**API Windows en C.**

**Par CGi**

**Sommaire :**

1 - La fenêtre principale.

2 - Contrôles et messages.

3 - Les commandes et notifications.

4 - Les boîtes de dialogue communes.

5 - Les contextes de périphériques.

6 - Dessiner.

7 - Les ressources.

8 - Les boîtes de dialogue personnalisées.

9 - Communiquer avec les boîtes de dialogue.

10 - Les accélérateurs clavier.

11 - Une boîte de dialogue comme fenêtre principale.

12 - Les contrôles supplémentaires de Windows.

13 - Donner le style XP à vos application.

14 - Sous classement de fenêtres.

15 - Afficher une image.

**Les boîtes de dialogue :**

- la boîte de dialogue d'ouverture de fichier. (Chap. 4)

- la boîte de dialogue de sauvegarde.

- la boîte de dialogue de choix de police de caractères.

- la boîte de dialogue de choix de couleur.

**Les contrôles :**

- Le contrôle d'édition. (Chap. 9)

- Le contrôle CheckBox ou case à cocher.

- Le contrôle Radiobutton ou bouton radio.

- Le contrôle ListBox ou liste de choix.

- Le contrôle ComboBox ou liste déroulante.

- Le contrôle Scroll bar ou ascenseur.

- Le contrôle Static.

- Le contrôle Progress Bar ou Barre de progression. (Chap. 12)

- Le contrôle Status Bar ou Barre d'état.

- Le contrôle TrackBar ou Slider.

- Le contrôle List View.

- Le contrôle Tree View.

- Le contrôle Tab ou boîte à onglets.

- Le contrôle Toolbar.

- Le contrôle ToolTips.

- Le contrôle PropertySheet.

**Divers :**

- Les Timers.

- Les menus contextuels.

- Les boîtes de dialogues non modales.

- Path and clip.

- Le Tray Icon.

- Enumération de ressources.

- Afficher une image sous Windows NT.

- Les services NT.

- Une seule instance pour l'application.

**Bonne lecture,**

CGi.

**1 - La fenêtre principale.**

**Introduction :**

Windows est un système d'exploitation proposant une interface graphique. Dans ce premier article, nous allons créer une application composée seulement d'une fenêtre principale, dont le but et de montrer l'architecture d'un programme Windows.



Prenez garde au terme de fenêtre, dans Windows. Il signifie objet visuel, donc un bouton est une fenêtre ! Celle apparaissant au démarrage de l'application est souvent appelée fenêtre principale. Nous emploierons tout de même le mot fenêtre pour désigner une fenêtre du type de la fenêtre principale et le mot contrôle pour désigner les objets visuels qu'elles peuvent contenir...

**La fonction WinMain :**

Le point d'entrée d'une application Windows est la fonction **WinMain**. C'est l'équivalent de la fonction main des applications classiques. Elle est appelée par le système d'exploitation au lancement du programme. Il lui fournit 4 paramètres.

**int** WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, **int** nCmdShow);

Le premier paramètre est le handle d'instance de l'application. C'est un numéro unique attribué par le système d'exploitation qui lui permet de l'identifier.
Le second paramètre est toujours NULL pour les applications Win32.
Le troisième paramètre est un pointeur sur la ligne de commande.
Et enfin le quatrième et dernier paramètre est un identificateur qui indique comment Windows désire afficher la fenêtre (normal, caché, minimisé ...).
C'est donc dans cette fonction (WinMain) que nous allons créer la fenêtre principale.

**Création de la fenêtre :**

La fonction qui permet de créer une fenêtre se nomme **CreateWindow** :

HWND CreateWindow(

 LPCTSTR lpClassName, // Pointeur sur une classe de fenêtre.

 LPCTSTR lpWindowName, // Pointeur sur le texte de la fenêtre.

 DWORD dwStyle, // Style de la fenêtre.

 **int** x, // Position horizontale de la fenêtre.

 **int** y, // Position verticale de la fenêtre.

 **int** nWidth, // Largeur de la fenêtre.

 **int** nHeight, // Hauteur de la fenêtre.

 HWND hWndParent, // Handle de la fenêtre parent.

 HMENU hMenu, // Handle de menu ou ID de contrôle.

 HANDLE hInstance, // Handle d'instance de l'application.

 LPVOID lpParam // Pointeur sur des données passées à WM\_CREATE.

 );

**En voici un exemple d'utilisation :**

 HWND hwnd;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

Le premier paramètre qu'elle reçoit est un pointeur sur une chaîne de caractères identifiant la classe de fenêtre. Les classes de fenêtre sont des modéles pour construire les fenêtres (le terme de classe n'a rien à voir avec les classes du C++). Si pour les contrôles standard nous avons des classes de fenêtres prédéfinies et globale, nous devrons en créer une pour la fenêtre principale. Nous devons pour cela remplir une structure de type WNDCLASS (définie dans winuser.h)

 WNDCLASS wc;

 wc.style = 0;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = (HBRUSH)(1 + COLOR\_BTNFACE);

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

Les champs qu'il nous importe d'affecter pour l'instant à cette structure sont :
Le champ *lpszClassName* avec un pointeur sur la chaîne de caractères identifiant la classe de fenêtre. Celle que l'on a passé comme paramètre à la fonction CreateWindow.
Le champ *hInstance* sera initialisé avec le handle d'instance hinstance que nous a fourni la fonction WinMain.
Le champ *hIcon* avec le handle d'une icône standard de Windows qui nous est retournée par la fonction LoadIcon.
Le champ *hCursor* avec le handle d'un curseur prédéfini de Windows (en l'occurrence la flèche) qui nous est retourné par la fonction LoadCursor.
Le champ *hbrBackground* avec le handle d'un pinceau qui servira à peindre la surface client de la fenêtre. Dans cet exemple nous lui affecterons une couleur prédéfinie du système *(1 + (l'identificateur de la couleur))*.
Le champ *lpfnWndProc* de type WNDPROC reçoit un pointeur de fonction sur la procédure de fenêtre que nous allons voir plus loin dans ce document.
Nous passons sous silence pour l'instant les autres champs de cette structure.
Remplir cette structure ne suffit pas, il faut aussi l'enregistrer au niveau du sytème (souvenez-vous qu'elles sont identifiées par une chaîne de caractères). Cet enregistrement sera effectué par la fonction **RegisterClass** qui reçoit comme paramètre l'adresse de la variable wc de type WNDCLASS créée précédemment :

 RegisterClass(&wc)

Revenons-en maintenant à la fonction CreateWindow. Son deuxième paramètre reçoit un pointeur sur le texte de la fenêtre. Dans le cas de la fenêtre principale, ce sera le texte du titre (si cela avait été un bouton, le texte se serait affiché sur le bouton).
Le troisième paramètre est le style de la fenêtre. Ce sont des identificateurs qui sont défini dans le fichier winuser.h. Nous ne les passerons pas en revue car ils sont trop nombreux. Dans l'exemple, nous lui donnerons la valeur WS\_OVERLAPPEDWINDOW. Ce paramètre permet d'avoir une fenêtre avec une barre de titre, un menu system, les boutons minimiser et maximiser et que l'on peut redimensionner.
Les deux paramètres suivants sont la position à l'écran de la fenêtre par rapport au coin haut gauche du bureau. Dans l'exemple, ils ont la valeur par défaut fournie par Windows : CW\_USEDEFAULT.
Les deux paramètres suivants sont la taille de la fenêtre : largeur et hauteur. Ils peuvent aussi avoir la valeur par défaut CW\_USEDEFAULT.
Le paramètre suivant désigne le parent de la fenêtre. Il est utile pour les contrôles enfants. En ce qui concerne la fenêtre principale nous l'initialiserons à NULL.
Le paramètre suivant désigne un menu pour la fenêtre, il peut être à NULL s'il n'y a pas de menu ou si le menu a été référencé dans le champ lpszMenuName de la classe de fenêtre.
L'avant-dernier paramètre reçoit le handle d'instance de l'application qui crée la fenêtre, celui qui est fourni par WinMain.
Le dernier paramètre sert à passer des données à la procédure de fenêtre quand elle reçoit un message WM\_CREATE. Dans cet exemple, il sera initialisé à NULL.
Après s'être exécutée, la fonction CreateWindow nous retourne le handle de la fenêtre qu'elle vient de créer. Les handles de fenêtre sont des identificateurs uniques attribués par Windows qui désignent chaque fenêtres. Ils sont de type HWND.
Notre fenêtre étant maintenant créée, il faut la rendre visible par l'appel de la fonction ShowWindow :

 ShowWindow(hwnd, nCmdShow);

Cette fonction reçoit comme paramètre le handle de la fenêtre et un identificateur qui indique comment on désire l'afficher (normal, caché, minimisé ...). En général, on lui donne celui fourni par WinMain (voir plus haut dans ce document).

Ensuite, on appelle UpdateWindow avec comme paramètre le handle de la fenêtre. Cette fonction sert à rafraîchir l'affichage de la zone client. Dans cet exemple, elle n'est pas vraiment utile puisque la zone client est vide.

 UpdateWindow(hwnd);

**La boucle de messages :**

Revenons au fonctionnement de Windows. Pour communiquer avec une application ou ses diverses fenêtres, Windows leurs envoie des messages. Par exemple, si vous cliquez sur le bouton fermeture (en haut à droite de la fenêtre), Windows va créer un message approprié qu'il va envoyer dans la file d'attente de l'application. La file d'attente est un tampon où sont stockés les messages en attente de traitement.
C'est à nous de coder l'extraction des messages de la file d'attente :

 MSG msg;

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

C'est le rôle de la fonction **GetMessage**, dont le premier paramètre est un pointeur sur une variable de type MSG où elle doit stocker les données du message qu'elle vient de récupérer.
Nous devons donc définir cette variable de type **MSG**, qui est une structure composé d'un champ **hwnd** qui contient le handle de la fenêtre à qui il est destiné, un champ **message** identifiant le message (par exemple WM\_CLOSE pour demander à la fenêtre de ce fermer) et de deux champs **wParam** et **lParam** contenant des données ayant rapport au message et de significations différentes selon le message.
Revenons à GetMessage, son deuxième paramètre est le handle de fenêtre dont on veut récupérer les messages. Il sera mis à NULL car on veut récupérer tous les messages de l'application (destinés à toutes ses fenêtres).
Les deux derniers paramètres seront mis à zéro, ils servent au filtrage des messages.
GetMessage renvoie toujours la valeur TRUE sauf si elle vient de récupérer le message WM\_QUIT, ce qui met fin à la boucle de message donc à l'application.
Ensuite on passe le message à la fonction **TranslateMessage** qui traduit les messages WM\_KEYDOWN (appui des touches clavier) en message WM\_CHAR avec le code de caractère correspondant dans le champ wParam du message.
Ensuite, c'est à la fonction **DispatchMessage** de traiter le message. Son rôle est de l'envoyer à la procédure de fenêtre. Celle à qui il est destiné.

**La procédure de fenêtre :**

Voici la procédure de fenêtre de notre fenêtre principale dont nous avions passé un pointeur à la classe de fenêtre au début de ce document (on passe ce pointeur à la classe de fenêtre car c'est nous qui la créons, mais le système d'exploitation qui l'appelle, il doit donc la localiser).

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 **switch** (uMsg)

 {

 **case** WM\_CREATE:

 **return** 0;

 **case** WM\_DESTROY:

 PostQuitMessage(0);

 **return** 0;

 **default**:

 **return** DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

Les paramètres qu'elle reçoit sont les données du message en cours de traitement. Celui qui a été envoyé par DispatchMessage. C'est à nous, développeurs, d'implémenter des actions en fonction du message que la procédure de fenêtre reçoit. Dans cet exemple, un seul message est traité : WM\_DESTROY. Il nous indique que la fenêtre est fermée et a été détruite. Comme il s'agit de la fenêtre principale de l'application, il est d'usage de fermer l'application. Ce que nous ferons en appelant la fonction **PostQuitMessage** qui a pour fonction de poster un message WM\_QUIT dans la file d'attente. Qui comme nous l'avons vu plus haut dans ce document, met fin à la boucle de messages et donc à l'application. Les messages non traités doivent l'être par la fonction **DefWindowProc**. Fonction qui implémente le comportement par défaut d'une fenêtre.

**Code complet :**

#include <windows.h>

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

**int** WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, **int** nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 wc.style = 0;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = (HBRUSH)(1 + COLOR\_BTNFACE);

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 **if**(!RegisterClass(&wc)) **return** FALSE;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

 **if** (!hwnd) **return** FALSE;

 ShowWindow(hwnd, nCmdShow);

 UpdateWindow(hwnd);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 **return** msg.wParam;

}

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 **switch** (uMsg)

 {

 **case** WM\_CREATE:

 **return** 0;

 **case** WM\_DESTROY:

 PostQuitMessage(0);

 **return** 0;

 **default**:

 **return** DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

**2 - Contrôles et messages :**

**Introduction :**

Dans ce chapitre, nous allons revenir sur le traitement des messages.
Nous allons aussi créer un contrôle enfant que nous allons mettre sur la fenêtre principale.
Ce contrôle sera un contrôle d'édition multi lignes. Il devra se positionner sur toute la surface client de la fenêtre principale (on appelle surface client d'une fenêtre toute sa partie intérieure, c'est-à-dire la fenêtre sans sa bordure et sa barre de titre).



**Création d'un contrôle :**

Comme nous l'avons dit au [chapitre précédent](http://chgi.developpez.com/windows/winmain/) un contrôle est une fenêtre. Nous allons donc utiliser comme pour la fenêtre principale la fonction **CreateWindow** pour le créer.

 HWND hEdit;

 hEdit =CreateWindow("edit", "Texte",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

Pour les contrôles, les classes de fenêtres sont globales et définies par Windows. Celle du contrôle d'édition s'appelle : "EDIT". Nous la mettrons donc comme premier paramètre de CreateWindow.
Son deuxième paramètre attend un pointeur sur une chaîne de caractères au cas ou vous désirez qu'il contienne du texte dès sa création.
Le troisième attend les identificateurs de style. Ils sont ici complètement différents de ce qu'on avait mis pour la fenêtre principale :
WS\_VISIBLE rend la fenêtre visible dés sa création.
WS\_CHILD en fait un contrôle enfant. C'est obligatoire si l'on veut qu'il soit à l'intérieur de notre fenêtre.
ES\_MULTILINE multi lignes si on veut avoir la possibilité de saisir plusieurs lignes.
ES\_WANTRETURN autorise le retour à la ligne automatique.
WS\_VSCROLL pour lui mettre un ascenseur vertical.
(Les identificateurs de style étant très nombreux, je vous propose de consulter l'aide Win32 API CreateWindow, ou CreateWindowEx qui permet des styles supplémentaires).
Les quatre paramètres suivants sont comme pour la fenêtre principale, sa position et ses dimensions. Dans le code complet de l'exemple, elles sont toutes mises à zéro car le contrôle sera redimensionné automatiquement à chaque changement de dimensions de la fenêtre principale (voir plus loin dans ce document).
Le paramètre suivant est le handle de la fenêtre principale car elle en est le parent.
Et comme pour la fenêtre principale le handle d'instance de l'application pour l'avant dernier paramètre.
Dans le code de l'exemple, pour des raisons de simplicité, j'ai créé une variable globale pour le handle d'instance que j'ai initialisé dans la fonction WinMain.

**Les messages :**

Nous avons dit que notre contrôle d'édition devait se redimensionner pour occuper la surface client de la fenêtre principale. Mais comment savoir que les dimensions de la fenêtre principale ont changé ? Par l'intermédiaire d'un message ! Si un utilisateur a changé les dimensions de la fenêtre, Windows va envoyer un message à cette fenêtre pour l'en informer. Dans ce cas précis le message est WM\_SIZE. C'est à nous de traiter ce message dans la procédure de fenêtre :

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 **static** HWND hEdit;

 **switch** (uMsg)

 {

 **case** WM\_CREATE:

 hEdit =CreateWindow("edit", "Texte",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 **return** 0;

 **case** WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 **return** 0;

 **case** WM\_DESTROY:

 PostQuitMessage(0);

 **return** 0;

 **default**:

 **return** DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

Comme vous pouvez le voir dans l'exemple nous créons le contrôle d'édition à l'arrivée du message WM\_CREATE, message envoyé par Windows quand la création de la fenêtre principale est terminée, mais pas encore visible. A la réception du message WM\_SIZE, nous récupérons les dimensions de la surface client de la fenêtre dans le paramètre lParam joint au message. La largeur étant dans son mot de poids faible et la hauteur dans son mot de poids fort. Nous les extrayons à l'aide des macros LOWORD et HIWORD. Et nous affecterons ces dimensions au contrôle d'édition à l'aide de la fonction **MoveWindow** (son dernier paramètre à TRUE informe Windows qu'il doit redessiner le contrôle après l'appel de la fonction).
Son premier paramètre étant le handle du contrôle d'édition, car c'est lui dont on change les dimensions.
Il est évident que l'on ne va pas passer tous les messages en revue, vous pouvez pour cela consulter l'aide sur L'API Win32. Cet exemple vous donne déjà une idée de la manière dont on intercepte les événements dans Windows. Nous aurons l'occasion de voir et revoir les messages dans les chapitres suivants. Il sont omniprésent dans Windows. Le moindre événement qui survient et c'est un message qui est envoyé. Il est donc important d'être bien documenté, notament sur les divers messages existants. Et ils sont nombreux.
Comme dit au chapitre précédent, les messages non traités doivent l'être par la fonction **DefWindowProc**. C'est une procédure de fenêtre interne qui traite les messages afin de faire le minimum vital pour la fenêtre. Ne serait ce que simplement l'afficher.

**Structure MSG :**

typedef struct tagMSG {

 HWND hwnd; // Handle de la fenêtre de destination.

 UINT message; // Le message proprement dit.

 WPARAM wParam; // Paramètre dépendant du message.

 LPARAM lParam; // Paramètre dépendant du message.

 DWORD time; // Heure ou il a été posté.

 POINT pt; // Coordonnées du curseur au moment ou il a été posté.

} MSG;

**Code complet :**

#include <windows.h>

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

**int** WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, **int** nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = NULL;

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 **if**(!RegisterClass(&wc)) **return** FALSE;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

 **if** (!hwnd) **return** FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 **return** msg.wParam;

}

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 **static** HWND hEdit;

 **switch** (uMsg)

 {

 **case** WM\_CREATE:

 hEdit =CreateWindow("edit", "Texte",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 **return** 0;

 **case** WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 **return** 0;

 **case** WM\_DESTROY:

 PostQuitMessage(0);

 **return** 0;

 **default**:

 **return** DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

Dans le code de cet exemple j'ai mis le pinceau à NULL *(wc.hbrBackground = NULL)* dans la classe de fenêtre de la fenêtre principale, sinon un il y a scintillement désagréable sur l'ascenseur du contrôle d'édition lors des redimensionnements.

J'ai testé les compilations avec C++ Builder et DevC++ (Compilé avec DevC++ le programme ne fait que seulement 4 Ko).

A vos PC.

