1. Wstęp

Czym jest API ?

API Windows to zestaw funkcji, które pozwalają zbudować praktycznie dowolną aplikację działającą w systemach Windows 95, 98, NT, 2000 oraz CE. Za ich pomocą można manipulować graficznym interfejsem użytkownika, wyświetlać grafikę i sformatowany tekst, można także zarządzać zasobami systemu, takimi jak pamięć, pliki i procesy. Funkcje API zostały zaprojektowane dla programistów używających C\C++, choć możliwe jest oczywiście tworzenie aplikacji także w innych językach wykorzystując funkcje API. Aplikacje, które używają funkcji API wymagają oczywiście, aby był zainstalowany jeden z wyżej wymienionych systemów. Nie wszystkie funkcje API działają na wszystkich systemach serii Windows, niektóre, zwłaszcza te nowszej daty mogą wymagać Windows NT lub 2000. O tym jakie funkcje maja jakie wymagania można się dowiedzieć z różnorakiego bogactwa podręczników i plików pomocy (polecam książkę Petzold-a "Programowanie Windows 95/98/NT", oraz podręcznik pomocy, który jest rozprowadzany razem z pakietem VC++, firmy Microsoft - MSDN. API nie jest językiem. Jest to swego rodzaju "sposób na życie" dla programistów w Windows. Oczywiście można się obejść bez API wykorzystując biblioteki obiektowe, takie jak MFC (Microsoft) czy OWL (Borland) lub jeszcze inne. Jednak ich zastosowanie nie eliminuje złożoności systemu jakim jest niewątpliwie Windows, wcześniej czy póĽniej przyjdzie czas, gdy zmuszeni zostaniemy do sięgnięcia do Ľródeł czyli czystych funkcji API.

Dlaczego warto znać API Windows ?

Pierwsza, najważniejsza korzyść to taka, że poznajemy sposób działania systemu niejako od wewnątrz. Cały Windows opiera swe działanie na kilku bibliotekach dll, w których zawarte są najbardziej potrzebne funkcje API, które my programiści powinniśmy wykorzystywać. Druga korzyść to mały rozmiar programów. Skompilowany exec napisany z wykorzystaniem MFC lub innej biblioteki obiektowej zajmuje nierzadko kilka MB, podczas gdy aplikacja korzystająca tylko z API kilkadziesiąt, kilkaset kilobajtów (oczywiście pod warunkiem, że nie zawiera dziesiątek MB zasobów w postaci bitmap w TRUE COLOR). Aplikacje API nie potrzebują do działania żadnych zewnętrznych bibliotek (oprócz tych, które my jawnie do niej dołączymy i tych znajdujących się w katalogu Windows). Dla przykładu aplikacja wykorzystująca DirectX nie potrzebuje nic, prócz plików dll, które zostają umieszczone w systemie podczas instalacji sterowników DirectX, reszta potrzebnego kodu to biblioteki kernel(32).dll, user(32).dll i GDI(32).dll. Są to trzy podstawowe biblioteki, bez których niemożliwe byłoby działanie systemu Windows. Kernel32.dll odpowiada za funkcjonowanie jądra systemu (zarządzanie pamięcią, operacje I/O i uruchamianie zadań), user32.dll to obsługa interfejsu użytkownika (klawiatura, mysz itp. oraz logika okien). GDI32.dll to interfejs graficzny, który umożliwia nam rysowanie na ekranie lub drukowanie na drukarce.

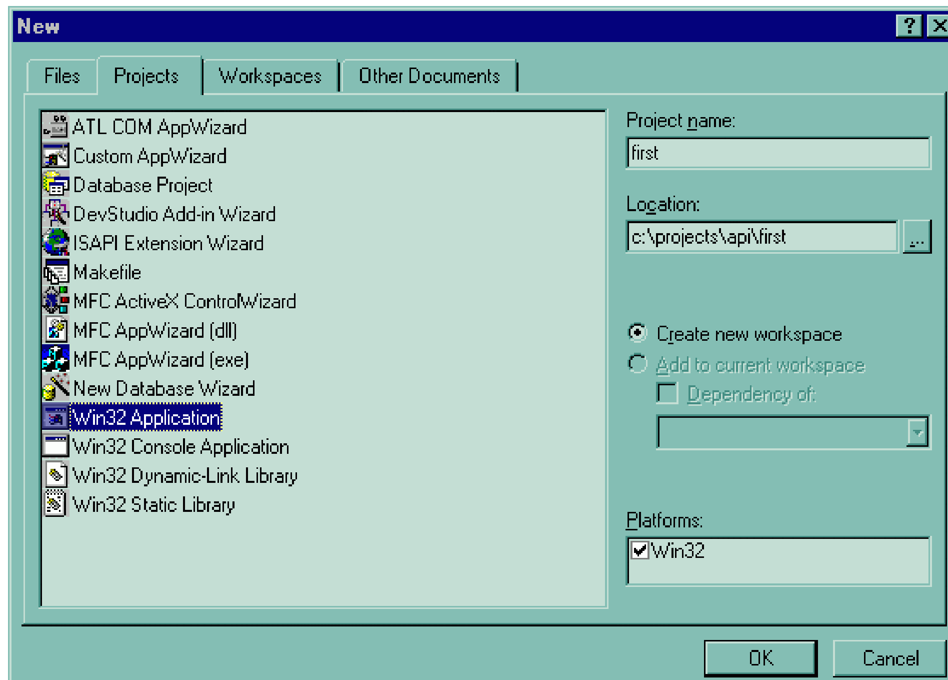
W czym pisać aplikacje korzystające z API?

