دليل البرمجة بلغة النظام وبمحرر

تأليف: بروك مايلز

ترجمة: توانتي عمر

Edited by: Brook Miles

Translated by: Touati Omar

Third Edition

الطبعة الثالثة

الفهرس

مقدمة

بسم الله الرحمن الرحيم، والصلاة والسلام على خاتم الأنبياء والمرسلين .

أخي القارئ والباحث عن المعرفة بلغة القرآن، إليك هذه الصفحات المتواضعة في ثالث كتاب أقوم بترجمته، أملا أن أسعى ولو بقدر بسيط على توفير الزاد العلمي وبحرف الضاد، في وقت تكاثرت فيه المعلومة بشكل رهيب، وتضخمت القواميس اللاتينية بآلاف المصطلحات الجديدة، وفي حين بقيت لغتنا الحبيبة تراود مكانها، نظرا لعدم استعدادنا كتاب وقراء، خبراء وهواة لملاحقة الركب، مكتفين بمبادرات فردية تنقصها عملية التنسيق، ومرتكزة في أغلبها على الجوانب الأدبية .

جئت بترجمة لهذا الكتاب بمبادرة فردية لعدة أسباب، أهمها عدم توفر الكتب والكتيبات الإلكترونية إلا بمقابل مادي غير متاح لأغلب المتعلمين، ومن جانب آخر لتوفير نوع معين من الكتب للراغبين في ولوج عالم البرمجة المعمقة .

هذا الكتاب، هو في حقيقة الأصل مجموعة من صفحات الإنترنت التي ألفها أحد خبراء عالم البرمجة تحت نظام الويندوز، وأقصد به الأستاذ "بروك ميلز" Brook Miles، وأملا في أن يستفيد المبرمج العربي المبتدئ أو المحترف من ذلك بأسلوب عربي يعيه جيدا، ارتأيت أن أترجم ما قدرت على ترجمته تحت الموافقة الرسمية لصاحب هذه الدروس، بأسلوب فردي كالعادة، تنقصه الحنكة الأدبية والكفاية العلمية، إلا أنه حسب ظني يفي بالغرض ولو بنسبة قليلة. ولقد اعتمدت على أسلوب الترجمة الحرفية، فتراني أحيانا أتحدث بضمير المتكلم، وهذا في حقيقة الأمر يخص مؤلف الكتاب الحقيقي، فأنا ليس لي دور إلا في الترجمة، فلا ألواد كانت من برمجتي، ولا تصاميم كانت من تخطيطي .

إذا كنت من المهتمين بهذا النوع من الدروس، فلابد وأنك ستصطدم بمصطلحات عربية جديدة، قد تختلف عما ألفته، ورغم أني كنت حريصا على أن أسير على ذات المنهج، إلا أن الكثير من المصطلحات لم أجد لها بديلا، ولقد حاولت الاجتهاد بأقصى ما أقدر، لهذا أريد منك أخي المبرمج أن تبدي رأيك في هذا الكتاب، أريد أن أعرف جوانب قوته ونقاط ضعفه (أقصد الترجمة)، وكاشفا لأي خطأ مطبعي أو برمجي، أو سوء في اختيار البديل العربي للمصطلحات اللاتينية، على أن أقوم بتصحيح الأخطاء وتحسين الأفكار بالاعتماد على نصائحكم، فلا تبخلوا بها علينا .

إذا ما أشكل عليك الأمر، فيمكنك الذهاب مباشرة إلى الموقع الرسمي الذي استقيت منه هذه الدروس: http://www.winprog.org/tutorial/index.html، أما لو كنت تريد إدراج إضافاتك في هذا الكتاب، فإنه بإمكانك مراسلتي حتى نقوم سويا بإثراء هذا الكتاب ودائما بالمجان .

إذا ما رغبت عزيزي القارئ في طبع الكتاب أو إهدائه، فلك ذلك، لكن من دون مقابل مادي، غير ذلك فأرجو مشورتي في الأمر .

هذا الكتاب مقسم إلى خمسة أجزاء كما سترها في الفهرس، ولقد قمت بترجمتها مرحليا، وهاهو الآن بين يديك عزيزي القارئ، هدية من أخ مسلم، لا يرجو من وراء ذلك إلا النفع لاخوته العرب والمسلمين بحروف عربية، راجيا أن لا تبخلوا عليه بالدعاء بالخير والصلاح والهداية والأمن والإيمان .

كيف تقرأ هذا الكتاب؟

عزيزي القارئ، من أجل الإلمام التام بمضمون هذا الكتاب، فإنني قمت بإدراج رموز من هذا الشكل ↺ أو هذا ↩ (لأن بعض الصور لا يدعمها برنامج Acrobat Reader) في أماكن مختلفة، بالنقر عليها تستطيع الرجوع إلى الموضوع الأساسي أو الشرح الكامل للمعلومات التي أنت بصدد مطالعتها، هذا الأسلوب يمكنك من مراجعة مختلف خصوصيات الوين 32 .

الآن وبعد مطالعتك لفقرة المراجعة، تريد أن تعود وتكمل نص الكتاب . حسنا، ليسهل عليك ذلك، قم فقط بالنقر على هذا الزر  والخاص ببرنامج Acrobat Reader، لينقلك للموقع السابق .

يمكنك تحميل أكود الأمثلة الواردة في كل أجزاء هذا الدليل بالضغط هنا.

شكر خاص:

أشكر كل من ساهم بطريقة مباشرة أو غير مباشرة في تحسين هذا الكتاب، وفي تصحيح أخطائه أو في إضافة ملاحظات وتعديلات عليه، وأخص بالذكر: الأخ WebDev، الأخت تهاني السبيت، الأخت هناء والأخ طلال السبيعي، وكل الاخوة محبي وزوار الموسوعة العربية للكمبيوتر والإنترنت.

تم بحمد الله وعونه إثراء وتنقيح الطبعة الثالثة من كتاب: دليل البرمجة تحت نظام الوينـدوز في 16 جانفي 2004 الموافق لـ 23 ذو القعدة 1424 .

أساسيات

I - مدخل عام:

هذه الدروس التي ستقضي معها ساعات طويلة، تقدم لك أساسيات البرمجة تحت نظام الـ WIN32 API . اللغة المستعملة هي السي، لكن أغلب مصرفات السي ++ تسمح بتصريف هذا النوع من الملفات. كذلك، أغلب المعلومات تطبق على بقية اللغات التي تسمح بالتعامل مع الـ API ، بما فيها الجافا، الأسمبلي والفيجول بيزك. سوف لن أقوم بذكر الأكواد المناسبة لهذه اللغات، لكن بعض الناس استعملوا هذا الدليل مع بقية اللغات، ولكنهم حققوا القليل من النجاح.

لن يسعى هذا الدليل لتقديم أوليات لغة السي، ولن يتطرق لكل الخطوات الواجب اتباعها لتنفيذ وتصحيح أخطاء الأكواد، سواء كان على البورلاند سي++ أو الفيجول سي++.

إذا كنت لا تعرف معنى "الماكرو" ولا "typedef"، ولا تعرف أسلوب عمل التعليمة switch، فيستحسن لك أن ترجع وتطالع بعض الكتب المختصة في ذلك قبل التطرق لصفحات هذه الدروس.

ملاحظات هامة:

أحيانا وأثناء دراسة هذا الدليل، أسعى لذكر بعض النقاط الهامة والتي يجب مطالعتها، لأنه كثيرا ما اختلطت المفاهيم لدى البعض، وإذا لم تطلع عليها، فلا محالة ستقع في نفس الفخ.

هذه النقاط هي:

- المصادر المعروضة في الأمثلة من نوع zip ليست مثالية:
لم أقم بإدراج كامل الكود ضمن النص، لكن فقط النقاط التي أنا بصدد التطرق إليها.
فإن كنت راغبا في معرفة أسلوب عمل هذا الجزء من الكود، فأنت مدعو لتحميل وتجريب الكود المرفق في ملفات الـ zip.
- طالع كل شيء:
إذا ما خطر على بالك سؤال أثناء مطالعتك لموضوع ما، فخذ قليلا من التأني لأنك كثيرا ما ستصادف الجواب أثناء مطالعتك لبقية الموضوع أو ابحث مباشرة في بقية صفحات هذه الدروس قبل طرح الأسئلة.

II - أبسط برنامج win32 :

إذا ما كنت مبتدئا للغاية، فدعنا نقوم بتصريف أبسط برنامج من برامج ويندوز. قم بنسخ الكود الآتي في محرر الشيفرة (source editor) الخاص بالمصرف (compiler)، وإذا ما سارت الأمور على ما يرام، فإنك ستحصل على أصغر برنامج في عالم الوين32.

لا تنسى بأن الكود هنا هو بالسي وليس بالسي++، وإن كان المصرف الخاص باللغة الثانية يقبل كلا النوعين، إلا أن هذا لا يمنع من وضع النقاط على الحروف، والسير في المسار السليم. لهذا فإننا سنسمي ملفنا test.c، والذي يحمل في طياته الكود الآتي:

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MessageBox(NULL, "Goodbye, cruel world!", "Note", MB_OK);
    return 0;
}
```

إذا لم يعمل هذا المثال، فأول شيء أنت مدعو إليه هو قراءة الأخطاء التي ستصادفك أثناء تنفيذ البرنامج، فإذا لم تستوعبها، فقم بمطالعة الكتيبات المرفقة مع المصرف.

تأكد ضمن برنامج الفيجول سي++ مثلا، من أنك قمت بتحديد WIN32 GUI وليس تطبيق من نوع الكونسول (Console).

أي نوع آخر من العراقيل، فإنه ليس بمقدورنا في هذا الدليل ذكر كل شيء، لأن الأخطاء وأسلوب معالجتها تختلف من مصرف لآخر.

يمكن في هذا المثال أن يقوم المصرف بتوليد بعض التحذيرات (warnings) بخصوص عدم استعمال البارامترات التي يتم تزويد الدالة WinMain بها عادة، فهذا وارد.

الآن، ماذا تعني هذه البتات البسيطة من الكود؟

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

WinMain(): هي الدالة المكافئة للدالة main() المستخدمة في الدوس أو اليونيكس. من هنا يبدأ البرنامج عملية التنفيذ. أما البارامترات فهي:

HINSTANCE hInstance	مقبض (handle) ملف البرنامج التنفيذي (ملف exe. في الذاكرة)
HINSTANCE hPrevInstance	دائما NULL بالنسبة لتطبيقات الوين32
LPSTR lpCmdLine	وهو سطر الوسيطات (arguments)، بحيث يكون من نوع سلسلة حروف (string)، من دون اسم البرنامج.
int nCmdShow	هذا الوسيط هو قيمة صحيحة (integer) والتي يمكن إرسالها إلى ShowWindow(). سنتطرق إليها لاحقا.

HInstance تستعمل لبعض المهام، كتحميل الموارد أو أي شيء آخر، بحيث يتم ذلك على مستوى الوحدة النمطية، والوحدة النمطية هي إما ملف exe. أو dll. تم تحميلة ضمن البرنامج، وفي دليلنا التعليمي هذا، سيتم التعامل مع نوع واحد من الوحدات النمطية، ألا وهو الـ exe.

HPrevInstance تستعمل لتكون مقبضا للمثيل (instance) السابق للبرنامج الجاري تنفيذه، إذا ما تواجد ذلك (غالبا في win16)، في الوين 32، يمكنك إهمال ذلك.

أسلوب الإستدعاء:

WINAPI تستخدم أسلوب للاستدعاء يعرف بـ stdcall_، إذا لم تكن تعلم ماهيتها، فلا تقلق بشأنها، فقط تذكر بأن وجودها ضروري، لأنها خارج إطار دروسنا في هذا الدليل.

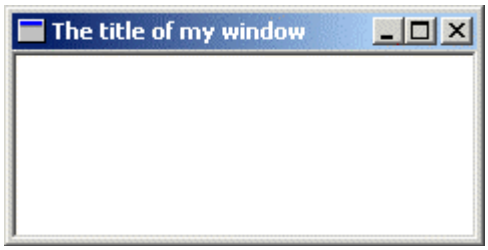
أنواع البيانات في الوين 32:

ستلاحظ أن بعض الأنواع لها تعريف مميز في بيئة الويندوز، UINT تعني (Unsigned INT) وهي لتعريف العدد الصحيح الغير مؤشر، كما أن LPSTR تعني (Long Pointer to STRing)، وهي مكافئة لـ char*، وهلم جرا، لهذا فإن استعمال مؤشر على حروف بدلا من مؤشر طويل على سلسلة حروف لا يضر بالبرنامج، فقط يجب أن تكون الأنواع متوافقة.

فقط تذكر بعض القواعد، وسيصير كل شيء سهل التفسير، فالسابقة LP تعني Long Pointer، في بيئة الوين 32، يعتبر Long مطلقا، إذن لا تقلق بشأنه. أما لو أنك لا تعرف المقصود بالمؤشر، فيستحسن دائما مراجعة كتب أخرى مهتمة بذلك قبل تتمة هذا الدليل.

يأتي بعد LP الحرف C والذي يعني Const، وهذا بالنسبة للأنواع من شاكلة LPCSTR، أي أن سلسلة الحروف هي الثابتة، ولا يمكن تغيير محتواها، في حين LPSTR ليست ثابتة، ويمكن تعديل محتواها.

أيضا يمكن أن نجد الحرف T ممزوجا بها، لا تقلق بشأن ذلك، فمادمت تعمل باليونيكود (Unicode)، فإنها لا تعني شيئا.



III - نافذة بسيطة:

المثال المرفق: simple_window

عمل نافذة بسيطة ليس بالأمر الصعب، ولكنه ليس عملا عشوائيا، فمعرفة مدلول كل دالة وتعليمة واجب حتى يمكن التحكم في البرنامج.

معرفة خطوات العمل الذي تنتهجها دوال الـ API لعمل نافذة هو أمر في غاية الأهمية، لأنها تعتبر الأرضية الأساسية لكل نوافذ الويندوز.

يستحسن دائما عرض الكود قبل مناقشته، وهاأنذا أعتمد نفس الأسلوب:

```
#include <windows.h>

const char g_szClassName[] = "myWindowClass";

/* الخطوة 4: إجراء النافذة */

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
switch(msg)
	{
	case WM_CLOSE:
	DestroyWindow(hwnd);
	break;
	case WM_DESTROY:
	PostQuitMessage(0);
	break;
	default:
	return DefWindowProc(hwnd, msg, wParam, lParam);
	}
return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
	WNDCLASSEX wc;
	HWND hwnd;
	MSG Msg;

/* الخطوة 1: تسجيل فئة النافذة */


	wc.cbSize		= sizeof(WNDCLASSEX);
	wc.style		= 0;
	wc.lpfnWndProc	= WndProc;
```

```
        wc.cbWndExtra    = 0;
        wc.hInstance     = hInstance;
        wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
        wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
        wc.lpszMenuName  = NULL;
        wc.lpszClassName = g_szClassName;
        wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

if(!RegisterClassEx(&wc))
        {
        MessageBox(NULL, "Window Registration Failed!", "Error!",
        MB_ICONEXCLAMATION | MB_OK);
        return 0;
        }

/* الخطوة 2: إنشاء النافذة */

hwnd = CreateWindowEx(WS_EX_CLIENTEDGE,g_szClassName,
                      "The title of my window",WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
                      NULL, NULL, hInstance, NULL);

if(hwnd == NULL)
        {
        MessageBox(NULL, "Window Creation Failed!", "Error!",
        MB_ICONEXCLAMATION | MB_OK);
        return 0;
        }

ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);

/* الخطوة 3: حلقة الرسائل */


while(GetMessage(&Msg, NULL, 0, 0) > 0)
        {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
        }
return Msg.wParam;
}
```

هذا أدنى كود يمكن إدراجه لتحقيق نافذة من نوافذ الوين32، هذا الكود في حدود 70 سطرا.
إذا ما تم تنفيذ المثال السابق بنجاح، فإن هذا المثال سيعمل من دون مشكلة.

الخطوة 1: تسجيل فئة النافذة

فئة النافذة (Window Class) تخزن معلومات تتعلق بنوع النافذة، بما في ذلك إجراءها الخاص بها (Window Procedure) والذي يراقب النافذة، الأيقونة (icon) الصغيرة والكبيرة للنافذة، ولون الخلفية. في هذا المجال يمكنك تسجيل فئة واحدة، وإنشاء ما ترغب فيه من نوافذ من خلال هذه الفئة، من دون إعادة ذكر الخصائص في كل مرة. أغلب ما تحدده من خصائص في فئة النافذة، يمكنك تعديله في النوافذ المشتقة من هذه الفئة.

فئة النافذة ليس لها نفس خصوصيا فئات السي++.

```
const char g_szClassName[] = "myWindowClass";
```

المتغير أعلاه هو من نوع سلسلة حروف ثابتة، ويستخدم هذا المتغير لتخزين إسم فئة النافذة، والذي سنستخدمه لاحقا لتسجيل فئة النافذة مع النظام.

```
WNDCLASSEX wc;
```

```
    wc.cbSize        = sizeof(WNDCLASSEX);
    wc.style         = 0;
    wc.lpfnWndProc   = WndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = g_szClassName;
    wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

    if(!RegisterClassEx(&wc))
    {
        MessageBox(NULL, "Window Registration Failed!", "Error!",
            MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }
```

هذا هو الكود الذي نستخدمه في WinMain() لتسجيل فئة النافذة، نقوم بملء أعضاء الهيكل wc ثم نستدعي الدالة RegisterClassEx() لتسجيل فئة النافذة مع النظام.

أعضاء فئة النافذة هي كالآتي:

cbSize	حجم الهيكل
style	نمط الفئة (*_CS)، لا يجب خلطه مع نمط النافذة (*_WS)، عادة تكون هذه القيمة مساوية لـ 0
lpfnWndProc	مؤشر نحو إجراء النافذة (window procedure) لهذه الفئة (أي لفئة النافذة)
cbClsExtra	كمية البيانات (data) المحجوزة في الذاكرة لهذه الفئة، عادة 0
cbWndExtra	كمية البيانات المحجوزة في الذاكرة لكل نافذة من هذه الفئة، القيمة عادة هي 0
hInstance	مقبض المثيل المستخلص من التطبيق (وهو نفس البارامتر المستخدم في دالة البرنامج الرئيسية (WinMain)
hIcon	أيقونة كبيرة يتم ملاحظتها عن الضغط على Alt+Tab (عادة تكون بحجم 32*32 بكسل)
hCursor	شكل المؤشر أثناء تحريك الماوس ضمن إطار النافذة
hbrBackground	فرشاة الخلفية، وهي لإعطاء لون معين للنافذة
lpszMenuName	اسم لمورد القائمة الممكن استخدامها في فائدة هذه النافذة
lpszClassName	إسم يمكننا من تعريف الفئة وتمييزها عن بقية الفئات
hIconSm	أيقونة صغيرة تظهر في الركن العلوي الأيسر للنافذة، وأيضا في شريط المهام الخاص بالويندوز (عادة تكون بحجم 16*16 بكسل)

قد تكون غير قادر على استيعاب كل هذه المعطيات، شيء طبيعي لا يستدعي القلق، أغلب هذه المعطيات سيتم شرحها بإسهاب في بقية دروس هذا الدليل.

بعدها نسعى لتسجيل هذه الفئة مع النظام RegisterClassEx(&wc) . في حالة الفشل، فإن النظام يرسل إلينا رسالة تخبرنا بذلك وعليه نغادر البرنامج ونعود إلى دالة البرنامج الرئيسية WinMain()

الخطوة 2: إنشاء النافذة

في حالة نجاح عملية التسجيل لفئة النافذة، فإنه يكون بإمكاننا إنشاء نافذة باستخدام هذه الفئة. يجب عليك ملاحظة بارامترات الدالة CreateWindowEx() (والذي يجب دائما فعله حالة استدعاء لدوال API)، لكن سأقوم بشرحها بسرعة في هذا الباب.

```
HWND hwnd;
```

```
hwnd = CreateWindowEx(WS_EX_CLIENTEDGE,g_szClassName,
                      "The title of my window",
                      WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
                      NULL, NULL, hInstance, NULL);
```

البارامتر الأول (WS_EX_CLIENTEDGE) هو نمط النافذة الموسع (extended)، في هذا المثال أعطينا حاشية باطنية غامرة لإطار النافذة. اجعل القيمة 0 إذا أردت معرفة الشكل المناسب لهذه القيمة. جرب بقية القيم ولاحظ النتيجة.

بعد ذلك، لدينا اسم الفئة (g_szClassName)، هذا البارامتر يخبر النظام عن أي نوع من النوافذ يجب إنشاؤه. وبما أننا نسعى لإنشاء نافذة من نوع الفئة التي قمنا بتسجيلها سابقا، فإننا نستعمل اسم هذه الفئة. بعد ذلك قمنا بتحديد اسم أو عنوان نافذتنا، وهو النص الذي سيظهر في التسمية (caption)أو شريط العنوان الخاص بنافذتنا.

البارامتر المعرف بـ WS_OVERLAPPEDWINDOW هو خاص بنمط النافذة، يوجد العديد من هذه البارامترات، ويمكنك تجريبها لمعرفة عمل كل نوع. سيتم التطرق لها لاحقا.

البارامترات الأربعة الموالية ليست معقدة كسابقاتها، فالبارامتران المعرفان بـ CW_USEDEFAULT مرتين هما لإحداثيات النافذة X و Y بداية من الركن العلوي الأيسر للشاشة. البارامتران اللذان يخصان موقع النافذة تم الإشارة إليهما بالقيمة CW_USEDEFAULT لترك الحرية للنظام لاختيار القيم الافتراضية لأي نافذ يتم إنشاؤها. أما القيمتين 320 و 240 فهما تخصان عرض وارتفاع النافذة.

كل القيم محسوبة بوحدة البكسل، وهي أصغر وحدة تقاس بها الأطوال على شاشة الكمبيوتر. عدل على هذه القيم، وبعد التنفيذ لاحظ لنتيجة (تغيرات في موقع وحجم النافذة).

ثم يأتي بعد ذلك (NULL, NULL, g-hIns, NULL) عندنا هنا في هذه البارامترات: مقبض (handle) النافذة الأم، مقبض القائمة، مقبض المثيل عن التطبيق، ومؤشر نحو بيانات (data) إنشاء النافذة.

كثرت المصطلحات لا محالة وتعقدت، لكنها أساسية في بيئة الوين32.

في الويندوز، النوافذ تكون مرتبة على تسلسل هرمي لآباء وأبناء. فعندما تلاحظ زر على النافذة، فإن هذا الزر هو الإبن، والنافذة الحاوية له هي الأم، في مثالنا فإن مقبض النافذة الأم (PerentWindow) معدوم (NULL)، وهذا معناه أن النافذة التي أنشأناها لا أب لها ولا أم، بل هي الآن تتموضع في قمة السلم الهرمي، ولا حكم خارجي عليها.

إن إغلاق أو تلوين النافذة الأم مثلا ينجر عنه نفس التأثير على النوافذ الأبناء (الأزرار ومربعات السرد و و و..) التابعة للنافذة الأم.

نعود إلى البارامترات المتبقية، فالقيمة NULL الثانية تخص مقبض القائمة، وهي معدومة مادمنا لم نشر إلى أي قائمة (menu) في نافذتنا.

البارامتر الثالث مشار إليه بالقيمة (g_hInst) ويخص مقبض المثيل عن التطبيق، وهو مهيأ على قيمة أول بارامتر من بارامترات الدالة الرئيسية ()WinMain.

أما آخر بارامتر من الأربعة المذكورين، والمستهل بالقيمة NULL فهو لبيانات الإنشاء (والتي أهملها دائما) وهذا البارامتر يستخدم لإرسال بيانات إضافية إلى النافذة التي تم إنشاؤها حينا.

NULL لا تعدو أن تكون سوى القيمة 0، حاليا في لغة السي NULL تمثل ((void *) 0) بما أنها تستخدم مع المؤشرات. لهذا فيمكن أن تحصل على تحذيرات حالة استخدامك لـ NULL مع قيم صحيحة. يمكنك استعمال القيمة 0 بدل NULL في هذه الحالة.

أغلب المشاكل التي تعترض المبرمجين ترجع في أغلبها لعدم تدقيقهم في القيم المعادة من الدوال، وهذا لتحديد نجاح أو فشل العملية المنوطة بهذه الدالة. فمثلا CreateWindow() يمكن لها الفشل في العديد من النقاط، إلا أن تكون خبير برمجة وتشفير، ببساطة لوجود إمكانية كبيرة لارتكاب الأخطاء، إلا في حالة قدرتك السريعة على تحديد هذه المعضلات، لهذا فيستحسن دائما لك أن تدقق في القيم المعادة من الدوال لتكملة المشوار البرمجي من دون مشاكل.

```
if(hwnd == NULL)
{
	MessageBox(NULL, "Window Creation Failed!", "Error!",
		MB_ICONEXCLAMATION | MB_OK);
	return 0;
}
```

بعد أن قمنا بإنشاء النافذة، ودققنا في أننا حصلنا على مقبض سليم نحوها، يمكننا بذلك عرض هذه النافذة، مستعملين آخر بارامتر للدالة WinMain() ثم نقوم بتحديثها لضمان أن النافذة قامت بإعادة تصميم نفسها على الشاشة، وهذا بفضل التعليمتين الآتيتين:

```
ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
```

البارامتر nCmdShow هو اختياري، حيث يمكنك عرض النافذة بشكل قياسي اعتمادا على SW_SHOWNORMAL كل الوقت، لكن باستخدام بارامتر يمرر إلى الدالة WinMain() يمكن لأي مستعمل للنافذة التي قمت بإنشائها من أن يتحكم أكثر فيها من خلال إظهارها على شكل مرئي أو مخفي أو مصغر أو غير ذلك.

سوف تجد الكثير من المميزات بمطالعتك لخواص هيكل النافذة، وهذا البارامتر هو لاختيار أسلوب إنجاز النافذة وتنفيذها.

الخطوة 3: حلقة الرسائل

هنا يقبع قلب البرنامج، كل نبضة وكل عملية تتم على مستوى برنامجك، تمر عبر هذه النقطة من التحكم.

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

```
return Msg.wParam;
```

الدالة GetMessage() تقوم بإلتقاط الرسالة من طابور الرسائل الخاص بالتطبيق، في أي وقت يقوم المستخدم بالنقر على زر الماوس أو تحريكها، أو الضغط على أزرار لوحة المفاتيح، أو اختيار بند (item) من بنود القائمة، أو عمل أي شيء آخر، فإن النظام يولد رسائل ويضعها في طابور الرسائل الخاص بالبرنامج المعني. باستدعاء لـ GetMessage() فإنك تقوم باستلام الرسالة السليمة الموالية ونزعها من طابور الرسائل لأجل معالجتها. فإذا لم يكن هناك أي رسالة (أثناء توقف المستخدم عن العمل على الماوس أو لوحة المفاتيح) فإن GetMessage() تكبح (Blocks). إذا لم تستسغ هذا المصطلح، فإنه يدل على أن البرنامج يتوقف عن المعالجة إلى أن يتم توليد رسائل جديدة، وبالتالي يعيدها إليك بفضل هذه الدالة.

الدالة TranslateMessage() تجري بعض المعالجة الإضافية على أحداث لوحة المفاتيح كتوليد رسائل من نوع WM_CHAR لمتابعة رسائل WM_KEYDOWN. أخيرا DispatchMessage() تقوم ببعث الرسالة خارجا إلى النافذة المعنية بالرسالة. هذه قد تكون نافذتنا الأساسية أو أي نافذة أخرى، أو أي تحكم (control)، وفي بعض الأحيان تكون النافذة المنشأة من قبل النظام أو من قبل برنامج آخر.

لا يجب أن تقلقك هذه المراسلات الداخلية، الذي يهم من كل ذلك أنه تم استلام رسائل، وإرسالها خارجا، والنظام يتكفل بالباقي كضمان ربط الرسالة بالنافذة المعنية.

الخطوة 4: إجراء النافذة

إذا كانت حلقة الرسائل (messages loop) تمثل قلب البرنامج، فإن إجراء النافذة يمثل دماغه. هنا حيث يتم ترقية ومعالجة كل الرسائل المرسلة إلى نافذتنا

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
	switch(msg)
	{
	case WM_CLOSE:
		DestroyWindow(hwnd);
		break;
	case WM_DESTROY:
		PostQuitMessage(0);
		break;
	default:
		return DefWindowProc(hwnd, msg, wParam, lParam);
	}
return 0;
}
```

يتم استدعاء هذا الإجراء (procedure) لكل رسالة، HWND هو المقبض الخاص بالنافذة، أي تلك المعنية بالرسالة، وتظهر أهمية ذلك حالة تواجد أكثر من نافذة مشتقة من نفس الفئة (class) وكل منها تستخدم نفس إجراء النافذة ((WinProc) (جرب إنشاء عدة نوافذ من نفس الفئة)، عندها نكون مضطرين لتحديد النافذة المعنية بالرسالة المولدة، فالاختلاف يكون في مقبض النافذة hwnd

المتعلق بكل نافذة. كمثال، عند حصولنا على الرسالة WM_CLOSE() ، فإننا نقوم بهدم النافذة، وبما أننا نستخدم مقابض مختلفة بين النوافذ، فإننا نستعمل هذا البارامتر لتحديد النافذة المعنية بعملية الهدم.

يتم بعث الرسالة WM_CLOSE عند قيام المستخدم بالضغط على الزر الموجود في الركن العلوي الأيمن، أو أثناء ضغطه على Alt+F4. هذا يؤدي فرضا إلى هدم النافذة، لكن أود أن أجعل عملية الهدم لا تتم تلقائيا، وهنا يمكنني مثلا مطالبة المستخدم عبر نافذة حوار من حفظ بياناته قبل إغلاق النافذة مثلا.

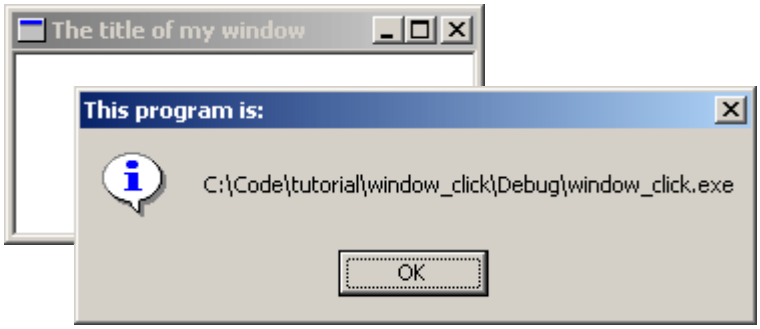
عندما نستدعي الدالة DestroyWindow() فإن النظام يرسل الرسالة WM_DESTROY إلى النافذة المطلوب هدمها، ومنه قبل عملية الهدم للنافذة يتم هدم كل النوافذ الأبناء (أزرار، مربعات...) قبل القيام بهدم النافذة الأم من طرف النظام. في مثالنا لا يوجد سوى نافذة واحدة، وكل النوافذ الأبناء معدومة، فإننا نستدعي الآن من البرنامج إنهاء المهمة، إذن نستدعي الدالة PostQuitMessage(). هذه الدالة ترسل رسالة من نوع WM_QUIT إلى حلقة الرسائل. لن نحصل على هذه الرسالة أبدا، لأنها تسبب عودة القيمة FALSE من الدالة GesMessage()، وكما ترى في كود حلقة الرسائل، فإنه لما يحدث عودة القيمة FALSE فإن الحلقة تتوقف عن معالجة الرسائل، وتعيد كود النتيجة النهائية، القيمة wParam الخاصة بـ WM_QUIT والتي سوف نقوم بإدراجها ضمن بارامترات الدالة PostQuitMessage(). القيمة المعادة تكون هامة فقط في حالة استدعاء برنامجك من قبل برنامج آخر، وتود عمل نشاط ما.

الخطوة 5: لا توجد خطوة خامسة

حتى هذا الموضع، وبدل من أن ينقشع الضباب، تزداد المفاهيم تعقيدا. لا تقلق وكن أكثر رزانة، لأنه ومع التعود أكثر على العمل وفق المثال المعروض ضمن هذا الدرس، فإنه يكون بإمكانك اكتشاف العديد من العناصر المبهمة وقد اتضحت أكثر.

راجع ملفات المساعدة المرفقة مع المصرف (compiler) الذي تمتلكه لمطالعة كل القيم التي يتخذها بارامتر معين، وكل البارامترات التابعة لدالة معينة.

IV - إلتقاط الرسائل:



المثال المرفق: window_click

حسنا، أنشأنا نافذة، لكنها لا تقوم إلا ببعض المهام المحددة من قبل DefWindowProc() كتكبير، وتصغير الإطار، عرض الاختصارات...

في نشاطنا الآتي، سأعرض عليك طريقة تعديل ما قمت بإنجازه من قبل. فقط اتبع الخطوات، ولا تتعجل.

في البداية، خذ المثال السابق، وتأكد من أنه يعمل من دون مشكلة. ويمكنك أن تقوم بتعديل بعض الكود عليه، أو أن تصنع نسخة احتياطية منه حتى تحصل على أمثلة مناسبة لكل فصل.

سوف نقوم بإضافة إمكانية عرض اسم البرنامج للمستخدم عندما يقوم بالنقر على نافذتنا. هذا ليس بالشيء المهم، لكنه يوضح لنا أسلوب التقاط الرسائل ومعالجتها. دعنا نرى ما كان موجودا عندنا في إجراء النافذة WndProc():

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
    switch(msg)
    {
        case WM_CLOSE:/*حالة إغلاق النافذة */
            DestroyWindow(hwnd);
        break;
        case WM_DESTROY:/*حالة هدم النافذة */
            PostQuitMessage(0);
        break;
        default:/* في بقية الحالات الغير محددة */
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

إذا كنا نرغب في التقاط نقرات الماوس، فإننا مطالبون بإضافة مقبض الرسالة WM_LBUTTONDOWN لتحسس النقر على الزر الأيسر للماوس، (أو WM_RBUTTONDOWN أو WM_MBUTTONDOWN لزري الماوس الأيمن والأوسط على الترتيب).

إذا كنت راغبا في التقاط هذه الرسالة، فإنك إذن ملزم بإضافة هذا النوع من الرسائل (أي الخاصة بالنقر على الزر الأيسر للماوس (WM_LBUTOONDOWN) في الإجراء WndProc() الخاص بفئة النافذة كما يأتي:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
switch(msg)
      {
      case WM_LBUTTONDOWN:      /*هذا الذي أضفناه إلى الكود*/
      break;                    /*وهذه أيضا*/
      case WM_CLOSE:
      DestroyWindow(hwnd);
      break;
      case WM_DESTROY:
      PostQuitMessage(0);
      break;
      default:
      return DefWindowProc(hwnd, msg, wParam, lParam);
      }
return 0;
}
```

ترتيب الحالات (cases) لا يهم، فقط تأكد من وجود التعليمة break وراء كل حالة.

في مثالنا أضفنا مقبض رسائل زر الماوس الأيسر، لكنه لا يعمل أي شيء. سوف نقوم إذن بإضافة كود حالة النقر على الزر الأيسر، وسيكون هذا الكود قادرا على تحديد اسم الملف الحالي ومساره كاملا (أو أن تعدل المثال بحيث يعرض مربع حوار فقط).

ما هي التعليمات التي تقوم بذلك؟ أولا سأعرض عليك الكود الذي ينجز هذه المهمة وكيفية إدراجه ضمن المكان المحدد، حتى تكون قادرا مستقبلا على عمل نفس الخطوات مع هذا الكود أو غيره من الأكواد.

```
GetModuleFileName(hInstance, szFileName, MAX_PATH);
MessageBox(hwnd, szFileName, "This program is:", MB_OK|
           MB_ICONINFORMATION);
```

هذا الكود لا يعمل من أي موقع كان، لكننا نود أن ينفذ حالة نقر المستخدم على الزر الأيسر للماوس، إذن الموقع المناسب يكون من آخر عملية تعديل قمنا بها في قلب الإجراء ()WndProc، هذا الإجراء سيصير على النحو الآتي:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
  switch(msg)
  {
   case WM_LBUTTONDOWN:
       /*بداية الكود الجديد*/
       {
       char szFileName[MAX_PATH];
```

```
        GetModuleFileName(hInstance, szFileName, MAX_PATH);
        MessageBox(hwnd, szFileName, "This program is:",
                   MB_OK | MB_ICONINFORMATION);
        }
        /*نهاية الكود الجديد*/
    break;
    case WM_CLOSE:
        DestroyWindow(hwnd);
    break;
    case WM_DESTROY:
        PostQuitMessage(0);
    break;
    default:
        return DefWindowProc(hwnd, msg, wParam, lParam);
    }
return 0;
}
```

ماذا تعني الأسطر الأربعة الجديدة؟؟؟

أولا تم إضافة الحاضنتين { } داخل حالة من حالات التعليمة switch، لأنه وكما هو معلوم في لغة السي، فإننا نضيف هاتين الحاضنتين عندما نقوم بالتصريح بمتغيرات جديدة داخل حالة من الحالات التابعة لـ switch. فلقد قمنا هنا بالتصريح بمتغيرين: hInstance و szFileName . فالدالة GetModuleFileName() تشترط أن يكون أول بارمتر فيها هو مقبض المثيل (HINSTANCE) والذي يشير إلى الوحدة النمطية المنفذة (أي برنامجنا، الملف exe.). لكن كيف نحصل على هذا المقبض؟ الدالة GetModuleHandle() تجيب عن ذلك، حيث إذا ما مررنا القيمة NULL إلى هذه الدالة، فإنها ستعيد لنا مقبض الملف الجاري تنفيذه، والذي هو ما نحتاج إليه في مثالنا هذا.

```
HINSTANCE hInstance = GetModuleHandle(NULL);
```

إذن في التعليمة أعلاه، قمنا بالتصريح بمتغير جديد hInstance وأوكلناه بمقبض الوحدة النمطية للبرنامج الجاري الذي حصلنا عليه من الدالة GetModuleHandle().

