# Les bases de la programmation sous Windows

par Jessee Edouard (Accueil)

Date de publication : 10 avril 2008

Dernière mise à jour : 17 décembre 2009

Ce tutoriel est le premier d'une série d'articles sur la programmation sous Windows en langage C. Il couvre pratiquement toutes les notions qu'il faut savoir pour bien programmer sous Windows, allant de la création d'un simple "Hello, world !" à l'intégration de ressources, en passant par la gestion des fenêtres, les messages et bien sûr le graphisme.
Commentez cet article :

# I - Création d'une application Windows

## I-A - Hello, world !

Ce programme affiche une boîte de dialogue qui affiche « Hello, world ! ».

hello.c
```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    MessageBox(NULL, "Hello, world !", "Hello", MB_OK);

    return 0;
}
```

La fonction **MessageBox** sert généralement à afficher un petit message à l'utilisateur comme pour l'informer du succès ou de l'échec d'une opération par exemple ou pour demander une action à effectuer lorsque le programme ne peut prendre de décision tout seul.

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

**Paramètres :**

- hWnd : handle d'une fenêtre à laquelle la boîte de dialogue se rapporte ou tout simplement NULL si on ne veut lui associer aucune fenêtre (nous verrons un peu plus loin ce que c'est qu'un handle)
- lpText : le texte à afficher
- lpCaption : titre de la boîte de dialogue
- uType : type de la boîte de dialogue. Ce type peut être une combinaison valide des constantes suivantes :

| Boutons | Icônes | Boutons par défaut |
|---|---|---|
| MB_OK | MB_ICONASTERISK | MB_DEFBUTTON1 |
| MB_YESNO | MB_ICONQUESTION | MB_DEFBUTTON2 |
| MB_YESNOCANCEL | MB_ICONEXCLAMATION | MB_DEFBUTTON3 |
| MB_ABORTRETRYIGNORE | MB_ICONSTOP | |

Par exemple : MB_ABORTRETRYIGNORE | MB_ICONSTOP | MB_DEFBUTTON1

**Valeur de retour :**

- Retourne un entier indiquant la décision prise par l'utilisateur. Peut être IDOK, IDCANCEL, IDYES, IDNO, IDABORT, IDRETRY, IDIGNORE

Nous reviendrons plus tard sur la fonction WinMain.

## I-B - Création et compilation du projet

### Sous Visual C++ 6

1 Lancez **Visual C++**.
2 Créez un **nouveau projet Win32** que vous allez nommer **helloworld** en choisissant **File > New > Project > Win32 Application**. Un dossier nommé helloworld est alors créé. Ce sera votre répertoire par défaut. Deux

fichiers, entre autres, sont également créés : **helloworld.prj** (votre projet) et **helloworld.dsw** (votre espace de travail).
3 Choisissez **An empty project** pour qu'aucun fichier ne soit automatiquement ajouté à notre projet.
4 Ajoutez au projet un nouveau fichier que vous allez nommer **hello.c** avec la commande **Project > Add to Project > New > C++ source file**. Nommez-le bien hello.c car si vous omettez l'extension, VC6 va automatiquement ajouter l'extension .cpp et vous aurez donc un fichier source C++ (hello.cpp) au lieu d'un fichier source C, à moins bien sûr que c'est justement ce que vous cherchiez ...
5 Dans l'explorateur de projet (normalement c'est sur votre gauche), cliquez sur **File View** pour afficher une vue des fichiers qui constituent votre projet. Ouvrez ensuite le dossier **Source Files** puis double cliquez sur **hello.c** pour l'ouvrir dans l'éditeur.
6 Saisissez maintenant le code puis compilez avec la commande **Build > Build helloworld.exe (F7)**. Le fichier **helloworld\Debug\helloworld.exe** est alors créé.
7 Pour tester votre programme sans quitter VC, choisissez **Build > Execute helloworld.exe (Ctrl + F5)**.

### Sous Visual Studio .NET

La procédure est presque les même que sous VC6. Créez un nouveau projet d'application Windows (**Win32 Project**) puis dans l'étape **Applications Settings** de l'assistant, choissez **Windows Application** puis **Empty Project**.
Sous Visual Studio 2005 et plus récents, allez ensuite dans **Project > Properties > Configuration Properties > General** puis positionnez la valeur de **Character Set** à **Not Set**. La raison de ce réglage supplémentaire sera expliquée un peu plus bas.
Compilez puis testez.

### Sous Code::Blocks 1.0

1 Lancez **Code::Blocks**.
2 Créez un nouveau projet d'application Windows (Projet Win32) que vous aller nommer **helloworld** en choisissant **File > New Project > Win32 GUI Application**. Enregistrez-le dans un répertoire de votre choix qui sera alors votre répertoire par défaut. Avant de valider, cochez l'option **Do not create any files** afin qu'aucun fichier ne soit automatiquement ajouté à notre projet. Une fois que vous avez validé, un fichier **helloworld.cbp** (votre projet) sera créé dans le répertoire de votre projet.
3 Créez un nouveau fichier que vous allez nommer **hello.c** avec la commande **File > New File**. Acceptez que ce fichier soit ajouté au projet.
4 Saisissez maintenant le code puis compilez avec la commande **Build > Build (Ctrl + F9)**. Le fichier **helloworld.exe** est alors créé.
5 Pour tester votre programme sans quitter Code::Blocks, choisissez **Build > Run (Ctrl + F10).**

## I-C - La fonction WinMain

Le point d'entrée d'une application Windows est la fonction WinMain.

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

**Paramètres :**

- hInstance : **handle** de l'instance de l'application. Un handle est un numéro qui identifie de manière unique un objet quelconque. Ici, hInstance identifie donc de manière unique notre application.
- hPrevInstance : handle de l'instance précédente. Vaut toujours NULL. Ce paramètre fut utilisé dans les premières versions de Windows et est uniquement gardé pour des raisons de compatibilité.
- lpCmdLine : pointeur les arguments de la ligne de commande de l'application. Pour obtenir la ligne de commande dans son intégralité, utilisez la fonction **GetCommandLine**. Cette fonction ne nécessite aucun argument et retourne un pointeur sur la ligne de commande toute entière.
- nCmdShow : indique comment l'utilisateur désire t-il que la fenêtre principale soit affichée : avec sa taille normale, agrandie ou réduite ? Rien n'oblige cependant le programme à considérer ce choix.

**Valeur de retour :**

- La fonction doit retourner un entier appelé **code d'erreur** de l'application.

**WINAPI** est une macro qui représente la convention d'appel utilisée par les fonctions de l'API Windows. Quant à **windows.h**, c'est bien sûr le fichier d'en-tête principal pour la programmation d'applications pour Windows. En fait, windows.h inclut lui-même d'autres fichiers dont windef.h (définitions des types de base), winnt.h (définitions des types et constantes spécifiques à Windows NT), winbase.h (APIs de base), winuser.h (tout ce qui touche l'interface utilisateur) et wingdi.h (tout ce qui touche la GDI, qui sera étudiée dans les chapitres suivants).

