

1

APLIKACJE OKIENKOWE

*Wyobraź sobie, że gdy w każdy czwartek
zwyczajnie zawiązujesz sobie buty, one eksplodują.
Coś takiego cały czas dzieje się z komputerami
i jakoś nikt na to nie narzeka.*

Jeff Raskin, wywiad dla „Doctor Dobb’s Journal”

Pierwsze komputery osobiste powstały już całkiem dawno temu, bo przy końcu lat siedemdziesiątych ubiegłego stulecia. Prawie od samego początku mogły też wykonywać całkiem współczesne operacje - z edycją dokumentów czy wykorzystaniem sieci włącznie. A jednak dopiero ostatnia dekada przyczyniła się do niezmiernego upowszechnienia pecetów, a umiejętność ich obsługi stała się powszechna i konieczna. Przyczyn można upatrywać się w szybkim rozwoju Internetu, jednak trudno sobie wyobrazić jego ekspansję oraz rozpowszechnienie samych komputerów, gdyby ich użytkowanie nie było proste i intuicyjne. Bez łatwych metod komunikacji z programami początkujący użytkownik byłby bowiem zawsze w trudnej sytuacji.

Po licznych „bojach” stoczonych z konsolą możemy z pewnością stwierdzić, że interfejs tekstowy niekiedy bywa wygodny. Faktycznie oferuje go każdy system operacyjny, a za jego pomocą często można szybciej i efektywniej wykonywać rutynowe zadania - szczególnie, kiedy mamy już pewne doświadczenie w obsłudze danego systemu. Nie da się jednak ukryć, że całkowity nowicjusz, posadzony przez ekranem z migającym tajemniczo kursorem, może poczuć się, delikatnie mówiąc, lekko zdezorientowany. Naturalnie mógłby on zajrzeć do stosownych dokumentacji czy też innych źródeł niezbędnych wiadomości, lecz procent użytkowników, którzy rzeczywiście tak czynią, oscyluje chyba gdzieś w granicach błędu statystycznego (jeśli ktokolwiek przeprowadzał kiedykolwiek takie badania) :D Czy to jest jednak tylko ich problem?...

Otóż nie, a właściwie - **już** nie. Oto bowiem w latach osiemdziesiątych wymyślono nowe sposoby dialogu aplikacji z użytkownikiem, z których najlepszy (dla użytkownika) okazał się **interfejs graficzny**. Prawdopodobnie zdecydowały tu proste analogie w stosunku do znanych urządzeń, które regulowało się najczęściej przy pomocy różnych przycisków, pokręteł, suwaków czy włączników. Koncepcje te dały się łatwo przenieść w świat wirtualny i znacznie rozszerzyć, dając w efekcie obecny wygląd interfejsu użytkownika w większości popularnych programów.

Graficzny interfejs użytkownika (ang. *graphical user interface* - w skrócie GUI) to sposób wymiany informacji między programem a użytkownikiem, oparty na wyświetlaniu **interaktywnej grafiki** i **reakcji** na działania, jakie są podejmowane w stosunku do niej.

Istnieje wiele powodów, dla których ten rodzaj interfejsu jest generalnie łatwiejszy w obsłudze niż rozwiązania oparte na tekście. Nietrudno znaleźć te powody, gdy porównamy jakąś aplikację konsolową i program wykorzystujący interfejs graficzny. Warto jednak wymienić te przyczyny - najlepiej w kolejności rosnącego znaczenia:

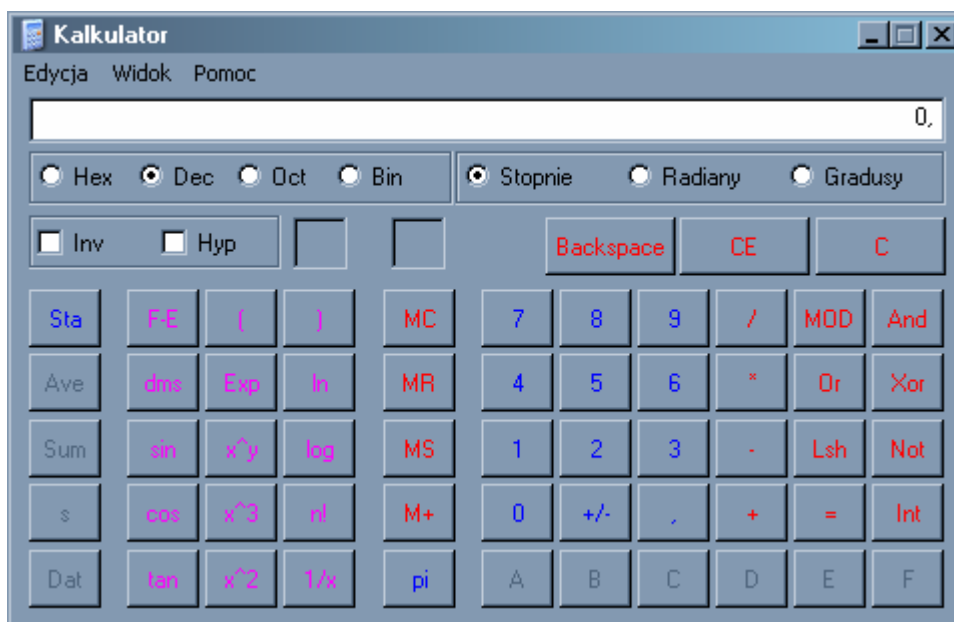
- program konsolowy jest często dla użytkownika „czarną skrzynką” (żeby nie powiedzieć - czarną magią ;D). O jego przeznaczeniu czy oferowanych przezeń funkcjach rzadko może się bowiem dowiedzieć z informacji prezentowanych mu na

ekranie. Są one zwykle tylko wynikami pracy programu lub też prośbami o wprowadzenie potrzebnych danych.

Tymczasem w programach o interfejsie graficznym konieczne są przynajmniej szcążkowe opisy poszczególnych opcji i funkcji, a to samo w sobie daje pewne wskazówki co do prawidłowej obsługi programu.

Nic więc dziwnego, że wielu użytkowników programów uczy się ich obsługi metodą prób i błędów, czyli kolejnego wypróbowywania oferowanych przez nie opcji i obserwacji efektów tych działań.

- elementy interfejsów graficznych są bardziej „namacalne” niż tekstowe polecenia, wpisywane z klawiatury. Łatwiej domyślić się, jak działa przycisk, suwak czy pole tekstowe - czego nie można powiedzieć o komendach konsolowych. Można więc stwierdzić, że występuje tu podobny efekt jak w przypadku programowania obiektowego. Coś, co możemy zobaczyć/wyobrazić sobie, jest po prostu łatwiejsze do przyswojenia niż abstrakcyjne koncepcje.



Screen 44. Nawet skomplikowany interfejs graficzny może być prostszy w obsłudze niż aplikacja konsolowa. Kalkulator mógłby oczywiście z powodzeniem działać w trybie tekstowym i oferować dużą funkcjonalność, np. w postaci obliczania złożonych wyrażeń. Można ją jednak zaimplementować także w aplikacji z graficznym interfejsem, zaś przyciski i inne elementy okna są z pewnością bardziej intuicyjne niż choćby lista dostępnych funkcji i operacji matematycznych.

- graficzne interfejsy użytkownika dają większą swobodę i kontrolę nad przebiegiem programu. O ile w aplikacjach konsolowych funkcjonowanie programu opiera się zazwyczaj na schemacie: *pobierz dane* → *pracuj* → *pokaż wynik*, o tyle interfejsy graficzne w większości przypadków zostawiają użytkownikowi olbrzymie pole manewru, jeżeli chodzi o podejmowane czynności i ich kolejność. Jest to całkowicie inny **model funkcjonowania programu**.

GUI jest zatem nie tylko zmianą w powierzchowności aplikacji, ale też fundamentalną różnicą, jeżeli chodzi o jej działanie - zarówno od strony użytkownika, jak i programisty. Tworzenie programów z interaktywnym interfejsem przebiega więc inaczej niż kodowanie aplikacji konsolowych, działających sekwencyjnie.

Ta druga czynność jest nam już doskonale znana, teraz więc przysłała pora na poznanie metod tworzenia aplikacji działających w środowiskach graficznych i wykorzystujących elementy GUI.

Posiadanie graficznego interfejsu nie oznacza aczkolwiek, że dany program jest w pełni interaktywny. Wiele z aplikacji niewątpliwie graficznych, np. kreatory instalacji, są w rzeczywistości programami sekwencyjnymi. Jednocześnie możliwe jest osiągnięcie interaktywności w środowisku tekstowym, czego najlepszym przykładem jest chyba popularny w czasach DOSa menedżer plików Norton Commander. Obecnie jednak aplikacje wyposaża się w graficzny interfejs właśnie po to, aby ich obsługa była interaktywna i możliwa na dowolne sposoby. Na tym opierają się dzisiejsze systemy operacyjne.

Zajmiemy się programowaniem **aplikacji okienkowych** przeznaczonych dla środowiska Windows. Pamiętajmy oczywiście, że naszym nadrzędnym celem jest poznanie technik programowania gier; znajomość podstaw tworzenia programów GUI jest jednak niezbędna, by móc korzystać z biblioteki graficznej DirectX, która przecież działa w graficznym środowisku Windows. Umiejętność posługiwania się narzędziami, jakie ten system oferuje, powinna też zapoczątkować w bliższej lub dalszej przyszłości i z pewnością okaże się pomocna.

A zatem - zaczynamy programowanie w Windows!

Wprowadzenie do programowania Windows

Pisanie programów działających w podsystemie GUI w Windows różni się zasadniczo od tworzenia aplikacji konsolowych. Wynika to nie tylko z nowych narzędzi programistycznych, jakie należy do tego wykorzystać, ale także, czy może przede wszystkim, z innego modelu funkcjonowania programów okienkowych. Wymaga to nieco innego podejścia do kodowania, myślę jednak, że jest ono nawet łatwiejsze i bardziej sensowne niż dla konsoli.

Na początek naszej przygody z programowaniem Windows poznamy więc ów nowy model działania aplikacji. Później zobaczymy również, jakie instrumenty wspomagające kodowanie oferuje ten system operacyjny.

Programowanie sterowane zdarzeniami

Elastyczność, jaką wykazują programy z interfejsem graficznym, w zakresie kontroli ich działania przez użytkownika jest niezwykle duża. Można w zasadzie stwierdzić, że dopiero takie aplikacje stają się przydatnymi narzędziami, posłusznymi swoim użytkownikom. Nie narzucają żadnych ścisłych wymogów co do sposobu obsługi, pozostawiając duże pole swobody i ergonomii.

Osiągnięcie takich efektów przy pomocy znanych nam technik programowania byłoby bardzo trudne, a na pewno naciągane - jeżeli nie niemożliwe. Graficzny interfejs aplikacji okienkowych wymaga bowiem zupełnie nowego sposobu kodowania: **programowania sterowanego zdarzeniami**.

Modele działania programów

Aby dokładnie zrozumieć tę ideę i móc niedługo stosować ją w praktyce, potrzebne są rzecz jasna stosowne wyjaśnienia. Przede wszystkim chciałoby się wiedzieć, na czym polegają różnice w tym sposobie programowania, gdy przyrównamy go do znanego dotychczas sekwencyjnego uruchamiania kodu. Nie od rzeczy byłoby także wskazanie zalet nowego modelu działania aplikacji. To właśnie uczynimy teraz.

Najpierw należałoby więc sprecyzować, co rozumiemy pod pojęciem modelu działania programu, gdyż stosowaliśmy ten termin już kilkakrotnie i najwyraźniej wydaje się on tu kluczowy. Mianowicie możemy powiedzieć krótko:

Model funkcjonowania aplikacji (ang. *application behavior model*) to, najogólniej mówiąc, pozycja, jaką zajmuje program w stosunku do użytkownika oraz do systemu operacyjnego. Określa on sposób, w jaki kod programu steruje jego działaniem, głównie wprowadzaniem danych wejściowych i wyprowadzaniem wyjściowych.

Wyjaśnienie to może wydawać się dosyć mgliste, ponieważ pojęcie modelu funkcjonowania aplikacji jest jedną z najbardziej fundamentalnych spraw w projektowaniu, kodowaniu, jak również w użytkowaniu wszystkich bez wyjątku programów. Jednocześnie trudno je rozpatrywać całkiem ogólnie, tak jak tutaj, i dlatego zwykle się tego nie robi; niemniej jednak jest to bardzo ważny aspekt programowania.

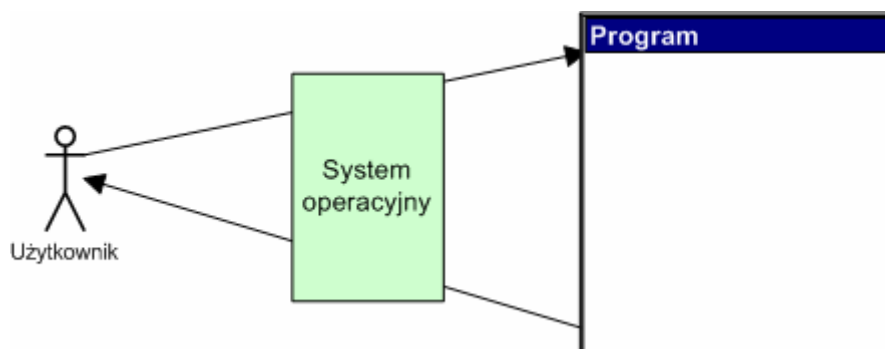
Najczęściej wszakże mówi się jedynie o odmianach modelu działania aplikacji, a zatem także i my je poznamy. Nie są one zresztą całkiem dla nas obce, a nawet nowopoznane koncepcje wydadzą się, jak sądzę, dosyć logiczne i rozsądne. Przyjrzyjmy się więc poszczególnym modelom.

Model sekwencyjny

Najstarszym i najwcześniej przez nas spotkanym w nauce programowania modelem jest **model sekwencyjny**. Był to w początkach rozwoju komputerów najbardziej oczywisty sposób, w jaki mogły działać ówczesne programy.

Ogólnym założeniem tego modelu jest ustawienie programu w pozycji **dialogu** z użytkownikiem. Taki dialog nie jest oczywiście normalną konwersacją, jako że komputery nigdy nie były, nie są i nie będą ani trochę inteligentne. Dlatego też przyjmuje ona formę **wywiadu**, któremu poddawany jest użytkownik.

Aplikacja zatem „zadaje pytania” i oczekuje na nie odpowiedzi w postaci potrzebnych sobie danych. Prezentuje też wyniki swojej pracy do wglądu użytkownika.

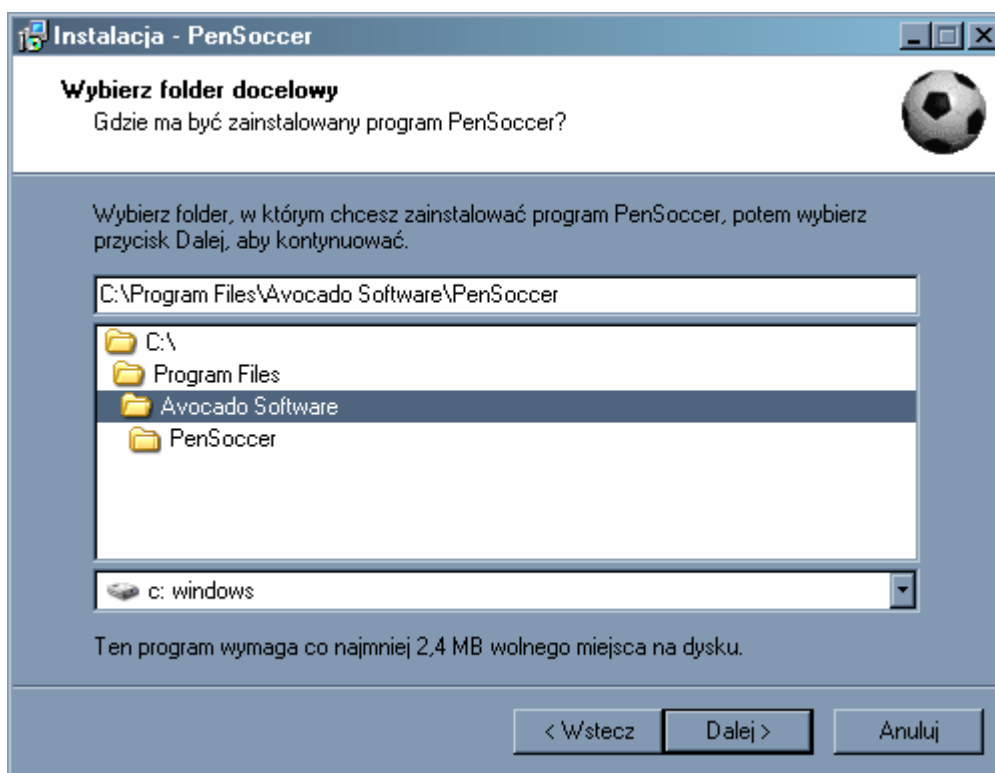


Schemat 36. Sekwencyjny model działania programu. Aplikacja zajmuje tu miejsce nadrzędne w stosunku do użytkownika (stąd jej wielkość na diagramie :D), a system operacyjny jest pośrednikiem w wymianie informacji (zaobrazowanej strzałkami).

Najważniejsze, że cały przebieg pracy programu jest kontrolowany przez programistę. To on ustala, kiedy należy odczytać dane z klawiatury, wyświetlić coś na ekranie czy wykonać inne akcje. Wszystko ma tu swój określony porządek i kolejność, na którą **użytkownik nie ma wpływu**. Właśnie ze względu na ową kolejność model ten nazywamy sekwencyjnym.

Najlepszym przykładem aplikacji, które działają w ten sposób, będą wszystkie napisane dotychczas w tym kursie programy konsolowe; w ogólności tyczy się to w zasadzie każdego programu konsolowego. Sama natura tego środowiska wymusza pobieranie oraz pokazywanie informacji w pewnej kolejności, niepozwalającej użytkownikowi na większą swobodę.