هذا كله من أجل البارامتر الأول الخاص بالدالة GetModuleFileName() . الآن نحتاج إلى البارامتر الثاني، والذي هو مؤشر يتم فيه تخزين إسم ومسار الملف الخاص بالوحدة النمطية، ونوع بيانات هذا المؤشر هو سلسلة حروف LPTSTR (أو LPSTR) علما أن LPSTR يعادل char*.

إذن يمكننا التصريح بنسق حروف (أو سلسلة حروف) على هذا الشكل:

```
char szFileName[MAX_PATH];
```

MAX_PATH هو عبارة عن ماكرو رائع، تم التصريح به داخل الملف windows.h ويحدد هذا الماكرو أقصى طول ممكن لاسم ومسار الملف في بيئة الوين 32.

أيضا نقوم بتمرير الماكرو MAX_PATH إلى الدالة GetModuleFileName() لتحديد طول الإسم والمسار، وهذا هو البارامتر الثالث في هذه الدالة.

بعد استدعاء هذه الدالة، فإن المتغير (أو المؤشر) szFileName يحوي الآن سلسلة حروف تنتهي بالبايت الخالي حاوية بذلك لاسم ومسار ملفنا التنفيذي. نقوم بعدها بتمرير هذه السلسلة إلى MessageBox() على اعتبار هذه الدالة أبسط وسيلة لعرض الناتج للمستخدم.

قم بتصريف كامل الكود، وعند التنفيذ، قم بالنقر على الزر الأيسر للماوس (ضمن إطار النافذة)، ستحصل على مربع حوار يخبرك باسم الملف ومساره.

إذا لم يعمل الكود المعدل لوجود خطأ ما، فإليك كامل البرنامج، انسخه وقم بتنفيذه. لا محالة سيعمل.

```
#include <windows.h>
const char g_szClassName[] = "myWindowClass";
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam,
                         LPARAM lParam)
{
switch(msg)
       {
       case WM_LBUTTONDOWN:
            {
            char szFileName[MAX_PATH];
            HINSTANCE hInstance = GetModuleHandle(NULL);
            GetModuleFileName(hInstance, szFileName, MAX_PATH);
            MessageBox(hwnd, szFileName, "This program is:",MB_OK|
                         MB_ICONINFORMATION);
       }
       break;
       case WM_CLOSE:
            DestroyWindow(hwnd);
       break;
       case WM_DESTROY:
            PostQuitMessage(0);
       break;
       default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
}
return 0;
}
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;    HWND hwnd;    MSG Msg;
    wc.cbSize        = sizeof(WNDCLASSEX);
    wc.style         = 0;
    wc.lpfnWndProc   = WndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
```

```
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = g_szClassName;
    wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
    if(!RegisterClassEx(&wc))    {
        MessageBox(NULL, "Window Registration Failed!", "Error!",
            MB_ICONEXCLAMATION | MB_OK);        return 0;    }
    hwnd = CreateWindowEx(WS_EX_CLIENTEDGE, g_szClassName,
           "The title of my window", WS_OVERLAPPEDWINDOW,
           CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
           NULL, NULL, hInstance, NULL);
    if(hwnd == NULL)
    {
         MessageBox(NULL, "Window Creation Failed!", "Error!",
                    MB_ICONEXCLAMATION | MB_OK);
         return 0;
    }
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);
    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```

V - إدراج حلقة الرسائل:

معرفة حلقة الرسائل، وهيكل الرسالة المرسلة في برامج الويندوز يعتبر أمرا في غاية الأهمية في حالة كتابة برامج موجهة لأداء مهام محددة. الآن نحن مطالبون في التعمق في أسلوب معالجة الرسائل، لأنه من غير إدراك تام لهذا فإن المفاهيم سيعتريها التشويش لاحقا.

ما هي الرسالة؟

الرسالة هي قيمة صحيحة. فإذا ما طالعت ملفات الرأس (أو المقدمة) فإنك ستجد تصريحات على هذه الشاكلة:

```
#define WM_INITDIALOG                   0x0110
#define WM_COMMAND                      0x0111

#define WM_LBUTTONDOWN                  0x0201
```

... وما إلى ذلك. الرسائل تستعمل كوسيلة اتصال بين النظام والنوافذ المعنية، وتقريبا هذه الرسائل تعمل على نقل كل شيء بين هذه الأطراف، فإذا كنت تريد من نافذة أو تحكم (control) (والذي هو في حد ذاته نوع خاص من النوافذ) فما عليك إلا إرسال رسالة إلى هذه النافذة. وإذا ما أرادت نافذة أخرى منك شيئا، فإنها ترسل لك رسالة. إذا ما حصل حدث (event) ما كنقر المستخدم على زر الماوس أو تحريكها أو ضغطه على زر من أزرار لوحة المفاتيح، فإن النظام يولد رسائل ويرسلها إلى النافذة المعنية بذلك. إذا كنت نافذة من هذه النوافذ فإنك تلتقط الرسالة، وتتفاعل معها.

كل رسالة من رسائل النوافذ تحمل في ثناياها بارامترين اثنين، wParam و lParam. أصلا، كانت wParam ذات حجم 16 بت، و lParam بحجم 32 بت، لكن في الوين32 كل منهما صار بحجم 32 بت.

ليست كل الرسائل تستعمل هذين البارامترين، وأيضا طريقة استعمال هذين البارامترين تختلف باختلاف الرسائل. كمثال: الرسالة WM_CLOSE لا تستعمل أيا من البارامترين، وينبغي عليك إذن إهمالهما في هذه الحالة. أما الرسالة WM_COMMAND فإنها تستعمل كلا البارامترين (wParam و lParam)؛ البارامتر الأول wParam يحوي قيمتين، HIWORD و LOWORD (يمكننا تسميتهما: كلمة عليا، وكلمة دنيا)، حيث HIWORD الخاص بـ wParam هو إشعار الرسالة (إذا كان قابلا للتطبيق)، و LOWORD هو التحكم (control) أو معرف القائمة (menu identificator) الذي أرسل الرسالة. في حين lParam هو مقبض النافذة (HWND) أو التحكم الذي أرسل الرسالة ، وفي حالة لم يكن التحكم هو المرسل لهذه الرسالة فإن البارامتر lParam سيحوي القيمة NULL.

HIWORD() و LOWORD() هما 2 ماكرو (هل يمكن تثنية هذه الكلمة؟) تم تعريفهما من قبل ويندوز، مجال القيمة الأولى في البايتين المرتفعين (High Word) لكلمة مضاعفة ذات 32 بت، أي وفقا للقيمة السدس عشرية الآتية 0xffff0000، أما الكلمة الدنيا فتحدد بالبايتين المنخفضين 0x0000ffff. لأنه وكما هو معلوم، فإن حجم الكلمة WORD في الوين 32 هو 16 بت (2 بايت)، وحجم الكلمة المضاعفة DWORD 32 بت (4 بايت).

لإرسال رسالة، يمكنك استخدام PostMessage() أو SendMessage().

الدالة الأولى: PostMessage() تضع الرسالة في طابور الرسائل وتعود في الحين. هذا يعني أنه بعد مهمة هذه الدالة فإن الرسالة قد يتم معالجتها أو قد لا يتم. في حين الدالة SendMessage() ترسل الرسالة مباشرة إلى النافذة ولا تعود إلا بعد أن يكمل ويندوز عملية المعالجة لها.

إذا كنا نود إغلاق النافذة، فإننا سنرسل لها الرسالة WM_CLOSE على هذا الشكل PostMessage(hwnd, WM_CLOSE, 0, 0) والتي لها نفس عمل الزر لاحظ أن كلا من wParam و lParam يساوي 0، لأن الرسالة WM_CLOSE وكما أشرنا من قبل لا تستعمل هذين البارامترين.

مربعات الحوار:

عندما تبدأ في استعمال مربعات الحوار، فإنك ستضطر إلى بعث رسائل إلى التحكمات من أجل التخاطب معها. يمكنك عمل ذلك على خطوتين، أول يجب أن تتحصل على مقبض (handle) العنصر التابع لمربع الحوار من خلال الدالة GetDlgItem() عن طريق المعرف (ID) ثم بعدها تستعمل SendMessage()، أو يمكنك مباشرة استعمال الدالة SendDlgItemMessage() التي تجمع الخطوتين معا. يعطى لهذه الدالة مقبض النافذة، ومعرف النافذة الإبن (مذكر<>مؤنث للضرورة)، وسوف تحصل بذلك على مقبض النافذة الابن، ثم أرسل إليها رسالة.

SendDlgItemMessage() وبقية دوال الـ API كالدالة GetDlgItemText() تعمل على كل النوافذ، وليس فقط على مربعات الحوار.

ما هو طابور الرسائل؟

لنفرض أنك أثناء إرسالك لرسالة WM_PAINT للنافذة من أجل إعادة رسم نفسها، فإن المستخدم قام بالضغط على مجموعة أوامر على لوحة المفاتيح. ماذا يحدث بالضبط؟؟ هل ستقوم بتوقيف عملية الرسم من أجل التقاط رسائل لوحة المفاتيح أم أنك تهمل هذه الرسائل أثناء عملية الرسم؟ كلا القرارين يعتبر غير موضوعي. إذن لدينا طابور الرسائل، لما تولد هذه الرسائل يتم إضافتها إلى تلك الموجودة في طابور الرسائل، ولما يتم التقاطها وتحديد مقابضها لأجل معالجتها فإنها تحذف من هذا الطابور. هذا يضمن بأنك لن تهمل أية رسالة، فإذا قمت بالتقاط إحداها فإن البقية تتقدم نحو المقدمة حسب الترتيب إلى أن يتم التقاطها كلها.

ما هي حلقة الرسائل؟

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

1. حلقة الرسائل تستدعي الدالة GetMessgae()، هذه الأخيرة تبحث في طابور الرسائل، فإذا وجدت أن الطابور فارغا فإن البرنامج يتوقف منتظرا لأية رسالة جديدة.

2. لما يحصل حدث (event) ما متسببا في توليد رسالة (تحريك الماوس مثلا) فإنه يتم وضعها في طابور الرسائل، فتعيد الدالة GetMessage() قيمة موجبة (أكبر من الصفر) مشيرة بذلك إلى وجود رسالة أو رسائل يجب معالجتها، وتقوم بذلك بملء البيانات الأعضاء للهيكل

MSG (الرسالة msg هي من نوع الهيكل MSG، بالتالي فهي تتوفر على مجموعة من البيانات الأعضاء، والتي يمكن مراجعتها من ملفات المساعدة المرفقة مع المصرف (compiler))، قلت أنها تملأ البيانات الأعضاء للمتغير msg الممرر إلى هذه الدالة. كما أن هذه الدالة تعيد القيمة 0 إذا كانت الرسالة من نوع WM_QUIT، أو قيمة سالبة إذا ما اعترضتها أخطاء.

3. نأخذ الرسالة (المحفوظة في المتغير msg) ونمررها إلى الدالة (TranslateMessage، هذه الأخيرة تقوم ببعض المعالجة على الرسالة، حيث تترجم رسائل ذات مفتاح وهمي إلى رسائل ذات رموز. هذه العملية في الغالب زائدة، لكن بعض الأشياء لا تعمل إلا إذا قامت هذه الدالة بمهمتها.

4. بعد ذلك، نقوم بتمرير الرسالة إلى الدالة (DispatchMessage، لكن ماذا تفعل هذه الدالة للرسالة؟ إنها تدقق في النافذة المعنية بالرسالة، وبالتالي تبعثها إلى إجراء النافذة (Window Procedure) الخاص بهذه النافذة. يتم تمرير الرسالة ومقبض النافذة (ضروري حالة اشتراك عدة نوافذ في نفس إجراء النافذة، وبالتالي فهي تختلف في المقابض) والبارامترين wParam و lParam.

5. في إجراء النافذة الخاصة بك، تقوم بالتدقيق في الرسالة وبارامتراتها، وبالتالي اختيار النشاط الذي ترغب فيه، كما يمكنك استعمال الدالة (DefWindowProc التي تترجم الرسالة بالنشاط الافتراضي (غلق النافذة، تكبيرها، تصغيرها، عرض الإختصارات، تلوين الأزرار...).

6. لما تنتهي من معالجة الرسالة، فإن إجراء النافذة يعود (returns)، فينتقل التحكم إلى (DispatchMessage، فتعود هذه الأخيرة ويعود معها التحكم إلى حلقة الرسائل، أي أننا نرجع إلى البداية، وهذا تتواصل حركة الرسائل بين مد وجزر.

الموضوع معقد وفي نفس الوقت مهم للغاية، فهذا هو مفهوم برامج الويندوز. إجراء نافذتك لا يتم استدعاءه بأسلوب سحري من قبل النظام، لكنك أنت الذي تقوم باستدعاءه بطريقة غير مباشرة من خلال استخدامك للدالة (DispatchMessage. يمكنك استدعاء إجراء النافذة مباشرة من خلال إستدعاء الدالة (GetWindowLong في مقبض النافذة المعنية بالرسالة للوصول إلى إجراء النافذة.

```
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
 WNDPROC fWndProc = (WNDPROC)GetWindowLong(Msg.hwnd, GWL_WNDPROC);
 fWndProc(Msg.hwnd, Msg.message, Msg.wParam, Msg.lParam);
}
```

لقد قمت بتجريب هذا الكود مع المثال السابق، وعمل بشكل سليم، لكن توجد هناك نقاط لا يجب إهمالها، كاستخدام مترجم Unicode/ANSI، طلب ردود المؤقت (callback)، وما إلى ذلك. عمل كهذا قد يسبب تعطل البرنامج، لهذا فإن انتهاج الأسلوب المباشر في طلب إجراء النافذة يعتبر عمل تجريبي لك أن تقوم به، وليس وسيلة برمجية عليك اتباعها.

لاحظ أننا استخدمنا الدالة (GetWindowLong لاستخلاص إجراء النافذة المشترك مع النافذة. لماذا لا نقوم فقط باستدعاء (WndProc مباشرة؟ حسنا، إن حلقة الرسائل مسؤولة عن كل النوافذ في برنامجنا، هذا يعني إدراج أشياء كالأزرار ومربعات السرد والتي لها هي أيضا إجراءات النوافذ الخاصة بها، إذن نحتاج إلى التأكد من أننا نستدعي إجراء النافذة المناسب. وبما أن أكثر من نافذة يمكنها الاشتراك في نفس إجراء النافذة، فإن أول بارامتر (مقبض النافذة) يستخدم لإخبار الإجراء عن النافذة المعنية بالرسالة المرسلة

كما تلاحظ، فإن تطبيقك يقضي معظم وقته في عملية تبادل الرسائل بين حلقة الرسائل وإجراء النافذة، وتبقى الدوامة تعمل، لكن ماذا لو أردت من برنامج أن ينهي عمله؟ بما أننا نستخدم حلقة(while، فإذا أعادت الدالة GetMessage() القيمة FALSE (أي 0)، الحلقة تنهي عملها، ونجد أنفسنا أمام نهاية الدالة الرئيسية في البرنامج WinMain(). وهذا ما تقوم بها الدالة PostQuitMessage(). حيث تضع WM_QUIT في طابور الرسائل، وبدلا من إعادة قيمة موجبة، فإن GetMessage() ستملأ هيكل الرسالة (Msg) وتعيد القيمة 0.

في هذا الموضع، wParam الخاص بالرسالة يحوي القيمة التي ستمررها إلى الدالة PostQuitMessage() والتي يمكنك إهمالها أو أن تعيدها إلى WinMain() التي تستخدمها ككود النهاية بعد اكتمال العمل.

ملاحظة مهمة: GetMessage() ستعيد القيمة –1 إذا ما صادفتها أخطاء. إذن كن متأكدا من أنك لن تنسى هذا، لأنه يحتمل أن يتعطل عملك في بعض النقاط...رغم أن GetMessage() تم تعريفها على أنها تعيد قيما بولبانية (true & false) على اعتبار أن BOOL هي معرفة على أساس UINT (عدد صحيح من غير إشارة).

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

```
while(GetMessage(&Msg, NULL, 0, 0) != 0)
```

```
while(GetMessage(&Msg, NULL, 0, 0) == TRUE)
```

الأكواد أعلاه كلها خاطئة!!!

كما تلاحظ، لقد استخدمت الكود الأول في الأعمال التطبيقية المرفقة مع هذا الدليل، وبقي يعمل مادامت الدالة GetMessage() لم تعرف أية عراقيل، لقد أغفلت تنبيهك في كثير من الأحيان أنك باستخدامك لمثل ذلك فإن كودك سيكون غير صالح في غالب الأحيان، وأن الدالة GetMessage() ستفشل في بعض النقاط. سأقوم لاحقا بتفسير ذلك. فلا تغضب مني إن أهملت بعضا من الملاحظات أو نسيتها.

```
while(GetMessage(&Msg, NULL, 0, 0) > 0)
```

يستحسن دائما أن تستخدم هذا الأسلوب من الكود.

أتمنى أن يكون الضباب قد انقشع عن مفهوم حلقة رسائل الويندوز، أما إن لم يتم ذلك، فلا تقلق، لأن الأمور ستزداد اتضاحا مع الوقت.

VI - إستخدام الموارد:

تستطيع مطالعة الملحق الموجود في أخر هذه الدروس، لمزيد من المعلومات حول الموارد في الفيجول سي++ والبورلاند سي++.

قبل التعمق أكثر في ميدان الوين32، فضلت تغطية موضوع الموارد بدلا من أقوم بإعادة كتابتها في كل فصل من فصول هذا الدليل.

الموارد هي بتات من البانات (data) معرفة مسبقا، ومخزنة على شكل ثنائي (binary) داخل ملفك التنفيذي.

يمكنك إنشاء موارد في سكريبت الموارد، وهو ملف ذو توسع ".rc". أغلب المصرفات المعروضة في السوق، تتوفر على محرر موارد مرئي، هذا الأخير يمكنك من إنشاء موارد بيانيا من دون اللجوء إلى تدوين أسطر برمجية يدويا، لكن أحيانا يعتبر الخيار الثاني الوسيلة الوحيدة القادرة على تحقيق ذلك، خاصة إذا كان مصرفك لا يتوفر على محرر موارد، وهذا ما قد يعني أنه لا يقدر على دعم العمل الدقيق الذي تسعى إليه.

للأسف، فإن مختلف المصرفات تختلف في معالجتها لملفات الموارد، لهذا سأعمل جاهدا لإبراز أكثر النقاط شيوعا بينها، وأدنى اختلاف ممكن لتحقيق موارد بصفة عامة.

محرر الموارد المرفق مع الفيجول سي++ يجعل عملية تعديل الموارد يدويا (بكتابة الكود يدويا) عملية صعبة، ويفرض شكلا معينا عليها، وفي الأخير هذا المحرر يتلف الملف إذا ما قمت بحفظه بعدما كتبت كوده يدويا.

بصفة عامة، لا يمكن إنشاء ملف rc. بدءا من الصفر من خلال الكود، لكن معرفة أسلوب تعديل ذلك يدويا يعتبر أمرا أساسيا.

شيء آخر مزعج، وهو أن فيجول سي++ يعطي اسما افتراضيا (default) للملف الرأسي للمورد (resource.h) إلا إذا رغبت في تحديد الاسم كما تشاء. لكن من أجل تبسيط هذا الموضوع فإني سأقبل بالأمور الافتراضية (default)، وسأقوم في الملحق الخاص بالمصرف بتوضيح طريقة تعديل ذلك.

أولا دعنا نأخذ سكريبت مورد بسيط، مع أيقونة (icon) واحدة.

```
#include "resource.h"

IDI_MYICON ICON "my_icon.ico"
```

هذا كل ما يوجد في الملف. IDI_MYICON هو معرف المورد، ICON هو النوع، و"my_icon.ico" هو اسم الملف الخارجي الذي يحوي هذه الأيقونة. يجب أن يعمل هذا على أغلب المصرفات.

الآن ماذا يعني سطر الإدراج #include "resource.h" ؟ حسنا، برنامجك يحتاج إلى طريقة يقدر منها تعريف الأيقونة، وأحسن طريقة لذلك هي من خلال تخصيص معرف (ID) وحيد (في مثالنا

IDI_MYICON). يمكننا القيام بذلك من خلال إنشاء ملف رأس "resource.h" وإدراجه في كل من سكريبت المورد و ملف المصدر. فملف الرأس الخاص بالمورد في مثالنا يحوي السطر الآتي:

```
#define IDI_MYICON  101
```

كما ترى، لقد أسندنا إلى المعرف IDI_MYICON القيمة 101. يمكننا إهمال هذا المعرف وتعويضه مباشرة بالقيمة 101 للإشارة إلى الأيقونة في أي موضع نحتاج فيه إلى ذلك، لكن IDI_MYICON يعتبر أكثر وضوحا من القيمة الرقمية وخاصة عند كثرة الموارد والتي تكثر معها الأرقام.

الآن نود إضافة قائمة إلى ملف المورد:

```
#include "resource.h"
IDI_MYICON ICON "my_icon.ico"
IDR_MYMENU MENU
BEGIN
      POPUP "&File"
      BEGIN
           MENUITEM "E&xit", ID_FILE_EXIT
      END
END
```

مرة أخرى، IDR_MYMENU هو اسم المورد و MENU هو نوعه. الآن هناك نقطة مهمة، لاحظ المفتاحين BEGIN و END. بعض المصرفات تستخدم الحاضنات بدلا من هذين المفتاحين، { بديلا عن BEGIN، و} بديلا عنEND. أما لو كان مصرفك يدعم كلا أسلوبين، فأنت حر في اختيار أيهما، فقط تأكد من ذلك.

قمنا أيضا بإضافة معرف جديد، ID_FILE_EXIT، إذن نحتاج لإضافة هذا المعرف والقيمة التي تساويه في ملف الرأس الخاص المورد، نعود إلى resource.h ونكتب فيه الآتي:

```
#define  IDI_MYICON      101
#define  ID_FILE_EXIT    4001
```

إن توليد ومتابعة آثار هذه المعرفات يصير أمرا روتينيا في المشاريع الكبرى، لهذا فإن أغلب المبرمجين يستخدمون محرر المورد المرئي (visual resource editor) الذي ينجز هذه المهمة بدقة.

مع الوقت وكثرة التجريب، يقل الإخفاق، وتقدر لوحدك من إضافة بنود كثيرة وذات تفرعات مختلفة، معتمدا في ذلك على نفس أسلوب التصريح بالقائمة في ملف المورد.

وهذا مثال يعرض طريقة استخدام المورد في برنامجك.

```
HICON hMyIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MYICON));
```

البارامتر الأول للدالة LoadIcon() وكبقية دوال استعمال المورد هو مقبض المثيل الحالي (الذي أعطيناه لـ WinMain() والذي يمكن دائما الحصول عليه بفضل الدالة GetModuleHandle() كما تم توضيحه في الفصل السابق) ↺ . أما البارامتر الثاني فهو معرف المورد.

لعلك مستغرب لوجود الدالة MAKEINTRESOURCE() ولعلك أيضا مستغرب لوجود بارامتر من نوع LPCTSTR بدلا من UINT في الدالة LoadIcon() لما قمنا بتمريرها المعرف (ID).

مهمة الدالة MAKEINTRESOURCE() هو أنها تقوم بالتحويل القسري (casting) من العدد الصحيح (وهو نوع المعرفات التي استخدمناها) إلى LPCTSTR، والذي تترقبه الدالة LoadIcon(). هذا يقودنا إلى الطريقة الثانية المعتمدة في تعريف الموارد، والتي تكون من نوع سلسلة حروف. لا أحد الآن يستخدم هذا، لكن إذا لم تستعمل #define لإسناد قيمة عددية لمواردك، إذن فإسم المورد سيتم ترجمته إلى سلسلة حروف، وسيشار إليه في برنامجك على هذا الشكل:

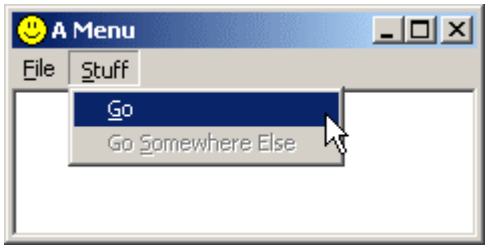
```
HICON hMyIcon = LoadIcon(hInstance, "MYICON");
```

إن الدالة LoadIcon() وبقية دوال الـ API لتحميل الموارد يمكنها التمييز بين قيمة صحيحة (integer) أو مؤشر على سلسلة حروف من خلال التدقيق في الكلمة العليا (High Word) لهذه القيمة الممررة للدالة. إذا كانت 0 (وهي حالة بعض الأعداد الصحيحة ذات قيمة أقل أو يساوي 65535) فإنها تعتبرها معرف مورد (ID). وهذا بطبيعة الحال يحدد مواردك لاستخدام معرفات أقل من 65535، ومادامت مواردك ليست هائلة العدد فإن ذلك لا يخلق أية مشاكل. أما لو كانت الكلمة العليا للقيمة الممررة لدوال API تختلف عن الصفر فإن هذه الدوال تعتبرها مؤشرا، وستأخذ في الاعتبار الموارد حسب الاسم. لا تعتمد أبدا على الـAPI من أجل عمل ذلك إلا إذا تم تحديدها في الوثائق المرفقة.

كمثال، هذا لا يعمل بالنسبة لبنود القائمة كالبند ID_FILE_EXIT، حيث أنه لا يمكنها إلا أن تكون من نوع صحيح.

VII - القوائم والأيقونات:

المثال المرفق: menu_one



هذا فصل مختصر يعرض عليك طريقة إضافة قوائم بسيطة إلى نافذتك. عادة تستخدم مورد قائمة نصف محضر. هذا المورد يكون في ملف rc. ويتم تصريفه وربطه بالملف التنفيذي exe.

هذا يخص بعض المصرفات، لكن المصرفات التجارية تحوي محرر مورد يتيح لك إمكانية إنشاء قائمتك الخاصة، لكن في هذا المثال سأعرض كود الملف rc.، إذن يمكنك إضافته يدويا. كما أنني في العادة أستخدم ملف رأسي h. والذي أدرجه من خلال التعليمة #include في كل من ملف المصدر c. وملف المورد.rc.، هذا الملف يحوي معرفات التحكمات (controls) وبنود القائمة...إلخ.

في هذا المثال، يمكنك البدء من خلال كود النافذة المتوفر في المثال السابق simple_window وأضف إليه بعد ذلك الكود الآتي كما هو معروض.

أولا في الملف h.، وعادة هذا الملف يسمى " resource .h "

```
#define     IDR_MYMENU     101
#define     IDI_MYICON     201

#define     ID_FILE_EXIT   9001
#define     ID_STUFF_GO    9002
```

ليس بالشيء الكثير، لكن قائمتنا هنا ستكون بسيطة للغاية. الأسماء والقيم في هذا المثال ليست إلزامية، بل يمكنك اختيار ما تشاء.

سنكتب الآن ملف المورد (.rc)

```
#include "resource.h"
IDR_MYMENU MENU
BEGIN
    POPUP "&File"
        BEGIN
            MENUITEM "E&xit", ID_FILE_EXIT
        END
    POPUP "&Stuff"
        BEGIN
            MENUITEM "&Go", ID_STUFF_GO
            MENUITEM "G&o somewhere else", 0, GRAYED
        END
END
IDI_MYICON ICON "menu_one.ico"
```

ثم أنت مطالب بإضافة ملف rc. إلى مشروعك، كما أنك مطالب أيضا بإدراج ملف المورد resource.h في ملف المصدر (.c) من خلال التعليمة #include وهذا لتعريف بنود القائمة ومواردها.

أبسط طريقة لوصل القائمة والأيقونة بالنافذة هو أن تحددها لما تقوم بتعريف فئة النافذة (window class)، على هذا الشكل:

```
wc.lpszMenuName     = MAKEINTRESOURCE(IDR_MYMENU);
```

```
wc.hIcon    = LoadIcon(GetModuleHandle(NULL),
                       MAKEINTRESOURCE(IDI_MYICON));

wc.hIconSm = (HICON)LoadImage(GetModuleHandle(NULL),
                              MAKEINTRESOURCE(IDI_MYICON),
                              IMAGE_ICON, 16, 16, 0);
```

غير هذه الأسطر ولاحظ النتيجة. نافذتك الآن ينبغي أن تتوفر على ملف وقائمة خامدة، مرفقة ببنود مصنفة في الأسفل. هذا يعني أن ملف المورد rc. تم تصريفه وربطه بنجاح.

ينبغي للأيقونة الصغيرة (16*16) أن تظهر في الركن العلوي الأيسر من النافذة، وأيضا في شريط المهام، ولما تضغط على Alt+Tab فإن الإصدار الكبير من الأيقونة (32*32) سيظهر في لائحة التطبيقات.

لقد استخدمت الدالة LoadIcon() لتحميل الأيقونة الكبيرة، وهذا شيء منطقي، لأن هذه الدالة تحمل الأيقونة بالقياسات الافتراضية (default) فقط (32*32)، إذن ما العمل لتحميل الأيقونة الصغيرة؟ نحن مضطرون في هذه الحالة لتحميلها على أساس أنها صورة، وهذا من خلال الدالة LoadImage(). فقط كن حذرا من أن موارد وملفات الأيقونات قد تحوي أحيانا عدة صور معا، أما في المثال الذي أرفقته فإن ملف الأيقونة يحوي صورتين فقط.

المثال الثاني: menu_two

الهدف من وراء استخدام مورد القائمة هو إنشاء قائمة أثناء التنفيذ. هذا يستدعي بعض الكود الإضافي، ولكنه يعتبر ضروريا في بعض الحالات.

يمكنك كذلك استخدام الأيقونات الغير مخزنة كموارد، حيث تحفظ الأيقونة كملفات منفصلة، ثم تقوم بتحميلها أثناء وقت التنفيذ. هذا يتيح للمستخدم إمكانية اختيار أيقونات على مزاجه من خلال مربعات حوار أو شيء آخر يؤدي نفس المهمة.

إبدأ مرة أخرى بـ simple_window من دون إضافة h. أو rc.، والآن سنلتقط رسالة WM_CREATE ونضيف قائمة إلى نافذتنا.

```
#define     ID_FILE_EXIT   9001
#define     ID_STUFF_GO    9002
```

ضع هذين السطرين في مقدمة ملفك التنفيذي c. أسفل تعليمات #include (لأننا لا نملك ملف رأس h. أو مورد rc.).

أضف بعدها رسالة WM_CREATE إلى إجراء النافذة، واكتب فيه الكود الآتي:

```
case WM_CREATE:
{
   HMENU hMenu, hSubMenu;
   HICON hIcon, hIconSm;
   hMenu = CreateMenu();
   hSubMenu = CreatePopupMenu();
   AppendMenu(hSubMenu, MF_STRING, ID_FILE_EXIT, "E&xit");
   AppendMenu(hMenu, MF_STRING | MF_POPUP,(UINT)hSubMenu,"&File");
   hSubMenu = CreatePopupMenu();
   AppendMenu(hSubMenu, MF_STRING, ID_STUFF_GO, "&Go");
   AppendMenu(hMenu, MF_STRING|MF_POPUP,(UINT)hSubMenu, "&Stuff");
   SetMenu(hwnd, hMenu);
   hIcon = LoadImage(NULL, "menu_two.ico", IMAGE_ICON,
                     32, 32, LR_LOADFROMFILE);
   if(hIcon)
      SendMessage(hwnd, WM_SETICON, ICON_BIG,(LPARAM)hIcon);
   else
      MessageBox(hwnd, "Could not load large icon!",
                    "Error", MB_OK | MB_ICONERROR);
   hIconSm = LoadImage(NULL, "menu_two.ico", IMAGE_ICON,
                          16, 16, LR_LOADFROMFILE);
   if(hIconSm)
      SendMessage(hwnd, WM_SETICON, ICON_SMALL, (LPARAM)hIconSm);
   else
      MessageBox(hwnd, "Could not load small icon!", "Error",
                    MB_OK | MB_ICONERROR);
}
break;
```

أولا، لا تقلق لضخامة هذا الكود، فقط أمعن النظر فيه وستجد الكثير من التعليمات المكررة.

هذا الكود ينشئ قائمة تماما مثل الذي أنشأناه باستخدام المورد، ثم يقوم بوصله بنافذتنا، هذا النوع من القوائم يتم حذفه مباشرة بعد نهاية البرنامج، إذن فلن نقلق أبدا من عملية التخلص من هذه القائمة، أما لو كنا نريد عمل ذلك يدويا درءا للشك، فإننا سنكون ملزمين عندها بتوظيف GetMenu() و DestroyMenu().

الكود الخاص بالأيقونات غاية في البساطة، حيث نقوم باستدعاء الدالة LoadImage() مرتين، لتحميل الأيقونة بحجم 16*16 ومرة ثانية بحجم 32*32. لا يمكننا استخدام الدالة LoadIcon()، لا للحجم الصغير ولا للكبير، لأنها في هذه الحالة تحمل الموارد ولا تحمل الملف. كما أننا حددنا القيمة NULL لبارامتر مقبض المثيل (Instance Handle) لأننا لا نقوم بتحميل المورد من وحدتنا النمطية (Module)، وبدلا من تمرير معرف (ID) الأيقونة، فإننا مررنا اسم ملف الأيقونة الذي نريد تحميله. ختاما، قمنا بتمرير الراية LR_LOADFROMFILE للإشارة إلى أننا نريد من الدالة أن تتعامل مع سلسلة الحروف التي أعطيناها لها على أنها اسم ملف وليس اسم مورد.

إذا أفلح كل استدعاء، فإننا سنسند مقبض الأيقونة إلى نافذتنا عن طريق WM_SETICON، وإن فشل استدعاء ملف الأيقونة فإن البرنامج يعرض مربع حوار يخبرنا بأن خطأ ما قد حدث.

ملاحظة: الدالة LoadImage() قد تفشل في عملها إذا لم يتواجد ملف الأيقونة في نفس المجلد الخاص بالتطبيق الحالي. إذا كنت تستعمل الفيجول سي++، وأنك تنفذ البرنامج من الواجهة البرمجية، فإن مجلد العمل الحالي سيكون ذاك الذي يتواجد فيه ملف المشروع. أما لو كنت تنفذ البرنامج من مجلد Debug أو Release من مستكشف ويندوز فإنك مطالب بنسخ ملف الأيقونة إلى ذاك المجلد أملا في أن يعثر عليه البرنامج عند الإقلاع. إذا فشل كل ذلك، فما عليك إلا أن تحدد كامل مسار ملف الأيقونة "c:\\path\\to\\icon.ico".

حسنا، الآن نتوفر على قائمتنا، ونريد أن تقوم بنشاط ما. هذا بسيط، كل ما نحتاج إليه هو التقاط مقبض الرسالة WM_COMMAND ↺. كما نحتاج إلى تحديد نوع الأمر وكيفية التعامل معه. إذن إجراء النافذة (window procedure) الخاص بنا يلزمه احتواء ذلك، وسيكون هذا الإجراء على هذه الشاكلة:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT Message, WPARAM wParam,
LPARAM lParam)
{
    switch(Message)
    {
        case WM_CREATE:
        {
            HMENU hMenu, hSubMenu;
            hMenu = CreateMenu();
            hSubMenu = CreatePopupMenu();
            AppendMenu(hSubMenu, MF_STRING, ID_FILE_EXIT,"E&xit");
            AppendMenu(hMenu, MF_STRING | MF_POPUP,(UINT)hSubMenu,
                        "&File");

            hSubMenu = CreatePopupMenu();
            AppendMenu(hSubMenu, MF_STRING, ID_STUFF_GO, "&Go");
            AppendMenu(hMenu, MF_STRING | MF_POPUP,(UINT)hSubMenu,
                        "&Stuff");

            SetMenu(hwnd, hMenu);

            hIcon = LoadImage(NULL, "menu_two.ico", IMAGE_ICON,
                              32, 32, LR_LOADFROMFILE);
            if(hIcon)
                SendMessage(hwnd, WM_SETICON, ICON_BIG,
                              (LPARAM)hIcon);
            else
                MessageBox(hwnd, "Could not load large icon!",
                        "Error", MB_OK | MB_ICONERROR);

            hIconSm = LoadImage(NULL, "menu_two.ico", IMAGE_ICON,
                              16, 16, LR_LOADFROMFILE);
            if(hIconSm)
                SendMessage(hwnd, WM_SETICON, ICON_SMALL,
```

```
            else
                MessageBox(hwnd, "Could not load small icon!",
                           "Error", MB_OK | MB_ICONERROR);
        }
        break;
        case WM_COMMAND:
            switch(LOWORD(wParam))
            {
                case ID_FILE_EXIT:

                break;
                case ID_STUFF_GO:

                break;
            }
        break;
        case WM_CLOSE:
            DestroyWindow(hwnd);
        break;
        case WM_DESTROY:
            PostQuitMessage(0);
        break;
        default:
            return DefWindowProc(hwnd, Message, wParam, lParam);
    }
    return 0;
}
```

كما ترى، فإنه لدينا WM_COMMAND الخاص بنا، والتي تحوي بداخلها حلقة switch فرعية؛ هذه الأخيرة تعتمد على الكلمة الدنيا للبارامتر wParam ↺ (الكلمة الدنيا LOWORD =16بت) كقيمة يتم من خلالها اختيار الحالة المناسبة، والتي كما نعلم تحوي معرف القائمة أو التحكم (control) الذي أرسل الرسالة.

