

# 2

## ANATOMIA OKNA

*Nauczycielem wszystkiego jest praktyka.*  
Juliusz Cezar

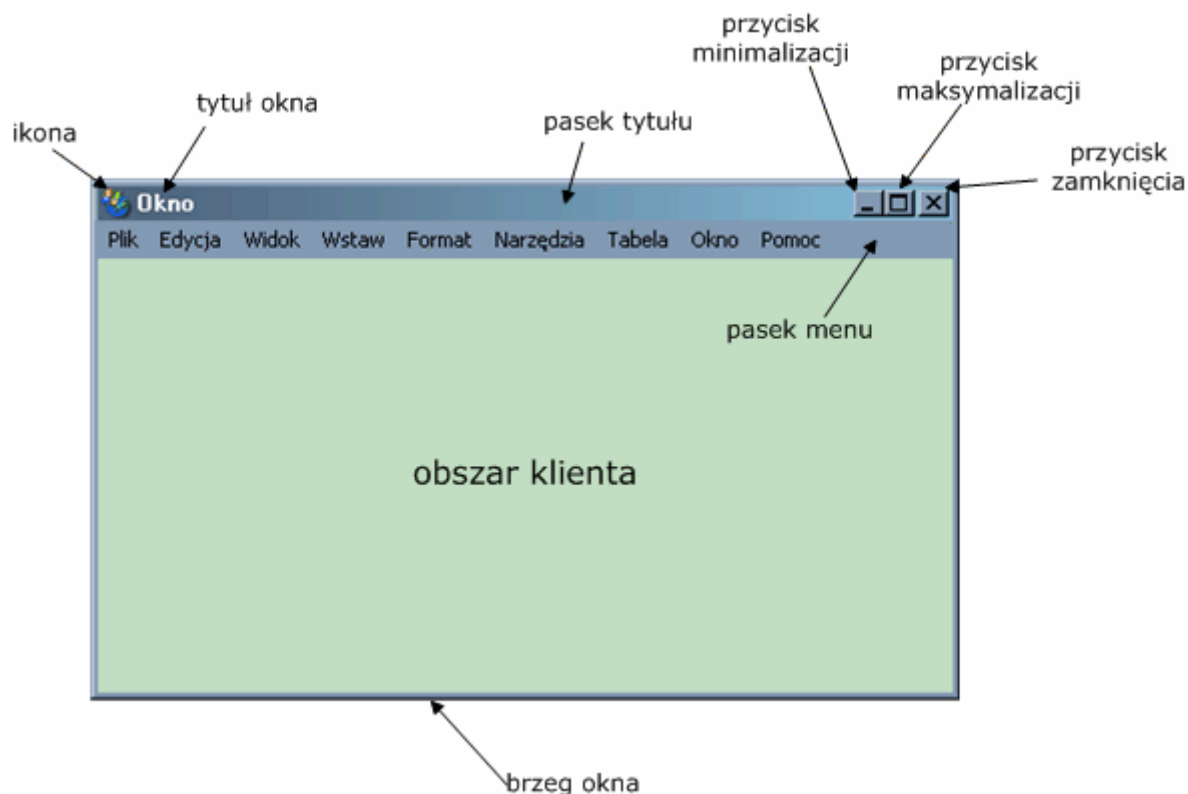
W systemie Windows składniki interfejsu użytkownika nazywamy oknami. Nadmieniałem w poprzednim rozdziale, iż określeniem tym obdarzamy zarówno te prostokątne obszary ekranu, które również użytkownicy nazywają oknami, jak i umieszczone na nich kontrolki w rodzaju przycisków czy pól tekstowych.

Obecnie jednak zajmiemy się tylko oknami w rozumieniu potocznym - dogłębnie poznamy proces ich tworzenia oraz istnienia i interaktywności.

Do nich też i tylko do nich będzie się w tym rozdziale odnosić sam termin 'okno'.

Okna, o których będziemy mówili, są często (szczególnie w wizualnych środowiskach programowania) zwane formularzami, formatkami lub formami (ang. *forms*).

Zanim jeszcze przejdziemy do zaplanowanych zagadnień, musimy sprecyzować sobie kilka(naście) pojęć odnoszących do okien w Windows. Mam przez to na myśli poszczególne elementy okna, przedstawione na poniższym rysunku:



Rysunek 7. Elementy okna

Większość z nich nie powinna być ci obca, jako że nieustannie stykasz się z nimi, używając komputer. Warto aczkolwiek wyjaśnić sobie wszystkie:

- **obszar klienta** lub obszar kliencki (ang. *client area*) zajmuje prawie cały rysunek i nie jest to przypadek: stanowi on bowiem jedyną obowiązkową (!) część okna. Wszystkie pozostałe elementy, zwane łącznie **obszarem pozaklienckim** (ang. *non-client area*), mogą być zmieniane za pośrednictwem styli okna; powiemy sobie o tym dokładnie w dalszym ciągu rozdziału. Czym jest jednak obszar klienta? Otóż jest to właściwa „treść” okna: umieszczane są tutaj kontrolki potomne, tutaj przebiega też rysowanie za pośrednictwem funkcji Windows GDI, ogólnie mówiąc - jest to rejon okna, którego zagospodarowanie leży całkowicie w rękach programisty. System operacyjny zasadniczo „nie wtrąca” się więc ani do wyglądu obszaru klienta, ani do jego zachowania się w reakcji na zdarzenia
- **pasek tytułu** jest umiejscowiony na górze okna. Zgodnie ze swoją nazwą ma on przede wszystkim wyróżniać je spośród innych okien w systemie. Posiada również dodatkową funkcję, mianowicie pozwala na przesuwanie okna po ekranie poprzez proste i intuicyjne przeciąganie. Pasek tytułu zawiera też takie elementy jak:
  - ✓ **ikona** - oprócz efektu wizualnego ma ona także bardziej praktyczny sens: jednokrotne kliknięcie w nią przyciskiem myszy powoduje pokazanie **menu sterowania** oknem, pozwalającego na zmianę jego rozmiarów i położenia, a także na zamknięcie okna (co można też zrobić poprzez dwukrotne kliknięcie w ikonę lub jednokrotne przyciśnięcie prawego przycisku myszy w dowolnym innym miejscu pasku tytułu)
  - ✓ **tytuł okna** zawiera zwykle nazwę aplikacji, jeżeli jest to okno główne programu, lub inny znaczący napis
  - ✓ **przycisk minimalizacji** redukuje okno do postaci przycisku na pasku zadań. Niektóre aplikacje zmieniają aczkolwiek jego działanie i sprawiają, że po kliknięciu w ten przycisk okno pojawia się jako ikona na zasobniku systemowym (ang. *system tray*), obok zegara. Dotyczy to najczęściej programów przeznaczonych do pracy w tle
  - ✓ **przycisk maksymalizacji** rozszerza gabaryty okna tak, że zajmuje ono cały ekran. Ten przycisk może być jednak nieaktywny - jest tak szczególnie wtedy, kiedy okno w ogóle nie pozwala na zmianę swych rozmiarów. Warto też przypomnieć, że dwukrotne kliknięcie w pasek tytułu między ikoną a trójką przycisków także powoduje maksymalizację okna
  - ✓ **przycisk zamknięcia** służy do zakończenia pracy z oknem i zamknięcia go, co w przypadku okna głównego wiąże się z zamknięciem całej aplikacji. Programy pracujące w tle często zmieniają znaczenie tego przycisku na odesłanie okna do zasobnika systemowego; wówczas muszą jednak zapewnić inny sposób zakończenia swej pracy
- **pasek menu** jest obecny najczęściej w głównych oknach programów i zazwyczaj zawiera wszystkie oferowane przez nie funkcje, pogrupowane w logiczne listy (to, na ile są one logiczne, zależy już jednak od umiejętności projektanta aplikacji :))
- **brzeg okna** zamyka jego ramy ze wszystkich stron i w większości przypadków pozwala też na zmianę rozmiarów okna poprzez przeciąganie wybranej krawędzi. Niekiedy jednak okno ma ustalone stałe rozmiary i wówczas brzeg okna jest jego całkowicie statycznym elementem

W tym przeglądzie składników okna należałoby jeszcze zwrócić uwagę na to, że w funkcjonowanie prawie każdego z nich, z wyjątkiem obszaru klienta, w jakiś sposób ingeruje system operacyjny. W przykładzie z poprzedniego rozdziału nie musieliśmy przecież pisać kodu odpowiedzialnego za zmianę wymiarów okna, a mimo to taka zmiana była jak najbardziej możliwa. Podobnie jest z przesuwaniem, maksymalizacją czy minimalizacją - zadania te bierze na siebie sam system Windows, pozwalając jednak programiście na wykonywanie przy ich okazji jakichś innych, własnych czynności.

Po tym krótkim zaprzyjaźnieniu się z oknem i jego elementami możemy już zająć się właściwymi zadaniami, związanymi z tworzeniem okna, różnorodnymi manipulacjami na nim oraz reakcją na najważniejsze komunikaty o zdarzeniach. Temu wszystkiego będzie poświęcony niniejszy rozdział.

## Początki okna

Zdajemy sobie chyba sprawę, że utworzenie własnego okna nie jest wcale tak „proste” jak wywołanie `CreateWindowEx()`. O nie, absolutnie nie jest tak różowo :) Najpierw należy przecież zarejestrować klasę okna (co wiąże się z napisaniem jego procedury zdarzeniowej), a dopiero potem możemy myśleć o wykreowaniu tego własnego „kawałka” interfejsu użytkownika, jakim jest niewątpliwie okno. Proces jego tworzenia składa się więc z dwóch etapów, którym musimy się stanowczo lepiej przyjrzeć.

### *Niemal wszystko o klasie okna*

Jestem prawie pewien, że zadajesz sobie to pytanie: Dlaczego musimy rejestrować klasę okna? Czy nie lepiej byłoby, gdyby jego utworzenie sprowadzało się tylko do wywołania jednej funkcji, przyjmującej może więcej parametrów albo jedną strukturę? Tak byłoby przecież wygodniej, czyż nie?...

Rzeczywiście, pytania te są uzasadnione zwłaszcza teraz, gdy zajmujemy się tylko jednym głównym oknem aplikacji. Jednakże zdajemy sobie chyba sprawę, że nie wszystkie programy tak robią. Ba, większość tworzy kilka kopii swoich okien - weźmy choćby Eksplorator Windows czy niektóre bardziej zaawansowane edytory tekstu. Nie wspomnę już o tym, że kontrolki takie jak przyciski czy pola tekstowe również posiadają swe własne klasy (poznamy je wkrótce) i tym przypadku jest to ogromna korzyść dla każdej aplikacji, która nie musi się zajmować każdym szczegółem GUI z osobna; zrzuca ten obowiązek na system Windows właśnie za pomocą wbudowanych klas okien.

A zatem klasy okien nie są wcale stworzone ku pogębieniu programisty, lecz raczej dla jego wygody. Niezależnie od tego, jak jest naprawdę, powinniśmy dokładnie omówić wszystkie cechy owych klas, przechowywane w strukturze `WNDCLASSEX`. I tym się zaraz zajmiemy; mam nadzieję, że pamiętasz, jakie pola zawiera ta struktura :) Ewentualnie możesz je sobie przypomnieć, zaglądając do poprzedniego rozdziału.

Teraz jednak przejdźmy już do rzeczy.

### *Dwa kluczowe pola*

Spśród wszystkich pól `WNDCLASSEX` bodaj najważniejsze są te dwa: `lpszClassName` i `lpfnWndProc`. Sądząc po ich nazwach w notacji węgierskiej, oba są wskaźnikami - ale na tym podobieństwa się kończą.

#### *Nazwa klasy okna*

`lpszClassName` przechowuje **nazwę klasy** w postaci stałego łańcucha znaków typu `C`, a więc zmiennej typu `const char*` (albo `const wchar_t*`), lub też, będąc w zgodzie z nazewnictwem nagłówek Windows, typu `LPCTSTR`. Nazwa ta powinna oddawać charakter przyszłych okien klasy - szczególnie rolę, jaką będą pełnić w programie. Dobrze też, aby nazwa ta była możliwie krótka.

#### *Procedura zdarzeniowa*

Drugim bardzo ważnym polem struktury `WNDCLASSEX` jest `lpfnWndProc`. Pole to ma zawierać **wskaźnik do procedury zdarzeniowej**, jaka będzie początkowo odpowiedzialna za reakcję na zdarzenia powstające w oknach klasy. Jest to więc zwykły

wskaźnik na funkcję w C++, ze wszystkimi tego konsekwencjami. Typ `WNDPROC`, jakim legitymuje się to pole, jest zaś zdefiniowany jako:

```
typedef LRESULT (* CALLBACK WNDPROC)(HWND, UINT, WPARAM, LPARAM);
```

Funkcja, jaką przypiszemy do `lpfnWndProc`, musi się zatem ściśle zgadzać z prototypem procedury zdarzeniowej, który podaliśmy sobie w zeszłym rozdziale. Pod to wymaganie podpadają oczywiście właściwie zdefiniowane funkcje globalne, ale dobrze jest też pamiętać, iż procedurą zdarzeniową może być również **statyczna metoda klasy**. Wiedza o tym staje się bowiem użyteczna w momencie, gdy chcemy napisać obiektową otoczkę na funkcje Windows API dotyczące okien.

### Instancja aplikacji

Nie mniej istotny niż te dwa pola jest także uchwyt `hInstance`. Identyfikuje on **instancję programu**, która zarejestrowała daną klasę okna. Można spytać: po co systemowi taka informacja?

Otóż przy jej pomocy może on w odpowiednim czasie (po zakończeniu programu) wyrejestrować naszą klasę. Nie zmusza nas w ten sposób do wywoływania funkcji `UnregisterClassEx()`, która tym się zajmuje. Mimo to niektórzy przywołują ją tuż przed zwróceniem wyniku przez `WinMain()`, aby zachować pozory doskonałego porządku ;D Nie jest to jednak konieczne.

### Styl klasy okna

Następne pole `style` określa niektóre dodatkowe ustawienia, które łącznie nazywamy **stylem klasy okna**. To pole jest kombinacją flag bitowych wybranych między innymi spośród tych przedstawionych w tabeli:

<i>flaga</i>	<i>znaczenie</i>
<code>CS_DBLCLKS</code>	Ustawienie tej flagi powoduje, że okno należące do klasy będzie otrzymywało komunikaty o <b>dwukrotnych kliknięciach</b> myszą, zachodzących w jego obrębie (zarówno w obszarze klienta, jak i poza nim). Bez tej flagi okno będzie mogło reagować tylko na pojedyncze kliknięcia przycisków myszy.
<code>CS_DROPSHADOW</code>	Flaga ta, działająca póki co tylko w Windows XP, powoduje włączenie dla okien klasy wizualnego efektu polegającego na <b>rzucaniu półprzezroczystego cienia</b> - pod warunkiem, że użytkownik nie wyłączył tego typu efektów w Panelu Sterowania.
<code>CS_NOCLOSE</code>	Przy tej fladze włączonej okna należące do klasy mają <b>nieaktywny przycisk zamknięcia</b> . Zwykle muszą więc udostępniać inne sposoby zakończenia pracy.
<code>CS_OWNDC</code>	Flaga ta powoduje, że każde okno posiada swój <b>własny, stały kontekst urządzenia</b> . Jest to przydatne przy rysowaniu na powierzchni okna, do którego ten kontekst jest potrzebny; o rysowaniu powiemy sobie co nieco w tym rozdziale, a wyczerpująco w rozdziale o Windows GDI.
<code>CS_CLASSDC</code>	Ta flaga, wykluczająca poprzednią, sprawia, że wszystkie klasy okna dzielą <b>wspólny kontekst urządzenia</b> .
<code>CS_HREDRAW</code>	Włączenie tej opcji wymusza odrysowywanie całej zawartości okna po zmianie jego szerokości.
<code>CS_VREDRAW</code>	Flaga podobna do poprzedniej, tyle że powoduje odrysowywanie całego obszaru klienta po zmianie wysokości okna.
<code>CS_SAVEBITS</code>	Ta opcja sprawia, że <b>zachowywaniem wizualnej zawartości okna</b> zajmuje się sam system Windows. Otóż dla każdego okna utrzymuje on dodatkową bitmapę, którą wykorzystuje w momencie odtwarzania

<i>flaga</i>	<i>znaczenie</i>
	wyglądu okna. Jego odrysowywaniem nie jest więc wtedy obciążona aplikacja, ale w zamian zostaje zajęta pewna część pamięci operacyjnej, potrzebna dla przechowywania wspomnianej bitmapy. Z tego względu styl <code>CS_SAVEBITS</code> działa efektywnie tylko dla małych okien.

**Tabela 26. Flagi bitowe stylów klasy okna**

Spora część stylów okna dotyczy kontroli jego wyglądu na ekranie, czyli procesu, który nazywamy tutaj **odrysowywaniem**. Polega on na wysyłaniu do okna komunikatu `WM_PAINT` w chwili, gdy ma ono pokazać konkretny fragment swego obszaru klienta. Procedura zdarzeniowa może wtedy obsłużyć ten komunikat i wykonać odpowiednie czynności, zazwyczaj przy użyciu interfejsu graficznego GDI. Rozwiązanie to sprawia, że system operacyjny nie musi składować „fotografii” bieżącego stanu każdego okna i oszczędza w ten sposób mnóstwo pamięci RAM. Dotyczy to aczkolwiek tylko tych okien, których klasy nie zawierają stylu `CS_SAVEBITS`.

Co do samego odrysowywania, to na ten temat wypowiem się szerzej przy okazji prezentacji komunikatu `WM_PAINT`.

## *Ikony i kursor*

Pola `hIcon`, `hIconSm` i `hCursor` przechowują uchwyty do trzech ważnych dla okna obrazków.

`hIcon` i `hIconSm` określają odpowiednio: dużą i małą ikonę okna. Ta druga jest zwykle wyświetlana w jego lewym górnym rogu oraz na pasku zadań. Duża ikona pojawia się natomiast po wciśnięciu kombinacji klawiszy `Alt+Tab`, służącej przełączaniu się między programami.

`hIcon` powinno zawierać uchwyt do obrazka mającego wymiary co najmniej  $32 \times 32$  pikseli, zaś `hIconSm` musi wskazywać na ikonę o rozmiarze przynajmniej  $16 \times 16$  pikseli. W rzeczywistości obrazki te mogą być większe, a więc w praktyce obu polom przypisuje się często ten sam uchwyt do dużej ikony ( $32 \times 32$ ). Jest to jak najbardziej poprawne rozwiązanie.

Ten sam skutek daje też wyzerowanie pola `hIconSm`.

Pole `hCursor` utrzymuje z kolei uchwyt do obrazka kursora myszy. Kiedy wskaźnik komputerowego gryzonia zatrzyma się na oknie przynależnym tworzonej klasie, wówczas kursor przyjmie wygląd podanego tutaj obrazka. Może to być własna bitmapa, ale najczęściej stosuje się standardową strzałkę lub jeden z pozostałych kursorów systemowych.

Wczytywanie obrazków ikon oraz kursorów nie jest wcale banalnym zadaniem i dlatego warto przedyskutować je dokładnie, co też uczynimy tutaj.

## *Krótkie wprowadzenie do zasobów*

Ikony oraz kursory są dość specyficznymi rodzajami obrazków, gdyż są w zasadzie niezbędne do działania aplikacji w Windows. Dlatego też system operacyjny umożliwia przechowywanie ich w samym pliku wykonywalnym EXE w postaci tzw. **zasobów** (ang. *resources*).

Zagadnienie zasobów (którymi mogą być nie tylko bitmapy) jest na tyle interesujące, że poświęcimy mu jeden z przyszłych rozdziałów. Na razie jednak powinniśmy wiedzieć tylko, że każdy zasób programu jest identyfikowany poprzez unikalną liczbę całkowitą lub łańcuch znaków; dla wygody liczbom nadaje się w kodzie znaczące nazwy stałych.

Identyfikatory zasobów są ustalane przez programistę piszącego tzw. **skrypt zasobów** (ang. *resource script*), który po kompilacji i linkowaniu staje się częścią gotowego programu, wraz z samymi zasobami. Program może teraz, w trakcie swego działania, sięgać do zapisanych w swoim pliku EXE zasobów i „wyciągać” z nich chociażby obrazki ikon czy kursorów. Wykorzystuje je potem na przykład przy rejestrowaniu klas okien.