Do tej grupy możemy też zaliczyć bliższe nam aplikacje, funkcjonujące jako **kreatory** (ang. *wizards*), z kreatorami instalacji na czele. Posiadają one wprawdzie interfejs graficzny, ale sposób i porządek ich działania jest ściśle ustalony. Obrazują to nawet kolejne kroki - ekrany, które po kolei pokonuje użytkownik, podając dane i obserwując efekty swoich działań.



Screen 45. Kreatory instalacji (ang. *setup wizards*) są przykładami programów działających sekwencyjnie. Zawierają wprawdzie elementy interfejsu graficznego właściwe innym aplikacjom, ale ich działanie jest precyzyjnie ustalone i podzielone na kroki, które należy pokonywać w określonej kolejności.

Poza wspomnianymi kreatorami (które zazwyczaj są tylko częścią większych aplikacji), programy działające sekwencyjnie nie występują zbyt licznie i nie mają poważniejszych zastosowań. Ich niewrażliwość na intencje użytkownika i zatwardziałe trzymanie się ustalonych schematów funkcjonowania sprawiają, że nie można przy ich pomocy swobodnie wykonywać swoich zajęć.

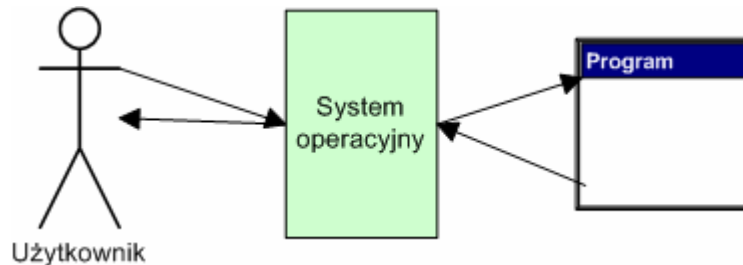
Model zdarzeniowy

Zupełnie inne podejście jest prezentowane w **modelu zdarzeniowym**, zwanym też **programowaniem sterowanym zdarzeniami** (ang. *event-driven programming*). Model ten opiera się na całkiem odmiennych zasadach niż model sekwencyjny, oferując dzięki nim nieporównywalnie większą elastyczność działania.

Podstawową wytyczną jest tu zmiana roli programu. Nie jest on już ciągiem kolejno podejmowanych kroków, które składają się na wykonywaną przezeń czynność, ale raczej czymś w rodzaju witryny sklepowej, z której użytkownik może wybierać pożądane w danej chwili funkcje.

Dlatego też działanie programu polega na odpowiedniej reakcji na występujące **zdarzenia** (ang. *events*). Tymi zdarzeniami mogą być na przykład wciśnięcia klawiszy, ruch myszy, zmiana rozmiaru okna, uzyskanie połączenia sieciowego i jeszcze wiele innych. Program może być informowany o tych zdarzeniach i reagować na nie we właściwy sobie sposób.

Najważniejszą cechą tego modelu programowania jest jednak samo **wykrywanie zdarzeń**. Otóż leży ono całkowicie poza obowiązkami programisty. Nie musi on już organizować czekania na wciśnięcie klawisza czy też wystąpienie innego zdarzenia - wyręcza go w tym system operacyjny. Programista powinien jedynie zapewnić kod reakcji na te zdarzenia, które są ważne dla pisanej przez niego aplikacji.



Schemat 37. Zdarzeniowy model działania programu. Pozycja użytkownika jest tu znacznie ważniejsza niż w modelu sekwencyjnym, gdyż poprzez zdarzenia może on w bardzo dużym stopniu wpływać na pracę programu. Rola pośrednicząca systemu operacyjnego jest też bardziej rozbudowana.

Większość zdarzeń będzie pochodzić od użytkownika - szczególnie te związane z urządzeniami wejścia, jak klawiaturą czy myszą. Aplikacja będzie natomiast otrzymywać informacje o nich przez **cały swój czas działania**, nie zaś tylko wtedy, gdy sama o to poprosi. W reakcji na owe zdarzenia program powinien wykonywać odpowiednie dla siebie czynności i zazwyczaj to właśnie robi. Ponieważ więc zdarzenia są **wywoływane** przez użytkownika, a aplikacja musi jedynie **reagować** na nie, więc sposób jej działania jest wówczas prawie całkiem **dowolny**. To użytkownik decyduje, co program ma w danej chwili robić - a nie on sam.

W modelu zdarzeniowym działanie programu jest podporządkowane przede wszystkim **woli użytkownika**.



Screen 46 i 47. Przykłady programów wykorzystujących model zdarzeniowy. Ich użytkownicy mogą je kontrolować, wywołując takie zdarzenia jak kliknięcie przycisku lub wybór opcji z menu.

Z początku może to wydawać się niezwykle ograniczające: aby program mógł coś zrobić, musi poczekać na wystąpienie jakiegoś zdarzenia. W istocie jednak wcale nie jest to niedogodnością i można się o tym przekonać, uświadamiając sobie dwa fakty. Przede wszystkim każdy program powinien być „świadomy” warunków zewnętrznych, które mogą wpływać na jego działanie. W modelu zdarzeniowym to „uświadamianie” przebiega poprzez informacje o zachodzących zdarzeniach; bez tego aplikacja i tak

musiałyby w jakiś sposób dowiadywać się o tych zdarzeniach, by móc w ogóle poprawnie funkcjonować.

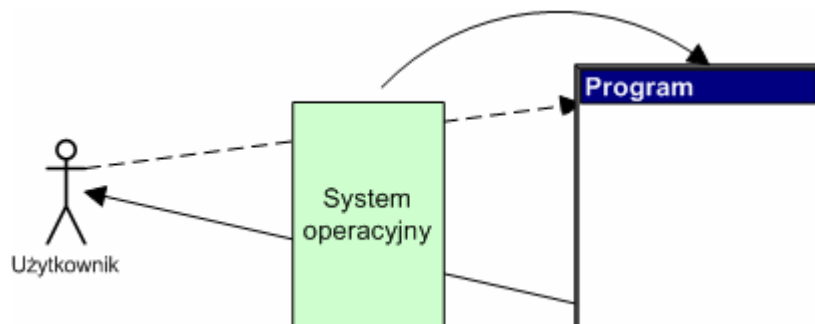
Po drugie sytuacje, w których należy robić coś niezależnie od zachodzących zdarzeń, należą do względnej rzadkości. Jeżeli nawet jest to konieczne, system operacyjny z pewnością udostępni sposoby, poprzez które można taki efekt osiągnąć.

A zatem model zdarzeniowy jest najbardziej optymalnym wariantem działania programu, z punktu widzenia zarówno użytkownika (pełna swoboda w korzystaniu z aplikacji), jak i programisty (zautomatyzowane wykrywanie zdarzeń i konieczność jedynie reakcji na nie). Nic więc dziwnego, że obecnie niemal wszystkie porządne programy funkcjonują w zgodzie z tym modelem. W kolejnych rozdziałach my także nauczymy się tworzenia aplikacji działających w ten sposób.

Model czasu rzeczywistego

Dla niektórych programów model zdarzeniowy jest jednak niewystarczający lub nieodpowiedni. Ich natura zmusza bowiem do ciągłej pracy i wykonywania kodu niezależnie od zachodzących zdarzeń. O takim programach mówimy, iż działają w **czasie rzeczywistym** (ang. *real time*).

W praktyce wiele programów wykonuje podczas swej pracy dodatkowe **zadania w tle** (ang. *background tasks*), niezależne od zachodzących zdarzeń. Przykładowo, dobre edytory tekstu często dokonują sprawdzania poprawności językowej dokumentów podczas ich edycji. Podobnie niektóre systemy operacyjne przeprowadzają defragmentację dysków w sposób ciągły, przez cały czas swego działania. O takich aplikacjach nie możemy jednak powiedzieć, że wykorzystują model czasu rzeczywistego. Jakkolwiek różnica między nim a modelem zdarzeniowym wydaje się płynna, to za programy czasu rzeczywistego można uznać wyłącznie te, dla których czynności wykonywane w sposób ciągły są **głównym** (a nie pobocznym) celem działania.

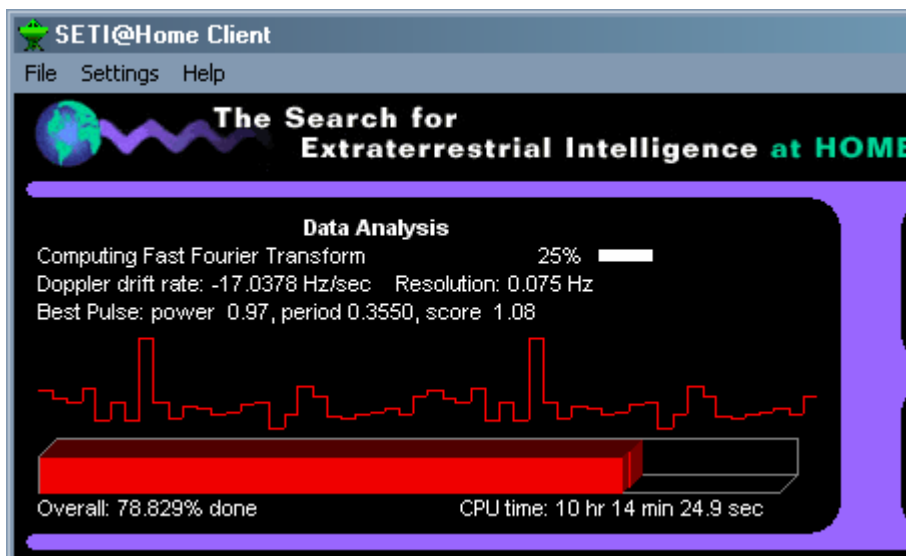


Schemat 38. Model działania programu czasu rzeczywistego. Programy tego rodzaju zwykle same dbają o pobieranie odpowiednich danych od systemu operacyjnego; mogą sobie na to pozwolić, gdyż nieprzerwanie wykonują swój kod. Natomiast ich interakcja z użytkownikiem jest zwykle dość ograniczona.

Całkiem spora liczba aplikacji działa w ten sposób, tyle że zazwyczaj trudno to zauważyć. Należą do nich bowiem wszelkie programy działające w tle: od sterowników urządzeń, po liczniki czasu połączeń internetowych, firewalle, skanery antywirusowe, menedżery pamięci operacyjnej lub aplikacje dokonujące jakichś skomplikowanych obliczeń naukowych. Swoją pracę muszą one wykonywać przez cały czas - niezależnie od tego, czy jest to monitoring zewnętrznych urządzeń, procesów systemowych czy też pracochłonne algorytmy. Koncentrują na tym prawie wszystkie swoje zasoby, choć mogą naturalnie zapewniać jakąś formę kontaktu z użytkownikiem, podobnie jak programy sterowane zdarzeniami.

Zwykle też aplikacje czasu rzeczywistego tworzy się podobnie jak programy w modelu zdarzeniowym. Uzupełnia się je tylko o pewne dodatkowe procedury, wykonywane przez

cały czas trwania programu lub też wtedy, gdy nie są odbierane żadne informacje o zdarzeniach.



Screen 48. Programy czasu rzeczywistego mogą działać w tle i wykonywać przez cały czas właściwe sobie czynności. Klient [SETI@home](http://setiathome.org) dokonuje na przykład analizy informacji zbieranych przez radioteleskopy w poszukiwaniu sygnałów od inteligentnych cywilizacji pozaziemskich.

Drugą niezwykle ważną (szczególnie dla nas) grupą aplikacji, które wykorzystują ten model funkcjonowania, są **gry**. Przez cały swój czas działania wykonują one pracę określaną w skrócie jako **generowanie klatek**, czyli obrazów, które są wyświetlane potem na ekranie komputera. Aby dawały one złudzenie ruchu, muszą zmieniać się wiele razy w ciągu sekundy, zatem na ich tworzenie powinien być przeznaczony cały dostępny grze czas i zasoby systemowe. Tak też faktycznie się dzieje, a generowanie klatek przeprowadza się nieustannie i bez przerwy.

Model czasu rzeczywistego jest więc najbardziej nas, przyszłych programistów gier, interesującym sposobem działania programów. Aby jednak tworzyć aplikacje oparte na tym modelu, trzeba dobrze poznać także programowanie sterowane zdarzeniami, jako że jest ono z nim nierozdzielnie związane. Umiejętność tworzenia aplikacji okienkowych w Windows jest bowiem pierwszym i niezbędnym wymaganiem, jakie jest stawiane przed adeptami programowania gier, działających w tym systemie operacyjnym.

Dalej więc poznamy bliżej ideę programowania sterowanego zdarzeniami i przyjrzymy się, jak jest ona realizowana w praktyce.

Zdarzenia i reakcje na nie

Aby program mógł wykonywać jakiś kod w reakcji na pewne zdarzenie, musi się o tym zdarzeniu dowiedzieć, zidentyfikować jego rodzaj oraz ewentualne dodatkowe informacje, związane z nim. Bez tego nie ma mowy o programowaniu sterowanym zdarzeniami.

W systemach operacyjnych takich jak DOS czy UNIX rozwiązywano ten problem w dość pokrętny sposób. Otóż jeżeli program nie miał działać sekwencyjnie, lecz reagować na niezależne od niego zdarzenia, to musiał nieustannie prowadzić monitorowanie ich potencjalnych źródeł. Musiał więc „nasłuchiwać” w oczekiwaniu na wciśnięcia klawiszy, kliknięcia myszą czy inne wydarzenia i w przypadku ich wystąpienia podejmować odpowiednie akcje. Proces ten odbywał się niezależnie do systemu operacyjnego, który „nie wtrącał” się w działanie programu.

W dzisiejszych systemach operacyjnych, które są w całości sterowane zdarzeniami, ich wykrywanie odbywa się już automatycznie i poszczególne programy nie muszą o to dbać.

Są one aczkolwiek w odpowiedni sposób powiadamiane, gdy zajdzie jakiegokolwiek zdarzenie systemowe.

Jak to się dzieje?... Intensywnie wykorzystywany jest tu mechanizm **funkcji zwrotnych** (ang. *callback functions*). Funkcje takie są pisane przez twórcę aplikacji, ale ich wywoływaniem zajmuje się system operacyjny; robi to, gdy wystąpi jakieś zdarzenie. Przy uruchamianiu programu funkcje te muszą więc być w jakiś sposób przekazane do systemu, by ten mógł je we właściwym momencie wywoływać. Przypomina to zamawianie budzenia w hotelu na określoną godzinę: najpierw dzwoniemy do recepcji, by zamówić usługę, a potem możemy już spokojnie położyć się do snu. O wyznaczonej godzinie zadzwoni bowiem telefon, którego dźwięk z pewnością wybudzi nas z drzemki. Podobnie każdy program „zamawia usługę” powiadamiania o zdarzeniach w „recepcji” systemu operacyjnego. Przekazuje mu przy tym **wskaźnik do funkcji**, która ma być wywołana, gdy zajdzie jakieś zdarzenie. Gdy istotnie tak się stanie, system operacyjny „oddzwoni” do programu, wywołując podaną funkcję. Aplikacja może wtedy zareagować na dane zdarzenie.

Rola, jaką w tym procesie odgrywają wskaźniki do funkcji, jest bodaj ich najważniejszym programistycznym zastosowaniem.

Funkcje, które są wywoływane w następstwie zdarzeń, nazywamy **procedurami zdarzeniowymi** (ang. *event procedures*).

Rozróżnianie zdarzeń

Informacja o zdarzeniu powinna oczywiście zawierać także dane o jego rodzaju; należy przecież odróżnić zdarzenia pochodzące od klawiatury, myszy, okien, systemu plików czy jeszcze innych kategorii i obiektów. Można to uczynić kilkoma drogami, a wszystkie mają swoje wady i zalety.

Pierwszy z nich zakłada obecność tylko jednej procedury zdarzeniowej, która dostaje informacje o wszystkich występujących zdarzeniach. W takim wypadku konieczne są dodatkowe parametry funkcji, poprzez które przekazywany będzie rodzaj zdarzenia (np. jako odpowiednia stała wyliczeniowa) oraz ewentualne dane z nim związane. W treści takiej procedury wystąpią zapewne odpowiednie instrukcje warunkowe, dzięki którym podjęte zostaną akcje właściwe danym rodzajom zdarzeń.

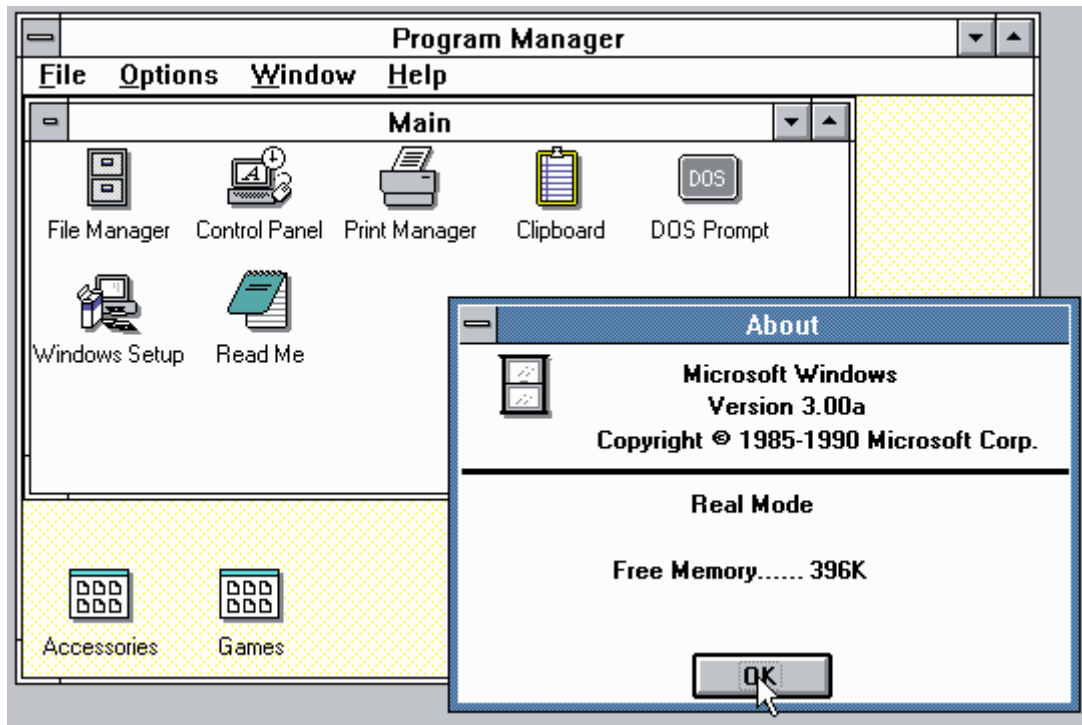
Inny wariant zakłada zgrupowanie podobnych zdarzeń w taki sposób, aby o ich wystąpieniu były informowane oddzielne procedury. Przy tym rozwiązaniu można mieć osobne funkcje reagujące na zdarzenia myszy, osobne dla obsługi klawiatury itp. Trzeci sposób wiąże się ze specjalnymi procedurami dla każdego zdarzenia. Gdy go wykorzystujemy, o każdym rodzaju zdarzeń (wciśnięcie klawisza, kliknięcie przyciskiem myszy, zakończenie programu itd.) dowiadujemy się poprzez wywołanie unikalnej dla niego procedury zdarzeniowej. Jednemu zdarzeniu odpowiada więc jedna taka procedura.

