au maximum supporté par le système.
EM_GETLIMITTEXT : Retourne le nombre maximum de caractères que l'utilisateur peut entrer. Ne nécessite aucun paramètre.
EM_SETPASSWORDCHAR : Spécifie le caractère à utiliser pour masquer les mots de passe (par défaut : '*'). Le seul paramètre requis est le caractère de remplacement (wParam). Macro équivalente :

```
void Edit_SetPasswordChar(HWND hwndCtl, UINT ch);
```

EM_GETPASSWORDCHAR : Retourne le caractère actuellement utilisé pour masquer les mots de passe. Macro équivalente :

```
TCHAR Edit_GetPasswordChar(HWND hwndCtl);
```

EM_GETLINECOUNT : Retourne le nombre actuel de lignes d'un contrôle multi lignes. Ce nombre est toujours supérieur ou égal à 1, même si le contrôle est vide (auquel cas la valeur retournée est 1, comme s'il y avait une ligne). Macro équivalente :

```
int Edit_GetLineCount(HWND hwndCtl);
```

EM_LINELENGTH : Retourne la longueur c'est-à-dire le nombre de caractères d'une ligne. Ce message ne nécessite qu'un seul paramètre (wParam) à savoir de numéro de la ligne que l'on veut interroger. Le numéro de la première ligne est 0. Macro équivalente :

```
int Edit_LineLength(HWND hwndCtl, int line);
```

EM_GETLINE : Copie le contenu d'une ligne dans un buffer. Il faut placer le numéro de la ligne à copier dans le paramètre wParam et l'adresse du buffer destiné à recevoir les caractères copiés dans lParam. Il faut également placer la taille du buffer dans le premier mot (WORD) de ce buffer. Le caractère de fin de chaîne n'est pas ajouté. En retour, on a le nombre de caractères copiés.
En général, on envoie donc dans un premier temps le message EM_LINELENGTH avant d'envoyer le message EM_GETLINE, comme montré dans l'extrait de code suivant :

```
size_t len = (size_t)SendMessage(hEdit, EM_LINELENGTH, (WPARAM)iLine, 0);

TCHAR * lpBuffer = malloc( max(len + 1, sizeof(WORD)) * sizeof(TCHAR) );

if (lpBuffer != NULL)
{
    memcpy(lpBuffer, &len, sizeof(WORD));
    SendMessage(hEdit, EM_GETLINE, (WPARAM)iLine, (LPARAM)lpBuffer);
    lpBuffer[len] = '\0';

    MessageBox(hWnd, lpBuffer, "", MB_OK);

    free(lpBuffer);
}
```

La Macro équivalente est :

```
int Edit_GetLine(HWND hwndCtl, int line, LPTSTR lpBuffer, WORD wBufLength);
```

Bien entendu on peut toujours utiliser **GetWindowText** pour récupérer tout le texte d'un contrôle Edit comme avec n'importe quelle fenêtre. **GetWindowTextLength** permet de connaître le nombre de caractères constituant le texte d'une fenêtre.
Mais il ya tout de même un problème. En fait, GetWindowText n'est pas trop recommandé pour récupérer le contenu d'un contrôle multi lignes puisque cette opération peut consommer beaucoup de mémoire. Pour avoir accès à tout le contenu, il vaut mieux accéder directement à la mémoire utilisée par le contrôle lui-même. C'est là qu'entre en jeu le message **EM_GETHANDLE**.
EM_GETHANDLE : Retourne le handle de la mémoire utilisée par un contrôle multi lignes (en effet son utilisation avec les contrôles à une seule ligne présente peu d'intérêt). Ce handle, passé à **LocalLock**, permet d'obtenir ensuite un pointeur vers cette mémoire, qui est un tableau de caractères (ANSI ou UNICODE selon le jeu de caractères que vous utilisez) terminé par zéro. Le programme peut lire le contenu de cette mémoire mais ne peut pas la modifier. Pour déverrouiller le contrôle, on appellera la fonction **LocalUnlock**. Par exemple :

```
HLOCAL hMemory = (HLOCAL)SendMessage(hEdit, EM_GETHANDLE, 0, 0);

TCHAR * pMemory = LocalLock(hMemory);

MessageBox(hWnd, pMemory, "", MB_OK);

LocalUnlock(hMemory);
```

La Macro équivalente est :

```
HLOCAL Edit_GetHandle(HWND hwndCtl);
```

EM_GETSEL : Permet de récupérer le début et la fin de la sélection courante. Nécessite en paramètres les adresses respectives de la variable destinée à recevoir l'indice du début de la sélection (wParam) et de celle destinée à recevoir l'indice de la fin de sélection (lParam). La fin de la sélection n'est pas le dernier caractère sélectionné mais le caractère suivant.
EM_SETSEL : Modifie la sélection courante. Ce message nécessite en paramètres les indices respectifs du début (wParam) et de la fin (lParam) de la nouvelle sélection. Utilisez respectivement 0 et -1 pour tout sélectionner.
EM_REPLACESEL : Remplace le texte de la sélection courante par un autre (lParam). Il faut également spécifier (wParam) si l'action peut être annulé (TRUE) ou non (FALSE).
EM_GETMODIFY : Retourne l'état du flag « Modified » indiquant si le contenu a été modifié depuis la dernière fois qu'on l'a marqué comme sain et sauf. TRUE indique que le contenu a été modifié. Macro équivalente :

```
BOOL Edit_GetModify(HWND hwndCtl);
```

EM_SETMODIFY : Permet de modifier l'état du flag « Modified » du contrôle. Mettre FALSE (wParam) pour indiquer que le contenu est sain et sauf. Macro équivalente :

```
void Edit_SetModify(HWND hwndCtl, UINT fModified);
```

I-B-2-c - Les notifications

EN_UPDATE : Envoyé chaque fois que l'utilisateur modifie le contenu du contrôle. Cette notification est envoyée avant que la modification ne prenne effet.
EN_CHANGE : Envoyé chaque fois que l'utilisateur modifie le contenu du contrôle. Cette notification est envoyée après que la modification ait pris effet.
EN_MAXTEXT : Envoyé lorsque l'utilisateur a tenté d'entrer plus de texte que le contrôle ne peut supporter.
EN_SETFOCUS : Envoyé lorsque le contrôle reçoit le focus du clavier.
EN_KILLFOCUS : Envoyé lorsque le contrôle vient de perdre le focus du clavier.

I-B-3 - Le contrôle Static

Certainement un des plus simples. Par contre les styles sont très nombreux mais nous ne les détaillerons pas tous, MSDN est là pour ça. Le contrôle étiquette est la plupart du temps utilisé pour afficher du texte (les styles **SS_LEFT**, **SS_RIGHT** et **SS_CENTER** permettent de spécifier l'alignement) mais sachez qu'il peut également afficher une image (une icône ou un bitmap). Pour cela il suffit de spécifier le style **SS_ICON** ou **SS_BITMAP** selon le type d'image qu'on veut afficher puis envoyer le message **STM_SETIMAGE** pour spécifier l'image à afficher. A part ça, ce contrôle n'envoie des notifications (**SN_CLICKED**, **SN_DBLCLK**, etc.) que si le style **SS_NOTIFY** est spécifié. En général on n'a pas besoin des notifications venant de ce contrôle.

I-B-4 - Le contrôle Zone de liste (ListBox)

I-B-4-a - Les styles

LBS_DISABLENOSCROLL : Spécifie que la barre de défilement du contrôle doit toujours être visible. Par défaut, cette barre n'apparaît que lorsque le contrôle contient trop d'éléments pour être tous affichés en même temps.
LBS_EXTENDEDSEL : Spécifie que l'utilisateur peut sélectionner plusieurs éléments en même temps à l'aide de la souris et les touches SHIFT ou CTRL ou à l'aide d'une combinaison spéciale de touches.
LBS_MULTIPLESEL : Spécifie que l'utilisateur peut sélectionner plusieurs éléments en même temps en cliquant successivement sur les éléments à sélectionner. La sélection / désélection d'un élément se fait à l'aide d'un simple clic.
LBS_NOSEL : Spécifie que les éléments de la liste peuvent être uniquement visualisés et non sélectionnés.
LBS_SORT : Spécifie que les éléments de la liste doivent être triés alphabétiquement.
LBS_NOTIFY : Autorise le contrôle à envoyer des notifications (**LBN_CLICKED**, **LBN_DBLCLK**, etc.).
LBS_STANDARD : Zone de liste standard (inclut les styles les plus courants, à savoir WS_BORDER, LBS_SORT et LBS_NOTIFY).