Do wczytywania tych obrazków z zasobów służą spotkane już wcześniej funkcje `LoadIcon()` i `LoadCursor()`. Istnieją wszakże również inne sposoby na uzyskiwanie takich bitmap - o nich też sobie powiemy.













### *Ikony małe i duże*

Najprostszą drogą uzyskania uchwytu do ikony okna jest użycie funkcji `LoadIcon()`:

```
HICON LoadIcon(HINSTANCE hInstance,
                LPCTSTR lpIconName);
```

Żąda ona dwóch parametrów, z których pierwszy określa uchwyt instancji naszego programu; podajemy go, gdy chcemy wczytać ikonę z zasobów. Jeśli natomiast zadowolamy się jedną ze standardowych ikon systemu Windows (a tak będzie jeszcze przez jakiś czas), wpisujemy tutaj wartość `NULL`.

Drugi parametr to identyfikator wczytywanego zasobu. Windows stosuje całkiem pomysłowy sposób, który pozwala przekazać tutaj zarówno liczbę, jak i napis w stylu C. Dla nas jednak ważniejsze jest to, że możemy tu podać także jedną ze stałych odpowiadających standardowym ikonom systemowym:

<i>stałe</i>	<i>ikona w Win 9x</i>	<i>ikona w Win XP</i>	<i>uwagi</i>
IDI_APPLICATION			domyślna ikona aplikacji; tą ikoną są opatrzone pliki EXE, które nie mają w swoich zasobach żadnych innych ikon
IDI_ASTERISK IDI_INFORMATION			są to te same ikony, które spotkaliśmy przy okazji omawiania funkcji <code>MessageBox()</code>
IDI_ERROR IDI_HAND			
IDI_EXCLAMATION IDI_WARNING			
IDI_QUESTION			
IDI_WINLOGO			w Windows XP logo systemu zostało zastąpione przez tą samą ikonę, której odpowiada stała <code>IDI_APPLICATION</code>

**Tabela 27. Standardowe ikony okien w Windows**

Wybór nie jest zbyt duży, ale dla potrzeb nauki Windows API okaże się chyba wystarczający :) Gdy zaś nauczysz się korzystać z zasobów (a może i wcześniej?...), wówczas upiększysz swoje okna dowolnymi ikonami.

### *Kształt kursora*

Analogiczną do `LoadIcon()` funkcją, wczytującą obrazek kursora, jest `LoadCursor()`:

```
HCURSOR LoadCursor(HINSTANCE hInstance,
                    LPCTSTR lpCursorName);
```

Takie samo znaczenie mają również jej parametry. W przypadku pierwszego z nich będziemy jednak częściej niż w `LoadIcon()` wpisywać `NULL`, ażeby skorzystać z jednego ze standardowych kursorów:

<i>stała</i>	<i>nazwa kursora</i>	<i>obrazek kursora</i>	<i>uwagi</i>
IDC_ARROW	wybór normalny		standardowy i domyślny kursor Windows
IDC_WAIT	zajęty		kursor ten pojawia się, gdy wykonywane jest jakieś pracochłonne zadanie, które nie pozwala na normalną pracę aplikacji
IDC_APPSTARTING	praca w tle		ten kursor wskazuje na wykonywanie jakiegoś zadania, które nie zakłóca zbytnio użytkownika programu
IDC_HELP	wybór Pomocy		tym kursorem wskazujemy element interfejsu użytkownika, na temat którego chcemy uzyskać pomoc kontekstową
IDC_CROSS	wybór precyzyjny		może służyć jako celownik ;-)
IDC_NO	niedostępny		pojawia się przy przeciąganiu w niedozwolone miejsce
IDC_IBEAM	wybór tekstowy		wybór miejsca w polu tekstowym
IDC_HAND	wybór łącza		pokazuje się, gdy przywieziemy mysz nad hiperłącze
IDC_UPARROW	wybór alternatywny		
IDC_SIZEALL	przenieś		ukazuje się nie tylko przy przenoszeniu, ale i przy zmianie rozmiarów we wszystkich czterech kierunkach (np. w edytorach 3D)
IDC_SIZENESW	zmiana rozmiaru po przekątnej slashowej		kursory te pojawiają się, kiedy chcemy zmienić rozmiar okna
IDC_SIZENS	zmiana rozmiaru pionowego		
IDC_SIZENWSE	zmiana rozmiaru po przekątnej backslashowej		
IDC_SIZEWE	zmiana rozmiaru poziomego		

**Tabela 28. Standardowe kursory Windows (nazwy z Panelu Sterowania lub własne)**

Prawie zawsze wybierać będziemy zwykłą strzałkę, czyli wariant `IDC_ARROW`.

### *Lepszy model*

Funkcje `LoadIcon()` i `LoadCursor()` mogą się słusznie wydawać ograniczone. Teoretycznie zostały one nawet zastąpione przez inną funkcję, `LoadImage()`:

```
HANDLE LoadImage(HINSTANCE hInstance, // uchwyt instancji zasobów
                 LPCTSTR lpszName,    // nazwa zasobu lub pliku
                 UINT uType,          // typ obrazka
                 int cxDesired,       // docelowa szerokość
```

```
int cyDesired,           // docelowa wysokość
UINT fuLoad);          // flagi wczytywania
```

Już na pierwszy rzut oka wydaje się ona bardziej skomplikowana, a zatem musi oferować większe możliwości - i tak rzeczywiście jest. Dokładne omówienie tej funkcji nie jest nam jednak teraz potrzebne, jako że dokonamy go przy opisie Windows GDI. Skoncentrujemy się raczej na dwóch zagadnieniach: sposobie, w jaki `LoadImage()` zastępuje obie opisane wcześniej funkcje oraz wczytywaniu obrazków ikon i kursorów z plików na dysku.

Dociekliwi mogą naturalnie zajrzeć do MSDN po kompletny [opis funkcji `LoadImage\(\)`](#).

A więc - żeby wczytać ikonę i kursor dla klasy okna możemy użyć instrukcji podobnych do tych:

```
KlasaOkna.hIcon = (HICON) LoadImage(NULL, IDI_APPLICATION, IMAGE_ICON,
                                     0, 0, 0);
KlasaOkna.hCursor = (HCURSOR) LoadImage(NULL, IDC_CURSOR, IMAGE_CURSOR,
                                          0, 0, 0);
```

Widzimy, że pierwsze dwa parametry `LoadImage()` są identyczne jak w przypadku `LoadIcon()` i `LoadCursor()`. Trzeci parametr określa typ wczytywanego obrazka, a użyte w nim stałe `IMAGE_ICON` i `IMAGE_CURSOR` oznaczają odpowiednio ikonę i obrazek kursora (jest jeszcze `IMAGE_BITMAP`, wskazująca na zwykłą bitmapę). Pozostałe parametry są zaś wyzerowane, zatem Windows przyjmie dla nich wartości domyślne.

Na koniec, po wywołaniu funkcji `LoadImage()`, musimy jeszcze rzutować wartość, którą ona zwróci. Jest to bowiem ogólny uchwyt typu `HANDLE`, natomiast my potrzebujemy bardziej szczegółowego rodzaju: uchwytu do ikony (`HICON`) oraz do kursora (`HCURSOR`). Odpowiednie rzutowanie załatwia więc tę drobną sprawę.

Nietrudno zmiarkować, że użycie `LoadImage()` jest nieco bardziej kłopotliwe niż dwóch poznanych wcześniej funkcji. Możemy przeto zrezygnować z niego, gdy zależy nam tylko na wczytaniu ikony lub kursora z zasobów programu czy ze zbiorów systemowych. Jeśli jednak zamierzamy odczytać obrazek z pliku graficznego na dysku, wówczas nie mamy już takiego wyboru; zobaczymy zatem, jak należy wtedy postąpić.

Oto instrukcja wczytująca ikonę okna z pliku dyskowego:

```
KlasaOkna.hIcon = (HICON) LoadImage(NULL, "C:\\Windows\\ikona.ico",
                                     IMAGE_ICON, 0, 0, LR_LOADFROMFILE);
```

Nazwę tego pliku podajemy w drugim parametrze, a oprócz niej musimy jeszcze poinformować funkcję o tym, skąd chcemy uzyskać ikonę - robimy to poprzez flagę `LR_LOADFROMFILE` w ostatnim parametrze.

## Tło okna

To co nazywamy potocznie tłem okna jest tak naprawdę tłem jego **obszaru klienta**, albo raczej sposobem jego wypełniania. W Windows GDI za graficzne wypełnianie jakiegoś kształtu odpowiada obiekt zwany **pędzlem** (ang. *brush*). Może on definiować nie tylko jednolity kolor, ale i dwubarwny deseń czy nawet kafelkowanie bitmapą. Wszystkie te sposoby wypełniania można zaś stosować do obszaru klienta okna: należy jedynie podać uchwyt do odpowiedniego pędzla w parametrze `hbrBackground` struktury `WNDCLASSEX`.

Skąd jednak wziąć potrzebny pędzel? Zasadniczo są po temu trzy sposoby.

Pierwszym z nich jest posłużenie się stałą, za którą kryje się jeden z kolorów systemowych Windows. Kolory te ustala użytkownik w Panelu Sterowania wedle własnych upodobań, a spośród nich najbardziej interesujący dla nas jest ten, który został wybrany dla wnętrza okien. Reprezentuje go stała `COLOR_WINDOW`; możemy więc użyć jej tak:








```
KlasaOkna.hbrBackground = (HBRUSH) COLOR_WINDOW;
```

Drugim nierzadko spotykanym sposobem jest wykorzystanie jednego z globalnych obiektów pędzli, które Windows utrzymuje dla wygody programisty. Uchwyt do któregoś z tych pędzli można uzyskać za pośrednictwem funkcji `GetStockObject()`:

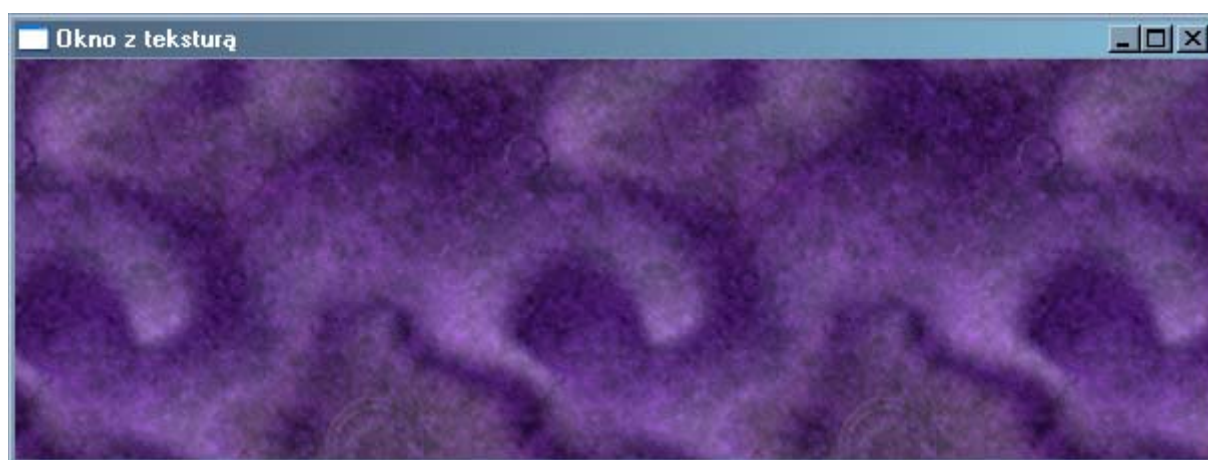
```
KlasaOkna.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
```

Powyższa linijka sprawi, że okno będzie wypełnione jednolitym kolorem białym - wskazuje na to parametr `WHITE_BRUSH`. Wszelako istnieją także inne pędzle do wyboru (wszystkie jednokolorowe) - przedstawia je poniższa tabelka:

<i>stała</i>	<i>nazwa koloru</i>	<i>kolor</i>
BLACK_BRUSH	czarny	
DKGRAY_BRUSH	ciemnoszary	
GRAY_BRUSH	szary	
LTGRAY_BRUSH	jasnoszary	
WHITE_BRUSH	biały	

**Tabela 29.** Kolorowe pędzle dostępne poprzez funkcję `GetStockObject()`

Nareszcie, właściwy pędzel dla wypełnienia okna możemy stworzyć sobie sami. Jest to najbardziej elastyczne rozwiązanie, jako że mamy przy nim dostęp do wszystkich możliwości pędzli, jakie oferuje Windows GDI - nie musimy chociażby ograniczać się do jednolitego koloru pędzla. Spójrzmy na przykład na okno wypełnione wzorem obrazków ściśle przylegających do siebie:



**Screen 57.** Okno wypełnione sąsiadującymi kopiami bitmapy

Efekt ten można osiągnąć w całkiem prosty sposób. Należy w tym celu stworzyć po prostu odpowiedni pędzel:

```
// PatternWindow - okno wypełnione kafelkami bitmapy
// (fragmenty funkcji WinMain())

HBITMAP hBitmapa;
HBRUSH hPedzelOkna;

// tworzymy pędzel wypełnienia okna
hBitmapa = (HBITMAP) LoadImage(NULL, "pattern.bmp", IMAGE_BITMAP,
                                0, 0, LR_LOADFROMFILE);
hPedzelOkna = CreatePatternBrush(hBitmapa);
```

Tworzymy go przy pomocy funkcji `CreatePatternBrush()`. Jest ona jedną z kilku funkcji Windows GDI służących temu celowi, które omówimy dokładnie przy okazji rozdziału poświęconego tej bibliotece. Funkcja żąda uchwytu do bitmapy, która ma być „kafelkowana”; bitmapę tą wczytujemy znaną już metodą poprzez funkcję `LoadImage()`, podając jej nazwę pliku graficznego (plik ten jest dołączony do programu przykładowego).

Mając zaś uchwyt do pędzla, przypisujemy go do składowej `hbrBackground`:

```
KlasaOkna.hbrBackground = hPedzelOkna;
```

Dalej możemy już zwyczajnie zarejestrować klasę okna i stworzyć samo okno. Po zakończeniu pętli komunikatów, tuż przed zwróceniem wyniku funkcji `WinMain()`, musimy jeszcze pamiętać o usunięciu pędzla oraz skojarzonej z nim bitmapy:

```
DeleteObject (hPedzelOkna);
DeleteObject (hBitmapa);
```

Nie jest to trudne - wystarczy posłużyć się procedurą `DeleteObject()`, podając jej oba uchwyt.

### *Pozostałe składowe*

Pozostały jeszcze trzy pola struktury `WNDCLASSEX`, a skoro zostawiliśmy je na koniec, to pewnie nie są zbyt ważne ;) Faktycznie, zwykle wpisuje się w nich zera.

`lpszMenuName` jest identyfikatorem zasobu paska menu, które to menu będzie posiadało każde okno danej klasy. Otóż tak się składa, że menu jest jednym z rodzajów zasobów Windows, zapisywanym wraz z kodem aplikacji (ale poza nim) w pliku EXE. W polu `lpszMenuName` podajemy więc identyfikator tego zasobu; obowiązują tu takie same zasady, jak przy podawaniu podobnych identyfikatorów w funkcjach `LoadImage()`, `LoadIcon()` czy `LoadCursor()` - identyfikator może więc być łańcuchem znaków albo liczbą.

Najczęściej aczkolwiek wpisujemy tu `NULL`, nawet jeśli docelowo okno ma posiadać jakiś pasek menu. Można go bowiem stworzyć innymi drogami.

Ostatnie dwa pola - `cbClsExtra` i `cbWndExtra` - specyfikują ilość bajtów dodatkowej pamięci, jaką system Windows zaalokuje (odpowiednio) dla całej klasy i dla każdego jej okna. Niestety, trudno posądzić te pola o większą przydatność, skoro maksymalna ilość takiej pamięci to „aż” 40 bajtów (!). W dodatku dostęp do niej jest bardzo kłopotliwy, gdyż musi się odbywać wyłącznie poprzez funkcje `SetClassLong[Ptr]()` i `SetWindowLong[Ptr]()`.

Powody te wyjaśniają, dlaczego niemal zawsze w obu polach ustawia się wartość 0.

### *Rejestracja klasy okna*

Po wypełnieniu struktury `WNDCLASSEX` możemy zarejestrować klasę okna, używając do tego funkcji `RegisterClassEx()`:

```
ATOM RegisterClassEx(CONST WNDCLASSEX* lpwctx);
```

Jak doskonale wiemy, przyjmuje ona jeden parametr - wskaźnik na naszą strukturę. W zamian zwraca zaś tzw. **atom**, który jest 16-bitową liczbą identyfikującą zarejestrowaną klasę. Atom ten możemy zachować w osobnej zmiennej i używać w wywołaniach tych funkcji, które żądają nazwy klasy okna (jak np. `CreateWindowEx()`). W zasadzie jednak nie jest to zbyt powszechna praktyka.

Rezultatem zwracanym przez `RegisterClassEx()` można się aczkolwiek zainteresować także z innego powodu. Stosuje się bowiem do niego tradycyjna konwencja Windows API, na mocy której zero jest wynikiem świadczącym o błędzie. Do celów diagnostycznych możemy więc używać instrukcji `if` podobnej do tej:

```
if (!RegisterClassEx(&KlasaOkna))
{
    MessageBox (NULL, "Błąd podczas rejestracji klasy okna!", "Błąd",
                MB_OK | MB_ICONERROR);
    return 1;
}
```

Wystąpienie błędu przy rejestracji okna wskazuje na niepoprawność jednej ze składowych `WNDCLASSEX` - najczęściej chodzi tu zapewne o któryś z uchwytów.

## Tworzenie okna właściwego

Gdy zarejestrujemy klasę okna, mamy już za sobą pierwszy krok jego kreacji. Teraz możemy już wywołać `CreateWindowEx()` i podać jej ten tuzin wymaganych parametrów, w zamian cieszyć się stworzonym oknem i otrzymać uchwyt do niego.

By tego dokonać trzeba oczywiście wiedzieć, jakie informacje podać funkcji w tych kilkunastu parametrach. Spójrzmy więc na nie.

W tym momencie przydałoby się, abyś przypomniał sobie prototyp funkcji `CreateWindowEx()`.

### Nazwa klasy okna i uchwyt instancji programu

Ponownie jedną z najważniejszych danych jest **nazwa klasy okna**; tym razem potrzebuje jej system, by powiązać tworzone okna z zarejestrowaną wcześniej klasą. Nazwę tej klasy podajemy w drugim parametrze, `lpClassName`. Warto już teraz wiedzieć (będzie nam to potrzebne przy okazji kontrolek), że nazwa ta nie musi odnosić się do klasy zdefiniowanej przez nas samych, lecz także do jednej z wbudowanych w system globalnych klas okien (kontrolek).

Alternatywnie możemy też podać w `lpClassName` atom, który zwróciła w wyniku funkcja `RegisterClassEx()`.

W parze z klasą okną idzie też **uchwyt do instancji programu**, który należy podać jako jedenasty (przedostatni) parametr `CreateWindowEx()`. Należy go podać, ponieważ w Windows klasy okna rejestrowane przez programistę są związane właśnie z instancjami programów; dzięki temu system unika kłopotów ze zdublowanymi nazwami tych klas.

### Tytuł okna

Trzeci parametr, `lpWindowName`, nazywany **tytułem okna**, jest tak naprawdę tylko tekstem, który będzie w tym oknie wyświetlony... albo i nie. W przypadku omawianych przez nas okien pojawi się on na pasku tytułu - pod warunkiem, że okno będzie takowy pasek posiadać (bo wcale nie musi!).

Jeżeli zaś chodzi na przykład o kontrolki, to interpretacja napisu podanego w `lpWindowName` zależy ściśle od ich rodzaju.

Tytuł (tekst) istniejącego okna można zmieniać za pomocą funkcji [SetWindowText\(\)](#).