Jeżeli chodzi o wykorzystanie w praktyce, to stosuje się zwykle pierwszą lub trzecią możliwość. Pojedyncza procedura zdarzeniowa występuje w programowaniu Windows przy użyciu jego API - poznamy ją jeszcze w tym rozdziale. Osobne procedury dla każdego możliwego zdarzenia są natomiast częste w wizualnych środowiskach programistycznych, takich jak C++ Builder czy Delphi.

Fundamenty Windows

Windows jest systemem operacyjnym znanym chyba wszystkim użytkownikom komputerów i nie tylko. Chociaż wiele osób narzeka na niego z różnych powodów, nie sposób nie docenić jego roli w rozwoju komputerów. To w zasadzie dzięki Windows trafiły one pod strzechy.

Pierwsza wersja tego systemu (oznaczona numerem 1.0) została wydana niemal dwadzieścia lat temu - w listopadzie 1985 roku. Dzisiaj jest to już więc zamierzchła prehistoria, a przez kolejne lata doczekaliśmy się wielu nowych wydań tego systemu. Od samego początku posiadał on jednak graficzny interfejs oparty na oknach oraz wiele innych cech, które będą kluczowe przy tworzeniu aplikacji pracujących na nim.



Screen 49. Menedżer programów w Windows 3.0. Seria 3.x Windows była, obok Windows 95, tą, która przyniosła systemowi największą część z obecnej popularności. (screen pochodzi z serwisu Nathan's Toasty Technology)

O dokładnej historii Windows możesz przeczytać w [internetowym serwisie Microsoftu](#).

Tym fundamentom Windows poświęcimy teraz nieco uwagi.

Okna

W Windows najważniejsze są okna; są na tyle ważne, że system wziął od nich nawet swoją nazwę. Chociaż więc pomysł zamknięcia interfejsu użytkownika w takie prostokątne obszary ekranu pochodzi od MacOS'a, jego popularyzację zawdzięczamy przede wszystkim systemowi z Redmond. Dzisiaj okna występują w każdym graficznym systemie operacyjnym, od Linuxów po QNX.

Intuicyjnie za okno uważamy kolorowy prostokąt „zawierający program”. Ma on obramowanie, kilka przycisków, pasek tytułu oraz ewentualnie inne elementy. Dla systemu pojęcie to jest jednak szersze:

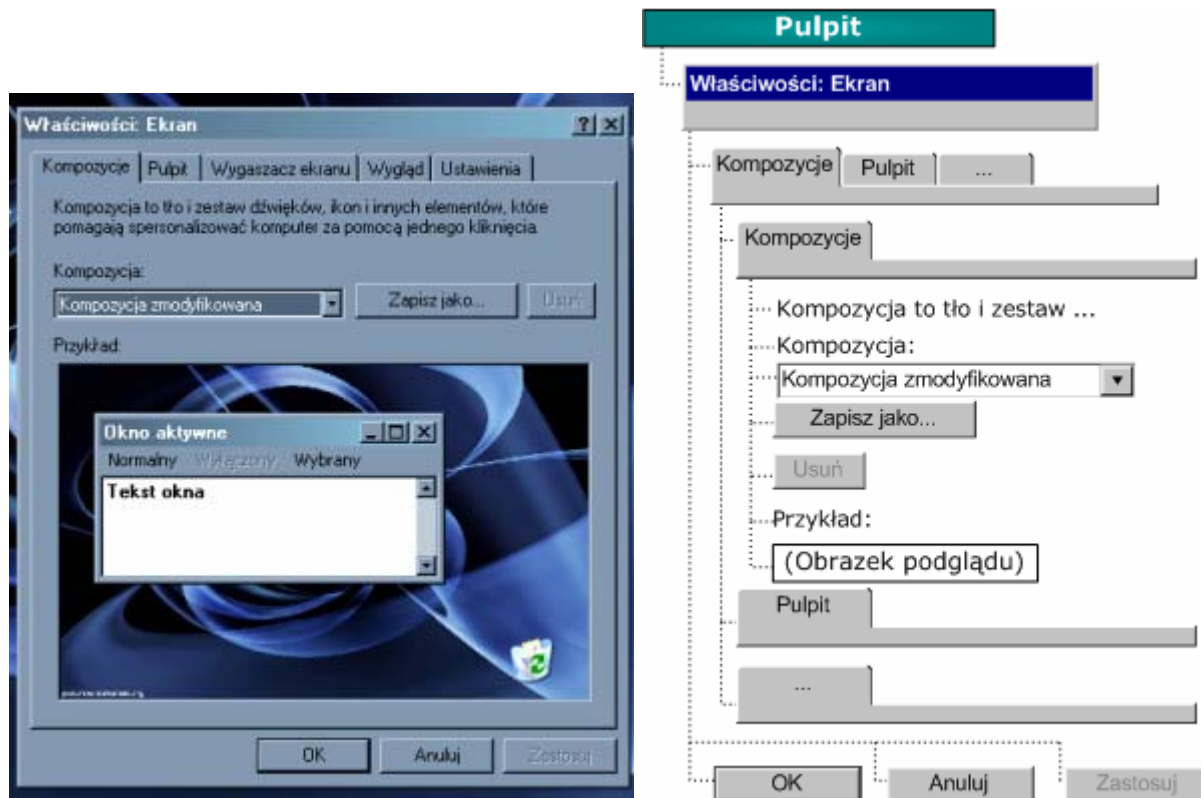
Okno (ang. *window*) to w systemie Windows dowolny element graficznego interfejsu użytkownika.

Oznacza ono, że za swoiste okna są uważane także przyciski, pola tekstowe, wyboru i inne kontrolki. Takie podejście może się wydawać dziwne, sztuczne i nielogiczne, jednak ma uzasadnienie programistyczne, o którym rychło się dowiemy,

Hierarchia okien

Jeżeli za okno uznamy każdy element GUI, wtedy dostrzeżemy także, że tworzą one **hierarchię**: pewne okno może być **nadrzędnym** dla innego, **podrzędnego**.

Na szczycie tej hierarchii widnieje **pulpit** - okno, które istnieje przez cały czas działania systemu. Bezpośrednio podległe są mu okna poszczególnych aplikacji (lub inne okna systemowe), zaś dalej hierarchia może sięgać aż do pojedynczych kontrolki (przycisków itd.).



Screen 50 i Schemat 39. Przykładowa hierarchia okien

Dzięki takiemu porządkowi Windows może w prawidłowy sposób kontrolować zachowania okien - począwszy od ich wyświetlania, a kończąc na przekazywaniu doń komunikatów (o czym będziemy mówili niedługo).

Aplikacje i procesy

Żadne okno w systemie Windows nie istnieje jednak samo dla siebie. Zawsze musi być ono związane z jakimś programem, a dokładniej z jego **instancją**.

Instancją programu (ang. *application instance*) nazywamy pojedynczy egzemplarz uruchomionej aplikacji.

Uruchomienie programu z pliku wykonywalnego EXE pociąga więc za sobą stworzenie jego instancji. Do niej są następnie „doczepiane” kolejno tworzone przez aplikację okna. Gdy zaś działanie programu dobiegnie końca, są one wszystkie niszczone. O tworzeniu i niszczeniu okien powiemy sobie w następnym podrozdziale.

Oprócz tego uruchomiony program egzystuje w pamięci operacyjnej w postaci jednego (najczęściej) lub kilku **procesów** (ang. *processes*). Cechą szczególną procesu w Windows jest to, iż posiada on wyłączną i **własną przestrzeń adresową**. Dostęp do tej

przestrzeni jest zarezerwowany tylko i wyłącznie dla niego - wszelkie inne próby nieuprawnionego odczytu lub zapisu spowodują wyjątek.

Warto też przypomnieć, że Windows jako system 32-bitowy używa **płaskiego modelu** adresowania pamięci. Każdy proces może więc teoretycznie posiadać cztery gigabajty pamięci operacyjnej do swej wyłącznej dyspozycji. W praktyce zależy to oczywiście od ilości zamontowanej w komputerze pamięci fizycznej oraz wielkości pliku wymiany.

Dynamicznie dołączane biblioteki

Jedną z przyczyn sukcesu Windows jest łatwość obsługi programów pracujących pod kontrolą tego systemu. Każda aplikacja wygląda tu podobnie, posiada zbliżony interfejs użytkownika. Nauka korzystania z nowego programu nie oznacza więc przymusu poznawania nowych elementów interfejsu, które w ogromnej większości są takie same w każdym programie.

Pamiętajmy jednak, że każdy interfejs użytkownika wymaga odpowiedniego kodu, zajmującego się jego wyświetlaniem, reakcją na kliknięcia i wciśnięcia klawiszy oraz innymi jeszcze aspektami funkcjonowania. GUI występujące w Windows nie jest tu żadnym wyjątkiem, a skoro każdy program okienkowy korzysta z tego interfejsu, musi mieć dostęp do wspomnianego kodu.

Nierozsądne byłoby jednak zakładanie, że każda aplikacja posiada jego własną **kopię**. Pomijając już marnotrawstwo miejsca na dysku i w pamięci, które by się z tym wiązało, trzeba zauważyć, że łatwo mogłyby to prowadzić do konfliktów w zakresie wersji systemu. Najprawdopodobniej należałoby wtedy całkiem zapomnieć o kompatybilności wstecz, a każda aplikacja działałaby tylko na właściwej sobie wersji systemu operacyjnego.

Problemy te są bardzo poważne i doczekały się równie poważnego rozwiązania. Lekarstwem na te bolączki są mianowicie **dynamicznie dołączane biblioteki**.

Dynamicznie dołączane biblioteki (ang. *dynamically linked libraries*, w skrócie DLL), zwane też **bibliotekami DLL** lub po prostu **DLL'ami**, są skompilowanymi modułami, zawierającymi kod (funkcje, zmienne, klasy itd.), który może być wykorzystywany przez **wiele programów** jednocześnie. Kod ten istnieje przy tym tylko w **jednej kopii** - zarówno na dysku, jak i w pamięci operacyjnej.

Biblioteki takie istnieją w postaci plików z rozszerzeniem *.dll* i są zwykle umieszczone w katalogu systemowym⁹⁷, względnie w folderach wykorzystujących je aplikacji. Udostępniają one (**eksportują**) zbiory symboli, które mogą być użyte (**zaimportowane**) w programach pracujących w Windows.

Z punktu widzenia programisty C++ korzystanie z kodu zawartego w bibliotekach DLL nie różni się wiele od stosowania zasobów Biblioteki Standardowej lub też funkcji w rodzaju `system()`, `getch()` czy `rand()`. Różnica polega na tym, że biblioteki DLL nie są statycznie dołączane do pliku wykonywalnego aplikacji, lecz **linkowane dynamicznie** (stąd ich nazwa) w czasie działania programu. W pliku EXE muszą się jedynie znaleźć informacje o nazwach wykorzystywanych bibliotek oraz o symbolach, które są z nich importowane. Dane te są automatycznie zapisywane przez kompilator jako tzw. **tabele importu**.

Wyodrębnienie kluczowego kodu systemu Windows w postaci bibliotek DLL likwiduje zatem wszystkie dolegliwości związane z jego wykorzystaniem w aplikacjach. Mechanizm dynamicznych bibliotek pozwala ponadto na tworzenie innych, własnych skarbnic kodu, które mogą być współużytkowane przez wiele programów. W takiej postaci istnieje na

⁹⁷ W Windows 9x jest to `\WINDOWS\SYSTEM\`, w Windows NT zaś `\WINDOWS\SYSTEM32\`.

przykład platforma DirectX czy moduł FMod, które będziemy w przyszłości wykorzystywać, pisząc swoje gry.

Windows API

Kod, który będziemy wykorzystywać w tworzeniu aplikacji okienkowych, nosi ogólną nazwę **Windows API**. API to skrót od *Applications' Programmed Interface*, czyli „interfejsu programowanego aplikacji”.

Windows API (czasem zwane Win32API lub po prostu WinAPI) to zbiór funkcji, typów danych i innych zasobów programistycznych, pozwalający tworzyć programy działające w trybie graficznym pod kontrolą systemu Windows.

Nauka pisania programów okienkowych polega w dużej mierze na przyswojeniu sobie umiejętności posługiwania się tą biblioteką. Temu właśnie celowi będą podporządkowane najbliższe rozdziały niniejszego kursu, łącznie z aktualnym.

Na początek aczkolwiek przyjrzymy się WinAPI jako całości i poznamy kilka przydatnych zasad, wspomagających programowanie z użyciem jego zasobów.

Biblioteki DLL

Windows API jest zawarte w bibliotekach DLL. Wraz z kolejnymi wersjami systemu bibliotek tych przybywało - pojawiły się moduły odpowiedzialne za multimedia, komunikację sieciową, Internet i jeszcze wiele innych.

Najważniejsze trzy z nich były jednak obecne od samego początku⁹⁸ i to one tworzą zasadniczą część interfejsu programistycznego Windows. Są to:

- *kernel32.dll* - w niej zawarte są funkcje sterujące jądrem (ang. *kernel*) systemu, zarządzające pamięcią, procesami, wątkami i innymi niskopoziomowymi sprawami, które są kluczowe dla funkcjonowania systemu operacyjnego.
- *user32.dll* - odpowiada za graficzny interfejs użytkownika, czyli za okna - ich wyświetlanie i interaktywność.
- *gdi32.dll* - jest to elastyczna biblioteka graficzna, pozwalająca rysować skomplikowane kształty, bitmapy oraz tekst na dowolnym rodzaju urządzeń wyjściowych. Zapoznamy się z nią w rozdziale 3, *Windows GDI*.

Każda z tych bibliotek eksportuje setki funkcji. Bogactwo to może przyprawić o zawrót głowy, ale wkrótce przekonasz się, że korzystanie z niego jest całkiem proste. Poza tym, jak wiadomo, od przybytku głowa nie boli ;-)

Pliki nagłówkowe

Biblioteki DLL są zasadniczo przeznaczone dla samych programów; z nich czerpią one kod potrzebnych funkcji Windows API. Dla nas, programistów, ważniejsze są mechanizmy, które pozwalają użyć tychże funkcji w C++.

I tu spotyka nas miła niespodzianka: wykorzystanie WinAPI w aplikacjach pisanych w C++ przebiega bowiem podobnie, jak stosowanie modułów Biblioteki Standardowej. Wymaga mianowicie dołączenia odpowiedniego pliku nagłówkowego. Tym plikiem jest *windows.h*. Dołączając go, otrzymujemy dostęp do wszystkich podstawowych i części bardziej zaawansowanych funkcji Windows API. Warto przy tym podkreślić, że nawet owe „podstawowe” funkcje pozwalają tworzyć rozbudowane i skomplikowane aplikacje, a dla programistów chcących pisać głównie gry w DirectX będą one znacznie więcej niż wystarczające.

⁹⁸ Chociaż nie zawsze były 32-bitowe i ich nazwy nie kończyły się na 32.

Jeszcze przed dołączeniem (za pomocą dyrektywy `#include`) nagłówka *windows.h* dobrze jest zdefiniować (poprzez `#define`) makro `WIN32_LEAN_AND_MEAN`. Wyłączy to niektóre rzadziej używane fragmenty API, zmniejszając rozmiar powstałych plików wykonywalnych i skracając czas potrzebny na ich zbudowanie. Będę stosował tę sztuczkę we wszystkich programach przykładowych, w których będzie to możliwe.

windows.h wewnątrznie dołącza także wiele innych plików nagłówkowych, z których najważniejszymi są:

- *windef.h*, zawierający definicje typów (głównie strukturalnych) używanych w Windows API
- *winbase.h*, który udostępnia funkcje jądra systemu (z biblioteki *kernel32.dll*)
- *winuser.h*, odpowiedzialny za interfejs użytkownika (czyli bibliotekę *user32.dll*)
- *wingdi.h*, udostępniający moduł graficzny GDI (biblioteka *gdi32.dll*)

Oprócz tych nagłówków istnieje także całe mnóstwo rzadziej używanych, odpowiadających na przykład za programowanie sieciowe (*winsock.h*) czy też obsługę multimediów (*winmm.h*). Będziesz je dołączał, jeżeli zechcesz skorzystać z bardziej zaawansowanych możliwości systemu Windows.

O funkcjach Windows API

Większą część wymienionych plików nagłówkowych stanowią prototypy funkcji, używanych w programach okienkowych. Jest ich przynajmniej kilkaset, zgrupowanych w kilkakanaście zespołów zajmujących się poszczególnymi aspektami systemu operacyjnego.

Mnogość tych funkcji nie powinna jednak przerażać. Znacząca przede wszystkim to, iż Windows API jest niezwykle potężnym narzędziem, które oferuje wiele przydatnych możliwości. Tak naprawdę bardzo niewiele jest czynności, których wykonanie przy pomocy tego ogromnego zbioru jest niemożliwe.