I-B-4-b - Les messages

LB_ADDSTRING : Ajoute une chaîne (lParam) à la fin de la liste. Macro équivalente :

```
int ListBox_AddString(HWND hwndCtl, LPCTSTR lpszString);
```

LB_INSERTSTRING : Insère une chaîne (lParam) à la position numéro n (wParam) sachant que l'indice du premier élément est 0. Si wParam = -1, la chaîne sera ajoutée à la fin. Macro équivalente :

```
int ListBox_InsertString(HWND hwndCtl, int index, LPCTSTR lpszString);
```

LB_DELETESTRING : Supprime une chaîne de la liste. Le seul paramètre requis est l'indice de la chaîne à supprimer (wParam). Macro équivalente :

```
int ListBox_AddString(HWND hwndCtl, int index);
```

LB_GETCOUNT : Retourne le nombre d'éléments dans la liste. Macro équivalente :

```
int ListBox_GetCount(HWND hwndCtl);
```

LB_GETTEXTLEN : Retourne la longueur (strlen) du texte de l'élément spécifié (wParam). Macro équivalente :

```
int ListBox_GetTextLen(hwndCtl, int index);
```

LB_GETTEXT : Copie le texte de l'élément spécifié (wParam) pour le placer dans un buffer (lParam). Ce buffer doit être suffisamment grand pour contenir le texte copié ainsi que le caractère de fin de chaîne (qui est automatiquement ajouté). Macro équivalente :

```
int ListBox_GetTextLen(hwndCtl, int index, LPTSTR lpszBuffer);
```

LB_FINDSTRING : Recherche une chaîne (lParam) à partir de la position spécifiée (wParam), plus précisément à partir de la position suivante. Donc pour chercher depuis le début de la liste (position 0), il faut passer -1 en paramètre. En fait, LB_FINDSTRING cherche une chaîne contenant le préfixe placé dans lParam et non la chaîne exacte. Pour chercher la chaîne exacte, il faut utiliser **LB_FINDSTRINGEXACT** à la place de LB_FINDSTRING. Dans tous les cas, la recherche ne tient pas compte de la casse et la valeur retournée est l'indice du premier élément trouvé ou LB_ERR si aucun élément n'a été trouvé. Macro équivalente :

```
int ListBox_FindString(HWND hwndCtl, int indexStart, LPCTSTR lpszString);
```

LB_SETITEMDATA : Associe une valeur (lParam) à un élément de la liste (wParam). Macro équivalente :

```
int ListBox_SetItemData(HWND hwndCtl, int index, LPARAM data);
```

LB_GETITEMDATA : Retourne la valeur associée à un élément de la liste (wParam). Macro équivalente :

```
LRESULT ListBox_SetItemData(HWND hwndCtl, int index);
```

LB_GETSEL : Retourne l'état d'un élément de la liste (wParam). Une valeur positive indique que l'élément est sélectionné et 0 indique qu'il est désélectionné. LB_ERR est retourné en cas d'erreur. Macro équivalente :

```
int ListBox_GetSel(HWND hwndCtl, int index);
```

LB_SETSEL : Modifie l'état (wParam) d'un élément de la liste (lParam). Utilisez TRUE pour sélectionner l'élément et FALSE pour le désélectionner. Si lParam = -1, la modification sera appliquée à tous les éléments de la liste. Macro équivalente :

```
int ListBox_GetSel(HWND hwndCtl, BOOL fSelect, int index);
```

LB_GETCURSEL : Dans un contrôle à sélection simple, retourne l'indice de la sélection courante ou LB_ERR si aucun élément n'est sélectionné. Macro équivalente :

```
int ListBox_GetCurSel(HWND hwndCtl);
```

LB_SETCURSEL : Sélectionne un élément (wParam). Macro équivalente :

```
int ListBox_GetCurSel(HWND hwndCtl, int index);
```

LB_GETSELCOUNT : Dans un contrôle à sélection multiple, retourne le nombre d'éléments sélectionnés, 0 si aucun élément n'est sélectionné. Macro équivalente :

```
int ListBox_GetSelCount(HWND hwndCtl);
```

LB_GETSELITEMS : Permet de récupérer dans un tableau (lParam) d'entiers les indices des éléments sélectionnés. On devra également placer dans wParam le nombre maximum d'éléments que peut accepter le tableau. Macro équivalente :

```
int ListBox_GetSelItems(HWND hwndCtl, int cItems, int * lpItems)
```

LB_RESETCONTENT : Réinitialise la liste (supprime tous les éléments). Macro équivalente :

```
BOOL ListBox_ResetContent(HWND hwndCtl);
```

I-B-5 - Le contrôle Zone de liste combinée (ComboBox)

Le contrôle **ComboBox** combine les contrôles **Edit** et **ListBox** bien qu'il n'implémente pas forcément toutes leurs fonctionnalités. Le style **CBS_SIMPLE** affiche en permanence le contrôle **ListBox** alors qu'avec le style **CBS_DROPDOWN** ce dernier n'apparaît que lorsque l'utilisateur déroule la liste. Avec le style **CBS_DROPDOWNLIST**, l'utilisateur ne peut plus saisir directement du texte dans le contrôle Edit mais doit choisir un élément dans la liste. Les messages LB_GETTEXT et LB_GETTEXTLEN ont été traduits en **CB_GETLBTEXT** et **CB_GETLBTEXTLEN** (et non en CB_GETTEXT et CB_GETTEXTLEN auxquels on aurait pu s'attendre) et les noms de macros commencent tous par ComboBox (**ComboBox_AddString**, **ComboBox_GetCursel**, **ComboBox_GetLBText**, **ComboBox_GetLBTextLen**, etc.). De même, beaucoup des styles des contrôles Edit et ListBox n'ont plus été retenus, en particulier le style **CBS_NOTIFY** qui n'existe plus car n'étant plus nécessaire étant donné que ce contrôle émet abondamment des notifications dont voici une liste non exhaustive :
CBN_DROPDOWN : La zone de liste est sur le point d'être déroulée (envoyé par ceux qui ont le style CBS_DROPDOWN ou CBS_DROPDOWNLIST uniquement).
CBN_DBLCLK : Un élément de la zone de liste a été double-cliqué.
CBN_CLOSEUP : La zone de liste est sur le point d'être repliée (envoyé par ceux qui ont le style CBS_DROPDOWN ou CBS_DROPDOWNLIST uniquement).
CBN_SELCHANGE : La sélection dans zone de la liste vient de changer.
CBN_SELENDCANCEL : L'utilisateur a quitté le contrôle sans valider la sélection.
CBN_SELENDOK : L'utilisateur a bien fermé la zone de liste, indiquant qu'il a validé son choix.
CBN_EDITCHANGE : L'utilisateur a modifié le texte du contrôle Edit du ComboBox.
CBN_EDITUPDATE : L'utilisateur tente de modifier le texte du contrôle Edit du ComboBox.
CBN_KILLFOCUS : Le contrôle a perdu le focus du clavier.

I-C - Les contrôles communs

I-C-1 - Introduction

Les **contrôles communs** sont les contrôles spécifiques d'une version donnée de Windows (barre d'outils, barre d'état, barre de progression, glissière, etc.). Ils sont implémentés dans le fichier **comctl32.dll**. Pour les utiliser, il faut se lier avec **comctl32.lib** et inclure **commctrl.h**.

I-C-2 - Initialisation

Il ne nous est plus à rappeler que la création d'une fenêtre se fait toujours à partir d'une classe de fenêtre existante. Lorsqu'un programme se lie avec user32.dll, le système enregistre les classes de fenêtre standard (les contrôles standard) avant même que la fonction WinMain ne soit appelée. C'est pour cette raison que nous n'avons jamais eu à enregistrer ces classes nous-mêmes. Ce n'est pas le cas avec les contrôles communs et l'application doit donc initialiser ces contrôles elle-même avant de pouvoir les utiliser. Cela se fait tout simplement en appelant la fonction **InitCommonControlsEx** (qui est venue remplacer la fonction **InitCommonControls**, devenue obsolète).

```
void InitCommonControls(VOID);
BOOL InitCommonControlsEx(LPINITCOMMONCONTROLSEX lpInitCommCtrls);
```

Le type **INITCOMMONCONTROLSEX** est tout simplement une structure comportant deux champs : **dwSize** et **dwICC**. Le premier doit indiquer la taille en octets de la structure et le dernier la ou les classes de fenêtre qu'on veut enregistrer (nous y reviendrons là-dessus un peu plus bas). La fonction InitCommonControls ne permet pas de spécifier individuellement les contrôles à initialiser.