**3 - Les commandes et notifications :**

**Introduction :**

Dans ce chapitre nous allons aborder un message particulier : WM\_COMMAND. Ce message sert à envoyer des commandes. Une commande est envoyé suite à l'action d'un contrôle par l'utilisateur. C'est-à-dire par exemple le click d'un bouton ou d'une option d'un menu... Nous devons donc l'intercepter pour implémenter l'action que nous voulons faire. Dans l'exemple nous créerons un menu qui va nous permettre d'envoyer des commandes utilisateur aux fenêtres de l'application.

Nous verrons aussi qu'un contrôle peut envoyer des informations à l'application avec ce message quand des événements propre à lui même se produisent. On les appelle des notifications.

Nous aborderons aussi l'envoi de messages par programme.



**Création du menu :**

Pour préparer notre exemple nous allons d'abord créer le menu. Il existe deux solutions pour créer des menus, soit par programme, soit à l'aide de ressources. Les ressources feront l'objet d'un autre chapitre. Nous allons donc utiliser la première solution (à savoir qu'il est plus courant d'utiliser les ressources pour construire un menu).

 #define IDM\_QUIT 1

 HMENU hMenu, hSousMenu;

 hSousMenu = CreateMenu();

 AppendMenu(hSousMenu, MF\_STRING, IDM\_QUIT, "Quitter");

 hMenu = CreateMenu();

 AppendMenu(hMenu,MF\_POPUP,(UINT)hSousMenu,"Fichier");

Les menus sont créés avec la fonction CreateMenu qui renvoie un handle de menu (HMENU). Dans l'exemple nous créons un menu principal et un sous-menu. Nous créons d'abord le sous-menu car il devra être ajouté au menu principal. Nous lui ajoutons une option de menu à l'aide de la fonction AppendMenu. Nous pouvons utiliser cette fonction autant de fois que l'on a d'options à rajouter. Elle reçoit comme paramètres :

**Le handle de menu auquel on veut l'ajouter.**

Une constante indiquant son aspect (MF\_STRING car c'est une chaîne de caractère).

L'identificateur de l'option qui la distinguera parmi les autres. C'est une constante de type UINT (unsigned int) que l'on peut nommer pour plus de clarté dans le code (#define IDM\_QUIT 1).

Le dernier paramètre est un pointeur sur la chaîne de caractères représentant l'option visuellement dans le menu ("Quitter" dans l'exemple).

Nous créons ensuite le menu principal de la même façon. Nous lui ajoutons le sous-menu créé précédemment toujours avec la fonction AppendMenu en lui passant le handle du sous-menu comme paramètre(troisième paramètre de AppendMenu). On lui donnera un aspect de type popup (deuxième paramètre de AppendMenu à MF\_POPUP).

Maintenant, il faut l'ajouter à la fenêtre principale de l'application. Dans l'exemple nous le ferons lors de la création de la fenêtre en passant son handle comme paramètre à la fonction CreateWindow :

hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, hMenu, hinstance, NULL);

**Les commandes :**

Comment savoir que l'utilisateur a actionné l'option du menu ?

Et bien en traitant le message WM\_COMMAND. Comme nous l'avons dit plus haut, ce message est envoyé par Windows quand une commande utilisateur a été actionnée.

Mais comment savoir que cette commande vient d'une option particulière d'un menu ?

Dans le mot de poids faible du paramètre wParam qui est envoyé avec le message WM\_COMMAND : LOWORD(wParam). Il contient l'identificateur que l'on avait attribué à l'option de menu lors de son ajout au menu (IDM\_QUIT dans l'exemple) :

 switch (uMsg)

 {

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 return 0;

Il suffit alors de faire un test sur ce paramètre si l'on est en train de traiter un message WM\_COMMAND et d'agir en consequence.

**Envoyer des messages :**

Dans cet exemple si l'on a affaire à la commande IDM\_QUIT nous demanderons à la fenêtre de se fermer en lui envoyant un message WM\_CLOSE avec la fonction PostMessage. Et oui, nous aussi nous pouvons envoyer des messages !

PostMessage permet de déposer des messages dans la file d'attente.

Son premier paramètre est le handle de la fenêtre de destination.

Son second paramètre est l'identificateur du message (WM\_CLOSE dans l'exemple).

Les deux paramètres suivants sont les valeurs des champs wParam et lParam du message (wParam et lParam sont inutilisés pour le message WM\_CLOSE).

Il existe une autre fonction pour envoyer des messages : SendMessage. Elle reçoit les mêmes paramètres que PostMessage et dans le même ordre. La différence est qu'elle envoie le message directement à la procédure de fenêtre sans passer par la file d'attente. Ensuite, elle attend que le message ai été traité, afin de retourner un résultat. Dans l'exemple nous l'utilisons pour ordonner au contrôle d'édition de changer sa police de caractère :

 HFONT hFont;

 hFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

Nous avons ici récupéré le handle d'une police de caractère prédéfinie de Windows avec la fonction GetStockObject (Cette fonction permet de récupérer des objets initialisés au démarrage de Windows, tel des pinceaux, crayons...).

Nous allons lui envoyer un second message pour mettre ses marges interieures droite et gauche à 5 pixels :

SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

La valeur des marges est envoyée dans un seul paramètre. Il est reconstitué en un DWORD avec la macro MAKELONG. Vous avez du remarquer que le nom de ce message est préfixé des lettre EM au lieu de WM. C'est tout simplement qu'il s'agit d'un message destiné spécifiquement aux contrôles d'édition. EM comme Edit Message et bien sûr WM comme Window Message qui eux s'adresse à tous les types de fenêtres.

**Les notifications :**

Pendant le déroulement du programme les contrôles envoient des messages WM\_COMMAND à leur fenêtre parent. Ceci quand ils subissent certains événements. Prenons comme exemple le contrôle d'édition, à chaque fois que vous modifiez son texte, il envoie un message WM\_COMMAND avec le paramètre EN\_CHANGE, pour nous le notifier. On les appelle les notifications. Elles sont envoyées dans le mot de poids fort du champ wParam : HIWORD(wParam) du message WM\_COMMAND. Je vous recommande de consulter l'aide API Win32 pour chaque contrôle, afin vous documenter sur leurs différentes notifications.

 switch (uMsg)

 {

 case WM\_COMMAND:

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

Dans l'exemple, à la réception de cette notification nous mémoriserons dans une variable ce changement. Variable que nous testerons à la fermeture de la fenêtre principale afin de proposer une boîte de message pour laisser le choix à l'utilisateur de fermer ou non la fenêtre s'il y a eu modification du texte du contrôle d'édition :

 switch (uMsg)

 {

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

Nous interceptons pour cela le message WM\_CLOSE. Nous testons la variable EditNotChg si elle est à TRUE, DestroyWindow est appelé, ce qui a pour conséquence d'envoyer un message WM\_DESTROY et donc de fermer l'application. Si EditNotChg est à FALSE la fonction MessageBox est appelée si elle retourne IDYES, nous appelons de même DestroyWindow, sinon le programme continue.

MessageBox, vous l'avez compris sert à appeler des boîtes de messages. Son premier paramètre est le handle de fenêtre de son parent. Le deuxième paramètre est le texte du message ou de la question. Le troisième paramètre est son titre. Le quatrième est une constante indiquant son type. (MB\_YESNO | MB\_ICONQUESTION) dans l'exemple car elle possède un bouton "Oui", un bouton "Non" et l'icône de question (voir l'aide API Win32 pour les autre possibilités).

**Code complet :**

#include <windows.h>

#define IDM\_QUIT 1

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 HMENU hMenu, hSousMenu;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = NULL;

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hSousMenu = CreateMenu();

 AppendMenu(hSousMenu, MF\_STRING, IDM\_QUIT, "Quitter");

 hMenu = CreateMenu();

 AppendMenu(hMenu,MF\_POPUP,(UINT)hSousMenu,"Fichier");

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, hMenu, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 static BOOL EditNotChg = TRUE;

 switch (uMsg)

 {

 case WM\_CREATE:

 {

 HFONT hFont;

 hEdit = CreateWindow("edit", "Texte",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 hFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

 return 0;

 }

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

Comme vous pouver le constater les message sont omniprésent dans Windows.

J'ai testé les compilations avec C++ Builder et DevC++.

**4 - Les boîtes de dialogue communes. (Ouverture de fichiers.)**

**Introduction :**

Les boîtes de dialogues communes sont celles qui font partie de Windows, comme les boîtes de dialogue d'ouverture ou de sauvegarde de fichier, de choix de polices de caractères, de choix de couleurs... Dans ce premier exemple nous metterons en oeuvre la boîte de dialogue d'ouverture de fichiers. Le principe est semblable pour les autres boîtes de dialogue.

**Préparation de l'exemple :**

Nous continuons sur la base de l'exemple du chapitre précédent. Nous ajoutons une option avec le texte "Ouvrir..." plus un séparateur au sous-menu. L'option "Ouvrir..." sera identifiée par la constante IDM\_OPEN :

#define IDM\_QUIT 1

#define IDM\_OPEN 2

 hSousMenu = CreateMenu();

 AppendMenu(hSousMenu, MF\_STRING, IDM\_OPEN, "Ouvrir...");

 AppendMenu(hSousMenu, MF\_SEPARATOR, 0, NULL);

 AppendMenu(hSousMenu, MF\_STRING, IDM\_QUIT, "Quitter");

Comme vu au chapitre précédent, son action sera interceptée dans la procédure de fenêtre avec un **message WM\_COMMAND :**

 switch (uMsg)

 {

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_OPEN)

 {

 /\* Appel de la boîte de dialogue ici. \*/

**La boîte de dialogue commune :**

Avant d'appeler une boîte de dialogue commune, nous devont remplir une structure de données la concernant. Elle sera passée comme paramètre à la fonction d'appel. Dans le cas de la boîte de dialogue destinée à l'ouverture de fichiers la structure se nomme OPENFILENAME :

 OPENFILENAME ofn;

 CHAR szFile[MAX\_PATH]={0};

 ZeroMemory(&ofn, sizeof(OPENFILENAME));

 ofn.lStructSize = sizeof(OPENFILENAME);

 ofn.hwndOwner = hwnd;

 ofn.lpstrFile = szFile;

 ofn.nMaxFile = MAX\_PATH;

 ofn.lpstrFilter =

 "Fichier source C\0\*.c\0Fichier source CPP\0\*.cpp\0";

 ofn.nFilterIndex = 1;

 ofn.Flags =

 OFN\_PATHMUSTEXIST | OFN\_FILEMUSTEXIST | OFN\_HIDEREADONLY;

Nous déclarons donc en premier lieu une variable du type de cette structure : ofn. Comme nous allons remplir seulement les champs qui nous intéressent, nous metterons la totalité de la structure à zéro. Ceci étant effectué en une seule opération par l'appel de la fonction ZeroMemory.

Ensuite, nous affectons son champ lpstrFile avec l'adresse d'un tableau de caractères que nous avons préalablement défini, pour y récupérer le nom et le chemin du fichier. Attention, il doit contenir une chaine valide avant l'appel de la fonction, car elle l'utilise aussi pour initialiser le champ d'édition : "Nom du fichier" de la boîte de dialogue.

Le champ lStructSize doit recevoir la taille de la structure.

Le champ hwndOwner le handle de la fenêtre parent.

Le champ nMaxFile la taille du tableau de caractère recevant le nom du fichier.

Le champ lpstrFilter avec l'adresse d'un tableau de caractère contenant des filtres de fichiers par leur extention. Les textes des filtres et les filtres étant séparés par des caractères null et le tableau ce terminant par deux caractères null (ce type de tableau de caractères est assez fréquent dans Windows).

Le champ nFilterIndex avec l'index du filtre que l'on désire afficher dans le champ d'édition "Fichier de type" à l'ouverture de la boîte de dialogue.

Le champ Flags avec différentes constantes qui influence sur son aspect ou ses fonctionnalités (voir l'aide API Win32).

Une fois les champs de la structure correctement rempli, nous appelons la fonction GetOpenFileName avec l'adresse de la structure précédement rempli comme paramètre. Si elle nous retourne la valeur TRUE, ce qui indique que l'utilisateur à fermé la boîte de dialogue avec le bouton "Ouvrir", nous pourons récupérer le nom et chemin du fichier sélectionné dans le buffer szFile afin de charger le fichier.

 if (GetOpenFileName(&ofn)==TRUE)

 {

 HANDLE hf;

 DWORD FileSize,nbcharRead ;

 CHAR \*buffer;

 hf = CreateFile(szFile, GENERIC\_READ, 0,NULL,

 OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

 FileSize = GetFileSize(hf, NULL);

 buffer = (PCHAR)LocalAlloc(LMEM\_FIXED, FileSize+1);

 ReadFile(hf, buffer, FileSize, &nbcharRead, NULL) ;

 buffer[FileSize] = 0;

 SendMessage(hEdit, WM\_SETTEXT, 0, (LPARAM)buffer);

 LocalFree(buffer);

 CloseHandle(hf);

 }

Connaissant maintenant le nom et le chemin du fichier, nous allons l'ouvrir avec la fonction de Windows CreateFile qui retourne un handle de fichier. Je vous invite à consuler l'aide API Win32 pour plus de détails sur cette fonction qui offre de nombreuses possibilités. Nous récupérons la taille du fichier avec la fonction GetFileSize. Nous créons un buffer de la taille du fichier + 1 octet avec la fonction de Windows LocalAlloc. Nous remplissons ce buffer avec le contenu du fichier à l'aide de la fonction ReadFile. Comme c'est un fichier texte nous ajoutons le zéro terminal au dernier élément du buffer pour en faire une chaîne de caractères.

Nous la copions dans le champs d'édition multi lignes, en lui envoyant un message WM\_SETTEXT avec l'adresse du buffer comme dernier paramètre de SendMessage. Pour terminer, nous libérons la mémoire avec la fonction LocalFree et fermons le fichier avec la fonction CloseHandle.

**Code complet :**

#include <windows.h>

#define IDM\_QUIT 1

#define IDM\_OPEN 2

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 HMENU hMenu, hSousMenu;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = NULL;

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hSousMenu = CreateMenu();

 AppendMenu(hSousMenu, MF\_STRING, IDM\_OPEN, "Ouvrir...");

 AppendMenu(hSousMenu, MF\_SEPARATOR, 0, NULL);

 AppendMenu(hSousMenu, MF\_STRING, IDM\_QUIT, "Quitter");

 hMenu = CreateMenu();

 AppendMenu(hMenu,MF\_POPUP,(UINT)hSousMenu,"Fichier");

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 640, 300,

 NULL, hMenu, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 static BOOL EditNotChg = TRUE;

 switch (uMsg)

 {

 case WM\_CREATE:

 {

 HFONT hFont;

 hEdit = CreateWindow("edit", "Texte", WS\_CHILD | WS\_VISIBLE |

 ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 hFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

 return 0;

 }

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_OPEN)

 {

 OPENFILENAME ofn;

 CHAR szFile[MAX\_PATH]={0};

 ZeroMemory(&ofn, sizeof(OPENFILENAME));

 ofn.lStructSize = sizeof(OPENFILENAME);

 ofn.hwndOwner = hwnd;

 ofn.lpstrFile = szFile;

 ofn.nMaxFile = MAX\_PATH;

 ofn.lpstrFilter =

 "Fichier source C\0\*.c\0Fichier source CPP\0\*.cpp\0";

 ofn.nFilterIndex = 1;

 ofn.Flags =

 OFN\_PATHMUSTEXIST | OFN\_FILEMUSTEXIST | OFN\_HIDEREADONLY;

 if (GetOpenFileName(&ofn)==TRUE)

 {

 HANDLE hf;

 DWORD FileSize,nbcharRead ;

 CHAR \*buffer;

 hf = CreateFile(szFile, GENERIC\_READ, 0,NULL,

 OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

 FileSize = GetFileSize(hf, NULL);

 buffer = (PCHAR)LocalAlloc(LMEM\_FIXED, FileSize+1);

 ReadFile(hf, buffer, FileSize, &nbcharRead, NULL) ;

 buffer[FileSize] = 0;

 SendMessage(hEdit, WM\_SETTEXT, 0, (LPARAM)buffer);

 LocalFree(buffer);

 CloseHandle(hf);

 }

 }

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

**En annexe voici 3 autres boîtes de dialogue commun :**

**- La boîte de dialogue de sauvegarde.**

**Save dialog :**

En annexe au chapitre "Boîtes de dialogue communes." Voici le code de l'exemple avec la sauvegarde du contenu du contrôle d'édition en plus. Le principe est identique que l'ouverture de fichier. C'est la même structure qui est utilisé : OPENFILENAME. La fonction d'appel ce nomme GetSaveFileName et s'utilise de la même façon que GetOpenFileName. L'écriture du fichier est réalisée avec la fonction WriteFile.