Naturalnie nie zawsze tak było. W ciągu tych kilkunastu lat istnienia systemu Windows jego API cały czas się rozrastało i ulegało poszerzeniu o nowe instrumenty i funkcje. Z czasem wprowadzono lepsze sposoby realizacji tych samych czynności; konsekwencją tego jest częsta obecność **dwóch wersji** funkcji realizujących to samo zadanie.

Jedna z nich jest wariantem **podstawowym** (ang. *basic*), wykonującym swoją pracę w pewien określony, domyślny sposób. Funkcje takie można poznać po ich zwyczajnych nazwach, jak na przykład `CreateWindow()` (stworzenie okna), `ReadFile()` (odczytanie danych z pliku), `ShellExecute()` (uruchomienie/otwarcie jakiegoś obiektu), itp. Ponadto istnieją też bardziej zaawansowane, **rozszerzone** (ang. *extended*) wersje niektórych funkcji. Poznać je można po przyrostku `Ex`, a także po tym, iż przyjmują one większą ilość danych jako swoje parametry⁹⁹. Pozwalają tym samym ściślej określić sposób wykonania danego zadania. Rozszerzonymi kuzynami poprzednio wymienionych funkcji są więc `CreateWindowEx()`, `ReadFileEx()` oraz `ShellExecuteEx()`.

Atoli nie wszystkie funkcje mają swe rozszerzone odpowiedniki - wręcz przeciwnie, większość z nich takowych nie posiada. Jeżeli jednak występują, wówczas zalecane jest używanie właśnie ich. Są to bowiem nowsze wersje funkcji, które mogą wykonywać zlecone sobie zadania nie tylko w bardziej elastyczny, ale też w ogólnie lepszy (czyli wydajniejszy, bezpieczniejszy itp.) sposób. Kiedy więc stajemy przed podobnym wyborem, pamiętajmy, że:

⁹⁹ Nie musi to od razu oznaczać, że przyjmują one **większą liczbę** parametrów. Niektóre (jak np. `ShellExecuteEx()`) żądają zamiast tego obszernej struktury, przekazanej jako parametr.

Użycie rozszerzonych funkcji Windows API (z nazwami zakończonymi na `Ex`) jest pożądane wszędzie tam, gdzie mogą one zastąpić swoje podstawowe wersje.

Podobne, choć bardziej szczegółowe zalecenia występują też w niemal każdym opisie rozszerzonej funkcji w MSDN. Dlatego też w tym kursie powyższa rada będzie skrupulatnie przestrzegana.

Jest jeszcze jeden przyrostek w nazwie funkcji, który ma specjalne znaczenie - chodzi o `Indirect` ('pośrednio'). Funkcje z tym zakończeniem różnią się od swych zwykłych krewniaków tym, że zamiast kilku(nastu) parametrów przyjmują strukturę, zawierającą pola dokładnie odpowiadające tymże parametrom.

Obiektowość symulowana przy pomocy uchwytów

Możliwe, że dziwisz się, dlaczego jest tu mowa tylko o funkcjach WinAPI, a ani słówkiem nie są wspomniane klasy, z których mogłaby składać się ta biblioteka. Czyżby więc nie korzystała ona z dobrodziejstw programowania obiektowego?...

W dużej mierze jest to prawdą. Większość składników Windows API została napisana w języku C, zatem nie może wykorzystywać obiektowych możliwości języka C++. Nie można jednakże powiedzieć, iż jest to biblioteka strukturalna - jej twórców nie zniechęciła bowiem ułomność języka programowania i zdołali z powodzeniem zaimplementować obiektowy projekt w zgoła nieobiektywnym środowisku. Nie da się ukryć, że było to niezbędne. Mnóstwo koncepcji Windows (z oknami na czele) daje się bowiem sensownie przedstawić jedynie za pomocą technik zbliżonych do OOP.

W języku C++ obiekty obsługujemy najczęściej poprzez wskaźniki na nie. Gdy wywołujemy ich metody, używamy składni w rodzaju:

```
obiekt->metoda (parametry);
```

Dla kompilatora jest to prawie zwyczajne wywołanie funkcji, tyle że z dodatkowym parametrem, który wewnątrz owej funkcji (metody) jest potem reprezentowany poprzez `this`. „Prawdziwa” postać powyższej instrukcji mogłaby więc wyglądać tak:

```
metoda (obiekt, parametry);
```

Taką też składnię mają wszystkie „metodopodobne” funkcje Windows API, operujące na oknach, plikach, blokach pamięci, procesach czy innych obiektach systemowych.

Istnieje tu jednak pewna różnica. Otóż biblioteka WinAPI nie może sobie pozwolić na udostępnianie programiście wskaźników do swych wewnętrznych struktur danych; mogłoby to skończyć się błędami o nieprzewidzianych konsekwencjach. Stosuje tu więc inną technikę: obiekty są użyczane koderowi poprzez swoje **uchwyty**.

Uchwyt (ang. *handle*) to unikalny liczbowy **identyfikator obiektu**, za pomocą którego można na tym obiekcie wykonywać operacje udostępniane przez funkcje biblioteczne.

Cała gama funkcji wykorzystuje uchwyty. Niektóre z nich tworzą obiekty i zwracają je w wyniku - tak robi na przykład `CreateWindowEx()`, tworząca okno. Inne służą do wykonywania określonych działań na obiektach, a kolejne odpowiadają wreszcie za ich niszczenie i sprzątanie po nich.

Jakkolwiek więc **uchwyty nie są wskaźnikami**, widać spore podobieństwo między obydwoma konstrukcjami. Dotyczy ono także konieczności zwalniania obiektów reprezentowanych przez uchwyty, gdy już nie będą nam potrzebne. Należy używać do tego odpowiednich funkcji, z których większość poznamy wkrótce.

Niezwolnienie obiektu poprzez jego uchwyt prowadzi do zjawiska **wycieku zasobów** (ang. *resource leak*), które jest przynajmniej tak samo groźne jak wyciek pamięci w przypadku wskaźników.

Ostatnią cechą wspólną z wskaźnikami jest specjalne traktowanie wartości `NULL`, czyli **zera**. Jako uchwyt nie reprezentuje ona żadnego obiektu, zatem pełni identyczną rolę, jak pusty wskaźnik.

Typy danych

W nagłówkach Windows API widnieje, oprócz prototypów funkcji, także bardzo wiele deklaracji nowych typów danych. Spora część z nich to struktury, które w programowaniu Windows są używane bardzo często.

Większość jednak jest tylko aliasami na typy podstawowe, głównie na liczbę całkowitą bez znaku. Nadmiarowe nazwy dla takich typów mają jednak swoje uzasadnienie: pozwalają łatwiej orientować się, jakie jest znaczenie danego typu oraz jaką dokładnie rolę pełni. Jest to szczególnie ważne, gdy nie można definiować własnych klas.

Warto więc przyjrzeć się, w jaki sposób tworzone są nazwy typów w Windows. Zaczniemy najpierw od najbardziej podstawowych, będących głównie prostymi przezwiskami znanych nam z C++ rodzajów danych.

Otóż w WinAPI posiadają one swoje dodatkowe miana, które tym tylko różnią się od oryginalnych, że są pisane w całości **wielkimi literami**¹⁰⁰. Konwencja ta dotyczy zresztą każdego innego typu:

W Windows API typy danych mają nazwy składające się wyłącznie z **wielkich liter**.

Mamy zatem typy `CHAR`, `FLOAT`, `VOID` czy `DOUBLE`.

Dodatkowo dla wygody programisty zdefiniowano aliasy na liczby bez znaku:

<i>nazwa</i>	<i>właściwy typ</i>	<i>opis</i>
<code>BYTE</code>	<code>unsigned char</code>	bajt (8-bitowa liczba całkowita bez znaku)
<code>UINT</code>	<code>unsigned int</code>	liczba całkowita bez znaku i określonego rozmiaru
<code>DWORD</code>	<code>unsigned long</code>	długa (32-bitowa) liczba całkowita bez znaku
<code>WORD</code>	<code>unsigned short</code>	krótka (16-bitowa) liczba całkowita bez znaku

Tabela 13. Typy liczb całkowitych bez znaku w Windows API

Istnieje również pokazny zbiór typów wskaźnikowych, które powstają poprzez dodanie przedrostka `P` (lub `LP`) do nazwy typu podstawowego. Jest więc typ `PINT`, `PDWORD`, `PBYTE` i jeszcze mnóstwo innych.

Zaprezentowane tu nazwy typów są także stosowane w bibliotece DirectX, a zatem nie rozstaniemy się z nimi zbyt szybko i warto się do nich przyzwyczać :) Obficie występują bowiem w dokumentacjach obu bibliotek, a także w innych pomocniczych narzędziach programistycznych dla Windows.

Niemal wszystkie pozostałe typy, których nazwy biorą się od znaczenia w programach, są aliasami na typ `DWORD`, czyli 32-bitową liczbę całkowitą bez znaku. Wśród nich poczesne miejsce zajmują wszelkiego typu uchwyty; kilka najważniejszych, które spotkasz najwcześniej i najczęściej, wymienia poniższa tabelka:

¹⁰⁰ Pewnym wyjątkiem od tej reguły jest typ `BOOL`, będący aliasem na `int`, a nie na `bool`. Powód takiego nazewnictwa stanowi chyba jedną z najbardziej tajemniczych zagadek Wszechświata ;D

<i>typ</i>	<i>uchwyt do...</i>	<i>uwagi</i>
HANDLE	—	uniwersalny uchwyt do czegokolwiek
HWND	okna	jednoznacznie identyfikuje każde okno w systemie
HINSTANCE	instancji programu	program otrzymuje go od systemu w funkcji <code>WinMain()</code>
HDC	kontekstu urządzenia	można na nim wykonywać operacje graficzne z modułu Windows GDI
HMENU	menu	reprezentuje pasek menu okna (jeżeli jest)

Tabela 14. Podstawowe typy uchwytów w Windows API

Jak łatwo zauważyć, wszystkie typy uchwytów mają nazwy zaczynające się na H.

Pełną listę typów danych występujących w WinAPI wraz z opisami znajdziesz rzecz jasna w [MSDN](#).

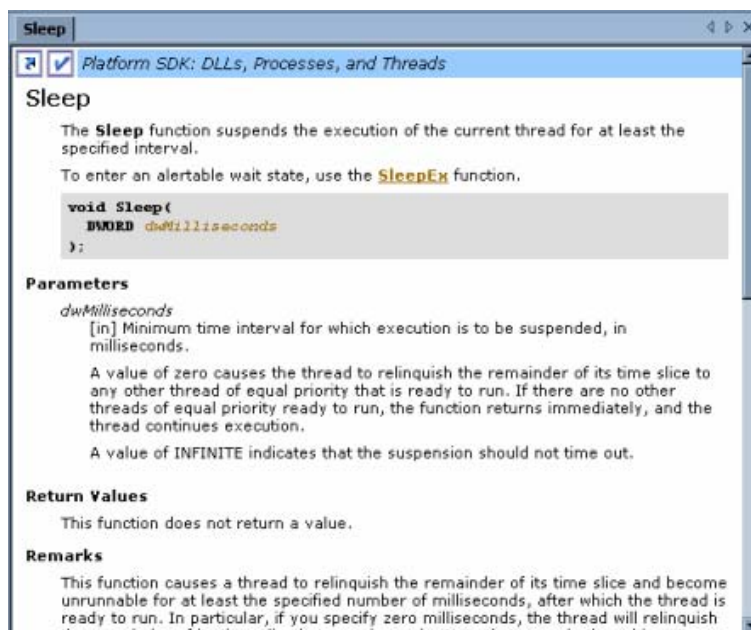
Dokumentacja

Względnie dobra znajomość nazw typów pojawiających się w Windows API jest bardzo przydatna, gdy chcemy korzystać z dokumentacji tej biblioteki. Ta dokumentacja jest bowiem najlepszym źródłem wiedzy o WinAPI.

Z początku miała ona formę pojedynczej publikacji *Win32 Programmers' Reference* (zarówno w postaci papierowej, jak i elektronicznej) i traktowała wyłącznie o podstawowych i średniozaawansowanych aspektach programowania Windows. Dzisiaj jako *Platform SDK* jest ona częścią MSDN.

To cenne źródło informacji jest domyślnie instalowane wraz z Visual Studio .NET, zatem z pewnością posiadasz już do niego dostęp (bardzo możliwe, że korzystałeś z niego już wcześniej w tym kursie). Teraz będzie ono dla ciebie szczególnie przydatne. Najczęstszym powodem, dla którego będziesz doń sięgać, jest poznanie zasady działania i użycia konkretnej funkcji. Potrzebne opisy łatwo znaleźć przy pomocy Indeksu; można też po prostu umieścić kursor w oknie kodu na nazwie interesującej funkcji i wcisnąć F1.

Dokumentacja poszczególnych funkcji ma też tę zaletę, iż posiada jednolitą strukturę w każdym przypadku.



Screen 51. Opis funkcji `Sleep()` w MSDN. Jest to chyba najprostsza funkcja Windows API; powoduje ona wstrzymanie działania programu na podaną liczbę milisekund.

Opis funkcji w MSDN składa się więc kolejno z:

- krótkiego wprowadzenia, przedstawiającego ogólnikowo sposób działania funkcji
- prototypu, z którego można dowiedzieć o liczbie, nazwach oraz typach parametrów funkcji oraz o typie zwracanej przez siebie wartości
- dokładnego opisu znaczenia każdego parametru, w kolejności ich występowania w deklaracji

Na początku każdego opisu w nawiasach kwadratowych widnieje zwykle oznaczenie jego roli. *in* oznacza, że dany parametr jest wejściową daną dla funkcji; *out* - że poprzez niego zwracana jest jakaś wartość wyjściowo; *retval* (tylko razem z *out*) - że owa wartość jest jednocześnie tą, którą funkcja zwraca w „normalny” sposób.

- informacji o wartości zwracanej przez funkcję. Jest tu podane, kiedy rezultat może być uznany za poprawny, a kiedy powinien być potraktowany jako błąd.
- dodatkowych uwag (*Remarks*) co do działania oraz stosowania funkcji
- przykładowego kodu, ilustrującego użycie funkcji
- wymaganiach systemowych, które muszą być spełnione, by można było skorzystać z funkcji. Jest tam także informacja, w której bibliotece funkcja jest zawarta i w jakim pliku nagłówkowym widnieje jej deklaracja.
- odsyłaczy do pokrewnych tematów

Ten standard dokumentacji okazał się na tyle dobry, że jest wykorzystywany niezwykle szeroko - także w projektach niezwiązanych nijak z Windows API, Microsoftem, C++, ani nawet z systemem Windows. Nic w tym dziwnego, gdyż z punktu widzenia programisty jest on bardzo wygodnym rozwiązaniem. Przekonasz się o tym sam, gdy sam zaczniesz intensywnie wykorzystywać MSDN w praktyce koderskiej.

Ten podrozdział był dość wyczerpującym wprowadzeniem w programowanie aplikacji okienkowych w Windows. Postarałem się wyjaśnić w nim wszystkie ważniejsze aspekty Windows i jego API, abyś dokładnie wiedział, w co się pakujesz ;D
W następnym podrozdziale przejdziemy wreszcie do właściwego kodowania i napiszemy swoje pierwsze prawdziwe aplikacje dla Windows.

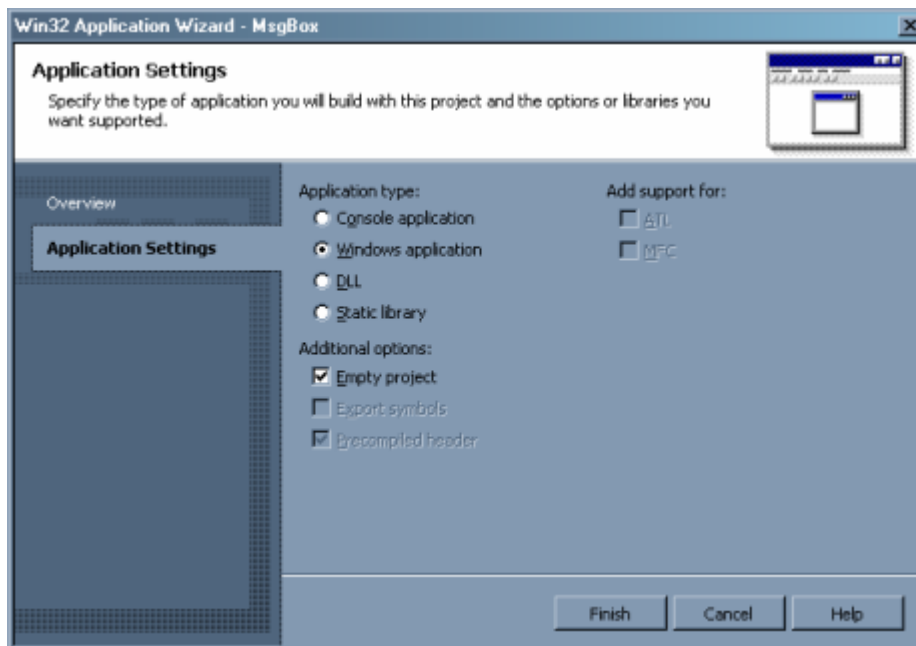
Pierwsze kroki

Gdy znamy już z grubsza całą programistyczną otoczkę Windows, czas wreszcie spróbować tworzenia aplikacji dla tego systemu. W tym podrozdziale napiszemy dwa takie programy: pierwszy pokaże jedynie prosty komunikat, ale za to w drugim stworzymy swoje pierwsze pełnowartościowe okno!
Jak najszybciej rozpoczniemy zatem właściwe programowanie dla środowiska Windows.

Najprostsza aplikacja

Od początku kursu napisałeś już pewnie całe mnóstwo programów, więc nieobca jest ci czynność uruchomienia IDE i stworzenia nowego projektu. Tym razem jednak muszą w niej zajść niewielkie zmiany.