I-C-3 - Les versions

Le fichier comctl32.dll n'est pas forcément le même sur tous les ordinateurs car sa version dépend du système d'exploitation et/ou logiciels installés (en particulier Internet Explorer). Il n'est pas exclu non plus que plusieurs versions soient simultanément présentes sur un même ordinateur (évidemment séparées dans des répertoires différents). Par exemple, Windows XP et Vista sont accompagnés des versions 5 (en l'occurrence 5.82) et 6 alors que Windows 2000 n'était accompagné que de la version 5 (5.81). Si vous avez Windows 95, vous avez la version 4.0 mais si vous installez ensuite Internet Explorer 3.x, vous bénéficierez de la version 4.70. Chaque nouvelle version est un sur ensemble des versions antérieures.
La version 6 intègre non seulement les contrôles supplémentaires mais aussi les contrôles standard, cependant ces derniers ne sont pas les mêmes que ceux de user32 (mais ils continuent à envoyer leurs notifications via le message WM_COMMAND, et non WM_NOTIFY comme le reste des contrôles communs). En effet, Windows XP est venu avec une nouvelle interface utilisateur personnalisable, permettant à l'utilisateur de choisir luimême le **style visuel** à appliquer. Les contrôles de la version 6 utilisent ce style alors que ceux de user32 et des versions antérieures à 6 sont basés sur le style classique. Ainsi, si vous voulez utiliser les styles visuels dans vos applications, vous devez utiliser les contrôles de la version 6 ou plus récente. Nous en reparlerons plus tard.
En outre, il faut également savoir que la fonction **InitCommonControlsEx** n'a été introduite que depuis la version 4.70 (livrée avec IE 3.x). Selon la version de votre fichier d'en-tête, il se peut donc que cette fonction ne soit déclarée que si vous définissez explicitement la macro **_WIN32_IE** à **0x0300** ou supérieur avant d'inclure **commctrl.h** (sous Visual Studio .NET, elle vaut par défaut 0x0500).

I-C-4 - Choix des bibliothèques

Comme nous venons tout juste de le dire, la version 6.0 de la Common Controls Library est la première à avoir supporté les styles visuels. Or les applications compilées avec Visual Studio .NET et 2005 utilisent par défaut les contrôles de user32.dll et de comctl32.dll version 5 pour être compatibles avec les anciennes versions de Windows. Pour utiliser la version 6 (ou une autre ...), c'est à l'application de le spécifier. En effet, quand on se lie avec comctl32.lib, il n'est spécifié nulle part quelle version de comctl32.dll veut-on utiliser. C'est là qu'interviennent les fichiers **MANIFEST**.
Un fichier manifest est un fichier texte, utilisant une grammaire XML, permettant dans Windows XP et plus récents (en effet il est ignoré par les versions antérieures) de spécifier entre autres les composants (les « dépendances ») requis par une application pour fonctionner. Le nom d'un fichier manifest doit être le nom de l'application suivi de l'extension **.manifest** (par exemple hello.exe.manifest). Par exemple, pour utiliser comctl32.dll version 6.0, il faut créer le manifest suivant :

```
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>

<assembly xmlns='urn:schemas-microsoft-com:asm.v1' manifestVersion='1.0'>
    <dependency>
        <dependentAssembly>
            <assemblyIdentity type='Win32'
                              name='Microsoft.Windows.Common-Controls'
                              version='6.0.0.0'
                              processorArchitecture='x86'
                              publicKeyToken='6595b64144ccf1df'
            />
        </dependentAssembly>
    </dependency>
</assembly>
```

Les paramètres sont plutôt parlants.

De plus, si vous utilisez par exemple Visual Studio 2005, vos applications utiliseront la version 8.0 du C Run-Time Library (**msvcr80.dll**), sauf bien sûr si vous utilisez la version statique. Dans ce cas, il faut également le spécifier dans le manifest. Donc si vous utilisez msvcr80.dll et comctl32.dll version 6.0, vous devez créez le manifest suivant :

```
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>

<assembly xmlns='urn:schemas-microsoft-com:asm.v1' manifestVersion='1.0'>

    <!-- Microsoft C Run-Time Library version 8.0 -->
    <dependency>
        <dependentAssembly>
            <assemblyIdentity type='win32'
                              name='Microsoft.VC80.CRT'
                              version='8.0.50608.0'
                              processorArchitecture='x86'
                              publicKeyToken='1fc8b3b9a1e18e3b'
            />
        </dependentAssembly>
    </dependency>

    <!-- Microsoft Common Controls Library version 6.0 -->
    <dependency>
        <dependentAssembly>
            <assemblyIdentity type='Win32'
                              name='Microsoft.Windows.Common-Controls'
                              version='6.0.0.0'
                              processorArchitecture='x86'
                              publicKeyToken='6595b64144ccf1df'
            />
        </dependentAssembly>
    </dependency>

</assembly>
```

En Debug vous mettrez plutôt VC80.DebugCRT à la place de VC80.CRT.
Il existe deux manières différentes d'utiliser un manifest (dans tous les cas, n'oubliez pas que le manifest n'est lu qu'à l'exécution) :

- En le plaçant dans le même répertoire que l'exécutable
- En l'embarquant à l'intérieur même de l'exécutable, autrement dit le mettre en ressource.

Lorsqu'on place un manifest dans le répertoire de l'exécutable, celui qui se trouve en ressource sera tout simplement ignoré par Windows. Sachez également que lorsqu'un manifest est utilisé, il n'est plus nécessaire d'appeler la fonction InitCommonControlsEx().
En fait, Visual Studio .NET et plus récents génèrent automatiquement un manifest, ne serait-ce que pour spécifier la version du CRT utilisée. Le fichier final, créé à partir d'un fichier "intermédiaire" nommé *votreapp.exe*.intermediate.manifest et des éventuelles options de génération que vous avez spécifiées, nommé *votreapp.exe*.embed.manifest, est ensuite embarqué dans l'exécutable de sorte que ce dernier soit plus ou moins autonome. Si vous voulez néanmoins l'utiliser en tant que fichier à part, accompagnant votre exécutable, vous n'avez qu'à renommer le fichier *votreapp.exe*.embed.manifest en *votreapp.exe*.manifest ou, tout simplement (et plus proprement !), de dire à Visual Studio de ne pas embarquer le manifest.
Visual Studio dispose d'une interface simple et intuitive permettant de contrôler très simplement et efficacement la génération de manifest. Vous pouvez par exemple spécifier des fichiers à fusionner avec le fichier par défaut utilisé pour générer le manifest grâce à l'option **Additional manifest Files** du gestionnaire de fichiers manifest (donc dans vos fichiers, vous ne devez plus lister la CRT parmi les dépendances requis car elle est déjà listée dans le fichier par défaut (*votreapp.exe*.intermediate.manifest)).
Ceci étant, voyons maintenant comment embarquer un manifest à l'intérieur de l'exécutable sans utiliser l'interface de Visual Studio. Et bien, pour mettre un manifest dans la section ressources de votre fichier exécutable, il suffit de créer un script de ressource contenant la ligne suivante (inclure windows.h) :

```
CREATEPROCESS_MANIFEST_RESOURCE_ID RT_MANIFEST "hello.exe.manifest"
```

I-C-5 - Exemple : Le contrôle ListView

Le contrôle ListView (**WC_LISTVIEW**) est un contrôle permettant, comme son nom l'indique, d'afficher une liste. La zone d'affichage des dossiers et des fichiers dans l'explorateur Windows, la zone d'affichage des processus en cours d'exécution dans le gestionnaire des tâches, etc. sont par exemple des contrôles ListView.
Le contrôle ListView peut afficher les éléments de la liste de 4 façons différentes dont voici les plus utilisées :

- Affichage en icônes (style **LVS_ICON**) : Les éléments sont affichés avec des grandes icônes avec leur en dessous de l'icône.
- Affichage en liste (style **LVS_LIST**) : Les éléments sont affichés en liste, organisée en colonnes, avec des petites icônes avec leur nom à droite de l'icône.
- Affichage en liste détaillée (style **LVS_REPORT**) : Chaque élément occupe une ligne et est chaque ligne comporte une ou plusieurs colonnes. La colonne la plus à gauche affiche l'icône et nom de l'élément (placé à droite de l'icône). Les autres colonnes servent à afficher d'autres informations. Chaque colonne est pourvu d'un en-tête qui sert à afficher son nom sauf si le style LVS_NOCOLUMNHEADER a été spécifié.