Wskazane jest posiadanie kompilatora, który umożliwia generowanie 32-bitowego kodu. Dobrym początkiem może być pakiet Microsoftu, Visual C++ Developer Studio, który zawiera nie tylko kompilator, ale wiele narzędzi do tworzenia i zarządzania zasobami, można tez w nim tworzyć aplikacje oparte na bibliotece obiektowej MFC. Polecana wersja to 5.0 lub 6.0, dostępne są tez Service Packi, osobiście używam nr 3 dla ver. 5.0.
To tylko nakreślenie czym jest tak naprawdę to straszne API, w rzeczywistości nie jest to nic trudnego, po kilku próbach można to nawet polubić ;)) Jeżeli zechcecie to przeprowadzimy kilka lekcji, które pozwolą opanować podstawy niezbędne do dalszego programowania w Windows (95 i 98, także NT) z wykorzystaniem API, a po nabraniu odrobiny wprawy postaramy się pokazać jak wykorzystać DirectX i OpenGL, no bo przecież o to nam właśnie chodzi, ale to już inna bajka...

2. Szablon

Przygotowanie projektu

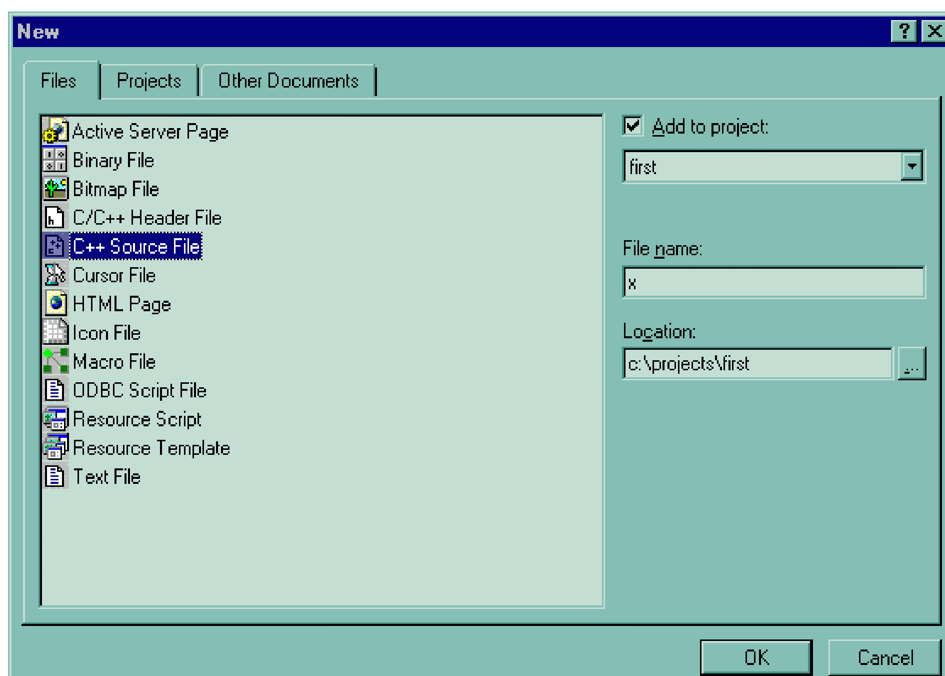
Tak, jak obiecałem poprzednim razem, zajmiemy się dzisiaj pierwszymi programami w API Windows. Ufam, ze kompilator został zainstalowany, ręce rozgrzane, klawiatura w pełni sprawna, obsługa Windows w miarę dobrze znana (ponieważ poprzednim razem zapomniałem dodać, że aby dobrze programować w Windows trzeba choć ociupinkę znać ten system i umieć się nim posługiwać :). Uruchamiamy wiec nasz kompilator (w naszym przypadku VC ver. 5.0), przechodzimy do menu "File", wybieramy opcje "New" i w tym momencie naszym oczom ukazuje się okienko z piekła rodem zawierające masę kolorowych ikonek:



Ma kilka zakładek, wybieramy (jeśli nie jest domyślnie wybrana) "Projects" i widzimy kilkanaście możliwych do utworzenia typów projektów. Nasz cel jest jeden ściśle określony: "Win32 Application", klikamy wiec bez chwili namysłu to pole, po prawej stronie w polu o nazwie "Project name" wpisujemy nazwę naszego projektu, może być np. imię naszej dziewczyny, albo jak kto woli "aqq", itp. wybór należy do Was. W polu "Location" określamy położenie katalogu naszego projektu (czyli folder w którym zostanie umieszczony katalog z naszym projektem). Jak nie trudno zauważyć nazwa naszego projektu to jednocześnie nazwa katalogu w którym zostanie umieszczony, czyż nie cudownie ? Na samym dole po prawej stronie powinno być okienko "Platforms", w którym jedyna dostępna opcja to Win32, domyślnie zaznaczona. Dla własnego (i innych) dobra zostawmy ja tak jak jest (czyli zaznaczona) i klikamy OK. I co ?....... I nic, błysnęło, zamieszało dyskiem i ciągle nic. Posiadacze VC 6.0 w tym momencie zaprotestują, jak to nic? Tu małe wyjaśnienie. Z niewiadomych przyczyn Microsoft dopiero w wersji 6.0 stworzył wizarda, który umożliwia automatyczne generowanie kodu szkieletowego dla aplikacji opartej o czyste API, Ci szczęśliwcy maja wiec z głowy a My musimy się męczyć (ale tylko do czasu). Mamy gotowy projekt, wiec... co dalej ?

Pliki źródłowe

Aby moc cokolwiek skompilować, trzeba by napisać choć kilka linii kodu, no nie? Tak wiec kontynuujmy nasze dzieło co by niepotrzebnie nie tracić czasu. Pierwsze co robimy to menu "Project", w którym wybieramy opcje "Add To Project" a w nim z kolei "New". Co dostajemy ?



Bardzo podobne okienko do tego na początku, z tym, że tym razem aktywna zakładka to "Files" i bardzo dobrze, bo o to nam dokładnie chodzi. Z listy dostępnych typów plików, które możemy dodać do projektu wybieramy "C++ Source File". Po prawej stronie widzimy jedno puste pole domagające się wpisania czegoś. Wpisujemy nazwę pliku, jaki zostanie dodany do projektu, nic nie kombinując w pozostałych okienkach, no bo po co, tak jak jest, jest dobrze. Po kliknieciu OK ukazuje nam się piękny czysty arkusz na który możemy przelać wymysły naszej chorej wyobraĽni.