الآن نريد صراحة من البند Exit أن ينهي البرنامج، إذن كود المقبض ID_FILE_EXIT التابع للاختيار المعنون تحت WM_COMMAND يصير على هذا الشكل:

```
PostMessage(hwnd, WM_CLOSE, 0, 0);
```

ومنه، يصير كود الحلقة الفرعية switch التابعة للرسالة WM_COMMAND هكذا:

```
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
            case ID_FILE_EXIT:
                PostMessage(hwnd, WM_CLOSE, 0, 0);//الكود المضاف
            break;
```

```
            break;
        }
    break;
```

أترك الباقي لك. قم بإضافة رسائل وأنشطة إلى بقية بنود القائمة (ID_STUFF_GO مثلا).

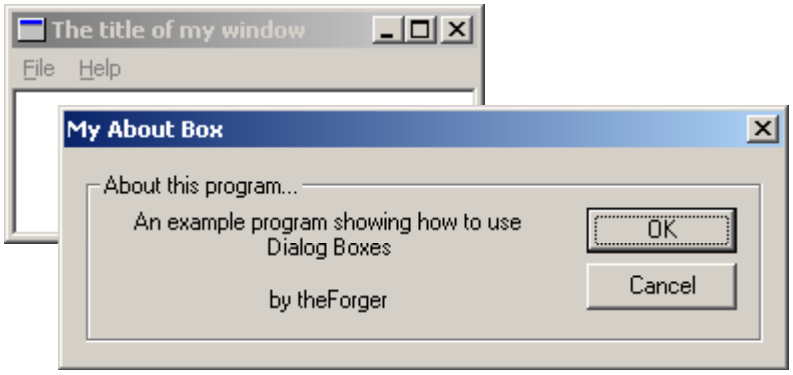
أيقونة ملف البرنامج:

لعلك لاحظت أن الملف menu_one.exe يعرض على شكل الأيقونة التي أضفناها كمورد، أما الملف menu_two.exe فلا يعرض بشكل الأيقونة، بما أننا قمنا بتحميل ملف خارجي. ببساطة، يقوم مستكشف الويندوز بعرض الأيقونة الأولى (اعتمادا على ترقيم المعرفات) في موارد ملفات البرنامج، إذن نحن لا نملك إلا أيقونة واحدة، لهذا تم عرضها. إذن إذا كنت تريد أن يعرض برنامجك التنفيذي على شكل أيقونة، فقم بإضافتها إلى المورد، وزودها برقم تعريف (ID) ذو قيمة دنيا كالرقم 1 مثلا.

لست مضطرا للإشارة إلى الملف في البرنامج، كما يمكنك تحميل العديد من الأيقونات المختلفة إذا اخترت ذلك.

VIII - الحوارات:

المثال المرفق: dlg_one



تكاد تنعدم البرامج التي لا يتم فيها استعمال مربعات الحوار، فقط إختر File->Open أو أي بند آخر، وإذا بك تواجه مربع حوار مميز، هذه الوسيلة التي تمكنك من اختيار الملف المراد فتحه مثلا.

مربعات الحوار (الحوارات اختصارا) ليست محددة بتلك المتعلقة بفتح الملفات القياسية، بل يمكنها أن تؤدي لك ما تريده منها.

الشيء المهم الذي يجذبنا للحوارات، هو أنها تختصر الطريق من أجل تنظيم وإنشاء واجهة المستخدم الرسومية GUI (Graphic User Interface) والعديد من المهام التي من خلالها يمكننا التقليل من الكم الهائل من الكود المطلوب كتابته.

الشيء المهم الذي يجب عليك أن تتذكره هو أن الحوارات ما هي إلا نوافذ. الشيء الذي يميز الحوارات عن النوافذ العادية هو أن النظام يجري بعض المهام الإضافية بخصوص الحوارات، مثلا كإنشاء و ابتداء التحكمات، وتحديد مقبض (handle) لترتيب الجدولة (tab order). تقريبا كل نوافذ الـ API التي تعتمد على نفس هيكل النوافذ العادية، يمكنها ببساطة العمل كحوارات والعكس صحيح.

كيف ننشئ مربع حوار؟

أول خطوة لفعل ذلك هي إنشاء مورد الحوار. وكبقية الموارد، ↺ أسلوب تحقيق ذلك يختلف من مصرف (compiler) لآخر. هنا سأعرض عليك فقط نص الحوار كما ينبغي أن يكون في الملف rc.، وأترك لك مهمة إدراجه في مشروعك.

```
IDD_ABOUT DIALOG DISCARDABLE    0, 0, 239, 66
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "My About Box"
FONT 8, "MS Sans Serif"
BEGIN
      DEFPUSHBUTTON  "&OK",IDOK,174,18,50,14
      PUSHBUTTON     "&Cancel",IDCANCEL,174,35,50,14
      GROUPBOX       "About this program..",IDC_STATIC,7,7,225,52
      CTEXT          "An example program showing how to use Dialog
                Boxes\r\n\r\nby theForger",IDC_STATIC,16,18,144,33
END
```

ماذا يعني هذا الكود؟

في السطر الأول، IDD_ABOUTDLG هو معرف المورد. DIALOG هو نوع المورد (تماما مثل تعريف الأيقونة)، أما الأرقام الأربعة فهي إحداثيات اليسار(Left)، الأعلى(Top)، العرض(Width)

والإرتفاع (Height) على الترتيب. وهذه القيم ليست بوحدة البكسل، بل بوحدة مربع الحوار، والتي تعتمد على حجم الخط المستخدم من قبل النظام (والمختارة من قبل المستخدم). فإذا كنت قد اخترت خط سميك فإن الحوار سيكون سميكا كذلك، وإذا ما اخترت خطا رقيقا، فإن الحوار سيكون كذلك. هذا مهم، لأنه يتيح للتحكمات (الكائنات الموجودة على مربع الحوار) أن تعرض نصوصها بالحجم الجاري. يمكنك تحويل وحدة الحوار إلى البكسل أثناء وقت التنفيذ من خلال الدالة MapDialogRect().

نرجع إلى نص الكود. DISCARDABLE تخبر النظام بأن يبدل ذاكرة المورد بالقرص في حالة عدم استخدامها، وهذا لحفظ موارد النظام.

السطر الثاني يبدأ بالمصطلح STYLE (نمط) ويتبع بنمط النافذة المستخدمة في إنشاء الحوار. هذه يستلزم شرحها في باب CreateWindow() في ملفات المساعدة المتوفرة لديك. في حالة استخدام ثوابت معرفة مسبقا، فإنك ملزم بإدراج ملف الرأس "windows.h" بفضل التعليمة #include في ملف المورد لديك rc.، أو الملف winres.h أو الملف afxres.h في حالة استعمالك للفيجول سي++. لو تستخدم محرر المورد فإن هذه الملفات، لا محالة، سيتم إدراجها آليا.

سطر التسمية (caption) لابد أن يشرح نفسه بنفسه...

سطر الخط يحدد حجم واسم الخط الذي تريد استخدامه لهذا المربع الحواري. هذا لا يتم تماما في كل الحواسيب، على اعتبار أن المستخدمين كثيرا ما يستعملون أنواعا مختلفة من الخطوط ومتفاوتة في الحجم. لكن هذا لا يدعو للارتباك، فلا تقلق بشأنه.

الآن لدينا لائحة (list) من التحكمات يجب إنشاءها على المربع الحواري:

```
DEFPUSHBUTTON      "&OK",IDOK,174,18,50,14
```

هذا هو سطر الزر OK، أما الرمز & في هذه الحالة فيعني الحرف المسطر من تسمية الزر (O) والذي يكفي بالضغط على Alt+O ليتم تحقيق عملية النقر على الزر (تماما كتلك المستعملة مع بنود القائمة). IDOK هو معرف التحكم، وبما أن IDOK هو معرف مسبقا، فإنه لا يلزمنا استخدام الماكرو #define. أما الأرقام الأربعة فهي اليسار، الأعلى، العرض والارتفاع، وكلها بوحدة الحوار (وليس البكسل).

هذه المعلومة ينبغي أن تكون تقليدية، فلأنك دائما تستخدم محرر المورد لإنشاء حوارات، فإنك أحيانا تكون مضطرا لمعرفة الكود الخاص بها، وخاصة إذا لم تكن تتوفر على محرر مورد مرئي.

تحكمان اثنان يملكان معرف من نوع IDC_STATIC (الذي هو -1)، هذا يستخدم للدلالة على أننا لا نريد أبدا أن نصل إليهما، إذن هما لا يحتاجان إلى أي معرف. لكن إضافة معرف (ID) لأي منهما لا يضر بالبرنامج بتاتا، كما أن محرر المورد قد يفعل ذلك بصورة آلية.

"\r\n" لعرض النص أو السطر في وضعيات مختلفة.

إذن، إضافة ذلك إلى ملف المورد rc. يعني أننا نحتاج لكتابة إجراء الحوار الذي يقوم بمعالجة رسالة هذا المربع الحواري. لا تقلق، هذا ليس بالشيء الجديد، عمليا هو شبيه بإجراء نافذتنا الأساسية (لكنه ليس تماما)، لأنه وكما هو معروف، كل نافذة لها إجراء النافذة الخاص بها.

```
BOOL CALLBACK AboutDlgProc(HWND hwnd, UINT Message, WPARAM wParam,
                           LPARAM lParam)
```

```
	switch(Message)
	{
		case WM_INITDIALOG:
		//إجراء عمليات ابتداء عند إنشاء مربع الحوار//
		return TRUE;
		case WM_COMMAND:
			//إختيار الحالة اعتماد على الكلمة الدنيا//
			switch(LOWORD(wParam))
			{
			case IDOK:
				EndDialog(hwnd, IDOK);
			break;
			case IDCANCEL:
				EndDialog(hwnd, IDCANCEL);
			break;
			}
		break;
		default:
		return FALSE;
	}
	return TRUE;
}
```

تلاحظ أن إجراء مربع الحوار شبيه جدا بإجراء نافذتنا الأساسية، إلا أنه توجد بعض الاختلافات البسيطة.

أولا هو أن طلب الدالة DefWindowProc() بالنسبة للرسائل التي لا ترغب في التقاطها، والتي كما ذكرنا سابقا تقوم بمهمة معالجتها بأسلوب مفترض، قلت إن طلب هذه الدالة غير مشروط، لأنه مع الحوارات، هذا يتم بصورة آلية (إلا إذا رغبت في فعل أمور خاصة).

ثانيا، في الغالب، أنت تعيد القيمة FALSE بالنسبة للرسائل التي لم تعالجها، والقيمة TRUE للرسائل التي تعالجها، إلا إذا تم تحديد قيمة خاصة عند العودة. لاحظ أن هذا ما قمنا به. العمل المفترض هو عدم القيام بأي شيء وإعادة القيمة FALSE، بينما الرسائل التي نلتقطها تفصل عمل الحلقة switch وتعيد القيمة TRUE.

ثالثا، أنت لا تطلب الدالة DestroyWindow() لإغلاق الحوار، وإنما تطلب الدالة EndDialog(). والبارامتر الثاني هو القيمة المعادة بعد طلب DialogBox().

أخير، وبدلا من التقاط مقبض الرسالة WM_CREATE فإنك تلتقط مقبض WM_INITDIALOG لإجراء بعض العمليات قبل ظهور مربع الحوار. ثم إعادة القيمة TRUE لأجل وضع لوحة المفاتيح في وضع التحكم الافتراضي. (يمكنك حاليا التقاط مقبض الرسالة WM_CREATE ، لكن هذه الرسالة يتم إرسالها قبل إنشاء أي تحكم، إذن لا يمكنك في ذلك الوقت النفاذ إليها. على عكس الرسالة WM_INITDIALOG التي يتم إرسالها بعد إنشاء التحكمات).

يكفي من القيل والقال، دعنا ننشئ ذلك:

```
case WM_COMMAND:
```

```
  {
     case ID_HELP_ABOUT:
     {
        //التصريح بمتغير صحيح يحفظ قيمة العودة
        int ret = DialogBox(GetModuleHandle(NULL),
                  MAKEINTRESOURCE(IDD_ABOUT), hwnd, AboutDlgProc);
        //إختيار الحالة المناسبة اعتمادا على قيمة العودة
        if(ret == IDOK)
        {
.           MessageBox(hwnd, "Dialog exited with IDOK.", "Notice",
                    MB_OK | MB_ICONINFORMATION);
        }
        else if(ret == IDCANCEL)
        {
            MessageBox(hwnd, "Dialog exited with IDCANCEL.",
                    "Notice", MB_OK | MB_ICONINFORMATION);
        }
        else if(ret == -1)
        {
            MessageBox(hwnd, "Dialog failed!", "Error",
                    MB_OK | MB_ICONINFORMATION);
        }
     }
     break;
    // Other menu commands...
  }
break;
```

هذا هو الكود الذي استخدمته لإنشاء مربع الحوار AboutBox، يمكنك تخمين أن هذا الكود سيكون في مقبض الرسالة WM_COMMAND، إذا لم تستوعب ذلك، فاضغط على هذا الرمز لمراجعة الدرس السابق.↺

معرف مربع الحوار ID_HELP_ABOUT هو معرف بند القائمة Help->About (يعني نفس القيمة الصحيحة)

بما أننا نريد من القائمة التابعة للنافذة الرئيسية أن تنشئ لنا مربع الحوار فبالتأكيد، سنقوم بإدراج الكود في WndProc() في نافذتنا الرئيسية، وليس في WndProc الخاصة بالحوار (بمربع الحوار).

الآن، قمنا بحفظ قيمة العودة من مربع الحوار، والتي تظهر أهميتها عند كثرة التحكمات في الحوار، في مثالنا يوجد زرين فقط، كيف نعرف ما هو الزر الذي ضغط عليه المستخدم، وما النشاط المتأثر بالزر "س" أو الزر "ع"،وماذا لو كان عندنا عدة تحكمات، ما هو دورنا إزاء اختيار المستخدم، يمكن فقط التحكم في كل ذلك من خلال قيمة العودة (return value) من إجراء الحوار.

```
DialogBox(GetModuleHandle(NULL), MAKEINTRESOURCE(IDD_ABOUT), hwnd,
          AboutDlgProc);
```

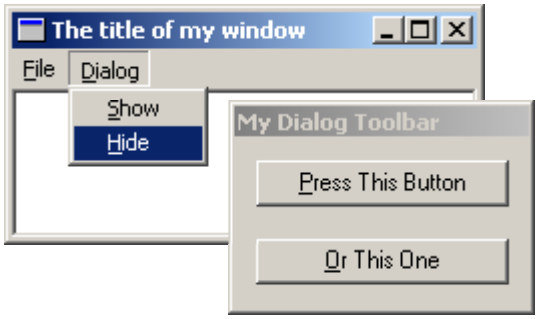
قيمة العودة الناتجة عن هذه الدالة DialogBox() تم حفظها في المتغير الصحيح ret، وهذا هو الجزء الهام في درسنا اليوم، ويمكنك وضعه في أي موضع من برنامجك، حيث يتم طلب الحوار.

IDD_ABOUT هو معرف مورد الحوار. Hwnd هو مقبض النافذة الأب للحوار (الذي هي نافذتنا الأساسية). AboutDlgProc() كما هو معلوم، هي إجراء الحوار المستخدم للتحكم (ntrolco) في مربع الحوار.

المبرمج المتفطن لا بد وأنه تساءل: مادامت DialogBox() لا تعيد أية قيمة إلا بعد إغلاق مربع الحوار، فيكف نعالج الرسائل والمربع الحواري مفتوح؟ حسنا، الشيء الأساسي الواجب تذكره أن DialogBox() في حد ذاته يملك حلقة الرسائل (دوامة الرسائل) الخاصة به، حيث أنه عند ظهور مربع الحوار فإن حلقة الرسائل الخاصة به تغطي الحلقة السابقة، ويصير مقبضها في متناول ويندوز، مما يعني أن أي نشاط يتعلق بهذه النافذة سيتم ترجمته (ترجمة أغلب الرسائل الافتراضية (default) المتعلقة بمربع الحوار) كما أسلفنا الذكر مع أي نافذة قياسية.

إذن عند تفعيل الحوار، سيتم إبطال النافذة الرئيسية، حتى يتم إغلاق الحوار، وهذا ما نريده غالبا، لكننا لا نريده دائما، فأحيانا نريد أن نبحر في كلا النافذتين (النافذة الرئيسية ومربع الحوار)، كما هو الحال مع نوافذ برنامج معالجة الرسومات مثلا، فكيف سيكون الحل؟ هذا هو موضوع فصلنا القادم.

IX - حوارات بلا قالب:



المثال المرفق: dlg_two

الآن سنتعرف على الدالة CreateDialog()، والتي تعتبر شقيقة الدالة DialogBox() التي رأيناها في الدرس السابق. الفرق الجوهري بين الدالتين هو أنه لما تقوم الدالة DialogBox() بتنفيذ حلقة الرسائل الخاصة بها فإنها لا تعود إلا بعد إغلاق الحوار، على عكس الدالة CreateDialog() التي تتعامل مع النظام كنافذة تم إنشاءها بالدالة CreateWindowEx() حيث أنها تعيد مقبض العمل إلى النافذة الرئيسية، ويتم التعامل معها كنافذة من نوافذ ويندوز القياسية. هذا النوع من الحوار يصطلح على تسميته بلا قالب (Modeless)، في حين تقوم الدالة DialogBox() بإنشاء حوارات ذات قالب (Modal).

يمكنك إنشاء مورد الحوار ↺ معتمدا على نفس خطوات الدرس السابق، كما يمكنك أن تعدل في "Tool Window" بحيث تختار الخاصية "النمط الموسع" لإعطاء لشريط العنوان أصغر تسمية ممكنة.

مورد الحوار الذي أنشأته هو كما يلي:

```
IDD_TOOLBAR DIALOGEX 0, 0, 98, 52
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
EXSTYLE WS_EX_TOOLWINDOW
CAPTION "My Dialog Toolbar"
FONT 8, "MS Sans Serif"
BEGIN
      PUSHBUTTON   "&Press This Button",IDC_PRESS,7,7,84,14
      PUSHBUTTON   "&Or This One",IDC_OTHER,7,31,84,14
END
```

لابد وأنك لاحظت أن محرر المورد قد قام بتعويض DIALOG بـ DIALOGEX مما يعني أننا نريد وضع EXSTYLE (النمط الموسع) في الحوار (أي في مربع الحوار).

بعده، ننشئ الحوار أثناء تنفيذ برنامجنا الرئيسي. أردت أن يكون ظاهرا في الحين، إذن الحل هو بوضع ذلك في مقبض الرسالة WM_CREATE. كما نريد أن نلتقط مقبض النافذة المنشأة بالدالة CreateDialog()، إذن لفعل ذلك، قمنا بالتصريح بمتغير شامل يحفظ هذا المقبض، والذي سنستعمله لاحقا.

ملاحظة: DialogBox() (في مثالنا السابق) لا تعيد مقبض النافذة إلا بعد هدم المربع الحواري نهائيا، على عكس CreateDialog() التي تعيد المقبض وتبقى مشيدة.

```
HWND g_hToolbar = NULL;
```

```
case WM_CREATE:
      g_hToolbar = CreateDialog(GetModuleHandle(NULL),
                  MAKEINTRESOURCE(IDD_TOOLBAR), hwnd, ToolDlgProc);
      if(g_hToolbar != NULL)
      {
           ShowWindow(g_hToolbar, SW_SHOW);
      }
      Else
      {
      MessageBox(hwnd, "CreateDialog returned NULL", "Warning!",
                   MB_OK | MB_ICONINFORMATION);
      }
break;
```

نقوم بتفحص القيمة المعادة، هذا يعتبر عملا جيدا، فإذا كانت هذه القيمة سليمة (NULL!) فإننا نقوم بإظهار النافذة بالدالة ShowWindow(). مع الدالة DialogBox() يعتبر الأمر غير ضروري، لماذا؟ لأن النظام يقوم بالمهمة عنا ويستدعي الدالة ShowWindow() لوحده.

الآن النافذة تم عرضها، لكنها لا تعمل، لم نقم بالتقاط الرسائل منها، إذن سنؤدي هذا العمل من خلال إجراء الحوار (لكل نافذة إجراءها الخاص بها) الخاص بحوارنا الذي أسميناه ToolBar:

```
BOOL CALLBACK ToolDlgProc(HWND hwnd, UINT Message, WPARAM wParam,
LPARAM lParam)
{
   switch(Message)
   {
     case WM_COMMAND:
        switch(LOWORD(wParam))
        {
            case IDC_PRESS:
               MessageBox(hwnd, "Hi!", "This is a message",
                          MB_OK | MB_ICONEXCLAMATION);
            break;
            case IDC_OTHER:
               MessageBox(hwnd, "Bye!", "This is also a message",
                          MB_OK | MB_ICONEXCLAMATION);
            break;
        }
        break;
        default:
            return FALSE;
   }
   return TRUE;
}
```

لابد وأنك لاحظت أن أغلب القواعد المطبقة مع مقابض الرسائل المعتمدة مع الدالة CreateDialog() هي تلك المعمول بها مع الدالة DialogBox()، لا تقم بطلب الدالة

DefWindowProc() وإنما اعد القيمة TRUE للرسائل التي تريد التقاطها، والقيمة FALSE لتلك التي لا تريد التقاطها.

تغيير طفيف عن المثال السابق، وهو أننا لا نستدعي الدالة EndDialog() لحوارات بلا قالب، يمكننا بدل ذلك استدعاء الدالة DestroyWindow() تماما كما فعلنا ذلك مع نوافذ ويندوز القياسية. في هذه الحالة قمنا بهدم الحوار لما تم هدم النافذة الرئيسية. هذا يعني أنه يجب التصريح بذلك في إجراء النافذة الرئيسية. إذن في إجراء النافذة الرئيسية نجري هذه التعديلات.

```
case WM_DESTROY:
	DestroyWindow(g_hToolbar);
	PostQuitMessage(0);
break;
```

أخيرا، نود أن نكون قادرين على إظهار وإخفاء الحوار (ToolBar) في أي وقت نريد ذلك، لهذا قمت بإدراج أمرين اثنين إلى القائمة (menu) التي أنشأتها في الدرس السابق، وقمت بتغطيتهما بمقبض الرسالة WM_COMMAND:

```
case WM_COMMAND:
	switch(LOWORD(wParam))
	{
		case ID_DIALOG_SHOW:
			ShowWindow(g_hToolbar, SW_SHOW);
		break;

		case ID_DIALOG_HIDE:
			ShowWindow(g_hToolbar, SW_HIDE);
		break;
		//بقية مقابض بنود القائمة
	}
break;
```

يجب أن تكون قادرا على إنشاء قائمتك الخاصة مستخدما في ذلك محرر المورد أو يدويا (بكتابة الكود)، لكن إن شق عليك ذلك، فيمكنك الإطلاع على المثال الموافق لهذا الدرس (dlg_two) المتوفر مع هذا الدليل.

لما تقوم الآن بتنفيذ البرنامج، فإنك ستكون قادرا على الوصول إلى كل من النافذتين: النافذة الرئيسية ومربع الحوار.

عند هذه النقطة، إذا ما نفذت برنامجك، فإنك ستلاحظ أمرين يهم معرفة سببهما، أولا: لما تضغط على الزر Tab فإن العملية اختيار أزرار الحوار لا تتغير، ويبقى دائما الزر الأول هو المختار، والشيء الثاني هو أنك بالضغط على Alt+O أو Alt+P فإن تنفيذ علية النقر على أي زر لا تعمل (جرب لتفهم). لماذا لا تعمل؟ في حين تم ذلك في المثال السابق مع الدالة DialogBox(). لأن الدالة الأخيرة تقوم بعملها آليا في حلقة الرسائل الخاصة بها، وتقوم بالتقاط الأحداث بصفة افتراضية (default)، أما الدالة CreateDialog() فلا تفعل ذلك. لهذا سنقوم بذلك بأنفسنا من خلال استدعاء الدالة IsDialogMessage() في حلقة الرسائل التي ستقوم بالعمل المفترض لنا.

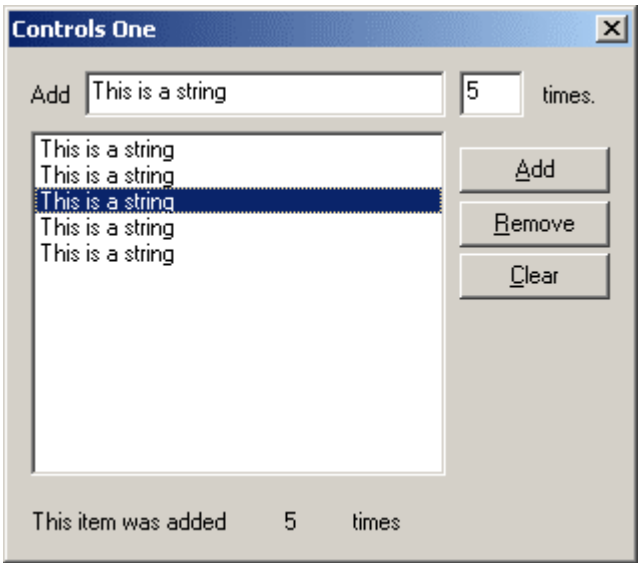
```
while(GetMessage(&Msg, NULL, 0, 0))
{
	if(!IsDialogMessage(g_hToolbar, &Msg))
	{
		TranslateMessage(&Msg);
		DispatchMessage(&Msg);
	}
}
```

في الكود أعلاه، قمنا بتمرير الرسالة إلى الدالة IsDialogMessage()، فإذا كانت الرسالة موجهة إلى الحوار ToolBar (مشار إليه من خلال مقبض الحوار الذي مررناه إلى هذه الدالة)، فإن النظام سيجري العمل الإفتراضي ويعيد القيمة TRUE. في هذه الحالة الرسالة مرفقة بمقبض، إذن لن نستدعي الدوال TranslateMessage() ولا DispatchMessage(). أما إذا كانت الرسالة موجهة إلى نافذة أخرى فإننا نعالجها كالمعتاد.

تجدر الإشارة إلى أنه يمكننا استعمال الدالة IsDialogMessage() أيضا مع النوافذ التي ليست من نوع حوار وهذا في حالة الرغبة في جعلها تسلك سلوك الحوار. فقط تذكر أن كل حوار هو نافذة، وأن أغلب (إن لم تكن كل) حوارات الـ API تعمل على كل النوافذ.

هذا فيم يخص الحوارات بلا قالب، لكن ماذا لو كان عندنا حوار آخر مشتق من نفس الفئة (حواران من نوع ToolBar أو أكثر)، الحل هو أن تتوفر على لائحة مقابض الحوارات، وإجراء عليها عملية بحث باستخدام حلقة مشروطة مثلا ونمرر المقابض (لائحة المقابض) إلى الدالة IsDialogMessage()، حتى نجد المقبض الصحيح، وإلا فإننا نقوم بتتمة المعالجة المعتادة. هذا يعتبر معضلة برمجية عامة، ولا تقتصر فقط على الوين32، ويمكن اعتبار ذلك كتطبيق للقارئ.

X - التحكمات القياسية: أزرار، تحريرات، مربعات سرد:



المثال المرفق: ctl_one

أود الإشارة إلى أنني قد قمت باستخدام الأزرار في الأمثلة السابقة، لهذا يجب أن تتعود عليها، لأنني قمت أيضا بإدراجها في هذا الدرس رفقة بقية التحكمات.

التحكمات:

شيء مهم يجب تذكره، هو أن التحكمات ما هي إلا نوافذ، تماما مثل بقية النوافذ القياسية، لها إجراء النافذة الخاص بها، و فئة النافذة (window class) التي يتم منها الاشتقاق والتي يتم تسجيلها من قبل النظام. أي شيء يمكنك عمله مع النوافذ العادية، يمكنك عمله مع التحكمات. (المثيلات)

الرسائل:

إذا كنت تذكر مناقشتنا السابقة حول حلقة الرسائل، فلقد أشرنا إلى أن ويندوز يتواصل مستعملا الرسائل.. ترسل رسالة إلى النظام حول التحكم لتحسس الأحداث، وعند حصول الحدث (event) المتعلق بهذا التحكم، فإن النظام يرسل رسالة تشعرنا بذلك. بالنسبة للتحكمات القياسية فإن الإشعار يكون من نوع WM_COMMAND كما رأينا ذلك سابقا مع القوائم والأزرار، أما بالنسبة للتحكمات الشائعة والتي سأتناولها لاحقا، فإن الرسائل ستكون من نوع WM_NOTIFY.

الرسائل التي تقوم بإرسالها تتراوح إلى حد ما من تحكم لآخر، وكل تحكم له تشكيلته الخاصة من الرسائل. لكن حينما نلج الحلقة while فإن نفس الرسالة قد تستعمل لأكثر من نوع من التحكمات، لكن عامة، تعمل الرسائل على التحكم الذي صممت له. هذا مزعج خصوصا مع رسائل مربع السرد (listbox) والقائمة المنسدلة (combobox)، أي الرسائل من نوع (LB_* و CB_*)، والتي تتميز بتنفيذها لمهام تكاد تكون متماثلة، ولكنها ليست متبادلة، ولقد قمت صدفة بمزجها وانسجم ذلك أكثر مما كنت أرغب.

من جانب آخر، الرسائل المولدة مثل WM_SETTEXT هي مقبولة تقريبا من طرف كل التحكمات، فالتحكمات ما هي إلا نوافذ ذات صبغة خاصة.

يمكنك إرسال الرسائل باستخدام دالة الـ API ()SendMessage، وتستخدم الدالة ()GetDlgItem للحصول على مقبض التحكم، أو تستخدم الدالة ()SendDlgItemMessage مباشرة، والتي تقوم بالعملين معا. كلا الطريقتين تعطي نفس النتيجة .

التحريرات (Edits):

يعتبر التحرير من أكثر التحكمات المستخدمة في بيئة الويندوز، إذ توظف لتمكين المستخدم من إدخال النص أو تعديله أو إجراء جملة من العمليات المختلفة عليه. مفكرة ويندوز مثلا هي نافذة عادية وقديمة تحوي تحكم تحرير (Edit) في داخلها.

الكود الآتي يستخدم للتواصل مع تحكم تحرير:

```
SetDlgItemText(hwnd, IDC_TEXT, "This is a string");
```

هذه الدالة تعمل على تغيير النص الموجود في التحكم (هذه تستعمل بكثرة مع التحكمات التي تحوي نص في مضمونها، كالتحكمات الساكنة، الأزرار، وهلم جرا...).

استرجاع النص من التحكم عمل سهل.

```
int len = GetWindowTextLength(GetDlgItem(hwnd, IDC_TEXT));
if(len > 0)
	{
	int i;
	char* buf;
	buf = (char*)GlobalAlloc(GPTR, len + 1);
	GetDlgItemText(hwnd, IDC_TEXT, buf, len + 1);

	//... do stuff with text ...
	GlobalFree((HANDLE)buf);
	}
```

أولا، نحتاج لحجز موقع ذاكري (allocation) نخزن سلسلة الحروف فيه، ونريد أن نحصل على المؤشر على السلسلة الرمزية الموجودة في الذاكرة. لعمل ذلك، يجب أن نعرف أولا حجم الذاكرة الواجب حجزه. لا يوجد GetDlgItemTextLength()، ولكن يوجد GetWindowTextLength()ـ إذن كل ما يجب فعله هو الحصول على مقبض التحكم من خلال استعمال الدالة GetDlgItem().↺

الآن وبحصولنا على طول النص، يمكننا حجز حجم الذاكرة المناسب. هنا أضفت عملية تحقيق لمعرفة ما إذا كانت سلسلة الحروف تحوي بعض النص للعمل عليه، لأنه ليس من الأصلح العمل على سلسلة فارغة... أحيانا تستطيع، لكن هذا يتعلق بك.

على افتراض وجود سلسلة حروف سنتعامل معها، إذن نستدعي الدالة GlobalAlloc() لحجز بعض الذاكرة. الدالة GlobalAlloc() هنا هي مكافئة للدالة calloc() في بيئة الدوس/يونيكس. إذ تقوم بحجز بعض الذاكرة، ابتداء (initializing) محتواها بالقيمة المعدومة 0، وتعيد مؤشر على هذا الموقع الذاكري. يوجد العديد من الرايات التي يمكنك تمريرها كأول بارامتر للدالة بحيث يسمح ذلك بأن تسلك سلوكات مختلفة باختلاف الأهداف، ولكن في هذا الدليل سلكت نهجا يظهر نوعا واحدا من الاستخدام.

انتبه إلى أنني قمت بإضافة 1 إلى الطول في موضعين، ما الداعي لذلك؟ حسنا، الدالة GetWindowTextLength() تعيد عدد الرموز الموجودة في تحكم، لكن من دون أخذ البايت الخالي بعين الاعتبار. هذا يعني أنه لو قمنا بحجز موقع ذاكري للسلسلة الرمزية من دون إضافة 1، فإن النص قد ينجح في التواجد في الذاكرة، لكن البايت الخالي في آخر السلسلة يتعدى الحجم المخصص، وقد يتسبب في تخريب البيانات (data)، مسببا نفاذا انتهاكيا أو عمليات أخرى غير مرغوبة. يجب أن تكون حذرا أثناء تعاملك مع أحجام السلاسل الرمزية في الويندوز، فبعض دوال الـ API وبعض الرسائل تعتمد إدراج البايت الخالي في طول النص، والبعض الآخر لا، لهذا يستحسن دائما مطالعة الوثائق الخاصة بهذه الدوال.

لم أتطرق للمميزات الأساسية السلاسل الرمزية، فإذا كنت تريد ذلك، فيستحسن لك أن تطالع كتبا أو مواقع إنترنت تتكلم عن أساسيات لغة السي.

أخيرا نستطيع استدعاء الدالة GetDlgItemText() لاسترجاع محتوى التحكم الموجود في بيفر (buffer) الذاكرة التي قمنا بحجزها. هذا الاستدعاء يعتمد حجم البيفر مع إدراج البايت الخالي. القيمة المعادة التي أهملناها هنا، هي عدد الرموز المنسوخة، لكن من دون البايت الخالي...

الآن، وقد أنجزنا ما نريد مع نصنا (الذي حصلنا عليه منذ قليل)، يجب إذن تحرير الذاكرة المحجوزة من قبل، حتى لا نثقل الذاكرة ونبطئ عمل وحدة المعالجة المركزية. لتحقيق ذلك، يكفينا فقط طلب الدالة GlobalFree() وتمرير مؤشرنا لها.

قد تكون على علم مسبقا أو في المستقبل بالمجموعة الثانية من دوال الـ API، والمعروفة بـ LocalAlloc()، LocalFree() ...إلخ، هذه المجموعة تخص النوافذ من نوع 16 بت.

في بيئة الوين32، تعتبر دوال الذاكرة من نوع Local* و Gloabl* متماثلة (تؤدي نفس المهمة).

تحريرات مع الأرقام:

إدخال النص هو عمل بسيط وجميل، لكن ماذا لو أراد المستخدم إدخال أرقام؟ هذه مهمة تكاد تكون عامة، لكن الـ API تجعل العسير يسيرا، فهي تتولى الحجز الديناميكي وتحويل القيم المدخلة من سلسلة حروف إلى عدد صحيح (integer).

```
BOOL bSuccess;
   int nTimes = GetDlgItemInt(hwnd, IDC_NUMBER, &bSuccess, FALSE);
```

مهمة الدالة GetDlgItemInt() هي نفسها مهمة الدالة السابقة GetDlgItem() إلا أنه وبدلا من نسخ السلسلة الرمزية إلى البيفر (buffer)، فإنها تحولها داخليا إلى قيمة صحيحة (integer)، وتعيد القيمة إليك. البارامتر الثالث يعتبر اختياريا، وهو يتخذ مؤشرا نحو قيمة بوليانية (bool). فبما أن الدالة تعيد القيمة 0 حالة الفشل، فإنه لا يوجد أي دليل يحدد ما إذا كانت الدالة هي التي فشلت وأعادت القيمة 0 أو أن المستخدم هو الذي أدخل هذه القيمة. إذا كنت ترتاح لهذه القيمة المعادة حالة الفشل، فقم بإهمال هذا البارامتر إذن، وإلا فإن استخدامه يزيل هذا الشك.

وسيلة أخرى هامة وهي النمط المعتمد من قبل ES_NUMBER لتحرير التحكمات، والتي تسمح بإدخال الأعداد من 0 إلى 9 فقط. هذا جيد جدا في حالة رغبتك فقط في الحصول على قيم صحيحة موجبة، ولكن من جانب آخر، لا يعتبر ذلك مجديا في حالة الرغبة على الحصول على قيم تحوي إشارة الناقص أو الفاصلة العشرية (النقطة العشرية).

مربعات السرد:

تحكم آخر مفيد، وهو مربع السرد. هذا آخر تحكم قياسي (standard control) سأتعرض له الآن.