Depuis Windows XP (plus précisément comctl32.dll version 6.0), il est également possible d'utiliser d'autres styles d'affichage comme l'affichage en mosaïques (messages en jeu : LVM_SETVIEW, LVM_SETTILEVIEWINFO et LVM_SETTILEINFO) ou par groupe (messages en jeu : LVM_ENABLEGROUP, LVM_SETGROUPINFO et LVM_SETGROUPMETRICS) par exemple.
Comme la plupart des contrôles communs, le contrôle ListView envoie ses notifications à la fenêtre parent via le message WM_NOTIFY avec dans wParam son identifiant (ID), comme le fait n'importe quel contrôle commun, et dans lParam un pointeur vers une structure dérivée de la structure de base NMHDR. En fait, cette dérivation (spécialisation) de la structure NMHDR est utilisée par la quasi-totalité des contrôles communs. Il faut lire la documentation du contrôle pour savoir quelle structure est utilisée dans quelles circonstances.
L'exemple suivant a pour but de vous aider à comprendre le fonctionnement des contrôles communs à travers la création et l'utilisation d'un contrôle ListView.

```
#include <windows.h>
#include <commctrl.h>

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
void OnCreate(HWND hwnd, LPCREATESTRUCT lpCreateStruct);
void lvInitColumns(HWND hwndLV);
void lvInsertItems(HWND hwndLV);
void OnNotify(HWND hwnd, LPNMHDR lpnmhdr);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    INITCOMMONCONTROLSEX icc;
    WNDCLASS wc;
    HWND hWnd;
    MSG msg;

    icc.dwSize = sizeof(icc);
    icc.dwICC = ICC_LISTVIEW_CLASSES;

    InitCommonControlsEx(&icc);

    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
    wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wc.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wc.hInstance      = hInstance;
    wc.lpfnWndProc    = WndProc;
    wc.lpszClassName  = "Classe 1";
    wc.lpszMenuName   = NULL;
    wc.style          = CS_HREDRAW | CS_VREDRAW;

    RegisterClass(&wc);

    hWnd = CreateWindow( "Classe 1", "Contrôle ListView",
        WS_POPUP | WS_BORDER | WS_CAPTION | WS_MINIMIZEBOX | WS_SYSMENU,
```

```
        300, 150, 400, 200,
        NULL, NULL, hInstance, NULL
    );

    ShowWindow(hWnd, nCmdShow);

    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return (int)msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch(message)
    {
    case WM_CREATE:
        OnCreate(hwnd, (LPCREATESTRUCT)lParam);
        break;

    case WM_NOTIFY:
        OnNotify(hwnd, (LPNMHDR)lParam);
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0L;
}

void OnCreate(HWND hwnd, LPCREATESTRUCT lpCreateStruct)
{
    HWND hwndLV;
    RECT r;

    GetClientRect(hwnd, &r);

    hwndLV = CreateWindow( WC_LISTVIEW, "", WS_CHILD | WS_VISIBLE | LVS_REPORT,
        r.left, r.top, r.right, r.bottom,
        hwnd, (HMENU)1, lpCreateStruct->hInstance, NULL
    );

    lvInitColumns(hwndLV);
    lvInsertItems(hwndLV);
}

void lvInitColumns(HWND hwndLV)
{
    LVCOLUMN lvc;

    /* Paramètres communs à toutes les colonnes. */

    lvc.mask = LVCF_TEXT | LVCF_FMT;
    lvc.fmt = LVCFMT_LEFT;

    /* Colonne 0 (première colonne). */

    lvc.pszText = "Langage";
    ListView_InsertColumn(hwndLV, 0, &lvc);
    ListView_SetColumnWidth(hwndLV, 0, 100);

    /* Colonne 1 (deuxième colonne). */

    lvc.pszText = "Créateur";
```

```
    ListView_InsertColumn(hwndLV, 1, &lvc);
    ListView_SetColumnWidth(hwndLV, 1, 200);
}

void lvInsertItems(HWND hwndLV)
{
    LVITEM lvi;

    /* Paramètres communs à tous les éléments que nous allons insérer. */

    lvi.mask = LVIF_TEXT;

    /* Ligne 0 (première ligne). */

    lvi.iItem = 0;

    /* Ligne 0 - Colonne 0. */
    lvi.iSubItem = 0;
    lvi.pszText = "C";
    ListView_InsertItem(hwndLV, &lvi);

    /* Ligne 0 - Colonne 1. */
    lvi.iSubItem = 1;
    lvi.pszText = "Brian Kernighan & Denis Ritchie";
    ListView_SetItem(hwndLV, &lvi);
}

void OnNotify(HWND hwnd, LPNMHDR lpnmhdr)
{
    if (lpnmhdr->idFrom == 1) /* 1 est l'ID de notre ListView. */
    {
        HWND hwndLV = lpnmhdr->hwndFrom;

        if (lpnmhdr->code == NM_DBLCLK)
        {
            /* ComCtl32 version 4.71 et plus récents : Quand lpnmhdr->code vaut NM_DBLCLK, */
            /* la structure complète est une structure de type NMITEMACTIVATE.             */
            /* Cette structure contient entre autres les coordonnées du point du clic.     */

            LPNMITEMACTIVATE lpnmia = (LPNMITEMACTIVATE)lpnmhdr;
            LVHITTESTINFO lvhti; /* Contiendra le numéro de ligne et de colonne de la "cellule" double-
cliquée. */
            int ret;

            /* Récupérons le numéro de ligne et de colonne de la cellule double-cliquée. */

            lvhti.pt = lpnmia->ptAction; /* Coordonnées du point du clic. */
            ret = ListView_SubItemHitTest(hwndLV, &lvhti);

            /* Il faut tester ret car le clic a pu avoir lieu hors d'une cellule ... */

            if (ret != -1)
            {
                /* Récupérons puis affichons le texte de la cellule. */

                LVITEM lvi;
                char lpBuffer[256];

                lvi.mask = LVIF_TEXT;
                lvi.pszText = lpBuffer;
                lvi.cchTextMax = sizeof(lpBuffer);
                lvi.iItem = lvhti.iItem;
                lvi.iSubItem = lvhti.iSubItem;
                ListView_GetItem(hwndLV, &lvi);

                /* Il n'est pas garanti que le texte ait réellement été placé dans lpBuffer.  */
                /* Il est possible que le système ait utilisé un autre buffer.               */
                /* Il faut donc toujours utiliser lvi.pszText pour récupérer le texte.       */
                /* Il y a aussi la macro ListView_GetItemText qui est plus simple à utiliser. */

                MessageBox(hwnd, lvi.pszText, "", MB_OK);
            }
```

```
            }
        }
    }
```

I-D - Personnalisation des contrôles

I-D-1 - Choisir la brosse, la couleur de fond et la couleur du texte

Au temps des versions 16 bits de Windows, le message **WM_CTLCOLOR** fut envoyé chaque fois qu'un contrôle est sur le point d'être dessiné. En traitant ce message, la fenêtre parent peut spécifier une brosse, une couleur de fond et une couler du texte à utiliser avec le contexte de périphérique du contrôle. Les paramètres de ce message sont :

- Dans wParam : handle du contexte de périphérique du contrôle.
- Dans lParam : handle du contrôle ayant envoyé le message.

Si l'application traite ce message, elle doit retourner le handle de la brosse à utiliser avec le contexte de périphérique du contrôle. Par contre on utilise tout simplement SetBkColor, SetBkMode et SetTextColor pour spécifier respectivement la couleur de fond, la transparence du fond et la couleur du texte.
Depuis les versions 32 bits, ce message a été remplacé par des messages plus spécifiques comme :

- **WM_CTLCOLORBTN** (envoyé par un bouton)
- **WM_CTLCOLOREDIT** (envoyé par un contrôle Edit)
- **WM_CTLCOLORLISTBOX** (envoyé par une ListBox)
- **WM_CTLCOLORSTATIC** (envoyé par un contrôle Static)
- ...

Mais le principe est le même. A noter toutefois qu'un contrôle Edit, lorsqu'il est désactivé ou en lecture seule, envoie le message WM_CTLCOLORSTATIC à la place de WM_CTLCOLOREDIT (sachez également que le rectangle des cases à cocher et des boutons radio sont en fait de vrais contrôles Static). En outre, les boutons de commande n'utilisent pas la brosse retournée puisque leur apparence dépend de leur état (enfoncé / relâché). Pour personnaliser l'apparence des boutons de commande, il faudra utiliser les boutons personnalisés.

I-D-2 - Choisir la police des caractères

Pour choisir la police à utiliser avec un contrôle, il suffit de lui envoyer le message **WM_SETFONT** avec le handle de la police dans WPARAM et MAKELPARAM(fRedraw, 0) dans lParam où fRedraw doit être égal à TRUE si on veut que le contrôle soit immédiatement redessiné dès qu'il aura reçu le message.

I-D-3 - Dessiner soi-même ses contrôles

I-D-3-a - La théorie

La technique la plus puissante pour obtenir des contrôles très personnalisés c'est évidemment de les dessiner soi-même. Fondamentalement, cela se fait tout simplement en traitant le message WM_PAINT à l'intérieur de la procédure de fenêtre du contrôle. On peut spécifier une nouvelle procédure de fenêtre pour n'importe quelle fenêtre déjà créée en modifiant l'adresse indiquée dans sa "variable interne" GWL_WNDPROC à l'aide de la fonction SetWindowLong. En fait, SetWindowLong est une fonction obsolète car elle ne peut être utilisée que sur les plateformes 32 bits. Pour avoir du code compatible avec tous les processeurs (32 ou 64 bits), il faudra utiliser **SetWindowLongPtr** (et **GWLP_WNDPROC**).
En pratique, on n'aura pas qu'à traiter le message WM_PAINT pour dessiner le contrôle cela demande généralement trop de détails à prendre en charge. De plus, un contrôle doit généralement changer d'aspect en fonction de son état. Par exemple, pour un bouton, les états possibles sont normal, survolé par la souris, enfoncé, etc. Or les paramètres qui accompagnent le message WM_PAINT ne permettent pas de connaître dans quel état est actuellement le contrôle.