## I-D - La notation hongroise

La **notation hongroise** est la pratique de préfixer chaque identificateur pour coder des informations qui sont bons à savoir à son sujet. Cette pratique est très utilisée par les programmeurs Windows. Par exemple le préfixe p ou lp sert à indiquer qu'il s'agit d'un pointeur, le préfixe h qu'il s'agit d'un handle, etc. Les préfixes couramment utilisés sont :

| Préfixe | Type de la variable |
|---|---|
| h | handle |
| p, lp | pointeur |
| sz, lpsz | chaîne terminée par 0 |
| i, n | int |
| u | UINT (unsigned int) |
| ul | ULONG (unsigned long) |
| dw | DWORD (unsigned long) |
| c | effectif (count) |
| cb | taille en octets (count of bytes) |
| b | BOOL (int) |

## I-E - Unicode

**Unicode** est un jeu de caractères dérivé de l'**ASCII** utilisant **16 bits** pour représenter chaque caractère. Il est utilisé en interne par Windows mais les applications peuvent également utiliser des caractères ou chaînes de caractères ANSI (caractères codés sur 8 bits) pour communiquer avec le système grâce à un jeu de conversions totalement transparent pour l'utilisateur. Par contre lorsqu'on développe des programmes qui doivent dialoguer directement avec le noyau du système (comme les pilotes de périphériques par exemple), on est obligé d'utiliser des chaînes en Unicode uniquement.
En langage C, sous Windows, le type **wchar_t** (wide character) est utilisé pour représenter des caractères Unicode. Ces caractères sont appelés **caractères larges**. char et wchar_t étant des types différents, un caractère "large" (un caractère de type wchar_t) ne s'écrit pas de la même façon qu'un simple caractère. Les caractères larges doivent toujours être précédés d'un L. Par exemple L'A', L'*' et L'1' sont des caractères larges. C'est valable également pour les chaînes de caractères larges, par exemple : L"Bonjour" au lieu de "Bonjour". De même, les fonctions de manipulation des caractères et/ou chaînes de caractères larges ne sont pas les mêmes que celles qui sont dédiées aux caractères et/ou chaînes de caractères simples bien que l'on puisse noter une certaine similitude au niveau des noms et des paramètres requis. Par exemple, pour les caractères et chaînes de caractères larges, on utilise wprintf à la place de printf, wcscpy à la place de strcpy, iswalpha à la place de isalpha, etc. Selon les fonctions que vous utilisez, vous devez inclure wchar.h et/ou wctype.h.
Le programme suivant montre un exemple d'utilisation des caractères larges.

```
#include <stdio.h>
#include <wchar.h>

int main()
{
    wchar_t s[256];
```

```
    wcscpy(s, L"Hello");
    wcscat(s, L", world !");
    wprintf(L"%s\n", s);

    return 0;
}
```

Pour ne pas rendre les programmes dépendants du jeu de caractères utilisé (ANSI ou Unicode), les développeurs de Microsoft ont décidé d'ajouter à la bibliothèque standard du C des macros et types qui permettent de compiler n'importe quel programme avec n'importe quel jeu de caractères sans aucune modification du code source. Ces macros et types sont définis dans le fichier d'en-tête tchar.h (qui est actuellement un fichier d'en-tête "standard" sous Windows).
A la compilation, si la macro **_UNICODE** n'est pas définie, **_TCHAR** est remplacé par char, les "fonctions" par leur version ANSI (_tcscpy -> strcpy, _tprintf -> printf, etc.) et _TEXT("...") par "...". Sinon _TCHAR est remplacé par wchar_t, les "fonctions" par leur version Unicode (_tcscpy -> wcscpy, _tprintf -> wprintf, etc.) et _TEXT("...") par L"...".
Voici un exemple de programme utilisant les _TCHAR :

```
#include <stdio.h>
#include <tchar.h>

int main()
{
    _TCHAR s[256];

    _tcscpy(s, _TEXT("Hello"));
    _tcscat(s, _TEXT(", world !"));
    _tprintf(_TEXT("%s\n"), s);

    return 0;
}
```

Le SDK (le kit de développement de logiciels pour Windows) utilise également cette technique. En effet, MessageBox par exemple en réalité n'est pas le nom d'une fonction mais celui d'une macro qui sera remplacée par MessageBoxA (MessageBox ANSI Characters) si la macro **UNICODE** (à ne pas confondre avec _UNICODE) n'est pas définie et par MessageBoxW (MessageBox Wide Characters) si elle l'est.
A la compilation, si la macro **UNICODE** n'est pas définie, **TCHAR** est remplacé par char, les "fonctions" par leur version ANSI (par exemple : MessageBox -> MessageBoxA) et TEXT("...") par "...". Sinon _TCHAR est remplacé par wchar_t, les "fonctions" par leur version Unicode (par exemple : MessageBox -> MessageBoxW) et TEXT("...") par L"...".
Comme vos applications utiliseront généralement à la fois des fonctions de la bibliothèque du C et des fonctions de l'API Windows, n'oubliez donc pas de définir _UNICODE si vous définissez UNICODE et de ne pas définir la permière si vous ne définissez pas la seconde. Sous Visual Studio .NET, la définition de ces macros peut être contrôlée via les paramètres du projet : **Project > Properties > Configuration Properties > General > Character Set**. Chosissez **Not Set** pour utiliser le jeu de caractères ANSI et **Unicode** pour utiliser le jeu de caractères Unicode. A noter que depuis Visual Studio 2005, c'est Unicode qui est utilisé par défaut.
Voici un programme qui utilise la macro TEXT afin de pouvoir compiler quel que le soit le jeu de caractères utilisé, contrairement à notre premier programme qui ne fonctionnait que pour le jeu de caractères ANSI.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    MessageBox(NULL, TEXT("Hello, world !"), TEXT("Hello"), MB_OK);

    return 0;
}
```

Il est également temps d'expliquer ce que sont que les types LPSTR et LPCTSTR. Et bien, ce sont tout simplement des pointeurs de caractère. En effet on a :

```
typedef char CHAR;
typedef wchar_t WCHAR;
```

```
#ifndef UNICODE
typedef CHAR TCHAR;
#else
typedef WCHAR TCHAR;
#endif

#define CONST const

typedef CHAR * LPSTR;
typedef CONST CHAR * LPCSTR;
typedef WCHAR * LPWSTR;
typedef CONST WCHAR * LPCWSTR;
typedef TCHAR * LPTSTR;
typedef CONST TCHAR * LPCTSTR;
```

Dans la suite, nous allons toujours utiliser le jeu de caractères ANSI afin de rendre nos programmes plus simple à lire mais dans des projets plus sérieux, il est recommandé d'écrire du code "portable", c'est-à-dire utilisant les TCHAR.

# II - Les fenêtres

## II-A - Considérations générales

Les étapes à suivre pour créer une fenêtre sont les suivantes :

- Enregistrer une classe (ou modèle) de fenêtre
- Créer une fenêtre (... à partir d'un modèle existant)
- L'afficher (en effet, la fenêtre est initialement invisible)
- Intercepter tous les messages (souris, clavier, etc.) puis les passer à la **procédure de fenêtre**.

Une procédure de fenêtre est une fonction chargée de traiter les messages reçus.

## II-B - Enregistrer une classe de fenêtre

L'enregistrement d'une nouvelle classe de fenêtre peut se faire avec la fonction **RegisterClass**. Cette fonction nécessite comme paramètre l'adresse d'une structure de type **WNDCLASS**.
Exemple :

Enregistrer une classe de fenêtre

```
WNDCLASS wc;

wc.cbClsExtra     = 0;
wc.cbWndExtra     = 0;
wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
wc.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
wc.hInstance      = <Instance de notre application>;
wc.lpfnWndProc    = <Adresse d-une procédure de fenêtre>;
wc.lpszClassName  = "Classe 1";
wc.lpszMenuName   = NULL;
wc.style          = CS_HREDRAW | CS_VREDRAW;