إضافة بنود

أول عمل تقوم به مع مربع السرد هو إضافة بنود إليه.

```
int index = SendDlgItemMessage(hwnd, IDC_LIST, LB_ADDSTRING, 0,
                               (LPARAM)"Hi there!");
```

كما ترى، هذه مهمة جد بسيطة. إذا كان مربع السرد مهيأ على نمط LBS_SORT (LBS== ListBoxStyle)، فإن البند الجديد سيتم إضافته حسب الترتيب الأبجدي، أما غير ذلك، فإن البند (item) سيضاف في آخر القائمة.

هذه الرسالة ستعيد فهرس البند الجديد (item index) مهما كانت صفته، ويمكننا استعمال ذلك لتحقيق مهام أخرى بخصوص البند الجديد، كإسناده بعض البيانات. عادة هذا يكون مشابها لمؤشر على هيكل يحوي العديد من المعلومات، أو يكون معرفا (ID) بحيث تستخدمه لتعريف البند. هذا يقتصر عليك.

```
SendDlgItemMessage(hwnd, IDC_LIST, LB_SETITEMDATA, (WPARAM)index,
(LPARAM)nTimes);
```

الإنذارات:

الهدف العام لصندوق القوائم هو تمكين المستخدم من اختيار بنود من القائمة. لكن أحيانا لا نبالي بما يقوم به المستخدم، فكمثال نأخذ الزر Remove، فنحن لا نحتاج لمعرفة متى قام المستخدم بإحداث التغييرات، ولكن متى قام بتفعيل الزر.

أحيانا تريد أن تكون قادرا على فعل أشياء في الحين، مثلا عرض البيانات بشكل مختلف، أو تحديث معلومات ترتكز على البند المختار. لفعل ذلك، نحتاج لالتقاط مقابض رسائل الإشعار التي يرسلها لنا صندوق القوائم. في هذه الحالة فإننا سنهتم بالإشعار LBN_SELCHANGE (LBN== ListBoxNotification)، والتي يخبرنا بأن عملية الاختيار قد تم تغييرها من قبل المستخدم. الاشعار LBN_SELCHANGE يتم إرساله من قبل WM_COMMAND لكن بمقبض يختلف عن تلك المقابض التي تستجيب للنقر، والتي رأيناها مع الأزرار وبنود القائمة. إذن صندوق القوائم يرسل WM_COMMAND لعدة أسباب، ونحن نحتاج لتدقيق ثاني لتحديد الطلب الذي تطلبه منا.

كود الإشعار يتم تمريره ككلمة عليا (HIWORD) للبارامتر wParam، ↺ النصف الثاني من البارامتر الذي يعطينا معرف التحكم (control ID) في المرتبة الأولى.

```
case WM_COMMAND:
	switch(LOWORD(wParam))
	{
		case IDC_LIST:
		// هذا صندوق القوائم الخاص بنا، التدقيق في كود الاشعار
		switch(HIWORD(wParam))
			{
			case LBN_SELCHANGE:
				// القيام بمهام هنا, تعديل الإختيار.
			break;
			}
		break;
		// ... بقية التحكمات
	}
	break;
```

الحصول على بيانات من مربع السرد

الآن، عرفنا أن عملية الاختيار قد تم تعديلها، نحتاج للحصول على الاختيار من مربع السرد وإجراء عمل ما عليه.

في مثالنا هذا، استخدمت الاختيار المتعدد في مربع السرد، يعني الحصول على لائحة (list) تحوي البنود المختارة سيكون خدعة بسيطة. فإذا كان اختيار وحيد في مربع السرد ، فيكفي لذلك أن تقوم بإرسال LB_GETCURSEL للحصول على فهرس البند.

أولا نحتاج للحصول على عدد البنود المختارة، إذن سنحجز بيفر (buffer) في الذاكرة يحفظ هذا الفهارس.

```
	HWND hList = GetDlgItem(hwnd, IDC_LIST);
	int count = SendMessage(hList, LB_GETSELCOUNT, 0, 0);
```

في نفس الوقت نحجز بيفر (buffer) يستند إلى عدد البنود المختارة، ثم يرسل LB_GETSELITEMS لملئها في المصفوفة (array).

```
int *buf = GlobalAlloc(GPTR, sizeof(int) * count);
SendMessage(hList, LB_GETSELITEMS, (WPARAM)count,(LPARAM)buf);

//...القيام ببعض العمليات مع الفهارس التي تم اختيارها

GlobalFree(buf);
```

في هذا المثال، buf[0] هو الفهرس الأول، وهكذا مع بقية الفهارس buf[count-1].

من بين المهم التي تريد غالبا القيام بها هو إيجاد البيانات (أي القيم المختارة) المرفقة مع كل فهرس، وإجراء عليها بعض العمليات. هذه مهمة بسيطة وتقليدية، يلزمنا فقط إرسال رسالة أخرى.

```
int data = SendMessage(hList, LB_GETITEMDATA, (WPARAM)index, 0);
```

إذا كانت البيانات المحصل عليها ذات قيمة من نوع آخر (أي شيء بحجم 32 بت) يجب عليك فقط إجراء التحويل القسري (casting) للبيانات إلى النوع المحدد. كمثال، إذا قمت بحفظ HBITMAPS بدلا من النوع الصحيح.

```
HBITMAP hData = (HBITMAP) SendMessage(hList, LB_GETITEMDATA,
                                      (WPARAM)index,0);
```

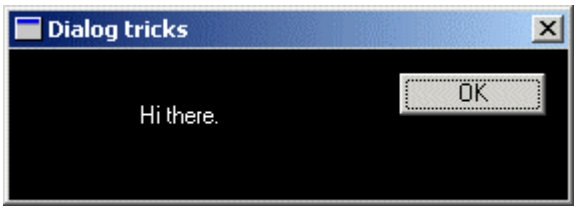
التحكمات الساكنة:

التحكمات الساكنة لا تعتبر هامة في الغالب، لكن من باب أن تكون هذه الدروس شاملة أو لأغراض أخرى معينة، فإني قد قمت بإدراجها هنا. وصفت بأنها ساكنة لأنها لا تقوم إلا بعرض النصوص للمستخدم. مع ذلك، تستطيع أن تجعلها أكثر استخداما من خلال إسنادها معرف وحيد (unique ID)، (الفيجول سي ++ يسند لها فرضا المعرف ICD_STATIC، والذي هو في حد ذاته القيمة –1، أي أنها تعني "لا معرف- No ID")، وبالتالي جعل النص وقت التنفيذ (at RunTime) يعرض البيانات الديناميكية للمستخدم.

في المثال الآتي، استخدمت إحداها لعرض بيانات البند المختار من صندوق القوائم، معتبرا في ذلك أنه مجرد بند وحيد فقط تم اختياره.

```
SetDlgItemInt(hwnd, IDC_SHOWCOUNT, data, FALSE);
```

XI - في الحوار أكثر الأسئلة شيوعا:



المثال المرفق dlg-three

كثيرا ما طرحت أسئلة تتعلق بهذا الموضوع، لهذا ارتأيت إدراجها هنا لتعم الفائدة.

تغيير الألوان:

بصفة عامة، تريد أن تضع وصلة في مربع الحوار، أو أي شيء مماثل، لأن برنامجك إذا ما تم إرفاقه بباقة من الألوان فإنه سيصير سيئا وصعب المنال في أعين المستخدمين، لكن هذا لم يمنع العديد من المبرمجين من القيام بذلك، ولهم في ذلك دوافع مقبولة.

يقوم ويندوز بإرسال مجموعة من الرسائل المتعلقة بالألوان إلى إجراء نافذتك، ↺ وبالتقاطك لمقابض هذه الرسائل، فإنك قد تتمكن من تغيير ألوان الأشياء المعروضة. كمثال، لتغيير لون مربع الحوار نفسه، فإنه يمكنك التقاط الرسالة WM_CTLCOLORDLG، أما لو كنت تريد تغيير لون تحكم ساكن فإنك مطالب بالتقاط الرسالة WM_CTLCOLORSTATIC وهلم جرا...

أولا يمكنك إنشاء فرشاة لتلوين الخلفية ثم تحفظها للاستعمال لاحقا. الرسالة WM_CTLCOLORDLG وبقية الرسائل المتعلقة بها يتم استدعاءها أثناء عمل البرنامج في كثير من الأحيان، وإذا ما قمت بإنشاء فرشاة عند كل عملية فإن الذاكرة سيتم إغراقها بهذا النوع من المتغيرات. بانتهاجنا للأسلوب الأول فإننا سنتحكم أكثر في برنامجنا، ويمكننا حذف الفرشاة لما يتم هدم الحوار حيث أننا لن نحتاجها في ذلك الوقت.

```
HBRUSH g_hbrBackground = CreateSolidBrush(RGB(0, 0, 0));
```

```
case WM_CTLCOLORDLG:
      return (LONG)g_hbrBackground;
case WM_CTLCOLORSTATIC:
    {
      HDC hdcStatic = (HDC)wParam;
      SetTextColor(hdcStatic, RGB(255, 255, 255));
      SetBkMode(hdcStatic, TRANSPARENT);
      return (LONG)g_hbrBackground;
    }
    break;
```

انتبه إلى السطر الذي يجعل الخلفية في شكل شفاف... إذا أبطلت مفعول هذا السطر، فإن الخلفية سيتم ملئها بالفرشاة التي تحددها أنت، لكن لما يقوم التحكم بكتابة النص فإنه سيكتبه

بلون الخلفية... إذن جعل النص يكتب في الوضع الشفاف يحل هذه الإشكالية. الاختيار الآخر هو القيام بعمل لون الخلفية SetBkColor() بنفس لون الفرشاة، لكنني أفضل هذا الحل أكثر.

تغيير الألوان لأغلب التحكمات القياسية الأخرى يجرى بنفس الوتيرة، فقط أنظر إلى الرسالة من نوع WM_CTLCOLOR* في دليل الوين32. لاحظ أن تحكم التحرير (edit) يرسل رسالة من نوع WM_COLORSTATIC إذا كان للقراءة فقط (Read Only)، ومن نوع WM_CTLCOLOREDIT إذا لم يكن كذلك.

إذا كان لديك أكثر من تحكم ساكن والذي تود تغيير لونه فإنك مطالب بالتدقيق في المعرف (ID) الخاص به أثناء إرساله للرسالة، لأنك ستقوم بتغيير لونه اعتمادا على هذا المعرف، حيث أنك تمرر HWND الخاص بالتحكم في البارامتر lParam، وتحصل على معرف التحكم من خلال استخدام الدالة GetDlgCtrlID(). لا تنسى أن التحكمات الساكنة يتم تعريفها فرضا بالمعرف IDC_STATIC (أي القيمة -1) من قبل محرر المورد، ↺ إذن إذا كنت ترغب في تصنيف بعضها عن بعض، فأنت مطالب في هذه الحالة بأن تخصص لها معرفات جديدة ووحيدة.

تزويد الحوار بأيقونة

مهمة بسيطة جدا، تحتاج فقط لإرسال الرسالة WM_SETICON لحوارك. بما أن ويندوز يستعمل أيقونتين معا، إذن أنت ملزم باستدعاءهما مرتين، واحدة للأيقونة الصغيرة المعروضة في الركن الأيسر العلوي، وأخرى للأيقونة الكبيرة التي تظهر لما يتم الضغط على Alt+Tab. يمكنك فقط إرسال نفس المقبض لكل منهما إلا في حالة توفرك على أيقونتين مختلفتين في الحجم.

لإعداد أيقونة (icon) البرنامج الافتراضية (default) فقط، فإنه بإمكانك استعمال الكود الآتي:

```
SendMessage(hwnd, WM_SETICON, ICON_SMALL, (LPARAM)LoadIcon(NULL,
                  MAKEINTRESOURCE(IDI_APPLICATION)));

SendMessage(hwnd, WM_SETICON, ICON_BIG, (LPARAM)LoadIcon(NULL,
                  MAKEINTRESOURCE(IDI_APPLICATION)));
```

لما تبدل مورد أيقونتك لتكون افتراضية، فلا تنسى بأن تغير البارامتر HINSTANCE ↺ للدالة LoadIcon() ↺ بمقبض (handle) التطبيق الخاص بك (والذي يمكنك الحصول عليه باستدعائك للدالة GetModuleHandle(NULL) ↺ إذا لم تقم بحفظها في الدالة (WinMain()). ↺

لماذا لا تعمل القائمة المنسدلة الذي أنشأتها؟

مشكلة شائعة بين أغلب المبرمجين الذين يقومون بإدراج قوائم منسدلة في حواراتهم، لكنهم يتفاجأون بأنها لا تعرض المحتويات لما ينقرون على السهم المشير إلى الأسفل (زر الإفلات). هذه المشكلة مفهومة، بما أن الحل ليس نشاطا حدسيا.

لما تنشئ قائمة منسدلة (combo box) وتقوم بتحديد الارتفاع الخاص به، فإنك بذلك تعطي ارتفاعا لكل المربع بما فيه المحتوى الناتج عن down-drop، وليس ارتفاع التحكم لما يتم إطباقه، والذي يتم تحديده من قبل النظام آخذا في الاعتبار حجم الخط المستخدم.

كمثال، لو قمت بإعطاء للتحكم ارتفاعا بقدر 100 بكسل، فإن النظام يحجم التحكم نفسه بالقيمة الافتراضية (لنقل 30 في هذه الحالة)، ولما تقوم بالنقر على السهم السفلي، فإن اللائحة المنسدلة ستكون بحجم 70 بكسل، أي يكون حاصل الارتفاع 30+70 = 100 بكسل.

إذا كنت تستخدم محرر الموارد الخاص بالفيجول سي++ لوضع مربع سرد (list box) في الحوار، فإنه سيتم إعلامك بعدم قدرتك على تحجيمه عموديا (شاقوليا). إلا إذا نقرت على السهم في المحرر، وسيتغير محور المستطيل ليشير إلى أنك تقوم بتحجيم اللائحة المنسدلة، ويمكنك أثنائها بأن تثبت الارتفاع بأي قياس تريد.

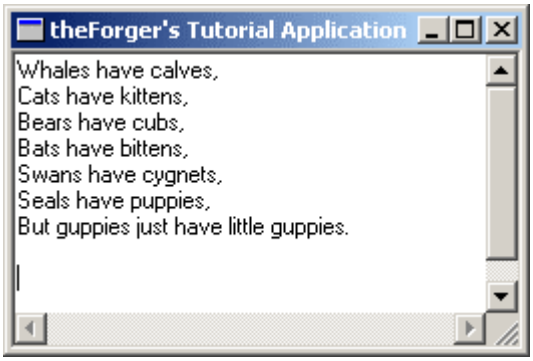
ماذا بخصوص بقية التحكمات الأخرى؟

كان يجب علي أن أقوم بإعطاء مثال عن كل تحكم، لكن لأجل هذا وجدت MSDN و Petzold، إذا لم تستوعب طريقة استخدامها، فإنه يستحسن لك أن تعيد قراءة بعضا من أجزاء هذا الدليل أو كله (وهذا هو الأحسن)، أو أن تحصل على كتب تتعمق في شرحها.

كان من الأحسن أن أزودك بعنوان كل موضوع في مكتبة MSDN، لكن نظرا لتغير بعض المحتويات دوريا، ونظرا لتوقف بعضها أحيانا، فإنني أكتفي بتوجيهك للموقع الرسمي للمكتبة MSDN - Windows Controls ، وفيه يمكنك الإبحار في الموقع، فقط يجب أن تكون ملما أكثر بأساليب البحث عن المواضيع، وأيضا من خلال التبويبات كخدمات واجهة المستخدم User Interface Services، وتحكمات الويندوز Windows Controls، غالبا في جناح Platform SDK.

إنشاء تطبيق بسيط:

I - الجزء الأول: إنشاء تحكمات وقت التنفيذ



المثال المرفق: app_one

هذه الصفحة ستغطي خطوتنا الأولى، والتي ستكون ببساطة عملية إنشاء نافذة وتحكم تحرير (Edit control) ↺ الذي سيكون في مركز نافذتنا.

ستكون بدايتنا مع برنامجنا باختيار لتطبيق بسيط يحوي في مقدمته التعليمة #define للتصريح بمعرف تحكمنا (control ID)، وبإدراج لمقبضي رسالتين يتم وضعهما على مستوى إجراء النافذة.

```
#define IDC_MAIN_EDIT     101
```

```
    case WM_CREATE:
    {
        HFONT hfDefault;
        HWND hEdit;

        hEdit = CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", "",
             WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL|
             ES_MULTILINE | ES_AUTOVSCROLL | ES_AUTOHSCROLL,
             0, 0, 100, 100, hwnd, (HMENU)IDC_MAIN_EDIT,
             GetModuleHandle(NULL), NULL);
        if(hEdit == NULL)
             MessageBox(hwnd, "Could not create edit box.", "Error",
             MB_OK | MB_ICONERROR);

        hfDefault = GetStockObject(DEFAULT_GUI_FONT);
        SendMessage(hEdit, WM_SETFONT, (WPARAM)hfDefault,
                    MAKELPARAM(FALSE, 0));
    }
    break;
    case WM_SIZE:
    {
        HWND hEdit;
        RECT rcClient;

        GetClientRect(hwnd, &rcClient);
```

```
        hEdit = GetDlgItem(hwnd, IDC_MAIN_EDIT);
        SetWindowPos(hEdit, NULL, 0, 0, rcClient.right,
                     rcClient.bottom, SWP_NOZORDER);
    }
    break;
```

إنشاء تحكمات:

عملية إنشاء التحكمات تشبه إلى حد كبير عملية إنشاء النوافذ، فاستخدام الدالة CreateWindowEx() وارد في هذه الحالة. نمرر فئتنا المسجلة مسبقا، في مثالنا هذا ستكون فئة (class) تحكم التحرير (EDIT)، وسنحصل على نافذة تحكم تحرير قياسية. عندما نستخدم الحوارات لإنشاء تحكماتنا، فنحن في الحقيقة نكتب جملة من التحكمات الواجب إنشاءها، ثم نستدعي الدالة DialogBox() أو CreateDialog()، فيقوم النظام بقراءة لائحة التحكمات في مورد الحوار وبالتالي يستدعي الدالة CreateWindowEx() لكل تحكم، مع المحافظة على الموقع و النمط الذي تم التصريح بهما في المورد.

```
        hEdit = CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", "",
                    WS_CHILD | WS_VISIBLE | WS_VSCROLL |
                    WS_HSCROLL | ES_MULTILINE |ES_AUTOVSCROLL |
                    ES_AUTOHSCROLL,0, 0, 100, 100, hwnd,
                    (HMENU)IDC_MAIN_EDIT, GetModuleHandle(NULL),
                    NULL);
        if(hEdit == NULL)
             MessageBox(hwnd, "Could not create edit box.","Error",
                    MB_OK | MB_ICONERROR);
```

لعلك لاحظت أن هذا الاستدعاء للدالة CreateWindowEx() قد خصها بالعديد من الأنماط، وأنه ليس غريبا أن نحدد أكثر من ذلك، وخاصة للتحكمات الشائعة الغنية بالأنماط والاختيارات. الأنماط الأربعة WS_* يجب أن تكون واضحة، فنحن نقوم بإنشاء تحكمنا على أساس أنه ابن لنافذتنا الأم، فنحن نريده أن يكون ظاهرا، ويملك شريطي تمرير، أفقي وعمودي.

ثلاثة أنماط مخصصة لتحكمات EDIT (ES_MULTILINE|ES_AUTOVSCROLL|ES_AUTOHSCROLL) والتي تصف هيئة التحكم، في مثالنا يجب على تحكم EDIT أن يحوي أسطر نصوص متعددة، مع إظهار شريطي التمرير الأفقي والعمودي بصفة آلية، إعتمادا على محتوى التحكم.

أنماط النافذة القياسية (WS_*) تجدها هنا، وأنماط النافذة الموسعة (WS_EX_*) تراها أسفل المرجع CreateWindowEx() في مكتبة الـ MSDN، وفي هذا الموقع كذلك، ستجد وصلات نحو أنماط خاصة بكل تحكم (ES_* في مثالنا تتعلق بتحكمات التحرير).

لقد وصفنا مقبض نافذتنا بأنه أب (parent) للتحكم الذي أنشأناه، وأسندنا له معرف من نوع IDC_MAIN_EDIT والذي سنستخدمه لاحقا للإشارة إلى التحكم، تماما كما يحدث لو أنك قمت بإنشاء هذا التحكم في الحوار (مربع الحوار). بارامترات الموقع والحجم لا تهمنا في هذا الوقت، لأننا سنقوم بتحديد ذلك ديناميكيا عبر الرسالة WM_SIZE، والتي تقوم بصورة مستمرة بتكييف نافذتنا مع الوضع السائد.

تصنيف التحكمات المنشأة ديناميكيا حسب الحجم

عامة، إذا كانت نافذتك قابلة لتعديل الحجم فإنك في هذه الحالة ستكون مضطرا لإدراج كود يسمح بتعديل أحجام التحكمات المنبسطة على مساحتها.

```
GetClientRect(hwnd, &rcClient);

        hEdit = GetDlgItem(hwnd, IDC_MAIN_EDIT);
        SetWindowPos(hEdit, NULL, 0, 0, rcClient.right,
                     rcClient.bottom, SWP_NOZORDER);
```

بما أننا نملك تحكما واحدا فقط، فإن المهمة ستكون نوعا ما يسيرة. سنستخدم لذلك الدالة GetClientRect() التي ستزودنا بأبعاد منطقة العميل لنافذتنا (المساحة الخاصة بالنافذة من دون إدراج القوائم والتسميات والحواشي) هذه الدالة ستملأ البيانات (data) في الهيكل RECT، القيمتان left و top ستحويان دائما 0، لهذا يمكن تجاهلهما، في حين ستكون القيمتان right و bottom مزودتين بعرض وارتفاع منطقة العميل.

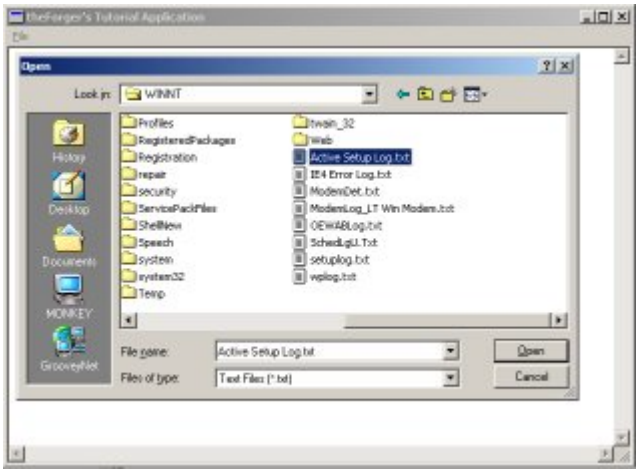
بعدها نستخلص مقبض تحكمنا (تحكم edit) من خلال الدالة GetDlgItem() ↺ والتي تعمل نفس الشيء، سواء مع النوافذ القياسية أو مع الحوارات، ثم نستدعي الدالة SetWindowPos() لتحريك وتحجيم التحكم حتى يناسب كامل منطقة العميل (Client Area). طبعا، يمكنك تغيير القيم الممررة إلى الدالة SetWindowPos() للتمكن مثلا من حجز فقط نصف المساحة (عندما نمرر نصف قيمة الارتفاع) حتى يمكننا استغلال المساحة الثانية لإضافة تحكمات أخرى مثلا.

إنشاء تحكمات أخرى وقت التنفيذ

لن أقوم في هذه الحالة بتقديم أمثلة لكيفية إنشاء تحكمات أخرى ديناميكيا كالأزرار و مربعات سرد...، لأنه وببساطة ستعتمد على نفس الخطوات التي اتبعناها مع تحكم التحرير، لكن إذا ما راجعت الوصلة التي أشرت إليها سابقا والخاصة بمكتبة الـ MSDN أو حتى في المراجع الخاصة بدوال الوين32 فإنك لا محالة ستحصل على المعلومات اللازمة لإنشاء جملة من التحكمات القياسية.

سنتطرق في الفصلين الآتيين إلى نفس العملية مع التحكمات الشائعة (common controls)، إذن كن مستعدا لتطبيق الأمثلة التي ستصادفها.

II - استخدام الملفات والحوارات الشائعة:



المثال المرفق: app_two

حوارات الملفات الشائعة

الخطوة الأولى لفتح أو حفظ ملف ما هو الحصول على اسم الملف المراد استخدامه... طبعا، يمكن إدراج الاسم مباشرة ضمن كود البرنامج، لكن هذا في الواقع غير مستعمل بكثرة، كما أنه يقلل من قيمة البرنامج ويجعله أقل مرونة.

بما أن هذه المهمة تكاد تكون مشتركة، فإنه وجدت حوارات معرفة مسبقا تمكن المستخدم من اختيار اسم الملف الذي يحلو له. أغلب الحوارات المشهورة هي تلك المتعلقة بفتح وحفظ الملفات، ويمكن الوصول إليها من خلال الدالتين GetOpenFileName() و GetSaveFileName() على الترتيب، كلتا الدالتين تتخذ الهيكل OPENFILENAME كبارامتر لها.

```
OPENFILENAME ofn;
char szFileName[MAX_PATH] = "";

ZeroMemory(&ofn, sizeof(ofn));

ofn.lStructSize = sizeof(ofn); // SEE NOTE BELOW
ofn.hwndOwner = hwnd;
ofn.lpstrFilter = "Text Files(*.txt)\0*.txt\0All Files(*.*)\
                   0*.*\0";
ofn.lpstrFile = szFileName;
ofn.nMaxFile = MAX_PATH;
ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST | OFN_HIDEREADONLY;
ofn.lpstrDefExt = "txt";

if(GetOpenFileName(&ofn))
{
  // Do something usefull with the filename stored in szFileName
}
```

لاحظ أننا قمنا باستدعاء الدالة ZeroMemory() من أجل الابتداء (الاستهلال) بالقيمة 0. هذا أسلوب برمجي جيد وينصح باعتماده، لأنه كما هو معروف عن أغلب دوال الـ API أنها دقيقة اتجاه أعضاء الهياكل الغير مبتدئة بالقيمة المعدومة (0)، فبهذا الأسلوب تكون غير ملزم بابتداء قيم الهيكل كل على حدى.

يمكنك مطالعة الوثائق أو ملفات المساعدة لمعرفة مقصد كل عضو (member) من أعضاء الهيكل. فالعضو lpstrFilter يؤشر على سلسلة حروف تنتهي ببايت خالي مضاعف، ويمكنك ملاحظة ذلك في المثال من خلال تواجد العديد من '\0' في كل جزء من السلسلة، بما في ذلك نهايتها... يقوم بعدها المصرف (compiler) بإدراج بايت خالي آخر في نهاية السلسلة كما يقوم بذلك مع كل

السلال الثابتة (والتي لست مضطرا لإضافة البايت الخالي إليها بنفسك). البايتات الخالية في هذه السلسلة تقسمها إلى أقسام في المرشحات (Filters). وكل قسم مركب من جزأين، الجزء الأول يمثل وصف الملف: "Text Files (*.txt)"(لاحظ أن التوسع في هذا الوصف ليس مشروطا بل من أجل إضافة بعض التوضيح) أما الجزء الثاني فهو يخص توسع هذا النوع، والذي يسمونه كذلك بالـ wildcard ، فللترشيح الأول استعملنا "*.txt". كذلك قمنا بنفس العمل مع الترشيح الثاني، لكن مع استثناء بسيط وهو أن التوسع يخص كل أنواع الملفات. يمكنك إضافة مرشحات أخرى لكن مع المحافظة على أسلوب كتابتها.

المؤشر lpstrFile يؤشر على البفر (buffer) الذي حجزناه لنحفظ فيه اسم الملف، وبما أن هذا الأخير لا يمكن له أن يتعدى مقدار MAX_PATH ↺ فإن هذا المقدار هو الذي قمنا باستخدامه أثناء الحجز الذاكري (dynamic allocation).

الرايات تعني أن الحوار يسمح فقط للمستخدم بأن يفتح الملفات الموجودة حقا، ولا يمكنه إنشاء ملفات جديدة، كما أن الملفات الموجودة في وضع القراءة فقط سيتم إخفاءها، لأننا لن نتعامل معها. أخيرا نقوم بتجهيز التوسع المعرف فرضا، فإذا قام المستخدم مثلا بكتابة الاسم "Test" ولم يكن موجودا، فإنه سيتم إدراج التوسع المعرف فرضا وبعدها يبحث البرنامج عن الملف "Test.txt".

لاختيار عملية حفظ ملف بدل من فتحه، فإن الكود المبرمج لذلك يكاد يكون مماثلا، فقط يتم طلب الدالة GetSaveFileName() كما نحتاج إلى تعديل الراية (flag) للحصول على اختيارات أكثر ملاءمة لعملية الحفظ، كأن لا يشترط تواجد الملف من قبل.

```
ofn.Flags = OFN_EXPLORER | OFN_PATHMUSTEXIST | OFN_HIDEREADONLY |
            OFN_OVERWRITEPROMPT;
```

في هذه الحالة والتي تخص عملية الحفظ، فنحن غير مجبرين على أن نكون متوفرين على ملف من قبل كما كان الحال مع عملية الفتح، لكننا مجبرين على التوفر على مسار الخاص بالملف بما فيه آخر مجلد، والذي لم نقم بإنشائه من قبل. كما أننا نستحث المستخدم إذا ما قام باختيار ملف موجود مسبقا حتى تتم عملية الكتابة فوقه (OverWriting)..

قراءة وكتابة ملفات

مع نظام الويندوز ستجد العديد من ميزات القراءة من الملفات والكتابة فيها، وهذا حسب المكتبة المستخدمة، فبإدراجنا للملف "io.h" يمكنك استخدام read() و write()، أما بإدراجك للملف "stdio.h" فإنك ستكون قادرا على توظيف الدوال fopen() ، fread() و fwrite()، أما لو كنت تعمل على بساط السي++، فيمكنك عندها استخدام الدوال المتوفرة في الملف "iostream.h".

على كل، في بيئة الوين32، يمكنك استدعاء هذه الدوال أو أن تقوم بالعمل بالدوال المتوفرة والمحسنة. فلفتح ملف، يمكنك استخدام الدالة OpenFile() أو الدالة CreateFile(). الدالة الأخيرة متفوقة أكثر وغنية بالتحكمات لأجل فتح ملف ما.

القراءة

لنفرض مثلا أنك أعطيت للمستخدم إمكانية فتح الملف مستعملا الدالة GetOpenFileNAme()

```
BOOL LoadTextFileToEdit(HWND hEdit, LPCTSTR pszFileName)
{
	HANDLE hFile;
	BOOL bSuccess = FALSE;

	Hfile= CreateFile(pszFileName, GENERIC_READ,FILE_SHARE_READ,
			NULL, OPEN_EXISTING, 0, NULL);
	if(hFile != INVALID_HANDLE_VALUE)
	{
		DWORD dwFileSize;

		dwFileSize = GetFileSize(hFile, NULL);
		if(dwFileSize != 0xFFFFFFFF)
		{
			LPSTR pszFileText;
			PszFileText = GlobalAlloc(GPTR, dwFileSize +1);
			if(pszFileText != NULL)
			{
			    DWORD dwRead;
			    if(ReadFile(hFile, pszFileText, dwFileSize,
					&dwRead, NULL))
			      {
				  // Add null terminator
				  pszFileText[dwFileSize] = 0;
				  if(SetWindowText(hEdit, pszFileText))
					bSuccess = TRUE;//It worked!
			      }
			    GlobalFree(pszFileText);
			}
		}
		CloseHandle(hFile);
	}
	return bSuccess;
}
```

يوجد دالة كاملة لقراءة ملف نصي في تحكم تحرير (Edit). وتتخذ كبارمترات لها مقبض تحكم التحرير واسم الملف المراد قراءته. هذه الدالة تمتلك أسلوبا جميلا للتدقيق في الأخطاء، وعملية قراءة أو كتابة بيانات ملف ما ليس بالشيء الهين، لهذا فمعرفة نوعية الأخطاء وتصليحها هو أمر في غاية الأهمية.

ملاحظة: المتغير dwRead لم نستعمله إلا لأن الدالة ReadFile() تشترط وجوده، ومن دونه سيفشل استدعاء هذه الدالة.

المتغير GENERIC_READ في استدعائنا للدالة CreateFiel() يعني أننا نريد فقط الوصول إلى عملية القراءة. FILE_SHARE_READ تعني أنه يمكننا مشاركة برامج أخرى في نفس الملف أثناء عملية القراءة فقط، ولا نريد من هذه البرامج أن تقوم بعملية الكتابة لما نكون نحن نمارس عملية القراءة. و OPEN_EXISTING تعني أنه يتم فتح الملف فقط إذا كان موجودا، ولن يتم بذلك إنشاءه إذا لم يكن.

بعد قيامنا بعملية الفتح، وتأكدنا من أن الدالة CreateFile() نجحت في مهمتها، فإننا سندقق في حجم الملف لنحجز مساحة ذاكرية تكفي له، فقد نرغب في قراءة كامل محتوى الملف دفعة واحدة. إذن نقوم بعملية حجز الذاكرة، كما سنتأكد من أن عملية الحجز تمت بنجاح، ثم نستدعي بعدها الدالة ReadFile() لتحميل محتوى الملف من القرص إلى البيفر الذاكري (buffer) الذي خصصناه لذلك.

دوال الـAPI الخاصة بالملفات ليس لها فكرة حول الملفات النصية، فهي لن تقوم بقراءة سطر من النص أو إضافة البايت الخالي في نهاية السلسلة الرمزية. لهذا السبب قمنا بحجز بايت إضافي وبعد قراءتنا للملف قمنا بإدراج البايت الخالي بأنفسنا، إذن يمكننا بذلك تمرير البيفر إلى الدالة SetWindowText() على أنه سلسلة حروف.

إذا نجح كل شيء فإننا نقوم بإسناد القيمة TRUE للمتغير bSeccess ثم نحرر الذاكرة المحتجزة ونغلق الملف المفتوح قبل أن نعود إلى الدالة التي قامت باستدعائنا.

الكتابة

```
BOOL SaveTextFileFromEdit(HWND hEdit, LPCTSTR pszFileName)
{
    HANDLE hFile;
    BOOL bSuccess = FALSE;

    hFile = CreateFile(pszFileName, GENERIC_WRITE, 0, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if(hFile != INVALID_HANDLE_VALUE)
    {
        DWORD dwTextLength;

        dwTextLength = GetWindowTextLength(hEdit);
        // No need to bother if there's no text.
        if(dwTextLength > 0)
        {
            LPSTR pszText;
            DWORD dwBufferSize = dwTextLength + 1;

            pszText = GlobalAlloc(GPTR, dwBufferSize);
            if(pszText != NULL)
            {
                if(GetWindowText(hEdit, pszText, dwBufferSize))
                {
                    DWORD dwWritten;

                    if(WriteFile(hFile, pszText, dwTextLength,
                                    &dwWritten, NULL))
                        bSuccess = TRUE;
                }
                GlobalFree(pszText);
            }
        }
        CloseHandle(hFile);
```

```
    return bSuccess;
}
```

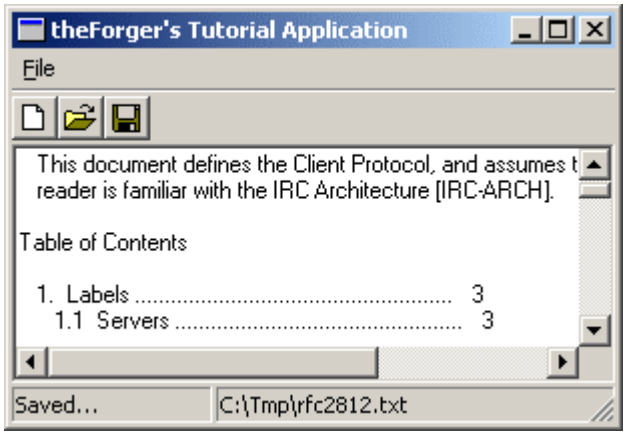
هذا الكود مماثل تمام لكودنا السابق المعتمد في عملية القراءة، دالة الكتابة في الملفات تحوي بعض التعديلات. قبل كل شيء، لما نقوم باستدعاء للدالة CreateFile() فإننا نعني أننا نريد الكتابة في الملف، ويعني ذلك أن الملف سيصير جديدا (وإذا كان موجود مسبقا فسيتم محو محتواه) فإذا لم يكن موجودا من قبل فإنه سيتم إنشاءه في الحين وبخصائص عادية.

ثم نحتاج لطول البيفر (buffer) الذي سيحجز لحفظ البيانات من قبل تحكم التحرير، باعتباره مصدر البيانات الواجب تسجيلها في الملف. لما نحجز البيفر الذاكري فإننا نستخلص سلسلة الحروف من تحكم التحرير مستخدمين في ذلك الدالة GetWindowText() ثم نكتبها في الملف عن طريق الدالة WriteFile(). مرة أخرى، وكما تم مع عملية القراءة، فإن البارامتر الذي يعيد عدد مرات عملية الكتابة يجب أن يكون موجودا ضمن قائمة البارامترات رغم أننا لم نستخدمه.