C'est pourquoi, lorsqu'on veut dessiner soi-même ses contrôles, il est généralement préférable d'utiliser les « **owner-draw controls** ».
Chaque fois qu'un owner-draw control doit être dessiné ou redessiné, Windows envoie à sa fenêtre parent le message **WM_DRAWITEM**. Si le message provient d'un contrôle, alors le paramètre wParam contient l'ID de ce contrôle sinon zéro. lParam quant à lui contient toujours l'adresse d'une structure de type **DRAWITEMSTRUCT** contenant les informations nécessaires permettant d'effectuer un traitement convenable.

```
typedef struct tagDRAWITEMSTRUCT {
    UINT CtlType;
    UINT CtlID;
    UINT itemID;
    UINT itemAction;
    UINT itemState;
    HWND hwndItem;
    HDC hDC;
    RECT rcItem;
    ULONG_PTR itemData;
} DRAWITEMSTRUCT;
```

La description complète de cette structure peut être évidemment trouvée dans l'aide de MSDN (elle n'est d'ailleurs pas compliquée), mais ce qu'il faut au moins retenir ce sont les champs **CtlID** (ID du contrôle), **hDC** (handle du DC à utiliser pour dessiner le contrôle) et **rcItem** (le rectangle qui décrit le contour du contrôle). Le champ **itemState** a également son importance car il spécifie l'état actuel du contrôle. Sa valeur peut être une ou une combinaison de constantes (indépendantes) définies dans winuser.h parmi lesquels ODS_SELECTED (l'élément est selectionné c'est-à-dire maintenu enfoncé), ODS_DISABLED (l'élément a été désactivé) et ODS_FOCUS (l'élément a le focus clavier).

I-D-3-b - Exemple : Un bouton personnalisé

Un owner-draw button (bouton personnalisé) est un bouton possédant le style **BS_OWNERDRAW**. Ce style ne peut être mélangé avec les autres styles de bouton (BS_XXX). Pour dessiner le bouton, il n'est généralement pas nécessaire de traiter le message WM_PAINT. Par contre, il est nécessaire d'ajouter un traitement sur réception des messages **WM_MOUSEMOVE** et **WM_MOUSELEAVE** pour le faire changer d'aspect respectivement quand la souris vient le survoler et quand la souris le quitte. Le dessin ne se fera cependant que dans le traitement du message WM_DRAWITEM, c'est-à-dire dans la procédure de fenêtre de la fenêtre parent.
A noter que WM_MOUSELEAVE n'est pas envoyé automatiquement. Pour recevoir ce message, il faut avoir préalablement appelé la fonction TrackMouseEvent qui n'existe que depuis Windows 2000.
Voici un exemple de création et d'utilisation d'un owner-draw button.

```
#define _WIN32_WINNT 0x0500 /* TrackMouseEvent n'existe que depuis Windows 2000 (Windows NT 5.0). */

#include <windows.h>

/* La structure MYAPP regroupe les variables globales utilisées par notre application. */

typedef struct _MYAPP {
    HWND hCtl; /* Handle du contrôle actuellement survolé par la souris. */
} MYAPP;

MYAPP MyApp;
WNDPROC DefButtonProc; /* "Pointeur" vers l'ancienne procédure de fenêtre de notre bouton. */

#define BUTTON_WIDTH 100
#define BUTTON_HEIGHT 24
#define BUTTON_TEXT "QUIT"

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
void OnCreate(HWND hwnd);
LRESULT CALLBACK ButtonProc(HWND hButton, UINT message, WPARAM wParam, LPARAM lParam);
void OnDrawItem(HWND hwnd, LPDRAWITEMSTRUCT lpdis);
void OnCommand(HWND hwnd, int CtrlID, int Event);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
```

```
{
    WNDCLASS wc;
    HWND hWnd;
    MSG msg;

    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hbrBackground  = CreateSolidBrush(RGB(0x99, 0xCC, 0xFF));
    wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wc.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wc.hInstance      = hInstance;
    wc.lpfnWndProc    = WndProc;
    wc.lpszClassName  = "Classe 1";
    wc.lpszMenuName   = NULL;
    wc.style          = CS_HREDRAW | CS_VREDRAW;

    RegisterClass(&wc);

    hWnd = CreateWindow( "Classe 1", "Owner-Draw Button",
        WS_POPUP |WS_BORDER | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX,
        100, 100, 200, 100, NULL, NULL, hInstance, NULL
    );

    ShowWindow(hWnd, nCmdShow);

    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return (int)msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch(message)
    {
    case WM_CREATE:
        OnCreate(hwnd);
        break;

    case WM_DRAWITEM:
        OnDrawItem(hwnd, (LPDRAWITEMSTRUCT)lParam);
        break;

    case WM_COMMAND:
        OnCommand(hwnd, LOWORD(wParam), HIWORD(wParam));
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0L;
}

void OnCreate(HWND hwnd)
{
    RECT r;
    int x, y;
    HWND hButton;

    /* Centrer le bouton. */

    GetClientRect(hwnd, &r);
    x = (r.right - BUTTON_WIDTH) / 2;
    y = (r.bottom - BUTTON_HEIGHT) / 2;
```

```
    hButton = CreateWindow( "BUTTON", "", WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
        x, y, BUTTON_WIDTH, BUTTON_HEIGHT, hwnd, (HMENU)1, NULL, NULL
    );

    /* On modifie la procédure de fenêtre du bouton. */

    DefButtonProc = (WNDPROC)SetWindowLongPtr(hButton, GWLP_WNDPROC, (LONG_PTR)ButtonProc);
}

LRESULT CALLBACK ButtonProc(HWND hButton, UINT message, WPARAM wParam, LPARAM lParam)
{
    LRESULT ret = 0L;

    /* A la fin du traitement d'un message, il reste parfois nécessaire d'appeler l'ancienne  */
    /* procédure de fenêtre. Il n'est cependant pas garanti que DefButtonProc soit réellement */
    /* un pointeur de fonction. Il faut appeler CallWindowProc pour utiliser DefButtonProc.   */

    switch(message)
    {
    case WM_MOUSEMOVE: /* Quand la souris survole le bouton ... */

        if (MyApp.hCtl != hButton)
        {
            TRACKMOUSEEVENT tme;

            MyApp.hCtl = hButton; /* Le contrôle actuellement survolé par la souris est hButton. */

            /* Générer WM_MOUSELEAVE quand la souris quitte le bouton. */

            tme.cbSize = sizeof(tme);
            tme.dwFlags = TME_LEAVE;
            tme.hwndTrack = hButton;
            TrackMouseEvent(&tme);

            InvalidateRect(hButton, NULL, TRUE); /* Redessiner le bouton. */
        }

        ret = CallWindowProc(DefButtonProc, hButton, message, wParam, lParam);

        break;

    case WM_MOUSELEAVE: /* Quand la souris quitte le bouton ... */

        MyApp.hCtl = NULL; /* Aucun contrôle est actuellement survolé par la souris. */

        InvalidateRect(hButton, NULL, TRUE); /* Redessiner le bouton. */

        ret = CallWindowProc(DefButtonProc, hButton, message, wParam, lParam);

        break;

    default:

        ret = CallWindowProc(DefButtonProc, hButton, message, wParam, lParam);
    }

    return ret;
}

void OnDrawItem(HWND hwnd, LPDRAWITEMSTRUCT lpdis)
{
    if (lpdis->CtlID == 1) /* 1 est l'ID de notre bouton. */
    {
        HDC hDC = lpdis->hDC;
        LPRECT lprcItem = &lpdis->rcItem;
        COLORREF Color;
        HBRUSH hBrush;

        if (lpdis->itemState & ODS_SELECTED) /* Le bouton est pressé. */
            Color = RGB(0x33, 0x33, 0xCC);
        else if (MyApp.hCtl == lpdis->hwndItem) /* Le bouton est survolé par la souris. */
```

```
            Color = RGB(0x66, 0x99, 0xFF);
        else /* Le bouton est dans son état normal. */
            Color = RGB(0x66, 0xCC, 0xFF);

        hBrush = CreateSolidBrush(Color);

        SelectObject(hDC, hBrush);
        SetBkMode(hDC, TRANSPARENT);
        Rectangle(hDC, lprcItem->left, lprcItem->top, lprcItem->right, lprcItem->bottom);
        DrawText(hDC, BUTTON_TEXT, (int)strlen(BUTTON_TEXT), lprcItem, DT_CENTER | DT_VCENTER |
 DT_SINGLELINE);

        DeleteObject(hBrush);
    }
}

void OnCommand(HWND hwnd, int CtrlID, int Event)
{
    if (CtrlID == 1 && Event == BN_CLICKED)
        SendMessage(hwnd, WM_CLOSE, 0, 0);
}
```

II - Les boîtes de dialogue

II-A - Introduction

Une **boîte de dialogue** est une fenêtre, généralement temporaire, qui contient normalement un ou plusieurs contrôles permettant à l'utilisateur d'entrer des informations pour le programme. Cependant, au niveau de la programmation, les boîtes de dialogue ne s'utilisent pas tout à fait de la même façon que les fenêtres « normales » (bien qu'elles soient également des fenêtres) en raison justement de leur caractère temporaire (mais bien entendu, rien ne vous empêche d'avoir une boîte de dialogue comme fenêtre principale).