RegisterClass(&wc);
```

## II-C - Créer une fenêtre, puis l'afficher

Après avoir enregistré une classe de fenêtre, on peut désormais créer une fenêtre.

Créer une fenêtre
```
HWND hWnd;

hWnd = CreateWindow("Classe 1", /* Classe de la fenêtre */
                    "Notre première fenêtre", /* Titre de la fenêtre */
                    WS_OVERLAPPEDWINDOW, /* Style de la fenêtre */
                    100, /* Abscisse du coin supérieur gauche */
                    100, /* Ordonnée du coin supérieur gauche */
                    600, /* Largeur de la fenêtre */
                    300, /* Hauteur de la fenêtre */
                    NULL, /* Fenêtre parent */
                    NULL, /* Menu */
                    <Instance de notre application>,
                    NULL /* Paramètres additionnels */);
```

Le paramètre style peut être une ou une combinaisons de constantes parmi lesquelles :

| Constante | Description |
|---|---|
| WS_POPUP | Fenêtre pop-up (fenêtre "nue") |
| WS_BORDER | Fenêtre comportant une bordure |
| WS_CAPTION | Fenêtre avec barre de titre (inclut le style WS_BORDER) |
| WS_MINIMIZEBOX | Fenêtre avec un bouton Réduire |
| WS_MAXIMIZEBOX | Fenêtre avec un bouton Agrandir |
| WS_SYSMENU | Fenêtre avec menu système (+ bouton Fermer) |
| WS_SIZEBOX (ou WS_THICKFRAME) | Fenêtre redimensionnable |
| WS_OVERLAPPED (ou WS_TILED) | Fenêtre recouvrable |
| **WS_OVERLAPPEDWINDOW** | Combine tous les styles ci-dessus ! |
| WS_CHILD | Fenêtre enfant (fenêtre dans une fenêtre) |
| WS_VISIBLE | Fenêtre initialement visible |

Ensuite on affiche la fenêtre :

Afficher une fenêtre
```
ShowWindow(hWnd, <ShowCmd>);
```

Où <ShowCmd> est un entier censé indiquer la manière dont comment on souhaite afficher la fenêtre. On pourra utiliser par exemple les constantes SW_SHOW, SW_HIDE, SW_MINIMIZE, SZ_MAXIMIZE, etc. D'habitude, on lui passe le paramètre **nCmdShow** de **WinMain** afin que la fenêtre s'affiche tout comme l'utilisateur l'a demandé.

## II-D - Intercepter les messages

Intercepter les messages
```
MSG msg;

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

La fonction **GetMessage** retourne TRUE tant qu'elle n'a pas reçu le message **WM_QUIT**. Une application doit donc envoyer ce message pour quitter la boucle. Une fois qu'on a quitté la boucle, on termine le programme.

## II-E - Ecrire la procédure de fenêtre

Exemple de procédure de fenêtre

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch(message)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        break;

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

    return 0L;
}
```

Le message **WM_DESTROY** est envoyé par Windows lorsque la fenêtre est sur le point d'être détruite (après que l'utilisateur l'a fermée par exemple). A ce moment, nous devons alors poster le message WM_QUIT. C'est ce qu'on a fait avec la fonction **PostQuitMessage**. Le 0 passé en argument de cette fonction sera placé dans le paramètre **wParam** du message. C'est en quelque sorte un entier qui indique la raison pour laquelle on a posté le message. 0 indique une fin normale. Et enfin, il ne faut jamais ignorer un message. Si le message ne nous intéresse pas, laissons à Windows le soin de s'en occuper, en appelant tout simplement **DefWindowProc**.

## II-F - Code complet

Code complet

```
#include <windows.h>

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

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

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

    RegisterClass(&wc);

    hWnd = CreateWindow("Classe 1",
                        "Notre première fenêtre",
                        WS_OVERLAPPEDWINDOW,
                        100, 100, 600, 300,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);

    ShowWindow(hWnd, nCmdShow);

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

Code complet

```
    }

    return (int)msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch(message)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        break;

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

    return 0L;
}
```

## II-G - Attributs d'une fenêtre

### II-G-1 - Définition

Les **attributs** d'une fenêtre son sa classe, son nom, son style, sa position, sa largeur et sa hauteur, sa fenêtre parent, l'instance de l'application à laquelle elle appartient ainsi que d'autres attributs que vous avez vous-même définis.

### II-G-2 - Nom d'une fenêtre

Le **nom d'une fenêtre**, appelé également **texte de la fenêtre**, est une chaîne de caractères qui sert à identifier la fenêtre pour l'utilisateur. Pour une fenêtre (ou feuille), ce nom apparaît dans la barre de titre (si elle en possède évidemment). La fonction **GetWindowText** permet de récupérer le texte d'une fenêtre et **SetWindowText** de le modifier.

```
BOOL SetWindowText(HWND hWnd, LPCTSTR lpString);
int GetWindowText(HWND hWnd, LPTSTR lpString, int nMaxCount);
```

On peut récupérer le handle d'une fenêtre connaissant sa classe et son nom avec la fonction :

```
HWND FindWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName);
```

### II-G-3 - Style

Le **style** d'une fenêtre définit l'apparence et le comportement de la fenêtre. Nous avons déjà vu comment spécifier le style avec la fonction CreateWindow. La fonction **CreateWindowEx** permet de créer une fenêtre avec, en premier argument, un **style étendu**. Il s'utilise quasiment de la même manière que la fonction CreateWindow. Nous ne parlerons de ces styles que lorsque cela est vraiment nécessaire.

### II-G-4 - Position et dimensions

On peut déplacer et/ou redimensionner une fenêtre avec la fonction :

```
BOOL MoveWindow(HWND hWnd, int x, int y, int nWidth, int nHeight, BOOL bRepaint);
```

La fonction **GetWindowRect** permet de récupérer la position et les dimensions d'une fenêtre.

```
BOOL GetWindowRect(HWND hWnd, LPRECT lpRect);
```

Une structure de type **RECT** est composée de 4 champs de type LONG left, top, right et bottom définissant le point supérieur gauche (left, top) et le point inférieur droit (right, bottom) d'un rectangle.
Lorsqu'on crée une fenêtre, les dimensions passées à CreateWindow ou CreateWindowEx sont celles de la fenêtre et non de la zone cliente (l'intérieur de la fenêtre). On peut connaître la position de la zone cliente par rapport à son coin supérieur gauche (qui sera alors le point de coordonnées (0, 0)) à l'aide de la fonction **GetClientRect** qui s'utilise de la même manière que GetWindowRect. Pour ajuster un rectangle aux dimensions (et à la position) qu'il faut pour obtenir un fenêtre qui aura une zone cliente qui correspond au rectangle spécifié au départ, on pourra utiliser la fonction **AdjustWindowRect** (ou AdjustWindowRectEx si on veut spécifier un style étendu).

```
BOOL AdjustWindowRect(LPRECT lpRect, DWORD dwStyle, BOOL bHasMenu);
```

Le code suivant permet d'obtenir une fenêtre avec une zone cliente contenue dans le rectangle (100, 100) □ 320 x 200 par rapport à l'écran.

```
RECT rect;
LONG x = 100, y = 200, width = 320, height = 200;

/* On veut une zone cliente comme ceci : */
rect.left = x;
rect.top = y;
rect.right = x + (width - 1);
rect.bottom = y + (height - 1);

/* On ajuste le rectangle pour qu'il corresponde à celui de la fenêtre  ... */
AdjustWindowRect(&rect, WS_BORDER | WS_CAPTION | WS_SYSMENU, FALSE);

/* ... et le tour est joué ! */
x = rect.left;
y = rect.top;
width = (rect.right - rect.left) + 1;
height = (rect.bottom - rect.top) + 1;

/* Il ne nous reste plus qu'à créer la fenêtre */
CreateWindow("Classe 1",
             "Notre première fenêtre",
             WS_BORDER | WS_CAPTION | WS_SYSMENU,
             x, y, width, height,
             NULL,
             NULL,
             hInstance,
             NULL);
```

## II-H - Le Z Order

Le **Z order** est une **liste** qui maintient la position des fenêtres le long d'un **axe z** orienté vers l'extérieur de l'écran. Ainsi, la fenêtre qui se trouve au sommet du Z order se trouve au premier plan et recouvre toutes les autres fenêtres. La fonction **GetForegroundWindow** retourne le handle de la fenêtre se trouvant au premier plan et **SetForeground** de spécifier une nouvelle fenêtre.

```
HWND GetForegroundWindow(void);
BOOL SetForegroundWindow(HWND hWnd);
```

Le Z order est une **file à priorité**. Les plus prioritaires sont les fenêtres possédant le style étendu **WS_EX_TOPMOST** et apparaissent donc toujours en premier avant n'importe quelle autre fenêtre. Les fenêtres enfants sont groupées avec leur fenêtre parent. Lorsqu'on crée une fenêtre, Windows place cette fenêtre devant toutes les autres fenêtres de même priorité.