III - الجزء الثالث: شريط الأدوات وشريط الحالة

المثال المرفق: app_three

ملاحظة مهمة حول التحكمات الشائعة



كما نفعل مع كل التحكمات الشائعة، يجب دائما استدعاء الدالة InitCommonControls() قبل محاولة استخدامها. يجب أن تقوم بإدراج ملف الرأس <commctrl.h> للتمكن من استعمال هذه الدالة وللحصول على الدوال والتصريحات اللازمة والمتعلقة بالتحكمات الشائعة. كما انك ستحتاج كذلك لإدراج الملف comctl32.lib إلى خيارات الرابط (Linker Settings) إذا لم تكن محددة. تذكر دائما أن الدالة InitCommonControls() هي دالة API قديمة، ولأغلبية التحكمات يمكنك استخدام InitCommonControlsEx() والمطلوبة بكثرة من قبل أغلب التحكمات الحديثة. على كل، بما أنني لم أستخدم خصائص متقدمة فإنني سأستعمل في أمثلتي الدالة InitCommonControls() لبساطتها وكفايتها.

أشرطة الأدوات

يمكنك إنشاء شريط أدوات من خلال الدالة CreateToolbarEx() لكنني لا أعتزم اعتماد ذلك في مثالي، إذن فباعتبار شريط الأدوات نافذة ذات ميزة خاصة، فإننا سننشئ هذا الشريط معتمدين على هذا المثال:

```
hTool = CreateWindowEx (0, TOOLBARCLASSNAME, NULL,
                        WS_CHILD | WS_VISIBLE, 0, 0, 0, 0,
                        hwnd, (HMENU)IDC_MAIN_TOOL,
                        GetModuleHandle(NULL), NULL);
```

ببساطة، هذا يكفي لإنشاء شريط أدوات، الآن دعنا نطلع على بارامترات الدالة CreateWindowEx() ↺ فالبارامتر TOOLBARCLASSNAME هو ثابت معرف في ملفات الأس (derhea) للتحكمات الشائعة ↺ . hwnd هو مقبض النافذة الأم ↺ التي سيتوضع عليها شريط الأدوات. IDC_MAIN_TOOL هو معرف هذا الشريط، لأنك قد تحتاج إليه مستقبلا للحصول على مقبض الشريط وذلك باستخدام للدالة GetDlgItem() ↺ إذا ما رغبت في ذلك.

```
// Send the TB_BUTTONSTRUCTSIZE message, which is required for
//backward compatibility.
      SendMessage( hTool , TB_BUTTONSTRUCTSIZE,
                   (WPARAM)sizeof(TBBUTTON), 0);
```

الرسالة TB_BUTTONSTRUCTSIZE (لاحظ أنها تختلف عن الرسائل من نوع *_WM) ضرورية لتمكين النظام من تحديد إصدار مكتبة التحكمات الشائعة التي تستخدمها. بما أن آخر الإصدارات تضيف أشياء جديدة للهيكل، فإنك بإعطائك للحجم المطلوب يمكنها عرض النتائج المرتقبة لك من دون مشاكل.

أزرار شريط الأدوات

الأزرار ذات صور البيتماب في أشرطة الأدوات الأساسية تأتي على تشكيلتين، أزرار قياسية مقدمة من comctl32 ، وأزرار معرفة من المستخدم، أي تلك التي تنشئها بنفسك. انتبه إلى أن الأزرار وصور البيتماب تضاف إلى أشرطة الأدوات منفصلة (كل على حدى) ... أولا تضيف لائحة من الصور لأجل استعمالها، ثم تضيف لائحة أخرى لكنها من الأزرار، ثم تحدد لكل زر الصورة الموافقة له.

إضافة أزرار قياسية

الآن وقد أنشأنا شريط الأدوات، نحتاج لإضافة باقة من الأزرار إليه. أكثر صور البيتماب شيوعا تجدها متوفرة في مكتبة التحكمات الشائعة (common controls)، إذن لن نكون مجبرين على إعادة إنشائها ولا إضافتها لأي ملف تنفيذي قد يستخدمها.

أولا نصرح بـ TBBUTTON و TBADDBITMAP

```
TBBUTTON tbb[3];// التصريح بثلاثة أزرار لشريط الأدوات
TBADDBITMAP tbab;
```

ثم نقوم بإضافة صور البيتماب القياسية إلى شريط الأدوات، وهذا بعد استعمال للقائمة المعرفة مسبقا في مكتبة التحكمات الشائعة...

```
tbab.hInst = HINST_COMMCTRL;
tbab.nID = IDB_STD_SMALL_COLOR;
SendMessage(hTool, TB_ADDBITMAP, 0, (LPARAM)&tbab);
```

بما أننا الآن حملنا صورنا، فيمكننا الآن إضافة الأزرار التي ستستعملها.

```
ZeroMemory(tbb, sizeof(tbb));
tbb[0].iBitmap = STD_FILENEW;
tbb[0].fsState = TBSTATE_ENABLED;
tbb[0].fsStyle = TBSTYLE_BUTTON;
tbb[0].idCommand = ID_FILE_NEW;

tbb[1].iBitmap = STD_FILEOPEN;
tbb[1].fsState = TBSTATE_ENABLED;
tbb[1].fsStyle = TBSTYLE_BUTTON;
tbb[1].idCommand = ID_FILE_OPEN;
```

```
        tbb[2].fsState = TBSTATE_ENABLED;
        tbb[2].fsStyle = TBSTYLE_BUTTON;
        tbb[2].idCommand = ID_FILE_SAVEAS;

        SendMessage(hTool, TB_ADDBUTTONS,
                    sizeof(tbb)/sizeof(TBBUTTON), (LPARAM)&tbb);
```

لقد أضفنا الأزرار الآتية: جديد، فتح و حفظ، كما أسندنا لها لائحة صور البيتماب المعرفة مسبقا، وهذا أسلوب جميل اعتاد عليه أغلب المبرمجين، وصار شيئا بديهيا بينهم.

فهرس (index) كل صورة نقطية (bitmap) في لائحة الصور معرف مسبقا في ملفات الرأس للتحكمات الشائعة، كما أنها موجودة في الـ MSDN

لقد أسندنا لكل زر معرف (ID_FILE_NEW، ...إلخ) ، هذه المعرفات تماثل تلك الخاصة ببنود القائمة. ↺ هذه الأزرار ستولد رسائل من نوع WM_COMMAN ↺ تطابق تلك المولدة من القائمة، إذن لن نحتاج لعمليات جديدة، أما لو قمنا بإدراج زر لا يوجد له بند (item) يوافقه في القائمة، فإننا في هذه الحالة سنسند له معرفا جديدا، ونلتقط مقابض رسائله.

إذا كنت مندهشا من وجود wParam المخيف والذي قمت بتمريره إلى TB_ADDBUTTON، السبب هو أنها تسمح بإجراء عملية عد للأزرار المتواجدة في مصفوفة tbb (array)، إذن لن نحتاج لكود معقد لتحقيق ذلك. لو قمت بكتابة العدد 3 بدل ذلك، فإن الكود سيكون صحيحا، لكن إذا ما قمت وأضفت زرا جديدا فإنني سأضطر لكتابة 4، أو 5 أو .. وهذا في عالم البرمجة يعتبر أمرا سيئا.

إذا كان sizeof(TBBUTTON) بحجم 16 بايت (مثال فقط، لأن الحجم الحقيقي يختلف حسب منصة العمل) وبما أننا نملك 3 أزرار فإن حجم sizeof(tbb) سيكون 3 * 16 = 48 بايت. ومنه سيعطينا ناتج 48÷16 عدد الأزرار الموجودة.

أشرطة الحالة

كثيرا ما نجد التطبيقات تحوي أشرطة حالة، والتي نقصد بها تلك النافذة الطويلة الموجودة أسفل نافذة البرنامج الرئيسية، بحيث تقوم بعرض معلومات تختلف باختلاف نوعية التطبيق. هي بسيطة الاستعمال، فقط قم بإنشائها.

```
hStatus = CreateWindowEx(0, STATUSCLASSNAME, NULL,
      WS_CHILD | WS_VISIBLE | SBARS_SIZEGRIP, 0, 0, 0, 0,
      hwnd, (HMENU)IDC_MAIN_STATUS, GetModuleHandle(NULL), NULL);
```

ثم قم (اختياريا) بتثبيت عدد الأجزاء إذا ما رغبت في ذلك. في مثالي لم أقم بتثبيت أي منها، سيكون لدينا جزء واحد بعرض كامل الشريط، كما يمكنك كتابة وقراءة النص من خلال الدالة SetWindowText() كما هو الحال مع بقية التحكمات. أما لو كنت تسعى لإنشاء شريط أدوات بعدة أجزاء، ففي هذه الحالة أنت مدعو لتحديد عرض كل جزء، ثم تستعمل SB_SETTEXT لكتابة النص في كل واحد.

لتعيين عرض كل جزء فإننا سنصرح بمصفوفة (array) من الأعداد الصحيحة، بحيث تكون كل قيمة تمثل عرض جزء ما. ولو كنت تريد جزءا واحدا، ففي هذه الحالة تبث قيمة العرض على –1.

```
    int statwidths[] = {100, -1};

    SendMessage(hStatus, SB_SETPARTS,
            sizeof(statwidths)/sizeof(int), (LPARAM)statwidths);
    SendMessage(hStatus, SB_SETTEXT, 0, (LPARAM)"Hi there :)");
```

مرة أخرى يصادفك البارامتر wParam ، هنا أيضا استعملناه لحساب عدد العناصر الموجودة في المصفوفة. لما ندرج أجزاء إضافية فإننا نبتدئ قيمة أول عنصر (element) من المصفوفة بـ 0.

قياسات ملائمة

على عكس القوائم، فإن أشرطة الأدوات وأشرطة الحالة تعتبر تحكمات منفصلة تسبح داخل منطقة المستخدم للنافذة الأم. لهذا، فإذا قمنا بترك كود الرسالة WM_SIZE كما كان من قبل، فإن هذه الأشرطة ستغطي جزءا من تحكم التحرير الذي أضفناه من قبل. هذه مشكلة يمكن ببساطة حلها... في مقطع الكود الخاص بالرسالة WM_SIZE الموجودة في إجراء النافذة (window procedure)، سنقوم بتحريك كل من شريط الأدوات وشريط الحالة إلى موقعيهما، ثم نطرح قيمة عرضيهما وارتفاعيهما من قيمتي عرض وارتفاع منطقة العميل، مما يسمح لنا بتحريك تحكم التحرير وتعديل قياساته حتى يناسب الحجم المتبقي...

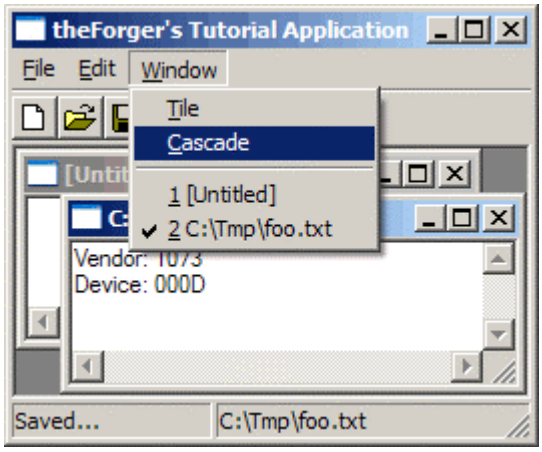
```
    HWND hTool;
    RECT rcTool;
    int iToolHeight;

    HWND hStatus;
    RECT rcStatus;
    int iStatusHeight;

    HWND hEdit;
    int iEditHeight;
    RECT rcClient;

    // Size toolbar and get height

    hTool = GetDlgItem(hwnd, IDC_MAIN_TOOL);
    SendMessage(hTool, TB_AUTOSIZE, 0, 0);

    GetWindowRect(hTool, &rcTool);
    iToolHeight = rcTool.bottom - rcTool.top;

    // Size status bar and get height

    hStatus = GetDlgItem(hwnd, IDC_MAIN_STATUS);
    SendMessage(hStatus, WM_SIZE, 0, 0);
```

```
    iStatusHeight = rcStatus.bottom - rcStatus.top;

    // Calculate remaining height and size edit

    GetClientRect(hwnd, &rcClient);

    iEditHeight = rcClient.bottom - iToolHeight - iStatusHeight;

    hEdit = GetDlgItem(hwnd, IDC_MAIN_EDIT);
    SetWindowPos(hEdit, NULL, 0, iToolHeight, rcClient.right,
iEditHeight, SWP_NOZORDER);
```

قد يكون هذا المقطع من الكود نوعا ما طويل، لكنه في الحقيقة بسيط... أشرطة الأدوات ستقوم بتعديل مواقعها بصورة آلية لما نرسل لها الرسالة TB_AUTOSIZE، وأشرطة الحالة ستقوم بنفس الشيء لما نرسل لها الرسالة WM_SIZE (مكتبات التحكمات الشائعة غير معروفة بالتطابق).

IV - الجزء الرابع: واجهة متعددة المستندات MDI

المثال المرفق: app_four



نبذة عن الـ MDI

القاعدة الأساسية التي ننطلق منها، هي أن لكل نافذة منطقة عميل خاصة بها، هنا حيث تقوم أغلب البرامج بتصميم رسوماتها، بتنظيم تحكماتها ...إلخ، منطقة العميل ليست منفصلة عن النافذة نفسها، هي ببساطة ناحية مميزة من نواحي النافذة. أحيانا قد تكون النافذة كلها منطقة عميل، ولا شيء يزاحمها، وأحيانا منطقة العميل تكون صغيرة لترك أماكن للقوائم، أشرطة الأدوات، أشرطة الحالة، أشرطة التمرير... إلخ.

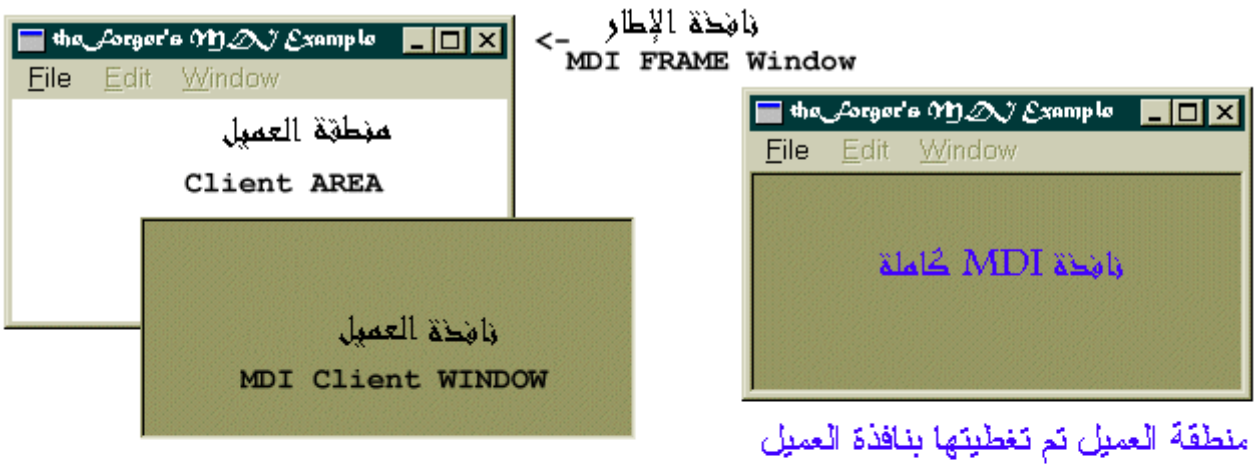
في قاموس الـ MDI، تسمى النافذة الأساسية بنافذة الإطار (Frame Window) واختصارا **بالإطار**، هذه قد تكون النافذة الوحيدة التي يمكنك الحصول عليها في حالة العمل على برامج الـSDI (Single Document Interface).

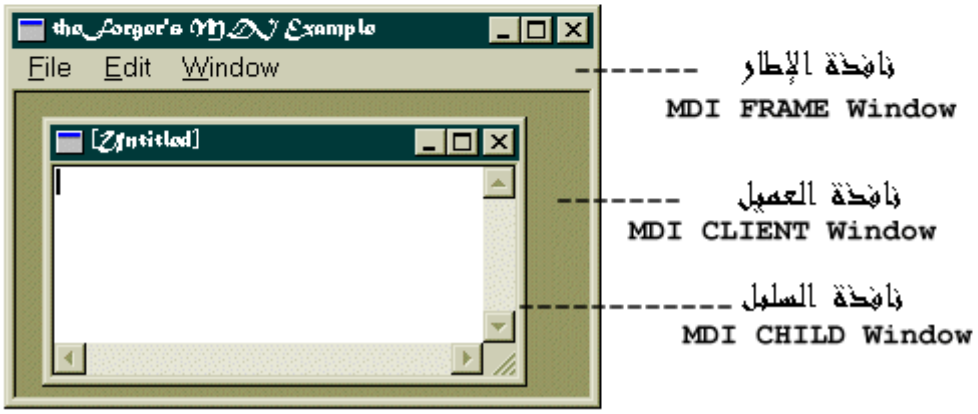
في الـ MDI توجد نافذة إضافية، تدعى بنافذة العميل للواجهة المتعددة المستندات (MDI Client Window)، وفي قاموس الـ MDI تدعى بسليل نافذة الإطار (Child of the Frame Window) واختصارا **بالسليل**. وعلى عكس منطقة العميل التي أشرنا إلى أنها غير منفصلة عن النافذة الأم، فإن السليل هو نافذة كاملة ومنفصلة عن الإطار، يملك السليل منطقة العميل الخاصة به (لاحظ الشكل) وقليل من الحاشية. لا يمكنك التقاط رسائل لعميل الـ MDI مباشرة، إنما هي تتم بفضل الفئة (class) المعرفة مسبقا "MDICLIENT". يمكنك التواصل والعمل على كل من نافذة العميل والنوافذ التابعة لها عبر الرسائل.

لما تصل إلى الويندوز الذي يقوم بعرض المستند أو أي شيء يعرضه برنامجك، فإنه يمكنك إرسال رسالة إلى عميل الـ MDI لمطالبته بأن ينشئ نافذة من النوع الذي حددته. النافذة الجديدة يتم إنشاءها على أساس أنها إبن لعميل الـ MDI، وليس كإبن لإطار النافذة. هذه النافذة الجديدة هي سليل من نوع MDI، وكما قلنا، فهي إبن للعميل، هذا الأخير هو أيضا إبن للإطار (لاحظ الشكل). ولتعقيد الفهم أكثر، فإنه يمكن لسليل الـ MDI أن يحوي أيضا نوافذ صغيرة أخرى تابعة له. وكمثال بسيط: تحكم التحرير (Edit) الذي رأيناه في الدرس السابق (يمكنه أن يكون إبنا لسليل الـ MDI).

أنت مسؤول عن كتابة إجرائي نافذتين اثنين (أو أكثر). واحد، تماما كبقية الدروس، لنافذتك الأم (الإطار). والآخر لسليل الـ MDI. يمكنك كذلك أن تملك نوعا مغايرا من السليل، وهذا وارد في حالة رغبتك في فصل إجراء النافذة لكل نوع.

الشكل الآتي يستعرض كلامنا المعقد في تصميم بياني مفهوم.





بداية العمل مع ألـ MDI

تحتاج الـ MDI لبعض التغييرات الجدية في كثير من قطاعات البرنامج، لهذا فيستحسن لك أن تقرأ هذا الفصل بحذر... قد تكون محظوظا إذا ما استطعت تعقب الأخطاء حينما يتوقف برنامجك أو يتصرف تصرفا غريبا، مما يساعدك على معرفة الأكواد الشاذة التي قمت بكتابتها.

نافذة عميل الـ MDI

قبل أن ننشئ نافذة الـ MDI الخاصة بنا، فإننا نحتاج للقيام ببعض التعديلات على معالجة الرسائل الافتراضية (default)، والتي تتم على مستوى إجراء نافذتنا... فبما أننا أنشأنا إطار النافذة الذي سيحتضن عميل الـ MDI فإننا نحتاج لتعديل الاستدعاء لـ DefWindowProc() إلى DefFrameProc() والتي تضيف مقبض رسالة مميز لنوافذ الإطار.

```
default:
    return DefFrameProc(hwnd, g_hMDIClient, msg, wParam, lParam);
```

الخطوة الموالية هو إنشاء نافذة عميل الـ MDI ، باعتبارها سليل لنافذة إطارنا. سنقوم بذلك في WM_CREATE كما هو معتاد...

```
CLIENTCREATESTRUCT ccs;

ccs.hWindowMenu  = GetSubMenu(GetMenu(hwnd), 2);
ccs.idFirstChild = ID_MDI_FIRSTCHILD;

g_hMDIClient = CreateWindowEx(WS_EX_CLIENTEDGE, "mdiclient", NULL,
WS_CHILD | WS_CLIPCHILDREN | WS_VSCROLL | WS_HSCROLL | WS_VISIBLE,
CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
hwnd, (HMENU)IDC_MAIN_MDI, GetModuleHandle(NULL), (LPVOID)&ccs);
```

مقبض القائمة (menu handle) هو المقبض المخصص للقائمة الظرفية التي سيقوم فيها عميل الـ MDI بإضافة بنود للإشارة إلى كل نافذة يتم إنشاءها، سامحا بذلك للمستخدم باختيار النافذة التي يود تنشيطها من خلال القائمة، سنقوم بإضافة الإجراءات اللازمة قريبا لتأدية هذه المهمة. في هذا المثال ستكون ثالث قائمة ظرفية بما أنني قمت بإدراج تحكم تحرير ونافذة للقائمة بعد البند File.

ccs.idFirstChild هو رقم يتم إسناده كمعرف للبنود التي يقوم العميل بإضافتها لقائمة النافذة... إذا كنت تريد أن تكون سهلة التمييز عن معرفات القائمة الخاصة بك، فإنه يمكنك التقاط مقابض تحكمات القائمة وتمرير تحكمات قائمة النافذة لـ DefFrameProc() لتتم معالجتها. في هذا المثال قمت بتحديد معرف بقيمة 50000، بحيث لا أتوقع أن تكون معرفات قائمتي أكبر منه.

الآن، ولجعل هذه القائمة تعمل بانسجام فإننا ملزمون بإضافة بعض المعالجة لمقبض رسالتنا WM_COMMAND.

```
case WM_COMMAND:
    switch(LOWORD(wParam))
    {
        case ID_FILE_EXIT:
            PostMessage(hwnd, WM_CLOSE, 0, 0);
        break;

        // ... handle other regular IDs ...

        // Handle MDI Window commands
        default:
        {
          if(LOWORD(wParam) >= ID_MDI_FIRSTCHILD)
            {
            DefFrameProc(hwnd, g_hMDIClient, msg, wParam, lParam);
            }
          else
            {
            HWND hChild = (HWND)SendMessage(g_hMDIClient,
                                            WM_MDIGETACTIVE, 0,0);
```

```
                {
                  SendMessage(hChild, WM_COMMAND, wParam, lParam);
                }
            }
        }
    }
break;
```

لقد أضفت الحالة : default التي ستقوم بالتقاط كل التحكمات التي لم أقم بمعالجتها مباشرة، كما أنها تقوم بعملية تدقيق لمعرفة ما إذا كانت القيمة أكبر أو تساوي ID_MDI_FIRSTCHILD . فإذا كانت كذلك، فإن المستخدم قد قام بالنقر على واحدة من بنود قائمة النافذة ثم نرسل الرسالة إلى DefFrameProc() لتتم معالجتها.

إذا لم تكن واحدة من معرفات النافذة، فإنني ألتقط المقبض لتنشيط نافذة السليل وأسند له الرسالة لتتم معالجتها. هذا يسمح لك بتخويل المسؤولية لنوافذ السليل كي تؤدي بعض الأنشطة، وتسمح لمختلف نوافذ السليل بالتقاط التحكمات (controls) في عدة أساليب إذا ما كنت تود ذلك. في هذا المثال قمت فقط بالتقاط التحكمات التي توصف بأنها شاملة بالنسبة لإجراء نافذة الاطار، وإرسال التحكمات التي ترتبط بمستند ما أو نافذة سليل إلى نافذة السليل نفسها لتتم معالجتها.

لقد أنشأنا في مثالنا السابق كودا يقوم بتحجيم تحكم التحرير، هذا الكود سيكون ملائما لتحجيم نافذة السليل، والذي يأخذ في الحسبان حجم وموقع كل من شريطي الحالة والأدوات، إذا لن تتداخل في نافذة عميل الـ MDI.

نحتاج كذلك لإجراء بعض التعديلات الطفيفة على حلقة الرسائل...

```
    while(GetMessage(&Msg, NULL, 0, 0))
    {
        if (!TranslateMDISysAccel(g_hMDIClient, &Msg))
        {
            TranslateMessage(&Msg);
            DispatchMessage(&Msg);
        }
    }
```

لقد أضفنا خطوة عجيبة: TranslateMDISysAccel()، هذه الدالة تقوم بالتدقيق في مفاتيح التسريع المعرفة مسبقا، Ctrl+F6 والتي تقوم بالتبديل بين النوافذ، Ctrl+F4 التي تغلق السليل، وهلم جرا... إذا لم تقم بإضافتها في هذا التدقيق فإنك ستزعج مستخدمي برنامجك من خلال عدم تزويدهم بالأساليب التي اعتادوا القيام بها، أو أنك ستقوم بإضافة كودها يدويا.

فئة نافذة السليل

بالإضافة إلى نافذة البرنامج الرئيسية (نافذة الإطار) فإننا نحتاج لإنشاء فئات نوافذ جديدة لكل نوع من نوافذ السليل التي نريدها. كمثال: يمكن أن تكون لك واحدة لعرض النصوص وواحدة لعرض الصور أو الرسومات البيانية. في هذا المثال سننشئ نوعا واحدا فقط من نوافذ السليل، والتي ستكون مماثلة لبرنامج التحرير المنشأ في المثال السابق.

```
BOOL SetUpMDIChildWindowClass(HINSTANCE hInstance)
{
    WNDCLASSEX wc;

    wc.cbSize        = sizeof(WNDCLASSEX);
    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = MDIChildWndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_3DFACE+1);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = g_szChildClassName;
    wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

    if(!RegisterClassEx(&wc))
    {
        MessageBox(0, "Could Not Register Child Window", "Oh...",
                   MB_ICONEXCLAMATION | MB_OK);
        return FALSE;
    }
    else
        return TRUE;
}
```

عملية التسجيل تماثل في مجملها تلك المعتمدة مع نوافذ الإطار الاعتيادية، لا توجد رايات خاصة يتم استعمالها مع MDI . لقد أعددنا القائمة على القيمة NULL، وإجراء النافذة سيؤشر نحو إجراء نافذة السليل والذي سنكتبه لاحقا.

إجراء سليل الـ MDI

إجراء نافذة الـ MDI يماثل في تركيبته أي إجراء آخر لكن مع قليل من الاستثناءات. قبل كل شيء، الرسائل الافتراضية (default) يتم تمريرها إلى DefMDIChildProc() بدلا من DefWindowProc().

بخصوص هذه الحالة، نود كذلك تعطيل كل من التحرير وقوائم النافذة لما لا نحتاج إليها (فقط لأنه تمرين جميل)، إذن سنلتقط الرسالة WM_MDIACTIVATE ونفعل أو نعطل التحكمات السابقة اعتمادا على قيمة الرسالة. إذا كنت تملك أنواعا عديدة من نوافذ السليل، في هذا الموضع حيث يمكنك تعديل إعدادات القائمة أو شريط الأدوات أو استبدال واجهة البرنامج كي تنعكس الأنشطة والتحكمات المتميزة حسب شكل النافذة المفعلة.

ولإتمام العملية أكثر، يمكننا إبطال كل من بندي القائمة File، ونقصد بذلك Close و Save، حيث مع عدم تفعيل النوافذ، فإن هذين البندين لا يلزم أن يعملا. بصفة افتراضية، قد أبطلت مفعول

كل هذه البنود في المورد، لهذا لن أحتاج لإضافة كود زائد لتأدية هذه المهمة عند تنفيذ البرنامج لأول مرة.

```
LRESULT CALLBACK MDIChildWndProc(HWND hwnd, UINT msg, WPARAM
                                 wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_CREATE:
        {
            HFONT hfDefault;
            HWND hEdit;

            // Create Edit Control

            hEdit = CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", "",
                WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL |
                ES_MULTILINE | ES_AUTOVSCROLL | ES_AUTOHSCROLL,
                0, 0, 100, 100, hwnd, (HMENU)IDC_CHILD_EDIT,
                GetModuleHandle(NULL), NULL);
            if(hEdit == NULL)
                MessageBox(hwnd, "Could not create edit box.",
                           "Error", MB_OK | MB_ICONERROR);

            hfDefault = GetStockObject(DEFAULT_GUI_FONT);
            SendMessage(hEdit, WM_SETFONT, (WPARAM)hfDefault,
                        MAKELPARAM(FALSE, 0));
        }
        break;
        case WM_MDIACTIVATE:
        {
            HMENU hMenu, hFileMenu;
            UINT EnableFlag;

            hMenu = GetMenu(g_hMainWindow);
            if(hwnd == (HWND)lParam)
            {       //being activated, enable the menus
                EnableFlag = MF_ENABLED;
            }
            else
            {   //being de-activated, gray the menus
                EnableFlag = MF_GRAYED;
            }

            EnableMenuItem(hMenu, 1, MF_BYPOSITION | EnableFlag);
            EnableMenuItem(hMenu, 2, MF_BYPOSITION | EnableFlag);

            hFileMenu = GetSubMenu(hMenu, 0);
```

```
                            MF_BYCOMMAND | EnableFlag);

            EnableMenuItem(hFileMenu, ID_FILE_CLOSE,
                            MF_BYCOMMAND | EnableFlag);
            EnableMenuItem(hFileMenu, ID_FILE_CLOSEALL,
                            MF_BYCOMMAND | EnableFlag);
            DrawMenuBar(g_hMainWindow);
        }
        break;
        case WM_COMMAND:
            switch(LOWORD(wParam))
            {
                case ID_FILE_OPEN:
                    DoFileOpen(hwnd);
                break;
                case ID_FILE_SAVEAS:
                    DoFileSave(hwnd);
                break;
                case ID_EDIT_CUT:
                    SendDlgItemMessage(hwnd, IDC_CHILD_EDIT,
                                       WM_CUT, 0, 0);
                break;
                case ID_EDIT_COPY:
                    SendDlgItemMessage(hwnd, IDC_CHILD_EDIT,
                                       WM_COPY, 0, 0);
                break;
                case ID_EDIT_PASTE:
                    SendDlgItemMessage(hwnd, IDC_CHILD_EDIT,
                                       WM_PASTE, 0, 0);
                break;
            }
        break;
        case WM_SIZE:
        {
            HWND hEdit;
            RECT rcClient;

            // Calculate remaining height and size edit
            GetClientRect(hwnd, &rcClient);

            hEdit = GetDlgItem(hwnd, IDC_CHILD_EDIT);
            SetWindowPos(hEdit, NULL, 0, 0, rcClient.right,
                         rcClient.bottom, SWP_NOZORDER);
        }
        return DefMDIChildProc(hwnd, msg, wParam, lParam);
        default:
            return DefMDIChildProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

لقد جعلت كلا من File Open و File Save على شكل تحكمات (controls)، الدالتان DoFileOpen() و DoFileSave() تشبهان في عملهما لما تم اعتماده في مثالنا السابق مع معرف تحكم التحرير المتغير، زيادة على أنهما تدرجان عنوان سليل الـ MDI في اسم الملف.

تحكمات التحرير تعتبر بسيطة، لأنها أنشئت بدعم منها، لهذا يكفينا مطالبتها بما نريد.

تذكر أنني أشرت إلى وجود بعض الأشياء الصغيرة الواجب عدم إهمالها أو يصير برنامجك غريبا. لاحظ أنني استدعيت الدالة DefMDIChildProc() في نهاية WM_SIZE، هذا مهم في حالة رغبة النظام في نصيب من المعالجة الذاتية للرسائل. يمكنك مطالعة DefMDIChildProc() في مكتبة الـ MSDN لمعرفة لائحة الرسائل الممكن معالجتها، ودائما كن واثقا من تمرير هذا النوع من الرسائل لهذه الدالة.

تنشيط وهدم النوافذ

سليل نوافذ الـ MDI لا يتم إنشاءها مباشرة، بدلا من ذلك سنرسل رسالة WM_MDICREATE إلى نافذة العميل مخبرين إياها عن أي نوع من النوافذ نود إنشاءه، وهذا طبعا من خلال تعديل في إعدادات أعضاء الهيكل MDICREATESTRUCT. يمكنك مراجعة مختلف الأعضاء في الوثائق الخاصة بهذا الهيكل. القيمة المعادة من الرسالة WM_MDICREATE هي مقبض النافذة الجديدة المنشأة.

```
HWND CreateNewMDIChild(HWND hMDIClient)
{
    MDICREATESTRUCT mcs;
    HWND hChild;

    mcs.szTitle = "[Untitled]";
    mcs.szClass = g_szChildClassName;
    mcs.hOwner  = GetModuleHandle(NULL);
    mcs.x = mcs.cx = CW_USEDEFAULT;
    mcs.y = mcs.cy = CW_USEDEFAULT;
    mcs.style = MDIS_ALLCHILDSTYLES;

    hChild = (HWND)SendMessage(hMDIClient, WM_MDICREATE, 0,
                               (LONG)&mcs);
    if(!hChild)
    {
       MessageBox(hMDIClient, "MDI Child creation failed.", "Oh.",
                  MB_ICONEXCLAMATION | MB_OK);
    }
    return hChild;
}
```

أحد أعضاء الهيكل MDICREATESTRUCT والذي لم أستخدمه رغم أنه كثير الاستعمال، ألا وهو العضو lParam. هذا يمكن استخدامه لإرسال قيم ذات 32 بت (كالمؤشرات) إلى السليل الذي قمت بإنشائه أملا في أن تزوده ببعض المعلومات الخاصة. في موقع التقاط لمقبض الرسالة WM_CREATE الخاصة بنافذة السليل، قيمة lParam الخاصة بالرسالة WM_CREATE ستؤشر على

الهيكل CREATESTRUCT. العضو lpCreateParams الخاص بالهيكل سيؤشر على MDICREATESTRUCT الذي أرسلته مع WM_MDICREATE.

```
    case WM_CREATE:
    {
        CREATESTRUCT* pCreateStruct;
        MDICREATESTRUCT* pMDICreateStruct;

        pCreateStruct = (CREATESTRUCT*)lParam;
        pMDICreateStruct = (MDICREATESTRUCT*)
                            pCreateStruct->lpCreateParams;

        /*
        pMDICreateStruct now points to the same MDICREATESTRUCT
        that you sent along with the WM_MDICREATE message and you
        can use it to access the lParam.
        */
    }
    break;
```

إذا لم ترغب في أن تتعب نفسك بهذين المؤشرين الإضافيين فإنه بإمكانك الوصول إلى lParam دفعة واحدة من خلال التعبير الآتي:

```
((MDICREATESTRUCT*)((CREATESTRUCT*)lParam)->lpCreateParams)->lParam
```

الآن يمكننا تنفيذ أوامر File في قائمتنا في إجراء نافذة الإطار الخاصة بنا:

```
    case ID_FILE_NEW:
        CreateNewMDIChild(g_hMDIClient);
    break;
    case ID_FILE_OPEN:
    {
        HWND hChild = CreateNewMDIChild(g_hMDIClient);
        if(hChild)
        {
            DoFileOpen(hChild);
        }
    }
    break;
    case ID_FILE_CLOSE:
    {
        HWND hChild = (HWND)SendMessage(g_hMDIClient,
                                        WM_MDIGETACTIVE,0,0);
        if(hChild)
```

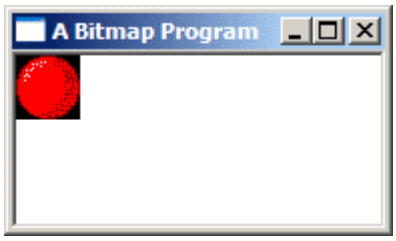
```
            SendMessage(hChild, WM_CLOSE, 0, 0);
        }
    }
    break;
```

يمكننا كذلك التزود ببعض العمليات الافتراضية الخاصة بالـ MDI في بيئة الويندوز وهذا بخصوص القائمة التي أنشأناها، وبما أن الـ MDI تتوافق مع هذا، فإن العمل لن يكون صعبا.