**Code complet :**

#include <windows.h>

#define IDM\_QUIT 1

#define IDM\_OPEN 2

#define IDM\_SAVE 3

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 HMENU hMenu, hSousMenu;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = NULL;

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hSousMenu = CreateMenu();

 AppendMenu(hSousMenu, MF\_STRING, IDM\_OPEN, "Ouvrir...");

 AppendMenu(hSousMenu, MF\_STRING, IDM\_SAVE, "Enregistrer...");

 AppendMenu(hSousMenu, MF\_SEPARATOR, 0, NULL);

 AppendMenu(hSousMenu, MF\_STRING, IDM\_QUIT, "Quitter");

 hMenu = CreateMenu();

 AppendMenu(hMenu,MF\_POPUP,(UINT)hSousMenu,"Fichier");

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 640, 300,

 NULL, hMenu, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 static BOOL EditNotChg = TRUE;

 switch (uMsg)

 {

 case WM\_CREATE:

 {

 HFONT hFont;

 hEdit = CreateWindow("edit", "Texte", WS\_CHILD | WS\_VISIBLE |

 ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 hFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

 return 0;

 }

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_OPEN)

 {

 OPENFILENAME ofn;

 CHAR szFile[MAX\_PATH]={0};

 ZeroMemory(&ofn, sizeof(OPENFILENAME));

#ifdef OPENFILENAME\_SIZE\_VERSION\_400

 ofn.lStructSize = OPENFILENAME\_SIZE\_VERSION\_400;

#else

 ofn.lStructSize = sizeof(OPENFILENAME);

#endif

 ofn.hwndOwner = hwnd;

 ofn.lpstrFile = szFile;

 ofn.nMaxFile = MAX\_PATH;

 ofn.lpstrFilter =

 "Fichier source C\0\*.c\0Fichier source CPP\0\*.cpp\0";

 ofn.nFilterIndex = 1;

 ofn.Flags =

 OFN\_PATHMUSTEXIST | OFN\_FILEMUSTEXIST | OFN\_HIDEREADONLY;

 if (GetOpenFileName(&ofn)==TRUE)

 {

 HANDLE hf;

 DWORD FileSize,nbcharRead ;

 char \*buffer;

 hf = CreateFile(szFile, GENERIC\_READ, 0,NULL,

 OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

 FileSize = GetFileSize(hf, NULL);

 buffer = (char\*)LocalAlloc(LMEM\_FIXED, FileSize+1);

 ReadFile(hf, buffer, FileSize, &nbcharRead, NULL) ;

 buffer[FileSize] = 0;

 SendMessage(hEdit, WM\_SETTEXT, 0, (LPARAM)buffer);

 LocalFree(buffer);

 CloseHandle(hf);

 }

 } /\*if IDM\_OPEN\*/

 if(LOWORD(wParam) == IDM\_SAVE)

 {

 OPENFILENAME ofn;

 CHAR szFile[260]={0};

 ZeroMemory(&ofn, sizeof(OPENFILENAME));

#ifdef OPENFILENAME\_SIZE\_VERSION\_400

 ofn.lStructSize = OPENFILENAME\_SIZE\_VERSION\_400;

#else

 ofn.lStructSize = sizeof(OPENFILENAME);

#endif

 ofn.hwndOwner = hwnd;

 ofn.lpstrFile = szFile;

 ofn.nMaxFile = sizeof(szFile);

 ofn.lpstrFilter =

 "Fichier source C\0\*.c\0Fichier source CPP\0\*.cpp\0";

 ofn.nFilterIndex = 1;

 ofn.Flags =

 OFN\_PATHMUSTEXIST | OFN\_HIDEREADONLY | OFN\_OVERWRITEPROMPT;

 if (GetSaveFileName(&ofn)==TRUE)

 {

 HANDLE hf;

 DWORD FileSize,nbcharRead ;

 char \*buffer;

 FileSize = GetWindowTextLength(hEdit);

 buffer = (char\*)LocalAlloc(LMEM\_FIXED, FileSize+1);

 GetWindowText(hEdit, buffer, FileSize+1);

 hf = CreateFile(szFile, GENERIC\_WRITE, 0,NULL,

 CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

 WriteFile(hf, buffer, FileSize, &nbcharRead, NULL) ;

 CloseHandle(hf);

 LocalFree(buffer);

 }

 } /\*if IDM\_SAVE\*/

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

**- La boîte de dialogue de choix de police de caractères.**

**Font dialog :**

En annexe au chapitre Boîtes de dialogue communes. Voici le code de l'exemple avec la possibilité de changer la police de caractères du contrôle d'édition. La fonction d'appel de la boîte de dialogue se nomme ChooseFont. Elle utilise une structure CHOOSEFONT pour s'initialiser. Elle a entre autre un champ recevant un pointeur sur une structure LOGFONT. Il permet d'initialiser la boîte de dialogue avec la police en cours et de recevoir celle qui a été choisie dans la boîte de dialogue.

 static HFONT hFont;

 static LOGFONT lf;

 /\*... \*/

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_FONT)

 {

 CHOOSEFONT cf;

 ZeroMemory(&cf, sizeof(CHOOSEFONT));

 cf.lStructSize = sizeof (CHOOSEFONT);

 cf.hwndOwner = hwnd;

 cf.lpLogFont = &lf;

 cf.Flags = CF\_SCREENFONTS | CF\_INITTOLOGFONTSTRUCT;

 if (ChooseFont(&cf))

 {

 DeleteObject(hFont);

 hFont = CreateFontIndirect(&lf);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 }

 }

Pour plus de précisions sur les polices de caractères vous pouvez consulter ce chapitre : Dessiner avec Windows.

**Code complet :**

#include <windows.h>

#define IDM\_QUIT 1

#define IDM\_OPEN 2

#define IDM\_FONT 3

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 HMENU hMenu, hSousMenu;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = NULL;

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hSousMenu = CreateMenu();

 AppendMenu(hSousMenu, MF\_STRING, IDM\_OPEN, "Ouvrir...");

 AppendMenu(hSousMenu, MF\_STRING, IDM\_FONT, "Font...");

 AppendMenu(hSousMenu, MF\_SEPARATOR, 0, NULL);

 AppendMenu(hSousMenu, MF\_STRING, IDM\_QUIT, "Quitter");

 hMenu = CreateMenu();

 AppendMenu(hMenu,MF\_POPUP,(UINT)hSousMenu,"Fichier");

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 640, 300,

 NULL, hMenu, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 static HFONT hFont;

 static LOGFONT lf;

 static BOOL EditNotChg = TRUE;

 switch (uMsg)

 {

 case WM\_CREATE:

 {

 hEdit = CreateWindow("edit", "Texte", WS\_CHILD | WS\_VISIBLE |

 ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 ZeroMemory(&lf, sizeof(LOGFONT));

 lstrcpy(lf.lfFaceName,"Courier");

 lf.lfHeight = 10;

 hFont = CreateFontIndirect(&lf);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

 return 0;

 }

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_OPEN)

 {

 OPENFILENAME ofn;

 CHAR szFile[MAX\_PATH]={0};

 ZeroMemory(&ofn, sizeof(OPENFILENAME));

#ifdef OPENFILENAME\_SIZE\_VERSION\_400

 ofn.lStructSize = OPENFILENAME\_SIZE\_VERSION\_400;

#else

 ofn.lStructSize = sizeof(OPENFILENAME);

#endif

 ofn.hwndOwner = hwnd;

 ofn.lpstrFile = szFile;

 ofn.nMaxFile = MAX\_PATH;

 ofn.lpstrFilter =

 "Fichier source C\0\*.c\0Fichier source CPP\0\*.cpp\0";

 ofn.nFilterIndex = 1;

 ofn.Flags =

 OFN\_PATHMUSTEXIST | OFN\_FILEMUSTEXIST | OFN\_HIDEREADONLY;

 if (GetOpenFileName(&ofn))

 {

 HANDLE hf;

 DWORD FileSize,nbcharRead ;

 char \*buffer;

 hf = CreateFile(szFile, GENERIC\_READ, 0,NULL,

 OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

 FileSize = GetFileSize(hf, NULL);

 buffer = (char\*)LocalAlloc(LMEM\_FIXED, FileSize+1);

 ReadFile(hf, buffer, FileSize, &nbcharRead, NULL) ;

 buffer[FileSize] = 0;

 SendMessage(hEdit, WM\_SETTEXT, 0, (LPARAM)buffer);

 LocalFree(buffer);

 CloseHandle(hf);

 }

 } /\*if IDM\_OPEN\*/

 if(LOWORD(wParam) == IDM\_FONT)

 {

 CHOOSEFONT cf;

 ZeroMemory(&cf, sizeof(CHOOSEFONT));

 cf.lStructSize = sizeof (CHOOSEFONT);

 cf.hwndOwner = hwnd;

 cf.lpLogFont = &lf;

 cf.Flags = CF\_SCREENFONTS | CF\_INITTOLOGFONTSTRUCT;

 if (ChooseFont(&cf))

 {

 DeleteObject(hFont);

 hFont = CreateFontIndirect(&lf);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 }

 } /\*if IDM\_FONT\*/

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 DeleteObject(hFont);

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

J'ai testé les compilations avec C++ Builder et DevC++.

**- La boîte de dialogue de choix de couleur.**

**Introduction :**

En annexe au chapitre : "Les boîtes de dialogue communes", voici une mise en application de la boîte de dialogue de choix de couleur.

**Préparation de l'exemple :**

Sur la base du même exemple. Nous ajoutons une option avec le texte "Couleur du texte..." au sous-menu. L'option "Couleur du texte..." sera identifiée par la constante IDM\_TEXTCOLOR :

#define IDM\_QUIT 1

#define IDM\_OPEN 2

#define IDM\_TEXTCOLOR 3

 /\*... \*/

 hSousMenu = CreateMenu();

 AppendMenu(hSousMenu, MF\_STRING, IDM\_OPEN, "Ouvrir...");

 AppendMenu(hSousMenu, MF\_STRING, IDM\_TEXTCOLOR, "Couleur du texte...");

 AppendMenu(hSousMenu, MF\_SEPARATOR, 0, NULL);

 AppendMenu(hSousMenu, MF\_STRING, IDM\_QUIT, "Quitter");

Nous nous en servirons pour changer la couleur du texte du contrôle d'édition multi lignes.

Pour changer cette couleur nous emploierons la fonction SetTextColor. Elle reçoit comme paramètre un handle de contexte de périphérique et une couleur.

Mais, comment obtenir un contexte de périphérique pour le contrôle d'édition ?

Windows avant d'effectuer le dessin des contrôles envoie un message à sa fenêtre parent : WM\_CTLCOLOREDIT pour le contrôle d'édition et y joint son handle de contexte de périphérique (HDC) dans le paramètre wParam et son handle de fenêtre dans le paramètre lParam.

 case WM\_CTLCOLOREDIT:

 if((HWND)lParam == hEdit)

 {

 SetTextColor((HDC)wParam, TextColor);

 SetBkColor((HDC)wParam, BkEditcolor);

 return EditBrush;

 }

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

Quand on traite ce message on doit retourner un pinceau. Dans l'exemple si l'on est en train de traiter ce message pour notre contrôle d'édition, nous changeons la couleur de son texte, de son fond et retournons un pinceau de la même couleur que le fond. Ce pinceau ayant été initialisé à la création de la fenêtre principale :

 static HBRUSH EditBrush;

 static COLORREF BkEditcolor;

 static COLORREF TextColor = 0;

 /\*... \*/

 case WM\_CREATE:

 {

 BkEditcolor = GetSysColor(COLOR\_WINDOW);

 EditBrush = CreateSolidBrush(BkEditcolor);

Si nous ne traitons pas le message WM\_CTLCOLOREDIT, c'est DefWindowProc qui met les valeurs par défaut pour le contrôle.

**La boîte de dialogue de choix couleur :**

Comme toutes les boîtes de dialogue commune on utilise une structure (CHOOSECOLOR) pour l'initialiser :

 static COLORREF TextColor = 0;

 /\*... \*/

 CHOOSECOLOR cc;

 static COLORREF acrCustClr[16];

 ZeroMemory(&cc, sizeof(CHOOSECOLOR));

 cc.lStructSize = sizeof(CHOOSECOLOR);

 cc.hwndOwner = hwnd;

 cc.lpCustColors = NULL;

 cc.lpCustColors = (LPDWORD) acrCustClr;

 cc.rgbResult = TextColor;

 cc.Flags = CC\_FULLOPEN | CC\_RGBINIT;

Puis on appelle la fonction ChooseColor pour ouvrir la boîte de dialogue :

 if (ChooseColor(&cc)==TRUE)

 {

 TextColor = cc.rgbResult;

 PostMessage(hEdit,WM\_SETREDRAW,TRUE,0);

 }

Elle retourne la couleur choisie dans le champ rgbResult de la variable de type CHOOSECOLOR qu'elle a reçu comme pramètre. Nous nous en servons pour affecter la variable TextColor. La même que l'on avait utilisée pour changer la couleur du controle précédement.

**Code complet :**

#include <windows.h>

#define IDM\_QUIT 1

#define IDM\_OPEN 2

#define IDM\_TEXTCOLOR 3

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 HMENU hMenu, hSousMenu;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = NULL;

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hSousMenu = CreateMenu();

 AppendMenu(hSousMenu, MF\_STRING, IDM\_OPEN, "Ouvrir...");

 AppendMenu(hSousMenu, MF\_STRING, IDM\_TEXTCOLOR, "Couleur du texte...");

 AppendMenu(hSousMenu, MF\_SEPARATOR, 0, NULL);

 AppendMenu(hSousMenu, MF\_STRING, IDM\_QUIT, "Quitter");

 hMenu = CreateMenu();

 AppendMenu(hMenu,MF\_POPUP,(UINT)hSousMenu,"Fichier");

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 640, 300,

 NULL, hMenu, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 static HBRUSH EditBrush;

 static COLORREF BkEditcolor;

 static BOOL EditNotChg = TRUE;

 static COLORREF TextColor = 0;

 switch (uMsg)

 {

 case WM\_CREATE:

 {

 HFONT hFont;

 hEdit = CreateWindow("edit", "Texte", WS\_CHILD | WS\_VISIBLE |

 ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 BkEditcolor = GetSysColor(COLOR\_WINDOW);

 EditBrush = CreateSolidBrush(BkEditcolor);

 hFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

 return 0;

 }

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_OPEN)

 {

 OPENFILENAME ofn;

 CHAR szFile[256]={0};

 ZeroMemory(&ofn, sizeof(OPENFILENAME));

 ofn.lStructSize = sizeof(OPENFILENAME);

 ofn.hwndOwner = hwnd;

 ofn.lpstrFile = szFile;

 ofn.nMaxFile = sizeof(szFile);

 ofn.lpstrFilter =

 "Fichier source C\0\*.c\0Fichier source CPP\0\*.cpp\0";

 ofn.nFilterIndex = 1;

 ofn.Flags =

 OFN\_PATHMUSTEXIST | OFN\_FILEMUSTEXIST | OFN\_HIDEREADONLY;

 if (GetOpenFileName(&ofn)==TRUE)

 {

 HANDLE hf;

 DWORD FileSize,nbcharRead ;

 char \*buffer;

 hf = CreateFile(szFile, GENERIC\_READ, 0,NULL,

 OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

 FileSize = GetFileSize(hf, NULL);

 buffer = (char\*)LocalAlloc(LMEM\_FIXED, FileSize+1);

 ReadFile(hf, buffer, FileSize, &nbcharRead, NULL) ;

 buffer[FileSize] = 0;

 SendMessage(hEdit, WM\_SETTEXT, 0, (LPARAM)buffer);

 LocalFree(buffer);

 CloseHandle(hf);

 }

 } /\*if IDM\_OPEN\*/

 if(LOWORD(wParam) == IDM\_TEXTCOLOR)

 {

 CHOOSECOLOR cc;

 static COLORREF acrCustClr[16];

 ZeroMemory(&cc, sizeof(CHOOSECOLOR));

 cc.lStructSize = sizeof(CHOOSECOLOR);

 cc.hwndOwner = hwnd;

 cc.lpCustColors = NULL;

 cc.lpCustColors = (LPDWORD) acrCustClr;

 cc.rgbResult = TextColor;

 cc.Flags = CC\_FULLOPEN | CC\_RGBINIT;

 if (ChooseColor(&cc)==TRUE)

 {

 TextColor = cc.rgbResult;

 PostMessage(hEdit,WM\_SETREDRAW,TRUE,0);

 }

 } /\*if IDM\_TEXTCOLOR\*/

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

 case WM\_CTLCOLOREDIT:

 if((HWND)lParam == hEdit)

 {

 SetTextColor((HDC)wParam, TextColor);

 SetBkColor((HDC)wParam, BkEditcolor);

 return EditBrush;

 }

 break;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 DeleteObject(EditBrush);

 PostQuitMessage(0);

 return 0;

 }

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

}

J'ai testé les compilations avec C++ Builder et DevC++.

J'ai testé les compilations avec C++ Builder 4 (Win 95), C++ Builder 6 (Win XP), et DevC++ (Win XP).

**Remarques :**

Compilé avec BCB6, la boîte de dialogue ne s'ouvrait pas si le programme était exécuté sous une plateforme Win95. Une erreur CDERR\_STRUCTSIZE était retourné. Aprés une recherche sur le Web, j'ai trouvé que les compilations faite pour Win 2000 pouvait causer ce problème si les fichier header de Windows était recent. Je n'ai pas eu ce problème avec DevC++ avec une compilation sous Win XP.

La solution pour parrer à ce problème est d'intitaliser le champs lStructSize de la structure OPENFILENAME avec la constante : OPENFILENAME\_SIZE\_VERSION\_400.

 ofn.lStructSize = OPENFILENAME\_SIZE\_VERSION\_400;

Voir une compilation conditionnelle :

#ifdef OPENFILENAME\_SIZE\_VERSION\_400

 ofn.lStructSize = OPENFILENAME\_SIZE\_VERSION\_400;

#else

 ofn.lStructSize = sizeof(OPENFILENAME);

#endif

A tester selon vos configuration.

**5 - Les contextes de périphériques.**

**Introduction :**

Nous allons voir dans ce nouveau chapitre "les sorties écrans", ou plutôt l'écriture sur les fenêtres. Sous Windows on parlera plutôt de dessin.

Windows a les facultés de pouvoir dessiner sur n'importe quel périphériques de sortie (Ecran, imprimante ...) sans se préoccuper de comment il fonctionne. Ce sont les drivers qui font le travail.

Pour dessiner sous Windows on doit obtenir du périphérique sur lequel l'on veut dessiner un contexte de périphérique (DC). Pour simplifier nous allons nous représenter le contexte de périphérique comme une surface de dessin que nous aura fourni la fenêtre sur laquelle l'on veut dessiner.



**Dessin :**

Nous allons reprendre l'exemple du chapitre 1. Nous y interceptons le message WM\_KEYDOWN qui est envoyé à la procédure de fenêtre quand on appuie sur une touche et qu'elle possède le focus clavier. Pour une fenêtre (ou contrôle) avoir le focus clavier signifie qu'elle est la fenêtre active et que par conséquent, c'est elle qui reçoit les entrées clavier.

 switch (uMsg)

 {

 case WM\_KEYDOWN :

 {

 int y;

 char st[] = "Bienvenue sur Developpez.com" ;

 HDC hdc = GetDC(hwnd);

 SetBkMode(hdc, TRANSPARENT);

 for(y=10; y <= 200; y += 20)

 TextOut(hdc, 10, y, st , lstrlen(st));

 ReleaseDC(hwnd, hdc);

 return 0;

 }

A la réception de ce message, nous écrirons 10 lignes avec la chaîne de caractères : "Bienvenue sur Developpez.com" décalées de 20 pixels verticalement les unes par rapport aux autres. Première opération, obtenir le handle d'un contexte de périphérique (HDC) pour la fenêtre en question à l'aide de la fonction GetDC qui reçoit comme paramètre de handle de la fenêtre sur laquelle l'on veux écrire ou dessiner. Quand on écrit ou dessine sous Windows, le fond est aussi peint par la même occasion. Comme il ne nous importe pas pour l'instant nous le mettrons transparent à l'aide de la fonction SetBkMode qui recevra donc comme paramètres le handle du contexte de périphérique et une constante de mode : TRANSPARENT. Nous pouvons alors écrire notre chaîne de caractères à l'aide de la fonction TextOut qui reçoit le handle du contexte de périphérique comme premier paramètre. Les deux suivants étant les coordonnées d'affichage (x,y). Le suivant étant un pointeur sur la chaîne de caractères à dessiner. Le dernier étant la longueur de la chaîne de caractères retournée par la fonction lstrlen (fonction de l'API Windows).

Nous devons libérer le contexte de périphérique après son utilisation. C'est le rôle de la fonction ReleaseDC.

Mais voilà, un problème se pose, à chaque fois que la fenêtre est cachée par une autre, le texte qui s'y trouve, disparaît. Ce qui est fort ennuyeux. Dans le cas présent vous pouvez bien le faire réapparaître en appuyant une autre fois sur une touche, mais ce n'est pas très pratique. Mais alors comment faire ? Et bien c'est encore un message qui intervient WM\_PAINT :

**Le message WM\_PAINT :**

A chaque fois qu'une fenêtre ou une partie de fenêtre à besoin de se redessiner Windows lui envoie un message WM\_PAINT. Il nous suffira donc d'intercepter ce message dans sa procédure de fenêtre et d'appeler les fonctions de dessin suite à ce message:

 case WM\_PAINT :

 {

 int y;

 char st[] = "Bienvenue sur Developpez.com" ;

 HDC hdc = GetDC(hwnd);

 SetBkMode(hdc, TRANSPARENT);

 for(y=10; y <= 200; y += 20)

 TextOut(hdc, 10, y, st , lstrlen(st));

 ReleaseDC(hwnd, hdc);

 return 0;

 }

Miracle ! Le texte ne disparaît plus.

Il est redessiné à chaque fois que cela est nécessaire.