# III - Les messages

## III-A - Introduction

Sous Windows, la communication entre une fenêtre et le système se fait par l'intermédiaire de **messages**. Par exemple, lorsqu'une fenêtre est redimensionnée, Windows envoie à celle-ci le message **WM_SIZE** pour l'informer de cet événement. Chaque fenêtre doit avoir une fonction, appelée **procédure de fenêtre**, que Windows appellera automatiquement chaque fois que la fenêtre a reçu un message. Afin qu'un message atteigne effectivement cette procédure de fenêtre, l'application doit explicitement envoyer le message à la procédure en question en appelant la fonction **DispatchMessage**.
Une procédure de fenêtre doit avoir le prototype suivant :

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

Bien entendu, on peut appeler la procédure de fenêtre y, Mocassin ou WindowProc au lieu de WndProc mais en l'appelant **WndProc**, notre code sera plus facile à lire par d'autres programmeurs puisque ce nom est largement utilisé dans la communauté des programmeurs Windows. Parlons maintenant du rôle de chaque paramètre de cette fonction.
Un message est en fait une structure déclarée dans winuser.h comme suit :

```
typedef struct tagMSG {
    HWND    hwnd;
    UINT    message;
    WPARAM  wParam;
    LPARAM  lParam;
    DWORD   time;
    POINT   pt;
} MSG;
```

- **hwnd** identifie le destinataire du message.
- **message** spécifie le message proprement dit (ex : WM_KEYDOWN, WM_SIZE, etc.).
- **wParam** et **lParam** contiennent d'éventuelles informations supplémentaires concernant le message. Leur signification est donc entièrement dépendante du message.
- **time** spécifie quand le message a été envoyé.
- Et **pt** est une structure de type POINT qui contient la position du pointeur de la souris au moment où le message a été envoyé.

Lorsqu'on envoie un message à la procédure de fenêtre d'une fenêtre donnée (avec la fonction DispatchMessage), seuls les 4 premiers paramètres à savoir **hwnd**, **message**, **wParam** et **lParam** sont passés. La procédure de fenêtre étant chargée de traiter tous les messages reçus, son corps ressemble donc la plupart du temps à ceci :

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch(message)
    {
    case XXXXX:
        /* Traitement du message XXXXX */
        break;

    case YYYYY:
        /* Traitement du message YYYYY */
        break;

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

    return 0;
}
```

Une procédure de fenêtre normalement constituée ne doit jamais ignorer un message (sauf si vous êtes bien conscient de ce que vous faites). Si le message ne nécessite aucun traitement particulier (c'est d'ailleurs le cas de la plupart des messages), alors elle doit renvoyer ce message au système pour que ce dernier puisse effectuer un traitement par défaut. C'est le rôle de la fonction **DefWindowProc**.
Certains messages placent plus d'une information utile dans le paramètre wParam ou lParam. Par exemple, dans le cas du message WM_SIZE, wParam contient un entier qui spécifie comment la fenêtre a été redimensionnée et lParam les nouvelles dimensions de la fenêtre : la nouvelle largeur dans la partie basse (ou mot de poids le plus faible) et la nouvelle hauteur dans la partie haute (ou mot de poids le plus fort). Un **mot** représente une valeur sur 16 bits. Les types **WPARAM** et **LPARAM** sont définis de la manière suivante :

```
typedef UINT_PTR WPARAM;
typedef LONG_PTR LPARAM;
```

Avec les types **UINT_PTR** et **LONG_PTR** définis comme suit :

```
#ifdef _WIN64
    typedef unsigned __int64 UINT_PTR;
    typedef __int64 LONG_PTR;
#else
    typedef unsigned int UINT_PTR;
    typedef long LONG_PTR;
#endif
```

Dans les versions 32 bits, WPARAM et LPARAM représentent donc des valeurs 32 bits.
Les macros **LOWORD** et **HIWORD** permettent d'extraire respectivement le mot haut et le mot bas d'une valeur DWORD (qui occupe 32 bits).
Comme nous pouvons le constater, les définitions des types et macros Windows peuvent varier d'une cible à une autre (32 ou 64 bits), mais c'est justement pourquoi ils faut les utiliser car ils permettent de compiler le même code pour des processeurs différents sans aucune modification.
Les paragraphes suivants présentent quelques messages assez courants.

## III-B - Le message WM_CREATE

Envoyé lorsqu'une fenêtre a été créée, avant même que CreateWindow (ou CreateWindowEx) ne retourne. Paramètres :

- lParam : adresse d'une structure de type **CREATESTRUCT** qui contient les paramètres qui ont été passées à CreateWindow (ou CreateWindowEx)

## III-C - Le message WM_CLOSE

Envoyé lorsqu'une fenêtre est sur le point d'être fermée. Ce message est envoyé par exemple lorsque l'utilisateur a cliqué sur le bouton fermer. Si on passe ce message à DefWindowProc, la fonction DestroyWindow est appelée et la fenêtre sera donc détruite.

## III-D - Le message WM_DESTROY

Envoyé lorsqu'une fenêtre est sur le point d'être détruite. Le terme détruire signifie vraiment détruire (c'est-à-dire libérer les ressources utilisées) et pas seulement fermer (cacher).

## III-E - Le message WM_SIZE

Envoyé lorsqu'une fenêtre a été redimensionnée. Paramètres :

- wParam : un entier qui spécifie comment la fenêtre a été redimensionnée. Par exemple :

| Valeur | Signification |
|---|---|
| 0 | Rien à signaler (la fenêtre a été redimensionnée ...) |
| SIZE_MAXIMIZED | La fenêtre a été agrandie |
| SIZE_MINIMIZED | La fenêtre a été réduite |
| SIZE_RESTORED | La fenêtre a été restaurée |

- LOWORD(lParam) : la nouvelle largeur de la zone cliente
- HIWORD(lParam) : la nouvelle hauteur de la zone cliente

## III-F - Les messages provenant du clavier

### III-F-1 - Généralités

Ces messages sont envoyés à une fenêtre uniquement lorsque celle-ci a le focus de l'utilisateur c'est-à-dire quand elle est active.

### III-F-2 - Les messages WM_KEYDOWN et WM_KEYUP

Envoyés respectivement lorsqu'une touche a été enfoncée ou relâchée. Les paramètres sont les suivants :

- wParam : le code (**Virtual Key Code**) de la touche ayant provoqué le message. Par exemple :

| Code | Touche |
|---|---|
| VK_TAB | TAB |
| VK_ESCAPE | ECHAP |
| VK_RETURN | ENTREE |
| VK_BACK | BACKSPACE |
| VK_DELETE | DEL |
| VK_F1 | F1 |
| VK_F2 | F2 |
| VK_UP Fleche | 'Haut' |
| VK_DOWN | Fleche 'Bas' |
| VK_LEFT | Fleche 'Gauche' |
| VK_RIGHT | Fleche 'Droite' |
| 'A' | Touche A |
| 'B' | Touche B |
| 'C' | Touche C |

- lParam : contexte matériel (Repeat Count, Scan Code, etc.). Peu intéressant pour le moment.

Il arrive également assez souvent que l'on veuille connaître l'état d'une ou plusieurs touches auxiliaires (VK_SHIFT, VK_LSHIFT, VK_RSHIFT, VK_CONTROL, etc.) lorsqu'on traite les entrées de l'utilisateur, qu'il s'agisse d'une entrée provenant du clavier ou de la souris. Pour cela, on a la fonction :

```
int GetKeyState(int nVirtKey);
```

Qui retourne un entier indiquant l'état de la touche dont le code a été passé en argument. La signification de la valeur retournée par cette fonction est telle que :

- Le bit de poids le plus faible indique si la touche est maintenue enfoncée.
- Le bit de poids le plus fort (et si vous êtes malin, vous remarquez qu'il s'agit du bit de signe !) indique si la touche est togglée (s'utilise normalement avec des touches telles que VK_CAPSLOCK, VK_NUMLOCK, VK_SCROLL, etc.).

On peut également simuler un événement clavier avec la fonction :