II-B - Deux types de boîte de dialogue

Une boîte de dialogue peut être créée avec **DialogBox** ou **CreateDialog** (qui sont en fait des macros qui utilisent **CreateWindowEx** en passant bien sûr les bons paramètres). La première crée une boîte de dialogue **modale** et la seconde une boîte de dialogue **non modale**. Une boîte de dialogue modale suspend le travail de sa fenêtre parent jusqu'à ce que l'utilisateur en a terminé avec. Une boîte de dialogue non modale est une fenêtre qui attend des informations venant de l'utilisateur sans suspendre le travail des autres fenêtres. Je ne vous apprends donc rien, puisque vous l'avez certainement deviné, en affirmant que DialogBox attend que l'utilisateur en ait fini avec la fenêtre (la boîte de dialogue) avant de retourner tandis que CreateDialog retourne immédiatement après l'appel.
Il faut également savoir qu'une boîte de dialogue modale sera toujours affichée par Windows que le style **WS_VISIBLE** ait été spécifié ou non. Par contre Windows n'affiche pas automatiquement une boîte de dialogue non modale (il faut donc avoir spécifié ou alors appeler la fonction **ShowWindow**).
La plupart du temps, une boîte de dialogue possède une **fenêtre parent**. Sauf dans le cas où la elle est utilisée comme fenêtre principale de l'application, il est en fortement conseillé de toujours spécifier une fenêtre parent lors de sa création. En effet, lorsqu'on ne spécifie aucun parent, la boîte de dialogue devient complètement autonome ce qui va évidemment compliquer sa gestion.

II-C - Traitement

La boîte de dialogue doit tout d'abord être décrite (par exemple en ressource) puis créée à l'aide de **DialogBox** ou **CreateDialog**.

```
INT_PTR DialogBox(HINSTANCE hInstance, LPCTSTR lpTemplate, HWND hWnd, DLGPROC lpDialogProc);
```

Où lpTemplate est le nom de la boîte de dialogue qu'on veut afficher et lpDialogProc l'adresse d'une fonction qui sera la procédure de traitement des messages.

```
INT_PTR CALLBACK DialogProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
```

Cette procédure doit retourner TRUE si le message a été traité, FALSE dans le cas contraire (sachez également que, dans les anciennes versions du SDK Windows, cette fonction ne retournait pas un INT_PTR mais BOOL). Donc si FALSE est retourné, Windows va effectuer le traitement par défaut. On n'a donc plus besoin de DefWindowProc. Et enfin, sachez qu'avec les boîtes de dialogue, le message WM_CREATE est remplacé par le message **WM_INITDIALOG**. Si l'on souhaite passer des paramètres à travers ce message, On utilisera **DialogBoxParam** ou **CreateDialogParam** à la place de DialogBox ou CreateDialog.

```
INT_PTR DialogBoxParam( HINSTANCE hInstance, LPCTSTR lpTemplate, HWND hWnd,
                        DLGPROC lpDialogProc, LPARAM dwInitParam );
```

II-D - Création

Nous allons maintenant voir comment créer un **modèle** (ou **template**) de boîte de dialogue (autrement dit : une boîte de dialogue !) en ressource. Là encore, comme vous pouvez le constater, les éditeurs de ressources sont plus que jamais d'une aide précieuse mais ici nous allons quand même taper le code nous-mêmes.

```
#include <windows.h>

MyDialog  DIALOGEX  0, 0, 100, 50
STYLE     DS_CENTER | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION   "Ma boîte de dialogue"
{
    CTEXT          "Hello, world !", -1, 0, 8, 100, 12
    DEFPUSHBUTTON  "OK", IDOK, 25, 24, 50, 14
}
```

DIALOGEX est une amélioration de DIALOG (que pouvez toutefois continuer à utiliser, mais cela est-il justifié ?)

```
_name    DIALOGEX _x, _y, _width, _height
STYLE    _style
CAPTION  _caption
FONT     _pointsize, _facename, _weight, _italic, _charset
MENU     _idmenu
{
         _content
}
```

Pour les contrôles, la syntaxe est :

```
<TYPE DU CONTROLE> _text, _id, _x, _y, _width, _height, _style, _exstyle
```

Où le type du contrôle peut être LTEXT, RTEXT, CTEXT (contrôles Static), PUSHBUTTON, DEFPUSHBUTTON, RADIOBUTTON, AUTORADIOBUTTON, CHECKBOX, AUTOCHECKBOX, STATE3, AUTO3STATE, PUSHBOX, GROUPBOX, EDITTEXT, LISTBOX, COMBOXBOX ou SCROLLBAR. Bien entendu, ces contrôles s'utilisent de la même façon que les contrôles créés dynamiquement.
On peut également créer un contrôle avec le mot-clé générique **CONTROL** :

```
CONTROL _text, _id, _classname, _x, _y, _width, _height, _style, _exstyle
```

Attention ! L'unité utilisée dans les boîtes de dialogue n'est pas le pixel mais les **Dialog Template Units**. Ce ne sont pas des unités fixes. L'interêt d'utiliser de telles unités est de pouvoir créer des boîtes de dialogue ayant la même proportion indépendamment de la résolution de l'écran. Alors juste pour info, on appelle **Dialog Base Units** d'une boîte de dialogue la largeur et la hauteur moyennes des caractères dans la police utilisée. Si la boîte de dialogue utilise la police système, on peut récupérer leurs équivalents en pixels à l'aide de la fonction **GetDialogBaseUnits**. Sinon, un autre moyen très simple de les récupérer est d'utiliser la fonction **GetTextEntentPoint32** sachant qu'on appelle largeur moyenne des caractères dans une police donnée la largeur du caractère x. Si on connaît les Dialog Base Units, on peut directement convertir les Dialog Template Units en pixels sachant que :

```
1 XDialogTemplateUnit = 1/4 (XDialogBaseUnit)
1 YDialogTemplateUnit = 1/8 (YDialogBaseUnit)
```

En réalité, pour convertir les Dialog Template Units en pixels, on utilisera tout simplement la fonction **MapDialogRect**.

II-E - Une boîte de dialogue comme fenêtre principale

Nous allons enfin voir un exemple qui affiche une boîte de dialogue comme fenêtre principale. Voici donc le code :

```
include <windows.h>

INT_PTR CALLBACK DialogProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    DialogBox(hInstance, "MyDialog", NULL, DialogProc);
    return 0;
}

INT_PTR CALLBACK DialogProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch(message)
    {
    case WM_INITDIALOG:
        break;

    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
        case IDOK:
            SendMessage(hwnd, WM_CLOSE, 0, 0);
            break;
        }

        break;

    case WM_CLOSE:
        EndDialog(hwnd, 0);
        break;

    default:
        return FALSE;
    }

    return TRUE;
}
```

La fonction **EndDialog** permet de détruire une boîte de dialogue modale. La valeur passée en dernier paramètre sera utilisée par **DialogBox** comme valeur de retour. Dans le cas d'une boîte de dialogue non modale, on utilise plutôt la fonction DestroyWindow (en fait, EndDialog utilise en interne DestroyWindow mais après avoir recupéré la valeur que DialogBox doit retourner).

II-F - Les boîtes de dialogue communes

Dans les paragraphes précédents nous avons appris à utiliser des boîtes de dialogue que nous avons nous-mêmes créées. Ici ce qui nous intéressera ce sont les **boîtes de dialogue communes**, c'est-à-dire celles qui sont fournies par Windows. Il s'agit des boîtes de dialogue Ouvrir, Enregistrer (Enregistrer sous), Imprimer, etc. En particulier, sachez que le boîtes de dialogue Ouvrir et Enregistrer sous sont en fait les mêmes (ou presque) ! Pour afficher la boîte de dialogue (Ouvrir ou Enregistrer sous), on initialise une structure de type **OPENFILENAME** qui décrit la boîte que l'on veut afficher ensuite on appelle **GetOpenFileName** pour afficher la boîte de dialogue Ouvrir et **GetSaveFileName** pour afficher la boîte de dialogue Enregistrer sous (ces boîtes de dialogue ont la même apparence, la seule véritable différence vient du texte du bouton par défaut qui est Ouvrir pour la boîte de dialogue Ouvrir et Enregistrer pour la boîte de dialogue Enregistrer sous), et c'est aussi simple que cela ! Par exemple :

```
OPENFILENAME ofn;

char lpszFile[MAX_PATH] = "";
char lpszFileTitle[MAX_PATH] = "";

ZeroMemory(&ofn, sizeof(ofn));
ofn.lStructSize     = sizeof (OPENFILENAME);
ofn.hwndOwner       = hwnd;
ofn.lpstrFilter     = "Texte (*.txt)\0*.txt\0Tous (*.*)\0*.*\0";
ofn.lpstrFile       = lpszFile;
ofn.nMaxFile        = sizeof(lpszFile) / sizeof(lpszFile[0]);
ofn.lpstrFileTitle  = lpszFileTitle;
ofn.nMaxFileTitle   = sizeof(lpszFileTitle) / sizeof(lpszFileTitle[0]);
ofn.Flags           = OFN_EXPLORER | OFN_HIDEREADONLY | OFN_CREATEPROMPT;

if (GetOpenFileName(&ofn))
     MessageBox(hwnd, lpszFile, lpszFileTitle, MB_OK);
else
     MessageBox(hwnd, "Opération annulée", "GetOpenFileName", MB_OK);
```