Il subsiste un scintillement désagréable, surtout lors des redimensionnements. Ceci est du au fait que la zone client de la fenêtre est entièrement redessinée à chaque message WM\_PAINT qu'elle reçoit. Une solution existe pour éliminer ce phénomène. C'est de redessiner seulement la partie de la fenêtre qui en à besoin. On l'appelle cette partie la zone invalide.

Pour cela, il faut obtenir le handle du contexte de périphérique avec la fonction BeginPaint au lieu de GetDC. Cette fonction nécessite un pointeur sur une variable préalablement défini de type (struct PAINTSTRUCT) comme paramètre afin d'y mémoriser des informations.

 case WM\_PAINT :

 {

 int y;

 char st[] = "Bienvenue sur Developpez.com" ;

 PAINTSTRUCT ps;

 HDC hdc = BeginPaint(hwnd, &ps);

 SetBkMode(hdc, TRANSPARENT);

 for(y=10; y <= 200; y += 20)

 TextOut(hdc, 10, y, st , lstrlen(st));

 EndPaint(hwnd, &ps);

 return 0;

 }

Tout handle de contexte de périphérique obtenu avec BeginPaint doit être libéré avec EndPaint.

Remarque : Les fonctions BeginPaint et EndPaint ne peuvent être utilisées qu'exclusivement à la réception d'un message WM\_PAINT.

**Code complet :**

#include <windows.h>

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 wc.style = 0;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = (HBRUSH)(1 + COLOR\_BTNFACE);

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

 UpdateWindow(hwnd);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 switch (uMsg)

 {

 /\* mis en commentaire et remplacé par WM\_PAINT

 case WM\_KEYDOWN :

 {

 int y;

 char st[] = "Bienvenue sur Developpez.com" ;

 HDC hdc = GetDC(hwnd);

 SetBkMode(hdc, TRANSPARENT);

 for(y=10; y <= 200; y += 20)

 TextOut(hdc, 10, y, st , lstrlen(st));

 ReleaseDC(hwnd, hdc);

 return 0;

 } \*/

 /\* mis en commentaire et remplacé par WM\_PAINT suivant

 case WM\_PAINT :

 {

 int y;

 char st[] = "Bienvenue sur Developpez.com" ;

 HDC hdc = GetDC(hwnd);

 SetBkMode(hdc, TRANSPARENT);

 for(y=10; y <= 200; y += 20)

 TextOut(hdc, 10, y, st , lstrlen(st));

 ReleaseDC(hwnd, hdc);

 return 0;

 } \*/

 case WM\_PAINT :

 {

 int y;

 char st[] = "Bienvenue sur Developpez.com" ;

 PAINTSTRUCT ps;

 HDC hdc = BeginPaint(hwnd, &ps);

 SetBkMode(hdc, TRANSPARENT);

 for(y=10; y <= 200; y += 20)

 TextOut(hdc, 10, y, st , lstrlen(st));

 EndPaint(hwnd, &ps);

 return 0;

 }

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

**6 - Dessiner.**

**Introduction :**

Après avoir vu les contextes de périphériques, nous allons aborder les principales fonctions de dessin.



Comme vu au chapitre 5 nous dessinerons suite à la réception du message WM\_PAINT dans un contexte de périphérique obtenu avec BeginPaint.

**Dessin :**

Nous allons continuer sur la base de l'exemple du chapitre précédent. Nous y écrirons tout d'abord la même chaîne de caractère avec TextOut, mais une seule fois. Ensuite, nous tracerons un rectangle au dessous de la chaîne de caractères :

 Rectangle(hdc, 20, 60, 120, 100);

Cette fonction reçoit comme paramètres le handle de contexte de périphérique et les coordonnées des points haut gauche et bas droit du rectangle. Windows pour dessiner ce rectangle utilise un crayon et un pinceau. Quand vous obtenez un handle de contexte de périphérique, il possède déjà un crayon et un pinceau. Mais vous pouvez aussi dessiner avec des crayons et des pinceaux que vous avez créés.

 HBRUSH hbRed, hbrOld;

 HPEN hp2px, hpOld;

 /\*... ...\*/

 hp2px = CreatePen(PS\_SOLID, 2, 0x00FF0000);

 hpOld = SelectObject(hdc,hp2px);

 hbRed = CreateSolidBrush(0x000000FF);

 hbrOld = SelectObject(hdc,hbRed);

 Rectangle(hdc, 160, 60, 260, 100);

 SelectObject(hdc,hbrOld);

 DeleteObject(hbRed);

 SelectObject(hdc,hpOld);

 DeleteObject(hp2px);

La création d'un crayon ce fait avec la fonction CreatePen qui reçoit comme paramètres une constante indiquant le type de trait (continu dans l'exemple), ensuite la largeur du trait (2 pixels) et enfin sa couleur (bleu). Après son appel, elle retourne un handle de crayon.

Le pinceau rouge est créé avec la fonction CreateSolidBrush qui reçoit comme paramètre une couleur. Mais pour pouvoir les utiliser avec un contexte de périphérique, il faut les sélectionner. C'est le rôle de la fonction SelectObject, qui reçoit le handle du contexte de périphérique sur lequel on veut utiliser l'objet (pinceau, crayon ...) et le handle de l'objet comme second paramètre. Le second rectangle est donc dessiné avec un contour tracé avec le crayon bleu d'une largeur de deux pixels et son intérieur peint avec le pinceau rouge.

Quand vous en avez terminé avec l'utilisation des objets que vous avez créé, il faut les détruire avec la fonction DeleteObject. De même avant de libérer le contexte de périphérique il faut toujours sélectionner ses objets d'origine avec SelectObject. Le handle de ces objets étant retourné par la fonction SelectObject.

Pour la forme nous dessinerons ensuite un cercle à l'aide le la fonction Ellipse et d'un pinceau à hachures diagonales croisés, puis un trait bleu à l'aide des fonctions MoveToEx qui positionne le point de départ et LineTo qui trace jusqu'au point d'arrivée quelle reçoit comme paramètre et qui devient le nouveau point de départ.

Nous n'avons vu que quelques fonctions de dessin, veuillez consulter l'aide API Win32 pour plus de détail.

 hbGreen = CreateHatchBrush(HS\_DIAGCROSS, 0x0000BB00);

 SelectObject(hdc,hbGreen);

 DeleteObject(hbRed);

 Ellipse(hdc, 300, 20, 400, 120);

 MoveToEx(hdc, 20, 140, NULL);

 LineTo(hdc, 400, 140);

Je vous invite de même à consulter l'aide Win32 API pour voir toutes les possibilités que fournissent les crayons et les pinceaux.

**Les polices de caractères :**

Nous avons dessiné une chaîne de caractère sur la fenêtre, mais comme vous le savez Windows peut utiliser d'autres polices de caractères installées sur le système tel les polices "True Type".

Pour les utiliser le principe est semblable aux crayons ou pinceaux. Il faut créer l'objet police avec CreateFont ou CreateFontIndirect qui retournent un handle de police. Dans l'exemple nous utiliserons CreateFontIndirect car elle est plus simple à utiliser. CreateFontIndirect utilise une structure LOGFONT pour initialiser la police. Nous remplirons seulement deux de ses champs dans l'exemple pour qu'il reste simple. Les autres serons initialisé à zéro à l'aide de la fonction ZeroMemory :

 HFONT NewFont, OldFont;

 LOGFONT lf;

 /\*... ...\*/

 ZeroMemory(&lf, sizeof(LOGFONT));

 lstrcpy(lf.lfFaceName,"Times New Roman");

 lf.lfHeight = 36;

 NewFont = CreateFontIndirect(&lf);

 OldFont = SelectObject(hdc,NewFont);

 SetTextColor(hdc, 0x000000FF);

 TextOut(hdc, 10, 180, st, lstrlen(st));

 SelectObject(hdc,OldFont);

 DeleteObject(NewFont);

Première opération, remplir les champs de la variable lf de type LOGFONT. Elle possède un champ lfFaceName : tableau de caractères où nous copions le nom de la police que nous désirons utiliser avec la fonction lstrcpy. Ce tableau a une taille de 32 caractères (y compris le zéro terminal). Nous mettons le champ lfHeight correspondant à la hauteur des caractères à 36. Nous positionnerons le champ lfItalic à true et le champ lfWeight à FW\_BOLD pour écrire la chaîne de caractères en gras italique.

Tout comme pour les crayons ou pinceaux, il faut sélectionner les objets Fonts pour les utiliser et les détruire après utilisation, sans oublier de sélectionner la police d'origine avant de libérer le contexte de périphérique.

Pour plus de précisions sur les polices de caractères, veuillez consulter l'aide API Win32.

**Code complet :**

#include <windows.h>

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 wc.style = 0;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = (HBRUSH)(1 + COLOR\_BTNFACE);

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Dessiner", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 460, 300,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

 UpdateWindow(hwnd);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_PAINT :

 {

 HBRUSH hbRed, hbGreen, hbrOld;

 HPEN hp2px, hpOld;

 HFONT NewFont, OldFont;

 LOGFONT lf;

 char st[] = "Bienvenue sur Developpez.com" ;

 PAINTSTRUCT ps;

 HDC hdc = BeginPaint(hwnd, &ps);

 SetBkMode(hdc, TRANSPARENT);

 TextOut(hdc, 10, 20, st, lstrlen(st));

 Rectangle(hdc, 20, 60, 120, 100);

 hp2px = CreatePen(PS\_SOLID, 2, 0x00FF0000);

 hpOld = SelectObject(hdc,hp2px);

 hbRed = CreateSolidBrush(0x000000FF);

 hbrOld = SelectObject(hdc,hbRed);

 Rectangle(hdc, 160, 60, 260, 100);

 hbGreen = CreateHatchBrush(HS\_DIAGCROSS, 0x0000BB00);

 SelectObject(hdc,hbGreen);

 DeleteObject(hbRed);

 Ellipse(hdc, 300, 20, 400, 120);

 MoveToEx(hdc, 20, 140, NULL);

 LineTo(hdc, 400, 140);

 SelectObject(hdc,hbrOld);

 DeleteObject(hbGreen);

 SelectObject(hdc,hpOld);

 DeleteObject(hp2px);

 ZeroMemory(&lf, sizeof(LOGFONT));

 lstrcpy(lf.lfFaceName,"Times New Roman");

 lf.lfHeight = 36;

 lf.lfItalic = TRUE;

 lf.lfWeight = FW\_BOLD;

 NewFont = CreateFontIndirect(&lf);

 OldFont = SelectObject(hdc,NewFont);

 SetTextColor(hdc, 0x000000FF);

 TextOut(hdc, 10, 180, st, lstrlen(st));

 SelectObject(hdc,OldFont);

 DeleteObject(NewFont);

 EndPaint(hwnd, &ps);

 return 0;

 }

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

**7 - Les ressources.**

**Introduction :**

Dans ce chapitre nous allons aborder les ressources.

Tout d'abord qu'appelle t'on les ressources dans un programme Windows ?

Ce sont des données de toutes natures qui sont incorporées à l'exécutable. Cela peut être des images, des icônes, des chaînes de caractères, des blocs de données de toutes sortes... Mais aussi des boîtes de dialogue et des menus qui seront construits à partir de script.



**Incorporation des ressources à l'exécutable :**

L'incorporation des ressources à l'exécutable est réalisée par le linker. Nous devons lui fournir un fichier contenant les ressources. Ces fichiers ont l'extension ".res". Ils sont compilés par un compilateur de ressources, auquel on fourni un fichier de script de ressources. Les fichiers de script de ressources sont des fichiers texte ayant l'extension ".rc". Il existe aussi des éditeurs de ressources permettant de créer visuellement les fichiers ressources.

Ce tutoriel n'étant pas déstiné à des outils particuliés, nous utiliserons uniquement les scripts de ressources dans les exemples qui vont suivre.

Si vous utilisez un EDI, il suffit d'ajouter les scripts de ressources au projet (pour Dev-C++ il faut les ajouter par le menu "Projects" puis "Projects Options" puis par le bouton "Load Resource", pour C++ Builder il faut tout simplement ajouter les fichiers au projet). La compilation des ressources sera dans ce cas transparente.

Nous allons en voir deux dans ce chapitre : les icônes et les menus.

**Le script de ressources :**

Nous allons rependre l'exemple du chapitre 3 - Les commandes et notifications. Nous allons d'abord ajouter des icônes dans les ressources de l'exécutable. Créons tout d'abord un fichier script de ressources nommé "resource.rc" avec le texte suivant :

1 ICON icone.ico

2 ICON autre.ico

Chaque ligne est composée de trois éléments, le premier est un identificateur, le second le type de ressource (ICON pour une icône dans l'exemple) le dernier est le contenu, dans le cas de l'exemple ce sera le contenu d'un fichier. Ils doivent bien sur exister. Les ayant mit dans le même dossier que les fichiers sources du programme, je n'ai pas indiqué le chemin pour les localiser. S'ils se trouvent ailleurs il faudra bien sur mettre le chemin (j'ai du tout de même le mettre pour la compilation avec Dev-C++ malgré qu'ils se trouvaient dans le même dossier).

Certain compilateur de resources demande d'inclure les fichiers entêtes de Windows qui contiennent la définition des identificateurs utilisés dans le fichier script de ressources.

Il peux y avoir des différences notables entre les différents compilateurs de ressources. Je vous invite à consulter l'aide de celui que vous utilisez si vous rencontrez des problèmes de compilation.

Je le répète une dernière fois, ce fichier devra être ajouté au projet si vous possédez un EDI ou compilé à l'aide d'un compilateur de ressources et lié à l'exécutable si vous n'avez pas d'EDI.

A ce stade, si vous compilez et que vous visualisez le fichier exécutable dans l'explorateur de fichiers, il devrait avoir la première icône de la liste (ce n'est pas forcement vrai si vous utilisez un EDI, ils peuvent ajouter automatiquement une icône aux ressources de l'exécutable).

Nous allons maintenant mettre la deuxième icône comme icône de la fenêtre (celle qui ce trouve à gauche sur la barre de titre) :

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 WNDCLASS wc;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = NULL;

 wc.hIcon = LoadIcon(hinstance,MAKEINTRESOURCE(2));

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = "MyMenu";

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

Pour cela on affecte le champ hIcon de la classe de fenêtre avec une icône que l'on charge avec la fonction LoadIcon. Elle reçoit comme paramètre le handle d'instance de l'application, pour son second paramètre elle attend un pointeur sur une chaîne de caractères identifiant l'icône. Dans le script de ressource nous l'avons identifié avec une valeur numérique. C'est avec la macro MAKEINTRESOURCE que nous faions cette conversion.

Nous aurions pu les identifier avec des valeurs littérales :

ICONE1 ICON icone.ico

ICONE2 ICON autre.ico

Dans ce cas LoadIcon aurait reçu un pointeur sur la chaîne de caractère identifiant l'icône dans son second paramètre.

 wc.hIcon = LoadIcon(hinstance,"ICONE2");

**Ressources de type menu :**

Toujours avec le même exemple, nous allons supprimer le menu que l'on avait créé par programme au chapitre 3 et le créer avec un script de ressources :

LEMENU MENU

BEGIN

 POPUP "Fichier"

 BEGIN

 MENUITEM "&Nouveau", IDM\_NEW

 MENUITEM SEPARATOR

 MENUITEM "&Quitter", IDM\_QUIT

 END

 POPUP "Aide"

 BEGIN

 MENUITEM "A propos...", IDM\_ABOUT

 END

END

Nous retrouvons là aussi nos trois éléments : l'identificateur de la ressource (littérale dans cet exemple), suivi de son type (MENU : barre de menu) puis de son contenu. Ici le contenu n'est pas un fichier, mais un script de construction de barre de menu, balisé par les mots BEGIN et END. Elle possède deux sous menu (POPUP : menu déroulant) "fichier" et "Aide". Le premier possède trois options (MENUITEM) "&Nouveau", SEPARATOR et "&Quitter". La liste des options d'un sous menu est aussi balisée par les mots BEGIN et END. Chaque option doit être identifiée par une constante numérique. Cette constante sera envoyée à la procédure de fenêtre quand l'option du menu sera actionnée. Il est préférable de les nommer pour plus de clarté dans le code. Comme elles devront être connues du script de ressource et du fichier source du programme, nous les définissons dans un fichier entête séparé, nommé dans cet exemple "resource.h" :

#define IDM\_QUIT 1

#define IDM\_NEW 2

#define IDM\_ABOUT 3

Ce fichier devra donc être inclut dans le fichier source et dans le fichier script de ressource (voir code à la fin de ce document). Le menu sera ensuite affecté à la classe de fenêtre. Ceci en affectant au champ lpszMenuName de la classe de fenêtre, un pointeur sur une chaîne de caractères contenant le nom litéral du menu :

 wc.lpszMenuName = "LEMENU";

C'est tout pour sa construction. La récupération des actions du menu ce fait sur réception des messages WM\_COMMAND de la même manière qu'au chapitre 3 :

 switch (uMsg)

 {

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 /\*... \*/

Vous avez du remarquer que j'ai mis le caractère '&' dans le texte des options de menu "&Nouveau" et "&Quitter". Ce signe sert à souligner le caractère qui le suit et à donner la possibilité d'activer l'option correspondante avec la touche du clavier correspondant à ce caractère. Ceci uniquement quand le popup menu est ouvert. A ne pas confondre avec les accélérateurs clavier que l'on verra lors d'un chapitre ultérieur.

Nous aurons l'occasion de revenir sur les différents types de ressources au cours des chapitres suivants.

**Code complet :**

**resource.h :**

#define IDM\_QUIT 1

#define IDM\_NEW 2

#define IDM\_ABOUT 3

resource.rc :

#include <windows.h>

#include "resource.h"

1 ICON icone.ico

2 ICON autre.ico

LEMENU MENU

BEGIN

 POPUP "Fichier"

 BEGIN

 MENUITEM "&Nouveau", IDM\_NEW

 MENUITEM SEPARATOR

 MENUITEM "&Quitter", IDM\_QUIT

 END

 POPUP "Aide"

 BEGIN

 MENUITEM "A propos...", IDM\_ABOUT

 END

END

**winmain.c :**

#include <windows.h>

#include "resource.h"

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(hinstance,MAKEINTRESOURCE(2));

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = "LEMENU";

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 static BOOL EditNotChg = TRUE;

 switch (uMsg)

 {

 case WM\_CREATE:

 {

 HFONT hFont;

 hEdit = CreateWindow("edit", "Texte",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 hFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

 return 0;

 }

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(LOWORD(wParam) == IDM\_NEW)

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer votre travail ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 {

 SendMessage(hEdit,WM\_SETTEXT,0,(long)"");

 EditNotChg = TRUE;

 }

 if(LOWORD(wParam) == IDM\_ABOUT)

 MessageBox(hwnd,"Mon beau programme !","A propos !",MB\_OK);

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

J'ai testé les compilations avec C++ Builder et DevC++

**8 - Les boîtes de dialogue personnalisées.**

**Introduction :**

Dans ce chapitre nous allons voir comment créer et appeler une boîte de dialogue personnalisée. Nous continuerons avec l'exemple du chapitre précédent. Nous remplacerons la boîte de message appelée par le menu "A propos..." par une boîte de dialogue personnalisée contenant un bouton, une icône et un texte.