Zmieni się nam mianowicie **rodzaj projektu**, który mamy zamiar stworzyć. Porzucamy przecież programy konsolowe, a chcemy kreować aplikacje okienkowe, działające w trybie graficznym. W opcjach projektu, na zakładce *Application Settings*, w pozycji *Application type* wybieramy zatem wariant *Windows application*:



Screen 52. Opcje projektu aplikacji okienkowej

Jako że tradycyjnie zaznaczamy także pole *Empty project*, po kliknięciu przycisku *Finish* nie zobaczymy żadnych widocznych różnic w stosunku do projektów programów konsolowych. Nasza nowa aplikacja okienkowa jest więc na razie całkowicie pusta. Kompilator wie aczkolwiek, że ma tutaj do czynienia z programem funkcjonującym w trybie GUI.

Zmianę podsystemu z GUI na konsolę lub odwrotnie można przeprowadzić, wyświetlając właściwości projektu (pozycja *Properties* z menu podręcznego w *Solution Explorer*), przechodząc do sekcji *Linker|System* i wybierając odpowiednią pozycję na liście w polu *SubSystem* (*Windows /SUBSYSTEM:WINDOWS* dla programów okienkowych lub *Console /SUBSYSTEM:CONSOLE* dla aplikacji konsolowych).

Trzeba jednakże pamiętać, że oba rodzaje projektów wymagają innych funkcji startowych: dla konsoli jest to `main()`, a dla GUI `WinMain()`. Brak właściwej funkcji objawi się natomiast błędem linkera.

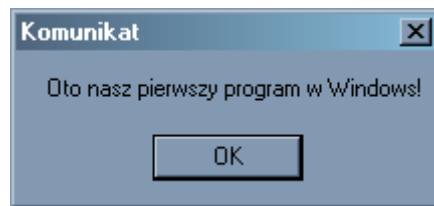
Dodajmy teraz do projektu nowy moduł *main.cpp* i wpiszmy do niego ten oto kod:

```
// MsgBox - okno komunikatu

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpszCmdLine, int nCmdShow)
{
    MessageBox (NULL, "Oto nasz pierwszy program w Windows!",
               "Komunikat", NULL);
    return 0;
}
```

Nie jest on ani specjalnie długi, ani szczególnie zawiły, gdyż jest to listing bodaj najprostszej możliwej aplikacji dla Windows. Wykonywane przezeń za danie także nie jest bardzo złożone - pokazuje ona bowiem poniższe okno z komunikatem:



Screen 53. Okno prezentujące pewien komunikat

Znika ono zaraz po kliknięciu przycisku *OK*, a to kończy również cały program. Nie zmienia to jednak faktu, że oto wyświetliliśmy na ekranie swoje pierwsze (na razie skromne) okno, żegnając się jednocześnie z czarno-białą konsolą; nie było tu po niej najmniejszego śladu. Możemy zatem z absolutną pewnością stwierdzić, iż napisany przez nas program jest rzeczywiście aplikacją okienkową!

Pokrzepieni tą motywującą wiadomością możemy teraz przystąpić do oględzin kodu naszego krótkiego programu.

Niezbędny nagłówek

Listing rozpoczyna się dwoma dyrektywami dla preprocesora:

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

Z tej pary konieczna jest oczywiście fraza `#include`. Powoduje ona dołączenie do kodu pliku nagłówkowego *windows.h*, zawierającego (bezpośrednio lub pośrednio) deklaracje niemal wszystkich symboli składających się na Windows API.

Plik ten jest więc dość spory, a rzadko mamy okazję skorzystać choćby z większości określonych tam funkcji, typów i stałych. Niezależnie od tego nazwy wszystkich tych funkcji będą jednak włączone do tabeli importu wynikowego pliku EXE, zajmując w nim nieco miejsca.

Zapobiegamy temu w pewnym stopniu, stosując drugą dyrektywę. Jak widać definiuje ona makro `WIN32_LEAN_AND_MEAN`, nie wiążąc z nim żadnej konkretnej wartości.

Makro to ma aczkolwiek swoją rolę, którą dobrze oddaje jego nazwa - w swobodnym tłumaczeniu: „Windows chudy i skąpy”. Zdefiniowanie go powoduje mianowicie wyłączenie kilku rzadko używanych mechanizmów WinAPI, przez co skompilowany program staje się mniejszy.

Dyrektywa `#define` dla tego makra musi się oczywiście znaleźć przed `#include`, dołączającym *windows.h*. W tymże nagłówku umieszczony jest bowiem szereg instrukcji `#if` i `#ifdef`, które uzależniają kompilację pewnych jego fragmentów od tego, czy omawiane makro nie zostało wcześniej zdefiniowane. Powinno więc ono być określone zanim jeszcze preprocesor zajmie się przetwarzaniem nagłówka *windows.h* - i tak też dzieje się w naszym kodzie.

Musisz wiedzieć, że z tego użytecznego makra możesz korzystać w zdecydowanej większości zwykłych programów okienkowych, a także w aplikacjach wykorzystujących biblioteki DirectX. Będzie ono występować również w przykładowych programach.

Funkcja *WinMain()*

Dalszą i zdecydowanie największą część programu zajmuje funkcja `WinMain()`; jest to jednocześnie jedyna procedura w naszej aplikacji. Musi więc pełnić w niej rolę wyjątkową i tak jest w istocie: oto bowiem punkt startowy i końcowy dla programu. `WinMain()` jest zatem **windowsowym odpowiednikiem funkcji `main()`**.

Łatwo też zauważyć, że postać tej funkcji jest znacznie bardziej skomplikowana niż `main()`. Prototyp wygląda bowiem następująco:

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

Analizując go od początku, widzimy, że funkcja zwraca wartość typu `int`. Ten sam typ może zwracać także funkcja `main()` (choć nie musi¹⁰¹), a zwany jest **kodem wyjścia**. Informuje on podmiot uruchamiający nasz program o skutkach jego wykonania. Przyjęła się tu konwencja, że `0` oznacza wykonanie bez przeszkód, natomiast inna wartość jest sygnałem jakiegoś błędu.

To dokładnie odwrotnie niż w przypadku funkcji Windows API, które będziemy sami wywoływać (`WinMain()` wywołuje bowiem system operacyjny). Tam zerowy rezultat jest sygnałem błędu - dzięki temu możliwe jest wykorzystanie tej wartości w charakterze warunku instrukcji `if`, zgadza to się także z zasadą, iż pusty uchwyt (uchwyty są często zwracane przez funkcje WinAPI) ma numer `0`. Do pobierania kodu błędu służy zaś oddzielna funkcja `GetLastError()`, o której powiemy sobie w swoim czasie (ewentualnie sam sobie o niej poczytasz we właściwym źródle :D).

Kolejną częścią prototypu jest tajemnicza fraza `WINAPI`, o której pewnie nikt nie ma pojęcia, do czego służy ;) W rzeczywistości jest to proste makro zdefiniowane jako:

```
#define WINAPI __stdcall
```

Zastępuje ono słowo kluczowe `__stdcall`, oznaczające konwencję wywołania funkcji `WinMain()`. Sposób `stdcall` jest standardową drogą do wywołania tej funkcji, a także wszystkich procedur całego Windows API (o czym wszak nie trzeba wiedzieć, aby móc je poprawnie stosować). Makro `WINAPI` (czasem zastępowane przez `APIENTRY`) jest więc koniecznym i niezbędnym składnikiem sygnatury `WinMain()`, którym aczkolwiek nie trzeba się szczególnie przejmować :)

Z pewnością najbardziej interesujące są parametry funkcji `WinMain()`, prezentujące się w bardzo słusznej liczbie czterech sztuk. Nie wszystkie są równie istotne i przydatne, a znaczenie każdego opisuje poniższa tabelka:

<i>typ</i>	<i>nazwa</i>	<i>opis</i>
HINSTANCE	<code>hInstance</code>	<code>hInstance</code> to uchwyt instancji naszego programu. Jest to więc liczba jednoznacznie identyfikująca uruchomiony egzemplarz aplikacji. To ogromnie ważna i niezwykle przydatna wartość, wymagana przy wywołaniach wielu funkcji Windows API i tak fundamentalnych czynnościach jak np. tworzenie okna. Warto zatem zapisać ją w miejscu, z którego będzie dostępna w całej aplikacji (choćby w zmiennej globalnej lub statycznym polu klasy).
HINSTANCE	<code>hPrevInstance</code>	Parametr <code>hPrevInstance</code> jest już wyłącznie reliktem przeszłości, pochodzącym z czasów 16-bitowego systemu Windows. Wówczas zawierał on uchwyt do ewentualnej

¹⁰¹ Mówiąc ściśle, to jednak musi :) Ogromna większość kompilatorów akceptuje oczywiście funkcję `main()` ze zwracanym typem `void`, ale Standard C++ głosi, że jedyną przenośną jej wersją jest `int main(int argc, char* argv[]);`. Ponieważ jednak nie zajmujemy się już konsolą, możemy nie rozstrząsać dalej tego problemu i skoncentrować się raczej na funkcji `WinMain()`.

<i>typ</i>	<i>nazwa</i>	<i>opis</i>
		<p>poprzedniej instancji uruchamianego programu. Obecnie jest on zawsze uchwytem pustym, a więc zawiera wartość <code>NULL</code>, czyli zero.</p> <p>Gdy chcemy wykryć wielokrotne uruchamianie naszej aplikacji, musimy zatem posłużyć się innymi mechanizmem. Najczęściej przegląda się listę procesów aktywnych w systemie lub też szuka innego egzemplarza głównego okna aplikacji przy pomocy funkcji <code>FindWindow()</code>.</p>
<code>[L]PSTR</code>	<code>lpzCmdLine</code>	<p>Są to argumenty wiersza poleceń, podane mu podczas jego uruchamiania. Ten sposób podawania danych jest już Windows rzadko stosowany, gdyż wymaga albo uruchomienia konsoli, albo utworzenia skrótu do programu. Niemniej znajomość parametrów, z jakimi wywołano program bywa przydatna; <code>lpzCmdLine</code> przechowuje je jako łańcuch znaków w stylu C, który można przypisać do zmiennej typu <code>std::string</code> i operować na nim wedle potrzeb.</p> <p>Zauważmy, że jest to jeden ciąg znaków. Funkcja <code>main()</code> zwykła bowiem rozbijać go na pojedyncze parametry oddzielone spacją lub ujęte w cudzysłowy. Podobnego rozbicia można zresztą w prosty sposób dokonać samodzielnie na tekście podanym w <code>lpzCmdLine</code>.</p>
<code>int</code>	<code>nCmdShow</code>	<p>Parametr ten określa sposób wyświetlenia głównego okna aplikacji. Jest on najczęściej ustawiany we właściwościach skrótu do programu i może przyjmować wartość równą jednej z kilku stałych, które poznamy w następnym rozdziale. Parametrem <code>nCmdShow</code> można więc sugerować się przy wyświetlaniu okna programu, ale nie jest to obowiązkowe (choć wskazane).</p>

Tabela 15. Parametry funkcji `WinMain()`

Widać z niej, że z nie wszystkich parametrów będziemy zawsze korzystać (z jednego nawet nigdy). W takich wypadkach warto skorzystać z możliwości, jaką oferuje C++, tzn. pominięcia nazwy niewykorzystywanego parametru. Skróćmy w ten sposób nagłówek funkcji `WinMain()`.

Okno komunikatu i funkcja `MessageBox()`

W bloku `WinMain()`, a więc we właściwych instrukcjach składających się na nasz program, dokonujemy jedynego wywołania funkcji Windows API. Jest nią funkcja `MessageBox()`, której należy bez wątplenia przyjrzeć się bliżej. Uczynimy to w tej chwili.

Prototyp

Nagłówek tej funkcji jawi się następująco:

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

Można zauważyć, że przyjmuje ona cztery parametry, których przeznaczenie zwyczajowo zaprezentujemy w odpowiedniej tabelce:

<i>typ</i>	<i>nazwa</i>	<i>opis</i>
HWND	hWindow	W parametrze tym podajemy uchwyt do okna nadrzędnego względem okna komunikatu. Zwykle używa się do tego aktywnego okna aplikacji, lecz można także użyć <code>NULL</code> (np. wtedy, gdy nie stworzyliśmy jeszcze własnego okna) - wówczas komunikat nie podlega żadnemu oknu.
LPCTSTR ¹⁰²	lpText	Tekst komunikatu , który ma być wyświetlony. Jest to stały łańcuch znaków w stylu C, który może być podany dosłownie lub np. odczytany ze zmiennej typu <code>std::string</code> przy pomocy jej metody <code>c_str()</code> .
LPCTSTR	lpCaption	Tutaj podajemy tytuł okna , którym będzie opatrzony nasz komunikat; jest to taki sam łańcuch znaków jak sam tekst wiadomości. W tym parametrze można również wstawić wskaźnik pusty (o wartości <code>NULL</code> , czyli zero), a wtedy zostanie użyty domyślny (i najczęściej nieadekwatny) tytuł "Error".
UINT	uFlags	Są to dotychczasowe parametry okna komunikatu, które określają m.in. zestaw dostępnych przycisków, wyrównanie tekstu, ewentualną ikonkę itd. Zostaną one omówione w dalszej części paragrafu. Ten parametr może także przyjąć wartość zerową, a wówczas w oknie komunikatu pojawi się jedynie przycisk <code>OK</code> .

Tabela 16. Parametry funkcji `MessageBox()`

Wynika z niej, iż pierwszy i ostatni parametr niniejszej funkcji może zostać pominięty poprzez wpisanie doń zera (`NULL`). Wtedy też pokazane okno jest najprostszym możliwym w Windows sposobem na **przekazanie użytkownikowi jakiejś informacji**. Wiadomość taką może on jedynie zaakceptować, wciskając przycisk `OK` - tak też dzieje się w naszej przykładowej aplikacji `MsgBox`.

Na tym jednakże nie kończą się możliwości funkcji `MessageBox()`. Reszta ukrywa się bowiem w jej czwartym parametrze - czas więc przyjrzeć się niektórym z tych opcji.

Opcje okna komunikatu

Ostatni parametr funkcji `MessageBox()`, nazwany tutaj `uFlags`, odpowiada za kilka aspektów wyglądu oraz zachowania pokazywanego okna komunikatu. Można w nim mianowicie ustawić:

- rodzaj przycisków, jakie pojawią się w oknie
- domyślny przycisk (jeżeli do wyboru jest kilka)
- ikonkę, jaką ma być opatrzony komunikat
- modalność okna komunikatu
- sposób wyświetlania (wyrównanie) tekstu zawiadomienia
- parametry samego okna komunikatu

Bogactwo opcji jest zatem spore i aż dziw bierze, w jaki sposób mogą się one „zmieścić” w jednej liczbie całkowitej bez znaku. Jest to jednak możliwe, ponieważ każdej z tych opcji przypisano stałą (której nazwa rozpoczyna się od `MB_`) o odpowiedniej wartości,

¹⁰² Typ `LPCTSTR` jest wskaźnikiem do ciągu znaków, czyli zasadniczo napisem w stylu C. Może on być jednak zarówno tekstem zapisanym znakami ANSI (typu `char`), jak i znakami Unicode (typu `wchar_t`). To, na który typ `LPCTSTR` jest aliasem, zostaje ustalone podczas kompilacji: gdy jest zdefiniowane makro `UNICODE`, wtedy staje się on typem `const wchar_t*`, w przeciwnym wypadku - `const char*`.

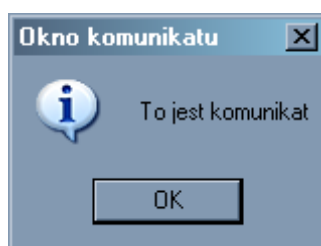
mającej w zapisie binarnym tylko jedną jedynkę na właściwym sobie miejscu. Dzięki temu poszczególne opcje (tzw. **flagi**) można „składać” w całość, posługując się do tego operatorem **alternatywy bitowej** |¹⁰³. W identyczny sposób jest to rozwiązane w innych funkcjach Windows API (i nie tylko), w których jest to konieczne.

Mechanizm ten nazywamy **kombinacją flag bitowych**, a jest on szerzej opisany w Dodatku B, *Reprezentacja danych w pamięci*.

Przykładowe użycie tego rozwiązania wygląda choćby tak:

```
MessageBox (NULL, "To jest komunikat", "Okno komunikatu", MB_OK |
MB_ICONINFORMATION);
```

Użyto w nim dwóch możliwych flag opcji: jedna określa zestaw dostępnych przycisków, a druga ikonkę widoczną w oknie komunikatu. Rzeczony okno wygląda zaś mniej więcej w ten sposób:



Screen 54. Przykładowe okno komunikatu z przyciskiem **OK** i ikonką informacyjną

Oddane do dyspozycji programisty flagi dzielą się zaś, jak to wykazaliśmy na początku, na kilka grup.

Do (naj)częściej używanych należą zapewne opcje określające **zestaw przycisków**, które pojawią się w wyświetlonym oknie komunikatu. Domyślnie zbiór ten składa się wyłącznie z przycisku **OK**, ale dopuszczalnych wariantów jest nieco więcej, co obrazuje poniższa tabelka:

flaga	przyciski	uwagi
MB_OK	<i>OK</i>	Użytkownik może wyłącznie przeczytać komunikat i zamknąć go, klikając w <i>OK</i> . Jest to domyślne ustawienie.
MB_OKCANCEL	<i>OK, Anuluj</i>	Daje prawo wyboru zaakceptowania lub odrzucenia działań zaproponowanych przez program.
MB_RETRYCANCEL	<i>Ponów próbę, Anuluj</i>	Zestaw wyświetlany zwykle wtedy, gdy jakaś operacja (np. odczyt z wymiennego dysku) nie powiodła się, ale można spróbować przeprowadzić ją ponownie.
MB_ABORTRETRYIGNORE	<i>Przerwij, Ponów próbę, Zignoruj</i>	Bardziej elastyczny wariant poprzedniego rozwiązania, stosowany przy złożonych procesach, z których pewne etapy mogą się nie powieść. Pozwala on użytkownikowi nie tylko na ponowną próbę lub zakończenie całego procesu, lecz także zignorowanie błędu.