```
void keybd_event(BYTE bVirtKey, BYTE bScanCode, DWORD dwFlags, DWORD dwExtraInfo);
```

Le paramètre dwFlags doit avoir la valeur **0** pour simuler l'enfoncement de la touche et **KEYEVENTF_KEYUP** pour simuler le relâchement de la touche. Il y a également une fonction, **SendInput**, qui est plus générique (peut simuler de nombreuses entrées et non uniquement des événements clavier) et qui permet de spécifier plus d'options.

### III-F-3 - Les messages WM_CHAR et WM_DEADCHAR

Ces messages sont générés à partir des messages **WM_KEYDOWN** et **WM_KEYUP** en réponse à un **TranslateMessage**. Les paramètres sont les mêmes sauf que wParam contient le code (ANSI ou Unicode selon le jeu de caractères que vous utilisez) du caractère au lieu de celui de la touche.

### III-F-4 - Les messages WM_SYSKEYDOWN et WM_SYSKEYUP

Envoyés lorsque la touche ALT (VK_MENU) a été enfoncée au moment de l'événement clavier.

## III-G - Les messages provenant de la souris

Les messages suivants sont envoyés à la fenêtre active lorsque le pointeur de la souris se trouve à l'intérieur de sa zone cliente.

| Message | Evénement |
|---|---|
| WM_LBUTTONDOWN | Bouton gauche enfoncé |
| WM_RBUTTONDOWN | Bouton droit enfoncé |
| WM_MBUTTONDOWN | Bouton du milieu enfoncé |
| WM_LBUTTONUP | Bouton gauche relâché |
| WM_RBUTTONUP | Bouton droit relâché |
| WM_MBUTTONUP | Bouton du milieu relâché |
| WM_LBUTTONDBLCLK | Double-clique avec le bouton gauche |
| WM_RBUTTONDBLCLK | Double-clique avec le bouton droit |
| WM_MBUTTONDBLCLK | Double-clique avec le bouton du milieu |
| WM_MOUSEMOVE | Le curseur s'est déplacé |

Les paramètres sont les suivants :

- wParam : Etat des boutons et/ou de certaines touches clavier. Cette valeur peut être une ou une combinaison des valeurs (indépendantes) suivantes : MK_LBUTTON, MK_RBUTTON, MK_MBUTTON, MK_SHIFT, MK_CONTROL, ...
- LOWORD(lParam) : Abscisse du pointeur de la souris
- HIWORD(lParam) : Ordonnée du pointeur de la souris

Les coordonnées sont calculées par rapport au coin supérieur gauche de la zone cliente.

# IV - Le graphisme

## IV-A - Introduction

### IV-A-1 - L'interface des périphériques graphiques

La **GDI** (**G**raphics **D**evcice **I**nterface) ou Interface des Périphériques Graphiques est une API de niveau moyen (ni trop bas ni trop haut) permettant de dessiner sur n'importe quel périphérique graphique (écran ou imprimante par exemple) de manière standard c'est-à-dire sans avoir à communiquer directement avec le pilote. L'étude un peu plus approfondie de cette API fera l'objet du prochain tutoriel. Pour l'instant, nous nous contenterons de découvrir les bases.

### IV-A-2 - Zone invalide

Une partie de la zone cliente d'une fenêtre est dite **invalide** lorsqu'elle doit être dessinée ou redessinée, ce qui se produit lorsqu'elle vient tout juste d'apparaître alors qu'elle était auparavant cachée (par une autre fenêtre par exemple). Windows informe une fenêtre qu'une partie de sa zone cliente est invalide en lui envoyant le message **WM_PAINT** avec bien sûr en paramètre (c'est-à-dire dans wParam et lParam) les informations supplémentaires, parmi lesquels un pointeur vers une structure appelée **ps** (**P**aint information **S**tructure) contenant, entre autres, les coordonnées du plus petit rectangle contenant la **région invalide**, appelé **rectangle invalide** (en anglais : *invalid rectangle* ou encore *update rectangle*). Si la fenêtre possède les styles CS_HREDRAW et/ou CS_VREDRAW, non seulement le message WM_PAINT sera également envoyé chaque fois que celle-ci est redimensionnée (verticalement ou horizontalement, suivant le ou les styles spécifiés), mais en plus la fenêtre toute entière sera redessinée (raffraîchie). Si une autre partie de la zone cliente devient invalide avant qu'un message WM_PAINT en attente n'ait été traité, Windows calcule une nouvelle région invalide (et donc aussi un nouveau rectangle invalide) et modifie en conséquence les informations contenues dans la ps. Il ne peut donc y avoir au plus qu'un message WM_PAINT (et pas plus) dans la file des messages.
On peut invalider un rectangle avec la fonction **InvalidateRect** que nous verrons plus loin. Si le rectangle invalide est validé avant même qu'on ait eu le temps de traiter le message WM_PAINT, le message WM_PAINT sera supprimé de la queue des messages. On peut à tout moment connaître les coordonnées du rectangle invalide à l'aide de la fonction **GetUpdateRect** puis valider un rectangle avec la fonction **ValidateRect**.
Evidemment, rien ne dit que l'on ne doit dessiner que sur reception du message WM_PAINT. On peut dessiner à tout moment, chaque fois qu'on en a envie cependant, placer le code de dessin dans le traitement de ce message permet de mieux structurer le programme. Chaque fois qu'on veut mettre à jour un dessin, il suffit d'invalider tout simplement la partie que l'on veut redessiner.

### IV-A-3 - Contexte de périphérique

Avant toute chose, il faut savoir qu'une fenêtre est un dessin (dessiné sur l'écran). Pour dessiner dans une fenêtre, il faut donc connaître quelle partie de l'écran est actuellement utilisée par la fenêtre et dessiner à l'intérieur de cette surface. Cette surface est connue sous le nom de **contexte de périphérique** (**Device Context** ou tout simplement **DC**) de la fenêtre. La prochaine étape est donc de récupérer un handle de ce contexte de périphérique. Une fois ce handle obtenu, on peut désormais utiliser ce DC. On a des fonctions pour dessiner des points, des lignes, des rectangles, etc., des fonctions pour afficher du texte, pour créer des polices de caractère, etc. mais aussi des fonctions pour récupérer des informations sur le périphérique, etc.
Sachez également qu'il existe plusieurs types de contexte de périphérique. Il y a par exemple des contextes de périphérique d'une fenêtre, des contextes de périphérique d'une imprimante et même des contextes de périphérique en mémoire ! En général, les fonctions de la GDI ne font pas la distinction entre ces types de DC mais cela n'empêche pas l'existence de certaines fonctions et/ou options utilisables uniquement sur un type de DC particulier.

# IV-A-4 - Dessiner dans la zone cliente d'une fenêtre

## IV-A-4-a - Le classique "Hello, world !"

Voici la méthode généralement utilisée pour dessiner dans la zone cliente d'une fenêtre :
Premièrement, il faut avoir au moins deux variables : une de type **HDC** (handle de DC) et une autre de type **PAINTSTRUCT** (requis par Windows, rien à dicsuter là dessus).

```
HDC hDC;
PAINTSTRUCT ps;
```

Ensuite, dans le traitement du message WM_PAINT, il faut avoir le HDC de la fenêtre et entourer tout le code dessin par BeginPaint()/EndPaint().

```
hDC = BeginPaint(hwnd, &ps);
/* Code de dessin ... */
EndPaint(hwnd, &ps);
```

Par exemple :

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;

    switch(message)
    {
    case WM_PAINT:
        hDC = BeginPaint(hwnd, &ps);
        TextOut(hDC, 0, 0, "Hello, world !", 14);
        EndPaint(hwnd, &ps);
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

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

    return 0L;
}
```

Ce code permet d'afficher « Hello, world ! » à l'intérieur de la fenêtre. Ne vous préoccupez pas avec TextOut pour l'instant, ce n'est pas l'important.