**Le script de ressources :**

Les boîtes de dialogue personnalisées sont créées à partir de ressources :

DIALOG1 DIALOG

 60, 60, 160, 80

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "A propos"

BEGIN

 DEFPUSHBUTTON "Ok", IDOK, 56, 50, 42, 12

 ICON 2, -1, 20, 15, 32, 32

 LTEXT "Mon beau programme !", -1, 60, 18, 80, 10

END

La ressource est composée de son identificateur littéral, suivi de son type (DIALOG), suivie de ses propriétés (position, dimensions, style et titre), suivi de son contenu balisé par les mots BEGIN et END. Il est en général constitué de contrôles. Chaque contrôle est décrit par son type, suivi de son identification visuelle (texte pour le bouton et le contrôle texte, identificateur de ressource pour l'icône), suivi de sa constante numérique d'identification, puis de sa position dans la boîte de dialogue et enfin de ses dimensions (les largeurs et hauteurs des contrôles ne sont pas en pixel, mais dépendent de la police de caractères utilisée). L'icône et le le contrôle texte ont un identificateur à -1 car il ne sont là que pour la décoration, mais il faut tout de même leur mettre un identificateur. Celui du bouton est à IDOK que nous n'avons pourtant pas défini. IDOK est défini dans Windows, il est envoyé à la procédure de fenêtre de la boîte de dialogue quand on appuie sur la touche "Entrée". Est défini aussi l'identificateur IDCANCEL qui lui est envoyé quand on appuie sur la touche "Echap" ou que l'on tente de fermer la boîte de dialogue.

**Appel de la boîte de dialogue :**

La boîte de dialogue étant maintenant dans les ressources, nous pouvons l'appeler afin de l'ouvrir. C'est le rôle de la fonction DialogBox. Son premier paramètre est le handle d'instance de l'application, le second, un pointeur sur la chaîne de caractères identifiant la ressource, le troisième est le handle de la fenêtre parent et le dernier un pointeur sur sa procédure de fenêtre.

 switch (uMsg)

 {

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_ABOUT)

 DialogBox(hinst, "DIALOG1" , hwnd, (DLGPROC)Dialog1Proc);

**Procédure de fenêtre de la boîte de dialogue :**

Comme pour la procédure de fenêtre de la fenêtre principale c'est à vous de la définir. C'est Windows qui l'appellera quand elle aura un message.

LRESULT CALLBACK Dialog1Proc(HWND, UINT, WPARAM, LPARAM);

BOOL APIENTRY Dialog1Proc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 return TRUE;

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDCANCEL || LOWORD(wParam) == IDOK)

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

 default:

 return FALSE;

 }

}

Comme vous pouvez le constater elle est fort semblable à celle de la fenêtre principale. Si le message est traité, elle doit renvoyer TRUE sinon elle doit renvoyer FALSE. WM\_INITDIALOG doit être intercepté et renvoyer TRUE afin que les commandes IDOK et IDCANCEL soit envoyées lors des appuis sur les touches correspondantes. Le message WM\_INITDIALOG est envoyé par Windows après la création de la boîte de dialogue, mais avant qu'elle soit visible. Ceci afin de permettre d'initialiser les contrôles qu'elle contient.

Dans l'exemple nous interceptons seulement les commandes IDOK et IDCANCEL afin de la fermer par l'appel de la fonction EndDialog.

**Code complet :**

**resource.h :**

#define IDM\_QUIT 1

#define IDM\_NEW 2

#define IDM\_ABOUT 3

resource.rc

#include <windows.h>

#include "resource.h"

1 ICON icone.ico

2 ICON autre.ico

LEMENU MENU

BEGIN

 POPUP "Fichier"

 BEGIN

 MENUITEM "Nouveau", IDM\_NEW

 MENUITEM SEPARATOR

 MENUITEM "Quitter", IDM\_QUIT

 END

 POPUP "Aide"

 BEGIN

 MENUITEM "A propos...", IDM\_ABOUT

 END

END

DIALOG1 DIALOG

 60, 60, 160, 80

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "A propos"

BEGIN

 DEFPUSHBUTTON "Ok", IDOK, 56, 50, 42, 12

 ICON 2, -1, 20, 15, 32, 32

 LTEXT "Mon beau programme !", -1, 60, 18, 80, 10

END

**winmain.c :**

#include <windows.h>

#include "resource.h"

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

BOOL APIENTRY Dialog1Proc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(hinstance,MAKEINTRESOURCE(2));

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = "LEMENU";

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 static BOOL EditNotChg = TRUE;

 switch (uMsg)

 {

 case WM\_CREATE:

 {

 HFONT hFont;

 hEdit = CreateWindow("edit", "Texte",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 hFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

 return 0;

 }

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(LOWORD(wParam) == IDM\_NEW)

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer votre travail ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 {

 SendMessage(hEdit,WM\_SETTEXT,0,(long)"");

 EditNotChg = TRUE;

 }

 if(LOWORD(wParam) == IDM\_ABOUT)

 DialogBox(hinst, "DIALOG1" , hwnd, (DLGPROC)Dialog1Proc);

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL APIENTRY Dialog1Proc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 return TRUE;

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDCANCEL || LOWORD(wParam) == IDOK)

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

default:

 return FALSE;

 }

}

J'ai testé les compilations avec C++ Builder et DevC++. Le compilateur de ressources de DevC++ a été défaillant pour l'affichage de l'icône dans la boîte de dialogue.

**9 - Communiquer avec les boîtes de dialogue.**

**Introduction :**

Maintenant que nous savons créer une boîte de dialogue, nous allons voir comment initialiser ses contrôles et comment restituer leur contenu à la fenêtre qui a appelé la boîte de dialogue. Nous continuons toujours avec le même exemple où nous ajouterons une option au menu nommé "Paramètres...". Elle appellera une nouvelle boîte de dialogue qui comportera un champ de saisie (EDITTEXT), un bouton "Ok" et un bouton "Annuler". A l'ouverture de la boîte de dialogue, le champ de saisie devra être initialisé avec le titre de la fenêtre principale. A sa fermeture, le titre de la fenêtre principale devra être modifié si la boîte de dialogue est fermée avec le bouton "Ok".



**Le script de ressources :**

Le script de ressources est fort semblable à celui de la boîte de dialogue du chapitre précédent :

DIALOG2 DIALOG

 60, 60, 182, 70

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "Paramètres"

BEGIN

 DEFPUSHBUTTON "OK", IDOK, 36, 42, 42, 12

 PUSHBUTTON "Cancel", IDCANCEL, 96, 42, 42, 12

 EDITTEXT IDE\_EDIT1, 88, 15, 74, 12

 LTEXT "Titre de la fenêtre", -1, 24, 18, 60, 10

END

On peut remarquer deux types de boutons "DEFPUSHBUTTON" et "PUSHBUTTON". Le bouton qui à le type "DEFPUSHBUTTON" est le bouton par défaut. Le contrôle d'édition "EDITTEXT" est identifié par la constante IDE\_EDIT1 définie dans le fichier resource.h.

**Appel de la boîte de dialogue :**

Nous allons appeler la boîte de dialogue avec la fonction DialogBoxParam. Cette fonction permet de passer un paramètre supplémentaire de notre choix. Il pourra être récupéré dans sa procédure de fenêtre.

Dans l'exemple, nous lui passerons le handle de la fenêtre qui l'appelle.

 switch (uMsg)

 {

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_PARAM)

 DialogBoxParam(hinst, "DIALOG2" , hwnd,

 (DLGPROC)Dialog2Proc, (LPARAM)hwnd);

**Procédure de fenêtre de la boîte de dialogue :**

**Récupération des données venant de la fenêtre principale :**

BOOL APIENTRY Dialog2Proc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 static HWND hParent;

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 {

 CHAR st[256];

 hParent = (HWND)lParam;

 GetWindowText(hParent, st, 256);

 SetDlgItemText(hDlg, IDE\_EDIT1, st);

 }

 return TRUE;

A la réception du message WM\_INITDIALOG nous récupérons le titre de la fenêtre qui l'a appelée avec GetWindowText. Son handle est récupéré dans la variable lParam transmise avec WM\_INITDIALOG. C'est le paramètre supplémentaire transmit par la fonction DialogBoxParam. Nous affectons ce titre au champ d'édition avec la fonction SetDlgItemText. Elle reçoit comme paramètres le handle de la boîte de dialogue, puis la constante identifiant le contrôle d'édition, puis le pointeur sur la chaîne de caractères.

Maintenant que notre boîte de dialogue est ouverte, nous pouvons modifier le champ de saisie, puis la fermer.

Si nous la fermons avec le bouton "Ok" :

 switch (uMsg)

 {

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDOK )

 {

 CHAR st[256];

 GetDlgItemText(hDlg, IDE\_EDIT1, st, 256);

 SetWindowText(hParent,st);

 EndDialog(hDlg,0);

 return TRUE;

 }

Nous faisons l'opération inverse. Nous récupérons le contenu du champ d'édition avec la fonction GetDlgItemText et nous l'affectons à la fenêtre principale dont on avait mémorisé le handle dans la variable hParent.

Si vous m'avez suivi jusque là, vous vous rendez compte qu'il y a plusieurs possibilités pour transmettre les données entres fenêtres. Celle-ci n'était peut-être pas la meilleure dans cet exemple. On aurait pu utiliser des variables globales ou transmettre un pointeur sur structure dans le paramètre supplémentaire de DialogBoxParam...

**Styles étendus des contrôles :**

Vous avez remarqué que le contrôle d'édition dans notre boîte de dialogue a un aspect 3D et pas celui qui ce trouve sur la fenêtre principale. Pour donner un aspect 3D à un contrôle qui se trouve sur une fenêtre qui n'est pas une boîte de dialogue, il faut lui donner le style étendu WS\_EX\_CLIENTEDGE. Pour cela il faut créer le contrôle avec la fonction CreateWindowEx. Les styles étendus sont donnés dans le premier paramètre de CreateWindowEx. Les autres paramètres de cette fonction sont les mêmes que ceux de CreateWindow.

 hEdit = CreateWindowEx(WS\_EX\_CLIENTEDGE, "edit", "Texte",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

**Code complet :**

**resource.h :**

#define IDM\_QUIT 1

#define IDM\_NEW 2

#define IDM\_ABOUT 3

#define IDM\_PARAM 4

#define IDE\_EDIT1 101

**resource.rc :**

#include <windows.h>

#include "resource.h"

1 ICON icone.ico

2 ICON autre.ico

LEMENU MENU

BEGIN

 POPUP "Fichier"

 BEGIN

 MENUITEM "Nouveau", IDM\_NEW

 MENUITEM "Paramètres...", IDM\_PARAM

 MENUITEM SEPARATOR

 MENUITEM "Quitter", IDM\_QUIT

 END

 POPUP "Aide"

 BEGIN

 MENUITEM "A propos...", IDM\_ABOUT

 END

END

DIALOG2 DIALOG

 60, 60, 182, 70

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "Paramètres"

BEGIN

 DEFPUSHBUTTON "OK", IDOK, 36, 42, 42, 12

 PUSHBUTTON "Cancel", IDCANCEL, 96, 42, 42, 12

 EDITTEXT IDE\_EDIT1, 88, 15, 74, 12

 LTEXT "Titre de la fenêtre", -1, 24, 18, 60, 10

END

DIALOG1 DIALOG

 60, 60, 160, 80

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "A propos"

BEGIN

 DEFPUSHBUTTON "Ok", IDOK, 56, 50, 42, 12

 ICON 2, -1, 20, 15, 32, 32

 LTEXT "Mon beau programme !", -1, 60, 18, 80, 10

END

winmain.c :

#include <windows.h>

#include "resource.h"

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

BOOL APIENTRY Dialog1Proc(HWND, UINT, WPARAM, LPARAM);

BOOL APIENTRY Dialog2Proc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(hinstance,MAKEINTRESOURCE(2));

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = "LEMENU";

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 static BOOL EditNotChg = TRUE;

 switch (uMsg)

 {

 case WM\_CREATE:

 {

 HFONT hFont;

 hEdit = CreateWindowEx(WS\_EX\_CLIENTEDGE, "edit", "Texte",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 hFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

 return 0;

 }

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(LOWORD(wParam) == IDM\_NEW)

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer votre travail ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 {

 SendMessage(hEdit,WM\_SETTEXT,0,(LPARAM)"");

 EditNotChg = TRUE;

 }

 if(LOWORD(wParam) == IDM\_ABOUT)

 DialogBox(hinst, "DIALOG1" , hwnd, (DLGPROC)Dialog1Proc);

 if(LOWORD(wParam) == IDM\_PARAM)

 DialogBoxParam(hinst, "DIALOG2" , hwnd,

 (DLGPROC)Dialog2Proc, (LPARAM)hwnd);

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL APIENTRY Dialog1Proc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 return TRUE;

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDCANCEL || LOWORD(wParam) == IDOK)

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

 default:

 return FALSE;

 }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL APIENTRY Dialog2Proc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 static HWND hParent;

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 {

 CHAR st[256];

 hParent = (HWND)lParam;

 GetWindowText(hParent, st, 256);

 SetDlgItemText(hDlg, IDE\_EDIT1, st);

 }

 return TRUE;

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDOK )

 {

 CHAR st[256];

 GetDlgItemText(hDlg, IDE\_EDIT1, st, 256);

 SetWindowText(hParent,st);

 EndDialog(hDlg,0);

 return TRUE;

 }

 if (LOWORD(wParam) == IDCANCEL )

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

 default:

 return FALSE;

 }

}

J'ai testé les compilations avec C++ Builder et DevC++.

Voici un exemple semblable avec le contrôle case à cocher : **Le contrôle CheckBox ou case à cocher.**

**Le contrôle CheckBox ou case à cocher.**

**Introduction :**

Dans ce chapitre nous allons aborder le traitement des cases à cocher dans les boîtes de dialogue. Vous connaissez la case à cocher, elle sert à refléter un état vrai ou faux, parfois indéterminé. Pour cet exemple nous dessinerons un carré et un cercle sur la fenêtre principale. Ils auront la faculté d'être visible ou caché. L'utilisateur pourra changer cet état dans une boîte de dialogue.



**Le script de ressources :**

DIALOG1 DIALOG

 60, 70, 175, 80

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "Propriétés"

BEGIN

 DEFPUSHBUTTON "OK", IDOK, 36, 52, 42, 12

 PUSHBUTTON "Cancel", IDCANCEL, 96, 52, 42, 12

 AUTOCHECKBOX "Carré visible ", ID\_CB1, 60, 15, 80, 15

 AUTOCHECKBOX "Cercle visible", ID\_CB2, 60, 30, 80, 15

END

Du point de vu des ressources, il existe deux type de case à cocher la CHECKBOX et l'AUTOCHECKBOX. Dans cet exemple nous choisirons l'AUTOCHECKBOX car elle reflète automatiquement son état quand l'utilisateur clique dessus. Dans l'exemple nous utiliserons deux cases à cocher identifiées par les constantes ID\_CB1 et ID\_CB2.

**Passage des paramètres :**

Nous appellerons la boîte de dialogue avec la fonction DialogBox suite à l'action de l'option de menu "propriété..." identifié par la constante IDM\_PROP. Nous demandons de redessiner la zone client de la fenêtre principale au retour de l'appel de la boîte de dialogue à l'aide de la fonction InvalidateRect :

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_PROP)

 {

 DialogBox(hinst, "DIALOG1" , hwnd, (DLGPROC)Dialog1Proc);

 InvalidateRect(hwnd,NULL,TRUE);

 }

Nous mémoriserons les propriétés "visible" du carré et du cercle dans une variable globale. Elle aura comme type une structure PROPRIETE préalablement défini :

typedef struct

{

 BOOL CarreVisible;

 BOOL CercleVisible;

} PROPRIETE ;

PROPRIETE prop;

**Récupération des données :**

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 {

 if(prop.CarreVisible == TRUE)

 CheckDlgButton(hDlg, ID\_CB1, BST\_CHECKED);

 if(prop.CercleVisible == TRUE)

 CheckDlgButton(hDlg, ID\_CB2, BST\_CHECKED);

 return TRUE;

 }

A la réception du message WM\_INITDIALOG dans la procédure de fenêtre de la boîte de dialogue, nous testons les valeurs des propriétés du carré et du cercle mémorisées la variable prop, puis nous reflétons ces valeurs dans les cases à cocher correspondantes. Ceci à l'aide de la fonction CheckDlgButton. Elle reçoit comme paramètre le handle de la boîte de dialogue, puis l'identificateur de la case à cocher, puis l'état de la case à cocher (BST\_CHECKED pour coché et BST\_UNCHECKED pour non coché)

Maintenant que notre boîte de dialogue est ouverte, nous pouvons faire notre sélection, puis la fermer.

Si nous la fermons avec le bouton "Ok" :

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDOK )

 {

 if(IsDlgButtonChecked(hDlg, ID\_CB1) == BST\_CHECKED)

 prop.CarreVisible = TRUE;

 else prop.CarreVisible = FALSE;

 if(IsDlgButtonChecked(hDlg, ID\_CB2) == BST\_CHECKED)

 prop.CercleVisible = TRUE;

 else prop.CercleVisible = FALSE;

 EndDialog(hDlg,0);

 return TRUE;

 }

Nous faisons l'opération inverse. Nous testons l'état des cases à cocher avec la fonction IsDlgButtonChecked et nous affectons les propriétés du carré et du cercle en conséquence.

IsDlgButtonChecked reçoit comme paramètres le handle de la boîte de dialogue puis la constante d'identification de la case à cocher en question.