## Styl okna

Czwarty z kolei parametr - `dwStyle` - to **styl okna**. Ma on bodaj największy wpływ na zewnętrzną aparycję okna, a częściowo także i na jego zachowanie. Wartość ta jest przy tym kombinacją flag bitowych, a zatem umożliwia ustawienie wielu różnorodnych aspektów okna. Zostały one zebrane w poniższej tabeli:

<b>grupa</b>	<b>flagi</b>	<b>nazwa stylu</b>	<b>uwagi</b>
<b>rodzaj okna</b>	WS_CHILD WS_CHILDWINDOW	okno potomne	Jedyną cechą wyróżniającą okno potomne od zwykłego jest brak możliwości posiadania przez nie menu.
	WS_OVERLAPPED WS_TILED	okno trwałe	Trwałe okno to takie, które może być wyświetlane jako niemodalne.
	WS_POPUP	okno „wyskakujące”	Takie okno może być wyświetlane tylko jako modalne.
<b>krawędź okna</b>	WS_BORDER	stałe obramowanie	Krawędź okna z takim stylem jest narysowana cienką linią i nie reaguje na przeciąganie (nie można więc zmienić rozmiaru okna, ciągnąc za jego brzeg).
	WS_THICKFRAME WS_SIZEBOX	zmiennie obramowanie	Zmienne obramowanie okna jest narysowane grubą linią i pozwala na zmianę rozmiaru okna poprzez przeciąganie krawędzi.
	WS_DLGRAME	obramowanie okna dialogowego	Ustala obramowanie typowe dla okien dialogowych; okno z tym stylem nie może mieć paska tytułu (stylu WS_CAPTION).
<b>pasek tytułu</b>	WS_CAPTION	pasek tytułu	Okno z tym stylem posiada widoczny pasek tytułu.
	WS_SYSMENU	menu sterujące	Styl ten powoduje obecność ikonki w lewym górnym rogu okna. Kliknięcie na tą ikonkę powoduje pokazanie menu sterującego. By styl ten zadziałał, należy podać także flagę WS_CAPTION.
	WS_MAXIMIZEBOX	przycisk maksymalizacji /przywracania	Tworzy okno z obecnym przyciskiem maksymalizacji. Flaga WS_SYSMENU jest również wymagana.
	WS_MINIMIZEBOX	przycisk minimalizacji	Tworzy okno z obecnym przyciskiem minimalizacji. Flaga WS_SYSMENU jest również wymagana.
<b>początkowy stan okna</b>	WS_VISIBLE	widoczne okno	Okno z tym stylem jest widoczne na ekranie od razu po utworzeniu (nie trzeba stosować dlań funkcji ShowWindow()).

<b>grupa</b>	<b>flagi</b>	<b>nazwa stylu</b>	<b>uwagi</b>
	WS_MAXIMIZE	okno zmaksymalizowane	Po utworzeniu okno jest widoczne i zmaksymalizowane.
	WS_ICONIC WS_MINIMIZE	okno zminimalizowane	Okno z tym stylem jest z początku zredukowane do przycisku na pasku zadań.
	WS_DISABLED	nieaktywne okno	Styl ten dezaktywuje okno - tak, że nie reaguje ono na działania użytkownika. Okno można uaktywnić z powrotem za pomocą funkcji <a href="#">EnableWindow()</a> .  Styl ten stosuje się raczej do kontrolerek, rzadko do „normalnych” okien.
<b>paski przewijania</b>	WS_HSCROLL	poziomy pasek przewijania	Tworzy okno z poziomym paskiem przewijania.
	WS_VSCROLL	pionowy pasek przewijania	Tworzy okno z pionowym paskiem przewijania.

Tabela 30. Flagi bitowe stylu okna

Jak widać na załączonym obrazku, możliwych flag jest naprawdę mnóstwo. Dają one prawie nieograniczone możliwości w budowaniu wizerunku i funkcjonowania okna, gdyż uwzględniają wszystkie jego cechy. Bez stylu okno jest bowiem tylko „gołym” obszarem klienta, pozbawionym nawet obramowania, nie mówiąc już choćby o pasku tytułu.

Potęgą ta ma jednak swoją wadę: stworzenie przy jej pomocy sensownego okna wymaga podania przynajmniej kilku różnych stylów, co prowadzi do rozbudowanych alternatyw bitowych. Poza tym trzeba jeszcze pamiętać właściwe nazwy poszczególnych flag. Na szczęście problemy te zostały rozwiązane przez samych twórców Windows API. Wprowadzili oni mianowicie dodatkowe makra, kombinujące po kilka stylów, i nadali im krótsze nazwy. Makr tych możemy więc używać pojedynczo lub w połączeniu z innymi jeszcze stylami - zupełnie tak, jakby same były flagami bitowymi. Listę owych makr (nie jest ich zbyt dużo) przedstawia nam ta oto tabelka:

<b>makra</b>	<b>złączone flagi</b>	<b>uwagi</b>
WS_POPUPWINDOW	WS_BORDER WS_POPUP WS_SYSMENU	Wraz z flagą WS_CAPTION makro to stanowi odpowiedni styl dla okien modalnych.
WS_OVERLAPPEDWINDOW WS_TILEDWINDOW	WS_OVERLAPPED WS_CAPTION WS_SYSMENU WS_THICKFRAME WS_MINIMIZEBOX WS_MAXIMIZEBOX	Wariant właściwych dla okien głównych aplikacji, które mogą dopuszczać zmianę swego rozmiaru.

Tabela 31. Predefiniowane makra stylu okna

Są one zazwyczaj bardzo wygodne, ponieważ nawet jeśli nie odpowiada nam któraś z flag, jakie zawierają, możemy ją wyłączyć np. w ten sposób:

```
WS_OVERLAPPEDWINDOW & ~WS_MAXIMIZEBOX // wyłącza przycisk maksymalizacji
```

Oczywiście nic nie stoi też na przeszkodzie, byśmy sami definiowali sobie przydatne nam makra, jak choćby takie:

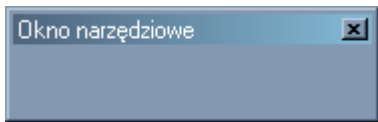
```
// trwałe okno o stałym rozmiarze
#define WS_FIXEDWINDOW (WS_OVERLAPPED | WS_CAPTION | WS_BORDER
                      | WS_SYSMENU | WS_MINIMIZEBOX)

// trwałe okno na pełnym ekranie
#define WS_FULLSCREENWINDOW (WS_OVERLAPPED | WS_MAXIMIZE)
```

Wszystko zależy bowiem od konkretnych potrzeb w pisany programie.

## Rozszerzony styl okna

Wraz z rozwojem systemu Windows pojawiły się nowe możliwości dostosowywania okien i kontroli ich zachowania. Te nowe opcje, nazywane **rozszerzonym stylem okna** (ang. *extended window style*), nie są niezbędne w każdej sytuacji. Spora część z nich ma zresztą dość specyficzne zastosowanie; niemniej jednak istnieje kilka flag, które mogą być niekiedy przydatne. Oto one:

<i>flaga</i>	<i>nazwa stylu</i>	<i>opis</i>
WS_EX_CONTEXTHELP	przycisk pomocy kontekstowej	Flaga ta powoduje pojawienie się na pasku tytułu przycisku ze znakiem zapytania. Kliknięcie w niego uaktywnia tryb pomocy kontekstowej; kiedy teraz użytkownik kliknie gdzieś we wnętrzu okna, otrzyma ono specjalny komunikat WM_HELP. W reakcji na niego powinno zostać pokazane „wyskakujące” okno z objaśnieniem. Styl ten nie może występować razem z WS_MINIMIZEBOX lub WS_MAXIMIZEBOX.
WS_EX_TOOLWINDOW	okno narzędziowe	Tworzy okno pływającego paska narzędzi. Takie okno ma węższy pasek tytułu, a sam tytuł jest w nim pisany mniejszą czcionką. Poza tym takie okno nie jest nigdy pokazywane na pasku zadań. <div style="text-align: center;">  </div> <p><b>Screen 58. Okno z włączonym stylem WS_EX_TOOLWINDOW</b></p>
WS_EX_TOPMOST	zawsze na wierzchu (ang. <i>stay on top</i> )	Okno oznaczone tą flagą jest wyświetlane przed wszystkimi innymi oknami, nawet wtedy, gdy jest ono nieaktywne. <div style="background-color: yellow; padding: 5px; margin-top: 10px;"> <p>Ten styl może być dodawany i usuwany również poprzez funkcję <a href="#">SetWindowPos()</a>.</p> </div>

**Tabela 32. Flagi bitowe rozszerzonego stylu okna**

Kombinację tych flag (którą jest najczęściej po prostu NULL) podajemy w pierwszym parametrze funkcji `CreateWindowEx()`, `dwExStyle`.

Obecność tego parametru jest zresztą jedyną różnicą między `CreateWindowEx()` a zwykłą `CreateWindow()`.

## Pozycja na ekranie i wymiary okna

Cztery parametry funkcji `CreateWindowEx()`, począwszy od piątego aż do ósmego, określają rejon na ekranie monitora, który będzie (początkowo) zajmowało tworzone

okno. Definiuje go pozycja lewego górnego rogu okna we współrzędnych ekranu, podawanych w parametrach  $x$  i  $y$ , oraz szerokość i wysokość okna - parametry `nWidth` i `nHeight`.

### Ważka kwestia rozdzielczości

Wszystkie te wielkości podajemy w pikselach. Sprawia to, że wizualny efekt, jaki wywiera okno, jest w dużym stopniu zależny od stosowanej u użytkownika rozdzielczości ekranu. Teoretycznie możemy ją programowo zmieniać, ale zachowanie takie jest wielce niepożądane w środowisku, gdzie naraz działa wiele aplikacji - jak w Windows.

W zasadzie jednak nie jest to aż takim problemem, jako że wymiary okna powinny być w większości przypadków stałe i niezależne od rozdzielczości. Pozostaje jedynie sprawa umiejscowienia okna na ekranie - najczęściej chcemy bowiem, aby sytuowało się ono dokładnie pośrodku pulpitu.

Aby to osiągnąć, musimy znać jego wymiary i posłużyć się odpowiednimi wzorami:

$$x = (w_{ekranu} - w_{okna}) / 2$$

$$y = (h_{ekranu} - h_{okna}) / 2$$

w których literki  $w$  i  $h$  oznaczają odpowiednio szerokość i wysokość ekranu lub okna, zaś  $x$  i  $y$  - docelowe współrzędne okna na ekranie.

I tu mamy kolejny problem, choć na szczęście jest on łatwo rozwiązywalny. Musimy przecież pobrać wymiary ekranu, czyli jego rozdzielczość, by móc wstawić je do powyższych formuł. Do tego celu należy posłużyć się funkcją `GetSystemMetrics()`:

```
int GetSystemMetrics(int nIndex);
```

Funkcja ta może zwracać przeróżne systemowe wartości, związane głównie z wyświetlaniem. Rodzaj poszukiwanych danych podajemy w jej jedynym parametrze; nas interesują najbardziej wielkości `SM_CXSCREEN` oraz `SM_CYSCREEN`, oznaczające szerokość i wysokość ekranu w pikselach. Jest to więc dokładnie to, o co nam teraz chodzi :)

Stworzenie okna umiejscowionego na środku ekranu może zatem wyglądać w ten sposób:

```
// zakładamy, że w uWidth i uHeight mamy wymiary okna

HWND hOkno;
hOkno = CreateWindowEx( // (pomijamy nieistotne teraz parametry)
    (GetSystemMetrics(SM_CXSCREEN) - uWidth) / 2,
    (GetSystemMetrics(SM_CYSCREEN) - uHeight) / 2,
    uWidth,
    uHeight,
    // (j.w)
);
```

Znając rozdzielczość ekranu możemy też uzależnić od niej szerokość i wysokość okna. Pamiętajmy jednakże, iż może to zburzyć nam ład ewentualnych kontrolek potomnych, umiejscowionych na oknie.

### Obszar klienta ma zawsze rację

W praktyce (szczególnie programisty gier!) nierzadko objawia się jeszcze jeden problem związany z umiejscowieniem okna. Czasem bowiem posiadamy wymiary nie samego okna, lecz tylko jego **obszaru klienta**. Dzieje się tak na przykład wtedy, gdy chcemy pełnoekranowej grze, działającej w jakiejś stałej rozdzielczości, zapewnić możliwość

uruchamiania się w także trybie okienkowym. Wówczas wymiary obszaru klienta okna gry muszą zgadzać się z tą ustaloną wcześniej „rozdzielczością pełnoekranową”. Jednak samo okno to przecież nie tylko obszar klienta: trzeba do niego dodać przynajmniej jakiś brzeg oraz pasek tytułu. Prawidłowe wymiary okna, które powinniśmy podać do `CreateWindowEx()`, będą więc większe; również pozycja okna musi zostać odpowiednio zmieniona. Jak podołać tym zadaniom?...

Otóż system Windows oferuje pewne rozwiązanie, którym jest funkcja `AdjustWindowRectEx()`:

```
BOOL AdjustWindowRectEx(LPRECT lpRect,
                        DWORD dwStyle,
                        BOOL bMenu,
                        DWORD dwExStyle);
```

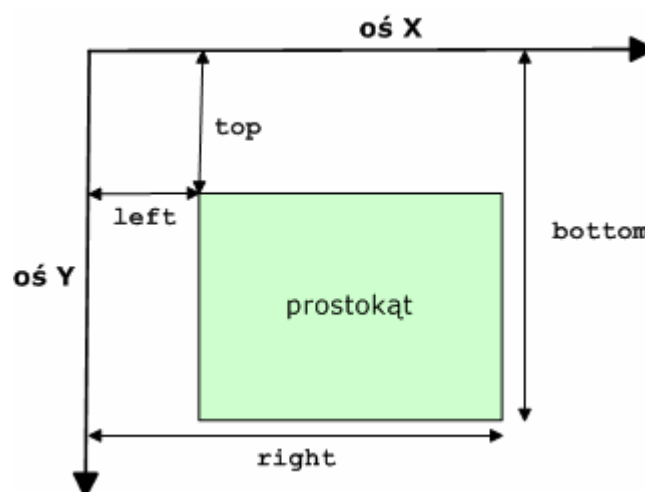
Wymaga ona podania kilku potrzebnych informacji, wśród których są:

- styl okna, który należy umieścić w parametrze `dwStyle` (jest to ten sam styl, który zamierzamy za chwilę podać do `CreateWindowEx()`. Należy więc zapisać go w jakiejś osobnej zmiennej i wykorzystać ją także tutaj)
- wartość boolowska (`true` lub `false`) określająca, czy okno posiada pasek menu
- rozszerzony styl okna (który także podamy zaraz dla `CreateWindowEx()`)

W zasadzie jednak najważniejszy jest pierwszy parametr, w którym określamy **prostokąt obszaru klienta** okna. Jest to wskaźnik do specjalnej struktury `RECT`:

```
struct RECT
{
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
};
```

Wyznacza ona prostokąt na ekranie w nieco inny sposób niż omawiane przez cały czas cztery parametry funkcji `CreateWindowEx()`. O ile bowiem pola `left` oraz `top` odpowiadają wartościom  $x$  i  $y$  - współrzędnych lewego górnego rogu prostokąta, o tyle `right` i `bottom` są koordynatami prawego dolnego wierzchołka. Aby obliczyć te składowe, musimy więc dokonać odpowiedniego sumowania:  $x + nWidth$  oraz  $y + nHeight$ .



Rysunek 8. Opis prostokąta na ekranie za pomocą pól struktury `RECT`



Ostatecznie skorzystanie z funkcji `AdjustWindowRectEx()` powinno się przedstawiać mniej więcej tak:

```
// zakładamy, że w zmiennych x, y, nClientWidth i nClientHeight mamy
// zapisaną pozycję (np. środek ekranu) oraz wymiary
// obszaru klienta okna

// wypełniamy strukturę RECT
RECT rcOkno;
rcOkno.left = x;
rcOkno.top = y;
rcOkno.right = x + nClientWidth;
rcOkno.bottom = y + nClientHeight;

// ustalamy styl i rozszerzony styl okna
DWORD dwStyle = WS_OVERLAPPEDWINDOW;
DWORD dwExStyle = NULL;

// wywołujemy funkcję AdjustWindowRectEx()
AdjustWindowRectEx (&rcOkno, dwStyle, false, dwExStyle);
```

No dobrze, ale co ta funkcja właściwie robi?... Cóż, to dobre pytanie :)

`AdjustWindowRectEx()` modyfikuje po prostu strukturę podaną jej w pierwszym parametrze w taki sposób, że po wywołaniu opisuje ona już nie obszar klienta okna, ale **całe okno** - wraz z jego obszarem pozaklienckim. Podczas dokonywania tych modyfikacji funkcja korzysta oczywiście z podanych jej stylów oraz obecności lub nieobecności paska menu.

Zatrzymajmy się na chwilę, gdyż w tym momencie w zmiennej `rcOkno` mamy zapisaną „charakterystykę przestrzenną” (tak to nazwijmy...) całego naszego okna. Możemy zatem wywołać już funkcję `CreateWindowEx()` i utworzyć je:

```
// tworzymy okno
HWND hOkno;
hOkno = CreateWindowEx(dwExStyle,
                      "Klasa_okna",
                      "Okno",
                      dwStyle,
                      rcOkno.left,
                      rcOkno.top,
                      rcOkno.right - rcOkno.left,
                      rcOkno.bottom - rcOkno.top,
                      NULL,
                      NULL,
                      hInstance,
                      NULL);
```

Podczas podawania parametrów dokonujemy też niezbędnego przeliczenia ze składowych `RECT` na współrzędne lewego górnego wierzchołka oraz szerokość i wysokość okna.

Jako podsumowanie ukazanej techniki możesz przeczytać opis funkcji `AdjustWindowRectEx()` w [MSDN](#).

## *Uchwyt do okna nadrzędnego*

Dziewiąty parametr `CreateWindowEx()` jest **uchwytem do okna nadrzędnego** względem tego okna, które właśnie zamierzamy stworzyć. Podając tutaj właściwy uchwyt, umożliwiamy systemowi budowę poprawnej hierarchii okien.

Gdy stworzymy okno nadrzędne (ang. *top-level*), czyli główne okno aplikacji, wówczas jego bezpośrednim i jedynym przodkiem jest tylko i wyłącznie pulpit. Możemy więc podać w parametrze `hWndParent` wartość uzyskaną z funkcji `GetDesktopWindow()` - jest to bowiem uchwyt do pulpitu właśnie. Nie ma jednak takiej potrzeby: `CreateWindowEx()` dopuszcza podanie w tym parametrze uchwytu pustego, czyli `NULL`; efekt będzie ten sam, a my nie musimy się zbyt wiele napisać :D

W Windows 2000 oraz XP jako `hWndParent` możemy podać także specjalną stałą `HWND_MESSAGE`. Spowoduje to utworzenie tzw. okna obsługi komunikatów (ang. *message-only window*), którego jedynym celem jest odbieranie i wysyłanie komunikatów. Takie okno nie jest widoczne na ekranie, ale umożliwia aplikacji (zwykle usłudze systemowej) normalną interakcję z systemem operacyjnym.

Poprawne dobranie okna nadrzędnego jest szczególnie ważne przy tworzeniu kontroltek, czym się zajmiemy w jednym z przyszłych rozdziałów.

### *Uchwyt do paska menu*

Parametr `hMenu` funkcji `CreateWindowEx()` reprezentuje **pasek menu**, jaki będzie posiadało kreowane okno. Przywilej posiadania takiego paska mają jedynie okna trwałe i wyskakujące, ponadto muszą one mieć także pasek tytułu.

Jak pewnie doskonale wiemy, menu jest zestawem opcji ułożonych w grupy, które dają dostęp do wszystkich funkcji programu. Jest to więc twór dość skomplikowany, zajęcie się którym wymaga nieco więcej wysiłku niż tylko wywołania paru funkcji. Przekonamy się o tym w dalszej części kursu.

### *Dodatkowy parametr*

Ostatni argument `CreateWindowEx()`, czyli `lpParam`, jest obecny tylko dla wygody programisty. Możemy w nim przekazać wskaźnik na **dowolne dane**, które okno ma otrzymać zaraz po swoich narodzinach. Dostanie go wraz z komunikatem `WM_CREATE`, wysyłanym jeszcze przed powrotem z funkcji `CreateWindowEx()`. Więcej informacji o tym parametrze możesz znaleźć przy opisie komunikatu `WM_CREATE` w niniejszym rozdziale.

\*\*\*

W ten sposób dotarliśmy do epilogu procesu tworzenia okna. Teraz powinieneś już na ten temat prawie wszystko... albo przynajmniej pamiętać, co gdzie zostało tutaj opisane ;D

W następnym podrozdziale zajmiemy się operacjami, jakie można przeprowadzać na utworzonym już oknie.

## **Okna pod kontrolą**

Windows API oferuje całe mnóstwo narzędzi przeznaczonych do pracy z oknami. Utworzenie własnego okna to bowiem tylko początek jego misji - dalej na właściwości stworzonego obiektu ma duży wpływ zarówno programista, jak i użytkownik.