¹⁰³ Dopuszczalne jest także użycie operatora dodawania, czyli plusa - w przypadku potęg dwójki, a takimi wartościami są właśnie flagi, będzie on miał takie samo działanie jak alternatywa bitowa. Nie jest on jednak zalecany, jako że jego podstawowe przeznaczenie jest zupełnie inne.





<i>flaga</i>	<i>przyciski</i>	<i>uwagi</i>
MB_CANCELTRYCONTINUE	Anuluj, Spróbuj ponownie, Kontynuuj	Nowsza wersja poprzedniego zestawu, zalecana do użycia w systemach Windows z serii NT.
MB_YESNO	Tak, Nie	Tak lub Nie - prosty wybór :)
MB_YESNOCANCEL	Tak, Nie, Anuluj	Oprócz wyboru <i>Tak</i> albo <i>Nie</i> można też wstrzymać się od głosu. Ten wariant jest używany np. w pytaniu o zakończenie programu, w którym pozostał niezapisany dokument.
MB_HELP	Pomoc	Flaga ta dodaje przycisk Pomoc do każdego z zestawów zaprezentowanych wcześniej. Wciśnięcie tego przycisku powoduje wysłanie zdarzenia WM_HELP do okna nadrzędnego względem komunikatu (podanego w pierwszym parametrze funkcji <code>MessageBox()</code>). Trzeba więc podać owe okno i zapewnić obsługę rzeczonego zdarzenia; o tym powiemy sobie za chwilę.
MB_DEFBUTTON1 MB_DEFBUTTON2 MB_DEFBUTTON3 MB_DEFBUTTON4	—	Wybierając jedną z tych flag określamy, który przycisk (licząc od lewej) będzie domyślny. Taki przycisk jest zaznaczony charakterystyczną czarną obwódką, a wciśnięcie Enter oznacza wskazanie właśnie jego. Jeżeli nie wybierzemy żadnej z wymienionych opcji, przyciskiem domyślnym zostanie pierwszy z nich.

Tabela 17. Flagi przycisków funkcji `MessageBox()`

Spośród powyższych opcji należy wybrać zawsze tylko jedną, odpowiednią do naszych potrzeb. Wyjątkiem są tu jedynie `MB_HELP` oraz flagi `MB_DEFBUTTONn`, które mogą być dołączone do dowolnego innego zestawu przycisków.

Wykrywaniem, który przycisk został naciśnięty, zajmiemy się natomiast w następnym akapicie.

Następna klasa opcji komunikatu nie pełni już tak kluczowej roli jak poprzednia, dotycząca przycisków, lecz także jest ważna. Pozwala bowiem na opatrzenie okna komunikatu jedną z czterech **ikonek**, zwracających uwagę użytkownika na treść i znaczenie podanej mu wiadomości. Możliwe opcje przedstawiają się tu następująco:

<i>flagi</i>	<i>typ</i>	<i>ikonka w Win 9x</i>	<i>ikonka w Win NT</i>	<i>uwagi</i>
MB_ICONASTERISK MB_ICONINFORMATION	informacja			Ikonki tej używamy, gdy chcemy przedstawić użytkownikowi jakąś zwyczajną informację, np. o pomyślnym zakończeniu zleconego zadania. Prawie zawsze idzie ona w parze z flagą <code>MB_OK</code> .
MB_ICONQUESTION	pytanie			Tą ikonką należy opatrzyć pytania, na które użytkownik powinien odpowiedzieć - z tym zastrzeżeniem, iż żadna z odpowiedzi nie może skutkować zniszczeniem




<i>flagi</i>	<i>typ</i>	<i>ikonka w Win 9x</i>	<i>ikonka w Win NT</i>	<i>uwagi</i>
MB_ICONEXCLAMATION MB_ICONWARNING	ostrzeżenie			jakichś danych. Tego symbolu używamy zarówno w ostrzeżeniach przez jakimiś niebezpiecznymi działaniami, jak i w pytaniach, z których pewne odpowiedzi mogą prowadzić do utraty danych (np. w pytaniu <i>Czy zachować aktualnie otwarty plik?</i>).
MB_ICONERROR MB_ICONHAND MB_ICONSTOP	błąd			Ten symbol służy do oznaczania wszelkich komunikatów o błędach programu lub systemu.

Tabela 18. Flagi ikonki funkcji `MessageBox()`

Warto zaznaczyć, że przyjęło się **zawsze** stosować którąś z powyższych ikonki. Pole ich zastosowań jest jednak na tyle szerokie, że wybór odpowiedniej nie powinien w konkretnym przypadku stanowić kłopotu.

Kolejne flagi nie mają już tak prostego wyjaśnienia, wiążą się bowiem z nowym pojęciem **modalności**.

Modalność (ang. *modality*) charakteryzuje te okna, których wyświetlanie **blokuje** dostęp do jednego lub więcej innych okien.

Okna modalne są używane, by pobrać od użytkownika jakieś informacje lub przedstawić mu takowe. Dobrym przykładem są okienka dialogowe, występujące niemal w każdej aplikacji, a najprostszym - właśnie okna komunikatu, tworzone przez `MessageBox()`. W Windows występuje kilka rodzajów modalności, różniących się zakresem blokady, jaką zakłada ona na pozostałe okna w systemie. Tym rodzajom odpowiadają flagi funkcji `MessageBox()`:

<i>flaga</i>	<i>modalność</i>	<i>uwagi</i>
MB_APPLMODAL	aplikacyjna	Użytkownik musi odpowiedzieć na komunikat, zanim będzie mógł uaktywnić jego okno nadrzędne (o uchwycie podanym w parametrze <code>hWindow</code> funkcji <code>MessageBox()</code>). Przy modalności aplikacyjnej komunikat blokuje zatem jedno okno (lub żadne, jeżeli w <code>hWindow</code> podamy <code>NULL</code>). Jest to domyślne ustawienie, jeżeli nie podamy innego.
MB_TASKMODAL	procesowa	Zachowanie jest niemal identyczne, jak w przypadku flagi <code>MB_APPLMODAL</code> , z tą różnicą, że gdy podamy <code>NULL</code> w parametrze <code>hWindow</code> , to blokowane są wszystkie okna aktualnej aplikacji.
MB_SYSTEMMODAL	systemowa	Jest to najsilniejszy typ modalności. Gdy go zostajemy, komunikat będzie widoczny na ekranie przez cały czas i nie zasłonią go inne okna. Użytkownik musi więc zaregować na niego, aby móc kontynuować normalną pracę. Z Tego względu modalność systemowa powinna być stosowana z rozwagą, jedynie w przypadku błędów

<i>flaga</i>	<i>modalność</i>	<i>uwagi</i>
		zagrożających całemu systemowi (np. brakowi pamięci czy miejsca na dysku).

Tabela 19. Flagi modalności funkcji `MessageBox()`

Ostatnią grupę flag stanowią różne inne przełączniki, których nie można połączyć w grupy podobne do poprzednich. Dotyczą one wszakże w większości zachowania samego okna komunikatu. Oto i one:

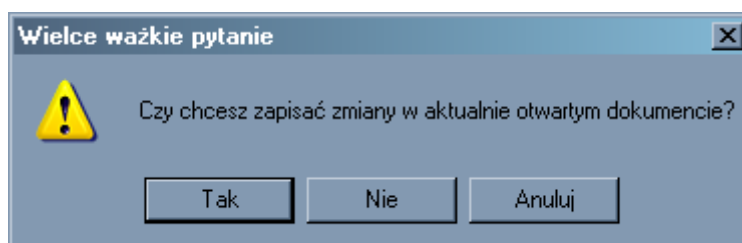
<i>flaga</i>	<i>opis</i>
<code>MB_SETFOREGROUND</code>	Zastosowanie tej flagi sprawia, że okno komunikatu zawsze „wyskakuje” na pierwszy plan, przestaniając chwilowo wszystkie pozostałe okna. Dzieje się tak nawet wtedy, gdy macierzysta aplikacja pozostaje zminimalizowana lub ukryta.
<code>MB_TOPMOST</code>	W tym ustawieniu okno nie tylko pojawia się na pierwszym planie, ale też trwale na nim pozostaje - aż do reakcji użytkownika na nie.
<code>MB_RIGHT</code>	Tekst komunikatu zostaje wyrównany do prawej strony.

Tabela 20. Pozostałe flagi funkcji `MessageBox()`

Uff, to już wszystko :) Wachlarz dostępnych opcji jest, jak widać, ogromny i z początku trudno się w nim odnaleźć. Nie musisz rzecz jasna zapamiętywać nazw i znaczenia wszystkich flag, jako że przyswoisz je sobie wówczas, gdy będziesz często korzystał z funkcji `MessageBox()`. A zapewniam cię, że tak właśnie będzie.

Rezultat funkcji

Prezentując możliwe flagi określające zestawy przycisków widocznych w oknie komunikatu, zauważyliśmy, że zdecydowana większość umożliwia użytkownikowi podjęcie jakiejś decyzji. Odbyna się ona poprzez kliknięcie w jeden z dostępnych przycisków:

Screen 55. Okno komunikatu z kilkoma przyciskami do wyboru (flagi `MB_YESNOCANCEL` | `MB_ICONWARNING`)

Informacja o wybranym przycisku jest przeznaczona dla programu, a otrzymuje on ją poprzez **wynik funkcji** `MessageBox()`. Jest to liczba typu `int`, która przyjmuje wartość jednej z następujących stałych:

<i>stała</i>	<i>przycisk</i>
<code>IDOK</code>	<i>OK</i>
<code>IDCANCEL</code>	<i>Anuluj</i>
<code>IDYES</code>	<i>Tak</i>
<code>IDNO</code>	<i>Nie</i>
<code>IDABORT</code>	<i>Przerwij</i>
<code>IDRETRY</code>	<i>Ponów próbę</i>
<code>IDIGNORE</code>	<i>Zignoruj</i>
<code>IDTRYAGAIN</code>	<i>Spróbuj ponownie</i>

<i>stała</i>	<i>przycisk</i>
IDCONTINUE	Kontynuuj

Tabela 21. Stałe zwracane przez funkcję `MessageBox()`

Naturalnie, aby funkcja mogła zwrócić wartość odpowiadającą danemu przyciskowi, ten musi zostać umieszczony w oknie komunikatu przy pomocy jednej z flag opisanych wcześniej.

Oprócz powyższych stałych `MessageBox()` może także zwrócić 0. Rezultat ten oznacza wystąpienie błędu.

Sprawdzenia decyzji użytkownika, objawiającej się wybraniem przycisku, a w konsekwencji zwróceniem wartości przez funkcję `MessageBox()`, najlepiej dokonać przy pomocy instrukcji `switch`. Jeżeli chcemy przy okazji zabezpieczyć się na ewentualność zaistnienia błędu, możemy z powodzeniem zastosować podany niżej, przykładowy kod:

```
if (UINT uDecyzja = MessageBox(NULL, "Czy wyraża Pan/Pani zgodę na
                                przystąpienie do dalszej nauki WinAPI?",
                                "Głosowanie", MB_YESNO | MB_ICONQUESTION))
{
    switch (uDecyzja)
    {
        case IDYES:
            // odpowiedź pozytywna
            break;
        case IDNO:
            // odpowiedź negatywna
            break;
    }
}
else
{
    // uwaga, błąd!
}
```

Najczęściej aczkolwiek stosuje się zwykłe porównanie wartości zwróconej przez `MessageBox()` z jakąś stałą (jedną z dwóch możliwych), na przykład `IDOK` czy `IDYES`.

Jedyna taka funkcja...

Na tym zakończymy opis funkcji `MessageBox()` - trzeba przyznać, opis dość obszerny. Zasadniczo jednak nie miałem zamiaru przepisywać dokumentacji, a tak rozbudowane objaśnienie tej funkcji ma swoje uzasadnienie.

Po pierwsze jest to jedna z najbardziej intensywnie używanych funkcji Windows API, wykorzystywana na wszystkie niemal sposoby, jakie oferuje. Dokładna znajomość metod jej wykorzystania jest zatem bardzo ważna.

Po wtóre, przy okazji omawiania tejże funkcji wyjaśniliśmy sobie kilka ważnych mechanizmów stosowanych w całym Windows API. Spośród nich najważniejszy jest sposób przekazywania opcji poprzez kombinacje flag bitowych.

Na tym jednak basta! Nie spotkasz już więcej w tym kursie tak rozbudowanych opisów funkcji Windows API. Kolejne poznawane procedury będą teraz opatrzone tylko krótkim omówieniem, a jedynie w przypadku ważniejszych funkcji przyjrzymy się bliżej również poszczególnym parametrom (ujmowanym w odpowiednią tabelkę).

Nie znaczy to wszakże, iż pozostałe funkcje WinAPI będziesz mógł znać tylko pobieżnie. Przeciwnie, nie powinieneś zapominać, że przez cały czas masz do dyspozycji obszerną dokumentację MSDN, z której możesz dowiedzieć się wszystkiego na temat każdego elementu biblioteki Windows API. Jak najczęściej korzystaj więc z tej skarbnicy wiedzy.

Możesz w niej przeczytać np. [kompletny opis funkcji MessageBox\(\)](#), uwzględniający te kilka szczegółów, które litościwie ci oszczędziłem ;D

Własne okno

Program z poprzedniego przykładu, pokazujący komunikat w małym okienku, jest z pewnością prawidłową aplikacją dla Windows, ale raczej nie tym, o co nam chodziło. Myśląc o programach Windows widzimy przede wszystkim duże okna zawierające przyciski, menu, paski narzędzi, pola edycyjne i inne kontrolki. A zatem mimo tego, że nasze programy potrafią już korzystać z graficznego interfejsu użytkownika, trudno jest nam nazwać je prawdziwie okienkowymi.

I to właśnie chcemy teraz zmienić. Nie napiszemy wprawdzie od razu jakiejś funkcjonalnej aplikacji GUI, lecz spróbujemy przynajmniej stworzyć swoje własne okno - takie, jakie mają wszystkie programy w Windows. Nie będzie to już tylko komunikat, na który użytkownik może co najwyżej popatrzeć i odwołać go kliknięciem w przycisk, lecz pełnowartościowe okno, zachowujące się tak, jak zdecydowana większość okien w systemie.

Mówimy więc o oknie **niemodalnym** (ang. *non-modal*), którego istnienie nie będzie w żaden sposób kolidowało z innymi oknami czy aplikacjami obecnymi w systemie. Będzie to po prostu nasza własna „piaskownica” - na razie pusta, ale już niedługo, w kolejnych rozdziałach, może zapełnić się różnymi ciekawymi rzeczami.

Tak więc chcemy napisać program składający się z jednego pustego okna. Oto jak może wyglądać jego kod:

```
// Window - pierwsze własne okno

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

// nazwa klasy okna
std::string g_strKlasaOkna = "od0dogk_Window";

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

LRESULT CALLBACK WindowEventProc(HWND hWindow, UINT uMsg,
                                  WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_DESTROY:
            // kończymy program
            PostQuitMessage (0);
            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) COLOR_WINDOW;

// rejestrujemy klasę okna
RegisterClassEx (&KlasaOkna);

/* tworzymy okno */

// tworzymy okno funkcją CreateWindowEx
HWND hOkno;
hOkno = CreateWindowEx(NULL, // rozszerzony styl
    g_strKlasaOkna.c_str(), // klasa okna
    "Pierwsze okno", // tekst na p. tytułu
    WS_OVERLAPPEDWINDOW, // styl okna
    CW_USEDEFAULT, // współrzędna X
    CW_USEDEFAULT, // współrzędna Y
    CW_USEDEFAULT, // szerokość
    CW_USEDEFAULT, // wysokość
    NULL, // okno nadrzędne
    NULL, // menu
    hInstance, // instancjs aplikacji
    NULL); // dodatkowe dane

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

/* pętla komunikatów */

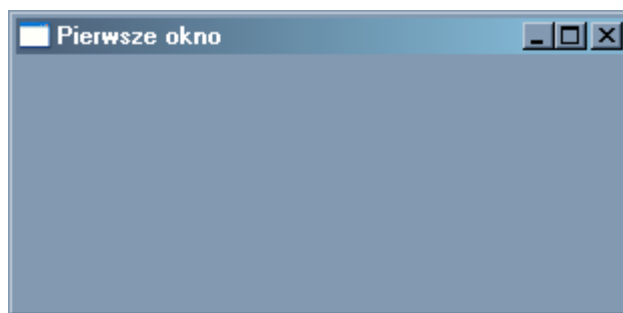
MSG msgKomunikat;
while (GetMessage(&msgKomunikat, NULL, 0, 0))
{
    TranslateMessage (&msgKomunikat);
    DispatchMessage (&msgKomunikat);
}

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

```

Taaak, na pewno nie jest to już równie proste, jak wyświetlenie tekstowego komunikatu. Widzimy tu wiele nieznanych i zapewne tajemniczych fragmentów. Bądźmy jednak spokojni, za chwilę krok po kroku wyjaśnimy sobie dokładnie wszystko, co zostało tutaj przedstawione.

Atoli teraz możesz skompilować i uruchomić powyższy program, by zobaczyć go w akcji. Ujrzysz wówczas mniej więcej coś takiego:



Screen 56. Twoje pierwsze prawdziwe okno w systemie Windows

Mimo że większość powyższego obrazka zionie pustką, możemy bez cienia wątpliwości stwierdzić, że oto wykreowaliśmy pełnowartościowe okno. Posiada ono bowiem wszystkie cechy, jakich możemy się spodziewać po oknach w Windows: możemy je przesuwac, zmieniać jego rozmiar, minimalizować, maksymalizować, przełączać się z niego do innych aplikacji czy wreszcie zamknąć, kończąc tym samym cały program. Jako że okno to nie posiada żadnej zawartości, może nam się wydać mało interesujące; zanim jednak nauczymy się wypełniać je „treścią”, musimy dogłębnie poznać sam proces jego tworzenia.

Utworzenie okna tego rodzaju, mogącego np. stanowić główną bazę jakiejś aplikacji, przebiega w dwóch etapach. Najpierw musimy zarejestrować w Windows klasę okna, a następnie stworzyć jej egzemplarz. Obie te czynności zostają u nas przeprowadzone w funkcji `WinMain()` i obecnie przyjrzymy się każdej z nich.