Sachez cependant que OPENFILENAME est une structure particulièrement riche et que nous n'avons ici utilisé que quelques uns de ses champs seulement. Elle est définie dans **commdlg.h** (inclus par windows.h) de la manière suivante :

```
typedef struct tagOFN {
    DWORD          lStructSize;        /
* [in]     Taille de la structure                                  */
    HWND           hwndOwner;          /
* [in]     Fenêtre parent de la boîte de dialogue                  */
    HINSTANCE      hInstance;          /
* [in]     Handle d'un module contenant une boîte de dialogue (1)  */
    LPCTSTR        lpstrFilter;        /
* [in]     Spécifie les filtres disponibles                        */
    LPTSTR         lpstrCustomFilter;  /* [in/
out] Contient le filtre préféré de l'utilisateur             */
    DWORD          nMaxCustFilter;     /
* [in]     Largeur du buffer lpstrCustomFilter                     */
    DWORD          nFilterIndex;       /* [in/
out] Contient l'index du filtre sélectionné (2)              */
    LPTSTR         lpstrFile;          /* [in/
out] Contient le nom du fic. Doit être initialisé            */
    DWORD          nMaxFile;           /
* [in]     Largeur du buffer lpstrFile                             */
    LPTSTR         lpstrFileTitle;     /* [in/
out] Contient le nom et l'ext. seulement du fic.             */
    DWORD          nMaxFileTitle;      /
* [in]     Largeur du buffer lpstrFileTitle                        */
    LPCTSTR        lpstrInitialDir;    /
* [in]     Pointeur vers le chemin du répertoire initial           */
    LPCTSTR        lpstrTitle;         /
* [in]     Titre (caption) de la boîte de dialogue                 */
    DWORD          Flags;              /* [in/
out] Style et comportement de la boîte de dialogue           */
    WORD           nFileOffset;        /
* [out]    Indice du 1er caractère du nom du fic. dans lpstrFile   */
    WORD           nFileExtension;     /
* [out]    Indice du 1er caractère de l'ext. du fic. dans lpstrFile */
    LPCTSTR        lpstrDefExt;        /
* [in]     Ext. par défaut (3)                                     */
    LPARAM         lCustData;          /
* [in]     Paramètre à passer la lpfnHook via le msg WM_INITDIALOG  */
    LPOFNHOOKPROC  lpfnHook;           /
* [in]     DlgProc de la boîte de dialogue (4)                     */
    LPCTSTR        lpTemplateName;     /
* [in]     Nom de la boîte de dialogue à charger                   */

#if (_WIN32_WINNT >= 0x0500)
    void *         pvReserved;         /
* [x]      Reservé. Doit être NULL                                 */
```

```
    DWORD           dwReserved;         /
* [x]      Reservé. Doit être 0                                    */
    DWORD           FlagsEx;            /
* [in]     Flags étendus                                           */
#endif /* (_WIN32_WINNT >= 0x0500) */

} OPENFILENAME, *LPOPENFILENAME;

/
* (1) : Requiert le flag OFN_ENABLETEMPLATEHANDLE.                                                    */
/
* (2) : En entrée, 0 sélectionne le filtre préféré.                                                   */
/
* (3) : Vous pouvez spéifiez une chaîne de 3 caractères tout au plus. Omettez le point.               */
/
* (4) : Requiert le flag OFN_ENABLEHOOK.                                                              */
```

Voici une liste des flags les plus courants :
OFN_ALLOWMULTISELECT : Autorise la sélection multiple.
OFN_CREATEPROMPT : Indique que l'utilisateur peut entrer un nom de fichier qui n'existe pas et dans ce cas il sera invité à confirmer son choix.
OFN_EXPLORER : Impose le style Explorer. En fait ce style est appliqué par défaut mais il est parfois annulé par certains flags. Ce flag permet de toujours utiliser le style Explorer.
OFN_FILEMUSTEXIST : Utilisable uniquement dans une boîte de dialogue Ouvrir. Permet de spécifier que l'utilisateur doit entrer un nom de fichier existant. Inclut le flag **OFN_PATHMUSTEXIST**.
OFN_FORCESHOWHIDDEN : Windows 2000 et plus récents uniquement. Affiche les dossiers et fichiers cachés et/ou systèmes, indépendamment des préférences de l'utilisateur.
OFN_HIDEREADONLY : N'a de sens que dans une boîte de dialogue Ouvrir. Cache l'option « ouvrir en lecture seulement ».
OFN_NODEREFERENCELINKS : Par défaut, si l'utilisateur sélectionne un raccourci (.lnk), le champ lpstrFile contient le chemin vers le fichier pointé par le lien. Le flag OFN_NODEREFERENCELINKS permet de spécifier que l'on désire dans un tel cas avoir le nom du lien et non celui du fichier pointé.
OFN_OVERWRITEPROMPT : Dans une boîte de dialogue Enregistrer sous, provoque une demande de confirmation si l'utilisateur sélectionne un nom de fichier existant.
OFN_PATHMUSTEXIST : Spécifie que l'utilisateur doit utiliser des caractères valides uniquement.
OFN_READONLY : N'a de sens que dans une boîte de dialogue Ouvrir. Coche l'option « ouvrir en lecture seulement ». Au retour de la fonction, ce flag indique l'état de ladite option.

III - Les menus

III-A - Création dynamique d'un menu

La fonction qui permet de créer un menu est **CreateMenu**. Après qu'on ait créé un menu, on peut lui ajouter des éléments à l'aide de la fonction **AppendMenu**. Evidemment un menu peut comporter d'autres menus. Par exemple les menus Fichier, Edition, Affichage etc. sont des sous menus du menu principal. Chacun de ces menus peuvent ensuite encore comporter d'autres menus et ainsi de suite. Le problème c'est que les éléments « terminaux » (tels que Enregistrer, Fermer, Quitter, etc.) sont également appelés menus ce qui est généralement source de confusion. On se réfère donc aux termes anglais « popup » pour désigner un menu qui peut en comporter d'autres (mais qui n'est pas le menu principal) et « menu item » pour désigner un élément terminal d'un menu. En résumé, **« menu »** est un terme générique (et confus) qui peut être à la fois utilisé pour désigner un **menu principal**, un **popup menu** ou un **menu item**. Notez également qu'un menu principal a toutes les fonctionnalités d'un popup menu mais ce n'est pas un popup menu (c'est un menu principal !).
Le fonctionnement des menus est très similaire à celui des boutons au sens où le fait de cliquer sur un menu (terminal) provoque l'émission du message **WM_COMMAND** avec comme paramètres l'identificateur du menu (LOWORD(wParam)) et NULL (lParam).
Dans l'exemple suivant, on va créer un menu (le menu principal) qui contiendra juste un sous menu : Fichier, qui contiendra à son tour 5 éléments terminaux à savoir Nouveau, Ouvrir, Enregistrer, Enregistrer sous et Quitter. Un séparateur (qui se présente sous forme d'une ligne horizontale) sera placé entre les 4 premiers et la dernière. On peut créer le menu avant ou après la fenêtre, tout dépend de la manière dont on veut s'y prendre. La première solution

permet de spécifier le handle du menu directement dans l'argument hMenu de **CreateWindow**. Dans cet exemple, c'est la deuxième méthode que nous allons utiliser.