### *Działania na oknach*

Z punktu widzenia programisty okna są tworamiami niemal całkowicie elastycznymi. Zmienić możemy każdy ich aspekt, posługując się do tego odpowiednimi funkcjami. Tym właśnie częściom WinAPI przyjrzymy się tutaj.

## Pokazywanie i ukrywanie

Typowym działaniem na oknie jest jego pokazanie (uczynienie go widocznym) i ukrycie lub też zamknięcie. Sprawdźmy, jak możemy wykonać te działania.

### Ukazanie i ukrycie okna

Dość nieoczekiwanie, zarówno do pokazywania, jak i do ukrywania okna, służy ta sama funkcja `ShowWindow()`:

```
BOOL ShowWindow(HWND hWnd, int nCmdShow);
```

Na pewno nie jest nam ona całkiem obca, jako że dość dokładnie opisaliśmy ją w poprzednim rozdziale - szczególnie stała, jakie może przyjąć w drugim parametrze. Teraz więc przypomnimy tylko, że:

- pokazanie okna z zachowaniem jego rozmiarów i pozycji wymaga wykorzystania stałej `SW_SHOW`
- maksymalizacja okna wymaga skorzystania z opcji `SW_MAXIMIZE`
- `SW_MINIMIZE` minimalizuje okno
- `SW_RESTORE` przywraca okno zarówno ze stanu maksy-, jak i minimalizacji
- `SW_HIDE` ukrywa okno, i to tak skutecznie, że nie będzie nigdzie widoczne - nawet na pasku zadań

Ostrożnie należy postępować ze stałą `SW_HIDE`, zwłaszcza jeśli ukrycie okna ma być spowodowane jakąś akcją użytkownika. Ukryte okno jest bowiem dla niego całkowicie niedostępne, zatem aby mógł on zobaczyć je ponownie, program musi oferować mechanizm przywołania schowanego okna znajdujący się w całkiem innym oknie.

Warto byłoby przyjrzeć się opisowi funkcji `ShowWindow()` w [MSDN](#).

### Sprawdzanie widoczności okna

Używając funkcji `ShowWindow()` zmieniamy **stan widoczności** (ang. *visibility state* albo po prostu *window state*) okna. Pobranie tego stanu jest również możliwe; nie służy jednak do tego pojedyncza funkcja (która mogłaby zwrócić jakąś stałą wyliczeniową), lecz w sumie aż trzy odrębne wywołania:

```
BOOL IsWindowVisible(HWND hWnd);
BOOL IsIconic(HWND hWnd);
BOOL IsZoomed(HWND hWnd);
```

Ich postać jest identyczna: każdej funkcji podajemy uchwyt do sprawdzanego okna, a w zamian otrzymujemy wartość boolowską, wskazującą na obecność lub nieobecność danego stanu okna. I tak:

- `IsWindowVisible()` informuje nas o tym, czy okno jest widoczne, czy nie
- `IsIconic()` mówi, czy okno jest zminimalizowane
- `IsZoomed()` powiadamia o ewentualnej maksymalizacji okna

Taki sposób pobierania stanu okna może nas aczkolwiek nie zadowalać. Nikt nam wówczas nie zabroni napisania sobie własnej funkcji, spełniającej to zadanie:

```
enum WINDOWSTATE { WS_HIDDEN, WS_NORMAL, WS_MAXIMIZED, WS_MINIMIZED };

WINDOWSTATE GetWindowState(HWND hWnd)
{
    // najpierw sprawdzamy, czy okno nie jest ukryte
    if (!IsWindowVisible(hWnd)) return WS_HIDDEN;
```

```

// dalej zajmujemy się maksymalizacją i minimalizacją
if (IsIconic(hWnd)) return WS_MINIMIZED;
else if (IsZoomed(hWnd)) return WS_MAXIMIZED;
else return WS_NORMAL;
}

```

Taka funkcja ma przynajmniej jedną drobną zaletę: lepiej sprawdza się w blokach `switch` niż trójka procedur Windows API.

## Pozycja i rozmiar

Z wyglądem okna na ekranie wiąże się nie tylko fakt jego widoczności lub niewidoczności, lecz także jego **pozycja** oraz **rozmiar**. Wszystkie te wielkości możemy oczywiście zmieniać przy pomocy odpowiednich funkcji WinAPI.

Zanim je poznamy, należy jeszcze wspomnieć o układzie odniesienia, jaki stosuje Windows podczas pozycjonowania okien. Otóż początek układu współrzędnych, wedle którego następuje ustawienie położenia okna, znajduje się zawsze w **lewym górnym rogu okna nadrzędnego**. Fakt ten nie ma specjalnego znaczenia dla głównych okien aplikacji, dla których nadrzędny jest tylko pulpit, szczelnie zakrywający cały ekran; kwestia ta nabierze jednak wymowy, gdy zaczniemy zajmować się oknami potomnymi (kontrolkami).

Po wyjaśnieniu tej drobnej sprawy przejdziemy już do sposobów pobierania i ustawiania położenia oraz wielkości okna.

## Zmiana pozycji i rozmiaru

Gdy chcemy jednocześnie zmienić zarówno umiejscowienie okna na ekranie, jak i jego szerokość i wysokość, wtedy najrozsądniejszym wyborem jest funkcja o dosyć mylącej nazwie `MoveWindow()`:

```

BOOL MoveWindow(HWND hWnd,          // uchwyt modyfikowanego okna
                int X,              // nowa współrzędna pozioma
                int Y,              // nowa współrzędna pionowa
                int nWidth,         // nowa szerokość okna
                int nHeight,        // nowa wysokość okna
                BOOL bRepaint);     // czy odrysowywać zawartość okna?

```

Podajemy w niej zarówno nowe współrzędne okna (o uchwycie wpisanym w pierwszym parametrze), jak i jego nowe wymiary - szerokość i wysokość. Cztery te wielkości zostaną więc bezwarunkowo zmienione.

Piąty parametr `bRepaint` określa, czy po dokonaniu operacji na oknie ma ono zostać odrysowane, a więc otrzymać komunikat `WM_PAINT`. Prawie zawsze chcemy, by tak się właśnie stało, bo wtedy na pewno wszystkie okna na ekranie będą wyglądały poprawnie. W parametrze `bRepaint` podajemy więc wartość `TRUE`.

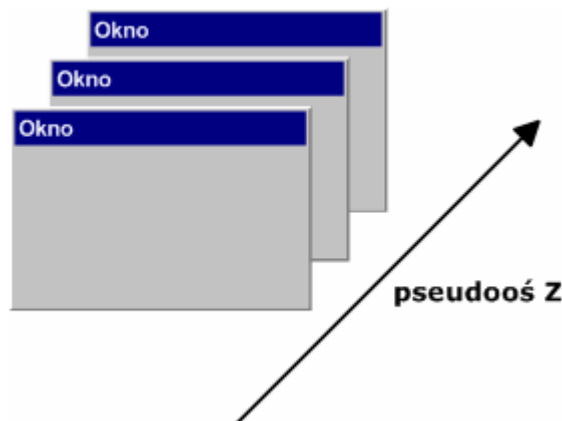
Nieco inaczej postępujemy, kiedy zależy nam tylko na położeniu okna albo tylko na zmianie jego rozmiaru. Wówczas powinniśmy bowiem skorzystać z bardziej elastycznej funkcji `SetWindowPos()`:

```

BOOL SetWindowPos(HWND hWnd,        // modyfikowane okna
                  HWND hWndInsertAfter, // przykrywające okno
                  int X,            // nowa współl. pozioma
                  int Y,            // nowa współl. pionowa
                  int cx,           // nowa szerokość
                  int cy,           // nowa wysokość
                  UINT uFlags);     // flagi

```

Właściwie to potrafi ona nie tylko przesuwać i skalować okno, ale także kontrolować przykrywanie go przez inne okna. Mowa tu o tzw. **kolejności przesłaniania** lub **porządku Z** (ang. *z-order*). Mechanizm ten odpowiada za warstwowe ułożenie okien w interfejsie użytkownika, w którym jedno okno może całkowicie lub częściowo zasłaniać inne. Nazwa 'porządek Z' bierze się stąd, iż zjawisko to sugeruje istnienie trzeciej, wirtualnej pseudoosi współrzędnych Z:



Rysunek 9. Porządek Z okien w systemie Windows

Oczywiście ta oś tak naprawdę nie istnieje, gdyż nie możemy ustawić okna na określonej współrzędnej Z. Możliwe jest jednak umiejscowienie go na niej w pozycji relatywnej do innego okna. W parametrze `hWndInsertAfter` funkcji `SetWindowPos()` możemy mianowicie podać uchwyt okna, które zostanie umieszczone **bezpośrednio przed** tym modyfikowanym (podanym w `hWnd`) w porządku Z (będzie je przesłaniać). Oprócz tego dopuszczalne jest również podanie w tym parametrze jednej z kilku specjalnych wartości o następującym znaczeniu:

<i>stała</i>	<i>opis</i>
<code>HWND_BOTTOM</code>	Umieszcza okno na spodzie kolejności przesłaniania. Modyfikowane okno będzie więc przykrywane przez wszystkie inne widoczne okna.
<code>HWND_TOP</code>	Okno zostaje umiejscowione na szczycie porządku Z, przykrywając wszystkie pozostałe okno.
<code>HWND_TOPMOST</code>	Umieszcza okno na szczycie kolejności przesłaniania, a ponadto czyni je oknem typu „zawsze na wierzchu” (ang. <i>stay on top</i> lub <i>topmost</i> ). Takie okno nie jest przesłanianie przez żadne inne okna <sup>111</sup> .
<code>HWND_NOTOPMOST</code>	Okno jest przesuwane za wszystkie okna typu „zawsze na wierzchu”, lecz przed wszystkimi innymi oknami. Jeżeli przemieszczane okno samo było oknem „zawsze na wierzchu”, traci ono tę cechę.

Tabela 33. Stałe dla parametru `hWndInsertAfter` funkcji `SetWindowPos()`

Pozostałe cztery parametry funkcji `SetWindowPos()` są analogiczne dla argumentów `MoveWindow()` i specyfikują odpowiednio: nowe współrzędne (X i Y) przesuwanego okna oraz jego nowe wymiary (`cx` i `cy`). Na czym więc polega różnica między tą funkcją a poprzednią?

Jest ona znacząca: `SetWindowPos()` pozwala na zdecydowanie, które z podanych jej parametrów (X i Y, `cx` i `cy` oraz `hWndInsertAfter`) mają być faktycznie brane pod uwagę i zmieniane dla okna o uchwycie `hWnd`. W przeciwieństwie do `MoveWindow()` nie jesteśmy zmuszeni do zmiany zarówno pozycji, jak i rozmiaru okna - przeciwnie, możemy samodzielnie zdecydować o poczynionych modyfikacjach.

<sup>111</sup> A dokładniej mówiąc: przez żadne okno, które samo nie jest typu „zawsze na wierzchu”.

Dokonujemy tego, podając odpowiednie flagi bitowe w ostatnim parametrze - `uFlags`. Wypadkowa kombinacja może się tu składać z poniższych wartości:

<i>flaga</i>	<i>opis</i>
<code>SWP_NOMOVE</code>	zapobiega przesuwaniu okna (ignoruje parametry <code>x</code> i <code>y</code> )
<code>SWP_NOSIZE</code>	nie pozwala na zmianę rozmiarów okna (ignoruje parametry <code>cx</code> i <code>cy</code> )
<code>SWP_NOZORDER</code>	pozostawia aktualną pozycję okna w porządku Z (ignoruje parametr <code>hWndInsertAfter</code> )

**Tabela 34. Flagi bitowe funkcji `SetWindowPos()`**

Tak więc ażeby dokonać jedynie przesunięcia okna po ekranie należy wywołać `SetWindowPos()` w sposób podobny tego:

```
SetWindowPos (hOkno, NULL, nX, nY, 0, 0, SWP_NOSIZE | SWP_NOZORDER);
```

Z kolei instrukcja w rodzaju takiej:

```
SetWindowPos (hOkno, HWND_TOP, 0, 0, 0, 0, SWP_NOSIZE | SWP_NOMOVE);
```

spowoduje „wyciągnięcie” okna przed wszystkie inne w kolejności przesłaniania, nie powodując jednak ani jego przemieszczania się, ani skalowania.

Teraz nie od rzeczy byłoby zapoznanie się z opisem funkcji [MoveWindow\(\)](#) i [SetWindowPos\(\)](#) w MSDN.

Istnieje również funkcja `SetWindowPlacement()`, która pozwala na zmianę nie tylko aktualnej pozycji okna, ale też tej, którą przyjmuje ono po maksymalizacji czy minimalizacji. Ponadto łączy ona w sobie także możliwość zmiany stanu widoczności okna, niczym `ShowWindow()`. W sumie jest to więc dość ciekawa funkcja, której opis możesz znaleźć w [MSDN](#).

### *Pobieranie umiejscowienia i wielkości okna*

Koordinaty przestrzenne okna możemy w Windows nie tylko, rzecz jasna, ustawiać, ale także pobierać je. Czynimy to przy pomocy funkcji `GetWindowRect()`:

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

Zwraca nam ona określenie prostokąta mieszczącego okno; znajdziemy je w strukturze typu `RECT`, do której wskaźnik musimy podać w drugim parametrze funkcji.

**Pamiętajmy, że współrzędne tego prostokąta są liczone względem okna nadrzędnego.**

Jak sądzę pamiętamy doskonale (było to przecież całkiem niedawno :D), że pola tej struktury nazwane `left` i `top` są współrzędnymi lewego górnego wierzchołka prostokąta, zaś `right` i `bottom` - prawego dolnego. Uzyskanie z tych danych szerokości i wysokości okna wymaga zatem tylko dwóch prostych działań:

```
RECT rcOkno;
GetWindowRect (hOkno, &rcOkno);

unsigned uWidth = rcOkno.right - rcOkno.left;
unsigned uHeight = rcOkno.bottom - rcOkno.top;
```

Co ciekawe (i trochę dziwne), Windows API nie udostępnia żadnej funkcji, która umożliwiłaby bardziej wybiórcze pobieranie danych o pozycji i rozmiarze okna. Zatem albo dostaniemy wszystko, albo nic :)

## Pasek tytułu

Zdecydowana i miażdżąca większość okien posiada swój pasek tytułu. Na nim zaś widoczny jest tytuł okna, opisujący zasadniczo jego zawartość. Jak każdy element okna, także i on może być zmieniany przez programistę.

## Ustawienie tytułu okna

Nie ma chyba nic prostszego od ustawiania tytułu okna. Przeznaczona do tego funkcja `SetWindowText()` ma chyba najprostszą możliwą i jednocześnie najbardziej intuicyjną postać:

```
BOOL SetWindowText(HWND hWnd, LPCTSTR lpString);
```

Wywnioskowanie sposobu użycia tej funkcji z jej prototypu jest, jak sądzę, oczywistym zadaniem. Spójrzmy aczkolwiek na odpowiedni kod:

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

// ...

std::stringstream Strumien;
Strumien << rand();
SetWindowText (hOkno, Strumien.str().c_str());
```

Pokazuje on, jak można ustawić losową liczbę jako tytuł okna. Przykład ten nie należy być może do wielce przydatnych, niemniej dobrze ilustrują funkcję `SetWindowText()`. Nie wymaga ona chyba więcej komentarza, prawda? :)

## Odczytywanie tytułu okna

Czynność odwrotna - pobieranie tytułu okna - może nastroczać pewnych problemów. Nie wynikają one jednak z toporności samego Windows API, lecz ich podłożem są łańcuchy znaków w stylu C. Napis na pasku tytułu musi być bowiem pobrany w tej właśnie postaci. Najprostszą (i wcale nie najlepszą) drogą jest użycie kodu zbliżonego do poniższego:

```
char szTytulOkna[256];
GetWindowText (hOkno, szTytulOkna, 256);
```

Zastosowano w nim funkcję `GetWindowText()`:

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

Zapisuje ona tytuł okna o uchwycie `hWnd` do tablicy znaków ze wskaźnika podanego w `lpString`. I pozornie wszystko byłoby w porządku, gdyby nie rozmiar owej tablicy: musimy go ustalić z odpowiednią dozą rezerwy i podać w trzecim parametrze `GetWindowText()`. Takie są niestety uroki napisów w stylu C.

WinAPI oferuje nam jednak pewną pomoc: przy pomocy `GetWindowTextLength()` możemy mianowicie pobrać samą długość tytułu okna, czyli ilość znaków, jakie musi pomieścić docelowa tablica (bufor). Wielkości tej potrafimy natomiast użyć do zaalokowania bufora o odpowiedniej pojemności.

Ostatecznie możemy pokusić się o napisanie znacznie wygodniejszej funkcji, pobierającej tytuł okna i zwracającej go jako łańcuch `std::string`:

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

std::string GetCaption(HWND hWnd)
{
    char* lpszBuffer;

    // pobieramy długość napisu i alokujemy pamięć dla bufora
    UINT uLength = GetWindowTextLength(hWnd);
    lpszBuffer = new char [uLength];

    // odczytujemy napis i zapisujemy go w zmiennej typu std::string
    GetWindowText(hWnd, lpszBuffer, uLength);
    std::string strCaption = lpszBuffer;

    // zwalniamy bufor i zwracamy tekst
    delete[] lpszBuffer;
    return strCaption;
}
```

Jeśli jednak bardziej zależy nam na szybkości niż efektywności pamięciowej programu, to alokację i zwalnianie bufora o zmiennej wielkości możemy zastąpić poprzez dużą (np. 1024 znaki), statyczną tablicę znaków. Do niej będziemy od razu zapisywać tytuł okna, z pominięciem pobierania jego długości poprzez `GetWindowTextLength()`.

O wszystkich trzech funkcjach ([SetWindowText\(\)](#), [GetWindowText\(\)](#) i [GetWindowTextLength\(\)](#)) dobrze byłoby poczytać coś więcej w MSDN.

## Inne informacje

Na deser zostawiłem potężną funkcję pobierającą informacje o oknie - `GetWindowInfo()`:

```
BOOL GetWindowInfo(HWND hWnd, PWINDOWINFO pwi);
```

O jej możliwościach trudno wywnioskować z prototypu, jako że kryją się one w strukturze `WINDOWINFO`, do której wskaźnik musimy podać w drugim parametrze. Sama struktura przedstawia się zaś następująco:

```
struct WINDOWINFO
{
    DWORD cbSize;
    RECT rcWindow;
    RECT rcClient;
    DWORD dwStyle;
    DWORD dwExStyle;
    DWORD dwWindowStatus;
    UINT cxWindowBorders;
    UINT cyWindowBorders;
    ATOM atomWindowType;
    WORD wCreatorVersion;
};
```

Zawiera ona całe mnóstwo danych dotyczących okna, które możemy bez problemu pobrać przy pomocy wymienionej funkcji `GetWindowInfo()`. Oto krótkie omówienie wszystkich składowych `WINDOWINFO`:



<i>typ</i>	<i>pola</i>	<i>opis</i>
DWORD	cbSize	Podobnie jak w WNDCLASSEX, pierwszy pole struktury WINDOWINFO określa jej <b>rozmiar w bajtach</b> . Musimy ustawić je na <code>sizeof(WINDOWINFO)</code> , zanim zechcemy wywołać funkcję <code>GetWindowInfo()</code> .
RECT	rcWindow rcClient	Te dwa pola określają prostokąty zawierające (kolejno): <b>całe okno</b> oraz jego <b>obszar klienta</b> .
DWORD	dwStyle dwExStyle	Z tych pól możemy odczytać <b>styl</b> oraz <b>rozszerzony styl</b> okna, czyli wartości, które zostały ongiś przekazane do <code>CreateWindow[Ex]()</code> podczas tworzenia okna.
DWORD	dwWindowStatus	Pole to określa <b>status okna</b> , tzn. to, czy jest ono aktywne, czy też nie. Wartość stałej <code>WS_ACTIVECAPTION</code> w tym polu oznacza pierwszą sytuację, zero - drugą.
UINT	cxWindowBorders cyWindowBorders	Ta para pól zawiera <b>szerokość</b> oraz <b>wysokość obrzeża</b> okna w pikselach.
ATOM	atomWindowType	W tym polu zapisany zostaje <b>atom identyfikujący klasę okna</b> . Jak (mam nadzieję) pamiętasz, atom ten zwraca funkcja <code>RegisterClass[Ex]()</code> po rejestracji klasy okna, a wartość ta może zostać użyta w miejsce nazwy tejże klasy w niektórych funkcjach, jak np. <code>CreateWindow[Ex]()</code> .
WORD	wCreatorVersion	Określa windowsową wersję aplikacji, która stworzyła okna.