## IV-A-4-b - Les fonctions BeginPaint et EndPaint

Le rôle de la fonction **BeginPaint** est, entre autres, de valider le rectangle invalide (ce qui va donc de supprimer le message WM_PAINT de la queue des messages) et de retourner un handle du contexte de périphérique de l'écran qui nous permettra alors de dessiner dans l'ex région invalide zone cliente de la fenêtre. Plus précisément ce handle nous permettra de dessiner uniquement à l'intérieur du rectangle invalide. Depuis Windows 2000, les applications aussi peuvent contrôler la partie dessinable d'un DC à l'aide de la fonction **SelectClipRgn** qui ne nous intéresse pas pour le moment. Pour pouvoir dessiner en dehors du rectangle invalide pendant le traitement de WM_PAINT, vous pouvez utiliser l'une des techniques suivantes :

- Etendre la zone invalide ! Autrement dit invalider un par un, avant d'appeler BeginPaint bien sûr, tous les rectangles à l'intérieur desquels on veut pouvoir dessiner. On peut invalider un rectangle à l'aide de la fonction **InvalidateRect** qui sera étudiée plus bas.
- Ignorer le HDC retourné par BeginPaint et récupérer un HDC de la fenêtre en utilisant la fonction **GetDC** que nous allons également voir tout à l'heure. Généralement, cela se fait dès le traitement du message WM_CREATE. Le HDC ainsi obtenu est ensuite libéré dans le traitement de WM_DESTROY.

La fonction **EndPaint** quant à elle doit juste être appelée lorsqu'on n'envisage plus de dessiner quoi que ce soit.

## IV-A-4-c - La fonction InvalidateRect

Cette fonction permet d'invalider un rectangle de la zone cliente. N'oubliez pas cependant qu'à chaque nouveau rectangle invalide, il n'existe pas désormais deux ou plusieurs rectangles invalides mais un seul : le plus grand rectangle englobant tous les (ex-)rectangles invalides.

```
BOOL InvalidateRect(HWND hWnd, CONST RECT * lpRect, BOOL bRedrawBackground);
```

Si lpRect vaut NULL, alors c'est toute la zone cliente qui sera invalidée.
Cette fonction permet également de spécifier, via le paramètre **bRedrawBackground**, s'il faut remetrre ou non le fond de la fenêtre à la place du dessin actuel dans le rectangle invalide avant que celui-ci ne soit validé (pou rappel, la validation est faite par BeginPaint et une fois le rectangle invalide validé, le message WM_PAINT est supprimé de la queue des messages). Si ce paramètre vaut TRUE, alors le message **WM_ERASEBKGND** (avec dans wParam la valeur du HDC) est envoyé (par BeginPaint) et une fois le traitement du message effectué, BeginPaint valide le rectangle invalide, sinon ce message ne sera pas envoyé (tout ce qui a déjà été dessiné ne sera donc pas effacé). DefWindowProc traite ce message en peignant le fond du rectangle invalide avec la brosse que vous avez définie pour la fenêtre (avec le membre **hbrBackground** de la structure WNDCLASS(EX)). Si hbrBackground vaut NULL, alors vous devez vous-même traiter ce message. Si vous traitez ce message et que vous avez vous-même peint le fond du rectangle invalide, alors vous devez retourner VRAI pour indiquer à Windows qu'il n'est plus nécessaire de le faire. Sinon vous devez retourner FAUX (et dans ce cas vous avez intérêt à avoir fourni une valeur non nulle à WNDCLASS::hbrBackground ...).

## IV-A-5 - Comment obtenir un HDC

En récupérant le retour de **BeginPaint** bien sûr mais il y a aussi d'autres fonctions à connaître parmi lesquelles **GetDC** qui permet entre autres de récupérer le handle du DC d'une fenêtre spécifiée en argument. Avec ce handle, on pourra dessiner dans toute la zone cliente et non seulement à l'intérieur du rectangle invalide. **GetWindowDC** retourne un handle permettant de dessiner dans toute la fenêtre et non seulement dans la zone cliente.

```
HDC GetDC(HWND hWnd);
HDC GetDC(HWND hWnd);
```

Si hWnd vaut NULL, GetDC et GetWindowDC retournent le handle du DC du périphérique graphique principal, par défaut l'écran.
Généralement, on appelle **GetDC** pendant le traitement du message **WM_CREATE**, **BeginPaint** et **EndPaint** (et le code de dessin) dans le traitement du message **WM_PAINT** et **ReleaseDC** dans le traitement du message **WM_DESTROY**.

```
int ReleaseDC(HWND hWnd, HDC hdc);
```

Donc voici également une autre manière d'afficher Hello world !

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HDC hDC;
    PAINTSTRUCT ps;
```

```
    switch(message)
    {
    case WM_CREATE:
        hDC = GetDC(hwnd);
        break;

    case WM_PAINT:
        BeginPaint(hwnd, &ps);
        TextOut(hDC, 0, 0, "Hello world !", 13);
        EndPaint(hwnd, &ps);
        break;

    case WM_DESTROY:
        ReleaseDC(hwnd, hDC);
        PostQuitMessage(0);
        break;

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

    return 0L;
}
```

# IV-B - Les fonctions de dessin

## IV-B-1 - Tracer des lignes et des points

Ces fonctions utilisent généralement la notion de **position courante**. Par exemple, la fonction **LineTo** trace une ligne à partir de la position courante jusqu'à une certaine position (non incluse) spécifiée en arguments, qui deviendra ensuite la position courante. Pour changer la position courante, on utilisera la fonction **MoveToEx** :

```
BOOL MoveToEx(HDC hdc, int x, int y, LPPOINT lpPoint);
```

Cette fonction accepte en dernier argument l'adresse d'une structure de type **POINT** pour stocker les coordonnées de l'ancienne position courante. On peut évidemment mettre NULL. La fonction **GetCurrentPositionEx** permet d'obtenir la position courante.

```
BOOL GetCurrentPositionEx(HDC hdc, LPPOINT lpPoint);
```

Les fonctions permettant de tracer des lignes et des points sont :

```
COLORREF SetPixel(HDC hdc, int x, int y, COLORREF crColor);
COLORREF GetPixel(HDC hdc, int x, int y);
BOOL LineTo(HDC hdc, int x, int y);
BOOL Polyline(HDC hdc, CONST POINT *lppt, int cPoints);
BOOL PolylineTo(HDC hdc, CONST POINT *lppt, int cCount);
```

Le type **COLORREF** sert évidemment à représenter une couleur RGB. Chacune des composantes R, G et B est stockée sur un octet. La macro **RGB** permet facilement de constituer une couleur à partir de composantes R, G et B.

```
COLORREF RGB(BYTE r, BYTE g, BYTE b);
```

Et on peut extraire les composantes R, G et B d'une couleur avec les macros **GetRValue**, **GetGValue** et **GetBValue**. On a également des fonctions pour tracer des courbes de Bézier. Les **courbes de Bézier** sont définies par 4 points (les points de contrôle) pour la fonction **PolyBezier** et 3 pour la fonction **PolyBezierTo**. Cette dernière utilise la position courante comme premier point de contrôle.

```
BOOL PolyBezier(HDC hdc, CONST POINT *lppt, int cPoints);
BOOL PolyBezierTo(HDC hdc, CONST POINT *lppt, int cCount);
```

## IV-B-2 - Rectangles, ellipses et polygones

Ces fonctions sont :

```
BOOL Rectangle(HDC hdc, int x1, int y1, int x2, int y2);
BOOL Ellipse(HDC hdc, int x1, int y1, int x2, int y2);
BOOL RoundRect(HDC hdc, int x1, int y1, int x2, int y2, int nWidth, int nHeight);
BOOL Polygon(HDC hdc, CONST POINT *lppt, int nCount);
```

La fonction RoundRect trace un rectangle aux coins arrondis en forme d'ellipse (plus précisément de quart d'ellipse ...) dont les dimensions (largeur et hauteur) sont spécifiées via les paramètres nWidth et nHeight.

## IV-B-3 - Les objets graphiques

Les objets permettant de dessiner avec la GDI sont le **crayon** (**pen**), la **brosse** (**brush**), les **polices** de caractères (**font**) et les images **bitmaps**. Pour illustrer la manière de les utiliser, nous allons dessiner un rectangle avec une bordure rouge d'épaisseur 2 pixels et un fond bleu. Notre crayon sera donc rouge et épais de 2 pixels tandis que la brosse sera tout simplement de couleur bleue.

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HDC hDC;
    PAINTSTRUCT ps;

    static HPEN hPen;
    static HBRUSH hBrush;

    switch(message)
    {
    case WM_CREATE:
        hDC = GetDC(hwnd);

        hPen = CreatePen(PS_SOLID, 2, RGB(255, 0, 0));
        SelectObject(hDC, hPen);

        hBrush = CreateSolidBrush(RGB(0, 0, 255));
        SelectObject(hDC, hBrush);

        break;

    case WM_PAINT:
        BeginPaint(hwnd, &ps);
        Rectangle(hDC, 100, 100, 300, 200);
        EndPaint(hwnd, &ps);

        break;

    case WM_DESTROY:
        ReleaseDC(hwnd, hDC);
        DeleteObject(hPen);
        DeleteObject(hBrush);
        PostQuitMessage(0);

        break;

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

    return 0L;
}
```