Nous devons bien sûr refléter le résultat sur le dessin du carré et du cercle. Ceci sera fait à la réception du message WM\_PAINT de la fenêtre principale.

 case WM\_PAINT :

 {

 PAINTSTRUCT ps;

 HDC hdc = BeginPaint(hwnd, &ps);

 if(prop.CarreVisible == TRUE) Rectangle(hdc, 20, 20, 170, 170);

 if(prop.CercleVisible == TRUE) Ellipse(hdc, 200, 20, 350, 170);

 EndPaint(hwnd, &ps);

 return 0;

 }

**Code complet :**

**resource.h :**

#define IDM\_QUIT 1

#define IDM\_PROP 2

#define ID\_CB1 101

#define ID\_CB2 102

**resource.rc :**

#include <windows.h>

#include "resource.h"

LEMENU MENU

BEGIN

 POPUP "Fichier"

 BEGIN

 MENUITEM "Propriétés...", IDM\_PROP

 MENUITEM SEPARATOR

 MENUITEM "Quitter", IDM\_QUIT

 END

END

DIALOG1 DIALOG

 60, 70, 175, 80

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "Propriétés"

BEGIN

 DEFPUSHBUTTON "OK", IDOK, 36, 52, 42, 12

 PUSHBUTTON "Cancel", IDCANCEL, 96, 52, 42, 12

 AUTOCHECKBOX "Carré visible ", ID\_CB1, 60, 15, 80, 15

 AUTOCHECKBOX "Cercle visible", ID\_CB2, 60, 30, 80, 15

END

**winmain.c :**

#include <windows.h>

#include "resource.h"

typedef struct

{

 BOOL CarreVisible;

 BOOL CercleVisible;

} PROPRIETE ;

PROPRIETE prop;

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

BOOL APIENTRY Dialog1Proc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 hinst = hinstance;

 wc.style = 0;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = (HBRUSH)(1 + COLOR\_BTNFACE);

 wc.lpszMenuName = "LEMENU";

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Check Box", WS\_OVERLAPPED | WS\_SYSMENU,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 260,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

 UpdateWindow(hwnd);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_CREATE:

 prop.CarreVisible = TRUE;

 prop.CercleVisible = TRUE;

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(LOWORD(wParam) == IDM\_PROP)

 {

 DialogBox(hinst, "DIALOG1" , hwnd, (DLGPROC)Dialog1Proc);

 InvalidateRect(hwnd,NULL,TRUE);

 }

 return 0;

 case WM\_PAINT :

 {

 PAINTSTRUCT ps;

 HDC hdc = BeginPaint(hwnd, &ps);

 if(prop.CarreVisible == TRUE) Rectangle(hdc, 20, 20, 170, 170);

 if(prop.CercleVisible == TRUE) Ellipse(hdc, 200, 20, 350, 170);

 EndPaint(hwnd, &ps);

 return 0;

 }

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL APIENTRY Dialog1Proc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 {

 if(prop.CarreVisible == TRUE)

 CheckDlgButton(hDlg, ID\_CB1, BST\_CHECKED);

 if(prop.CercleVisible == TRUE)

 CheckDlgButton(hDlg, ID\_CB2, BST\_CHECKED);

 return TRUE;

 }

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDOK )

 {

 if(IsDlgButtonChecked(hDlg, ID\_CB1) == BST\_CHECKED)

 prop.CarreVisible = TRUE;

 else prop.CarreVisible = FALSE;

 if(IsDlgButtonChecked(hDlg, ID\_CB2) == BST\_CHECKED)

 prop.CercleVisible = TRUE;

 else prop.CercleVisible = FALSE;

 EndDialog(hDlg,0);

 return TRUE;

 }

 if (LOWORD(wParam) == IDCANCEL)

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

 default:

 return FALSE;

 }

}

J'ai testé les compilations avec C++ Builder et DevC++.

**10 Les accélérateurs claviers.**

**Introduction :**

Les accélérateurs claviers sont des raccourcis clavier qui servent à lancer des actions. Actions qui la plus part du temps peuvent aussi se lancer depuis le menu.

Nous allons continuer sur la base de l'exemple du chapitre précédent. Nous affecterons un raccourcis clavier aux commandes des options suivantes du menu : Nouveau, Paramètres, A propos. Quitter à toujours un raccourci clavier qui est "Alt F4" :



**Le script de ressources :**

Les accélérateurs claviers sont définis dans les ressources sous forme d'une table :

LesAccel ACCELERATORS

BEGIN

 "N", IDM\_NEW, CONTROL, VIRTKEY

 "P", IDM\_PARAM, CONTROL, VIRTKEY

 VK\_F1, IDM\_ABOUT, VIRTKEY

END

Le premier élément est le nom littéral de la ressource, le deuxième son type (ACCELERATORS), suivi de son contenu balisé par BEGIN et END. Le contenu est la table de raccourcis clavier proprement dite où chaque raccourci est décrit par la touche concernée décrite sous forme de chaîne de caractère ou de code de touche virtuel, suivie de la constante d'identification de l'action. Ici les constantes sont les mêmes que les options de menu que l'on veut shunter, suivient de ou des touches étendues : CONTROL, ALT , SHIFT. Et de VIRTKEY ou ASCII.

**Chargement des tables :**

Pour être fonctionnelles, les tables de raccourcis clavier doivent être chargées par l'application :

 HACCEL haccel;

 haccel = LoadAccelerators(hinstance, "LesAccel");

C'est le rôle de la fonction LoadAccelerators qui reçoit comme paramètres le handle d'instance de l'application suivie du nom de la table. Celui que l'on lui avait donné dans le script de ressources. La fonction nous retourne ensuite un handle de table d'accélérateurs.

Comme le clavier nous envoie que des messages d'appui ou de relâchement de touche, nous devons donc traduire ces messages en commande pour les touches faisant partie de la table d'accélérateurs. C'est le rôle de la fonction TranslateAccelerator. Elle reçoit comme paramètres, le handle de la fenêtre, le handle de la table d'accélérateurs et l'adresse du message à traiter. Elle s'utilise dans la boucle de message :

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

 {

 if (!TranslateAccelerator(hwnd, haccel, &msg))

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 }

Quand TranslateAccelerator traite un message WM\_KEYDOWN, elle vérifie si le code de la touche se trouve dans la table. Si c'est le cas, elle envoie un message WM\_COMMAND à la procédure de fenêtre avec la commande correspondante. Dans ce cas, elle retourne alors la valeur TRUE, que l'on testera afin de ne pas traiter le message WM\_KEYDOWN.

Il est d'usage d'indiquer les raccourcis clavier sur le texte des options de menu :

LEMENU MENU

BEGIN

 POPUP "Fichier"

 BEGIN

 MENUITEM "Nouveau\tCtrl+N", IDM\_NEW

 MENUITEM "Paramètres...\tCtrl+P", IDM\_PARAM

 MENUITEM SEPARATOR

 MENUITEM "Quitter\tAlt+F4", IDM\_QUIT

 END

 POPUP "Aide"

 BEGIN

 MENUITEM "A propos...", IDM\_ABOUT

 END

END

**Code complet :**

**resource.h :**

#define IDM\_QUIT 1

#define IDM\_NEW 2

#define IDM\_ABOUT 3

#define IDM\_PARAM 4

#define IDE\_EDIT1 101

**resource.rc :**

#include <windows.h>

#include "resource.h"

1 ICON chip.ico

2 ICON factory.ico

LEMENU MENU

BEGIN

 POPUP "Fichier"

 BEGIN

 MENUITEM "Nouveau\tCtrl+N", IDM\_NEW

 MENUITEM "Paramètres...\tCtrl+P", IDM\_PARAM

 MENUITEM SEPARATOR

 MENUITEM "Quitter\tAlt+F4", IDM\_QUIT

 END

 POPUP "Aide"

 BEGIN

 MENUITEM "A propos...", IDM\_ABOUT

 END

END

LesAccel ACCELERATORS

BEGIN

 "N", IDM\_NEW, CONTROL, VIRTKEY

 "P", IDM\_PARAM, CONTROL, VIRTKEY

 VK\_F1, IDM\_ABOUT, VIRTKEY

END

DIALOG2 DIALOG

 60, 60, 182, 70

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "Paramètres"

BEGIN

 DEFPUSHBUTTON "OK", IDOK, 36, 42, 42, 12

 PUSHBUTTON "Cancel", IDCANCEL, 96, 42, 42, 12

 EDITTEXT IDE\_EDIT1, 88, 15, 74, 12

 LTEXT "Titre de la fenêtre", -1, 24, 18, 60, 10

END

DIALOG1 DIALOG

 60, 60, 160, 80

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "A propos"

BEGIN

 DEFPUSHBUTTON "Ok", IDOK, 56, 50, 42, 12

 ICON 2, -1, 20, 15, 32, 32

 LTEXT "Mon beau programme !", -1, 60, 18, 80, 10

END

**winmain.c :**

#include <windows.h>

#include "resource.h"

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

BOOL APIENTRY Dialog1Proc(HWND, UINT, WPARAM, LPARAM);

BOOL APIENTRY Dialog2Proc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 HACCEL haccel;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(hinstance,MAKEINTRESOURCE(2));

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = "LEMENU";

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

 haccel = LoadAccelerators(hinstance, "LesAccel");

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

 {

 if (!TranslateAccelerator(hwnd, haccel, &msg))

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 static BOOL EditNotChg = TRUE;

 switch (uMsg)

 {

 case WM\_CREATE:

 {

 HFONT hFont;

 hEdit = CreateWindowEx(WS\_EX\_CLIENTEDGE, "edit", "Texte",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 hFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

 return 0;

 }

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(LOWORD(wParam) == IDM\_NEW)

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer votre travail ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 {

 SendMessage(hEdit,WM\_SETTEXT,0,(LPARAM)"");

 EditNotChg = TRUE;

 }

 if(LOWORD(wParam) == IDM\_ABOUT)

 DialogBox(hinst, "DIALOG1" , hwnd, (DLGPROC)Dialog1Proc);

 if(LOWORD(wParam) == IDM\_PARAM)

 DialogBoxParam(hinst, "DIALOG2" , hwnd,

 (DLGPROC)Dialog2Proc, (LPARAM)hwnd);

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL APIENTRY Dialog1Proc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 return TRUE;

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDCANCEL || LOWORD(wParam) == IDOK)

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

 default:

 return FALSE;

 }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL APIENTRY Dialog2Proc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 static HWND hParent;

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 {

 char st[256];

 hParent = (HWND)lParam;

 GetWindowText(hParent, st, 255);

 SetDlgItemText(hDlg, IDE\_EDIT1, st);

 }

 return TRUE;

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDOK )

 {

 char st[256];

 GetDlgItemText(hDlg, IDE\_EDIT1, st, 255);

 SetWindowText(hParent,st);

 EndDialog(hDlg,0);

 return TRUE;

 }

 if (LOWORD(wParam) == IDCANCEL )

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

 default:

 return FALSE;

 }

}

J'ai testé les compilations avec C++ Builder et DevC++.

**11 - Une boîte de dialogue comme fenêtre principale.**

**Introduction :**

Un petit chapitre pour montrer qu'on peut utiliser une boîte de dialogue comme fenêtre principale. Cela peut être intéressant pour des applications dont la fenêtre principale comporte beaucoup de contrôles dans sa zone client. Nous allons pour cet exemple créer une boîte de dialogue comportant un contrôle texte, un contrôle d'édition et un bouton. Le contrôle d'édition ne pourra recevoir que des chiffres.



**Mise en oeuvre :**

La boîte de dialogue est construite à partir de ressources comme dans les autres chapitres. Mais dans ce cas nous l'appellerons directement dans la fonction WinMain :

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 DialogBox(hInstance,"DIALOG1",NULL,(DLGPROC)DlgProc);

 return 0;

}

Le traitement de la boîte de dialogue est identique à une boîte de dialogue ordinaire.

**Code complet :**

**resource.h :**

#define IDE\_EDIT1 101

**resource.rc :**

#include <windows.h>

#include "resource.h"

DIALOG1 DIALOG

 60, 60, 160, 80

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "Dialogue Principale"

BEGIN

 DEFPUSHBUTTON "Ok", IDOK, 56, 55, 42, 12

 LTEXT "Saisie d'un nombre :", -1, 45, 15, 80, 10

 EDITTEXT IDE\_EDIT1, 45, 30, 60, 12, ES\_NUMBER | ES\_RIGHT

END

**winmain.c :**

#include <windows.h>

#include "resource.h"

BOOL APIENTRY DlgProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 DialogBox(hInstance,"DIALOG1",NULL,(DLGPROC)DlgProc);

 return 0;

}

//---------------------------------------------------------------------------

BOOL APIENTRY DlgProc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 return TRUE;

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDOK)

 {

 CHAR st[256];

 UINT valeur = GetDlgItemInt(hDlg, IDE\_EDIT1, NULL, FALSE);

 wsprintf(st, "Vous avez choisi le nombre %d", valeur);

 MessageBox(hDlg, st, "Résultat", MB\_OK);

 }

 if (LOWORD(wParam) == IDCANCEL)

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

 default:

 return FALSE;

 }

}

J'ai testé les compilations avec C++ Builder et DevC++.

**12 - Les contrôles supplémentaires de Windows.**

**Introduction :**

Windows permet d'utiliser des contrôles supplémentaires gérés par une bibliothèque dynamique appelée : "Common control library" (comctrl32.dll)

Nous allons commencer par le plus simple : la barre de progression. Vous connaissez la barre de progression, celle qui nous fait patienter le temps d'un traitement long.

Nous allons pour cet exemple créer une boîte de dialogue appelée directement depuis la fonction d'entrée WinMain. Elle comportera : une barre de progression et un bouton qui la fera avancer d'une unité à chaque appui.



**Mise en oeuvre :**

Les compilateurs de ressource ne connaissant pas ce contrôle nous utiliserons le type générique CONTROL pour la créer.

 CONTROL "", IDE\_PB1, PROGRESS\_CLASS, 0 ,10, 20, 140, 12

Il est suivi du texte, puis de son identificateur, puis du nom de sa classe de fenêtre, puis de son style, puis de sa position et de ses dimensions.

Ces contrôles pour être utilisé doivent s'assurer que comctrl32.dll est chargée. C'est le rôle de la fonction InitCommonControls :

 InitCommonControls();

Cette fonction doit être appelée une seule fois avant l'utilisation du contrôle.

**La barre de progression :**

L'utilisation de la barre de progression est très simple. Avant son utilisation, vous lui envoyez un message PBM\_SETRANGE pour définir ses bornes :

 SendDlgItemMessage(hDlg, IDE\_PB1, PBM\_SETRANGE, 0, MAKELONG(0,10));

Celle-ci dans les mots de poids faible et de poids fort du paramètre lParam, reconstitué avec la macro MAKELONG.

Ensuite, Vous initialisez un pas de progression en lui envoyant un message PBM\_SETSTEP :

 SendDlgItemMessage(hDlg, IDE\_PB1, PBM\_SETSTEP, 1, 0);

Le pas étant passé avec le paramètre wParam du message.

Enfin, vous la faite avancer d'un pas avec le message PBM\_STEPIT :

 SendDlgItemMessage(hDlg, IDE\_PB1, PBM\_STEPIT, 0, 0);

Ce que nous ferons a chaque action du bouton.

**Code complet :**

**resource.h :**

#define IDE\_PB1 101

**resource.rc :**

#include <windows.h>

#include "resource.h"

DIALOG1 DIALOG

 60, 60, 160, 80

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "ProgressBar"

BEGIN

 DEFPUSHBUTTON "Progress", IDOK, 56, 55, 42, 12

 CONTROL "", IDE\_PB1, PROGRESS\_CLASS, 0 ,10, 20, 140, 12

END

**winmain.c :**

#include <windows.h>

#include <commctrl.h>

#include "resource.h"

BOOL APIENTRY DlgProc(HWND Dlg,UINT message,WPARAM wParam,LPARAM lParam);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 InitCommonControls();

 DialogBox(hInstance,"DIALOG1",NULL,(DLGPROC)DlgProc);

 return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL APIENTRY DlgProc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 SendDlgItemMessage(hDlg, IDE\_PB1, PBM\_SETRANGE, 0, MAKELONG(0,10));

 SendDlgItemMessage(hDlg, IDE\_PB1, PBM\_SETSTEP, 1, 0);

 return TRUE;

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDOK)

 {

 SendDlgItemMessage(hDlg, IDE\_PB1, PBM\_STEPIT, 0, 0);

 return TRUE;

 }

 if (LOWORD(wParam) == IDCANCEL)

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

 default:

 return FALSE;

 }

}

Dans le document suivant : "Le contrôle barre d'état". Un contrôle de la "Common control library" sera créé sur la fenêtre principale.

J'ai testé les compilations avec C++ Builder et Dev-C++.

**La barre d'état ou Status Bar.**

**Introduction :**

Après la barre de progression nous allons voir un deuxième contrôle de la bibliothèque dynamique "Common control library" : la barre d'état. Vous l'avez sûrement vu dans divers logiciel, c'est celle qui ce trouve en bas des fenêtres et qui sert à donner des informations.

Elle sera créée dans la fenêtre pricipale. Nous reprendrons l'exemple du chapitre 10 "Communiquer avec les boîtes de dialogue", auquel nous ajouterons une barre d'état. Elle affichera des informations textuelles pour chaque option de menu. Ceci quand elle seront sélectionnées.



**Mise en oeuvre :**

Comme elle sera sur la fenêtre principale nous la créons avec la fonction CreateStatusWindow. On aurait pu utiliser CreateWindow mais CreateStatusWindow est plus simple.

 InitCommonControls();

 hsb = CreateStatusWindow(WS\_CHILD | WS\_VISIBLE, "Texte", hwnd, -1);

Comme pour la barre de progression, il faut appeler la fonction InitCommonControls avant de créer la barre. C'est tout pour sa création. Mais il va falloir gérer son redimensionnement quand les dimensions de sa fenêtre parent sont changées par l'utilisateur. Cela se fera comme pour le contrôle d'édition, à la réception du message WM\_SIZE de la fenêtre parent.

 case WM\_SIZE:

 {

 RECT sbRect;

 UINT sbheight;

 GetWindowRect(hsb, &sbRect);

 sbheight = sbRect.bottom - sbRect.top;

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam)-sbheight,

 TRUE);

 MoveWindow(hsb, 0, HIWORD(lParam)-sbheight, LOWORD(lParam),

 sbheight, TRUE);

 return 0;

 }

On récupére ses dimensions à l'aide de la fonction GetWindowRect, puis on positionne le contrôle d'édition avec la fonction MoveWindow afin qu'il laisse la place pour la barre d'état et enfin on positionne la barre d'état sur la place restante toujours avec la fonction MoveWindow.

Notre but est maintenant d'y afficher des informations quand des options de menus sont sélectionnées (ça ne veut pas dire actionnées, mais seulement quelles sont en surbrillances). Pour cela nous allons intercepter le message WM\_MENUSELECT qui est envoyé à la fenêtre quand une option de son menu est sélectionnée.

 case WM\_MENUSELECT:

 {

 if (lParam == (LONG)GetMenu(hwnd))

 {

 if(LOWORD(wParam) == 0)

 SendMessage(hsb, SB\_SETTEXT, 0, (LONG)"Menu fichiers");

 if(LOWORD(wParam) == 1)

 SendMessage(hsb, SB\_SETTEXT, 0, (LONG)"Menu Aide");

 }

 else

 {

 if(LOWORD(wParam) == IDM\_QUIT)

 SendMessage(hsb, SB\_SETTEXT, 0,

 (LONG)"Quitter l'application.");

 if(LOWORD(wParam) == IDM\_NEW)

 SendMessage(hsb, SB\_SETTEXT, 0,

 (LONG)"Crée un nouveau fichier.");

 if(LOWORD(wParam) == IDM\_PARAM)

 SendMessage(hsb, SB\_SETTEXT, 0,

 (LONG)"Changer les paramètres.");

 if(LOWORD(wParam) == IDM\_ABOUT)

 SendMessage(hsb, SB\_SETTEXT, 0, (LONG)"A Propos...");

 }

 return 0;

 }

Puis nous testons le mot de poids faible du paramètre wParam qui est joint au message pour savoir de quelle option il s'agit. Ce mot contient identificateur de l'option. Le paramètre lParam contient le handle de menu, nous nous en servons pour savoir s'il s'agit du menu principal ou non.

Suite à la réception du message, nous modifions le texte de la barre d'état en lui envoyant un message SB\_SETTEXT avec un pointeur sur la chaîne de caractère contenant le texte souhaité dans son paramètre lParam.