Tabela 35. Pola struktury WINDOWINFO

Z ciekawszych składowych można z pewnością wymienić `atomWindowType`, dającą informację o klasie okna, oraz `rcClient`, określającą jej obszar klienta.

Wymiary obszaru klienta okna można też uzyskać poprzez funkcję `GetClientRect()`.

## Uzyskiwanie uchwytów do okien

Spośród zaprezentowanych funkcji każda, co do jednej, wymagała podania uchwytu do okna. W sumie jest to naturalne, skoro funkcje te służą właśnie do operacji na oknach. Uchwyt taki trzeba jednak posiadać.

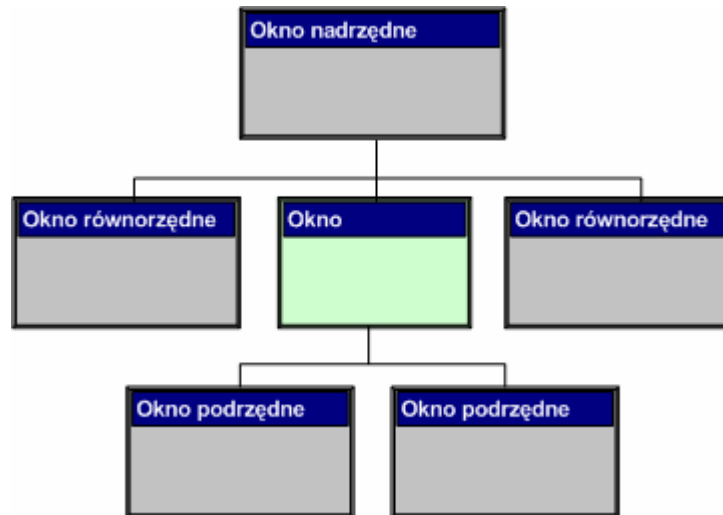
Zasadniczo nie jest to problemem, bo przecież funkcja tworząca okno, `CreateWindowEx()`, zwraca nam taki uchwyt typu `HWND`. Możemy jednak uzyskać uchwyty okien na wiele innych sposobów; co więcej, możliwe jest nawet pobranie identyfikatora od „nieswojego” okna! Spójrzmy zatem na funkcje, jakie Windows API oferuje nam w tym zakresie.

### Poruszanie się po hierarchii okien

Przypomnijmy, że każde stworzone w systemie okno należy do jego **hierarchii okien**. Wchodzi więc ono w różnorodne relacje z innymi istniejącymi oknami jako element swego rodzaju drzewa.

I tak dla każdego okna możemy wyróżnić nieraz całkiem liczną rodzinę, na którą składają się:

- **okno nadrzędne** albo **rodzic** (ang. *parent window*), znajdujące się o jeden poziom wyżej w hierarchii. Dla głównych okien aplikacji jest to `puplit`, one same stanowią zaś drugi poziom drzewa okien
- **okna równorzędne** albo **rodzeństwo** (ang. *sibling windows*), czyli takie okna, które istnieją na tym samym poziomie hierarchii i mają wspólnego rodzica
- **okna potomne** albo **dzieci** (ang. *child windows*), znajdujące się o jeden poziom niżej w hierarchii okien, mające rozpatrywane okno za rodzica



Schemat 40. Relacje między oknami w hierarchii

Najczęściej hierarchia rozciąga się na więcej niż trzy poziomy. Wówczas wszystkie okna powyżej rozważanego nazywamy jego **przodkami** (ang. *ancestors*), natomiast te poniżej - **potomkami** (ang. *descendants*).

Skoro znamy już nazewnictwo stosowane w hierarchii okien<sup>112</sup>, możemy nauczyć się uzyskiwać uchwyty do pokrewnych okien przy pomocy odpowiednich funkcji WinAPI.

Stosunkowo najprościej jest zdobyć uchwyt do okna nadrzędnego, ponieważ każde okno ma tylko jednego rodzica. Zwraca go funkcja `GetParent()`:

```
HWND GetParent(HWND hWnd);
```

Jej użycie ma sens dla kontroltek umieszczonych w oknie: wtedy bowiem podanie funkcji uchwytu do kontrolki skutkuje zwróceniem uchwytu do zawierającego ją okna. W przypadku jednak gdy funkcją `GetParent()` potraktujemy główne okno jakiejś aplikacji, nie otrzymamy, jak by się mogło wydawać, uchwytu okna pulpitu, lecz wartość `NULL`. Uchwyt pulpitu zdobędziemy natomiast poprzez `GetDesktopWindow()`.

Nieco trudniejsze jest pozyskanie okien równo- oraz podrzędnych - z tego względu, iż prawie zawsze istnieje wiele takich okien naraz. Windows API udostępnia nam wszakże funkcję `GetWindow()`:

```
HWND GetWindow(HWND hWnd, UINT uCmd);
```

Korzystając z niej, możemy poruszać się po aktualnym poziomie hierarchii okien<sup>113</sup> (oknach równorzędnych) lub też zejść niżej, do okien potomnych. Przeglądanie okien na danym poziomie odbywa się natomiast według ich porządku Z, czyli kolejności przesłaniania na ekranie. Możliwe jest więc przejście do okna leżącego zaraz „pod spodem” aktualnego oraz bezpośrednio „na nim” - pod warunkiem oczywiście, jest ono **na tym samym poziomie** hierarchii. `GetWindow()` daje ponadto możliwość skoku na wierzch i na sam spód porządku Z.

Wyboru interesującego nas działania dokonujemy, wpisując odpowiednią stałą w drugim parametrze funkcji:

<sup>112</sup> Jest ono zresztą stosowane nie tylko tam. Właściwie stosuje się ono do każdej struktury drzewiastej, używanej w programowaniu, a także np. do węzłów dokumentu XML.

<sup>113</sup> 'Aktualny' znaczy tutaj 'ten, na którym znajduje się okno o uchwycie podanym w pierwszym parametrze `GetWindow(), hWnd`'. Sądzę, że nietrudno było się tego domyślić :)

<i>stała</i>	<i>okno</i>	<i>opis</i>
GW_HWNDFIRST	najwyższe	Powoduje zwrócenie uchwytu do okna leżącego <b>na szczycie</b> kolejności przesłaniania, biorąc oczywiście pod uwagę tylko okna na tym samym poziomie, co te o uchwycie hWnd.
GW_HWNDPREV	poprzednie	Skutkiem użycia tej stałej jest otrzymanie okna <b>bezpośrednio przesłaniającego</b> okno hWnd i będącego naturalnie na tym samym poziomie hierarchii.
GW_HWNDNEXT	następne	Zwraca następne okno, leżące <b>bezpośrednio niżej</b> w kolejności przesłaniania, a na tym samym poziomie w hierarchii co hWnd.
GW_HWNDLAST	najniższe	Pobiera uchwyt okna, które jest <b>na spodzie</b> porządku Z i, rzecz jasna, na tym samym poziomie hierarchii okien.
GW_CHILD	potomne	Stała ta skutkuje zwróceniem uchwytu do <b>okna potomnego</b> względem hWnd, leżącego <b>na szczycie</b> kolejności przesłaniania.

**Tabela 36. Stałe pobierania uchwytów do okien w funkcji GetWindow()**

Istnieje oczywiście możliwość, że żadne okno o żądanych cechach nie zostanie znalezione - wtedy funkcji GetWindow() zwraca po prostu NULL (zero).

No dobrze, teoria teorią, ale jak skorzystać z tej funkcji w praktyce, ażeby np. wyliczyć wszystkie okna potomne względem danego?... Otóż odpowiedni kod może wyglądać tak:

```
// wyliczamy wszystkie dzieci okna hwndOkno

// pobieramy pierwsze dziecko (leżące na szczycie porządku Z)
HWND hwndDziecko = GetWindow(hwndOkno, GW_CHILD);

// uzyskujemy uchwyty do okien potomnych leżących niżej w z-order
do
{
    // tutaj coś robimy z uchwytem zapisanym w hwndDziecko
} while (GetWindow(hwndDziecko, GW_HWNDNEXT) /* != NULL */)
```

Możemy tak wyliczyć chociażby wszystkie główne okna aplikacji w systemie - wystarczy, że za hwndOkno podstawimy uchwyt pulpitu (przypominam, jest to wynik funkcji GetDesktopWindow()).

Zainteresowani, którzy z pewnością przeczytają [opis funkcji GetWindow\(\)](#) w MSDN, powinni jeszcze zwrócić uwagę na zagadnienie [posiadania okien](#) (ang. *owned windows*), również tam opisane.

### Poszukiwanie dowolnego okna

Potencjalnie ciekawsze możliwości na polu wyszukiwania okien posiada funkcja FindWindow():

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

Potrąfi ona znaleźć **dowolne okno** w systemie, należące do podanej klasy i/lub mające wskazany tekst na pasku tytułu. Przy jej pomocy możemy więc otworzyć sobie dostęp do okien innych aplikacji albo (co jest nawet bardziej interesujące) kluczowych okien systemu Windows.

Wyszukiwanie, jakiego dokonuje ta funkcja, może odbywać się przy pomocy dwóch kryteriów: klasy okna oraz jego tytułu. Nie musimy jednak korzystać z obu metod; jeśli w którymś z parametrów wpisujemy `NULL`, wówczas odpowiadające mu kryterium nie będzie po prostu brane pod uwagę.

Pierwszy parametr `lpClassName` jest przeznaczony dla nazwy klasy, której okna poszukujemy. Dopuszczalne jest tu podanie tej nazwy jako napisu w stylu C, można też wpisać atom odpowiedniej klasy. Wartość zerowa spowoduje natomiast, że wszystkie klasy okien będą pasować do wyszukiwania.

Drugi argument `lpWindowName` oczekuje na podanie tytułu okna - także jako tradycyjnego łańcucha znaków. I tak samo możemy wpisać tu `NULL`, aby pominąć dopasowywanie tytułów okien.

Zwracaną przez `FindWindow()` wartością jest uchwyt do okna głównego (*top-level*), pasującego do zadanych założeń. Możliwe jest rzecz jasna, iż żadne takie okno nie zostanie znalezione - wtedy otrzymujemy po prostu `NULL`.

`FindWindow()` przeszukuje tylko **okna główne** aplikacji, pomijając okna potomne.

Po tym teoretycznym wprowadzeniu czas na jakiś konkretny, interesujący przykład. Oto program, który potrafi ukrywać i pokazywać systemowy pasek zadań - czyli okno, które z całą pewnością nie należy do niego:

```
// TaskbarHider - program ukrywający pasek zadań

#include <string>
#define WIN32_LEAN_AND_MEAN
#include <windows.h>

// dane okna
std::string g_strKlasaOkna = "od0dogk_TaskbarHider_Window";
HWND g_hwndOkno = NULL;

// dane o pasku zadań
HWND g_hwndPasekZadan = NULL;
bool g_bWidocznyPasekZadan;

// ----- procedura zdarzeniowa okna -----

LRESULT CALLBACK WindowEventProc(HWND hWnd, UINT uMsg,
                                  WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_DESTROY:
            // odkrywamy pasek zadań
            ShowWindow (g_hwndPasekZadan, SW_SHOW);

            // kończymy program
            PostQuitMessage (0);
            return 0;

        case WM_LBUTTONDOWN:
            // zmieniamy stan widoczności na przeciwny
            g_bWidocznyPasekZadan = !g_bWidocznyPasekZadan;

            // pokazujemy/ukrywamy pasek zadań
            ShowWindow (g_hwndPasekZadan,
                        g_bWidocznyPasekZadan ? SW_SHOW : SW_HIDE);
    }
}
```

```

        // uaktywniamy własne okno i każemy je odrysować,
        // by pokazała się informacja
        SetFocus (hWindow);
        InvalidateRect (hWindow, NULL, true);

        return 0;

case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdcOkno;
    RECT rcObszarKlienta;

    // pobieramy obszar klienta naszego okna
    GetClientRect (hWindow, &rcObszarKlienta);

    // formatujemy napis
    std::string strNapis = "Pasek zadań jest ";
    strNapis += (g_bWidocznyPasekZadan ?
                "WIDOCZNY" : "NIEWIDOCZNY");

    // rysujemy napis informujący
    hdcOkno = BeginPaint(hWindow, &ps);
    DrawText (hdcOkno, strNapis.c_str(),
              (int) strNapis.length(),
              &rcObszarKlienta,
              DT_SINGLELINE | DT_CENTER | DT_VCENTER);
    EndPaint (hWindow, &ps);

    return 0;
}
}

return DefWindowProc(hWindow, uMsg, wParam, lParam);
}

// -----funkcja WinMain() -----

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int nCmdShow)
{
    /* rejestrujemy klasę okna */

    WNDCLASSEX KlasaOkna;

    // wypełniamy strukturę WNDCLASSEX
    ZeroMemory (&KlasaOkna, sizeof(WNDCLASSEX));
    KlasaOkna.cbSize = sizeof(WNDCLASSEX);
    KlasaOkna.hInstance = hInstance;
    KlasaOkna.lpfnWndProc = WindowEventProc;
    KlasaOkna.lpszClassName = g_strKlasaOkna.c_str();
    KlasaOkna.hCursor = LoadCursor(NULL, IDC_ARROW);
    KlasaOkna.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    KlasaOkna.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

    // rejestrujemy klasę okna
    RegisterClassEx (&KlasaOkna);

    /* tworzymy okno */

```

```

// tworzymy okno funkcja CreateWindowEx
g_hwndOkno = CreateWindowEx(WS_EX_TOOLWINDOW,
                            g_strKlasaOkna.c_str(),
                            "Ukrywacz paska zadań",
                            WS_OVERLAPPED | WS_BORDER
                            | WS_CAPTION | WS_SYSMENU,
                            CW_USEDEFAULT,
                            CW_USEDEFAULT,
                            250,
                            50,
                            NULL,
                            NULL,
                            hInstance,
                            NULL);

// pokazujemy nasze okno
ShowWindow (g_hwndOkno, nCmdShow);

/* uzyskujemy okno paska zadań */

// wywołujemy FindWindow(), by znaleźć uchwyt
g_hwndPasekZadan = FindWindow("Shell_TrayWnd", NULL);

// pobieramy stan widoczności paska (zapewne jest widoczny,
// ale ostrożność nie zaszkodzi :D)
g_bWidocznyPasekZadan = (IsWindowVisible(g_hwndPasekZadan) != FALSE);

/* pętla komunikatów */

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

// zwracamy kod wyjścia
return static_cast<int>(msgKomunikat.wParam);
}

```

Widoczność paska zadań możemy kontrolować, klikając w okno tego programu. W nim też widzimy informację o tym, czy ów pasek jest w danej chwili widoczny:



**Screen 59. Program ukrywający pasek zadań**

Jeżeli zaś chodzi o kod programu, to zawiera on sporo szczegółów, którymi nie musisz się obecnie zbytnio przejmować, a dotyczących głównie rysowania w oknie. Sporo wyjaśnień możesz znaleźć w komentarzach, jeszcze więcej - w opisie komunikatu `WM_PAINT` w tym rozdziale, a wszystko wyłożymy sobie dokładnie przy opisie biblioteki Windows GDI.

Z całej aplikacji najbardziej może nas interesować linijka:

```
g_hwndPasekZadan = FindWindow("Shell_TrayWnd", NULL);
```

Pokazuje ona świetnie, jak używać funkcji `FindWindow()`. W tym przypadku korzystamy z wyszukiwania według nazwy klasy okna - "Shell\_TrayWnd" jest bowiem klasą systemowego paska zadań. W wyniku wywołania funkcji otrzymujemy (jedyne) okno należące do tejże klasy, czyli właśnie pasek zadań. Teraz możemy więc zacząć swoją zabawę z nim :)

A kiedy znudzi nam się ukrywanie i odkrywanie paska, warto abyśmy zajrzeli do MSDN po opis funkcji [FindWindow\(\)](#), a także [FindWindowEx\(\)](#).

\*\*\*

Zakończony właśnie podrozdział stanowił przegląd najważniejszych operacji, jakich można dokonywać na oknach, oraz funkcji WinAPI, które do tego służą. Nie wyczerpują one naturalnie całego asortymentu dostępnych instrumentów, ale są, jak sądzę, najbardziej użytecznymi spośród nich. Nie oznacza to jednak, że z pozostałymi funkcjami obsługi okien również nie wypadałoby się zapoznać; w miarę swoich potrzeb powinien się więc uczyć to we własnym zakresie.

Teraz natomiast przejdziemy do idei, które nadają oknom cechy interaktywności - a więc do komunikatów o zdarzeniach. Przypomnimy sobie wiadomości o pętli komunikatów oraz przedstawimy najważniejsze rodzaje zdarzeń.

## Komunikaty o zdarzeniach

System Windows powiada okna o zainstnialych zdarzeniach, posługując się **komunikatami** (ang. *messages*). Komunikaty te są odbierane przez procedurę zdarzeniową okna, która zajmuje się ich przetwarzaniem. Praca ta jest zwykle widoczna jako objawy działania programu: za takie przesłanki uznajemy bowiem odpowiednie reakcje na kliknięcia w przyciski, przyciśnięcia klawiszy i tak dalej.

Komunikaty sterują więc funkcjonowaniem aplikacji i pozwalają jej działać zgodnie z oczekiwaniami programisty (czasami także i użytkownika :)) Realizacja tych oczekiwań odbywa się drogą poprawnej współpracy z mechanizmem komunikatów Windows. Na ten mechanizm ten składa się **pętla (pompa) komunikatów** (ang. *message loop*) oraz procedury zdarzeniowe okien. Ta pierwsza zajmuje się pobieraniem od systemu informacji o zdarzeniach i kierowaniem ich we właściwe miejsce; procedury zdarzeniowe są natomiast takim właśnie miejscem - w nich następuje odczytanie danych niesionych przez komunikat oraz ustalona przez twórcę aplikacji interpretacja zdarzenia.

Należyte wykorzystanie komunikatów wymaga stosownej wiedzy, przede wszystkim o ich rodzajach i odpowiadających im zdarzeniach systemowych. Tym właśnie zagadnieniem zajmiemy się w aktualnym podrozdziale, nie pomijając jednakże pozostałych kwestii związanych z systemem monitorowania zdarzeń w Windows.

### *Studium pętli komunikatów*

Jednym z praktycznych problemów związanych ze zdarzeniowym modelem działania programów jest jego nieprzystawanie do warunków, nazwijmy to, sprzętowych. Procesory komputerowe, czy nawet komputery w ogóle, są bowiem z zasady przystosowane do pracy sekwencyjnej, pasywnej - potrafią tylko wykonywać podany im kod maszynowy, nie dbając zupełnie o warunki „zewnętrzne”. Bez choćby najprostszego systemu operacyjnego (np. BIOSu), który podtrzymywałby komunikację z poszczególnymi podzespołami, mielibyśmy do czynienia tylko ze zlepkiem krzemowych obwodów, niepotrafiącym nawet zadbać o niezbędne zasilanie.

Procesor jest więc tylko robotnikiem, wymagającym ukierunkowania swojej pracy przez odpowiedniego nadzorcę. Zaś im bardziej szef jest wykwalifikowany, tym lepszy może zrobić użytek z działań swego podwładnego. Windows, wraz z wieloma innymi systemami operacyjnymi, należy do profesjonalnych menedżerów, którzy tak gospodarują czasem procesora, by stwarzać wrażenie interaktywności układu, który z założenia interaktywny nie jest.

Czyni to wszelako w dość prosty i narzucający się od razu sposób. Otóż dzieli on zasoby czasu procesora na bardzo krótkie (obecnie rzędu nanosekund) interwały, w których na przemian pozwala wykonywać swój kod poszczególnym programom, działającym „jednocześnie”. Tak szybkie przełączanie jest dla użytkownika oczywiście niezauważalne i dlatego wywołuje odczucie, iż wszystkie czynności są wykonywane w tym samym czasie. Równoległe system musi też liczyć się z możliwością zaistnienia niskopoziomych „zdarzeń sprzętowych” - **przerwań** (ang. *interrupts*), informujących (między innymi) o pojawieniu się nowych danych od urządzeń wejściowych. Misją systemu jest przeistoczenie tych zdarzeń we właściwe komunikaty i przesłanie ich do docelowych aplikacji. Wszystkie te zadania żądają funkcjonowania w sposób ciągły - **w pętli**.