Klasa okna

Każde okno w systemie Windows należy do jakiejś klasy. **Klasa okna** (ang. *window class*) jest czymś rodzaju wzorca, na podstawie którego tworzone są kolejne kopie okien. Wszystkie te okna, należące do jednej klasy, mają z początku pewne cechy wspólne oraz pewne odrębne, charakterystyczne tylko dla nich.

Mechanizm ten można ewentualnie porównać do kład w programowaniu obiektowym, z tą różnicą, że tam tyczył się każdego możliwego lub niemożliwego rodzaju obiektów. Klasy okien dotyczą natomiast tylko okien i nie są aż tak elastyczne.

Co dokładnie określa klasa okna?... Najważniejszą jej właściwością jest **nazwa** - to zrozumiałe. W systemie Windows każda klasa okna musi posiadać swoją unikalną nazwę, poprzez którą można ją zidentyfikować. Podajemy to miano, gdy chcemy utworzyć okno na podstawie klasy.

Drugą bardzo ważną sprawą jest **procedura zdarzeniowa**, zajmująca się przetwarzaniem zdarzeń systemowych. Windows jest tak skonstruowany, że owa procedura jest związana właśnie z klasą okna¹⁰⁴ - wynika stąd, że:

Wszystkie **okna** należące do **jednej klasy** reagują na zdarzenia przy pomocy **tej samej procedury zdarzeniowej**.

Pozostałe cechy klasy to np. tło, jakim jest wypełniane wnętrze okna (tzw. obszar klienta), ikonka, która pojawia się w jego lewym górnym rogu, wygląd kursora przemieszczającego się nad oknem, a także kilka innych opcji.

¹⁰⁴ Możliwa jest aczkolwiek zmiana tej procedury w już istniejącym oknie danej klasy bez wpływu na inne takie okna. Technika ta nazywa się *subclassing* i zostanie omówiona we właściwym czasie. Na razie przyjmij, że wszystkie okna jednej klasy mają tę samą procedurę zdarzeniową.

Wszystkie potrzebne informacje o klasie okna umieszczamy w specjalnej strukturze, a następnie przy pomocy odpowiedniej funkcji Windows API **rejestrujemy** klasę w systemie. Jeżeli operacja ta zakończy się sukcesem, możemy już przystąpić do utworzenia właściwego okna (lub okien) na podstawie zarejestrowanej klasy.

O rejestracji klasy okna powiemy sobie za moment; najpierw musimy jeszcze rzucić okiem na najważniejszą jej część, w dużym stopniu determinującą zachowanie się programu dla Windows - na procedurę zdarzeniową.

Procedura zdarzeniowa

W naszym programie `Window` ta ważna funkcja nazywa się `WindowEventProc`. Od razu trzeba jednak zaznaczyć, że nazwa procedury zdarzeniowej nie ma tak naprawdę żadnego znaczenia i może być obrana dowolnie - najlepiej z korzyścią dla programisty. Najczęstszymi nazwami są aczkolwiek `WindowProc`, `WndProc`, `EventProc` czy `MsgProc`, jako że dobrze ilustrują one czynność, którą ta procedura wykonuje.

A tą czynnością jest **odbieranie i przetwarzanie komunikatów o zdarzeniach**. Do procedury zdarzeniowej trafiają więc informacje na temat wszelkich zaistniałych w systemie zdarzeń, które dotyczą „obsługiwanego” przez procedurę okna. Zgodnie z zasadami modelu zdarzeniowego, gdy wystąpi jakaś potencjalnie interesująca sytuacja (np. kliknięcie myszą, przyciśnięcie klawisza), system operacyjny operacyjny wywołuje procedurę zdarzeniową i podaje jej przy tym właściwe dane o zaistniałym zdarzeniu. Rola programisty piszącego treść tej procedury jest zaś odebranie owych danych i posłużenie się nimi w należyty sposób we własnej aplikacji.

Dowiedzmy się zatem, jak możemy odebrać te cenne informacje i co należy z nimi zrobić. Przyjrzymy się tedy prototypowi procedury zdarzeniowej okna:

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

Niewykluczone iż domyślasz się, że słówko `CALLBACK` pełni tu podobną rolę, co `WINAPI` w funkcji `WinMain()`, W istocie, jest to makro zastępujące tą samą nawet frazę `__stdcall`, wskazującą na konwencję wywołania. Nazwa `CALLBACK` stanowi dla nas wskazówkę, że mamy do czynienia z **funkcją zwrotną**, której wywoływaniem zajmie się dla nas ktoś inny (tu: system operacyjny Windows).

Procedura posiada cztery parametry, zadaniem których jest dostarczanie informacji o zaistniałych zdarzeniach - przede wszystkim o ich rodzaju, a także o ewentualnych danych dodatkowych oraz o odbiorcy zdarzenia.

Informację o zdarzeniu nazywamy w Windows **komunikatem** (ang. *message*).

Znaczenie poszczególnych parametrów procedury zdarzeniowej, służących do przekazania komunikatu, jest zaś następujące:

<i>typ</i>	<i>nazwa</i>	<i>opis</i>
HWND	hWindow	Przechowuje uchwyt okna, u którego wystąpiło zdarzenie . O ile pamiętamy, procedura zdarzeniowa jest ściśle związana z klasą okna, a z takiej klasy może się przecież wywodzić wiele okien. Ich komunikaty trafiają więc do tej samej procedury, lecz dzięki parametrowi <code>hWindow</code> można będzie rozróżnić ich poszczególnych odbiorców, czyli pojedyncze okna.
UINT	uMsg	Jest to informacja o rodzaju zdarzenia . Wartość tego parametru to


typ	nazwa	opis
		jedna z kilkuset (!) zdefiniowanych w systemie stałych, których nazwy rozpoczynają się od <code>WM_</code> . Każda z tych stałych odpowiada jakiemuś zdarzeniu, mogącemu wystąpić w systemie; parametr <code>uMsg</code> służy więc do ich rozróżniania i odpowiedniej reakcji na te, które interesują programistę. Z racji swej niebagatelnej roli jest też sam nazywany czasem komunikatem, podobnie jak wartości, które może przyjmować.
WPARAM	wParam	W tym parametrze, będącym (jak większość zmiennych w WinAPI) 32-bitową liczbą całkowitą bez znaku, dostarczane są szczegółowe, pomocnicze informacje o zdarzeniu. Ich znaczenie jest więc zależne od wartości <code>uMsg</code> i zawsze podawane przy opisach komunikatów w dokumentacji Windows API.
LPARAM	lParam	Ten parametr jest drugą częścią danych o zdarzeniu, aczkolwiek rzadziej używaną niż pierwsza. Podobnie jak <code>wParam</code> , jest to czterobajtowa liczba naturalna.

Tabela 22. Parametry procedury zdarzeniowej okna

Zdecydowanie najbardziej znaczący jest parametr `uMsg` - to na jego podstawie możemy odróżniać jedne zdarzenia od drugich i podejmować dla nich osobne akcje. W tym celu trzeba po prostu porównywać wartość tego parametru ze stałymi komunikatami, które nas interesują.

Najlepiej wysłużyć się tutaj instrukcją `switch` i tak też robią programiści Windows. Treść procedury zdarzeniowej jest zatem w przeważającej części blokiem wyboru, podobnym do naszego:

```
switch (uMsg)
{
    case WM_DESTROY:
        // kończymy program
        PostQuitMessage (0);
        return 0;
}
```

U nas zajmujemy się aczkolwiek tylko jednym komunikatem, któremu odpowiada stała `WM_DESTROY`. Zdarzenie to zachodzi w momencie **niszczenia okna** przez system operacyjny. To zniszczenie może z kolei zostać wywołane chociażby poprzez zamknięcie okna, gdy użytkownik klika w przycisk  w prawym górnym rogu.

Po zniszczeniu okna już rzecz jasna nie ma, a zatem nie ma też widocznych oznak „życia” naszej aplikacji. Powinniśmy wówczas ją zakończyć, co też czynimy w odpowiedzi na zdarzenie `WM_DESTROY`. Wywołujemy mianowicie funkcję `PostQuitMessage()`, która wysyła do programu komunikat `WM_QUIT`. Jest to szczególny komunikat, gdyż nie trafia on do żadnego okna aplikacji, lecz w chwili otrzymania powoduje natychmiastowe **zakończenie programu**. Jednocześnie aplikacja zwraca kod wyjścia podany jako parametr w `PostQuitMessage()`.

Zanim jednak `WM_QUIT` dotrze do aplikacji, dalej trwa wykonywanie procedury zdarzeniowej. Nie ma ona już wszakże nic do roboty, a zatem powinniśmy ją z miejsca zakończyć. Czynimy to, zwracając przy okazji rezultat¹⁰⁵ równy `0`, mówiący o pomyślnym przetworzeniu komunikatu `WM_DESTROY`.

Tak więc nasza procedura `WindowEventProc` robi generalnie bardzo prostą rzecz: kiedy wykryje zdarzenie niszczenia okna (komunikat `WM_DESTROY`), powoduje wysłanie

¹⁰⁵ Procedura zdarzeniowa zwraca wartość typu `LRESULT` - tradycyjnie, jest to liczba 32-bitowa bez znaku.

specjalnego komunikatu `WM_QUIT` do aplikacji. To zaś skutkuje zakończeniem działania programu wraz z zamknięciem jego okna przez użytkownika.

Nasze okno reaguje więc tylko na jeden komunikat, i to u kresu swego istnienia. Czy odbiera jednak także inne?... Intuicja podpowiada ci pewnie odpowiedź pozytywną: możesz przecież do woli klikać myszą w wnętrzu swego okna, przesuwać je, skalować, minimalizować, itd. Wszystkie te działania, i jeszcze mnóstwo innych, powoduje wysyłanie do okna komunikatów o zdarzeniach, a jednak nie powodują one żadnej widocznej reakcji. Co się zatem z nimi dzieje?...

No cóż, nie rozplywają się w próżni. Windows oczekuje bowiem, iż **każde zdarzenie zostanie obsłużone**, ponieważ opiera się na tym architektura tego systemu operacyjnego. Wykazuje się on jednak krztyną rozsądku i nie każe nam pisać kodu obsługi każdego z *setek* rodzajów komunikatów o zdarzeniach. Udostępnia on mianowicie funkcję `DefWindowProc()`, do której możemy (i powinniśmy!) skierować wszystkie nieobsłużone komunikaty:

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

Funkcja ta zajmie się nimi w domyślny sposób i odda rezultat ich przetwarzania. My zaś zwrócimy go jako wynik swojej własnej procedury zdarzeniowej i dzięki temu wszyscy będą zadowoleni :)

Tak oto przedstawia się w skrócie zagadnienie procedury zdarzeniowej okna w systemie Windows. Na koniec warto jeszcze przytoczyć jej sensowną składnię:

```
LRESULT CALLBACK nazwa_procedury_zdarzeniowej(HWND uchwyty_okna,
                                               UINT komunikat,
                                               WPARAM wParam,
                                               LPARAM lParam)
{
    switch (komunikat)
    {
        case zdarzenie_1:
            obsługa_zdarzenia_1
            return 0;

        case zdarzenie_2:
            obsługa_zdarzenia_2
            return 0;

        ...

        case zdarzenie_n:
            obsługa_zdarzenia_n
            return 0;
    }

    return DefWindowProc(uchwyty_okna, komunikat, wParam, lParam);
}
```

Składni tej nie trzeba się trzymać co do joty, lecz jest ona dobrym punktem startowym. Gdy nabierzesz już wprawy w programowaniu Windows, będziesz być może pisał bardziej skomplikowany kod obsługi zdarzeń, który nie zawsze zwracał będzie rezultat pozytywny. Musisz jednakże pamiętać, iż:

Nieobsłużone komunikaty należy zawsze kierować do **funkcji** `DefWindowProc()`. Ich pominięcie spowoduje bowiem niepożądane zachowanie okna.

Nie usuwaj więc nigdy ostatniej linijki procedury zdarzeniowej, sytuującej się poza blokiem `switch`. Stanowi ona nieodłączną część wymaganego kodu obsługi zdarzeń.

Rejestracja klasy okna

Procedura zdarzeniowa to najważniejszy, ale nie jedyny składnik klasy okna - oprócz niego występuje jeszcze kilka innych. Wszystkie one są polami specjalnej struktury `WNDCLASSEX`; definicja tego typu wygląda zaś tak¹⁰⁶:

```
struct WNDCLASSEX
{
    UINT cbSize;
    HINSTANCE hInstance;
    LPCTSTR lpszClassName;
    WNDPROC lpfnWndProc;
    UINT style;
    HICON hIcon;
    HICON hIconSm;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    int cbClsExtra;
    int cbWndExtra;
};
```

Mamy w nim aż tuzin różnych pól, zadeklarowanych ku radości każdego koodera ;D Ich znaczenie przedstawia nam poniższa tabelka:

<i>typ</i>	<i>nazwa</i>	<i>opis</i>
UINT	cbSize	<p>W tym polu należy wpisać rozmiar struktury <code>WNDCLASSEX</code>. Wymóg ten może się wydawać dziwaczny, niemniej jest prawdziwy i trzeba mu się podporządkować. A zatem pierwszym krokiem w pracy ze strukturą <code>WNDCLASSEX</code> powinno być ustawienie pola <code>cbSize</code> na <code>sizeof(WNDCLASSEX)</code>. Podobnie rzecz ma się z innymi strukturami w WinAPI, które posiadają te pole.</p> <p>Jeżeli struktura w Windows API posiada pole <code>cbSize</code>, należy koniecznie ustawić je na wartość równą rozmiarowi owej struktury, jeszcze zanim przekażemy ją jakiegokolwiek funkcji WinAPI.</p>
HINSTANCE	hInstance	Tutaj podajemy uchwyt do instancji programu . Tak, jest to ten sam uchwyt, jaki dostajemy w pierwszym parametrze funkcji <code>WinMain()</code> . Musimy zatem oddać Windowsowi, co od Windowsa pochodzi ;)
LPCTSTR	lpszClassName	<p>To pole jest przeznaczone dla nazwy klasy okna. Nazwa ta powinna być unikalna w skali procesu, gdyż w przeciwnym wypadku rejestracja okna nie powiedzie się. Dobrym pomysłem jest więc wpisywanie jakiejś kombinacji nazwy pisanego programu i grupy koderskiej, której jest on dziełem.</p> <p>Nazwę tę dobrze jest też zachować w odrębnej zmiennej lub stałej, bo będzie nam potrzebna przy tworzeniu okna.</p>

¹⁰⁶ Naprawdę wygląda ona inaczej, jako że składnia `struct { ... };` jest niepoprawna w C. Definicja podana tutaj jest jednak w pełni równoważna, jeśli używamy języka C++ (a używamy :)). Dlatego też kolejne definicje struktur będą podane właśnie w ten, C++'owy sposób.

typ	nazwa	opis
WNDPROC	lpfnWndProc	Oto najważniejsze pole tej struktury: wskaźnik na procedurę zdarzeniową . Musi być ona zadeklarowana zgodnie z prototypem podanym w poprzednim akapicie, jako funkcja globalna lub statyczna metoda klasy.
UINT	style	Jest to kombinacja flag bitowych określających pewne szczególne opcje klasy . Najczęściej zostawiamy to pole wyzerowane lub ustawiamy je na CS_HREDRAW CS_VREDRAW. Część dostępnych flag zaprezentujemy w następnym rozdziale.
Wszystkie są wyliczone i opisane w MSDN .		
HICON	hIcon	W tym polu określamy ikonę, jaką będą opatrzone okna przynależne rejestrowanej klasie. Dokładniej mówiąc, podajemy tu uchwyt do ikony o wymiarach co najmniej 32×32 pikseli. O tym, skąd wziąć taką ikonę, dowiesz się za moment.
HICON	hIconSm	To pole jest uchwytem do małej ikony okna, pojawiającej się w jego lewym górnym rogu. Spokojnie możemy tu podać tę samą wartość, co w hIcon (także NULL, wtedy zostanie użyta właśnie ikona z hIcon).
HCURSOR	hCursor	Kolejne pole z gatunku wystroju graficznego okna - tym razem jest to uchwyt do kursora . Strzałka myszy przyjmie jego wygląd, gdy będzie przelatywać ponad oknem należącym do definiowanej klasy. O uzyskiwaniu uchwytu do kursora też sobie zaraz powiemy. Wartość NULL w tym polu oznacza natomiast całkowity brak kursora myszy.
HBRUSH	hbrBackground	W tym miejscu podajemy uchwyt do pędzla , wypełniającego wnętrze okna. Pędzel (ang. <i>brush</i>) jest do obiektem pochodzącym z Windows GDI; w skrócie można go określić jako sposób wypełniania jakiejś powierzchni kolorem oraz deseniem (kropkami, kreskami, itp.). Sposób ten zostanie zastosowany do całego wnętrza okna. Najczęściej stosuje się tu wartość COLOR_WINDOW (rzutowaną na typ HBRUSH), gdyż wówczas okno ma domyślny, jednolity kolor. Podanie NULL spowoduje zaś powstanie przezroczystego okna ¹⁰⁷ .
LPCTSTR	lpszMenuName	Zasadniczo jest to nazwa zasobu paska menu , który to pasek ma posiadać każde okno klasy. Wiem, że w tej chwili robisz wielkie oczy, zatem na razie uznaj, że należy w polu wpisywać NULL :) O zasobach powiemy sobie natomiast za czas jakiś (długi :D).
int	cbClsExtra	Określa ilość dodatkowych bajtów, jakie zostaną zaalokowane dla klasy. Prawie zawsze wpisuje się tu zero.
int	cbWndExtra	To z kolei ilość bajtów alokowanych wraz z każdym tworzonym oknem klasy. Również wpisuje się tu często zero.

Tabela 23. Pola struktury WNDCLASSEX

¹⁰⁷ Chyba że będzie ono poprawnie odrysowywało swoją zawartość w reakcji na komunikat WM_PAINT lub WM_ERASEBKGD.