Nous changerons de même ce texte quand le menu ne sera plus en cours d'utilisation :

 case WM\_EXITMENULOOP:

 SendMessage(hsb, SB\_SETTEXT, 0, (LONG)"Texte");

 return 0;

**Code complet :**

**resource.h :**

#define IDM\_QUIT 1

#define IDM\_NEW 2

#define IDM\_ABOUT 3

#define IDM\_PARAM 4

#define IDE\_EDIT1 101

#define IDSB\_SB1 201

**resource.rc :**

#include <windows.h>

#include "resource.h"

1 ICON icone.ico

2 ICON autre.ico

LEMENU MENU

BEGIN

 POPUP "Fichier"

 BEGIN

 MENUITEM "&Nouveau\tCtrl+N", IDM\_NEW

 MENUITEM "&Paramètres...\tCtrl+P", IDM\_PARAM

 MENUITEM SEPARATOR

 MENUITEM "Quitter\tAlt+F4", IDM\_QUIT

 END

 POPUP "Aide"

 BEGIN

 MENUITEM "A propos...", IDM\_ABOUT

 END

END

LesAccel ACCELERATORS

BEGIN

 "N", IDM\_NEW, CONTROL, VIRTKEY

 "P", IDM\_PARAM, CONTROL, VIRTKEY

 VK\_F1, IDM\_ABOUT, VIRTKEY

END

DIALOG2 DIALOG

 60, 60, 182, 70

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "Paramètres"

BEGIN

 DEFPUSHBUTTON "OK", IDOK, 36, 42, 42, 12

 PUSHBUTTON "Cancel", IDCANCEL, 96, 42, 42, 12

 EDITTEXT IDE\_EDIT1, 88, 15, 74, 12

 LTEXT "Titre de la fenêtre", -1, 24, 18, 60, 10

END

DIALOG1 DIALOG

 60, 60, 160, 80

 STYLE WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

 CAPTION "A propos"

BEGIN

 DEFPUSHBUTTON "Ok", IDOK, 56, 50, 42, 12

 ICON 2, -1, 20, 15, 32, 32

 LTEXT "Mon beau programme !", -1, 60, 18, 80, 10

END

**winmain.c :**

#include <windows.h>

#include <commctrl.h>

#include "resource.h"

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

BOOL APIENTRY Dialog1Proc(HWND, UINT, WPARAM, LPARAM);

BOOL APIENTRY Dialog2Proc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 HACCEL haccel;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(hinstance,MAKEINTRESOURCE(2));

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = "LEMENU";

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

 haccel = LoadAccelerators(hinstance, "LesAccel");

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

 {

 if (!TranslateAccelerator(hwnd, haccel, &msg))

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 static HWND hsb;

 static BOOL EditNotChg = TRUE;

 switch (uMsg)

 {

 case WM\_CREATE:

 {

 HFONT hFont;

 hEdit = CreateWindowEx(WS\_EX\_CLIENTEDGE ,"edit", "Texte",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 hFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT);

 SendMessage(hEdit,WM\_SETFONT,(UINT)hFont,TRUE);

 SendMessage(hEdit, EM\_SETMARGINS, EC\_LEFTMARGIN | EC\_RIGHTMARGIN,

 MAKELONG(5, 5));

 InitCommonControls();

 hsb = CreateStatusWindow(WS\_CHILD | WS\_VISIBLE, "Texte", hwnd, -1);

 return 0;

 }

 case WM\_CLOSE:

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer l'application ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 DestroyWindow(hwnd);

 return 0;

 case WM\_COMMAND:

 if(LOWORD(wParam) == IDM\_QUIT) PostMessage(hwnd, WM\_CLOSE,0,0);

 if(LOWORD(wParam) == IDM\_NEW)

 if(EditNotChg ||

 MessageBox(hwnd,"Le texte a été modifié.\r\nEtes vous sûr de \

vouloir fermer votre travail ?"

 ,"Question ?",MB\_YESNO | MB\_ICONQUESTION ) == IDYES)

 {

 SendMessage(hEdit,WM\_SETTEXT,0,(LPARAM)"");

 EditNotChg = TRUE;

 }

 if(LOWORD(wParam) == IDM\_ABOUT)

 DialogBox(hinst, "DIALOG1" , hwnd, (DLGPROC)Dialog1Proc);

 if(LOWORD(wParam) == IDM\_PARAM)

 DialogBoxParam(hinst, "DIALOG2" , hwnd,

 (DLGPROC)Dialog2Proc, (LPARAM)hwnd);

 if(HIWORD(wParam) == EN\_CHANGE) EditNotChg = FALSE;

 return 0;

 case WM\_SIZE:

 {

 RECT sbRect;

 UINT sbheight;

 GetWindowRect(hsb, &sbRect);

 sbheight = sbRect.bottom - sbRect.top;

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam)-sbheight,

 TRUE);

 MoveWindow(hsb, 0, HIWORD(lParam)-sbheight, LOWORD(lParam),

 sbheight, TRUE);

 return 0;

 }

 case WM\_MENUSELECT:

 {

 if (lParam == (LONG)GetMenu(hwnd))

 {

 if(LOWORD(wParam) == 0)

 SendMessage(hsb, SB\_SETTEXT, 0, (LONG)"Menu fichiers");

 if(LOWORD(wParam) == 1)

 SendMessage(hsb, SB\_SETTEXT, 0, (LONG)"Menu Aide");

 }

 else

 {

 if(LOWORD(wParam) == IDM\_QUIT)

 SendMessage(hsb, SB\_SETTEXT, 0,

 (LONG)"Quitter l'application.");

 if(LOWORD(wParam) == IDM\_NEW)

 SendMessage(hsb, SB\_SETTEXT, 0,

 (LONG)"Crée un nouveau fichier.");

 if(LOWORD(wParam) == IDM\_PARAM)

 SendMessage(hsb, SB\_SETTEXT, 0,

 (LONG)"Changer les paramètres.");

 if(LOWORD(wParam) == IDM\_ABOUT)

 SendMessage(hsb, SB\_SETTEXT, 0, (LONG)"A Propos...");

 }

 return 0;

 }

 case WM\_EXITMENULOOP:

 SendMessage(hsb, SB\_SETTEXT, 0, (LONG)"Texte");

 return 0;

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL APIENTRY Dialog1Proc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 return TRUE;

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDCANCEL || LOWORD(wParam) == IDOK)

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

 default:

 return FALSE;

 }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL APIENTRY Dialog2Proc(HWND hDlg,UINT uMsg,WPARAM wParam,LPARAM lParam)

{

 static HWND hParent;

 switch (uMsg)

 {

 case WM\_INITDIALOG:

 {

 int WindowTextLength;

 CHAR \*buffer;

 hParent = (HWND)lParam;

 WindowTextLength = GetWindowTextLength(hParent);

 buffer = (CHAR\*)LocalAlloc(LMEM\_FIXED, WindowTextLength+1);

 GetWindowText(hParent, buffer, WindowTextLength+1);

 SetDlgItemText(hDlg, IDE\_EDIT1, buffer);

 LocalFree(buffer);

 }

 return TRUE;

 case WM\_COMMAND:

 if (LOWORD(wParam) == IDOK )

 {

 CHAR st[128];

 GetDlgItemText(hDlg, IDE\_EDIT1, st, 128);

 SetWindowText(hParent,st);

 EndDialog(hDlg,0);

 return TRUE;

 }

 if (LOWORD(wParam) == IDCANCEL )

 {

 EndDialog(hDlg,0);

 return TRUE;

 }

 default:

 return FALSE;

 }

}

J'ai testé les compilations avec C++ Builder et Dev-C++.

N'oubliez pas d'ajouter la librairie d'importation "libcomctl32.a" avec Dev-C++ (Load object files dans Project options).

**A voir aussi : la barre de progression.**

Le compilateur de ressources de Dev-C++ n'ayant pas accepté la constante PROGRESS\_CLASS, j'ai du la remplacer par son véritable nom de classe de fenêtre.

 CONTROL "", IDE\_PB1, "msctls\_progress32", 0 ,10, 20, 140, 12

N'oubliez pas d'ajouter la librairie d'importation "libcomctl32.a" avec Dev-C++ (Load object files dans Project options).

A voir un autre exemple d'utilisation d'un contrôle de comctrl32.dll : **La barre d'état.**

**13-Donner le style XP à vos applications.**

**Introduction :**

Cet article a pour but de vous montrer comment mettre votre propre fichier manifest dans les ressources de vos applications, afin de leur donner le style graphique de Windows XP, sous Windows XP bien sûr. La copie d'écran si dessous est l'exemple de la barre d'état auquel nous avons donnée le style XP.



**Mise en application :**

Pour donner le style des thèmes XP à vos applications, il faut leur associer un fichier xml appelé manifest", dont voici un exemplaire ci-dessous :

**manifest.xml :**

<?xml version="1.0" encoding="UTF-9" standalone="yes"?>

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

 <assemblyIdentity version="1.0.0.0" processorArchitecture="X86"

 name="CompanyName.ProductName.YourApplication" type="win32"/>

 <description>Description de votre application ici.

 <dependency>

 <dependentAssembly>

 <assemblyIdentity type="win32" name="Microsoft.Windows.Common-Controls"

 version="6.0.0.0" processorArchitecture="X86"

 publicKeyToken="6595b64144ccf1df" language="\*"/>

 </dependentAssembly>

 </dependency>

 </assembly>

Ce fichier comporte des informations sur votre application, notamment son identité dans les attributs de l'élément assemblyIdentity, sa description dans l'élément description que vous pouvez remplir avec les données concernant votre application. L'élément important pour donner le style XP est l'élément assemblyIdentity de l'élément dependentAssembly de l'élément dependency. Il fournit les informations pour trouver l'assembly fournissant les contrôles au style XP. Il ne doit donc pas être modifié Pour information les assemblies sont des dlls spécifiques à la plateforme DotNet. (C'est un avant goùt la gestion des dlls sur les futurs versions de Windows)

La première solution pour utiliser ce fichier manifest est de le joindre à votre application. Il suffit pour cela de lui donner le même nom que votre exécutable avec l'extension "manifest" et de le déposer dans le même dossier. Exemple : si votre exécutable se nomme "WinMain.exe" le fichier manifest se nommera "WinMain.exe.manifest".

Mais sûrement que vous préfererez qu'il soit intégré l'exécutable. La solution est de le mettre dans ses ressources. Si vous avez fait des recherches sur le Net sur le sujet, vous avez sûrement trouvé à télécharger un fichier de ressource compilé .res à lier à votre exécutable. Nous, nous allons aller un tout petit peu plus loin, nous allons le créer nous-même, ce qui vous permettra d'en personnaliser la partie concernant votre application.

Pour cela nous allons d'abord nommer le fichier manifest en "manifest.xml", puis nous l'ajouterons script de ressources :

1 24 manifest.xml

Ici l'identificateur de la ressource est "1" son type est "24" son contenu est celui du fichier manifest.xml (mettre le chemin complet du fichier avec Dev-C++, même s'il ce trouve dans le même dossier).

Pour être un peu plus conventionnel, on aurait pu nommer l'identificateur et le type de la ressource par leurs noms. Normalement ils devraient êtree défini dans le fichier "winuser.h", mais la version de ce fichier jointe avec les compilateurs souvent trop ancienne, fait qu'il ne comporte pas cette définition. Nous pouvons alors par exemple les définir dans un fichier entête nommé "manifest.h".

**manifest.h :**

#ifndef CREATEPROCESS\_MANIFEST\_RESOURCE\_ID

 #define CREATEPROCESS\_MANIFEST\_RESOURCE\_ID 1

#endif

#ifndef RT\_MANIFEST

 #define RT\_MANIFEST 24

#endif

Et dans le script de ressources nous l'appelerons alors par :

#include "manifest.h"

CREATEPROCESS\_MANIFEST\_RESOURCE\_ID RT\_MANIFEST "manifest.xml"

Vous pouvez aussi modifier votre fichier "winuser.h". Voilà nous somme maintenant équipé pour donner du style (XP) à nos nos applications.

**14-Sous classement de fenêtres.**

**Introduction :**

Dans une application, vous traitez les messages des fenêtres ordinaires ou des boîtes de dialogue. Les contrôles leur envoient des messages de notification pour signaler un événement propre à eux mêmes. Malheureusement, tous les événements de ces contrôles n'ont pas de messages de notification qui leurs sont associés. Si vous voulez tout de même traiter un de ces événements, une seule solution, accéder à la procédure de fenêtre du contrôle.

Pour cet exemple, nous reprendrons l'exemple du chapitre 2 "Contrôles et Messages". Nous mettrons le contrôle d'édition en lecture seule, mais pas en lui donnant le style "lecture seule" qui lui donnerait un fond gris. Ce que ne nous ne désirons pas (pour l'exemple).

**Préparation de l'exemple:**

Après avoir créé le contrôle d'édition nous changerons son curseur par celui en forme de flèche. Ceci juste pour montrer visuellement que l'on ne peut pas l'éditer.

 static HWND hEdit;

 switch (uMsg)

 {

 case WM\_CREATE:

 hEdit =CreateWindow("edit", "Texte non modifiable.\r\nVous ne pouvez \

que le lire.\r\nMais il garde la couleur du fond d'un contrôle d'édition actif.",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 SetClassLong(hEdit, GCL\_HCURSOR, (LONG)LoadCursor(NULL, IDC\_ARROW));

Cette action étant réalisée en le changeant dans la classe de fenêtre du contrôle. Ceci sera réalisé à l'aide de la fonction SetClassLong. Elle permet de modifier des éléments d'une classe de fenêtre déjà créée.

**Mise en oeuvre:**

Accéder à la procédure de fenêtre du contrôle ne sera pas suffisant. On ne pourra en obtenir que seulement un pointeur. Il ne pourra donc servir qu'à l'appeler. Nous devons donc en créer une nouvelle.

LRESULT APIENTRY EditProc(HWND, UINT, WPARAM, LPARAM);

Ensuite, on affectera la nouvelle procédure de fenêtre au contrôle d'édition avec la fonction SetWindowLong, qui de plus nous retourne l'adresse de l'ancienne procédure de fenêtre.

#ifdef STRICT

 WNDPROC wpOrigEditProc ;

#else

 FARPROC wpOrigEditProc ;

#endif

/\*...

 ... \*/

 wpOrigEditProc = (PVOID)SetWindowLong(hEdit,

 GWL\_WNDPROC, (LONG)EditProc);

SetWindowLong permet de modifier les propriétés d'une fenêtre comme son style par exemple.

Le pointeur sur l'ancienne procédure de fenêtre nous est nécessaire pour traiter les messages par défaut du contrôle. Les traitements que l'on veux implémenter se font bien sûr dans la nouvelle procédure de fenêtre du contrôle :

LRESULT APIENTRY EditProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 if (uMsg == WM\_SETFOCUS || uMsg == WM\_CONTEXTMENU || uMsg == WM\_CHAR ||

 uMsg == WM\_KEYDOWN)

 return 0;

 return CallWindowProc(wpOrigEditProc, hwnd, uMsg, wParam, lParam);

}

Dans cette nouvelle procédure, on doit comme déjà dit, appeler l'ancienne procédure afin d'effectuer les actions par défaut du contrôle, ne serait-ce que de se dessiner par exemple. C'est le rôle de la fonction CallWindowProc qui reçoit bien sûr le pointeur sur l'ancienne procédure de fenêtre et les paramètres du message.

Dans notre exemple on bloquera les messages, WM\_SETFOCUS (pour qu'il ne puisse pas prendre le focus), WM\_CONTEXTMENU (pour ne pas ouvrir son menu contextuel), WM\_KEYDOWN et WM\_CHAR (pour qu'il ne puisse pas recevoir des commandes ou des caractères depuis le clavier). Ces messages ne seront donc pas envoyés à la procédure de fenêtre d'origine. Ceci afin d'en faire un contrôle d'édition en lecture seule qui n'est pas gris.

**Code complet :**

#include <windows.h>

#ifdef STRICT

 WNDPROC wpOrigEditProc ;

#else

 FARPROC wpOrigEditProc ;

#endif

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

LRESULT APIENTRY EditProc(HWND, UINT, WPARAM, LPARAM);

HINSTANCE hinst;

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = NULL;

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 switch (uMsg)

 {

 case WM\_CREATE:

 hEdit =CreateWindow("edit", "Texte non modifiable.\r\nVous ne pouvez \

que le lire.\r\nMais il garde la couleur du fond d'un contrôle d'édition actif.",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 SetClassLong(hEdit, GCL\_HCURSOR, (LONG)LoadCursor(NULL, IDC\_ARROW));

 wpOrigEditProc = (PVOID)SetWindowLong(hEdit,

 GWL\_WNDPROC, (LONG)EditProc);

 return 0;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 SetWindowLong(hEdit, GWL\_WNDPROC, (LONG)wpOrigEditProc);

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT APIENTRY EditProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 if (uMsg == WM\_SETFOCUS || uMsg == WM\_CONTEXTMENU || uMsg == WM\_CHAR ||

 uMsg == WM\_KEYDOWN)

 return 0;

 return CallWindowProc(wpOrigEditProc, hwnd, uMsg, wParam, lParam);

}

J'ai testé les compilations avec C++ Builder et DevC++.

## 14-Sous classement de fenêtres.

**Introduction :**



Dans une application, vous traitez les messages des fenêtres ordinaires ou des boîtes de dialogue. Les contrôles leur envoient des messages de notification pour signaler un événement propre à eux mêmes. Malheureusement, tous les événements de ces contrôles n'ont pas de messages de notification qui leurs sont associés. Si vous voulez tout de même traiter un de ces événements, une seule solution, accéder à la procédure de fenêtre du contrôle.

Pour cet exemple, nous reprendrons l'exemple du chapitre 2 "Contrôles et Messages". Nous mettrons le contrôle d'édition en lecture seule, mais pas en lui donnant le style "lecture seule" qui lui donnerait un fond gris. Ce que ne nous ne désirons pas (pour l'exemple).

**Préparation de l'exemple:**

Après avoir créé le contrôle d'édition nous changerons son curseur par celui en forme de flèche. Ceci juste pour montrer visuellement que l'on ne peut pas l'éditer.

 static HWND hEdit;

 switch (uMsg)

 {

 case WM\_CREATE:

 hEdit =CreateWindow("edit", "Texte non modifiable.\r\nVous ne pouvez \

que le lire.\r\nMais il garde la couleur du fond d'un contrôle d'édition actif.",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 SetClassLong(hEdit, GCL\_HCURSOR, (LONG)LoadCursor(NULL, IDC\_ARROW));

Cette action étant réalisée en le changeant dans la classe de fenêtre du contrôle. Ceci sera réalisé à l'aide de la fonction SetClassLong. Elle permet de modifier des éléments d'une classe de fenêtre déjà créée.

**Mise en oeuvre:**

Accéder à la procédure de fenêtre du contrôle ne sera pas suffisant. On ne pourra en obtenir que seulement un pointeur. Il ne pourra donc servir qu'à l'appeler. Nous devons donc en créer une nouvelle.

LRESULT APIENTRY EditProc(HWND, UINT, WPARAM, LPARAM);

Ensuite, on affectera la nouvelle procédure de fenêtre au contrôle d'édition avec la fonction SetWindowLong, qui de plus nous retourne l'adresse de l'ancienne procédure de fenêtre.

#ifdef STRICT

 WNDPROC wpOrigEditProc ;

#else

 FARPROC wpOrigEditProc ;

#endif

/\*...

 ... \*/

 wpOrigEditProc = (PVOID)SetWindowLong(hEdit,

 GWL\_WNDPROC, (LONG)EditProc);

SetWindowLong permet de modifier les propriétés d'une fenêtre comme son style par exemple.