```
    case ID_WINDOW_TILE:
        SendMessage(g_hMDIClient, WM_MDITILE, 0, 0);
    break;
    case ID_WINDOW_CASCADE:
        SendMessage(g_hMDIClient, WM_MDICASCADE, 0, 0);
    break;
```

واجهة المعدات البيانية

I - البيتماب، سياقات الجهاز و BitBlt



المثال المرفق: bmp_one

واجهة الجهاز الرسومية GDI

الشيء الجوهري في الويندوز، هو أنه يختلف عن الدوس الذي يفرض عليك معرفة بعض المعلومات عن العتاد الفيديو الذي تستعمله لعرض الرسومات. بدلا من ذلك، فإن الويندوز يوفر دوال الـ API تعرف بواجهة الجهاز الرسومية (Graphics Device Interface) واختصارا GDI.

واجهة الجهاز هذه تستعمل مجموعة من الكائنات الرسومية والتي يمكننا استخدامها للرسم على الشاشة أو الذاكرة أو حتى الطابعة.

سياقات الجهاز

واجهة الجهاز الرسومية تحوم حول كائن يسمى سياق الجهاز (Device Context) واختصارا DC، يتم تقديمه من قبل النوع HDC (أي مقبض سياق الجهاز). مقبض سياق الجهاز HDC ببساطة هو مقبض كل شيء يمكننا رسمه. يمكن هذا الرسم أن يمثل كل الشاشة، كل النافذة، منطقة العميل الخاصة بالنافذة (window client area)، صورة نقطية (bitmap) محفوظة في الذاكرة أو الطابعة. الشيء الأجمل هو أنك لست ملزم بمعرفة نوع السياق الذي سترسم عليه، لأنها ببساطة تسلك كلها ذات المسلك أثناء الرسم، وهذا يعتبر أمرا مهما في حالة رغبتك في كتابة دوالك الخاصة التي ستتعامل مع كل هذه الأجهزة دون الحاجة لإجراء تعديلات من أجل كل جهاز.

مقبض سياق الجهاز HDC يشبه جل كائنات واجهة الجهاز الرسومية من حيث أنه مغلف، أي أنك ليس بمقدورك الوصول إلى البيانات الأعضاء مباشرة... لكنه يمكنك تمرير العديد من دوال الـ GDI لها لتتم معالجتها أو لرسم شيء ما أو للحصول على قيمها وتعديلها أو أنك تحدث تغييرات جوهرية على الكائن برمته.

كمثال، إذا كنت تريد الرسم على نافذة، أولا يلزمك الحصول على مقبض سياق الجهاز الذي يمثل النافذة (HDC) وهذا بفضل الدالة GetDC()، ثم يمكنك استعمال أيا من دوال الـ GDI التي تأخذ مقبض سياق الجهاز كالـ BitBlt() مثلا من أجل رسم الصورة، والدالة TextOut() التي ترسم النص، والدالة LineTo() التي ترسم الخطوط وهكذا دواليك.

صور البيتماب

صور البيتماب يمكن تحميلها كأيقونات كما أشرنا إلى ذلك سابقا، توجد الدالة LoadBitmap() المعتمدة في تحميل مورد البيتماب، والدالة LoadImage() المستخدمة لتحميل صور نقطية من ملف *.bmp تماما كما هو الحال مع الأيقونات.

ميزة من ميزات واجهة الجهاز الرسومية (الـ GDI) هو عدم قدرتك على الرسم في البيتماب (من نوع HBITMAP) مباشرة. تذكر أن عمليات الرسم هي مخفية من قبل سياق الجهاز، إذن للتمكن من استعمال دوال الرسم هذه على صور البيتماب فإنك ملزم بإنشاء سياق الجهاز في الذاكرة، ثم تقوم بانتقاء HBITMAP بداخله من خلال الدالة ()SelectObject. النتيجة هي أن سياق الجهاز الذي يؤشر عليه المقبض HDC هو البيتماب في الذاكرة، ولما تقوم بالعمل على هذا المقبض فإن الناتج سيكون على البيتماب. وكما أشرت سابقا، فإن هذا الأسلوب يعد الأكثر ملاءمة للعمل على الرسومات، كما يمكنك كتابة كود يمكنه الرسم على مقبض HDC ما، ثم تستعمل هذا الكود على سياق جهاز النافذة أو سياق جهاز الذاكرة من دون أي مشكلة أو تعديل في الكود.

أنت تتوفر على إمكانية معالجة البيانات (data) في الذاكرة. يمكنك عمل ذلك بفضل صور البيتماب المستقلة للجهاز (Device Independent Bitmaps) واختصارا DIB. كما يمكنك كذلك المزج بين العمليات اليدوية وواجهة جهاز الرسم (GDI) مع صور البيتماب المستقلة للجهاز (DIB). لكن هذا الموضوع يخرج عن إطار دروسنا هذه الموجهة للمبتدئين، لهذا فإننا سنغطي تطبيقات واجهة الجهاز الرسومية GDI سطحيا.

استنزاف واجهة الجهاز الرسومية (GDI)

بعد الانتهاء من العمل بالـ HDC، فإنه يستحسن تحريره (تماما كما قمنا بإنشاء المقبض فإننا سنقوم بتحريره، وسنتطرق لذلك لاحقا). كائنات واجهة الجهاز محدودة العدد. في الإصدارات السابقة للويندوز (قبل ويندوز95)، لم تكن محدودة فقط بل وكانت أقل مشاركة في النظام، فمثلا لو قام برنامج بالرسم، واستغل أغلب موارد النظام، فإنه لن يكون بمقدور أي برنامج آخر أن يقوم بالرسم. حسنا، الآن وبعد إصدار الويندوز 2000 و XP يمكنك تحميل العديد من الموارد من دون أن يحدث شيء غير مرغوب فيه. كما يمكن نسيان تحرير كائنات الـ GDI، لأن هذه الأخيرة يمكنها تنفيذ تطبيقك من دون إدراج لها في بيئة الويندوز 9x.

نظريا، لا يجب أن تستنزف موارد الـ GDI في أنظمة الـ NT (NT/2K/XP) ولكنها قد تحدث في حالات قصوى.

إذا نفذ برنامجك لبعض الدقائق بصفة جيدة، ثم صارت عمليات الرسم بطيئة أو توقفت كلية، فإن ذلك إشارة واضحة على أن موارد الـ GDI قد استنزفت. مقابض سياقات الجهاز ليست كائنات الـ GDI الوحيدة التي يلزمك الحذر بخصوص إمكانية تحريرها، ولكن بصفة عامة يمكنك حفظ بعض الأشياء كصور البيتماب والخطوط طيلة حياة البرنامج بدل تحريرها ثم إعادة تحميلها في كل مرة تحتاج إليها.

كذلك، مقبض سياق الجهاز (HDC) يمكنه احتواء نوع واحد فقط من الكائنات في وقت معين (صورة نقطية، خط، قلم...)، أي أن آخر نوع تم إسناده لهذا المقبض هو الذي سيؤشر عليه. فكن إذن حذرا أثناء تعاملك مع هذا الكائن بالضبط. إذا ما تجاهلت نهائيا، فإنه سيضيع وسيكدس الذاكرة مشكلا استنزاف في الـ GDI. لما يتم إنشاء مقبض سياق الجهاز (HDC)، فإنه سيتم إنشاؤه مع بعض الكائنات المسندة له فرضا (by default) ... إذن هي فكرة جيدة أن تقوم بحفظ هذه المعلومات لما يتم إعادتها إليك، ثم بعدما تتم الرسم باستخدام الـ HDC ترجعها إلى وضعها الأول. هذا لا يقوم فقط بحذف بعض كائناتك من الـ HDC (والذي يعتبر عملا جيدا)، بل سيقوم أيضا بجعل الكائنات الافتراضية (default) جاهزة حين ترغب في هدم أو تحرير الـ HDC (والذي يعتبر عملا أجود).

ملاحظة: ليس كل الكائنات لها اختيارات افتراضية من قبل الـ HDC، ويمكنك مراجعة مكتبة الـ MSDN للإطلاع على تلك الغير معنية. حيث كنت غير متأكد من قبل بخصوص أي من مقابض صور البيتماب HBITMAP كان من ضمن مجموعة الكائنات الغير معنية، حيث تظهر الوثائق الخاصة بها على أنها ليست نهائية. والأمثلة المعروضة من قبل مايكروسوفت تهمل في مجملها البيتماب الافتراضية (

default bitmap). حيث قام Shaun Ivory، وهو أحد مهندسي البرمجيات بشركة مايكروسوفت بإعلامي في الدليل السابق أنه يجب تحرير صورة نقطية (bitmap) افتراضية.

كما يظهر جليا وجود خلل برمجي (bug) في حافظة الشاشة (screen saver) التي أنتجتها مايكروسوفت، وتم تبديلها، والسبب هو أن البيتماب الافتراضية لم يتم تعويضها أو هدمها، وفي الأخير قامت بإجهاد موارد الـ GDI. إذن كن حذرا، لأن خطأ كهذا يمكن الوقوع فيه بسهولة.

عرض صور البيتماب

الآن، نعود إلى العمل. عملية الرسم البسيطة على النافذة تحدث بالتقاط رسالة WM_PAINT فلما يتم عرض نافذتك لأول مرة أو إعادتها بعد تصغيرها أو بعد تغطيتها من قبل نافذة أخرى، فإن ويندوز يقوم بإرسال رسالة WM_PAINT إلى نافذتنا مخبرا فيها بأن تقوم بإعادة تصميم نفسها. ولما ترسم شيئا على النافذة، فإن ذلك لا يتم بصورة دائمة، بل ستكون مضطرا لإعادة تصميمه لما تتلقى رسالة WM_PAINT من نظام التشغيل.

```
HBITMAP g_hbmBall = NULL;
```

```
case WM_CREATE:
     g_hbmBall = LoadBitmap(GetModuleHandle(NULL),
                                MAKEINTRESOURCE(IDB_BALL));
     if(g_hbmBall == NULL)
          MessageBox(hwnd, "Could not load IDB_BALL!", "Error",
                         MB_OK | MB_ICONEXCLAMATION);
break;
```

الخطوة الأولى طبعا، هي تحميل البيتماب، هذه العملية بسيطة جدا من خلال مورد البيتماب، ولا يوجد أي اختلاف كبير بخصوص تحميل أنواع أخرى من الموارد. بعد ذلك يمكننا الرسم...

```
     case WM_PAINT:
     {
          BITMAP bm;
          PAINTSTRUCT ps;

          HDC hdc = BeginPaint(hwnd, &ps);
          HDC hdcMem = CreateCompatibleDC(hdc);
          HBITMAP hbmOld = SelectObject(hdcMem, g_hbmBall);

          GetObject(g_hbmBall, sizeof(bm), &bm);

          BitBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, hdcMem,
                  0, 0,SRCCOPY);
```

```
            DeleteDC(hdcMem);

            EndPaint(hwnd, &ps);
        }
        break;
```

الحصول على سياق جهاز النافذة

في البداية نقوم بالتصريح بمجموعة من المتغيرات التي سنحتاج إليها. إنتبه إلى أن أولها هو صورة نقطية BITMAP وليس مقبض الصورة HBITMAP ، فالبيتماب هي هيكل (structure) يحفظ معلومات تخص HBITMAP الذي هو في ذاته كائن من كائنات الـ GDI.

نحتاج لطريقة نحصل بها على ارتفاع وعرض الـ HBITMAP، إذن سنستدعي الدالة GetObject() والتي على عكس ما يشير إليها اسمها، فهي لا تقوم بالحصول على الكائن، ولكن بالأحرى معلومات كائن ما. لعل التسمية "GetObjectInfo" أكثر دلالة وأحسن. إذن الدالة GetObject() تعمل على أنواع عديد من كائنات الـ GDI والتي تتميز عن بعضها من خلال قيمة البارامتر الثاني، وحجم الهيكل.

الهيكل PAINTSTRUCT يحوي معلومات تخص النافذة المراد الرسم عليها، وكيف سيتم التعامل مع رسائل الرسم. يمكنك إهمال المعلومات التي يحويها هذا الهيكل بالنسبة لغالبية المهام البسيطة، ولكنها ستكون ضرورية لاستدعاء الدالتين BeginPaint() و EndPAint()، وكما تشير الأسماء، فإن هاتين الدالتين مصممتان لالتقاط مقبض الرسالة WM_PAINT. ولما لا تحتاج لالتقاط مقبض الرسالة WM_PAINT فإنك ملزم باستعمال الدالة GetDC() والتي سنراها في الدرس اللاحق...، ولكن من المهم استدعاء الدالتين BeginPaint() و EndPAint() للتعامل مع الرسالة WM_PAINT.

الدالة BeginPaint() تعيد لنا مقبض سياق الجهاز HDC الذي يشير إلى مقبض النافذة HWND التي قمنا بتمريرها إلى الدالة، وهذا هو الشيء الذي توجه إليه الرسالة WM_PAINT، أية عملية رسم نقوم بها على هذا مقبض سياق الجهاز HDC ستظهر في حينها على الشاشة.

إعداد ذاكرة سياق الجهاز للصورة النقطية

كما ذكرت آنفا، في حالة الرغبة في الرسم مباشرة على البيتماب، فإننا نحتاج لإنشاء سياق الجهاز (Device Context) في الذاكرة... أبسط سبيل مع ذلك الذي نتوفر عليه هو من خلال CreateCompatibleDC(). هذا الأسلوب يمنحنا سياق جهاز ذاكرة متوافق مع عمق اللون وخصائص العرض الخاصة بمقبض سياق جهاز النافذة HDC.

الآن نستدعي الدالة SelectObject() لاختيار البيتماب داخل سياق الجهاز النشط لحفظ البيتماب الافتراضية من أجل تبديلها بعد ذلك وليس لاستنزاف كائنات الـ GDI.

الرسم

لقد حصلنا على أبعاد البيتماب معبأة في الهيكل BITMAP، إذن يمكننا استدعاء BitBlt() لنسخ الصورة من سياق جهاز الذاكرة إلى سياق جهاز النافذة، إلى هذا الحد سيتم الرسم على

الشاشة. وككل مرة، يمكنك مراجعة مكتبة الـ MSDN، ولكن باختصار ما هي إلا: الوجهة، الموقع، الحجم، المورد وموقع المورد، وأخيرا كود ROP، الذي يهتم بتوضيح أسلوب إنشاء النسخة. في هذه الحالة، نريد فقط نسخة بسيطة ودقيقة للمورد المعمول، ولا نرغب في أشياء خارقة.

لعلى الدالة ()BitBlt أكثر دوال الـWIN32 API نشاطا وحيوية، والوجبة الأساسية لكل مبرمج يسعى لعمل تطبيقات رسومية أو ألعاب تحت نظام الويندوز. لعلى ذلك ما جعلها الدالة الأولى من دوال الـ API التي أحفظ كل بارامتراتها.

المحصلة

إلى هذا الدرجة، ينبغي أن تكون الصورة ظاهرة على الشاشة، وسنحتاج لضبطها بعد ذلك. أول شيء لعمل ذلك هو إعادة سياق جهاز الذاكرة إلى الحالة التي كان عليها عندما امتلكناه أول مرة، مما يعني تبديل صورتنا النقطية بالصورة الافتراضية التي قمنا بحفظها. بعد ذلك يمكننا حذفها كليا باستخدام الدالة ()DeleteDC.

أخير نقوم بتحرير سياق جهاز النافذة الذي حصلنا عليه من ()BeginPaint وهذا عن طريق شقيقتها ()EndPaint.

هدم مقبض سياق جهاز ما (HDC) يعتريه قليل من التشويش، والسبب راجع لتواجد ثلاثة أساليب لعمل ذلك، وكلها مرتبطة بالطريقة التي تم بها الحصول على السياق أول مرة. هذه لائحة لأكثر المناهج المستخدمة في الحصول على الـHDC وكيف يمكن تحريرها في النهاية:

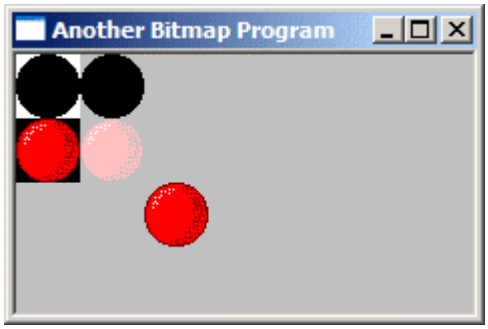
للحصول:	GetDC()	وللتحرير:	ReleaseDC()
للحصول:	BeginPaint()	وللتحرير:	EndPaint()
للحصول:	CreateCompatibleDC()	وللتحرير:	DeleteDC()

وأخيرا، في نهاية برنامجنا، نود تحرير أي موارد قمنا بحجزها. هذا غير مطلوب مطلقا من الناحية التقنية، بما أن الإصدارات الحديثة من ويندوز تقوم بتحرير كل شيء يتعلق بالبرنامج عند نهايته، ولكن يستحسن دائما تتبع أثر كائناتك الخاصة لأنك لو تثاقلت ولم تقم بحذفها فإنها ستعتاد على أن تصير منحلة، وبلا شك، فإن بعض الإصدارات القديمة من ويندوز تحمل في ثناياها أخلالا قد تعمل على عدم حدف كائنات الـGDI بعد نهاية برنامجك إذا لم تقم أنت بالعمل لوحدك.

```
    case WM_DESTROY:
        DeleteObject(g_hbmBall);
        PostQuitMessage(0);
    break;
```

II - صورة نقطية شفافة

المثال المرفق: bmp_two



الشفافية

إعطاء صور البيتماب إمكانية الظهور الشفاف أمر في غاية البساطة، وهذا ينطوي على استخدام صورة القناع الأبيض والأسود إضافة إلى الصورة الملونة التي نريد أن نراها شفافة.

الشروط الآتية ينبغي توفرها كلها من أجل تأدية المهمة على أحسن حال:

أولا، الصورة الملونة يجب أن تكون سوداء في كل نواحي المنطقة التي نريد أن تظهر شفافة. وثانيا، صورة القناع يجب أن تكون بيضاء في مناطق الشفافية، وسوداء في باقي النواحي. لأخذ فكرة أكثر وضوحا، لاحظ النافذة أعلاه، تظهر فيها صور القناع واللون في أقصى اليسار.

عمليات BitBlt

كيف يمكن لهذا أن يحقق لنا الشفافية؟ أولا نقوم بالنسخ المقطعي (بالدالة BitBlt()) لصورة القناع باستعمال عملية SRCAND كآخر بارامتر، وبعد ذلك، وفي الطبقة العليا من الصورة نقوم بالنسخ المقطعي للصورة الملونة مستعملين عملية SCRPAINT . النتيجة هي أن المناطق التي نود أن نجعلها شفافة لن تتغير في مقبض سياق الجهاز المستهدف (HDC) بينما يتم رسم باقي الصورة كالمعتاد.

التطبيق البرمجي قد يوضح المفاهيم أكثر:

```
SelectObject(hdcMem, g_hbmMask);
BitBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, hdcMem, 0, 0,
       SRCAND);
SelectObject(hdcMem, g_hbmBall);
BitBlt(hdc, 0, bm.bmHeight, bm.bmWidth, bm.bmHeight, hdcMem,
       0, 0, SRCPAINT);
```

إذا ظهر لك العمل بسيطا، فهذا شيء جميل، أما في حالة العكس، فإننا سنقوم لاحقا بشرح ذلك بالتفصيل.

حسنا لقد تم، لكن مازال سؤال يتبادر إلى الأذهان... من أين تم الحصول على القناع؟ هناك أسلوبين أساسيين لتحصيل القناع ...

قم بذلك بنفسك في أي برنامج رسومي أدرجت فيه لون البيتماب، وهذا يعتبر حلا مثاليا في حالة كان عدد الأشكال الرسومية محدودا. بهذا الأسلوب يمكنك فقط إدراج مورد القناع (mask resource) إلى برنامجك، وذلك بتحميله مستخدما الدالة LoadBitmap().

أن تقوم باصطناع ذلك بنفسك عندما ينفذ البرنامج، وهذا من خلال اختيار اللون في الصورة الأصلية لجعله يبدو لونا شفافا، ثم تنشئ قناعا يكون أبيضا في كل مكان يتواجد فيه اللون المذكور، وأسودا في باقي النواحي.

مادامت الطريقة الأولى ليست بالشيء الجديد، فإنه يمكنك أن تعمل هذه الأشياء بنفسك إذا ما رغبت في ذلك.

الطريقة الثانية تستلزم من BitBlt() الخداع، لذا سأشير إلى أسلوب يمكنك من إتمام هذه العملية.

إنشاء القناع

أبسط سبيل لعمل ذلك، هو أن تقوم بعمل مسح (scanning) لكل بكسل من الصورة الملونة، والتدقيق في قيمتها، ثم تضع البكسل الموافق للقناع يميل إلى الأبيض أو الأسود... SetPixel() هي دالة جد بطيئة، كما أنها في الحقيقة غير معمول بها.

أحسن أسلوب لعمل ذلك ينطوي على النسخ المقطعي وهذا باستخدام BitBlt() التي ستقوم بتحويل الصور الملونة إلى الأسود والأبيض. إذا قمت بالنسخ المقطعي مستعملا SRCCOPY من مقبض سياق جهاز HDC يؤشر على الصورة الملونة إلى مقبض سياق جهاز HDC يؤشر على صورة بيضاء وسوداء، فإنها ستدقق فيما هو اللون المعد على أنه لون للخلفية في الصورة الملونة فتجعله على الأبيض، أما بقية البكسلات فستجعلها على الأسود.

هذا يصب في صالحنا، لأن كل ما يلزمنا هو جعل لون الخلفية على اللون الذي نريده شفافا، والنسخ المقطعي بـ BitBlt() من الصورة الملونة إلى صورة القناع. انتبه إلى أن العمل هنا يتم مع بيتماب (bitmap) القناع ثنائية اللون (أبيض وأسود)... والمعروفة بأنها صور نقطية ذات 1 بت لكل بكسل. فإذا أردت تجريبها مع الصور الملونة ذات البكسلات البيضاء والسوداء فقط فإنها لن تعمل لك، لأن البيتماب في حد ذاتها تفوق 1 بت (16 أو 24 بت).

ما هو أول شرط لنجاح عملية الإخفاء المذكورة سابقا؟ هو أن الصورة الملونة تحتاج لأن تكون سوداء في أي مكان نود فيه الشفافية. بما أن صورتنا النقطية التي استعملناها في مثالنا السابق كانت تحقق الشرط المذكور، فإننا لم نتلقى أي إشكال، ولكن لو أردت استخدام هذا الكود مع صورة تتميز بلون مختلف يراد منه أن يكون شفافا (وردي فاتح هو حالة شائعة) فإننا في هذه الحالة مجبرون على السير خطوة ثانية، وسنستعمل القناع الذي أنشأناه منذ قليل لإحلال الصورة الأصلية، إذن في أي مكان نود أن يصير شفافا ينبغي أن يكون أسودا. سيكون كذلك إذا كانت باقي الأماكن سوداء، وبما أنها ليست بيضاء في القناع فإنها لا يجب أن تصير شفافة. يمكننا إتمام هذه العملية بـ BitBlt() من القناع الجديد إلى الصورة الملونة، مستعملين عملية SCRINVERT ↶ ، والتي تقوم بجعل كل المناطق البيضاء في القناع سوداء في الصورة الملونة.

هذا كل الكود المخصص لهذا إجراء المعقد، كما أن توفر دوال تنوب عنا في هذه المهمة لأمر في غاية الروعة، فإليك ما يجب:

```
HBITMAP CreateBitmapMask(HBITMAP hbmColour,
                         COLORREF crTransparent)
{
    HDC hdcMem, hdcMem2;
    HBITMAP hbmMask;
    BITMAP bm;
```

```
    // إنشاء صورة نقطية ثنائية اللون (1 بت)

    GetObject(hbmColour, sizeof(BITMAP), &bm);
    hbmMask = CreateBitmap(bm.bmWidth, bm.bmHeight, 1, 1, NULL);

    // الحصول على مقابض سياقات الجهاز المتوافقة

    hdcMem = CreateCompatibleDC(0);
    hdcMem2 = CreateCompatibleDC(0);

    SelectBitmap(hdcMem, hbmColour);
    SelectBitmap(hdcMem2, hbmMask);

    // جعل لون الخلفية للصورة النقطية على شكل
    // اللون الذي نريده أن يظهر شفافا
    SetBkColor(hdcMem, crTransparent);

    // نسخ البتات من الصورة الملونة إلى القناع الأبيض والأسود
    // كل شيء في الواجهة ينتهي إلى الأبيض لما ينتهي كل شيء
    // في الصورة إلى الأسود، وهذا ما نرغب فيه

    BitBlt(hdcMem2, 0, 0, bm.bmWidth, bm.bmHeight, hdcMem,
           0, 0, SRCCOPY);

    // استخدام قناعنا الجديد لتبديل اللون الشفاف في صورة
    // اللون الأصلي إلى الأسود، إذن التأثير الشفاف سيعمل بانتظام
    BitBlt(hdcMem, 0, 0, bm.bmWidth, bm.bmHeight, hdcMem2,
           0, 0, SRCINVERT);

    // تحرير الذاكرة المحتجزة.

    DeleteDC(hdcMem);
    DeleteDC(hdcMem2);

    return hbmMask;
}
```

ملاحظة: هذه الدالة تستدعي SelectObject() لاختيار الصورة الملونة التي قمنا بتمريرها إليها عن طريق مقبض سياق الجهاز (HDC). فالبيتماب لا يمكنها أن تكون محددة من قبل أكثر من HDC في وقت واحد، إذن كن متأكدا من أن البيتماب لم يتم التأشير عليها من قبل مقبض سياق جهاز آخر في نفس الوقت لما نستدعي هذه الدالة وإلا فإنها ستفشل. الآن وبما أننا نحوز على دالة ممتازة، فإنه بإمكاننا إنشاء قناع من الصورة الأصلية تماما بمجرد تحميلها:

```
    case WM_CREATE:
        g_hbmBall = LoadBitmap(GetModuleHandle(NULL),
                               MAKEINTRESOURCE(IDB_BALL));
        if(g_hbmBall == NULL)
            MessageBox(hwnd, "Could not load IDB_BALL!", "Error",
```

```
        g_hbmMask = CreateBitmapMask(g_hbmBall, RGB(0, 0, 0));
        if(g_hbmMask == NULL)
             MessageBox(hwnd, "Could not create mask!", "Error",
                        MB_OK | MB_ICONEXCLAMATION);
    break;
```

البارامتر الثاني هو طبعا اللون المستخلص من الصورة الأصلية والذي نريده أن يكون شفافا، في حالتنا هذه هو الأسود.

كيف يعمل كل هذا؟

يمكنك التساؤل. لكن أخذا في الحسبان خبرتك في لغة السي، فإنك ولابد لديك بعض الإلمام بالعمليات الثنائية (binary) كـ OR، XOR، AND ... الخ. لن أتطرق لشرح هذه التطبيقات، لكن سأبين كيفية استخدامها لمثالنا هذا. إذا ما كان شرحي يبدو غامضا (وهذا ما لا أتمناه) فإن مراجعة العمليات الثنائية (المنطقية) ستساعدك على الاستيعاب أكثر.

إدراك ذلك الآن ليس ضروريا لاستخدامها في الوقت الراهن، ويمكنك الاكتفاء بالاستخدام بعد التأكد من أنها تعمل بإتقان.

(SouRCe AND) SRCAND

العملية SRCAND أو الكود ROP للدالة BitBlt() يعني إجراء عمليات على البتات بتوظيف للمعامل AND. أي فقط البتات الموجبة في المصدر والوجهة (source & destination) هي التي ستظهر في النتيجة النهائية. لقد استخدمنا هذه مع قناعنا لجعل كل البكسلات التي ستملك لونا من الصورة الملونة، لجعلها سوداء. صورة القناع تحوي سوادا (في النظام الثنائي تعني القيمة 0) في أي ناحية نريد ألوانا، وبياضا (في النظام الثنائي تعني القيمة 1) في النواحي التي نريد فيها الشفافية. أي قيمة يتم مطابقتها مع 0 مستعملين AND ستؤول إلى 0، وبالتالي فإن كل البكسلات التي كانت سوداء في القناع سيتم جعلها مساوية لـ 0 في النتيجة، مما يعني أن البكسل الموافق لها سيظهر أسودا. وأي قيمة يتم مطابقتها مع 1 فإنها ستصير 1 إذا كانت 1، و 0 إذا كانت 0 ... إذن كل البكسلات البيضاء في القناع لن تتأثر بعد استدعاء الدالة BitBlt(). النتيجة هي الصورة العلوية اليمنى في شكلنا السابق.

(SouRCe PAINT) SRCPAINT

SCRPAINT تستخدم عملية OR، فإذا كان أحد البتات موجبا أو كلاهما، فإن النتيجة الموافقة ستكون كذلك. نستعمل هذا في الصور الملونة. لما يتم مزج الجزء الأسود (الشفاف) للصورة الملونة مع البيانات التابعة للوجهة (destination) مستعملين OR، فالنتيجة تكون عدم لمس البيانات (data) في الوجهة لأن المصدر معدوم، ولا تأثير للـ 0 على البيانات الأخرى عند التعامل مع المعامل OR.

من جانب آخر، باقي الصورة الملونة ليس أسودا، ولو كانت الوجهة أيضا ليست سوداء، فإنه سينتج مزج بين لوني المصدر والوجهة، والنتيجة يمكنك ملاحظتها في الكرة الثانية في العمود الثاني في مثالنا لهذا الدرس. هذا هو الدافع الرئيسي وراء استخدام القناع لجعل البكسلات التي نريد تلوينها تظهر سوداء في أول المطاف، والسبب هو أننا لما نستعمل OR مع الصورة الملونة، فإن البكسلات الملونة لن تمتزج مع أي شيء يكون تحتها.

(SouRCe INVERT) SRCINVERT

المعامل XOR يعمل على جعل اللون الشفاف في صورتنا الأصلية يتحول إلى الأسود (إذا لم يكن أسودا). مزج بكسل أسود من القناع مع بكسل ملون ليس تابعا للخلفية في الوجهة يتركها من دون تغيير، بينما مزج بكسل أبيض من القناع (تذكر أننا قمنا بتوليده من خلال وضع لون محدد كالخلفية مثلا) مع بكسل ملون تابع للخلفية في الوجهة يعادله ويحوله إلى الأسفل.

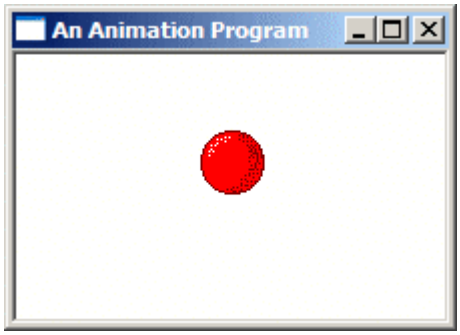
هذا كله حيلة برمجية من حيل الـ GDI تتعلق بها وعكس اللون المستعمل. المعالجة الأحادية اللون تضر بالرأس لمجرد التفكير فيها، لكنها رغم ذلك عمل برمجي لا يستهان به.

مثال

كود المثال في المشروع bmp_two الذي يتفق مع هذا الفصل يتضمن كودا لشكل مثالنا في هذه الصفحة. تتشكل من رسم القناع والصورة الملونة تماما كما يتم استعمال SRCCOPY، ثم يتم استخدام كل واحدة على انفراد بعمليات SRCAND و SRCPAINT على التوالي، وأخيرا المزج فيما بينهما للحصول على النتيجة النهائية.

الخلفية في هذا المثال معدة على الرمادي لجعل الشفافية واضحة أكثر، كما أن استعمال هذه العمليات على خلفية بيضاء أو سوداء يجعلها صعبة لإخبارنا إذا ما تعمل في الواقع أم لا.

III - المؤقت والمتحركات



المثال المرفق: anim_one

الإعداد

قبل أن نحصل على أشياء تتحرك، نحتاج إلى هيكل نخزن فيه وضعية الكرة بين عمليات التحديث. هذا الهيكل سيخزن الموقع الحالي للكرة وحجمها، كما يحفظ قيم الدلتا (Delta)، والمقادير الالخاصة بالحركة في كل إطار.

بعد تصريحنا بنوع الهيكل، فإننا نحتاج كذلك لمثيل شامل لهذا الهيكل (Global Instance). هذا ممكن مادمنا نملك كرة واحدة، إذا كنا سنحرك مجموعة من الكرات فإنك لابد وأن تستعمل مصفوفة (array) أو أي شيء آخر ينوب عنها (كالسلاسل المترابطة linked lists) لحفظ بياناتها التي ذكرناها بطريقة مناسبة وفعالة.

```
const int BALL_MOVE_DELTA = 2;

typedef struct _BALLINFO
{
    int width;
    int height;
    int x;
    int y;

    int dx;
    int dy;
}BALLINFO;
```

لقد قمنا أيضا بتعريف ثابت BALL_MOVE_DELTA والذي يحدد لنا المدى الذي نريد أن تتحرك إليه الكرة في كل تحديث. السبب الذي يدفعنا لتخزين الدلتا في الهيكل BALLINFO هو أننا نريد أن تتحرك كرتنا يمينا أو شمالا، إلى الأعلى أو الأسفل بكل حرية، BALL_MOVE_DELTA هو فقط إسم مناسب لإعطاء القيمة، إذن يمكننا تغييره فيما بعد إذا أردنا ذلك.

الآن نحتاج لابتداء (initializing) هذا الهيكل بعد أن نقوم بتحميل صورتنا:

```
    BITMAP bm;
```

```
    GetObject(g_hbmBall, sizeof(bm), &bm);

    ZeroMemory(&g_ballInfo, sizeof(g_ballInfo));
```

```
    g_ballInfo.height = bm.bmHeight;

    g_ballInfo.dx = BALL_MOVE_DELTA;
    g_ballInfo.dy = BALL_MOVE_DELTA;
```

راجع ZeroMemory هنا ↺

تبدأ الكرة في ركن أعلى اليسار، تنتقل إلى اليمين وإلى الأسفل طبقا لأعضاء الهيكل BALLINFO، ونقصد بذلك: dx و dy.

إعداد الساعة

أبسط سبيل يمكنك من إدراج مؤقت بسيط في برنامج ويندوز هو من خلال SetTimer()، صحيح أنها ليست الأفضل، بل ولا ينصح بها لبرامج الميلتيميديا والألعاب المعقدة، لكن لمثالنا هذا فهي جد كافية. لما تحتاج لأشياء أكثر قوة، يمكنك عندها إلقاء نظرة على TimeSetEvent() في مكتبة الـ MSDN، فإن ذلك أكثر دقة وإحكام.

```
const int ID_TIMER = 1;
```

```
    ret = SetTimer(hwnd, ID_TIMER, 50, NULL);
    if(ret == 0)
        MessageBox(hwnd, "Could not SetTimer()!", "Error", MB_OK |
                   MB_ICONEXCLAMATION);
```

هنا نقوم بتحديد معرف مؤقتنا (id) والذي سيمكننا من الإشارة إليه (وقتله لاحقا kill it) ثم نضع المؤقت في ملتقط رسالة WM_CREATE في نافذتنا الرئيسية. في كل مرة ينقضي عمر مؤقتنا، فإنه سيرسل رسالة من نوع WM_TIMER إلى النافذة، ويمرر لنا الـ ID في wParam. ومادمنا نملك مؤقت واحد فلسنا بذلك ملزمين بتعريفه بـ ID، ولكنه مفيد لك إذا أعددت أكثر من مؤقت وتحتاج للتمييز بينها.

لقد حددنا وقت انقضاء المؤقت بـ 50 ميلي ثانية، مما يعني بالتقريب 20 إطار في كل ثانية. قلت بالتقريب لأن SetTimer() غير دقيقة، لكن بعض الميلي ثانية في مثالنا هذا لا تدعو للقلق، لأن كودنا ليس بذات الأهمية.

التحريك في WM_TIMER

لما نحصل على WM_TIMER فإننا نرغب في حساب الموقع الجديد للكرة، وبالتالي رسم هذا الموقع الجديد.

```
case WM_TIMER:
    {
        RECT rcClient;
        HDC hdc = GetDC(hwnd);

        GetClientRect(hwnd, &rcClient);

        UpdateBall(&rcClient);
        DrawBall(hdc, &rcClient);

        ReleaseDC(hwnd, hdc);
    }
    break;
```

لقد وضعت الكود الخاص برسم وتحديث موقع الكرة في دوالها الخاصة بها. هذا سلوك حسن، يمكننا من رسم الكرة من قبل WM_TIMER و WM_PAINT بدون تكرار الكود، لاحظ أن الأسلوب الذي اعتمدناه للحصول على مقبض سياق الجهاز HDC يختلف من حالة لأخرى، إذن من الأفضل ترك هذا الكود في ملتقط مقبض الرسالة (message handler) وتمرير النتيجة في الدالة ()DrawBall.

```
void UpdateBall(RECT* prc)
{
    g_ballInfo.x += g_ballInfo.dx;
    g_ballInfo.y += g_ballInfo.dy;

    if(g_ballInfo.x < 0)
    {
        g_ballInfo.x = 0;
        g_ballInfo.dx = BALL_MOVE_DELTA;
    }
    else if(g_ballInfo.x + g_ballInfo.width > prc->right)
    {
        g_ballInfo.x = prc->right - g_ballInfo.width;
        g_ballInfo.dx = -BALL_MOVE_DELTA;
    }

    if(g_ballInfo.y < 0)
    {
        g_ballInfo.y = 0;
        g_ballInfo.dy = BALL_MOVE_DELTA;
    }
    else if(g_ballInfo.y + g_ballInfo.height > prc->bottom)
    {
        g_ballInfo.y = prc->bottom - g_ballInfo.height;
        g_ballInfo.dy = -BALL_MOVE_DELTA;
    }
}
```

كل هذا جزء من الرياضيات الأساسية، لقد أضفنا قيمة الدلتا لموقع x لتحريك الكرة. إذا ذهبت الكرة خارج منطقة العميل ↺ فقم بإعادتها إلى الحيز، وغير من قيمة دلتا إلى الإتجاه المعاكس، إذن ستنط الكرة على الجوانب.