On commence tout d'abord par créer les objets graphiques dont on a besoin (**CreatePen**, **CreateSolidBrush**). Ensuite, on sélectionne ces objets pour que tous les dessins qu'on effectuera par la suite seront dessinés avec (**SelectObject**). Et bien sûr, lorsqu'on n'a plus besoin des objets qu'on a créé, il faut les détruire (**DeleteObject**).
La fonction :

```
HGDIOBJ SelectObject(HDC hdc, HGDIOBJ hgdiobj);
```

Sélectionne un nouvel objet et retourne le handle de l'ancien objet.
Pour créer un objet de dessin, on a toujours le choix entre deux méthodes :

- la méthode dite **directe**, qui se fait par l'intermédiaire d'une fonction recevant directement tous les paramètres nécessaires à la création de l'objet (ex : CreatePen, CreateSolidBrush, etc.)
- et la méthode dite **indirecte**, qui se fait par l'intermédiaire d'une fonction qui attend en argument l'adresse d'une structure décrivant l'objet que l'on veut créer (CreatePenIndirect, CreateBrushIndirect, etc.)

Ainsi l'exemple précédent aurait pu également s'écrire de la manière suivante :

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HDC hDC;
    PAINTSTRUCT ps;

    static HPEN hPen;
    static HBRUSH hBrush;

    switch(message)
    {
        case WM_CREATE:
        {
            LOGPEN pen;
            LOGBRUSH brush;

            hDC = GetDC(hwnd);

            /* Création et sélection du crayon */
            pen.lopnColor = RGB(255, 0, 0);
            pen.lopnStyle = PS_SOLID;
            pen.lopnWidth.x = 2;

            hPen = CreatePenIndirect(&pen);
            SelectObject(hDC, hPen);

            /* Création et sélection de la brosse */
            brush.lbColor = RGB(0, 0, 255);
            brush.lbHatch = 0;
            brush.lbStyle = BS_SOLID;

            hBrush = CreateBrushIndirect(&brush);
            SelectObject(hDC, hBrush);

            break;
        }

        case WM_PAINT:
        {
            BeginPaint(hwnd, &ps);
            Rectangle(hDC, 100, 100, 300, 200);
            EndPaint(hwnd, &ps);

            break;
        }

        case WM_DESTROY:
        {
            ReleaseDC(hwnd, hDC);
            DeleteObject(hPen);
            DeleteObject(hBrush);
```

```
            PostQuitMessage(0);

            break;
        }

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

    return 0L;
}
```

Contrairement à ce qu'on aurait pu s'attendre, le membre lopnWidth qui spécifie l'épaisseur du crayon n'est pas un entier mais une structure de type POINT. Cependant, le membre y de cette structure n'est pas utilisé.
Pour les crayons, on a les styles PS_NULL, PS_SOLID, PS_DASH, PS_DOT, PS_DASHDOT et PS_DASHDOTDOT. Seul **PS_SOLID** accepte une épaisseur autre que 1. Lorsque le style est différent de PS_SOLID, et uniquement dans ce cas, vous pouvez utiliser la fonction **SetBkColor** pour choisir la couleur de fond (fond uni en dessous du coup de crayon).
Pour les brosses, les plus utilisés sont **BS_SOLID** et BS_HATCHED. Si le style BS_HATCHED est spécifié, alors on peut utiliser les constantes suivantes pour le paramètre lbHatch : HS_HORIZONTAL, HS_VERTICAL, HS_CROSS, HS_DIAGCROSS, HS_BDIAGONAL et HS_FDIAGONAL. Si vous voulez avoir à la fois une couleur de fond (fond uni en dessous du coup de brosse) et des hachures, et uniquement dans ce cas, alors vous pouvez utiliser la fonction **SetBkColor** pour choisir la couleur de fond.
La fonction **SetBkMode** permet de contrôler la couleur de fond des objets dessinés. Si le mode est **OPAQUE**, le fond sera coloré avec la couleur de fond courante (que l'on peut récupérer à l'aide de **GetBkColor**). Si le mode est **TRANSPARENT**, le fond est laissé tel quel (c'est-à-dire le même que celui de la fenêtre).

```
COLORREF SetBkColor(HDC hdc, COLORREF crColor);
COLORREF SetBkMode(HDC hdc, int iBkMode);
```

Et enfin, sachez que vous pouvez également utiliser des objets prédéfinis (appelés objets en stock) comme la brosse blanche (WHITE_BRUSH), noire (BLACK_BRUSH) ou grise (GRAY_BRUSH), etc. ou encore le crayon noir (BLACK_PEN) ou blanc (WHITE_PEN), etc. à l'aide de la fonction **GetStockObject**.

```
HGDIOBJ GetStockObject(int fnObject);
```

## IV-B-4 - Afficher du texte

La fonction la plus simple qui permette d'afficher du texte à l'intérieur d'une fenêtre est :

```
BOOL TextOut(HDC hdc, int x, int y, LPCTSTR lpString, int cbString);
```

Le texte n'a donc pas à être une chaîne terminée par zéro puisque le paramètre cbString indique le nombre de caractères à afficher.
La fonction **DrawText** offre plus de contrôle

```
int DrawText(HDC hdc, LPCTSTR lpString, int nCount, LPRECT lpRect, UINT uFormat);
```

Le texte sera affiché à l'intérieur du rectangle pointé par lpRect. Les valeurs les plus utilisées dans uFormat sont DT_LEFT, DT_RIGHT, DT_CENTER, DT_BOTTOM, DT_TOP, DT_VCENTER et DT_SINGLELINE. On peut bien sûr utiliser un format combiné tel que DT_SINGLELINE | DT_CENTER | DT_VCENTER.
**DrawTextEx** permet de spécifier encore plus d'options. Et enfin, on a les fonctions **SetTextColor**, **SetBkColor** et **SetBkMode** pour modifier la couleur du texte, modifier la couleur de fond et régler la transparence du fond.