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASS wc;
    HWND hWnd;
    MSG msg;

    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
    wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wc.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wc.hInstance      = hInstance;
    wc.lpfnWndProc    = WndProc;
    wc.lpszClassName  = "Menu Demo";
    wc.lpszMenuName   = NULL;
    wc.style          = CS_HREDRAW | CS_VREDRAW;

    RegisterClass(&wc);

    hWnd = CreateWindow("Menu Demo", "M-D", WS_OVERLAPPEDWINDOW, 100, 100, 600, 300, NULL, NULL,
 hInstance, NULL);

    ShowWindow(hWnd, nCmdShow);

    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return (int)msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HMENU hMenu, hFichier;

    switch(message)
    {
    case WM_CREATE:
        hMenu = CreateMenu();
        hFichier = CreateMenu();

        AppendMenu(hFichier, MF_STRING, (UINT_PTR)1, "&Nouveau");
        AppendMenu(hFichier, MF_STRING, (UINT_PTR)2, "&Ouvrir");
        AppendMenu(hFichier, MF_STRING, (UINT_PTR)3, "&Enregistrer");
        AppendMenu(hFichier, MF_STRING, (UINT_PTR)4, "En&registrer sous ...");
        AppendMenu(hFichier, MF_SEPARATOR, (UINT_PTR)-1, "");
        AppendMenu(hFichier, MF_STRING, (UINT_PTR)5, "&Quitter");
        AppendMenu(hMenu, MF_POPUP, (UINT_PTR)hFichier, "&Fichier");

        SetMenu(hwnd, hMenu);

        break;

    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
        case 1:
            MessageBox(hwnd, "Crée un nouveau document", "M-D", MB_OK);
            break;

        case 2:
            MessageBox(hwnd, "Ouvre un document", "M-D", MB_OK);
```

```
            break;

        case 3:
            MessageBox(hwnd, "Enregistre le document", "M-D", MB_OK);
            break;

        case 4:
            MessageBox(hwnd, "Enregistre sous un autre nom", "M-D", MB_OK);
            break;

        case 5:
            DestroyWindow(hwnd);
            break;
        }

        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

Vous-vous demandez certainement pourquoi il n'est pas nécessaire de détruire le menu avant de quitter le programme. La réponse est : si, il faut le détruire (avec **DestroyMenu**). Cependant, lorsqu'un menu est attaché à une fenêtre, il sera automatiquement détruit en même temps que cette fenêtre. D'accord, mais qu'en est-il des sous menus ? Ils ont également été détruits en même temps que le menu car DestroyMenu est récursive, c'est-à-dire que la destruction d'un menu entraîne également la destruction de ses sous-menus.

III-B - Création à l'aide d'un fichier de ressources

On peut également créer un menu en ressource. Dans le programme, il ne restera plus qu'à charger cette ressource avec la fonction **LoadMenu**, qui retournera le handle du menu en question.

```
HMENU LoadMenu(HINSTANCE hInstance, LPCTSTR lpMenuName);
```

Les éditeurs de ressources simplifient beaucoup la création de menu mais à défaut, on peut toujours taper le code nous-mêmes. Par exemple :

```
MyMenu MENU
{
    POPUP "&Fichier"
    {
        MENUITEM "&Nouveau", 1
        MENUITEM "&Ouvrir ...", 2
        MENUITEM "&Enregistrer", 3
        MENUITEM "En&registrer sous ...", 4
        MENUITEM SEPARATOR
        MENUITEM "&Quitter", 5
    }
}
```

On peut également directement spécifier le nom du menu (ici "MyMenu") dans le membre **lpszMenuName** de la structure **WNDCLASS** et dans ce cas on n'a plus besoin ni de LoadMenu, ni de SetMenu. En outre, l'argument hMenu de CreateWindow doit être tout simplement NULL.

III-C - Les menus contextuels

Dans une application standard, on affiche normalement un menu contextuel lorsqu'on reçoit le message **WM_CONTEXTMENU** qui, rappelons-le, est envoyé lorsque l'utilisateur a cliqué avec le bouton droit à l'intérieur de la fenêtre.
Un menu contextuel est un menu comme tous les autres. N'importe quel popup peut être affiché en tant que menu contextuel. Notez bien cependant que le menu doit être obligatoirement un popup menu, pas n'importe quel menu. Il faut donc utiliser **CreatePopupMenu** à la place de CreateMenu. A noter également que dans l'exemple précédent (création dynamique), on aurait aussi pu créer le menu Fichier à l'aide de CreatePopupMenu (puisqu'il s'agit bien d'un popup menu) par contre on ne peut pas utiliser cette fonction pour créer le menu principal (qui n'est pas un popup menu ...). Et enfin, sachez que, puisqu'ils sont reconnus par le système comme étant de vrais popup menus, les sousmenus, peu importe la manière dont ils on été créés (en ressource, avec CreateMenu, avec CreatePopupMenu, etc.), peuvent également être utilisés. Si on n'a pas le handle du sous menu (ce qui est fréquemment le cas lorsque le menu a été créé en ressource), on peut toujours le récupérer avec **GetSubMenu**. La fonction **GetMenu**, qui retourne le handle du menu d'une fenêtre, peut également s'avérer utile.

```
HMENU GetMenu(HWND hWnd);
HMENU GetSubMenu(HMENU hMenu, int nPos);
```

La fonction **TrackPopupMenu** permet d'afficher le menu.

```
BOOL TrackPopupMenu(HMENU hPopup, UINT uFlags, int x, int y, int nReserved, HWND hWnd, const RECT *
 prcIgnored);
```

Le dernier paramètre est de la décoration. Il est ignoré par Windows ! L'extrait de code suivant permet d'afficher un menu contextuel :

```
case WM_CONTEXTMENU:
    TrackPopupMenu(hFichier, 0, LOWORD(lParam), HIWORD(lParam), 0, hwnd, NULL);
    break;
```

III-D - Autres fonctions

EnableMenuItem

```
BOOL EnableMenuItem(HMENU hPopup, UINT uItem, UINT uEnable);
```

Le paramètre uEnable peut prendre l'une des valeurs suivantes :

- MF_ENABLED : active le menu de sorte que l'utilisateur peut le sélectionner
- MF_DISABLED : désactive le menu de sorte que l'utilisateur ne peut pas le sélectionner mais ne change pas son apparence
- MF_GRAYED : désactive le menu de sorte que l'utilisateur ne peut pas le sélectionner et change son apparence

En fait, on peut combiner ces valeurs avec MF_BYCOMMAND (par défaut) ou MF_BYPOSITION. MF_BYCOMMAND entraîne que uItem fait référence à l'ID du menu que l'on veut activer ou désactiver et MF_BYPOSITION que uItem fait référence à l'indice du menu à activer / désactiver (le premier élément possède l'indice 0).

CheckMenuItem

```
BOOL CheckMenuItem(HMENU hPopup, UINT uItem, UINT uCheck);
```

Les valeurs utilisables dans l'argument uCheck MF_CHECKED, MF_UNCHECKED, MF_BYCOMMAND et MF_BYPOSITION.

III-E - Les accélérateurs claviers

Les **accélérateurs** (ou **raccourcis**) claviers sont des combinaisons de touches qui permettent à l'utilisateur d'émettre une commande (c'est-à-dire le message WM_COMMAND) à l'aide du clavier. Ils sont généralement utilisés comme raccourcis vers une commande proposée par un menu ou un bouton par exemple. On va par exemple créer les raccourcis suivants pour notre menu Fichier : Ctrl + N (Nouveau), Ctrl + O (Ouvrir), Ctrl + S (Enregistrer) et F12 Enregistrer sous.
Traiter le message WM_KEYDOWN puis tester l'état de la touche Ctrl est inefficace car ce message n'est émis que si la fenêtre a le focus du clavier. Dès que le focus passe à un contrôle, ce message peut ne pas être envoyé. C'est pourquoi il faut utiliser les accélérateurs. Tout ce qu'on a à faire c'est d'associer les combinaisons de touches Ctrl + N, Ctrl + O, Ctrl + S et F12 aux commandes 1 (Nouveau), 2 (Ouvrir), 3 (Enregistrer) et 4 (Enregistrer sous). On peut faire cette association dynamiquement à l'aide de la fonction **CreateAcceleratorTable** mais c'est plus simple et plus vite de la créer en ressource.

```
#include <windows.h> /* pour VK_F12 (défini dans winuser.h) */

MyAccels ACCELERATORS
{
    "N",     1,  CONTROL,  VIRTKEY
    "O",     2,  CONTROL,  VIRTKEY
    "S",     3,  CONTROL,  VIRTKEY
    VK_F12,  4,            VIRTKEY
}
```

Le flag **VIRTKEY** spécifie que le code de la touche de l'accélérateur est un code de touche virtuelle et non le code d'un caractère ASCII. Si on omet ce flag, Windows suppose qu'il s'agit d'un code de caractère ASCII (donc "N" par exemple correspondra au caractère "N" majuscule, c'est-à-dire Shift + N, et non la touche N). Pour ce qui est des touches d'accompagnement, on peut utiliser n'importe quelle combinaison des flags CONTROL, ALT et SHIFT ou aucune.
Il faut ensuite charger la ressource.

```
HACCEL hAccel = LoadAccelerators(hInstance, "MyAccels");
```

Et enfin, adapter la boucle des messages.

```
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(hWnd, hAccel, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

La fonction **TranslateAccelerator** examine un message et si ce dernier provient du clavier et correspond à une entrée dans la table des accélérateurs, il sera traité comme il se doit et la fonction retourne TRUE, sinon FALSE est retournée. Si la fonction réussit, le message a donc déjà été traité et il ne faut plus appeler ni Translate ni Dispatch Message.

IV - Remerciements

Je tiens particulièrement à remercier **vicenzo** qui s'est beaucoup donné pour m'aider à finir ce tutoriel.