```
void DrawBall(HDC hdc, RECT* prc)
{
    HDC hdcBuffer = CreateCompatibleDC(hdc);
    HBITMAP hbmBuffer = CreateCompatibleBitmap(hdc, prc->right,
                                                prc->bottom);
    HBITMAP hbmOldBuffer = SelectObject(hdcBuffer, hbmBuffer);

    HDC hdcMem = CreateCompatibleDC(hdc);
    HBITMAP hbmOld = SelectObject(hdcMem, g_hbmMask);

    FillRect(hdcBuffer, prc, GetStockObject(WHITE_BRUSH));

    BitBlt(hdcBuffer, g_ballInfo.x, g_ballInfo.y,
           g_ballInfo.width, g_ballInfo.height, hdcMem,
           0, 0, SRCAND);

    SelectObject(hdcMem, g_hbmBall);
    BitBlt(hdcBuffer, g_ballInfo.x, g_ballInfo.y,
           g_ballInfo.width, g_ballInfo.height, hdcMem,
           0, 0, SRCPAINT);

    BitBlt(hdc, 0, 0, prc->right, prc->bottom, hdcBuffer,
           0, 0, SRCCOPY);

    SelectObject(hdcMem, hbmOld);
    DeleteDC(hdcMem);

    SelectObject(hdcBuffer, hbmOldBuffer);
    DeleteDC(hdcBuffer);
    DeleteObject(hbmBuffer);
}
```

أساسا، يعتبر هذا نفس كود الرسم المستخدم في الأمثلة السابقة. الإختلاف البسيط يكمن في استخدام الهيكل BALLINFO للحصول على موقع وأبعاد الكرة. ومع ذلك فتوجد نقطة اختلاف جد مهمة...

التخزين الذاكري المضاعف

لما تعمل رسوماتك مباشرة في مقبض سياق جهاز النافذة HDC، يمكن للشاشة أن تقوم بعملية تحديث قبل نهاية مهمتك... مثلا بعد أن تقوم برسم القناع وقبل إتمامك للصورة الملونة في السطح، فيمكن عندها أن يرى المستخدم وميضا للخلفية السوداء قبل أن يحصل برنامجك على فرصة تغطية الصورة بالألوان. مع بطء كمبيوترك، ومع كثرة عمليات الرسم وتكرارها، فإن الوميض سيظهر بوضوح، وفي النهاية سيبدو كفوضى ولخبطة كبيرة.

هذا مقلق للغاية، ويمكننا رغم ذلك حله ببساطة من خلال عمل كل الرسومات في أول الأمر في الذاكرة، ثم ننسخ الشكل كاملا عن طريق BitBlt() إلى الشاشة، مما يعني أن الشاشة سيتم تحديثها مباشرة من الصورة القديمة إلى الصورة الجديدة من دون عمليات منفصلة مرئية.

لعمل ذلك، ننشئ أولا مقبض الصورة النقطية (HBITMAP) في الذاكرة بنفس مقاس المنطقة المراد الرسم عليها من الشاشة. كما نحتاج أيضا لمقبض سياق الجهاز (HDC) الذي نقوم عليه بالنسخ المقطعي بـ BitBlt() نحو الصورة النقطية.

```
HDC hdcBuffer = CreateCompatibleDC(hdc);
HBITMAP hbmBuffer = CreateCompatibleBitmap(hdc, prc->right,
                                                prc->bottom);
HBITMAP hbmOldBuffer = SelectObject(hdcBuffer, hbmBuffer);
```

الآن وقد حصلنا على مكان في الذاكرة يمكننا الرسم فيه، كل عمليات الرسم ستستخدم hdcbuffer بدلا من hdc (النافذة) والنتائج ستحفظ في الصورة النقطية في الذاكرة حتى ننهي عملنا. بعدها يمكننا نسخ كامل العمل في النافذة دفعة واحدة.

```
BitBlt(hdc, 0, 0, prc->right, prc->bottom, hdcBuffer,
       0, 0, SRCCOPY);
```

هذا كل شيء، بعدها نمحي مقابض HDC و HBITMAP كالمعتاد.

تخزين ذاكري مضاعف أسرع ما يمكن

في هذا المثال قمت بإنشاء وهدم الصورة النقطية المستعملة في الحفظ الذاكري المضاعف في كل إطار (frame)، عملت هذا أساسا لأنني أردت أن أكون قادرا على أن أقيس أطوال النافذة، لهذا يستحسن التعود على إنشاء بيفر (buffer) جديد واستخلاص محتواه لما يتم التعديل في موقع النافذة وبالتالي إعادة تحجيم البيفر. قد يكون من الأجدر أن تقوم بالتصريح ببيفرين للصورة النقطية على أن يكونا شاملين (Global)، وأي منهما لا يسمح للنافذة على إعادة تحجيم نفسها، أو حتى تحجيم الصورة النقطية لما يتم إعادة تحجيم النافذة ككل، وهذا بدلا من إنشاء وهدم البيفر في كل مرة. هذا الأسلوب يقتصر عليك إذا ما كنت ترغب في تحسين لعبة رسومية أو أي شيء آخر على علاقة بذلك.

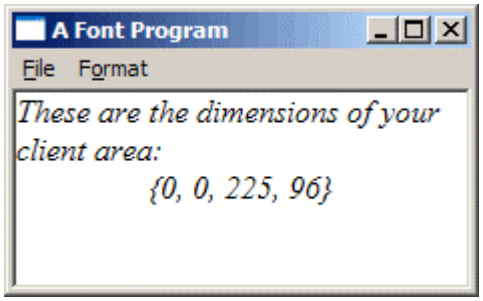
قتل المؤقت

لما يتم هدم نافذتنا، فإنه من الأحسن أن نقوم بتحرير كل الموارد المحتجزة، وفي حالتنا هذه لا ننسى المؤقت الذي قمنا بإدراجه في تطبيقنا. لإيقافه، يمكننا وببساطة إستدعاء الدالة KillTimer() مع تمرير المعرف الذي استخدمناه في عملية الإنشاء.

```
KillTimer(hwnd, ID_TIMER);
```

IV - النص، الخطوط والألوان

المثال المرفق: font_one



تحميل الخطوط

تملك واجهة الجهاز الرسومية (GDI) الخاصة بالوين 32 مجموعة من الإمكانيات الرائعة للتعامل مع حروف الطباعة المختلفة، الأنماط، اللغات ومجموعات الرموز. أحد عيوب ذلك هو أن التعامل بالخطوط كثيرا ما يخيف المبرمجين الجدد. الدالة CreateFont()، دالة الـ API الأولى لما أتت إلى الخطوط كانت تحمل في طياتها 14 بارامترا لتحديد الارتفاع، النمط، العرض، العائلة، ومختلف الخصائص الأخرى.

لكن في الحقيقة لم تكن صعبة كما تظهر، والكثير من العمل في تطبيقاتنا نستعمل فيه القيم الافتراضية (default)، كما أن كل البارامترات (ما عدا 2) التابعة لهذه الدالة يمكن وضعها في القيمة 0 أو NULL، وسيقوم البرنامج باستخدام القيم الافتراضية في مكانها مما ينتج عنه خط عادي وواضح.

الدالة CreateFont() تعيد لنا HFONT، والذي هو مقبض الخط المنطقي (Ligical Font Handle) في الذاكرة. البيانات الواقعة تحت إمرة (سيطرة) هذا المقبض يمكن استعادتها في هيكل من نوع LOGFONT باستعمال الدالة GetObject()، تماما كما يمكن ملء هيكل BITMAP انطلاقا من مقبض صورة نقطية (HBITMAP).

أعضاء الهيكل LOGFONT تتوافق في طبيعتها مع بارامترات CreateFont()، ولراحتك، يمكنك إنشاء خط مباشرة من خلال هيكل LOGFONT مستعملا في ذلك CreateFontIndirect(). هذا مفيد جدا، بما أنه يجعل عملية إنشاء خط جديد من مقبض خط موجود مسبقا عندما تريد فقط تغيير البعض من سماته. استعمل GetObject() لملء هيكل LOGFONT، عدل في قيم الأعضاء التي ترغب فيها، ثم قم بإنشاء خط جديد بالدالة CreateFontIndirect().

```
HFONT hf;
HDC hdc;
long lfHeight;

hdc = GetDC(NULL);
lfHeight = -MulDiv(12, GetDeviceCaps(hdc, LOGPIXELSY), 72);
ReleaseDC(NULL, hdc);

hf = CreateFont(lfHeight, 0, 0, 0, 0, TRUE, 0, 0, 0, 0, 0, 0,
                0, "Times New Roman");

if(hf)
{
    DeleteObject(g_hfFont);
    g_hfFont = hf;
}
else
{
    MessageBox(hwnd, "Font creation failed!", "Error",
```

```
}
```

هذا هو الكود المعتمد في إنشاء الخط في صورة مثالنا لهذا الدرس. الخط هو Times New Roman بحجم 12 نقطة، ونمط مائل (Italic). راية الميلان (italics flag) تقع في سادس بارامتر من بارامترات الدالة CreateFont() حيث يمكنك ملاحظة أننا جعلنا قيمة هذا البارامتر مساوية لـ TRUE. أما اسم الخط الذي نريده فيمثل آخر بارامتر في الدالة.

جزء الكود المراوغ في مثالنا هذا هو القيمة المستعملة في حجم الخط، البارامتر lfHeight للدالة CreateFont(). لقد اعتاد المستخدمون على العمل بحجم النقطة (Point Size)، حجم 10، حجم 12، ...إلخ عند التعامل مع الخطوط. بينما CreateFont() لا تقبل الأحجام بوحدة النقطة، بل تتطلب وحدات منطقية والتي تعتبر مختلفة في شاشتك كما هي في طابعتك، بل وحتى بين الشاشات والطابعات.

السبب لوجود مثل هذه الوضعية هو أن دقة الوضوح تختلف من جهاز لآخر... الطابعات يمكنها عرض من 600 إلى 1200 بكسل في البوصة بكل سهولة، بينما قد تكون الشاشة محظوظة لو حصلت على 200... إذا ما استخدمت نفس حجم الخط في كل من الشاشة والطابعة، فقد تكون غير قادر حتى على رؤية الحروف الفردية.

كل ما نحن مطالبين بالقيام به هو التحويل من الحجم النقطة الذي نريده إلى الحجم المنطقي للجهاز. في مثالنا هذا، الجهاز هو شاشة العرض، إذن نستخلص مقبض سياق الجهاز الخاص بالشاشة (HDC)، ثم نتحصل على عدد البكسلات المنطقية في البوصة باستخدام الدالة GetDeviceCaps() وندفع بذلك في معادلة تم التكرم بها في مكتبة الـ MSDN والتي تستعمل MulDiv()، هذه الأخيرة تقوم بالتحويل من حجم النقطة 12 إلى الحجم المنطقي الصحيح الذي تترقبه CreateFont().

الخطوط الافتراضية

لما تستدعي في بادئ الأمر GetDC() للحصول على مقبض سياق جهاز نافذتك (HDC)، فالخط الافتراضي سيكون ذاك الذي اختير للنظام، لكن ليس بالضرورة ما ترغب فيه. أبسط سبيل للحصول على خط مناسب يمكن العمل به (من دون اللجوء إلى CreateFont()) هو من خلال استدعاء الدالة CreateStockObject() والاستفسار حول الخط الافتراضي لواجهة المستخدم الرسومية (DEFAULT GUI FONT).

هذا كائن من كائنات النظام، ويمكنك استدعاءه متى تريد ولمرات عديدة، دون الخوف من استنزاف الذاكرة، بل ويمكن كذلك استدعاء الدالة DeleteObject() عليه والتي لن تفعل له أي شيء، وهذا شيء جميل لأنك لا تكون مطالبا بمتابعة مصدر خطك، أكان من CreateFont() أو من GetStockObject() قبل محاولة تحريره.

نص التصميم

الآن وبما أننا نملك خطا ممتازا، كيف يمكننا الحصول على نص على الشاشة؟ هذا يفترض عدم رغبتنا في استخدام تحكم تحرير (Edit control) أو تحكم ثابت (Static control).

خياراتك الأساسية هي TextOut() و DrawText(). فالدالة الأولى TextOut() هي الأبسط، لكنها ليست غنية، فهي لا تعمل لك التفافا (wrapping) أو محاداة (alignment).

```
    char szSize[100];
    char szTitle[] = "These are the dimensions of your client
                     area:";
    HFONT hfOld = SelectObject(hdc, hf);

    SetBkColor(hdc, g_rgbBackground);
    SetTextColor(hdc, g_rgbText);

    if(g_bOpaque)
    {
        SetBkMode(hdc, OPAQUE);
    }
    else
    {
        SetBkMode(hdc, TRANSPARENT);
    }

    DrawText(hdc, szTitle, -1, prc, DT_WORDBREAK);

    wsprintf(szSize, "{%d, %d, %d, %d}", prc->left, prc->top,
             prc->right, prc->bottom);
    DrawText(hdc, szSize, -1, prc, DT_SINGLELINE | DT_CENTER |
             DT_VCENTER);

    SelectObject(hdc, hfOld);
```

أول شيء نحن مطالبون باعتماده هو الحصول على الخط المراد في الـHDC، وتجهيزه للرسم. كل النصوص الموالية ستستخدم هذا الخط إلى أن يتم اختيار بديل عنه.

بعدها، نحدد ألوان النص والخلفية. تحديد لون الخلفية لا يجعل هذه الأخيرة كاملة بهذا اللون، إنما يؤثر فقط على بعض العمليات (إحداها النص) التي تستخدم لون الخلفية كأداة رسم. هذا كذلك يتعلق بطراز الخلفية الحالي (Background Mode). إذا كانت محددة في الوضع "عاتم" (OPAQUE) وهي القيمة الافتراضية فإن أي نص مرسوم سيتم ملئه في الخلف بلون الخلفية. أما إذا كانت محددة في الوضع "شفاف" (TRANSPARENT) فإن النص سيرسم من دون خلفية، وأي شيء يقع وراءه سيظهر، أي أن الخلفية في هذه الحالة ليس لها أي أثر.

في الواقع، نحن الآن نرسم النص مستخدمين ()DrawText، نمرر المقبض (HDC) الذي نود استخدامه، والنص الذي نريد رسمه. ثالث بارامتر هو طول السلسلة الحرفية، لكننا قمنا بتمرير 1– لأن ()DrawText بارعة بما فيه الكفاية بحيث ستكتشف طول النص بنفسها. البارمتر الرابع الذي نمرره (prc) هو مؤشر إلى مستطيل العميل (Client RECT)، الدالة ()DrawText ستقوم بالرسم داخل هذا المستطيل مرتكزة على بقية الرايات (flags) التي تقوم أنت بتزويدها بها.

في أول استدعاء، أشرنا إلى DT_WORDBREAK، والتي تنص على المحاذاة نحو الركن العلوي الأيسر، وأيضا بلف النص تلقائيا عند حواف المستطيل... مفيدة جدا.

في الاستدعاء الثاني، نقوم فقط بطبع خط واحد من دون التفاف، ونريده كذلك أن يكون في الوسط أفقيا وعموديا (والذي تقوم به ()DrawText عند رسم خط واحد).

العميل يعيد الرسم

فقط ملاحظة حول برنامج مثالنا... عند تسجيل WNDCLASS فإني قمت بإعداد كل نمطي الفئة CS_VREDRAW و CS_HREDRAW. هذا يفرض على منطقة العميل أن تقوم بإعادة رسم نفسها عند تحجيم النافذة، بينما المفروض هو أن يتم إعادة رسم الأجزاء المعرضة للتغيير فقط. ذلك يبدو سيئا بما أن النص المتوسط يتحرك من كل جانب ولا تقوم أنت بتحديث مضمونه.

أختيار الخطوط

بصفة عامة، أي برنامج يتعامل مع الخطوط يرغب في أن يستخدم المستخدم خطوطه الخاصة، تماما كما هو الحال بالنسبة للألوان والأنماط والخصائص الممكن استعمالها عند العمل بهذا الخط.

مثل الحوارات الشائعة للحصول على مربعات فتح وحفظ أسماء الملفات، هناك أيضا حوارات شائعة لاختيار الخط. ومن دون إشكال، يتم هذا باستدعاء الدالة ChooseFont() والتي ستتعامل مع الهيكل CHOOSEFONT لكي تضع القيم الافتراضية الممكن البدء بها، كما يمكن العودة بها في نهاية عملية اختيار المستخدم للخط.

```
HFONT g_hfFont = GetStockObject(DEFAULT_GUI_FONT);
COLORREF g_rgbText = RGB(0, 0, 0);
```

```
void DoSelectFont(HWND hwnd)
{
    CHOOSEFONT cf = {sizeof(CHOOSEFONT)};
    LOGFONT lf;

    GetObject(g_hfFont, sizeof(LOGFONT), &lf);

    cf.Flags= CF_EFFECTS |CF_INITTOLOGFONTSTRUCT | CF_SCREENFONTS;
    cf.hwndOwner = hwnd;
    cf.lpLogFont = &lf;
    cf.rgbColors = g_rgbText;

    if(ChooseFont(&cf))
    {
        HFONT hf = CreateFontIndirect(&lf);
        if(hf)
        {
            g_hfFont = hf;
        }
        else
        {
            MessageBox(hwnd, "Font creation failed!", "Error",
```

```
        }

        g_rgbText = cf.rgbColors;
    }
}
```

البارامتر hwnd في هذا الاستدعاء يشير إلى النافذة التي تريد أن تستعملها كأب (parent) لمربع حوار الخط.

أبسط طريق لاستخدام هذا المربع الحواري هو باقترانه مع الهيكل LOGFONT المتواجد، والذي في الغالب من نفس نوع HFONT المستعمل في تطبيقك. لقد وضعنا lpLogFont والذي هو عضو (member) من أعضاء الهيكل لكي يشير إلى الهيكل LOGFONT الذي قمنا بملئه منذ قليل بمعلوماتنا الحالية وقبل إضافة الراية CF_INITTOLOGFONTSTRUCT حتى تتعرف الدالة ChooseFont() كيف تستعمل هذا العضو. أما الراية CF_EFFECTS فإنها تخبر الدالة ChooseFont() بتمكين المستخدم من اختيار اللون، وأيضا صفة المسطر (Underline) وبقية الصفات.

الشيء العجيب، النمطين غليظ ومائل لا تؤخذان على أنهما مؤثرين، وإنما على أنهما جزءين من الخط نفسه، وفي الحقيقة، بعض الخطوط تأتي فقط في نمط غليظ أو مائل. إذا أردت أن تفحص أو تمنع المستخدم من اختيار خط غليظ أو مائل فإنه بإمكانك التدقيق في قيم عضوي الهيكل LOGFONT، ونقصد بذلك: IfWeight و IfItalic على الترتيب بعد أن قام المستخدم بتحديد اختياراته. يمكنك بعدها أن تستحث المستخدم على عمل اختيار آخر أو شيء ما يغير في قيم الأعظاء قبل استدعاء الدالة CreateFontIndirect().

لون الخط ليس مرتبطا مع HFONT، لذلك يجب تخزينهما منفصلين، العضو rgbColors التابع للهيكل CHOOSEFONT يستعمل في تمريره في لون الاستهلال (initial) وإيجاد اللون الجديد فيما بعد.

الراية CF_SCREENFONTS يبين أننا نريد خطوطا مصممة للعمل على الشاشة، على تلك المصممة للطابعات. البعض يدعم كلا الجهازين، والبعض يدعم أحدهما. الأمر يتعلق بالهدف من وراء استخدامك للخط. هذه الراية وبقية الرايات يمكن إيجادها في مكتبة الـ MSDN لكي تحدد تماما أنواع الخطوط التي يمكن أن يختار منها المستخدم مراده.

اختيار الألوان

لكي تسمح للمستخدم بأن يغير فقط لون الخط، أو أن يختار لونا جديدا لأي شيئ على الإطلاق، فهناك الحوار المشترك ChooseColor(). هذا هو الكود المستعمل لتمكين المستخدم من اختيار لون الخلفية في برنامج مثالنا.

```
COLORREF g_rgbBackground = RGB(255, 255, 255);
COLORREF g_rgbCustom[16] = {0};
void DoSelectColour(HWND hwnd)
{
    CHOOSECOLOR cc = {sizeof(CHOOSECOLOR)};

    cc.Flags = CC_RGBINIT | CC_FULLOPEN | CC_ANYCOLOR;
```

```
    cc.rgbResult = g_rgbBackground;
    cc.lpCustColors = g_rgbCustom;

    if(ChooseColor(&cc))
    {
        g_rgbBackground = cc.rgbResult;
    }
}
```

هذا بسيط جدا، مرة أخرى نستخدم البارامتر hwnd كأب لمربع الحوار. البارامتر CC_RGBINIT يدعو إلى البدء باللون الذي مررناه عبر البارامتر rgbResult، والذي أيضا نحصل عليه لما يقوم المستخدم باختيار اللون عندما ينغلق مربع الحوار.

المصفوفة g_rgbCustom ذات 16 عنصر (element) من نوع COLORREF مطلوبة لحفظ أية قيم يقررها المستخدم ليتم وضعها في جدول الألوان الخاصة بمربع الحوار. يمكنك أن تحفظ هذه القيم في مكان ما كالسجل مثلا، وإلا فإنها ستفقد لما ينتهي برنامجك. هذا البارامتر ليس طوعيا (optional).

خطوط التحكم

قد تريد في وقت ما تغيير الخط من على التحكمات التابعة لمربع الحوار أو النافذة. عادة تكون هذه الحالة عند استخدام الدالة CreateWindow() لإنشاء التحكمات كما أشرنا إلى ذلك في أمثلتنا السابقة. التحكمات مثل النوافذ، تستعمل النظام كقاعدة، إذن استخدمنا WM_SETFONT لوضع مقبض خط جديد (بـ GetStockObject()) للتحكم المراد استعماله. يمكن اتباع هذا الأسلوب مع الخطوط التي تنشئها بـ CreateFont() أيضا. ببساطة قم بتمرير مقبض الخط كـ wParam واجعل lParam في القيمة TRUE لجعل التحكم يقبل إعادة الرسم.

لقد قمت بهذا في المثال السابق، لكنه يعني نفس الشيء بذكرنا له هنا، لأنه على علاقة ومختصر جدا:

```
    SendDlgItemMessage(hwnd, IDC_OF_YOUR_CONTROL, WM_SETFONT,
                       (WPARAM)hfFont, TRUE);
```

حيث hfFont هو طبعا مقبض الخط (HFONT) الذي تريد أن تستعمله، و IDC_OF_YOUR_CONTROL هو معرف التحكم (control ID) الذي تريد تغيير خطه.

معدات ووثائق

I - كتب ومراجع ينصح باقتنائها

الكتب

إذا ما كنت تبحث عن من يمد لك يد العون مباشرة أثناء تعلمك، فأحسن معين لك هو كتب جيدة. فدروسنا هنا لا يمكن اعتبارها مكتبة ترجع إليها كلما احتجت إليها، وإنما فقط مفتاحا لولوج عالم البرمجة تحت نظام الويندوز، حيث تلاحظ أنها لم تكن أكثر دقة ولا أبسط شرحا، لهذا فالكتب المتوفرة عبر الانترنيت لهي أفضل سبيل لتحقيق مرادك التعلمي

برمجة النوافذ (programming windows)

تأليف تشارلز بيتزولد (Charles Petzold). الكتاب يلج عالم الوين 32. إذا كنت تود كتابة برامج تستعمل فقط دوال الـ API (وهذا ما تناولناه في هذا الكتاب) فإنك تحتاج لهذا الكتاب.

برمجة النوافذ بالـ MFC (Programming Windows with MFC)

تأليف جيف بروسايز (Jeff Prosise). إذا أردت المجازفة في الـ MFC (بعد أن تكون معتادا تماما على استخدام Win32 API)، فهذا الكتاب موجه لك. إذا كنت لا تحب الـ MFC لكن تعتزم على الحصول على عمل يختص بتطوير تطبيقات النوافذ، فالحصول على هذا الكتاب يفي بالغرض، والأفضل أن تعرف من أن لا تعرف.

برمجة تطبيقات للويندوز (Programming Applications for Windows)

تأليف جيفري ركتر (Jeffrey Richter). هذا الكتاب ليس موجها لحديثي العهد بالانترنيت، إذن إذا كنت تود إدارة العمليات والتريدات (processes & threads)، مكتبات الربط الديناميكي (dll)، إدارة ذاكرة الويندوز، معالجة الأخطاء والاستثناءات (exceptions)، والابحار في قلب النظام، إذا أردت كل هذا فما عليك إلا باقتناء هذا الكتاب.

برمجة شال الويندوز بالفيجول سي++ (Windows Shell Programming ++Visual C)

تأليف دينو إسبوزيتو (Dino Esposito). لأي شخص يهتم بالسمات البسيطة والمرئية للويندوز، هذا الكتاب يغطي تصميم الامتدادات لشال الويندوز، العمل بكفاءة مع الملفات والسحب والإفلات، تخصيص شريط الحالة ومستكشف الويندوز، والعديد من الحيل الأخرى. مفيد جدا لمن يرغب في كتابة تطبيقات واجهة المستخدم الرسومية (GUI) تحت نظام الويندوز.

برمجة الشبكة لمايكروسفت ويندوز (ming for Microsoft WindowsNetwork Program)

أحدث المعلومات عن برامج الشبكة، مع إدراج لـ NetBIOS، mailslots ورموز تحويل مدخلات ومخرجات البرامج (pipes)، وكذلك مقابس ويندوز المهمة (sockets)، مكملة بالـ winsock2 و raw sockets. يحتوي كذلك على معلومات تخص مختلف منصات الويندوز، ويتضمن ذلك 2000 و CE.

الوصلات

مكتبة الـ MSDN

هذا الموقع يضم مراجع لكل تكنولوجيات مايكروسوفت الممكنة، فهذه المكتبة تتضمن كل وثائق الـWin32 API و MFC. بالطبع إذا لم تكن متوفرة مع حزمة الفيجول سي++ التي بحوزتك، ثم هذا الموقع المجاني سيزودك بكل ما تحتاجه من معلومات. وتقدر من دون طرح أسئلة على الآخرين من أن تجد أجوبة على انشغالاتك إذا ما أجريت بحثا في هذه مكتبة الـ MSDN.

الصفحة الرئيسية لـ winprog#

يمكنك بعد زيارة هذه الصفحة من أن تجد أجوبة للأسئلة الأكثر شيوعا (FAQ).

II - أدوات سطر الأوامر المجانية للفيجول سي++

الحصول على سطر الأوامر

لقد قامت مايكروسوفت بإطلاق مصرف (compiler) وأدوات الرابط لسطر الأوامر (command line) كجزء من إطار عمل تطوير برامج الدوت نت (.NET Framework Software Developement Kit). إطار عمل الـ SDK تأتي بكل شيء تحتاج إليه لتطوير الدوت نت (مصرف السي شارب C#، ...إلخ) متضمنا بذلك لمصرف سطر الأوامر cl.exe، والذي رغم أنه موجه للاستعمال مع إطار عمل الدوت نت، إلا أنه نفس المصرف المرفق مع حزمة الفيجول سي ++ القياسية.

إطار عمل الـ SDK للدوت نت (تتطلب ويندوز XP أو 2000)

بما أن هذا هو إطار عمل تطوير برامج الدوت نت (.NET SDK)، فإنها لا تأتي مع ملفات الرأس (headers) والمكتبات المطلوبة لتطوير الـ Win 32 API، على اعتبار هذه جزء من منصة الـ SDK. وكما هو معلوم، فمنصة الـ SDK متوفرة مجانا. الشيء الوحيد الذي ستحتاج إليه هو Core SDK، لكن هذا لا يمنع من أن تتزود ببقية العناصر والكائنات الأخرى حسب رغباتك.

منصة الـ SDK

كمكافأة، إذا قمت بتحميل وثائق منصة الـ SDK (والذي أنصحك به)، فإنه سيكون بإمكانك الحصول على آخر إصدار من الوين 32، ومن دون اللجوء إلى مكتبة الـMSDN على شبكة الأنترنت.

تذكر فحص الخيارات لتسجيل المتغيرات المحيطة (Environment Variables) في كلا بنيتي الـ SDK، وإلا فإنك ستحتاج لإعداد المسار (PATH) وبقية المتغيرات بنفسك، قبل استخدام الأدوات مع سطر الأوامر.

استعمال سطر الأوامر

بما أننا نملك وثائق مفهومة وكاملة، وبما أنها متوفرة في مكتبة الـ MSDN، فإنك ستحتاج لمطالعة الدليل بنفسك، وهذا لدراسة مصرف وأدوات بيئة الفيجول سي++. لبداية العمل، إليك أبسط طريق لبناء برنامج ...

```
cl foo.c
```

لبناء تطبيق ويندوز بسيط شبيه بمثالنا في هذا الدليل:

```
rc dlg_one.rc
cl dlg_one.c dlg_one.res user32.lib
```

III - أدوات سطر الأوامر المجانية لبورلاند سي++

الحصول على سطر الأوامر

لحسن حظ من يريد ولوج عالم تطوير الويندوز، فشركة بورلاند قامت بعرض أدوات سطر الأوامر لعامة المبرمجين وبالمجان. ليس لطيفا من لدنهم؟؟ لأنه لا يوجد أي بيئة تطوير متكاملة (IDE) أو محرر مورد، لكن المستنجد لا يحق له أن يرفض، كما يجب علي الإقرار أن المصرف في حد ذاته أحسن بكثير من كل من: LCC-Win32 (والذي حتى لا يدعم السي++) أو مختلف منافذ الأدوات الأخرى، gcc، mingw، cygwin، djgpp ...إلخ.

قم بقراءة ملف readme لأخذ فكرة عن عملية الإعداد.

سطر أوامر Borland C++ 5.5

الشيء الجميل في ذلك هو أنه يأتي مرفقا بمعالج أخطاء (debugger). في الحقيقة أنا لا أستعمله، لهذا لا يمكنني أن أقدم توضيحات أكثر بخصوصه، لكن هذا أفضل من لا شيء. كما أنك لو قمت باستخدام التيربو سي++ تحت نظام الدوس، فإن هذا سيكون حليفا لك.

نظرا لبعض الأسباب، فإن أنترنت إكسبلورر يبدو أنه يلقى مشاكل لتحميل هذا الملف، إذن قم بالنقر بزر الماوس الأيمن واختر "Copy Shortcut"، ثم استعمل هذا العنوان في برنامج يهتم بالتحميل (flashget مثلا) لتتمكن من الحصول عليه.

برنامج Turbo Debugger

أخيرا وليس آخرا، ملف مساعدة الويندوز مع مرجع الـWin32 API المكتمل. هو قديم نوعا ما، لكنه على العموم دقيق وأكثر ملاءمة من مكتبة الـ MSDN على الشبكة، إلا إذا رغبت في أحدث الإضافات إلى الـ API، والذي أقوم باستخدامه بانتظام، فلا تقلق بشأنه.

مرجع Win32 API

إستعمال سطر الأوامر

الأوامر الأساسية

إذا كنت تريد تصريف ملف برنامج وحيد (كمثال: simple_window.c) فإنه بإمكانك استعمال الأوامر الآتية:

```
bcc32 -tW simple_window.c
```

المحول (-tw) يشير إلى أن التطبيق من نوع Win32 GUI، بدلا من تطبيق الكونسول الافتراضي. يمكنك تصريف العديد من الملفات في ملف تنفيذي واحد (.exe) وهذا بإضافة بقية الملفات إلى نهاية هذا الأمر.

ربط الموارد

هذه مسألة جد محبطة لكثير من المستخدمين لأدوات سطر الأوامر، ولا عجب في ذلك، حيث يظهر أن بورلاند جعلت عملية ربط الموارد في تطبيقاتك أصعب ما يمكن، فمصرف المورد brc32 لم يعد يعمل كما كان في إصداراته السابقة، حيث كان يقوم بربط المورد الذي تم تصريفه بنفس الملف التنفيذي. لما تشغل brc32 بدون أي خيار لكي تحصل على مساعدة حول كيفية العمل، فإنه يعرض خيار تعطيل ربط الملف التنفيذي، وهذا يظهر وكأنه أن لا سبيل لتشغيله.

لقد جربت مختلف الأوامر والخيارات في الوضع المركب، لكني لم أجد سبيلا لإدراج ملف مورد (.res) إلى الملف التنفيذي (.exe) تم بناءه بالطريقة السابقة. مما نتج عنه أسلوب جد معقد للقيام بالعملية حصلت عليه في آخر المطاف.

الطريقة البسيطة

يورلاند سي++ الآن تملك طريقة بديلة لإدراج موارد في البرنامج من خلال استخدام #pragma ، وهو تعليمة معالجة أولية ليست قياسية (non-standard preprocessor directive)، وبالتالي قد تتجاهلها مصرفات أخرى إذا لم تكن على علم بمدلولها.

```
#pragma resource "app_name.res"
```

بوضع هذا الكود في ملف الكود الرئيسي (.c أو .cpp) فإن المصرف سيقوم آليا بربط ملف المورد .res والذي تم توليده من الملف .rc (الملف .res هو كالملف .obj ، يخص الموارد).

استعمال #pragma سيسمح لك بتصريف برامج إلى حد ما بالبساطة التي رأيناها سابقا، لكن يلزمك أن تقوم بتصريف ملف المورد .rc أولا مستعملا في ذلك البرنامج brc32. وأما إذا مازلت ترغب في استخدام خيارات سطر الأوامر كما فعلت في دليلي التعليمي makefiles فاقرأه إذن (ملفات متوفرة مع أكواد أمثلتنا).

الطريقة المعقدة

الأوامر الآتية تستعمل لتصريف المثال dlg_one، وهذا مع إدراج للموارد.

```
bcc32 -c -tW dlg_one.c
ilink32 -aa -c -x -Gn dlg_one.obj c0w32.obj,dlg_one.exe,,import32.lib cw32.lib,,dlg_one.res
```

ما أجمل ذلك... فالخيار (-c) للبرنامج bcc32 يعني: قم بالتصريف ولا تقم بالربط بأي ملف تنفيذي. أما الخيارين (-x -Gn) هو أن يتخلص الرابط (linker) من أي ملفات يتم خلقها وأنت قد لا تحتاج إليها.

الشيء المقزز ☺ هنا هو أننا بمجرد قيامنا بتحديد أوامر الرابط يدويا، فإننا سنكون ملزمين بإدراج قائمة المكتبات الافتراضية (default libraries) والكائنات التي كان على المصرف أن يقوم بتضمينها بدلا منا. كما ترى أعلاه، فقد قمت بتحديد الملفات المناسبة لتطبيق ويندوز قياسي.

لجعل العمل أبسط ما يمكن، فإنه من الأحسن أن تقوم بكل هذا في الـmakefile. لقد قمت بإعداد واحد شامل يمكنه العمل مع كل أمثلة دليلنا التعليمي، وينبغي أن تكون قادرا على تكييفه مع أي برنامج من برامجك الخاصة.

```
APP      = dlg_one
EXEFILE  = $(APP).exe
OBJFILES = $(APP).obj
RESFILES = $(APP).res
LIBFILES =
DEFFILE  =

.AUTODEPEND
BCC32   = bcc32
ILINK32 = ilink32
BRC32   = brc32

CFLAGS  = -c -tWM- -w -w-par -w-inl -W -a1 -Od
LFLAGS  = -aa -V4.0 -c -x -Gn
RFLAGS  = -X -R
STDOBJS = c0w32.obj
STDLIBS = import32.lib cw32.lib

$(EXEFILE) : $(OBJFILES) $(RESFILES)
   $(ILINK32) $(LFLAGS) $(OBJFILES) $(STDOBJS), $(EXEFILE), , \
      $(LIBFILES) $(STDLIBS), $(DEFFILE), $(RESFILES)

clean:
   del *.obj *.res *.tds *.map
```

كل ما تحتاج إليه هو تعديل الأسطر الستة الأولى لتلائم معطياتك الخاصة.

ملحقات

I - الملحق أ: حلول للأخطاء الشائعة

الخطأ LNK2001 : الرمز الخارجي _main غير محسوم

يحدث هذا النوع من الأخطاء لما يسعى جزء من الكود باستدعاء لدالة في وحدة نمطية أخرى (module) والرابط لم يقدر على إيجاد هذه الدالة في أي مكتبة من المكتبات أو الوحدات النمطية المرتبطة مع الملف التنفيذي.

هذه الحالة، تعني نقطة من اثنتين: إما أنك تحاول كتابة تطبيق من نوع Win 32 GUI (أو أي نوع آخر ما عدى الكونسول) وبالخطأ قمت بتصريفه على أنه تطبيق من نوع الكونسول...، أو أنك تحاول تصريف تطبيق الكونسول ولكنك لم تكتبه أو تصرفه بالشكل الصحيح في الدالة ()main.

عموما، الحالة الأولى هي الأكثر شيوعا، فإذا قمت بتحديد تطبيق Win32 Console كنوع للتطبيق في بيئة الفيجول سي++ فإنك ستصادف هذا الخطأ. ويحتمل كذلك أن تصادف الخطأ إذا ما حاولت تصريف البرنامج من خلال سطر الأوامر الخاص بالبورلاند سي++ لكنك تهمل أو تجهل تحديد الخيارات الصحيحة لإخبار المصرف (compiler) بإنشاء تطبيق Win32 GUI بدلا من تطبيق الكونسول المعرف فرضا (as default).

التسوية

إذا كنت تستخدم الفيجول سي++ فقم بإعادة إنشاء المشروع على أن تختار في صفحة المشاريع النوع: "Win32 Application project" (وليس Console).

إذا ما كنت تستعمل سطر أوامر البورلاند سي++، فاستعمل (-tW) للإشارة إلى أن التطبيق من نوع تطبيقات الويندوز.

الخطأ C2440: لا يمكن التحويل من void* إلى HICON__* (أو ما شابه ذلك):

إذا قمت بتصريف الأكواد في دليلنا التعليمي هذا، فهذا قد يعني أنك تحاول تصريفه على أنه كود من نوع سي++.

هذا الكود تم كتابته لمصرفات السي (C Compiler) لكل من bcc32 و VC++، وبهذا فإنه قد لا يتم تصريفه بمصرفات السي++ (C++ Compiler)، على اعتبار أن هذا الأخير له عدة قيود على

عملية تحويل الأنواع. بالنسبة لمصرف السي فإنه يقوم بها بدلا عنك، أما مصرف السي++ فإنه يفرض عليك أن تجعل الأمور أكثر وضوحا.

مصرف الفيجول سي++ (وأغلب المصرفات) تقوم بصورة آلية بتصريف الملفات ذات التوسع cpp. على أنها مكتوبة بكود سي++، والملفات ذات التوسع c. على أنها مكتوبة بكود سي. إذن إذا قمت بإضافة كود هذا الدليل إلى ملفات ذات توسع cpp. فإنه يحتمل جدا أن يصادفك هذا النوع من الأخطاء.

إذا كنت تقوم بتصريف كود ليس تابعا لهذا الدليل، فإنني لا أستطيع أن أضمن أنه سليم من العيوب، لهذا فأنت تحتاج لخبرتك لتحويل القيم، والحكم على أي تحويل إذا ما كان يحتاج إلى تحويل قسري (casting) مما قد يزيل الخطأ، أو أنك تحتاج لتصحيح قيم متغيرات أنت تحاول أن تجعلها تعني شيئا هي ليست عليه.

التسوية

إذا أردت أن تستعمل السي، فببساطة قم بإلحاق ملفاتك بالتوسع c. بدل من cpp.، غير ذلك، قم بعمل تحويل قسري (casting) في الأماكن التي يشير إليها الخطأ.

في دليلنا التعليمي، كل الكود سيعمل تحت مصرفات السي++ بدون أية عراقيل.

على سبيل المثال، هذا الكود سيعمل تحت مصرفات السي:

```
HBITMAP hbmOldBuffer = SelectObject(hdcBuffer, hbmBuffer);
```

لكنه في السي++ سيحتاج إلى تحويل قسري.