Nie dziwi zatem, iż z programistycznego punktu widzenia model zdarzeniowy opiera się na pętlach. Część z nich może być głęboko ukryta - tak jest zapewne z niskopoziomym kodem rejestrowania zdarzeń w Windows. Istnieje jednak pętla, która bardzo interesuje każdego koderę; pętla, która pozwala programom sterowanym zdarzeniami działać w sposób ciągły; wreszcie pętla, która zajmuje się obsługą tych zdarzeń. W systemie Windows jest to **pętla komunikatów**.

Jako że jest to niezbędny element każdego programu okienkowego, spotkaliśmy się z nim już wcześniej. Ponieważ jednak jest to część modelu zdarzeniowego Windows, która wymaga głębszego omówienia, zajmiemy się nią bliżej właśnie teraz.

### *Traktat o wyższości PeekMessage ()*

Standardowa, elementarna postać pętli komunikatów jawi się następująco:

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

Treść jej bloku jest tak minimalistyczna, jak tylko może być. Mówiłem bowiem w poprzednim rozdziale, że wywołania `TranslateMessage()` i `DispatchMessage()` są absolutnie niezbędne do prawidłowej obsługi komunikatów. Tą częścią pętli nie będziemy się zatem na razie zajmować.

Znacznie więcej uwagi poświęcimy natomiast funkcji `GetMessage()`.

### *Funkcja GetMessage ()*

Bliższy wgląd w każdą nowopoznaną funkcję najlepiej zacząć od spojrzenia na jej prototyp. Deklaracja `GetMessage()` przedstawia się więc mniej więcej tak:

```
BOOL GetMessage(LPMSG lpMsg,
                HWND hWnd,
                UINT wMsgFilterMin,
                UINT wMsgFilterMax);
```

Cztery parametry tej funkcji wydają się aż nadmiarem dobroci, lecz mają one właściwe sobie znaczenie. Zdecydowanie nie zaszkodzi nam, jeżeli je poznamy. Spójrzmy przeto na poniższą tabelkę:



<b>typ</b>	<b>nazwa</b>	<b>opis</b>
LPMSG	lpMsg	To jest <b>wskaźnik na strukturę</b> typu MSG. W tej strukturze zostanie zapisany pobrany komunikat Windows, abyśmy mogli przekazać go funkcjom <code>TranslateMessage()</code> i <code>DispatchMessage()</code> .
HWND	hWnd	Podajemy tu <b>uchwyt okna</b> , którego docelowe komunikaty chcemy pobierać. Najczęściej jednak chcemy zająć się wszystkimi komunikatami i wobec tego podajemy tutaj <code>NULL</code> .
UINT	wMsgFilterMin wMsgFilterMax	<p>Jak pamiętamy, stałe komunikatów są liczbami, a więc mogą być ułożone według ich wartości. W tych dwóch parametrach możemy zaś podać <b>zakres</b> liczb, które nas interesują, uzyskując w ten sposób odfiltrowanie tylko tych komunikatów, które nas w danej chwili obchodzą.</p> <p>Zazwyczaj nie ma takiej potrzeby i dlatego podajemy tu dwa razy wartość zerową. Wtedy <code>GetMessage()</code> otrzymuje <b>wszystkie</b> komunikaty o zdarzeniach.</p> <p>Jest też kilka specjalnych stałych, które podane razem umożliwiają wyłowienie pewnego rodzaju zdarzeń. Na przykład <code>WM_KEYFIRST</code> i <code>WM_KEYLAST</code> powoduje odbieranie tylko komunikatów od klawiatury.</p> <p>Więcej informacji na ten temat możesz znaleźć w <a href="#">opisie funkcji GetMessage()</a> w MSDN.</p>

**Tabela 37. Parametry funkcji GetMessage()**

Jako typ zwracanej wartości `GetMessage()` deklaruje `BOOL`. Mimo to zwraca ona teoretycznie aż trzy możliwe wartości:

- niezerową, gdy funkcja poprawnie odebrała komunikat różny od `WM_QUIT`
- zerową, kiedy został poprawnie odebrany komunikat `WM_QUIT`
- `-1` w przypadku błędu

Z tego powodu zaleca się właściwie taką oto formę pętli komunikatów:

```

BOOL bWynik;
MSG msgKomunikat;

while ((bWynik = GetMessage(&msgKomunikat, NULL, 0, 0) != NULL)
{
    if (bWynik == -1)
    {
        // obsłuż błąd
    }
    else
    {
        TranslateMessage (&msgKomunikat);
        DispatchMessage (&msgKomunikat);
    }
}

```

Powyższa postać umożliwia poprawne zareagowanie na potencjalne błędy, czego nie zapewnia krótszy wariant, jaki zaprezentowaliśmy wcześniej. Nie oznacza to jednak, iż jest on gorszy. Wystąpienie błędu przy zaprezentowanym wywołaniu `GetMessage()` byłoby bowiem sytuacją nadzwyczaj krytyczną, której zaistnienie jest w zasadzie czysto

teoretyczne<sup>114</sup>. Nasza zwyczajna, sześciolinijkowa pętla komunikatów jest więc **całkowicie poprawna**.

### Niedostatki `GetMessage()`

Jakkolwiek `GetMessage()` wydaje się dobrze spełniać swoje zadanie, nie jest od wolna od mankamentów - konkretnie dwóch.

Pierwszym jest jej specjalne zachowanie się w przypadku odebrania komunikatu `WM_QUIT`, czyli zwrócenie wtedy wartości niezerowej; w innym razie funkcja zwraca zero. Jest to zupełne pogwałcenie zasady obowiązującej dla większości pozostałych funkcji Windows API, na mocy której zwrócenie zera oznacza błąd.

Co gorsza, fakt ten może być także przyczyną złej interpretacji prawidłowo napisanego kodu, np.:

```
while (GetMessage(&msgKomunikat, NULL, 0, 0)) { /* ... */ }
```

Mając w pamięci poprzednią uwagę można przypuszczać, że powyższa pętla będzie się wykonywała aż do momentu wystąpienia błędu w wywołaniu `GetMessage()`. To oczywiście nieprawda, podobnie jak przypuszczanie, iż to nieodebranie żadnego komunikatu jest warunkiem terminalnym pętli (przypominam po raz kolejny, że w tej postaci jest nim odebranie zdarzenia `WM_QUIT`).

Niecodzienne zachowanie `GetMessage()` dla `WM_QUIT` może też rodzić inny problem. Pojawia się on wtedy, gdy chcemy uzależnić przerwanie działania programu od wystąpienia także innego komunikatu. Takim komunikatem może być chociażby `WM_ENDSESSION`, wysyłany w momencie kończenia pracy całego systemu. Gdy chcemy zapewnić obsługę tego komunikatu, musimy uciekać się do kodu podobnego do tego:

```
MSG msgKomunikat;
while (GetMessage(&msgKomunikat, NULL, 0, 0))
{
    if (msgKomunikat.message == WM_ENDSESSION) return 0;
    TranslateMessage (&msgKomunikat);
    DispatchMessage (&msgKomunikat);
}
```

Niestety, patrząc na niego można bardzo łatwo ulec mylnej sugestii, że za przerwanie pętli i programu odpowiada **wyłącznie** komunikat `WM_ENDSESSION`! Nietrudno bowiem zapomnieć o ukrytym sprawdzaniu wystąpienia `WM_QUIT`, dokonywanego w funkcji `GetMessage()`.

Drugi kłopot jest teraz niezbyt dla nas zauważalny, ale tak naprawdę ma kolosalne znaczenie. Otóż wywołanie `GetMessage()` trwa **długo** - dokładniej mówiąc, trwa ono tak długo, aż w kolejce pojawi się jakiś komunikat do odebrania, jeśli dotychczas nie było żadnego. Ten czas jest zwykle liczony w milisekundach i dla użytkownika nie ma rzecz jasna żadnego znaczenia, ale dla programu oznacza to miliony straconych cykli procesora, które mogłyby przeznaczyć na swe czynności.

Większość aplikacji nie potrzebuje aczkolwiek każdego wolnego zasobu obliczeniowego, lecz dla wielu są one niezwykle pożądane. Do takich programów należą gry, wykonujące renderowanie kolejnych klatek między komunikatami, a także wszelkie działające w tle programy obliczeniowe typu klienty SETI@home czy distributed.net. Dla nich strata czasu na jałowe wykonywanie się `GetMessage()` jest nie do przyjęcia.

<sup>114</sup> Gdyby bowiem wystąpiła, to śmiało możnaby wątpić, czy po deklaracji w rodzaju `int x = 3;` zmienna `x` na pewno zawiera wartość `3` ;)

Dlatego w przypadku takich właśnie aplikacji (które przecież docelowo chcemy pisać w tym kursie) należy wynaleźć inny sposób na rozwiązanie problemu pętli komunikatów.

Nie znaczy to jednak, że ten prosty wariant sześcioletni, zaprezentowany na samym początku kursu WinAPI, jest zły. Faktycznie sprawdza się on w zasadzie **każdym programie użytkowym** i należy go stosować w takich aplikacjach.

Przedstawiamy `PeekMessage()`

Domyślasz się chyba, że narzekania na `GetMessage()`, którymi uraczyłem cię w poprzednim paragrafie, nie były całkiem bezproduktywne. Łajana procedura posiada bowiem bardziej elastycznego kuzyna w postaci funkcji `PeekMessage()`:

```
BOOL PeekMessage(LPMSG lpMsg,
                HWND hWnd,
                UINT wMsgFilterMin,
                UINT wMsgFilterMax,
                UINT wRemoveMsg);
```

Prototyp `PeekMessage()` wydaje się bardzo podobny do deklaracji `GetMessage()`. Między obiema funkcjami występują jednak dość spore różnice:

- `PeekMessage()` posiada dodatkowy parametr `wRemoveMsg`. Określamy w nim, czy pobierany komunikat ma być następnie usunięty z kolejki:
  - ✓ podanie stałej `PM_NOREMOVE` spowoduje pozostawienie komunikatu w kolejce (będzie można pobrać go ponownie)
  - ✓ wartość `PM_REMOVE` skutkuje usunięciem zdarzenia z kolejki

W tym parametrze można również podać kilka innych stałych, umożliwiających wybór rodzaju komunikatów do pobrania (niezależnie od wartości `wMsgFilterMin` i `wMsgFilterMax`). O tych stałych możesz poczytać w [MSDN](#).

- w przeciwieństwie do `GetMessage()`, funkcja `PeekMessage()` **nie czeka** na pojawienie się nowego komunikatu w kolejce, jeżeli ta jest pusta. W takiej sytuacji zwraca po prostu odpowiednią informację - a zatem...
- rezultatem funkcji `PeekMessage()` jest tylko informacja o wykrytej obecności (wartość niezerowa) lub nieobecności (zero) komunikatu w kolejce. Żadne inne czynniki nie wpływają na wynik funkcji - wobec tego...
- funkcja `PeekMessage()` nie traktuje w sposób specjalny ani `WM_QUIT`, ani żadnego innego komunikatu

*Pętla komunikatów z `PeekMessage()`*

Znając te nowe cechy, możemy napisać pętlę komunikatów z użyciem funkcji `PeekMessage()`:

```
MSG msgKomunikat;
for (;;)
{
    if (PeekMessage(&msgKomunikat, NULL, 0, 0, PM_REMOVE))
    {
        if (msgKomunikat.message == WM_QUIT) break; // 115
    }
}
```

<sup>115</sup> Zamiast porównania pola `message` można użyć wywołania `GetMessage()`, jako że funkcja ta dokonuje sprawdzenia komunikatu względem `WM_QUIT`. Wówczas jednak należałoby zmienić ostatni argument `PeekMessage()` na `PM_NOREMOVE`. Powstała pętla działałaby identycznie do podanej tutaj, z tym że ponowne pobranie tego samego komunikatu (dokonane już w `PeekMessage()`), a przeprowadzane po raz kolejny w

```

        TranslateMessage (&msgKomunikat);
        DispatchMessage (&msgKomunikat);
    }
}

```

Po takiej pętli powinna jeszcze wystąpić niezbędna instrukcja `return static_cast<int>(msgKomunikat.wParam);`, zwracająca poprawny kod wyjścia. Nie będę jej powtarzał w kolejnych kodach, jako że w zasadzie nie jest ona częścią pętli komunikatów.

Mimo pozornego skomplikowania pętla ta jest tak naprawdę łatwiejsza do rozwikłania. Dzieje się tak za sprawą jawnego przyrównywania rodzaju komunikatu do `WM_QUIT`.

Co niektórzy mogą jeszcze kręcić nosem na występującą tutaj pętlę nieskończoną `for (;)`. Pozbycie się jej nie sprawia jednak żadnego kłopotu, gdyż kryterium przerwania iteracji może być z łatwością przeniesione do warunku pętli `while`:

```

MSG msgKomunikat;
msgKomunikat.message = WM_NULL; // 116

while (msgKomunikat.message != WM_QUIT)
{
    if (PeekMessage(&msgKomunikat, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage (&msgKomunikat);
        DispatchMessage (&msgKomunikat);
    }
}

```

Wymaga to jeszcze początkowego wyzerowania pola określającego rodzaj komunikatu w strukturze `MSG`. Wartość `WM_NULL` oznacza taki właśnie „zerowy komunikat”. Dalej pętla przebiega w identyczny sposób, jak w poprzednim wydaniu.

### *Nie tracąc czasu*

Przewaga `PeekMessage()` nad `GetMessage()` ujawnia się najbardziej w momencie, gdy chcemy wykonywać jakieś dodatkowe działania **między przetwarzaniem** kolejnych komunikatów. Podałem już przykłady takich czynności, wśród których najważniejsze miejsce zajmuje generowanie grafiki w grach komputerowych oraz demach.

Osiągnięcie tego celu nie jest trudne, jeżeli pamiętamy, o czym informuje nas wartość zwracana przez `PeekMessage()`. Warunek:

```

if (PeekMessage(&msgKomunikat, NULL, 0, 0, PM_REMOVE))

```

jest więc prawdziwy tylko wówczas, gdy w kolejce komunikatów oczekiwało nieobsłużone jeszcze zdarzenie. W innym przypadku kod bloku `if` (czyli wywołania dwóch niezbędnych funkcji dla pobranego komunikatu) nie wykona się wcale.

Ten „inny przypadek” jest jednak właśnie tym, którego tak usilnie poszukujemy! Aby zatem spowodować wykonywanie jakiegoś kodu w czasie, gdy program nie musi

---

`GetMessage()` byłoby operacją nadmiarową; utracilibyśmy też klarowność bezpośredniego podania warunku przerwania pętli.

<sup>116</sup> Początkowe ustawienie pola `message` można zamienić na wywołanie `PeekMessage()` identyczne jak wewnątrz pętli, jeżeli obecność dwóch takich samych instrukcji w bliskim sąsiedztwie jest dla nas do przyjęcia.

zajmować się żadnym komunikatem, należy po prostu dodać odpowiednią frazę `else` do omawianej instrukcji `if`:

```
MSG msgKomunikat;
msgKomunikat.message = WM_NULL;

while (msgKomunikat.message != WM_QUIT)
{
    if (PeekMessage(&msgKomunikat, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage (&msgKomunikat);
        DispatchMessage (&msgKomunikat);
    }
    else
    {
        // tutaj możemy wpisać dodatkowy kod, wykonywany wtedy,
        // kiedy aplikacja nie ma już żadnych komunikatów do obsłużenia
    }
}
```

Pętla komunikatów w tej postaci jest już całkowicie wystarczająca dla prostych czy nawet bardziej skomplikowanych gier - niezależnie od tego, czy do wyświetlania grafiki używają one DirectX czy też innych bibliotek. Można ją również wykorzystać dla wszelkiego rodzaju programów działających w tle.

Ogólnie mówiąc, pętla ta jest więc właściwa dla wszystkich aplikacji działających zgodnie z **modelem czasu rzeczywistego**.

Powyższą pętlę zdołamy jeszcze udoskonalić, kiedy już na poważnie zajmiemy się programowaniem gier. Można bowiem poprawić efektywność jej działania dla aplikacji pełnoekranowych.

## Struktura komunikatu

W kodzie pętli komunikatów wielokrotnie używaliśmy między innymi struktury `MSG`. W niej też zostawały zapisywane dane o komunikatach odebranych przez `Get/PeekMessage()` oraz przesyłanych ostatecznie do okna. Czas teraz spojrzeć z bliska na budowę komunikatu.

Deklarację typu strukturalnego `MSG` można przedstawić w następujący sposób:

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

Zawiera ona wyszczególnienie sześciu pól, z których wszystkie zostały opisane w poniższej tabelce:

<i>typy</i>	<i>nazwy</i>	<i>opis</i>
HWND	hwnd	Te cztery pola są identyczne z parametrami procedury zdarzeniowej, która ostatecznie otrzymuje komunikat.
UINT	message	
WPARAM	wParam	
LPARAM	lParam	

<i>typy</i>	<i>nazwy</i>	<i>opis</i>
DWORD	time	Zapisywany jest tu <b>czas wystąpienia zdarzenia</b> . Format tego czasu jest przy tym dość niecodzienny, gdyż stanowi go liczba milisekund (tysięcznych części sekundy), jakie upłynęły od startu systemu.  Aktualny czas w tym samym formacie zwraca też funkcja <a href="#">GetTickCount()</a> . Natkniemy się na nią niejednokrotnie.
POINT	pt	To pole przechowuje <b>pozycję kursora myszy</b> w chwili zaistnienia zdarzenia. Typ POINT tego pola jest zdefiniowany jako bardzo prosta struktura:  <pre>struct POINT {     LONG x;     LONG y; };</pre> <p>Jak się pewnie domyślasz, <i>x</i> określa współrzędną poziomą, a <i>y</i> koordynat pionowy kursora.</p>

Tabela 38. Pola struktury MSG

Pola `time` i `pt` nie mają zbyt wielkiego znaczenia, jako że nie trafiają one ostatecznie do procedury zdarzeniowej okna. Dlatego też zajmiemy się przede wszystkim czterema pierwszymi polami, będącymi jednocześnie parametrami procedury.

### Uchwyt do okna

Pod nazwą `hwnd` kryje się uchwyt okna, którego dotyczy zdarzenie. Mówiłem już, że dzięki temu możliwe jest napisanie jednej procedury zdarzeniowej do obsługi wielu okien. Windows umożliwia bowiem istnienie więcej niż jednego okna danej klasy, a ponieważ klasa okna ma przypisaną tylko jedną procedurę zdarzeniową, więc musi być ona przygotowana do pracy z wieloma uchwytami do okien.

Trzeba też zaznaczyć, że procedura zdarzeniowa danej klasy okna będzie otrzymywała **wyłącznie** takie zdarzenia, których **macierzyste okna** (o uchwytach podanych w polu `hwnd` struktury MSG) będą **należały do tejże klasy**.

To zagmatwane (ale tylko pozornie ;D) stwierdzenie staje się ważne, gdy zaczynamy wyposażać okna w kontrolki potomne. Wówczas np. kliknięcie w przycisk, chociaż jest zdarzeniem samego przycisku, zostaje podane do procedury zdarzeniowej jako pochodzące od okna nadrzędnego względem przycisku.

O kontrolkach będziemy jeszcze obszernie mówić w przyszłych rozdziałach.

### Stała komunikatu

Pole `message` zawiera stałą określającą rodzaj komunikatu - czyli po prostu rodzaj zdarzenia. Pole to odpowiada drugiemu parametrowi procedury zdarzeniowej, zwanemu zwykle `uMsg`.

W Windows każdemu komunikatowi odpowiada właściwa stała zdefiniowana w pliku nagłówkowym `winuser.h`. Takich stałych jest bardzo dużo, co więcej możliwe jest także definiowanie własnych komunikatów, przydatnych w określonych sytuacjach. Spośród mnogości komunikatów najważniejsze są te, których nazwy rozpoczynają od `WM_`. Są to bowiem **jedyne** komunikaty odbierane przez procedury zdarzeniowe zwykłych okien - takich, jakimi zajmujemy się w tym rozdziale. Oprócz nich istnieją również komunikaty przeznaczone do pracy z kontrolkami potomnymi (jak na przykład `BM_` dotyczące się przycisków czy `EM_` od pól tekstowych). Tych aczkolwiek nie odbieramy w procedurach zdarzeniowych, chyba że stosujemy bardziej zaawansowane techniki

programowania Windows, zwane *subclassingiem* i *superclassingiem*. Zapewniam, że nie ominie cię poznanie ich ;)

Zdecydowanie najczęściej będziemy jednak zajmować się „zwykłymi” komunikatami okien, rozpoznawanymi po przedrostku `WM_`. Poznanie najważniejszych ich typów, tak samo jak zaznajomienie się z kluczowymi funkcjami Windows API, jest niezbędne do opanowania sztuki programowania Windows.