Le pointeur sur l'ancienne procédure de fenêtre nous est nécessaire pour traiter les messages par défaut du contrôle. Les traitements que l'on veux implémenter se font bien sûr dans la nouvelle procédure de fenêtre du contrôle :

LRESULT APIENTRY EditProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 if (uMsg == WM\_SETFOCUS || uMsg == WM\_CONTEXTMENU || uMsg == WM\_CHAR ||

 uMsg == WM\_KEYDOWN)

 return 0;

 return CallWindowProc(wpOrigEditProc, hwnd, uMsg, wParam, lParam);

}

Dans cette nouvelle procédure, on doit comme déjà dit, appeler l'ancienne procédure afin d'effectuer les actions par défaut du contrôle, ne serait-ce que de se dessiner par exemple. C'est le rôle de la fonction CallWindowProc qui reçoit bien sûr le pointeur sur l'ancienne procédure de fenêtre et les paramètres du message.

Dans notre exemple on bloquera les messages, WM\_SETFOCUS (pour qu'il ne puisse pas prendre le focus), WM\_CONTEXTMENU (pour ne pas ouvrir son menu contextuel), WM\_KEYDOWN et WM\_CHAR (pour qu'il ne puisse pas recevoir des commandes ou des caractères depuis le clavier). Ces messages ne seront donc pas envoyés à la procédure de fenêtre d'origine. Ceci afin d'en faire un contrôle d'édition en lecture seule qui n'est pas gris.

**Code complet :**

#include <windows.h>

#ifdef STRICT

 WNDPROC wpOrigEditProc ;

#else

 FARPROC wpOrigEditProc ;

#endif

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

LRESULT APIENTRY EditProc(HWND, UINT, WPARAM, LPARAM);

HINSTANCE hinst;

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 hinst = hinstance;

 wc.style = 0 ;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = NULL;

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = NULL;

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Titre", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 300,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 static HWND hEdit;

 switch (uMsg)

 {

 case WM\_CREATE:

 hEdit =CreateWindow("edit", "Texte non modifiable.\r\nVous ne pouvez \

que le lire.\r\nMais il garde la couleur du fond d'un contrôle d'édition actif.",

 WS\_CHILD | WS\_VISIBLE | ES\_MULTILINE | ES\_WANTRETURN | WS\_VSCROLL,

 0, 0, 0, 0, hwnd, NULL, hinst, NULL);

 SetClassLong(hEdit, GCL\_HCURSOR, (LONG)LoadCursor(NULL, IDC\_ARROW));

 wpOrigEditProc = (PVOID)SetWindowLong(hEdit,

 GWL\_WNDPROC, (LONG)EditProc);

 return 0;

 case WM\_SIZE:

 MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);

 return 0;

 case WM\_DESTROY:

 SetWindowLong(hEdit, GWL\_WNDPROC, (LONG)wpOrigEditProc);

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT APIENTRY EditProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 if (uMsg == WM\_SETFOCUS || uMsg == WM\_CONTEXTMENU || uMsg == WM\_CHAR ||

 uMsg == WM\_KEYDOWN)

 return 0;

 return CallWindowProc(wpOrigEditProc, hwnd, uMsg, wParam, lParam);

}

J'ai testé les compilations avec C++ Builder et DevC++.

**15-Afficher des images.**

**Introduction :**

Le traitement d'images dans Windows est un sujet très vaste. Dans ce petit article nous allons voir comment afficher simplement une image dans une fenêtre. Nous avons déjà vu l'affichage d'une image dans une boîte de dialogue à l'aide du contrôle static. Nous allons faire de même, mais dans une fenêtre standard. Nous parlerons bien sûr que d'images de type Bitmap (celle dont les fichiers ont l'extention bmp).

**Image sur contrôle static :**

Nous le créons comme tout contrôle à l'aide de la fonction CreateWindow avec comme nom de classe de fenêtre "STATIC" suivi du nom de la ressource image (le contrôle static ne peut afficher que des images provenant des ressources du programme). Il devra bien sûr avoir le style SS\_BITMAP. Les paramètres de dimensions n'ont aucun effet en ce cas, ils peuvent donc être mit à zéro.

 case WM\_CREATE :

 {

 HWND hstatic = CreateWindow("STATIC", "MyPicture",

 WS\_CHILD | WS\_VISIBLE| SS\_BITMAP,

 10, 10, 0, 0, hwnd, NULL, hinst, NULL);

 return 0;

 }

**Le fichier ressource :**

MyPicture BITMAP "image.bmp"

Cette façon de faire et plus lourde en terme de ressources systèmes, mais à l'avantage que le contrôle peut gérer les messages de notifications. Si l'on a pas besoin de gérer ces messages, il est préférable de dessiner l'image sur un contexe de périférique.

**Dessiner une image avec DrawState :**

Une seconde solution est de dessiner l'image dans la fenêtre en utilisant un contexte de périphérique. Nous chargerons l'image depuis les ressources avec la fonction LoadBitmap et l'affichons avec la fonction DrawState. Cette solution est moins gourmande en ressources système. Le fichier ressources étant le même que précédement.

 case WM\_PAINT:

 {

 HBITMAP hBmp;

 HDC hdc;

 PAINTSTRUCT ps;

 hBmp=LoadBitmap(hinst,"MyPicture");

 hdc = BeginPaint(hwnd, &ps);

 DrawState(hdc,NULL,NULL,(long)hBmp,NULL,10,10,0,0,DST\_BITMAP);

 EndPaint(hwnd, &ps);

 DeleteObject(hBmp);

 return 0;

 }

Sous les plateformes Windows 95 et Windows XP nous pouvons par la même methode afficher une image provenant d'un fichier (Voir note en bas de page pour Windows NT). Pour cela nous chargeons l'image avec la fonction LoadImage.

 case WM\_PAINT :

 {

 HBITMAP hBmp;

 HDC hdc;

 PAINTSTRUCT ps;

 hBmp=LoadImage(NULL,"image.bmp",IMAGE\_BITMAP,0,0,LR\_LOADFROMFILE);

 hdc = BeginPaint(hwnd, &ps);

 DrawState(hdc,NULL,NULL,(long)hBmp,NULL,10,10,0,0,DST\_BITMAP);

 EndPaint(hwnd, &ps);

 DeleteObject(hBmp);

 return 0;

 }

**Dessiner une image avec BitBlt :**

Une troisième solution est de sélectionner l'image sur un contexte de périphérique en mémoire, puis d'en faire la copie sur le contexte de périphérique d'affichage à l'aide de la fonction BitBlt. Cette solution permet de n'afficher qu'une partie du bitmap d'origine.

LONG APIENTRY WndProc(HWND hwnd, UINT uMsg, UINT wParam, LONG lParam)

{

 static HBITMAP hBmp;

 switch ( uMsg ) {

 case WM\_CREATE:

 {

 hBmp=LoadBitmap(hInst,"MyPicture");

 return 0;

 }

 case WM\_PAINT:

 {

 HDC hdc;

 PAINTSTRUCT ps;

 HBITMAP hbmTmp;

 BITMAP bmpi;

 HDC hdcMem;

 hdcMem = CreateCompatibleDC(NULL);

 hbmTmp = SelectObject(hdcMem,hBmp);

 GetObject(hBmp,sizeof(bmpi),&bmpi);

 hdc = BeginPaint(hwnd, &ps);

 BitBlt(hdc,10,10,bmpi.bmWidth,bmpi.bmHeight,hdcMem,0,0,SRCCOPY);

 EndPaint(hwnd, &ps);

 SelectObject(hdcMem,hbmTmp);

 DeleteDC(hdcMem);

 return 0;

 }

 case WM\_DESTROY:

 DeleteObject(hBmp);

 PostQuitMessage(0);

 return 0;

 }

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

}

Dans cet exemple on a chargé l'image dès la création de l'application. Le contexte de périphérique en mémoire est obtenu avec la fonction CreateCompatibleDC. Son unique paramètre est un contexte de périphérique compatible avec celui avec lequel on travaille. Dans l'exemple il est à NULL, ce qui le rend compatible avec l'écran. On sélectionne le bitmap sur ce contexte de périphérique avec la fonction SelectObject. La fonction BitBlt ayant besoin des dimensions de l'image, nous les obtenons à l'aide de la fonction GetObject qui les affectes à une structure BITMAP.

La fonction BitBlt reçoit en son premier paramètre le handle du contexte de périphérique de destination. Les deux paramètres suivants sont la position sur la destination. Les deux suivants sont la largeur et la hauteur de la copie. Le 6ème étant le handle du contexte de périphérique de la source. le 7ème et le 8ème la position de départ (haut, gauche) sur la source. Le dernier étant une constante déterminant le traitement à éffectuer. Dans l'exemple c'est une simple copie de la source vers la destination (SRCCOPY). Voir l'aide sur API Win32 pour les différents traitement pouvant être appliqué.

**Code complet :**

Code complet de la deuxième méthode.

**winmain.c :**

#include <windows.h>

HINSTANCE hinst;

LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 HWND hwnd;

 MSG msg;

 WNDCLASS wc;

 hinst = hinstance;

 wc.style = 0;

 wc.lpfnWndProc = MainWndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hinstance;

 wc.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

 wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

 wc.hbrBackground = (HBRUSH)(1 + COLOR\_BTNFACE);

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MaWinClass";

 if(!RegisterClass(&wc)) return FALSE;

 hwnd = CreateWindow("MaWinClass", "Dessiner", WS\_OVERLAPPEDWINDOW,

 CW\_USEDEFAULT, CW\_USEDEFAULT, 280, 240,

 NULL, NULL, hinstance, NULL);

 if (!hwnd) return FALSE;

 ShowWindow(hwnd, nCmdShow);

 UpdateWindow(hwnd);

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return msg.wParam;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

 switch (uMsg)

 {

 case WM\_PAINT:

 {

 HBITMAP hBmp;

 HDC hdc;

 PAINTSTRUCT ps;

 hBmp=LoadBitmap(hinst,"MyPicture");

 hdc = BeginPaint(hwnd, &ps);

 DrawState(hdc,NULL,NULL,(long)hBmp,NULL,10,10,0,0,DST\_BITMAP);

 EndPaint(hwnd, &ps);

 DeleteObject(hBmp);

 return 0;

 }

 case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

 default:

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

 }

}

**resources.rc :**

#include

MyPicture BITMAP "image.bmp"

J'ai testé les compilations avec C++ Builder et DevC++.

Sous Windows NT LoadImage ne permet pas de charger un bitmap depuis un fichier. Je vous met un exemple qui permet de le faire.

**Windows GDI: Tracés et découpes de contours.**

**Introduction :**



Dans cet article nous allons voir à l'aide d'un exemple comment tracer le contour d'un texte (contour des lettres), faire une découpe de ce contour et dessiner sur la découpe.

Pour arriver à nos fins nous utiliserons les fonctions de Paths et de Clipping du GDI de Windows. Le langage choisi sera le C et nous utiliserons l'API Windows pour que notre exemple soit compatible avec la plus part des compilateurs C et C++. Personnellement je l'ai compilé sur BCB4, BCB6 et Dev C++ .

Dans cet article nous ne reviendrons pas sur l'architecture des applications Windows, pour cela vous pouvez aller consulter ce tutoriel: API Windows : La fenêtre principale. .

**Explication :**

Nous créons donc une application traditionnelle Windows avec sa fonction d'entrée WinMain, nous y créons une fenêtre et sa procédure de fenêtre nommée pour l'occasion WndProc. Voir code complet en bas de ce document.

Nous allons nous concentrer sur le code exécuté à la réception du message WM\_PAINT (Je l'ai mis dans un if au lieu de l'habituel switch case juste pour des raisons de clarté du code.)

Tout d'abord nous récupérons un contexte de périphérique HDC à l'aide de la fonction BeginPaint. Le dessin sur ce contexte sera réellement effectif lors de l'appel de la fonction EndPaint. (BeginPaint et EndPaint ne peuvent s'utiliser uniquement que dans le traitement d'un message WM\_PAINT mais sont plus performantes que de dessiner sur un contexte de périphérique obtenu avec GetDC.)

 PAINTSTRUCT ps;

 HDC hdc;

 hdc = BeginPaint(hwnd, &ps);

 //Fonction de dessin ici

 EndPaint(hwnd, &ps);

Ensuite nous créons une fonte à l'aide de la fonction CreateFontIndirect qui utilise une structure LOGFONT initialisée avec les caractéristiques de la fonte dont nous mettrons la taille suffisamment importante pour la visibilité dans l'exemple. Puis nous sélectionnons cette fonte pour le contexte de périphérique précédemment obtenu.

 LOGFONT lf;

 HFONT hfont;

 HFONT hfontOld;

 char f[32]={"Arial Black"};

 ZeroMemory(&lf, sizeof(LOGFONT));

 strcpy(lf.lfFaceName,f);

 lf.lfHeight = 100;

 hfont = CreateFontIndirect(&lf);

 hfontOld = (HFONT)SelectObject(hdc, hfont);

 //Utilisation ici

 SelectObject(hdc, hfontOld);

 DeleteObject(hfont);

Même opération pour un crayon dont nous initialiserons la taille à 3 pixels. (Ne pas oublier de détruire les objets de dessin quand ils ne sont plus nécessaires.)

 HPEN hpen;

 HPEN hpenOld;

 hpen = CreatePen(PS\_SOLID, 3, 0);

 hpenOld = (HPEN)SelectObject(hdc, hpen);

 //Utilisation ici

 SelectObject(hdc, hpenOld);

 DeleteObject(hpen);

**Dessin du chemin :**

Un chemin se prépare avant d'être dessiné, entre les fonctions BeginPath et EndPath qui reçoivent en paramètre un contexte de périphérique. (Entre ces deux fonctions on peut utiliser toutes les fonctions de dessin du GDI.) Dans cet exemple nous dessinons donc un texte avec la fonction TextOut. C'est la fonction StrokePath qui dessine réellement le tracé préparé entre BeginPath et EndPath.

 BeginPath(hdc);

 TextOut(hdc, 10, 10, "Bonjour", 7);

 EndPath(hdc);

 StrokePath(hdc);

Nous aurions pu aussi utiliser la fonction FillPath qui peut remplir l'intérieur du chemin avec la brosse courante ou StrokeAndFillPath qui elle fait les deux. Quand vous avez appelé l'une de ces fonctions le chemin n'est plus utilisable. Un tracé qui n'est pas fermé peut être fermé avec la fonction CloseFigure.

**Remplissage d'une découpe :**

Pour faire une découpe il faut d'abord tracer son contour comme vu précédemment. Mais au lieu de le dessiner nous le découperons avec la fonction SelectClipPath.

 BeginPath(hdc);

 TextOut(hdc, 10, 10, "Bonjour", 7);

 EndPath(hdc);

 SelectClipPath(hdc, RGN\_XOR);

La découpe ce trouve maintenant en sorte qu'elle ne fait plus partie du contexte de périphérique. Donc tout dessins fait sur ce contexte de périphérique ne sera pas tracé sur la découpe. La fonction SelectClipPath a plusieurs possibilités pour la découpe selon la valeur de son deuxième argument : RGN\_AND, RGN\_COPY, RGN\_DIFF, RGN\_OR, RGN\_XOR.

Pour vous démonter ceci nous allons dessiner des traits horizontaux sur la surface de dessin dans une boucle.

 SIZE sz;

 int i;

 GetTextExtentPoint32(hdc, "Bonjour",7 , &sz);

 for (i = 121; i < (120 + sz.cy); i += 6) {

 MoveToEx(hdc, 11, i, (LPPOINT) NULL);

 LineTo(hdc, (9 + sz.cx), i);

 }

La fonction GetTextExtentPoint32 sert ici à récupérer les dimensions du rectangle qui contient le texte préalablement tracé. Les traits seront donc dessinés seulement dans ce rectangle ceci juste dans un but d'esthétique.

**Code de l'exemple :**

#include <windows.h>

HWND hwndMain ;

LONG APIENTRY WndProc(HWND hwnd, UINT uMsg, UINT wParam, LONG lParam);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

 LPSTR lpCmdLine, int nCmdShow)

{

 MSG msg;

 WNDCLASS wc;

 // Enregistrement de la classe de fenêtre.

 wc.style = 0;

 wc.lpfnWndProc = (WNDPROC) WndProc;

 wc.cbClsExtra = 0;

 wc.cbWndExtra = 0;

 wc.hInstance = hInstance;

 wc.hIcon = LoadIcon((HINSTANCE) NULL, IDI\_APPLICATION);

 wc.hCursor = LoadCursor((HINSTANCE) NULL, IDC\_ARROW);

 wc.hbrBackground = (HBRUSH)GetStockObject(WHITE\_BRUSH);

 wc.lpszMenuName = NULL;

 wc.lpszClassName = "MainWndClass";

 if (!RegisterClass(&wc)) return FALSE;

 // Création de la fenêtre.

 hwndMain = CreateWindow("MainWndClass", "Path and Clip",

 WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, CW\_USEDEFAULT,

 360, 280, NULL, NULL, hInstance, NULL);

 ShowWindow(hwndMain, nCmdShow);

 UpdateWindow(hwndMain);

 // Et la fameuse boucle de messages

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

 {

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 }

 return 0;

}

LONG APIENTRY WndProc(HWND hwnd, UINT uMsg, UINT wParam, LONG lParam)

{

 if (uMsg == WM\_PAINT)

 {

 LOGFONT lf;

 PAINTSTRUCT ps;

 HFONT hfont;

 HFONT hfontOld;

 HPEN hpen;

 HPEN hpenOld;

 SIZE sz;

 int i;

 HDC hdc;

 char f[32]={"Arial Black"};

 hdc = BeginPaint(hwnd, &ps);

 // Initialisation de la structure LOGFONT

 ZeroMemory(&lf, sizeof(LOGFONT));

 strcpy(lf.lfFaceName,f);

 lf.lfHeight = 100;

 // Céation de la fonte

 hfont = CreateFontIndirect(&lf);

 hfontOld = (HFONT)SelectObject(hdc, hfont);

 // Création du crayon

 hpen = CreatePen(PS\_SOLID, 3, 0);

 hpenOld = (HPEN)SelectObject(hdc, hpen);

 // Dessin du chemin

 BeginPath(hdc);

 TextOut(hdc, 10, 10, "Bonjour", 7);

 EndPath(hdc);

 StrokePath(hdc);

 // Dessin dans la découpe du chemin

 BeginPath(hdc);

 TextOut(hdc, 10, 120, "Bonjour", 7);

 EndPath(hdc);

 SelectClipPath(hdc, RGN\_XOR);

 GetTextExtentPoint32(hdc, "Bonjour",7 , &sz);

 for (i = 121; i < (120 + sz.cy); i += 6) {

 MoveToEx(hdc, 11, i, (LPPOINT) NULL);

 LineTo(hdc, (9 + sz.cx), i);

 }

 SelectObject(hdc, hfontOld);

 SelectObject(hdc, hpenOld);

 DeleteObject(hpen);

 DeleteObject(hfont);

 EndPaint(hwnd, &ps);

 return 0;

 }

 if (uMsg == WM\_DESTROY)

 {

 PostQuitMessage(0);

 }

 return DefWindowProc(hwnd, uMsg, wParam, lParam);

}

<http://chgi.developpez.com/windows/>