```
HBITMAP hbmOldBuffer = (HBITMAP)SelectObject(hdcBuffer,
                                             hbmBuffer);
```

الخطأ القاتل RC1015: لا يمكن فتح الملف الرأسي 'afxres.h'

شيء غريب، فالفيجول سي++ يضيف ملف المورد afxres.h حتى عند عدم استخدامك لمشروع من نوع MFC، ورغم ذلك، فالملف يتم فقط تركيبه إذا تم تركيب الـMFC. هذا الملف الخاص ليس مطلوبا في الواقع، لهذا فإذا أردت تسوية الخطأ فإنه بإمكانك تحرير الملف rc. في أي محرر، وتعويض كل تواجد للملف "afxres.h" بالملف "winres.h" (لاحظ أن اسم الملف موجود مرتين، وينبغي تبديله في كل مرة).

الخطأ LNK2001: الرمز الخارجي InitCommonControls غير محسوم

هذا الخطأ يشير إلى أنك لم تقم بربط البرنامج بالمكتبة comctl32.lib التي تم تعريف دالة الـ API فيها. هذه المكتبة لا يتم إدراجها افتراضيا، لهذا فأنت ملزم بإضافتها إلى المكتبات في سطر الأوامر أو من خلال البند "settings" التابع للقائمة "project" وفي الصفحة "Link" في النافذة الناتجة قم بإدراج هذه المكتبة.

الحوار لا يتم عرضه عند إضافة بعض التحكمات

التحكمات مثل ListView، TreeView، Hotkey، Progress Bar وأخرى قد تم تصنيفها على أنها تحكمات شائعة، كما أنها أضيفت إلى ويندوز في المكتبة comctl32.dll ولم تكن متوفرة في الويندوز 95. التحكمات BUTTON، EDIT، LISTBOX، إلخ... تعتبر شائعة، وفي الحقيقة هي ليست "تحكمات شائعة (common controls)"، وكثيرا ما أشرت إليها في دليلي هذا على أنها تحكمات قياسية "Standard Controls".

إذا قمت بإضافة تحكم شائع إلى الحوار، ولكن هذا الأخير يفشل في العرض، فيحتمل جدا أنك فشلت في استدعاء الدالة InitCommonControls() قبل تنفيذ الحوار. أحسن مكان لاستدعائها هو أولا في WinMain(). أما لو قمت باستدعائها في حالة الرسالة INITDIALOG (في دالة الردود (callback)) فإن الأمر سيكون متأخرا، حيث سيفشل الحوار قبل أن يصل إلى هذه النقطة وبالتالي لن يتم استدعاء دالتنا أبدا.

سيقول لك البعض كما ستقول لك الوثائق أن الدالة InitCommonControls() صارت مهملة، ويجب عليك إذن أن تستعمل InitCommonControlsEx(). أقدم على ذلك إذا أردت، لكن InitCommonControls() بسيطة ولا يوجد ما يعيق عملها عند استعمالها.

II - الملحق ب: لماذا ينبغي أن تتعلم الـAPI قبل الـ MFC

الإشكالية

كثيرا ما يأتي العديد من المبرمجين والمبتدئين إلى الـIRC ويسألون هذا السؤال: "ما هو الأفضل: MFC أو API؟"، كما أن الكثيرين يرغبون في القول أن MFC مازال في طور الرضاعة أو أن API كذلك إما بسبب المشاكل التي سببها في أول إصداراته، أو بسبب أشياء أخرى كما يقولون.

مختلف الأقوال تصب في ما يلي:

- ✓ API صعبة جدا
- ✓ MFC مربكة كثيرا
- ✓ API تستلزم كودا مستفيضا
- ✓ MFC منتفخة ومركزة
- ✓ API لا تملك أدوات سحرية
- ✓ MFC صممت بأسلوب سيء
- ✓ API ليست كائنية المنحى (oop)
- ✓ MFC ركلت كلبي ☺
- ✓ API خطفت صديقتي ☺

وهلم جرا...

جوابي

رأيي الخاص، مع أنه ليس الرأي الأوحد، هو أن تستخدم إطار العمل (framework) المناسب للعمل المناسب.

أولا يجب التوضيح ما هي الـAPI وما هي الـMFC. المصطلح API يعني واجهة برمجة التطبيقات (Application Programming Interface)، أما في عالم برمجة الويندوز فإنها تعني المستوى المنخفض من التفاعل القائم بين التطبيقات ونظام التشغيل. فالسواقات (Drivers) مثلا تعتبر ذات مستويات منخفضة، ومختلف الدوال المستدعاة للتعامل معها، لكن بالنسبة للغالبية العظمى من تطويرات النوافذ فالقضية لا تكمن في ذلك. الـMFC هي مكتبة فئة (*Class Library*)، وهي مجموعة من الطبقات المكتوبة بلغة السي للتقليل من العمل المضني الذي يتم تخصيصه لعمل أشياء معينة بدوال الـAPI. تقدم كذلك بنية كائنية المنحى في تطبيقك، وبالتالي الاستفادة منه، أو تجاهله إذا أردت، وهو ما يقوم به أغلب المبرمجين المبتدئين منذ أن صار إطار العمل (framework) لا يستهدف في الواقع تطبيقات من شاكلة: مشغل ملفات mp3، عملاء IRC أو تطبيقات الألعاب.

كل برنامج، سواء كتب بالـ MFC، Delphi، Visual Basic، Perl أو أي واجهة برمجية متطورة أخرى، فبعيدا عن اللغة أو الهيكل الذي يمكن أن تبرمج على أساسه، التطبيق في الأخير يبنى على دوال الـAPI. هذا التفاعل مخفي في أغلب الأحوال، إذن لا تتعامل مع الـAPI مباشرة، فمكتبات الدعم ووقت التنفيذ (runtime) تعمله بدلا عنك.

بعض المبرمجين يسألون: "الـMFC يقدر على عمل كذا وكذا وكذا، فهل تقدر الـAPI على المحاكاة؟؟" الإجابة على مثل هذا التساؤل منطقية: الـMFC يمكنها فقط عمل ما تقدر الـAPI على عمله، لأن الأولى مبنية على الثانية. بينما عمل أشياء بالـAPI من الصفر قد يسفر عن كود معقد وطويل للغاية، بينما كان يمكن عمل ذلك باستغلال الفئات المحضرة مسبقا في مكتبة MFC مما قد يختصر الكود ويسهل صيانته.

إذن، ما هو إطار العمل الأفضل؟

بالنسبة للمبتدئين، وللأشخاص الذين يودون فقط تعلم كيفية البرمجة، فإنني أنصحهم بالبدء بتعلم الـAPI حتى يجد المبرمج راحته في استيعاب أسلوب عمل تطبيقات النوافذ، وأن تكون ملما بكل الميكانيزمات الأساسية حول أشياء معينة مثل: حلقة الرسائل ↺ ، واجهة الجهاز الرسومية ↺ والتحكمات ↺ ، ولما لا المقابس (sockets) وتعدد التريدات (multithreading). هذه المواضيع ستكسبك فهما للأجزاء الجوهرية في عملية بناء تطبيقات الويندوز. هذه المعرفة ستنفعك مستقبلا إذا رغبت في البرمجة في إطار عمل الـMFC، Visual Basic أو أي إطار عمل آخر ترغب في ولوج عالمه لاحقا. هذا مهم على اعتبار أن إطارات العمل هذه (frameworks) لا تدعم كل ما تعمله الـAPI لأنها ببساطة تتخذ نصيبا محدودا مما يدعمه الـAPI تاركة بذلك الباقي لأن أغلب الناس لا يسعون إلى استغلاله. لذا إذا احتجت إلى هذه الجزئيات في تطبيقاتك فإنك ملزم بإضافتها بنفسك، ولا يمكنك أن تعتمد على إطار العمل ليعمله عنك لأنه لم يخصص لها حصة، وبالتالي ففهمك لعمل دوال الـAPI يتيح لك ما تريد والعكس يصير روتينا ألف المبرمجون مصادفته.

لكن، أليس الـMFC أسهل؟ من وجهة نظر معينة تعتبر سهلة لأنها تؤدي العديد من المهام عن المبرمج، هذا يقلل من الكود المكتوب. لكن، كود أقل لا يعني بالضرورة أنه أسهل، وهذا في حالة عدم فهمك لما أنت كاتب، أو كيف يعمل كل هذا الكود وراء الستار. بصفة عامة: المبتدئون الذين يستخدمون السحرة (wizards) لبدء تطبيقاتهم لا يملكون أدنى معرفة عما يعمله معظم الكود الذي تم توليده، ثم يقضي أغلب وقته في البحث عن الأماكن التي سيضيف فيها أشياءه، أو ما هي التغييرات التي سيقوم بها ليحصل على ما يريد. إذا ما ابتدأت برنامجك من الصفر، سواء بالـAPI أو الـ MFC، فإنك لا محالة تعلم أماكن الأشياء لأنك أنت من وضعها، وستستعمل السمات والخصائص التي تفهمها فقط.

عامل آخر مهم، وهو أن معظم من تعلموا الوين32 في بادئ الأمر لم تكن لديهم قاعدة صلبة في أبجديات السي++. محاولة فهم برمجة النوافذ بالـMFC وتعلم السي++ في نفس الوقت سيكون مهمة مضنية. ورغم أنها ليست مستحيلة إلا أنها تمكنك من أن تصير أكثر إنتاجية من أن تبقى تتعلم السي++ أو الـAPI فقط.

إذن أساسا ...

من خلال كل ما تقدم، أظن أنه يستحسن لك أن تتعلم الـAPI وتتعمق فيها حتى تجد نفسك مرتاحا فيها، قادرا على البرمجة، الإبداع، وصيانة الأخلال، ثم تسعى للإبحار في عالم الـMFC إذا ما ظهر أنه يعني لك شيئا ما، ويوفر عليك الوقت فعندها يكون استغلالها أمرا مستحبا.

لكن، وهذا هو الأهم... إذا عملت مباشرة على الـMFC دون الإلمام بالـAPI، ثم يأتي وقت تطلب فيه المساعدة حول مشكلة ما، فالإجابة تكون بمراجعة قواعد الـAPI "مثلا: استخدام مقبض سياق الجهاز (HDC) الموجود في الرسالة WM_CTLCOLORSTATIC"، وعندها ستقول: "ماذا يعني ذلك؟؟"، والسبب هو أنك لا تعرف كيف تقوم بترجمة موضوع من الـAPI إلى الـMFC، لتجد نفسك في ورطة لا مناص منها، وستبطل كل المحاولات لمساعدتك لأنك ببساطة لم تتعلم ما كان يجب عليك تعلمه قبل الخوض في الـMFC.

أنا شخصيا أفضل العمل بالـAPI، لأنه يتناسب مع تطلعاتي، لكن لكي أكتب وسائط برمجية لقواعد البيانات (database) أو أن أضيف مجموعة من تحكمات ActiveX فإنني سأستخدم الـMFC بكل واقعية، لأنه لا محالة سيقلص كثيرا من الكود المراد كتابته، ويعطي أحسن كود نموذجي بدل الوقوع في مشاكل برمجية.

III - الملحق ج: ملاحظات حول ملف المورد

الشيء المزعج الذي اشمأززت منه هو لما قمت بتحويل بيئة التطوير الخاصة بي من بورلاند سي++ إلى الفيجول سي++ هو الطريقة التي تقوم بها البيئة الجديدة في التقاط سكريبتات المورد (ملفات rc.).

في البورلاند سي++ تملك كل الحرية للتحكم في تصميم ومحتوى ملفات rc.، ولما تستخدم محرر المورد ↺ فإن التغييرات المحدثة في المحرر هي الوحيدة التي سيتم تغييرها في ملف المورد. لكن للأسف، في محرر المورد الخاص بالفيجول سي++ سيتم تغيير كل شيء مما يعني أن ملف المورد rc. ستعاد كتابته، وبالتالي قد يضيع أو يتجاهل ما أحدثته بنفسك من تغييرات.

هذا الاختلاف كان في بداية الأمر محبطا للمعنويات، لكن بعد فترة من الوقت استطعت التعامل معه، حيث أنني في الغالب لا أكتب أي قسط من الموارد يدويا، وذلك احتياطا من أدنى تغيير يمكن حصوله من قبل المحرر.

التوافق

تحدي بسيط في هذا الدليل كان جعل ملفات المورد تصرف بشكل حسن في كلتا البيئتين (بورلاند سي++ والفيجول سي++) من دون إحداث أي تغييرات.

في الدليل الأصلي قمت باستخدام بورلاند سي++ معتمدا على تسمية روتينية لملف الرأس (header) الخاص بالمورد، والذي كان project_name.rh. بينما وتحت الفيجول سي++ فإن الاسم الافتراضي لملف رأس المورد هو recource.h، لهذا فإن اعتمدت على هذا المنهاج الجديد في دليلي هذا علما أنه يتوافق كذلك مع بورلاند سي++.

للفضولي، يمكن تغيير اسم المورد الذي يستخدمه الفيجول سي++ من خلال تحرير الملف rc. يدويا وتغيير الاسم مرتين وفي موضعين، المرة الأولى في الموضع الذي تم فيه إدراجه من خلال #include، والمرة الثانية حيث تم احتواءه في مورد TEXTINCLUDE.

المشكلة الثانية هي أن الفيجول سي++ فرضا يحتاج لإدراج الملف afxres.h في ملفات المورد (rc.)، بينما بورلاند سي++ يملك كل ماكروات المعالجة الأولية الضرورية (preprocessor macros) ولا يحتاج بذلك لهذا النوع من الملفات.

شيء مربك آخر، وهو أن الملف afxres.h يتم تركيبه فقط إذا قمت بتركيب مكتبة الـMFC والتي لا يسعى إليها جميع الناس، حتى عندما تقوم بإنشاء تطبيق من نوع الـAPI والذي يحتاج فقط إلى الملف winres.h، وهذا الأخير دائما يركب مع المصرف.

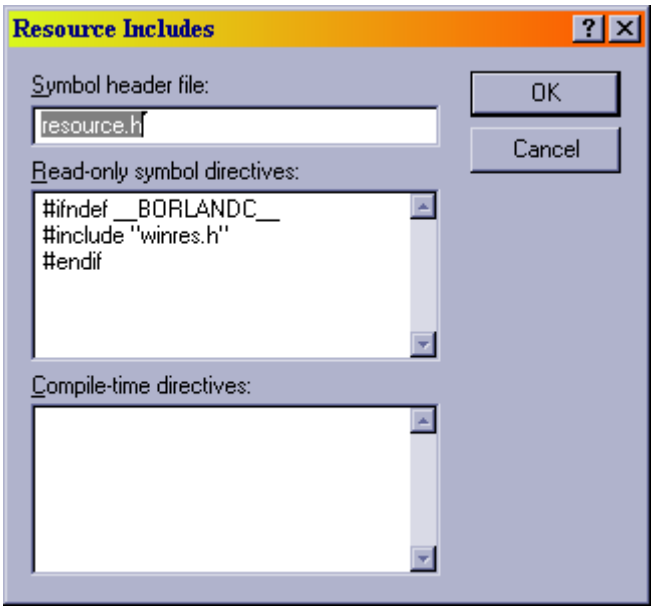
منذ بداية عملي على الفيجول سي++ واستخدامي لمحرر المورد الخاص به فإني وجدت حلا لهذه المشكلة بقليل من التعديل في كل ملف rc. تم توليده م خلال إدراج الأسطر الآتية:

```
#ifndef __BORLANDC__
#include "winres.h"
#endif
```

والذي وفقا للظروف المعرفة فرضا فإنه كان سيقرأ:

```
#include "afxres.h"
```

إذا كنت ممن يستخدم الفيجول سي++، فيمكنك أن تجد خيار تغيير هذا النص في بيئة التطوير المتكاملة (IDE) من خلال القائمة View، وباختيار البند Resource Includes، فإنك ستشاهد هذه النافذة:



عموما، لا يوجد أي حاجة لاستخدام هذا في التمرين العادي، إنما كان ذلك هو السبيل الذي اعتمدته لحل المشكلة، وجعل التطبيق يعمل في كلتا البيئتين.

لأولئك الذين يستخدمون بورلاند سي++، أنا متأسف بخصوص الفوضى التي أحدثتها في ملفات المورد التي يقوم محرر الفيجول سي++ بتوليدها، لكن لا ينبغي أن تتداخل هذه الزيادة مع أي شيء.

تصريف الموارد مع محرر البورلاند سي++

جربت حسب كل قدراتي ولم أصل إلى سبيل بسيط لتصريف برنامج يدرج ملفات من نوع rc. بالبورلاند سي++، وأخيرا تم الإقرار بشكل لا أظنه أنه الأمثل، وستجد ذلك في ملفات الـMakefile المرفقة مع أمثلة هذا الدليل. كما يمكنك مراجعة بعض الملاحظات حول مصرف بورلاند سي++ في أدوات سطر الأوامر المجانية لبورلاند سي++.

لائحة المصطلحات

المصطلح العربي	نبذة مختصرة	المصطلح الإنجليزي
حجز	الحجز غالبا ما يتم على مستوى الذاكرة، وفيه يتم تخصيص مقدار معين من الذاكرة لمتغير أو ثابت، سواء كان بسيطا أو معقدا كالهياكل والفئات	Allocate
وسيط	الوسيط هو كل متغير يشغل مقدارا معينا من الذاكرة ويؤدي مهام محددة لفترة زمنية أقل من فترة البرنامج، ويمكنه أن يكون عدديا محددا أو غير محدد كالمؤشر، كما يمكن أن يكون دالة أو فئة أو هيكلا أو شيئا آخر.	Argument
مصفوفة	المصفوفة هي مجموعة من العناصر التي تشترك في نفس النوع و/أو نفس الحجم، وتكون متصلة فيما بينها في مجموعة واحدة، كما أن كل عنصر من هذه المجموعة له رقم يشير إليه.	Array
ثنائي	يتم تحويل الأعداد في جهاز الكمبيوتر إلى الترقيم التنائي على اعتبار هذا الأسلوب هو السبيل الوحيد للتعامل مع الآلة، وهذا شيء في الحاسب يشار إليه برقم مكون من 1 و 0	Binary
بت	وهو أصغر وحدة لقياس أحجام البيانات، ويأخذ إما قيمة الـــ0 أو الـــ1	Bit
صورة نقطية	صورة على شكل مصفوفة ثنائية البعد، كل نقطة فيها (بكسل) تحمل قيمة تشير إلى اللون الواجب أن تظهر به	Bitmap
بولياني	قيمة منطقية تعني إما صحيح (TRUE) أو خاطئ (FALSE).	Bool
بيفر	اسم يطلق على المنطقة الذاكرية التي تم حجزها، وتخزين قيم عددية فيها.	Buffer
خلل برمجي	مشكلة قد تصادف البرنامج ويتم ملاحظتها أثناء عملية التصريف، وهي نوعان: خلل إملائي وخلل منطقي.	Bug
بايت	قيمة من 8 بتات، ويمكن بها تشفير 256 رمز مختلف، أي 256 حالة ممكنة بعد اللعب على الأصفار والوحدات.	Byte
ردود	مفتاح يسبق بعض الدوال والإجراءات للدلالة على أنها قابلة للاستدعاء من الويندوز. وبمفهوم عام، فالردود تعني إرسال دالة ضمن قائمة بارامترات دالة أخرى، وعند الانتهاء من الدالة الوسيطة فإن التحكم سيعود إلى الدالة الرئيسية.	Callback
تسمية	وهي النص الظاهر على الـــForm أو أي تحكم آخر، وكما هو معلوم فهذا النص ساكن، لا يمكن تبديله من قبل المستخدم إلا عند تمكين هذه الميزة من قبل المصمم.	Caption
التحويل القسري	عملية التحويل القسري يلجأ إليها المبرمج عند عدم إمكانية المصرف على تحويل قيم ذات مستوى مرتفع إلى آخر منخفض، وهذا حتى لا يحدث ضياع للبيانات، إلا أنها أحيانا تكون واجبة للوصول إلى المراد.	Casting

فئة	الفئة شكل مطور من الهيكل، وهي إحدى معالم السي++، تحوي الفئة متغيرات تسمى بالخصائص (attributs)، ودوال تسمى بالمناهج (methods)، كما أنها تقبل الاشتقاق، تعدد الأوجه، الكبسلة ...	Class
قائمة منسدلة	هي تحكم شبيه بالتحرير (Edit)، يحوي زرا على إحدى طرفيه، بالنقر عليه تظهر لائحة من العناصر الغير قابلة للتغيير (في الغالب)، وباختيار إحداها تكون القائمة المنسدلة قد أدت وظيفتها.	Combo Box
تحكمات شائعة	هي تحكمات تعمل كذلك على العديد من أنظمة التشغيل، وقد تم إدراجها تحت خانة "شائعة" لتوفرها في أغلب التطبيقات إن لم نقل كلها.	Common Controls
مصرف	هو برنامج يقوم بتحويل الكود المصدري الذي تكتبه بلغة ذات مستوى عالي، إلى آخر مكون من وحدات وأصفار ليتم التعامل معه من قبل نظام التشغيل، وبعدها يمكنك التخلي عن الكود الأصلي	Compiler
تحكم	وهو عبارة عن كائن (Object) يتم اشتقاقه من فئة، ويتوفر بدوره على خصائصه ودواله الخاصة، ويتم التعامل مع التحكم من خلال جملة من الأحداث التي تمكن المستخدم من استغلاله.	Control
بيانات	وهي كل قيمة ثابتة أو متغيرة يستطيع الحاسب التعامل معها، بغض النظر عن أحجامها أو أنواعها أو الشكل الموجودة عليه.	Data
افتراضي	تكون المتغيرات (أو الهياكل أو أي شيء آخر) يحوي قيم افتراضية إذا قام بخلقها عند عدم حصوله على قيم مدخلة من قبل المستخدم، ويتم غالبا إرفاق قيم افتراضية للبرامج لتسهيل مهمة الابتداء على المستخدم أو تيسير مهمة الاستغلال على من لا يعرف القيم المناسبة.	Default
سياق الجهاز	هو كائن يحوي الرسم الذي نريده أن يظهر على الشاشة، النافذة أو الطابعة، ويغنينا بذلك عن العديد من الأعمال الشاقة كحساب دقة الشاشة، تنوع شاشات العرض، حركة الرسم ... والعديد العديد من الأعمال.	Device Context
تحرير	هو إطار يتيح للمستخدم أن يكتب سطرا من الحروف والأرقام والرموز (أقل من 256 رمز)، ومن أمثلته: الخانات التي تكتب فيها الاسم أو العنوان الإلكتروني...	Edit
محرر	هو عبارة عن نافذة (MDI أو SDI أو MDI) تسمح للمستخدم بكتابة وتعديل وقراءة نص ما، سواء أكان كودا مصدريا أو نصا لغويا...	Editor
عنصر	يقصد به جزء من مجموعة من الأشياء ذات الخصائص المشتركة، وتكون هذه العناصر في الغالب مجموعة في مصفوفة، وكل عنصر له رقم يشير إليه، وقيمة يخزنها لأجل استعمالها لغرض ما.	Element
حدث	ينجم الحدث عن كل ضغطة زر على لوحة المفاتيح أو الماوس، كما ينجم عن حركة من حركات الماوس (بغض النظر عن بقية الملحقات المتصلة بالحاسب)، وعنده يمكن للمبرمج أن يستغل ذلك لإجراء عمليات أو تنفيذ تعليمات تناسب الغرض من التطبيق.	Event
موسع	يتم غالبا إضافة خصائص على دالة أو كائن أو أي شيء آخر، لينعت بعد ذلك بالموسع للإشارة إلى التحديثات الهامة التي أوكلت له.	Extended

مرشح	نقصد بالمرشحات تلك القائمة المنسدلة التي تظهر أسفل مربع الحوار، والتي عند اختيار بند من بنودها فإن النتائج المعروضة تكون بنفس النوع المحدد في المرشح.	Filter
راية	الراية هي قيمة صحيحة تدرج في أغلب الأحوال ضمن لائحة بارامترات لدالة ما للإشارة إلى خاصية من الخصائص، كما يمكن في كثير من الأحيان المزج بين الرايات عن طريق المعاملات المنطقية.	Flag
واجهة المستخدم الرسومية	_ من قاموس الموسوعة هي طريقة في تنظيم شاشة الكومبيوتر وإدارتها. تستخدمها بعض النظم والبرامج كي توفر للمستخدم أسلوبا سهلا لتنفيذ الأوامر وتشغيل البرامج واستخدامها.	Graphical User Interface
مقبض	قيمة صحيحة ومنفردة (unique) تشير إلى كائن ما لتمييزه عن بقية الكائنات، وبها يتعرف البرنامج على هذا الكائن ويتعامل معه.	Handle
رأسي	ملف الرأس هو ملف ذو توسع h. أو hpp.، وفيه يتم تعريف متغيرات، هياكل، دوال وفئات، وهذا الأسلوب يسمح بتنظيم هيكل البرنامج، وبإدراج الملف في أكثر من تطبيق من خلال جعله مكتبة قابلة للتعديل.	Header
أيقونة	هي رسم صغير ذو أبعاد 16×16 بكسل أو 32×32 بكسل، ويتم من خلاله ترميز التطبيقات أثناء العمل من خلال إظهارها في شريط مهام الويندوز، وشريط عنوان التطبيق.	Icon
معرف	هو سلسلة حروف يتم من خلالها تعريف الكائن أو المتغير لكي يقدر المبرمج على التعامل معه، وتمييزه عن بقية متغيراته.	Identificator
فهرس	قد لا يشير هذا المصطلح حقيقة إلى عمله الفعلي، ولكن لأجل تمييزه عن المؤشر (pointer) فإننا نستطيع القول أنه يؤشر على بند من البنود، أو عنصر من لائحة من العناصر، علما أن الفهرس في لغة السي يبدأ من القيمة 0	Index
ابتداء	وتكون عملية الابتداء بإعطاء قيم لا تؤثر على عملية سير البرنامج، ولا تربك المستخدم بقهره على إدخال قيم عددية في كل مرة يشغل فيها البرنامج. والابتداء في الغالب يكون بالقيمة 0	Initializing
مثيل	المثيل هو صورة طبق الأصل لكائن ما. إذ بإمكان كل تطبيق أو كائن أو فئة أو أي شيء برمجي آخر أن يزودنا بمثيلات عنه، إلا إذا كانت عملية إنشاء المثيل تؤثر على السير الحسن للتطبيق.	Instance
صحيح	إشارة إلى مجال معين من القيم العددية الخالية من الفواصل العشرية، وتكون الأعداد سالبة أو موجبة. في الغالب تكون بحجم 4 بايت، أي أنه في الإمكان كتابة أعداد محصورة بين $-(2^{31})$ و $(2^{31})-1$، أو موجبة كلها من 0 إلى $(2^{31})-1$، كما توجد كذلك مجالات أخرى للقيم الصحيحة (2 بايت، 8 بايت، 16 بايت).	Integer

بيئة التطوير المتكاملة	_ من قاموس الموسوعة _ اختصاره IDE وهي بيئة برمجية تتكون من برنامج تطبيقي ومحرر الكود (Code Editor) ومترجم (Compiler) ومصحح الأخطاء البرمجية (Debugger) والقدرة على بناء واجهة رسومية للمستخدم GUI والـ IDE قد يكون تطبيق قائم بحد ذاته أو يكون مضمن مع تطبيقات متشابهة ومكملة. والـ IDE يوفر بيئة سهلة الاستخدام للكثير من لغات البرمجة الحالية من مثل فيجول بيسك وجافا.	Integrated Development Environment
بند	لا يختلف البند عن العنصر إلا برمجيا، أما لغويا فكلاهما يشير إلى عنصر من مجموعة عناصر. فكثيرا ما يشار بالبند إلى أحد عناصر القائمة أو مربعات السرد أو القوائم المنسدلة .	Item
لائحة	عند تجميع العديد من العناصر أو التسجيلات أو أي حزمة من البيانات المشتركة في نفس الخواص فإنها تنعت بلائحة من الأشياء (لغويا قائمة).	List
مربع سرد	مربع السرد هو مربع كبير يحوي عدة بنود تظهر كلها (حسب مساحة المربع) وبعكس القائمة المنسدلة، يمكن اختيار بند واحد أو عدة بنود (عند تفعيل ذلك) وإجراء عمليات عليها.	List Box
حلقة	نعني بذلك تكرار تعليمة أو أكثر عدة مرات إلى أن يتحقق الشرط، وقد يكون هذا التكرار مقيدا أو مطلقا، محدودا أو غير منتهي.	Loop
عضو	البيانات التابعة للفئة أو الهيكل توصف بأنها أعضاء، ولا يمكنها أن تقوم بنشاط ما إلا من خلال الهيكل أو الفئة التابعة له. كما أن بعضها عام يمكن الوصول إليه والبعض الآخر محمي من تعديلات المستخدم.	Member
قائمة	أزرار مجمعة عموديا ومصنفة حسب أدوارها. لا تخلوا تطبيقات الويندوز حاليا منها. تكون القائمة موجودة في الركن العلوي الأيسر (غالبا) من التطبيقات، أسفل شريط العنوان.	Menu
وحدة نمطية	جزء من البرنامج يمكنه أن ينفذ بصفة منفصلة.	Module
مفكرة	برنامج تابع لملحقات الويندوز، يستعمل في تحرير نصوص عربية ولاتينية، ومن فئة: واجهة أحادية المستند (SDI)	Notepad
إشعار	نوع من الرسائل التي يتبادلها النظام مع البرنامج المنفذ، ويقتصر هذا النوع على التحكمات الشائعة، الرسالة تكون من نوع WM_NOTIFY ، بعكس التحكمات القياسية التي تكون رسائلها من نوع WM_COMMAND، ونحتاج لهذا التمييز على مستوى إجراء النافذة **M**.	Notification
البايت الخالي	ببساطة هو القيمة 0 ، وفي لغة السي كل الجمل تنتهي بهذه القيمة، وهذا من أكثر مميزات لغة السي التي تختلف فيها عن لغة البسكال.	Null Byte
إجراء	هي دالة لا تعيد أية قيمة، يمكنها تأدية مهمة داخلية أو أن تقوم بمعالجة مدخلات (بارامترات) وتعديل بعضها. لتصير الدالة إجراء في لغة السي، يكفي جعل القيمة المعادة من نوع void	Procedure

طابور	بما أن حركات الماوس وضغطات أزرار لوحة المفاتيح تبعث رسائل أسرع من أن يعالجها النظام في حينها، فإنه يحفظ كل الرسائل في لائحة، ويعالج الأولى فالأولى، مما يشكل لنا طابورا من الرسائل التي تنتظر المعالجة.	Queue
قراءة فقط	إشارة إلى أي ذاكرة (ديناميكية أو ساكنة) لا يمكن فيها إلا قراءة البيانات دون إحداث تغييرات عليها.	Readonly
مورد	نقصد بالمورد تلك القائمة أو الأيقونة أو المشيرة (Cursor) أو بقية الأدوات المحضرة مسبقا، والتي يمكننا ببساطة إدراجها في تطبيقاتنا وتعديلها حسب رغباتنا.	Resource
ملف المورد	نشير بذلك إلى الملف (.rc) الذي يحوي تصريحات (declarations) الموارد، واصفا إياها للمصرفات وفق منهاج متفق عليه **M**.	Resource File
حافظة الشاشة	_ من قاموس الموسوعة _ هي صور متحركة تنشط وتعرض عند عدم وجود نشاط على الكمبيوتر الشخصي لزمن معين، الهدف الرئيسي منه هو تجنب الاحتراق الداخلي، في الحقيقة الـــCRT تعرض تقنية تمنع الاحتراق الداخلي.	Screen Saver
شريط التمرير	هو شريط يحوي في طرفيه زري إفلات، وفي وسطه مستطيل يتناسب مع حجم الصفحة، كما أنه نوعان: أفقي وعمودي	Scrollbar
ملف المصدر	هو ملف الكود الذي سيقوم المصرف بتحويله إلى ملف شيئي، وهو الملف الذي نستدعي منه كل مواردنا ومكتباتنا في قالب برمجي يختلف من لغة لأخرى.	Source File
ساكنة	بالنسبة للمتغيرات، فإنها تكون ساكنة حالة محافظتها على قيمها السابقة حتى ولو تم التصريح بها على أنها محلية، أما المصفوفات الساكنة فهي التي نحدد أبعادها مسبقا، مما يعني عدم القدرة على تغيير الحجم المحجوز وقت التنفيذ (runtime).	Static
شريط الحالة	هو الشريط الموجود أسفل أغلب النوافذ، ويستخدمه أغلب المبرمجين لوصف حالة التطبيق في حينها، والتغيرات التي تطرأ عليه، كعرض موقع المشيرة على الشاشة، رقم الصفحة، وضعية بعض الأزرار (Ins، Lock Num).	Statusbar
سلسلة حروف	ونقصد بها في لغة السي مجموعة (أو واحد) من الحروف والرموز والأرقام التي تنتهي بالبايت الخالي. قد تكون هذه السلسلة ثابتة أو متغيرة، كما أنه يمكننا مقارنتها مع أخرى، وتقسيمها أو دمجها باستخدام دوال أعدت لذلك.	String
هيكل	هو مجموعة من المتغيرات القياسية أو المعرفة من قبل المستخدم والتي تعمل في قالب جماعي، والهيكل يقبل الاشتقاق ليصير بذلك نوعا جديدا معرفا من قبل المستخدم، كما أنه شبيه إلى حد ما بالفئة. التصريح بالهيكل يكون بتسبيقه بالمفتاح struct.	Structure
نمط	النمط هو الشكل الذي سيكون عليه الخط أو النافذة أو أي شيء آخر، وهو يختلف عن النوع، فالخط Times New Roman مثلا هو النوع، أما النمط فيكون ثخينا، مائلا، مسطرا أو مظللا...	Style

مؤقت	الموقت هو نوع من الدوال يسمح بإنشاء تطبيقات تعمل اعتمادا على أزمنة محددة يمكن تعديلها، وهذا النوع من الدوال يستخدم خاصة في برامج الألعاب والموسيقى	Timer
شريط الأدوات	شريط الأدوات يقع تحت القائمة، ويحوي أزرار مزركشة (كزر الفتح، الطبع، الإغلاق..) تخفف على المستخدم عناء البحث والقراءة في بنود القائمة. تطبيقات الويندوز اليوم لا تخلوا من مثل هذه الأشرطة.	Toolbar
نفاذ انتهاكي	يرسل المصرف هذه الرسالة في حالة محاولة المبرمج النفاذ (access) إلى مناطق من الذاكرة ليس مخولا له بالوصول إليها، أو في حالة عزم التطبيق على الكتابة في عنوان يتعدى المجال المحدد بدوال الحجز الذاكري.	Violation Access
تحذير	ترسل المصرفات تحذيرات إلى المبرمج تذكره فيها مثلا بأن يقوم بابتداء متغيراته قبل أن يستخدمها، كما تنبهه حالة قيامه بتحويل للقيم من احتمال قصور في الدقة (ضياع الجزء العشري مثلا عند التحويل من العدد الحقيقي إلى العدد الصحيح). المبرمجون المحترفون يعتبرون التحذيرات أخطاء برمجية يجب إصلاحها.	Warning