### Parametry komunikatu

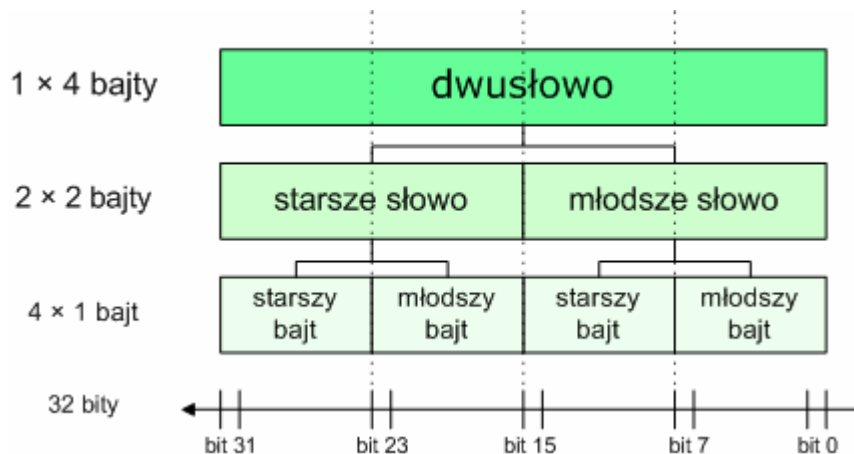
Wartości `wParam` oraz `lParam` zawierają dodatkowe dane o zainstniałym zdarzeniu. Ich dokładne znaczenie zależy od rodzaju komunikatu, jednak dzielą one wspólny sposób reprezentacji tych pomocniczych informacji.

Nazwy `wParam` i `lParam` pochodzą jeszcze z czasów 16-bitowych wersji Windows. Tam też parametr `wParam` był typu `WORD` (16-bitów), zaś `lParam` - `LONG` (32-bity). Stąd wzięły się te nazwy z przedrostkami, które zdołały przetrwać do dziś.

Oba pola (parametry) są 32-bitowymi liczbami całkowitymi bez znaku, co w Windows odpowiada typowi `DWORD` (aczkolwiek formalne typy tych pól to `WPARAM` i `LPARAM`). `DWORD` oznacza natomiast tak zwane **podwójne słowo** lub **dwusłowo** (ang. *double word*); jest to tradycyjna nazwa dla zespołu czterech bajtów złączonych w jedną wartość.

Miano to bierze się stąd, iż taki zespół składa się z dwóch 16-bitowych (2 bajtowych) liczb, zwanych **słowami**<sup>117</sup> (ang. *words*). Wyróżniamy przy tym tzw. **starsze słowo** (ang. *high-order word*), stanowiące górną połówkę wartości typu `DWORD`, oraz **młodsze słowo** (ang. *low-order word*) - dolną połowę.

Jakkolwiek brzmi to teraz dość abstrakcyjnie, powinno się wyjaśnić po spojrzeniu na poniższy schemat:



**Schemat 41. Podział dwusłowa na pojedyncze słowa i bajty. Na dole jest też pokazana numeracja bitów, poczynając od najmniej znaczącego (ang. *least significant bit* - LSB, bit 0) do najbardziej znaczącego (ang. *most significant bit* - MSB, tutaj bit 31).**

Określenia 'starszy' i 'młodszy', 'górnny' i 'dolny' oraz '(naj)bardziej' i '(naj)mniej znaczący' są synonimami i dotyczą zarówno słów, jak i bajtów oraz bitów. To, że dana

<sup>117</sup> Istnieje jeszcze termin słowa maszynowego (ang. *machine word*), odnoszący się do ciągu bitów o rozmiarze równym wielkości rejestru procesora na danych komputerze. Zatem dla naszych pecetów oznaczałoby ono 32 bity i dlatego czasami właśnie taką sekwencję nazywa się słowem. Dzieje się tak jednak bardzo rzadko (głównie w publikacjach naukowych), a w ogromnej większości wszelkich dokumentacji programistycznych (na czele z opisem WinAPI i DirectX) termin 'słowo' odpowiada ciągowi dokładnie 16-bitów, niezależnemu od używanej platformy sprzętowej.

część liczby jest starsza od innej oznacza, że jej zmiana ma większy wpływ na całą wartość.

Przykładowo, jeżeli mamy dwubajtowe słowo i dodamy 1 do jego dolnego bajtu, to całą liczbę zwiększymy również o jeden. Gdy jednak zinkrementujemy górny bajt, wówczas wartość słowa zwiększy się aż o 256.

Dużą pomocą w zrozumieniu tego mechanizmu będzie dla ciebie z pewnością Dodatek B, *Reprezentacja danych w pamięci*.

No dobrze, ale jak te szczegóły budowy dwusłowa mają się do komunikatów Windows?... Otóż są one związane bardzo ściśle. System operacyjny często bowiem wykorzystuje zmienne `wParam` i `lParam` nie jako liczby 32-bitowe, ale właśnie jako zespoły słów czy nawet bajtów. Robi tak, gdyż nierzadko potrzebuje przekazać więcej niż dwie wartości jako parametry zdarzenia; zapisuje je więc w dwubajtowych połówkach pól `wParam` i `lParam`. W ten sposób uzyskuje możliwość przechowania czterech wartości zamiast dwóch (oczywiście kosztem mniejszego zakresu liczb, ale zwykle nie jest to problemem).

Dla programisty piszącego kod obsługi komunikatów takie upakowanie wartości nie jest przy tym specjalnym problemem. Windows API udostępnia bowiem kilka użytecznych makr, potrafiących wyłuskać interesujące nas fragmenty dwusłów. Należą do nich:

- `HIWORD()` i `LOWORD()`, pobierające odpowiednio: starsze oraz młodsze słowo z wartości 32-bitowej. Wyrażenie `HIWORD(wParam)` zwróci więc górne 16-bitów z wartości `wParam`, zaś `LOWORD(lParam)` mniej znaczącą połówkę parametru `lParam`
- `HIBYTE()` i `LOBYTE()`, wyciągające starszy oraz młodszy bajt z wartości 16-bitowej. Makra te są przydatne, gdy w jednym z parametrów upakowano więcej niż dwie wartości. I tak np. `LOBYTE(HIWORD(wParam))` poda nam dolny bajt z górnego słowa pola `wParam` (w sumie będzie to więc trzeci bajt, licząc od prawej), zaś poprzez `HIBYTE(LOWORD(lParam))` możemy uzyskać starszy bajt młodsze słowa `lParam` (a więc drugi bajt od prawej)

Uważajmy, gdyż mamy tu do czynienia z makrami, które **nie dokonują sprawdzenia typów** tak jak funkcje. Zwracajmy zatem uwagę, by do `HIWORD()` oraz `LOWORD()` podawać tylko wartości 32-bitowe, a do `HIBYTE()` i `LOBYTE()` - 16-bitowe.

Oprócz powyższych makr WinAPI deklaruje też kilka bardziej konkretnych, przeznaczonych do współpracy ze specyficznymi komunikatami. Należy do nich na przykład `GET_X_LPARAM()`, służące do pobrania współrzędnej poziomej kursora przy obsłudze komunikatów myszy. O takich specjalnych makrach powiemy sobie, omawiając właściwe im zdarzenia.

Windows API ma też makra działające w odwrotny sposób do powyższych, tj. składające jedną większą wartość z dwóch połówek. Do takich makr należy `MAKELONG()`, tworzące słowo z dwóch bajtów, oraz `MAKELONG()`, łączące dwa słowa w jedno podwójne słowo. Listę wszystkich makr WinAPI możesz naturalnie znaleźć w [MSDN](#).

Przy opisach omawianych komunikatów będę rzecz jasna podawał, jak zapisane w `wParam` i `lParam` są ich ewentualne parametry. Informacje te są również w klarowny sposób podane w MSDN.

## *Komunikaty o zdarzeniach okna*

Na koniec tego podrozdziału przyjrzymy się jeszcze kilku najważniejszym komunikatom, związanym ze zdarzeniami pochodzącymi od samych okien.



## Tworzenie i niszczenie

Podczas tworzenia okna oraz w trakcie jego usuwania wysyłanych jest kilka komunikatów. Dzięki temu okno może zareagować odpowiednio na te dwie kluczowe dla niego akcje.

### *WM\_CREATE*

Komunikat `WM_CREATE` jest wysyłany do okna tuż po jego **stworzeniu**. Czyni to funkcja `CreateWindow[Ex]()`, czekając przy tym na obsłużenie przesłanego zdarzenia. Nie zakończy ona zatem swojej pracy, zanim procedura zdarzeniowa nowostworzonego okna nie zajmie się komunikatem `WM_CREATE`. Ma to swoje uzasadnienie, które podamy za chwilę.

Komunikat `WM_CREATE` przynosi ze sobą tylko jeden dodatkowy parametr, odczytywany z wartości `lParam`:

```
lpCreateStruct = reinterpret_cast<LPCREATESTRUCT>(lParam);
```

Za nim kryje się jednak całkiem spora struktura typu `CREATESTRUCT`, na którą ów parametr wskazuje:


```
struct CREATESTRUCT
{
    DWORD dwExStyle;
    LPCTSTR lpzClass;
    LPCTSTR lpzName;
    LONG style;
    int x;
    int y;
    int cx;
    int cy;
    HWND hwndParent;
    HMENU hMenu;
    HINSTANCE hInstance;
    LPVOID lpCreateParams;
};
```

Nie dziwiłbym się gdyby jej pola były dla Ciebie znajome. Są to bowiem dokładne odpowiedniki parametrów funkcji `CreateWindowEx()`, wywoływanej w celu utworzenia okna. Jest wśród nich także pole `lpCreateParams`, odpowiadające ostatniemu parametrowi funkcji, przeznaczonemu do swobodnego użytku programisty.

Gdy własnoręcznie przetwarzamy komunikat `WM_CREATE`, możemy w procedurze zdarzeniowej zwrócić rezultat tego działania. Zwykle jest to `0`, które w przypadku wszystkich komunikatów informuje system o poprawnym wykonaniu. Możliwe jest jednak zwrócenie `-1`. Wtedy Windows uznaje, że program nie akceptuje utworzonego okna i nakazuje jego zniszczenie. System posłusznie spełnia to żądanie, a wtedy funkcja `CreateWindow[Ex]()` (oczekująca na przetworzenie `WM_CREATE`) zwraca w wyniku `NULL`. Tworzenie okna kończy się wówczas niepowodzeniem, spowodowanym wyraźnym życzeniem aplikacji.

Co robimy w reakcji na `WM_CREATE`? Do najczęstszych czynności należy z pewnością utworzenie okien potomnych, chociażby kontrolek. W tym miejscu można też stworzyć obiekty, zaalokować pamięć oraz przygotować zasoby, które będą oknu potrzebne do pracy.

*WM\_CLOSE*

Okno odnotowuje zdarzenie `WM_CLOSE`, gdy użytkownik zechce je zamknąć. Jak wiemy, może to uczynić poprzez pojedyncze kliknięcie w przycisk  lub dwukrotne w ikonę okna w lewym górnym rogu.

Tak naprawdę jednak Windows nie wykonuje wtedy żadnej czynności, którą można by nazwać 'zamknięciem' okna. Takie pojęcie nie funkcjonuje w Windows API<sup>118</sup>; okno może być co najwyżej zniszczone, co wiąże się ze zwolnieniem wszystkich związanych z nim zasobów systemowych, z uchwycem na czele. Odpowiada za to funkcja

`DestroyWindow()`.

Funkcja ta jest zresztą wywoływana w domyślnej procedurze zdarzeniowej

`DefWindowProc()`, do której trafiają wszystkie nieobsłużone komunikaty okna. Tak więc:

**Jeżeli nie napiszemy własnego kodu obsługi komunikatu `WM_CLOSE`, to będzie on zawsze powodował zniszczenie okna.**

Niekiedy nam to odpowiada - wtedy po prostu nie zajmujemy się tym komunikatem. Jeśli jednak chcemy podjąć jakieś inne akcje przy zamykaniu okna, wtedy należy napisać kod obsługi zdarzenia `WM_CLOSE`.

Do najbardziej typowych zadań, jakie można w nim podjąć, zalicza się zapytanie użytkownika o potwierdzenie chęci zamknięcia okna. Dzieje się tak szczególnie często w przypadku głównych okien aplikacji, których zniszczenie pociąga za sobą zakończenie pracy całego programu. Wtedy też kod obsługi `WM_CLOSE` może się przedstawiać następująco:

```
case WM_CLOSE:
{
    // pytamy o potwierdzenie zakończenia działania programu
    if (MessageBox(hWnd, "Czy na pewno chcesz zakończyć program?",
        "Zakończenie", MB_YESNO | MB_ICONQUESTION) == IDYES)
        // niszczymy główne okno, co najpewniej zakończy cały program
        DestroyWindow (hWnd);

    return 0;
}
```

Komunikat ten nie ma żadnych dodatkowych parametrów, więc zmienne `wParam` i `lParam` są niewykorzystane. Również wartość, jaką zwrócimy przy jego (ewentualnym) przetwarzaniu nie ma znaczenia - tradycyjnie więc może być to zero.

*WM\_DESTROY*

Funkcja `DestroyWindow()`, przywoływana (zazwyczaj) w reakcji na `WM_CLOSE`, powoduje zniszczenie okna. W czasie tego procesu okno otrzymuje jeszcze jeden komunikat - `WM_DESTROY`.

Zdarzenie to ma już charakter czysto informacyjny, ponieważ na tym etapie nie ma już żadnych szans na ocalenie niszczonego okna. W tym momencie powinno ono tylko zwolnić obiekty i zasoby, które stworzyło u swych narodzin - jakkolwiek większość z nich, na czele z kontrolkami potomnymi, zostanie zniszczona automatycznie.

Główne okna aplikacji, napotykając `WM_DESTROY`, czynią jeszcze jedną ważną czynność: wywołują funkcję `PostQuitMessage()`:

```
case WM_DESTROY:
```

<sup>118</sup> Czasem jest może tylko utożsamiane z minimalizacją okna, jakkolwiek dziwnie by to brzmiało.

```
PostQuitMessage (0);
return 0;
```

Pamiętamy, że powoduje ona zakończenie pracy programu poprzez wysłanie komunikatu WM\_QUIT.

WM\_DESTROY, podobnie jak WM\_CLOSE, nie niesie żadnych pomocniczych informacji, a jego obsługa powinna zakończyć się zwróceniem zera.

*WM\_QUIT*

Ten komunikat jest pod kilkoma względami wyjątkowy. Jego przetwarzaniem zajmuje się bowiem nie procedura zdarzeniowa okna, lecz pętla komunikatów. Komunikat ten powoduje zresztą przerwanie tej pętli, a tym samym zakończenie programu - jest więc ostatnim zdarzeniem odbieranym przez aplikację.

WM\_QUIT posiada jeden parametr, zapisywany w wParam:

```
nExitCode = static_cast<int>(wParam);
```

Jest to **kod wyjścia** (ang. *exit code*) aplikacji, który informuje środowisko zewnętrzne o wyniku działania programu. Przypomnę, że wedle konwencji wartość zerowa oznacza poprawne wykonanie, a każda inna - błąd.

Kod wyjścia powinna zwrócić funkcja WinMain(), wobec tego jej końcówka musi wyglądać tak:

```
MSG msgKomunikat;

// tutaj pętla komunikatów zajmuje się pompowaniem zdarzeń we właściwe
// im okna, dopóki Peek/GetMessage() nie odbierze WM_QUIT i nie przerwie
// to pętli; wtedy ten komunikat zostaje w strukturze msgKomunikat

// zwracamy kod wyjścia zawarty w wParam komunikatu WM_QUIT
return static_cast<int>(msgKomunikat.wParam);
```

A skąd w ogóle WM\_QUIT bierze ten kod?... Otóż jest to parametr funkcji PostQuitMessage(), wywoływanej w momencie niszczenia (WM\_DESTROY) głównego okna aplikacji.

## Zmiana pozycji i rozmiaru okna

Przy okazji przesuwania i skalowania - niezależnie od tego, czy jego przyczyną jest użytkownik, czy sam program - okno otrzymuje szereg komunikatów. Dzielą się one na dwie grupy: jedne są bowiem otrzymywane tuż przed zmianą pozycji lub rozmiaru (albo w jej trakcie), a drugie już po jej dokonaniu.

### Przed faktem

Kiedy użytkownik przesuwa okno, przeciągając je za pasek tytułu, przedmiot tej zabawy otrzymuje komunikat WM\_MOVING. Zdarzenie to przynosi ze sobą jeden parametr, zapisany w lParam:

```
prcWindow = reinterpret_cast<LPRECT>(lParam);
```

Jest nim wskaźnik do struktury RECT, definiującej aktualną prostokątną obwiednię okna. Zmieniając jej pola, program może wpływać na pozycję przesuwanego okna.

Jednym z celów takiej zmiany może być „przyklejanie” okna do krawędzie jego okna nadrzędnego, na przykład pulpitu. Tak zachowuje się choćby okno odtwarzacza Winamp.

Jeżeli przetwarzamy ten komunikat, powinniśmy zwrócić do systemu wartość `TRUE`.

Podobnym komunikatem jest `WM_SIZING`, wysyłany podczas zmiany rozmiaru okna przeciągnięciem za jego krawędź. Przyjmuje on już dwa parametry:

```
dwEdge = wParam;
pRect = reinterpret_cast<LPRECT>(lParam);
```

`pRect` znaczy tu to samo, co w `WM_MOVING`, tj. określa prostokąt zamykający okno (czyli jego pozycję i wymiary). Z `dwEdge` możemy się natomiast dowiedzieć, którą część obrzeża okna użytkownik przeciąga. Parametr ten przyjmuje jedną z ustalonych stałych:

<i>stała</i>	<i>obrzeże</i>
<code>WMSZ_LEFT</code>	lewa krawędź
<code>WMSZ_TOP</code>	górną krawędź
<code>WMSZ_RIGHT</code>	prawa krawędź
<code>WMSZ_BOTTOM</code>	dolną krawędź
<code>WMSZ_TOPLEFT</code>	lewy górny róg
<code>WMSZ_TOPRIGHT</code>	prawy górny róg
<code>WMSZ_BOTTOMLEFT</code>	lewy dolny róg
<code>WMSZ_BOTTOMRIGHT</code>	prawy dolny róg

**Tabela 39.** Stałe parametru `dwEdge` (`wParam`) komunikatu `WM_SIZING`

Gdy zajmujemy się niniejszym komunikatem, powinniśmy (podobnie jak w `WM_MOVING`) zwrócić wartość `TRUE`.

Ostatnim komunikatem z omawianego rodzaju jest `WM_WINDOWPOSCHANGING`. Różni się on od dwóch pozostałych przyczyną swojego wystąpienia. Otrzymanie tego komunikatu nie jest bowiem skutkiem działań użytkownika, lecz samego programu: okno dostaje go, gdy jego pozycja i/lub rozmiar i/lub miejsce w porządku Z mają za chwilę zostać za pomocą funkcji w rodzaju `SetWindowPos()` czy `MoveWindow()`.

Razem z tym komunikatem otrzymujemy też pewne pomocne informacje:

```
pWindowPos = reinterpret_cast<WINDOWPOS*>(lParam);
```

Są one zawarte w strukturze `WINDOWPOS`, na którą wskaźnik dostajemy:

```
struct WINDOWPOS
{
    HWND hwnd;
    HWND hwndInsertAfter;
    int x;
    int y;
    int cx;
    int cy;
    UINT flags;
};
```

Nietrudno chyba zauważyć, że pola tej struktury odpowiadają dokładnie parametrom funkcji `SetWindowPos()`. Jeżeli więc chcesz poznać ich znaczenie, zajrzyj do tabelki z parametrami wspomnianej funkcji.

Kiedy zaś uporasz się z tym komunikatem, powinieneś oddać do systemu samo zero :)

Na temat komunikat [WM\\_MOVING](#), [WM\\_SIZING](#) oraz [WM\\_WINDOWPOSCHANGING](#) szeroko rozpisuje się też MSDN.

## Po fakcie

Po zakończonej operacji przesuwania i/lub skalowania okna otrzymuje ono kolejny komunikat. Tego rodzaju zdarzenia także występuje w liczbie trzech i tworzą pary z tymi zaprezentowanymi w poprzednim paragrafie.