Kod

A co przelać na ten arkusz, żebyśmy mogli ujrzeć tak długo oczekiwane okienko. Otóż już mówię. Napiszmy np. cos takiego:

```
#include <windows.h>

static TCHAR lpszAppName[] = TEXT( "API Windows" );

LRESULT CALLBACK MainWndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
switch (uMsg)
    {
    case WM_CREATE:
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    case WM_KEYDOWN:
        switch (wParam)
        {
            case VK_ESCAPE:
                PostQuitMessage(0);
                break;
        }
        break;

    default:
        return (DefWindowProc(hWnd, uMsg, wParam, lParam));
    }
    return(0L);
}

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

    wndclass.style  = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = MainWndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance  = hInstance;
    wndclass.hIcon  = NULL;
    wndclass.hCursor  = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = lpszAppName;

    if(RegisterClass(&wndclass) == 0)
        return FALSE;

    hWnd = CreateWindow(
        lpszAppName, lpszAppName,
        WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
        100, 100, 400, 300,
        NULL, NULL, hInstance, NULL);
```

```
    if(hWnd == NULL)
        return FALSE;

    ShowWindow(hWnd, SW_SHOW);
    UpdateWindow(hWnd);

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

Po wklepaniu wciskamy kombinacje Ctrl+F5, albo naciskami czerwonawy przycisk w kształcie wykrzyknika na pasku narzędzi (powinien domyślnie być doczepiony gdzieś tam, pod paskiem menu), zresztą każdy zagorzały programista powinien go mieć zawsze pod ręka. Jeśli nie popełniliśmy żadnych błędów powinniśmy otrzymać piękne białe okno, z paskiem tytułu, klawiszami do zamykania, minimalizowania i maksymalizowania okna. Co robi nasz program ? Jeżeli sądzić po efektach wizualno - dĽwiękowych NIC !. Z drugiej jednak strony robi to, co każda porządna aplikacja w Windows - reaguje na komunikaty. Jeśli jeszcze nie wiecie co komunikaty wyjaśni się to w dalszej części kursu. Na dziś to może tyle no bo nie można się przemęczać przecież, a za to następnym razem omówimy dokładniej budowę naszego programu, dowiemy się co robią funkcje WinMain() i MianWndProc() oraz co to są komunikaty.

3. Szablon - co i jak?

Zgodnie z obietnicą daną poprzednim razem, dzisiaj zajmiemy się omówieniem dokładniej naszego programu, co pozwoli także na nieco głębsze zapoznanie się z istota działania samego systemu Windows. Omówimy sobie poszczególne elementy programu podanego ostatnio jako przykład, powiemy o funkcjach **WinMain()** i **MainWndProc()**, strukturze **WNDCLASS** i pętli komunikatów. Wiec bez zbędnego macenia wody zaczynajmy.

Funkcje, komunikaty i ich pętla

Jeśli przyglądniemy się naszemu programowi z "lotu ptaka", nie wnikając w szczegóły zauważymy, ze mamy w nim tylko dwie funkcje, co może się wydawać początkującym programistom w Windows trochę niesamowite. To jednak prawda, do napisania zupełnie poprawnie pracującej aplikacji w Windz'ie wystarcza tylko te dwie funkcje, choć w aplikacji robiącej rożne pożyteczne rzeczy (np. w takim Quake4) jest to zdecydowanie za mało, to jednak bez tych dwóch ani rusz dalej. Wytężając bardziej wzrok dostrzegamy, ze jedna z nich to **WinMain()** a druga to **MainWndProc()**. Jak nadmieniłem już we wstępie funkcje API zostały napisane z myślą o wykorzystaniu ich w C\C++. W każdym działającym programie w ww. językach podstawa wszystkiego jest funkcja **main()**, a co ona robi nie trzeba chyba nikomu tłumaczyć (jeśli ktoś nie wie niech natychmiast śle maila ! :). Jako ze Windows jest znany powszechnie z tego, ze jest systemem bardzo pokręconym, w którym wszystko działa nie tak jak tego się można spodziewać, oczywiście musiało się to odbić na sposobie pisania aplikacji. Zamiast starej, dobrej funkcji **main()** widzimy tu jakiegoś koszmarnego stwora o nazwie może i nieco podobnej, ale już parametrami i wartością zwracana ni jak nie przypominającego swojego protoplastę z dos-a. Brzmi to mniej więcej tak:

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

brrrrrr !, aż ciarki przechodzą. Nie będę się tu może wgłębiał w drobne szczegóły w różnicach pobieranych typów przez ta funkcje w rożnych przykładowych programach, z którymi przyjdzie się nam zmierzyć, zapewniam jednak, ze zawsze chodzi o to samo. A o co chodzi już wyjaśniam:

`HINSTANCE hInstance` - jest to tzw. uchwyt realizacji (????). Czym z kolei jest uchwyt ? Uchwyt to po prostu liczba, której używa aplikacja do identyfikowania czegoś czego może "dotknąć". W naszym wypadku uchwyt przekazywany do tej funkcji identyfikuje nasz program. Nie musimy jednak zawracać sobie tym głowy bo będziemy z tego bardzo rzadko korzystać.

`HINSTANCE hPrevInstance` - jest to uchwyt (już wiemy co to) do poprzedniej realizacji programu. W Windows możemy uruchomić kilka kopii tego samego programu (o czym może nie wszyscy wiedza :), są to właśnie tzw. realizacje. W starym Windows każdy nowy program uruchamiany z pliku exe korzystał z zasobów programu, który był załadowany jako pierwszy, z tego samego exec'a, natomiast w nowym, tym bardziej pokręconym zrezygnowano z tego i ten parametr jest zawsze ustawiany na **NULL**, wiec nie powinien nas obchodzić w najmniejszym stopniu.

`LPSTR lpCmdLine` - w końcu cos znajomego. Nic innego jak linia poleceń programu. Tu umieszczany jest łańcuch znaków,

który następuje zaraz po nazwie naszego programu, np. "aqq.exe". Przeważnie oczywiście nie wpisujemy żadnych parametrów programu (zwłaszcza, że szczególnie w Windows jest to bardzo utrudnione), ale myślę, ze o tym parametrze warto pamiętać, bo może się kiedyś przydać.

`int nCmdShow` - parametr ten mówi naszej aplikacji w jakiej postaci ma zostać uruchomiony nasz program (dokładniej mówiąc jego okno), czy ma być zminimalizowane na pasku zadań, czy wypełniać pełny ekran czy jeszcze jakoś inaczej.

Można zauważyć, ze niejako bezpośrednio mamy wpływ tylko na linie komend programu, do której bezpośredni dostęp ma użytkownik. Nie możemy podać uchwytu realizacji przy uruchamianiu pliku exe ani sposobu pokazania okna (nie licząc parametrów linii komend). Tak wiec nasz główny wysiłek połóżmy w tym momencie na zawartość tej funkcji. Zanim jednak przystąpimy do tego niezbędne jest wyjaśnienie czym jest ten tajemniczy komunikat i ściśle z nim związana funkcja **MainWndProc()**. A wiec co to takiego ten komunikat ??? Nie jest to nic innego jak informacja docierająca do naszej aplikacji, że cos z nią zrobiliśmy (lub mamy zamiar zrobić tylko jeszcze nie wiemy jak). Najprostsze przykłady to: klikamy myszka w naszym oknie, co robi aplikacja ? NIC, naciskamy klawisze klawiatury, co robi aplikacja ? NIC. Ale, np. gdy wciśniemy kombinacje Alt+F4 to co ? Widać różnicę w zachowaniu się aplikacji ? Gdy dwa razy klikniemy szybko na pasku tytułowym to coś się dzieje ? Oczywiście rozmaitych typów komunikatów jest w Windows cale mnóstwo, my nie będziemy nich wszystkich omawiać bo nie ma oczywiście takiej potrzeby. Po co nam takie komunikaty ? Otóż aplikacje Windows są tzw. aplikacjami sterowanymi zdarzeniami. W samym programie, jeśli patrząc na niego z punktu widzenia programisty dos-a (czyli funkcji **WinMain()** przeważnie nie dzieje się prawie nic, a zarazem bardzo wiele jak za chwile się przekonamy). Cala brudna robota odwalana jest w drugiej funkcji czyli tzw. procedurze okna. Programy sterowane zdarzeniami same z siebie nie robią nic (przynajmniej tak ogólnie biorąc). Po uruchomieniu się i zainicjalizowaniu potrzebnych zmiennych (których w Windows oczywiście nie brakuje) oczekują one na nadejście jakiegoś zdarzenia. Jeśli takowe otrzymają natychmiast zostaje wysłany komunikat do naszej procedury okna. W ten mniej więcej sposób działają wszystkie aplikacje. A co robi taka procedura okna ? Jak potem zauważymy w niej znajduje się cały kod odpowiedzialny za reakcje na nasze komunikaty. Procedura okna to nic innego jak pętla, która wykonuje się przez cały czas działania programu bardzo szybko (w miarę możliwości systemu itd. co nie zawsze jest prawda w Windows :). Po nadejściu jakiegoś komunikatu następuje sprawdzenie czy mamy kod obsługi takiego komunikatu w naszej pętli i jeśli tak jest to jest to wykonywane. Jeśli nie mamy kodu obsługi komunikatu, który nadszedł jest on przekazywany domyślnej funkcji obsługi komunikatów Windows, która przeważnie robi bardzo brzydko i po prostu go ignoruje (co bardzo ładnie nazywa się "domyślną obsługą komunikatu").

No dobrze, skoro mamy już niewielkie pojecie co to komunikat, po co nam funkcje **WinMain()** i **MainWndProc()**, która jak nie trudno się chyba domyślić jest dla nas procedura okna, możemy przystąpić do bardziej szczegółowego omówienia naszego programu.

WinMain

Co takiego znajdziemy w funkcji **WinMain()** ? Przeglądnijmy ja od samego początku. Jak kultura programowania nakazuje, na początku mamy deklaracje zmiennych występujących w programie. No i żeby nie było nam za wesoło to patrzymy i widzimy co ? Ano takie cos:

```
MSG msg;
WNDCLASS wndclass;
HWND hWnd;
```

Wielu w tym momencie łapie się za głowy i pyta co to jest ? Co to za zmienne ??? I tu znowu kilka "słów pociechy" dla przyszłych asów programowania w API. Jeśli chcemy programować w Windows musimy się niestety przyzwyczaić do ogromniastych struktur, które przyjdzie nam bez przerwy wypełniać i odczytywać. W Windows takich struktur jest od gnoma i trochę i co gorsza nie można się bez nich obejść, trzeba się niestety nauczyć ich używania. Dzięki bibliotekom obiektowym, takim jak MFC ich obecność jest może trochę mniej odczuwalna, ale są one nieodłączną częścią systemu i bez nich nie da rady.
Nie będą tu oczywiście omawiał szczegółowo wszystkich struktur występujących w Windows, zaznaczę może tylko do czego niektóre z nich służą. Dokładny ich opis znajdziecie w każdym help-ie traktującym o API Windows.

`MSG` - jest to struktura komunikatu. W strukturze tej są przechowywane wszystkie dane, dotyczące nadchodzącego komunikatu, czyli okno (uchwyt) do jakiego jest skierowany komunikat, jego typ i parametry (tak, tak komunikat może mieć parametry :) i kilka innych mniej istotnych dla nas rzeczy.

`WNDCLASS` - bardzo ważna rzecz o której już za chwile dokładniej.

`HWND` - już nam znany uchwyt (identyfikator okna), czyli jakiś bliżej nieokreślony na razie unikalny numer.

Kilka linii wyżej mamy typ o nazwie **WNDCLASS**, a po kiego diabla nam to potrzebne ?. Już wyjaśniam, Każdego okno uruchomione w Windows, począwszy od okien aplikacji a skończywszy na klawiszach w oknach dialogowych (tak, tak to tez okna :), musi posiadać zarejestrowana w systemie klasę swojego okna. Okna takie jak klawisze, czy okienka dialogowe maja klasę ustalona z góry przez Windows, co nie znaczy wcale, ze nie mogą mieć innej. Natomiast każde okno aplikacji (

okno główne) musi mieć zarejestrowana przez program klasę okna w systemie. Aby zarejestrować w systemie klasę okna używamy funkcji **RegisterClass()**, która pobiera jako parametr adres struktury klasy okna. Zanim jednak zarejestrujemy klasę okna w systemie musimy oczywiście wypełnić wyżej wspomnianą ogromną strukturę typu **WNDCLASS** jakimiś wartościami, które będą miały określony sens. To właśnie robimy min. w funkcji **WinMain()**. Jak widać w naszym przykładzie mamy cos takiego:

```
wndclass.style   = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc = MainWndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.hInstance  = hInstance;
wndclass.hIcon  = NULL;
wndclass.hCursor  = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wndclass.lpszMenuName = NULL;
wndclass.lpszClassName = lpszAppName;
```

Dwa najważniejsze pola w naszej strukturze to pola drugie i ostatnie. Drugie pole: **wndclass.lpfnWndProc = MainWndProc** zawiera adres funkcji, która jest procedurą okna dla wszystkich okien utworzonych na podstawie tej klasy (no bo możemy oczywiście tworzyć wiele okien tej samej klasy i każde ma swoja pętlę obsługi komunikatów). Pole ostatnie: **wndclass.lpszClassName = lpszAppName** to nazwa naszej klasy, którą możemy sobie nazwać dowolnie, jest to tylko łańcuch znaków. Po opis pozostałych pól jak zwykle odsyłam do help-a, no chyba, ze zażyczycie sobie dokładnych opisów, wtedy cos pomyślimy na ten temat :). Pozostałe ważne pola to: style, które zwykle przyjmuje wartości takie jak w naszym przykładzie, a oznacza to mniej więcej tyle, ze okno po każdej zmianie rozmiarów ma być odrysowywane w pionie i poziomie, **hInstance**, które przechowuje uchwyt realizacji (pamiętamy co to uchwyt !!!), **hIcon** i **hCursor** chyba nietrudno się domyślić. **hbrBackground** określa styl pędzla, jakim jest malowane tło okna, **lpszMenuName**, to nazwa identyfikującą menu, umieszczone w pliku zasobów (co to zasoby i o ich rodzajach następnym razem). Tak w skrócie wygląda struktura **WNDCLASS**, którą musimy wypełnić zanim zaczniemy tworzyć okno.

Skoro mamy już wypełnioną strukturę **WNDCLASS**, możemy przystąpić do bardzo ważnej rzeczy, jaka jest rejestracja okna w systemie. Jak już pisałem wyżej robi to funkcja **RegisterClass()**, pobierając jako parametr adres struktury **WNDCLASS**. W naszym przykładzie mamy cos takiego:

```
if( RegisterClass( &wndclass ) == 0 )
  return FALSE;
```

Jak widać, jeśli funkcja zwróci po wykonaniu wartość zero, oznacza to, że nie można utworzyć klasy okna z podanej struktury i w tym momencie kończy się nasza krotka acz burzliwa kariera programisty w Windows, jednak bądĽmy dobrej myśli - po naszym kursie nie ma prawa nic takiego się zdarzyć :).

Skoro mamy już klasę okna, możemy przystąpić do tak długo oczekiwanego momentu czyli utworzenia okna. Robimy to po prostu tak:

```
hWnd = CreateWindow(
    lpszAppName, lpszAppName,
    WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
    100, 100,
    400, 300,
    NULL, NULL, hInstance, NULL );
```

Po wykonaniu funkcja zwraca nam uchwyt nowo utworzonego okna, lub **NULL** jeśli nie utworzy okna (co może się oczywiście zdarzyć :). Pierwszy parametr to nazwa naszej zarejestrowanej klasy okna (ostatnie pole w strukturze **WNDCLASS** !), drugie to nazwa okna (to co pojawi się na pasku tytułowym). Trzeci, straszliwie wyglądający stwór to kombinacja styli okna. Jest ich tez cala masa, nie będę się wgłębiał bo od tego macie dokumentacje, zaznaczę tylko, że style można łączyć używając operatora sumy '|' (OR). Cztery następne argumenty określają początkowe położenie okna na ekranie. Pierwsze dwa to współrzędne lewego, górnego rogu położenia okna, natomiast dwa pozostałe to odpowiednio szerokość i wysokość (liczona oczywiście w dół !) od lewego górnego rogu. Następny parametr (u nas **NULL**) to uchwyt okna rodzica. Jeśli nasze okno jest głównym oknem aplikacji to jest to oczywiście **NULL** (bo nie ma rodzica), jeśli nasze okienko miałoby być "dzieckiem" innego okna, wtedy należałoby tu podać uchwyt tegoż rodzica. Kolejny **NULL** to uchwyt menu lub identyfikator okna dziecka, w naszych zastosowaniach raczej nie używany, więc nie zawracajmy sobie nim głowy. **hInstance** to oczywiście uchwyt realizacji naszej aplikacji (patrz kilka linii wyżej :), natomiast ostatni to wskaĽnik do pewnej struktury. Ma ona min. zastosowanie przy aplikacjach wielodokumentowych (MDI), ale przecież w grach nie będziemy tego stosować (no chyba, ze ktoś wymyśli jakiś sensowny sposób wykorzystania MDI to wtedy to opiszemy :).
Tak oto przebrnęliśmy przez pierwszą, ale za to bardzo ważną, jeśli nie najważniejszą rzecz w programowaniu Windows. Nie jest to jednak koniec funkcji **WinMain()**. Popatrzmy co mamy dalej:

```
ShowWindow( hWnd, SW_SHOW );
UpdateWindow( hWnd );
```

Myślę, ze dla inteligentnych ludzi jest to choć odrobinę zrozumiale. **ShowWindow()**, oczywiście wiadomo, że ma nam pokazać okno, które jest identyfikowane przez uchwyt **hWnd** a rodzajów tego pokazywania mamy...(niech no spojrzę do help-a :), no tak, cos koło trzynastu !. Bardziej ambitni mogą poeksperymentować :).
UpdateWindow() powoduje odmalowanie okna , a dokładniej mówiąc obszaru klienta (wysyła do niego komunikat **WM_PAINT**, co powoduje natychmiastowe wykonanie kodu zawartego w obsłudze komunikatu, ale o tym póĽniej). Po tych liniach mamy już piękne okno na ekranie, ale ma ono jedna jedyna wadę, nie reaguje na komunikaty na co trzeba bezzwłocznie znaleĽć lekarstwo. Po kilkudniowym namyśle może przyjść nam do głowy na przykład takie cos:

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

Jeśli while to wiadomo, ze to... no tak pętla ! I to nie byle jaka pętla, ale pętla komunikatów. **GetMesage()** rozpoczyna pętlę komunikatów. Pobiera ona komunikat z kolejki komunikatów i umieszcza dane komunikatu (parametry :) w strukturze **MSG**. Pozostałe parametry nie będą nas zbytnio obchodzić. Powiem tylko, ze drugi jest uchwytem okna zawsze ustawianym na **NULL**, jeśli przechwytujemy komunikaty dla naszego okna. Tajemnicza funkcja **TranslateMessage()** przekazuje strukturę **MSG** z powrotem do Windows w celu specjalnego przetwarzania komunikatów klawiatury. Kiedyś jeszcze może o tym powiemy, na razie przyjmijmy na wiarę, ze tak ma być i koniec :). **DispatchMessage()** zwraca strukturę **MSG** do Windows i wtedy to właśnie Windows wysyła właściwy komunikat do właściwej procedury okna (u nas **MainWndProc()**). Po obsłudze tego komunikatu w naszej funkcji obsługi, następuje powrót na początek pętli i pobierany jest następny komunikat z kolejki. Dzieje się do czasu nadejścia komunikatu **WM_QUIT**. Kiedy takie cos ma miejsce, funkcja **GetMessage()** zwraca wartość zero a to oznacza co ? No właśnie - koniec pętli, a jak koniec pętli to koniec obsługi komunikatów, a jak koniec obsługi komunikatów to koniec aplikacji, a jak koniec aplikacji to... prawie koniec lekcji na dzisiaj :). Na końcu programu następuje zwrócenie wartości pola struktury **MSG - wParam**, a jako, ze ostatni obsłużony komunikat to **WM_QUIT** to parametr ten oznacza kod wyjścia z programu (patrz help).

MainWndProc

Teraz dowiemy się w jaki sposób obsługiwane są komunikaty w naszej pętli komunikatów. Jak widać w przykładzie funkcja obsługi komunikatów jest zdefiniowana u nas w sposób następujący:

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

LRESULT to znowu nowy specjalny typ (32 bitowa wartość, która może zwracać właśnie procedura okna (min.). Funkcja **CALLBACK** oznacza dla nas tyle, ze funkcja ta jest wywoływana zupełnie bez naszej wiedzy. Zauważmy, ze nigdzie w programie nie ma jawnego jej wywołania, robione jest to cały czas w trakcie jego działania. I cóż takiego ta funkcja pobiera ? Jak widzimy, po pierwsze uchwyt naszego okna (tzn. tego, do którego kierujemy komunikaty), czyli główne okno naszej aplikacji. Po drugie **uMsg**, czyli komunikat (czym jest komunikat nie musimy chyba już nikomu tłumaczyć ?). Tutaj jest przekazywany po prostu numer komunikatu (każdy ma swój) jako liczba **UINT**. Parametry **wParam** i **lParam** to nic innego jak parametry komunikatu :). A cóż takiego mamy w środku naszej funkcji ? Nic prostszego już chyba wymyślić nie można:

```
switch (uMsg)
{
    case WM_CREATE:
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    case WM_KEYDOWN:
        switch (wParam)
        {
            case VK_ESCAPE:
                PostQuitMessage(0);
                break;
        }
        break;

    default:
```

```
        return (DefWindowProc(hWnd, uMsg, wParam, lParam));
}
```

Cóż widzimy ? Najzwyklejsza na świecie instrukcja switch, dla której parametrem jest numer nadchodzącego komunikatu (uMsg). Po nadejściu komunikatu, w naszej procedurze okna dzięki tej instrukcji jest wykonywany odpowiedni kod dla odpowiedniego komunikatu. W tym momencie omówimy sobie kilka podstawowych komunikatów, bez których działanie naszej aplikacji byłoby zgoła utrudnione.
Pierwszy z nich to **WM_CREATE**. Każdy komunikat przeznaczony dla okna aplikacji zaczyna się od liter WM (Window Message), po podkreśleniu następuje jego nazwa, która (najczęściej :) ma nam się kojarzyć z czynnością przy której jest on wywoływany. I tak mamy co następuje:

`WM_CREATE` - wywoływany podczas tworzenia okna (funkcja **CreateWindow()**). W obsłudze tego komunikatu możemy umieścić co nam się tylko podoba (no, może prawie wszystko :), ale najczęściej są to inicjalizacje zmiennych, pobieranie kontekstów rysowania, itp. jednym słowem wszystko co powinno być zrobione na początku.

`WM_DESTROY` - jak się słusznie domyślamy przeciwieństwo komunikatu **WM_CREATE**. Tutaj niszczymy lub zwalniamy wszystko co stworzyliśmy zanim zamkniemy aplikacje. Tutaj mamy tez przykład użycia specyficznej funkcji do wysyłania komunikatów, a mianowicie **PostQuitMessage()**. Funkcja ta wysyła komunikat z procedury okna do... no właśnie :), do procedury okna ??? Przepływ komunikatów w Windows jest trochę dziwny, jednak ma swoja logikę, którą jeszcze będziemy mieli okazje omówić. Na razie poznajmy podstawy, które potem posłużą do poznania tajemnic Windows. Funkcja ta wysyła jeden jedyny komunikat o nazwie **WM_QUIT**, a co on powoduje to pamiętamy chyba z opisu funkcji **GetMessage()** ?

`WM_KEYDOWN` - tym razem jakiś większy stwór, przy okazji którego poznamy następna chora rzecz w Windows a mianowicie cos takiego jak parametry komunikatu.
Jak to działa ? Otóż już tłumacze. Jeśli do aplikacji nadejdzie komunikat, ze nacisnęliśmy klawisz (no bo to w końcu KeyDown), to w wyniku zadziałania instrukcji switch program wchodzi do obsługi komunikatu **WM_KEYDOWN**. Tutaj mamy z kolei następnego switch-a, w którym następuje badanie parametru **wParam**. Wartość ta jest właśnie parametrem komunikatu. Od razu mowie: dla rożnych komunikatów mamy rożne wartości w parametrze **wParam**, wszystko jak zwykle zresztą pisze w Helpie :). Dla komunikatu **WM_KEYDOWN** wyżej wymieniona wartość to po prostu kod klawisza jaki nacisnęliśmy. Jeśli był to klawisz Escape, to zostanie wykonane co ? (Patrz kilka linii wyżej).

I to tyle. W naszej aplikacji obsługujemy tylko 3 komunikaty. A co się dzieje z stekami innych ? (bo naprawdę są ich setki a może nawet tysiące ? :). Te tajemnice pomoże nam rozwiązać funkcja **DefWindowProc()**. Jak widać pobiera ona takie same parametry jak **MainWndProc()** i w rzeczywistości jest tym samym. Jest to tez procedura obsługi okna, z tą małą różnicą, że wykonuje ona domyślną obsługę komunikatów. Co to oznacza ? Otóż, wszystkie nie obsłużone przez nasza aplikacje komunikaty trafiają właśnie do **DefWindowProc()**, która obsługuje je wszystkie na sobie tylko znane, tajemnicze sposoby. Nas jednak cieszy to, ze nie obsłużone komunikaty nie psują nam aplikacji, bo nimi aplikacja zajmuje się już sama bez naszej wiedzy.

I to by było na tyle podstawowej wiedzy na ten raz. Wiemy jak działa aplikacja Windows, co to są komunikaty i ich pętla, wiemy jak je obsługiwać i poznaliśmy pierwsze ogromniaste struktury. Następnym razem zajmiemy się rysowaniem, czyli tym co tygrysy lubią najbardziej :) oraz dowiemy się co to takiego timer i jak to działa. Poznamy tez kilka nowych komunikatów. Oczywiście w razie niejasności proszę o pytania na maila, a postaram się odpowiedzieć jeśli tylko będę wiedział ;). No to na dzisiaj tyle.