```
COLORREF SetTextColor(HDC hdc, COLORREF crColor);
```

# V - Les ressources

## V-A - Introduction

Le développement d'applications est un art qui fait intervenir plusieurs techniques, moyens et objets. Il ne suffit pas tout simplement d'avoir un compilateur et les librairies nécessaires, il s'agit également d'intégrer des images, du son, de la vidéo, etc. Enfin, cela dépend du type d'applications qu'on veut faire et de la manière dont on veut s'y prendre mais bref ...
Sous Windows, ces éléments « externes » qui viennent s'ajouter au programme s'appellent les ressources (cette définition n'est pas complète). Vous pouvez évidemment les séparer du programme (c'est-à-dire en tant que fichiers à part) mais vous pouvez également les intégrer à l'intérieur même de votre exécutable ! Et ce n'est pas tout ! Il n'y a pas que des fichiers qu'on peut mettre en ressources, on peut aussi y placer des chaînes ou des informations de version (version du fichier, copyright, nom de l'auteur, etc.) par exemple. Nous n'allons pas tout dire des ressources dans ce tutoriel car il y a encore plein de choses que nous n'avons pas encore vu (de même nous n'allons pas encore voir comment afficher une image ou lire un mp3 ...), mais nous allons au moins voir comment créer un fichier de ressources, le lier à l'exécutable et charger une ressource pendant l'exécution.

## V-B - Exemple de fichier de ressources

Un **fichier de ressources** est un bête fichier texte qui (grossièrement) contient la liste des objets à embarquer. L'extension **.rc** est utilisée pour indiquer qu'il s'agit d'un fichier de ressources mais évidemment on peut utiliser n'importe quelle extension. Le listing contenu dans un fichier de ressources s'appelle un **script de ressources**. La syntaxe est simplissime, en voici un exemple :

Fichier : mes_ressources.rc
```
Logo ICON "mylogo.ico"
User ICON "user.ico"
```

Ajoutez donc ce fichier à votre projet. Lorsque vous compilerez ce dernier, tous les fichiers sources ainsi que le fichier de ressources seront compilés puis liés pour produire l'exécutable (avec une icône (mylogo.ico) cette fois-ci !). A noter que le **compilateur de ressources** est un programme à part (**RC**) et non le compilateur C lui-même. Il génère un fichier **.res** qui contient enfin effectivement les données à intégrer dans l'exécutable contrairement au fichier .rc qui n'est qu'un simple fichier texte.
Dans l'exemple donné plus haut, **Logo** sera donc le nom qui nous servira à identifier la ressource **mylogo.ico** dans le programme. De même, le nom **User** servira à identifier la ressource **user.ico**. Le mot-clé **ICON** sert à indiquer qu'il s'agit d'un fichier d'icône (le répertoire Common \ Graphics \ Icons de Visual Studio en contient plein au cas où ...).
Pour charger une icône, on utilisera la fonction **LoadIcon**. Cette fonction attend en premier paramètre le handle du module qui contient l'icône à charger, en second son nom, puis retourne un handle de l'icône ainsi chargée. Par exemple :

```
HICON hLogo = LoadIcon(hInstance, "Logo");
```

Où hInstance est évidemment le handle de l'instance de notre application.
On peut maintenant utiliser hLogo partout où le handle d'une icône est requis, comme dans le membre hIcon de la structure WNDCLASS par exemple. En fait, la fonction LoadIcon ne charge pas vraiment une icône, mais retourne tout simplement le handle d'une icône appartenant à un module qui lui a déjà été chargé (donc avec ses ressources également !). En conséquence, il ne faut surtout pas chercher à libérer la mémoire utilisée par cette icône puisqu'elle sera automatiquement déchargée de la mémoire en même temps que le programme lorsque celui-ci se sera terminé.
On peut également utiliser un nombre à la place d'un nom pour identifier une ressource. La macro **MAKEINTRESOURCE** permet de convertir un nombre en chaîne de caractères que l'on pourra alors passer à une fonction comme LoadIcon par exemple.
Le principe est le même pour les autres types de fichiers. Pour les curseurs et les bitmaps par exemple, on utilisera respectivement CURSOR, HCURSOR, **LoadCusror** et BITMAP, HBITMAP et **LoadBitmap**.

## V-C - Les icônes

Vous ne le savez peut-être pas encore mais une fenêtre utilise en fait deux icônes : une « petite » et une « grande ». **Petite icône** est le nom donné à celle qui sera utilisée dans la barre de titre de ladite fenêtre et **grande icône** est celui de celle qui apparaîtra dans la boîte de basculement rapide entre fenêtres (celle qui apparaît quand on appuie sur ALT + TAB). Lorsqu'on ne spécifie pas d'icône, Windows utilisera alors l'icône par défaut (IDI_APPLICATION). La structure WNDCLASS permet de spécifier une icône qui sera à la fois utilisée comme petite et grande icône. Utilisez la structure **WNDCLASSEX** pour spécifier individuellement les petite et grande icônes. Elle ressemble à la structure WNDCLASS mais possède deux champs supplémentaires : **cbSize** (mettre sizeof (WNDCLASSEX)) et **hIconSm** (permet de spécifier une petite icône tandis que hIcon sera utilisée pour comme grande icône).
On peut également changer dynamiquement l'icône d'une fenêtre en lui envoyant le message **WM_SETICON**. On mettra dans wParam **ICON_SMALL** ou **ICON_LARGE** suivant l'icône que l'on veut modifier et le handle de la nouvelle icône dans lParam.

## V-D - Charger une image

Nous avons vu que les fonctions LoadIcon, LoadCursor et LoadBitmap permettent de récupérer respectivement le handle d'une icône, d'un curseur ou d'une image bitmap. La fonction **LoadImage** est beaucoup plus souple et générique (voir MSDN). Elle permet de charger aussi bien une icône qu'un curseur ou une image bitmap et ce depuis une ressource ou bien depuis un fichier. Si l'image a été chargée depuis un fichier, alors il faudra libérer la mémoire lorsqu'on n'en aura plus besoin : **DestroyIcon** pour une icône, **DestroyCursor** pour un curseur et **DeleteObject** pour un bitmap.

## V-E - Mettre des données brutes en ressources

Le mot-clé **RCDATA** permet d'embarquer des données brutes en ressource. Par exemple :

```
MyData RCDATA
{
    "Bonjour.\0"
}
```

Qu'on aurait également pus écrire :

```
MyData RCDATA
{
    "B", "o", "n", "j", "o", "u", "r", ".", "\0"
}
```

Ou encore plus tordu ... Attention ! **RC n'ajoute pas automatiquement le caractère de fin de chaîne** c'est pourquoi il faut explicitement l'ajouter si on veut obtenir une chaîne terminée par zéro.
On peut également utiliser des entiers (éventuellement séparés par des virgules). Le type par défaut des constantes entières est WORD (unsigned short). Pour utiliser des entiers long (DWORD), il suffit d'ajouter le suffixe L. De même, on peut utiliser des chaînes UNICODE en préfixant la chaîne par L.
Pour accéder à la ressource, il faut suivre les étapes suivantes :

- Localiser la ressource (**FindResource**)
- Charger la ressource (**LoadResource**)
- Obtenir un pointeur sur la mémoire utilisée par la resource (**LockResource**)

Pour obtenir la taille de la ressource, on a la fonction **SizeofResource**. Voici un exemple d'utilisation de la ressource MyData :

Fichier : exemple.c

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

Fichier : exemple.c

```
int main()
{
    HINSTANCE hInstance;
    HRSRC hrcMyData;
    HGLOBAL hMyData;
    LPVOID pMyData;
    DWORD dwSizeofMyData;

    hInstance = GetModuleHandle(NULL);
    hrcMyData = FindResource(hInstance, "MyData", RT_RCDATA);
    hMyData = LoadResource(hInstance, hrcMyData);
    pMyData = LockResource(hMyData);
    dwSizeofMyData = SizeofResource(hInstance, hrcMyData);

    printf("%s\n", pMyData);

    return 0;
}
```

Dans la réalité on ajoutera évidemment des tests d'erreur. **GetModuleHandle**(NULL) retourne toujours le handle (HINSTANCE) de l'application courante. Dans une application GUI cette valeur nous est déjà fournie par le paramètre **hInstance** de la fonction WinMain. On notera également que certes, SizeofResource ne nous a été d'aucune utilité dans cet exemple mais au moins ça nous a permis de voir comment l'utiliser. D'ailleurs ici, on aurait pu simplement la calculer avec strlen(pMyData) + 1.

## V-F - Les ressources personnalisées

On peut évidemment définir ses propres types de ressource. Cela permet d'embarquer n'importe quoi (absolument n'importe quoi !) dans notre exécutable. Par exemple :

```
#define GENERIC_RESOURCE 0x100
```

Il est impératif d'utiliser une valeur supérieure ou égale à 0x100 car les valeurs comprises entre 0x00 et 0xFF sont déjà réservées. Voici un exemple :

```
MyFile GENERIC_RESOURCE "myfile.dat"

MyData GENERIC_RESOURCE
{
    L"Bonjour", 0x00
}
```

Pour localiser MyFile pendant l'exécution, il suffit de faire :

```
FindResource(hInstance, "MyFile", MAKEINTRESOURCE(GENERIC_RESOURCE));
```