I tak `WM_MOVE` jest odpowiednikiem `WM_MOVING`. Okno otrzymuje ten komunikat, gdy użytkownik zakończy już swoją zabawę z jego przesuwaniem. W zestawie okno dostaje również swe nowe współrzędne (względem obszaru klienta okna nadrzędnego):

```
nX = static_cast<short>(LOWORD(lParam));
nY = static_cast<short>(HIWORD(lParam));
```

Jak widać, są one zapisane w dwóch słowach parametru `lParam`, a do ich wydobycia możemy posłużyć się poznanymi makrami `LOWORD()` i `HIWORD()`.

Na koniec pracy z tym komunikatem powinniśmy zwrócić `0`.

Z kolei zdarzenie `WM_SIZE` jest związane z `WM_SIZING` i otrzymywane, kiedy użytkownik przestanie ciągnąć za krawędź lub dokonana maksymalizacji tudzież minimalizacji okna. Docelowe okno otrzymuje przy okazji także trzy dane:

```
dwResizingType = wParam;
wWidth = LOWORD(lParam);
wHeight = HIWORD(lParam);
```

`dwResizingType` określa, nazwijmy to, typ skalowania. W związku z tym przyjmuje ona jedną z kilku wyliczeniowych stałych, których część prezentuje poniższa tabelka:

<i>stała</i>	<i>znaczenie</i>
<code>SIZE_MAXIMIZED</code>	okno zostało zmaksymalizowane
<code>SIZE_MINIMIZED</code>	minimalizacja okna
<code>SIZE_RESTORED</code>	zwykła zmiana rozmiaru

**Tabela 40.** Niektóre stałe parametru `dwResizingType` (`wParam`) komunikatu `WM_SIZE`

`wWidth` i `wHeight` to, jak nietrudno się domyślić, nowe wymiary okna. Zostały one zapisane w 16-bitowych połówkach parametru `lParam`.

Po przetworzeniu komunikatu `WM_SIZE` należy zwrócić do systemu wartość zero.

Ostatnim komunikatem z tej grupy jest `WM_WINDOWPOSCHANGED`. Można wydedukować, że jest on wysłany po zmianie pozycji okna dokonanej przy pomocy `SetWindowPos()` lub innej funkcji tego rodzaju.

Ten komunikat przynosi dokładnie te same dane dodatkowe, co `WM_WINDOWPOSCHANGING`. W parametrze `lParam` można więc znaleźć wskaźnik na strukturę `WINDOWPOS`, reprezentującą wykonaną zmianę położenia i/lub rozmiaru okna; `wParam` pozostaje niewykorzystany.

Również tak samo jak poprzednio, przetworzywszy ten komunikat należy oddać systemowi liczbę zero.

O komunikatach [WM\\_MOVE](#), [WM\\_SIZE](#) i [WM\\_WINDOWPOSCHANGED](#) możesz poczytać dokładniej w MSDN.

## `WM_PAINT`

Ostatni z omawianych tutaj komunikatów jest na tyle ważny, że poświęcimy mu osobny paragraf.

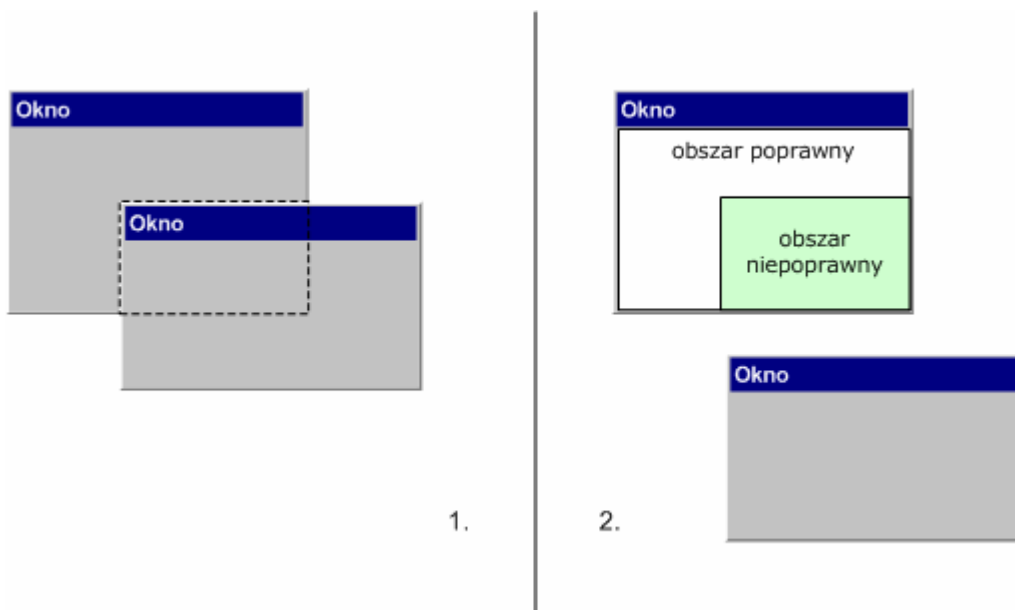
## O odrysowywaniu

Porządek Z, przesuwanie, skalowanie, minimalizacja i maksymalizacja sprawiają, że okna w Windows są często ukrywane lub przykrywane przez inne okna. Jednocześnie muszą one zachować swoją graficzną zawartość - tak, by użytkownik wiedział, czego może się po nich spodziewać.

Słyszając pierwszy raz o tym problemie i widząc, jak system radzi sobie z nim w praktyce, można łatwo uznać, iż czyni to poprzez zapisywanie obrazu okna w postaci bitmapy. Z tejsze bitmapy Windows miałby w odpowiednim czasie wybierać odpowiednie fragmenty i wypełniać nimi dopiero co odsłonięte połacie okna.

Jednak prawie nigdy system nie postępuje w ten sposób. Przechowywanie dodatkowej kopii zawartości każdego okna pochłaniałoby bowiem mnóstwo cennej pamięci operacyjnej, a jej przywracanie wymagałoby kopiowania dużych ilości danych. Dlatego też system domyślnie stosuje zupełnie inny sposób<sup>119</sup>.

Mianowicie dla każdego okna przechowuje on, zamiast pamięciożernej bitmapy, informacje o tym, które fragmenty jego obszaru klienta są **poprawne** (ang. *valid*), a które **niepoprawne** (ang. *invalid*) - w sensie konieczności ich odrysowania. Tak więc rejony pierwszego rodzaju są wyświetlane na ekranie we właściwy sposób i nie wymagają ponownego narysowania. Natomiast fragmenty niepoprawne zostały dopiero co odsłonięte użytkownikowi i muszą być ponownie wyrysowane, aby okno wyglądało prawidłowo.



Rysunek 10. Mechanizm odrysowywania zawartości okien

Jak to się dzieje? Otóż w momencie, gdy zostaje odsłonięty nowy fragment okna, wymagający ponownego narysowania (ang. *update region*), system Windows wysyła do tego okna (między innymi) komunikat `WM_PAINT`. W reakcji na niego powinno zostać wykonane żądane odrysowanie.

Graficzna zawartość okna jest zawsze przywracana w ten właśnie sposób.

<sup>119</sup> Można aczkolwiek zmusić go do opisanego wyżej zachowania w stosunku do tych okien, których klasy zarejestrowano z podaniem stylu `CS_SAVEBITS`.

## Reakcja na WM\_PAINT

Pewnie zauważyłeś, że nie we wszystkich dotychczasowych programach przykładowych zajmowaliśmy się przetwarzaniem tego komunikatu. Mimo to każde stworzone przez nas okno prawidłowo odrysowywało swój obszar klienta w razie potrzeby.

Działo się tak, ponieważ w domyślnej reakcji na WM\_PAINT system Windows wypełnia niepoprawny obszar (dokładniej: prostokąt) okna odpowiednim pędzlem. Tak jest, tym samym pędzlem, który podaliśmy podczas rejestracji klasy okna. Dzięki temu nie zawsze musimy zajmować się zdarzeniem WM\_PAINT.

Trzeba to aczkolwiek robić, jeżeli wypełnienie pędzlem nam nie wystarcza. Wówczas należy zareagować na ten komunikat, na przykład w ten sposób:

```
case WM_PAINT:
{
    // wypisanie tekstu

    PAINTSTRUCT ps;
    HDC hdcOkno;

    // rozpoczęcie rysowania (wypełnienie uaktualnianego obszaru pędzlem)
    hdcOkno = BeginPaint(hWnd, &ps);

    // wypisanie tekstu
    std::string strTekst = "123 - próba tekstu";
    TextOut (hdcOkno, ps.rcPaint.left, ps.rcPaint.top, strTekst.c_str(),
            strTekst.length());

    // zakończenie rysowania
    EndPaint (hWnd, &ps);
    return 0;
}
```

Może się wydawać to zaskakujące, ale komunikat WM\_PAINT nie przynosi żadnych informacji w swoich parametrach wParam i lParam. Zamiast tego, o regionie okna, który ma być odświeżony, należy dowiedzieć się w inny sposób.

Robimy to, wywołując funkcję BeginPaint(). Podajemy jej przy tym uchwyt do odmalowywanego okna oraz wskaźnik na specjalną strukturę PAINTSTRUCT:

```
struct PAINTSTRUCT
{
    HDC hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
};
```

Zawiera ona wiadomości o fragmencie okna, który powinien zostać ponownie narysowany. Poszczególne pola tej struktury omawia poniższa tabela:

typy	pola	opis
HDC	hdc	Jest to <b>uchwyt do konkretnego urządzenia</b> okna. Pamiętajmy być może, że jest to specjalny rodzaj uchwytu, służący do rysowania po jakiejś powierzchni przy pomocy funkcji Windows GDI. W tym przypadku ową powierzchnią jest obszar klienta okna.

<i>typy</i>	<i>pola</i>	<i>opis</i>
		Wartość tego pola jest też zwraca przez funkcję <code>BeginPaint()</code> .
BOOL	<code>fErase</code>	Jest to flaga boolowska określająca, czy tło uaktualnianego obszaru okna ma zostać wymazane. Zazwyczaj odpowiada za to funkcja <code>BeginPaint()</code> , wypełniając wspomniany obszar pędzlem okna. Jeżeli jednak podczas rejestrowania klasy nie ustawiliśmy żadnego takiego pędzla (pole <code>hbrBackground</code> struktury <code>WNDCLASSEX</code> miało wartość <code>NULL</code> ), wówczas musimy sami sprawdzić stan tego pola i w razie potrzeby wyczyścić odrysowywany prostokąt. Ponieważ jednak w większości przypadków wybieramy dla okna jakiś pędzel, nie musimy się tym polem przejmować.
RECT	<code>rcPaint</code>	To chyba najważniejsze pole: definiuje ono <b>prostokąt okna</b> , który ma być odrysowany. Podane tu współrzędne są relatywne do lewego górnego rogu obszaru klienta okna, dlatego mogą być użyte w funkcjach rysujących razem z kontekstem <code>hdc</code> . Używanie współrzędnych tego prostokąta jest oczywiście możliwe, jednak w praktyce wygodniej jest za każdym razem odrysować całe okno - szczególnie, jeżeli wymyślanie jakichś skomplikowanych algorytmów fragmentarycznego rysowania miałyby nam zająć zbyt dużo czasu.
BOOL BOOL BYTE [32]	<code>fRestore</code> <code>fIncUpdate</code> <code>rgbReserved</code>	Te trzy pola są zarezerwowane do wewnętrznego użytku systemu Windows, zatem nie powinny nas one w ogóle interesować :)

Tabela 41. Pola struktury `PAINTSTRUCT`

Kontekst urządzenia, zapisany w polu `hdc` oraz zwracany przez funkcję `BeginPaint()`, możemy wykorzystać do rysowania po powierzchni obszaru klienta za pomocą przeróżnych funkcji Windows GDI. Większość z nich poznamy w osobnym rozdziale, poświęconym w całości tej bibliotece; na razie mieliśmy okazję spotkać się dwoma, służącymi do wypisywania tekstu. Była to `DrawText()`, użyta w przykładowym programie `TaskbarHider`, oraz `TextOut()`:

```
std::string strTekst = "123 - próba tekstu";
TextOut (hdcOkno, ps.rcPaint.left, ps.rcPaint.top, strTekst.c_str(),
        strTekst.length());
```

Myślę, że nawet nie mając prototypu ani opisu, potrafiłbyś domyślić się jej działania oraz znaczenia parametrów. Wyjaśnimy je sobie dogłębnie, jak już mówiłem, w rozdziale o Windows GDI. Na razie łatwo wywnioskować, że parametrami `TextOut()` są kolejno:

- uchwyt do kontekstu urządzenia, reprezentujący powierzchnię, na której będziemy pisać
- pozioma i pionowa współrzędna tekstu
- sam tekst w postaci łańcucha znaków w stylu C
- liczba znaków w wypisywanym tekście

Wywołanie `TextOut()` jest jedyną czynnością stricte graficzną, jaką wykonujemy na rysowanym rejonie okna. Po jej zakończeniu finalizujemy zatem proces odświeżania za pomocą funkcji `EndPaint()`. Przekazujemy jej te same dwa parametry, jakie podaliśmy do `BeginPaint()`.



**Każde wywołanie** `BeginPaint()` musi być rekompensowane przez analogiczne wykonanie `EndPaint()`. Poza tym obie funkcje powinny być przywoływane **wyłącznie** w kodzie obsługi komunikatu `WM_PAINT`.

Typowy, poprawny schemat obsługi `WM_PAINT` wygląda więc następująco:

```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdcOkno;

    hdcOkno = BeginPaint(hWnd, &ps);

    odrysowywanie_wskazanego_obszaru_okna

    EndPaint(hWnd, &ps);
    return 0;
}
```

Kończące go zwrócenie zera jest również wymogiem systemowym.

### *Wymuszanie odrysowywania okna*

O tym, kiedy dokonać odrysowania zawartości okna decyduje w dużej mierze sam system operacyjny Windows. Nierzadko jednak konieczne jest ręczne wywołanie tego procesu; przykład można obserwować w programie `TaskbarHider`, gdzie kliknięcie lewego przycisku myszy (zdarzenie `WM_LBUTTONDOWN`) musiało spowodować odświeżenie okna.

Wydawałoby się, że nie ma w tym nic trudniejszego ponad wysłanie komunikatu `WM_PAINT` przy pomocy jednej z funkcji `SendMessage()` lub `PostMessage()`, służących przesyłaniu komunikatów:

```
SendMessage(hWnd, WM_PAINT, NULL, NULL);
```

Windows API przewiduje nawet odrębną funkcję `UpdateWindow()`, której wywołanie jest równoważne instrukcji powyżej.

Takie działanie nie daje jednak pożądaných rezultatów i daje się to w prosty sposób wyjaśnić. Okno otrzymuje oczywiście komunikat `WM_PAINT`, ale system Windows uznaje, iż cały obszar klienta okna jest **poprawny**, więc nie ma najmniejszej potrzeby jego odrysowania. W takiej sytuacji funkcja `BeginPaint()` zawodzi, podobnie jak wszystkie następne z obsługi `WM_PAINT`, i nie obserwujemy żadnej zmiany zawartości okna. Wynika stąd, że należałoby w jakiś sposób oszukać system i przekonać go, że oto cały obszar klienta okna jest niepoprawny i pilnie wymaga odrysowania. Jest to jak najbardziej możliwe przy pomocy funkcji `InvalidateRect()`:

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

W zasadzie funkcja ta służy do oznaczenia pewnego określonego prostokąta (podanego w parametrze `lpRect`) jako przeznaczonego do odświeżenia. Możliwe jest aczkolwiek podanie jej całego obszaru klienta okna `hWnd` - wówczas trzeba po prostu wpisać `NULL` jako drugi parametr funkcji. Trzeci parametr określa natomiast konieczność zamazania dostarczonego prostokąta pędzlem tła; jeżeli podamy w nim `TRUE`, wtedy prostokąt ów

zostanie wyczyszczony przez funkcję `BeginPaint()`; w przeciwnym wypadku pozostanie on bez zmian.

Najczęściej zależy nam wszelako na całkowitym wyczyszczeniu całego obszaru klienta danego okna. Można to uczynić prostym wywołaniem:

```
InvalidateRect (hWnd, NULL, TRUE);
```

Co więcej, spowoduje ona także natychmiastowe wysłanie `WM_PAINT` do rzeczonoego okna, zatem nie musimy się już tym kłopotać.

Do swojej wiadomości warto więc zapamiętać, że:

**Ponowne narysowanie zawartości całego obszaru klienta** okna `hWnd` można wymusić poprzez wywołanie `InvalidateRect(hWnd, NULL, TRUE);`.

\*\*\*

Tą użyteczną uwagą kończymy podrozdział poświęcony systemowi zdarzeń w Windows. Przypatrzyliśmy się w nim dokładnie pętli komunikatów oraz najważniejszym rodzajom zdarzeń, jakie otrzymują okna. Szczególnie dużo czasu poświęciliśmy na czynność odrysowywania zawartości okna, związaną z komunikatem `WM_PAINT`. Wszystko to nie jest może bardzo proste, ale mam nadzieję, że zrozumiałeś z tego przynajmniej „większą połowę” ;)

W tym momencie zakończyliśmy też przegląd podstawowych zagadnień związanych z oknami w systemie Windows.

## Podsumowanie

*Windows* nieprzypadkowo znaczy 'okna'. Jako elementy interfejsu użytkownika są one bowiem nieodzownym składnikiem każdego programu.

W kończącym się rozdziale skoncentrowaliśmy się jedynie na takich oknach, które są oknami także w potocznym rozumieniu użytkownika. Najpierw zajęliśmy się więc dwuetapowym procesem ich tworzenia, obejmującym rejestrację klasy okna i wywołanie funkcji `CreateWindowEx()`. Dalej pokazałem kilka typowych operacji, jakie można wykonywać na już istniejących oknach oraz sposobach na uzyskiwanie ich uchwytów. Na koniec zajrzeliśmy włąb pętli komunikatów i poznaliśmy najważniejsze zdarzenia dotyczące samych okien.


Następny rozdział będzie z kolei poświęcony współpracy naszych aplikacji z dwoma najważniejszymi urządzeniami wejściowymi: klawiaturą i myszką. Wreszcie zatem pisane przez nas programy nabiorą nieco większej interaktywności.

## Pytania i zadania

Zgodnie ze zwyczajem raczę cię na koniec odpowiednim zestawem pytań i ćwiczeń do samodzielnego wykonania.

### Pytania

1. Z jakich elementów składa się potocznie rozumiane okno w systemie Windows?
2. Co to jest obszar klienta okna?
3. W jakim celu wprowadzono w systemie Windows mechanizm klas okien?
4. Jakie informacje należy podać, rejestrując klasę okna?

5. W jaki sposób możemy wczytać ikonę lub kursor z zewnętrznego pliku graficznego?
6. Jak możemy uzyskać uchwyt do pędzla wypełniającego tło okna?
7. Jakie informacje podajemy przy tworzeniu okna należącego do zarejestrowanej już klasy?
8. Co określa styl okna?
9. Jak można dopasować rozmiary okna do znanych rozmiarów jego obszaru klienta?
10. Przy pomocy jakich funkcji pokazujemy i ukrywamy okno?
11. Jak możemy zmienić pozycję i/lub rozmiary okna?
12. Czym jest porządek Z, zwany też kolejnością przesłaniania?
13. Jak zbudowana jest hierarchia okien w systemie Windows? Przy pomocy jakich funkcji możemy się po niej poruszać?
14. Podaj metodę na uzyskanie uchwytu dowolnego okna znanej klasy.
15. Dlaczego funkcja `PeekMessage()` lepiej sprawdza się w pętli komunikatów niż `GetMessage()`?
16. W jaki sposób możemy zapewnić wykonywanie się kodu w czasie pomiędzy obsługą komunikatów o zdarzeniach?
17. **(Trudniejsze)** Jakie komunikaty otrzymuje kolejno główne okno aplikacji po kliknięciu w przycisk  (przy założeniu, że owo kliknięcie spowoduje poprawne zakończenie programu)?
18. Jak wygląda prawidłowa obsługa komunikatu `WM_PAINT` (jeżeli jest konieczna)?

## Ćwiczenia

1. Wypróbuj działania styli okna, szczególnie tych dotyczących paska tytułu.
2. **(Trudne)** Napisz program wyświetlający twoje imię na pulpicie.