Huh, to jest dopiero struktura, co się zowie :) Jak widać Windows żąda nadzwyczaj dużo informacji w trakcie rejestrowania klasy okna. Podajmy je więc; oto, jak w naszym programie przebiega wypełnianie struktury `WNDCLASSEX`:

```
// deklaracja i wyzerowanie struktury
WNDCLASSEX KlasaOkna;
ZeroMemory (&KlasaOkna, sizeof(WNDCLASSEX));

// zapisanie wartości do pól
KlasaOkna.cbSize = sizeof(WNDCLASSEX);           // 1
KlasaOkna.hInstance = hInstance;                 // 2
KlasaOkna.lpfnWndProc = WindowEventProc;         // 3
KlasaOkna.lpszClassName = g_strKlasaOkna.c_str(); // 4
KlasaOkna.hCursor = LoadCursor(NULL, IDC_ARROW); // 5
KlasaOkna.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 6
KlasaOkna.hbrBackground = (HBRUSH) COLOR_WINDOW; // 7
```

Rozpoczynamy od jej wyzerowania: funkcja `ZeroMemory()` wypełnia zerami podany jej obszar pamięci o wyznaczonym rozmiarze - przekazujemy jej zatem wskaźnik do naszej struktury i jej wielkość w bajtach.

Dalej zapisujemy ową wielkość (1) w polu `cbSize` - jak wspomniałem w tabeli, jest to konieczne i trzeba to uczynić. Podobnie w polu `hInstance` umieszczamy (2) uchwyt do instancji naszego programu.

W kolejnym przypisaniu (3) ustalamy procedurę zdarzeniową dla okien naszej klasy. W tym celu w polu `lpfnWndProc` zapisujemy wskaźnik do uprzednio napisanej funkcji `WindowEventProc()` - jak wiemy, wystarczy tutaj napisać po prostu nazwę funkcji bez końcowych nawiasów okrągłych.

Wreszcie w polu `lpszClassName` podajemy nazwę rejestrowanej klasy (4). Zapisaliśmy ją w globalnej zmiennej typu `std::string`, lecz struktura żąda tutaj napisu w stylu C i dlatego posługujemy się skrzętnie metodą `c_str()`.

Następne ustalenia są już bardziej skomplikowane. Oto wybieramy (5) obrazek, jaki będzie pojawiał się nad naszym oknem, gdy przesunie się tam kursor myszy. Posługujemy się do tego funkcją `LoadCursor()`; potrafi ona wczytać obrazek kursora i zwrócić doń uchwyt typu `HCURSOR`. Wczytujemy natomiast standardową bitmapę strzałki, będącą kursorem systemowym, oznaczonym przez `IDC_ARROW`. Jako że nie posługujemy się tutaj własnym rysunkiem, lecz korzystamy z tych udostępnianych przez Windows, w pierwszym parametrze funkcji wpisujemy `NULL`.

Bardzo podobnie przebiega ustawienie ikony okna (6). Tym razem posługujemy się funkcją `LoadIcon()`, działającą jednak niemal identycznie: pierwszy parametr to znowu `NULL`, gdyż posłużymy się ikonami systemowymi. `IDI_APPLICATION` wskazuje zaś, że pragniemy wydobyć domyślną ikonę aplikacji Windows.

Naturalnie Windows posiada znacznie więcej wbudowanych ikon i kursorów. Odpowiadające im stałe można znaleźć w dokumentacji funkcji [LoadIcon\(\)](#) i [LoadCursor\(\)](#).

Możliwe jest rzecz jasna stosowanie własnych ikon i kursorów w tworzonych oknach - w tym celu trzeba posłużyć się mechanizmem zasobów (ang. *resources*).

Ostatnie ustawienie (7) związane jest ze sposobem graficznego wypełnienia wnętrza okna. Jak to zostało nadmienione w opisie pola `hbrBackground`, stosujemy tutaj standardowy kolor okna Windows, reprezentowany poprzez stałą `COLOR_WINDOW` rzutowaną na typ `HBRUSH`.

Na tym kończymy wypełnianie treścią struktury `WNDCLASSEX`, chociaż nie zajęliśmy każdym z jej dwunastu pól. Pozostałe otrzymały wszelako zadowalające nas wartości w wyniku początkowego wyzerowania całej struktury.

W następnym rozdziale przyjrzymy się jednak dokładniej każdemu polu tejże struktury oraz wartościom, jakie może ono przyjmować.

Gdy mamy już gotowe wszystkie informacje o klasie okna, przychodzi czas na jej zarejestrowanie. Jest to operacja nadzwyczaj prosta i ogranicza się do wywołania jednej funkcji:

```
RegisterClassEx (&KlasaOkna);
```

Funkcja `RegisterClassEx()` potrzebuje jedynie wskaźnika na przygotowaną strukturę `WNDCLASSEX` - i tą właśnie daną przekazujemy jej. Po pomyślnym wykonaniu funkcji nasza klasa okna jest już zarejestrowana i możemy wreszcie przystąpić do tworzenia samego okna.

Utworzenie i pokazanie okna

Żeby było śmieszniej, stworzenie okna to także kwestia tylko jednej funkcji - z tą różnicą, że jej wywołanie nie wygląda już tak prosto:

```
HWND hOkno;
hOkno = CreateWindowEx(NULL,
                      g_strKlasaOkna.c_str(), // rozszerzony styl
                      "Pierwsze okno",      // klasa okna
                      WS_OVERLAPPEDWINDOW,  // tekst na p. tytułu
                      CW_USEDEFAULT,        // styl okna
                      CW_USEDEFAULT,        // współrzędna X
                      CW_USEDEFAULT,        // współrzędna Y
                      CW_USEDEFAULT,        // szerokość
                      CW_USEDEFAULT,        // wysokość
                      NULL,                  // okno nadrzędne
                      NULL,                  // menu
                      hInstance,            // instancja aplikacji
                      NULL);                // dodatkowe dane
```

Oto bowiem mamy kolejny tuzin (!) „absolutnie niezbędnych” danych, przekazywanych jako parametry funkcji `CreateWindowEx()`. Nie trzeba jednakże popadać w czarną rozpacz - wszystko przecież daje się poznać i zrozumieć, a gdy już coś poznamy i zrozumiemy, wówczas staje się to bardzo łatwe :D

A zatem przyjrzyjmy się tej okazałej i ważnej funkcji.

Stworzenie okna poprzez `CreateWindowEx()`

Od razu rzucimy okiem na jej prototyp:

```
HWND CreateWindowEx(DWORD dwExStyle,
                   LPCTSTR lpClassName,
                   LPCTSTR lpWindowName,
                   DWORD dwStyle,
                   int x,
                   int y,
                   int nWidth,
                   int nHeight,
                   HWND hWndParent,
                   HMENU hMenu,
                   HINSTANCE hInstance,
                   LPVOID lpParam);
```

Zanim zajmiemy się poszczególnymi parametrami, popatrzmy na typ wartości zwracanej - jest to `HWND`, a więc uchwyt do okna. Funkcja `CreateWindowEx()` tworzy zatem okno i zwraca nam jego uchwyt; poprzez tenże uchwyt będziemy mogli wykonywać na stworzonym oknie wszelkiego rodzaju operacje. Jest to więc kluczowa wartość w programie i powinna być zapisana w przeznaczony ku temu zmiennej.

Podobnie jak wiele poprzednich i następnych funkcji, `CreateWindowEx()` może też zwrócić zero (`NULL`), jeśli operacja tworzenia okna nie zakończy się sukcesem.

Ażeby jednak skończyła się powodzeniem, musimy przekazać systemowi operacyjnemu odpowiednie dane na temat kreowanego okna. Dokonujemy tego, podając właściwe wartości parametrów `CreateWindowEx()`:

<i>typ</i>	<i>nazwa</i>	<i>opis</i>
DWORD	<code>dwExStyle</code>	<p>Parametr ten jest kombinacją flag bitowych, stanowiącą tzw. rozszerzony styl okna (ang. <i>extended window style</i>). Ów styl określa raczej zaawansowane aspekty okna i dlatego zwykle wpisujemy weń <code>NULL</code>.</p> <p>I w takim też przypadku możemy używać funkcji <code>CreateWindow()</code>, która od omawianej różni się tylko tym, iż w ogóle nie posiada parametru <code>dwExStyle</code>. Ze względu jednak na ustalenie, mówiące, że w miarę możliwości będziemy korzystać tylko z funkcji z sufiksem <code>Ex</code>, do tworzenia okna zawsze posługiwaliśmy się będziemy wywołaniem <code>CreateWindowEx()</code>.</p>
LPCTSTR	<code>lpClassName</code>	Tutaj należy podać nazwę klasy , której przynależne będzie tworzone okno. Najczęściej jest to nasza własna klasa, zarejestrowana chwilę wcześniej; wartość tego parametru powinna być zatem taka sama, jak pola <code>lpszClassName</code> w strukturze <code>WNDCLASSEX</code> .
LPCTSTR	<code>lpWindowName</code>	W tym parametrze wpisujemy tytuł okna - jest to jednocześnie tekst pojawiający się na jego pasku tytułu (tym górnym kolorowym :)).
DWORD	<code>dwStyle</code>	<p>Jest to styl okna, w największym stopniu determinujący jego wygląd i zachowanie. Parametr ten jest kombinacją flag bitowych, a o możliwych stałych, jakie możemy tutaj „wkombinować”, powiemy sobie w następnym rozdziale.</p> <p>Stała <code>WS_OVERLAPPEDWINDOW</code>, którą użyliśmy w programie przykładowym, powoduje stworzenie najzwyklejszego okna z paskiem i przyciskami tytułu oraz skalowalnym obramowaniem. Jest to jednocześnie jeden z częściej stosowanych stylów okna.</p>
<code>int</code>	<code>x</code>	Wpisujemy tutaj współrzedną poziomą okna lub <code>CW_USEDEFAULT</code> - wówczas jego pozycja zostanie ustalona domyślnie.
<code>int</code>	<code>y</code>	<code>y</code> to współrzednia pionowa okna ; jeżeli w którymś z parametrów <code>x</code> lub <code>y</code> podamy stałą <code>CW_USEDEFAULT</code> , wtedy okno pojawi się w domyślnym, ustawionym przez system miejscu.
<code>int</code>	<code>nWidth</code>	W tym parametrze podajemy szerokość okna , którą przy pomocy <code>CW_USEDEFAULT</code> także może być wybrana domyślnie.
<code>int</code>	<code>nHeight</code>	Wysokość okna , jaką umieszczamy tutaj, również można zostawić do ustalenia dla systemu operacyjnego przy

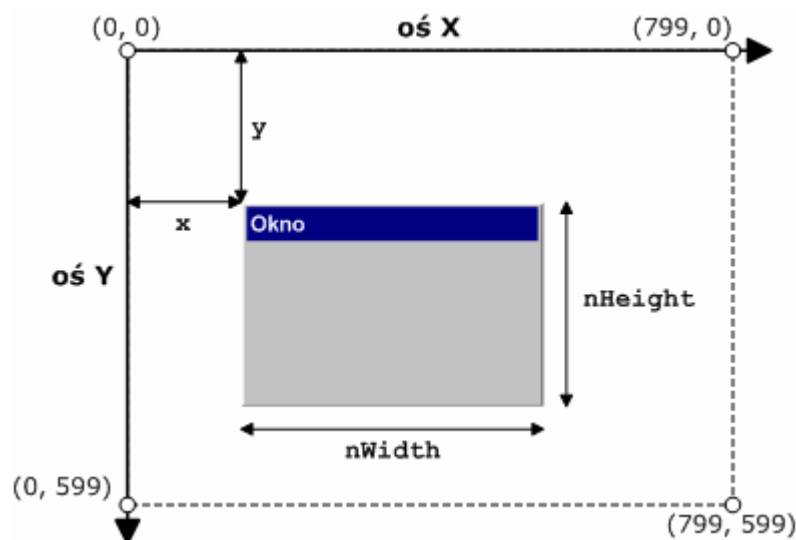
<i>typ</i>	<i>nazwa</i>	<i>opis</i>
		pomocy stałej CW_USEDEFAULT.
HWND	hWndParent	Oto uchwyt do okna nadrzędnego względem tego tworzonego przez nas. W przypadku głównych okien aplikacji (ang. <i>top-level</i>) podajemy tu NULL.
HMENU	hMenu	W tym parametrze ustalamy uchwyt do paska menu okna - oczywiście tylko wtedy, gdy ma ono takowy pasek posiadać, a my wiemy, jak go stworzyć (czego na razie nie wiemy, ale się w swoim czasie dowiemy :D). W przeciwnym razie Windows zadowolą się wartością NULL.
HINSTANCE	hInstance	Oto kolejne miejsce, w którym musimy podać swój uchwyt do instancji programu . Przypominam, że otrzymujemy go <i>explicité</i> jako parametr funkcji WinMain(), zatem nie powinno być z nim żadnego problemu.
LPVOID	lpParam	Na koniec możemy podać ewentualny dotychczasowy parametr , przekazywany do okna w chwili jego stworzenia ¹⁰⁸ . Zwykle nie ma takiej potrzeby, więc wpisujemy tu NULL.

Tabela 24. Parametry funkcji CreateWindowEx()

Nieco dłuższego wyjaśnienia wymagają parametry x , y , $nWidth$ i $nHeight$, związane z pozycją i wymiarami okna na ekranie. Otóż są one ustalane **w pikselach**, a więc zależne od rozdzielczości ekranu. Dodatkowo sposób pozycjonowania okien (i wszystkich innych obiektów) na ekranie różni się od analogicznych metod w geometrii analitycznej, bowiem:

W układzie współrzędnych ekranowych punkt (0, 0), czyli jego **początek**, znajduje się w **lewym górnym rogu** ekranu. Poza tym **oś pionowa Y** jest w tym układzie **zwrócona w dół**.

Obrazuje to dobrze poniższy rysunek:



Rysunek 6. Układ współrzędnych ekranowych w rozdzielczości 800x600

¹⁰⁸ A ściślej mówiąc, wraz z komunikatem WM_CREATE. Funkcja CreateWindow[Ex]() wysyła ten komunikat do okna zaraz po jego stworzeniu i nie oddaje kontroli do programu zanim zdarzenie to nie zostanie przetworzone w procedurze zdarzeniowej wykreowanego okna.

Widać na nim także, że parametry x i y są współrzędnymi lewego górnego rogu okna, a `nWidth` i `nHeight` określają rozmiar okna w poziomie pionie.

Te cztery parametry definiują jednocześnie pewien prostokąt na ekranie. Innym sposobem na określenia prostokąta może być także podanie pozycji jego znaczących wierzchołków, tzn. lewego górnego oraz prawego dolnego. Taki sposób jest zastosowany w strukturze `RECT`, którą poznamy z przyszłym rozdziale.

Wyświetlenie okna na ekranie

Utworzenie okna przy pomocy funkcji `CreateWindow[Ex]()` nie oznacza wszelako, że zostanie ono automatycznie pokazane¹⁰⁹. Należy bowiem zrobić to samodzielnie, co w naszym przypadku oznacza przywołanie jednej funkcji:

```
ShowWindow (hOkno, nCmdShow);
```

Nazywa się ona `ShowWindow()` i posiada dwa parametry. Pierwszy to naturalnie **uchwyty okna**, które chcemy pokazać - w tym przypadku jest to świeżo uzyskany identyfikator równie świeżo stworzonego przez nas okna :) Zapisaliśmy go w zmiennej `hOkno`. Drugi parametr określa **sposób pokazania** rzeczonego okna; u nas tym charakterze występuje wartość `nCmdShow`, parametru funkcji `WinMain()`. Decydujemy się tym samym na pokazanie głównego (i jedyne) okna programu całkowicie zgodnie z wolą jego użytkownika. Jeśli bowiem utworzy on skrót do aplikacji i określi w nim początkowy stan okna programu (normalny, zminimalizowany lub zmaksymalizowany), to tenże stan w postaci odpowiedniej stałej zostanie nam przekazany właśnie poprzez `nCmdShow`. Mówiliśmy zresztą o tym przy omawianiu funkcji `WinMain()`.

Stałe, jakie może w ogólności przyjmować drugi parametr funkcji `ShowWindow()`, są zaś następujące:

<i>stała</i>	<i>znaczenie</i>
<code>SW_SHOW</code>	pokazanie okna z zachowaniem jego pozycji i wymiarów
<code>SW_HIDE</code>	ukrycie okna
<code>SW_MAXIMIZE</code>	maksymalizacja okna („na pełny ekran”)
<code>SW_MINIMIZE</code>	minimalizacja okna („do ikony”)
<code>SW_RESTORE</code>	przywrócenie okna do normalnego stanu

Tabela 25. Ważniejsze sposoby pokazywania okna poprzez funkcję `ShowWindow()`

Jeżeli więc chcemy zignorować zapatrywania użytkownika i wyświetlać okno zawsze jako zmaksymalizowane, wówczas powinniśmy użyć instrukcji:

```
ShowWindow (hOkno, SW_MAXIMIZE);
```

Zalecane jest jednakże stosowanie parametru `nCmdShow`, przynajmniej w programach użytkowych.

Po zastosowaniu `ShowWindow()` często wywoływana jest także funkcja `UpdateWindow()`, która powoduje odrysowanie zawartości okna (poprzez wysłanie doń komunikatu `WM_PAINT`) - oczywiście tylko wtedy, gdy faktycznie coś na nim rysujemy. Zajmiemy się tym w rozdziale o Windows GDI.

¹⁰⁹ Chyba że dołączymy `WS_VISIBLE` do stylu okna (parametr `dwStyle`).

Pętla komunikatów

Dotarłszy do tego miejsca, mamy już zarejestrowaną klasę okna, a samo okno jest stworzone i widoczne na ekranie. Jego widokiem nie nacieszymy się jednak długo, jeżeli w tym momencie zakończymy pisanie funkcji `WinMain()` - co najwyżej mignie nam ono przez krótką chwilę, by zniknąć wraz z zakończeniem wykonywania tejże funkcji i, co za tym idzie, całego programu.

Trzeba więc powstrzymać funkcję `WinMain()` przed natychmiastowym zakończeniem, a jednocześnie zapewnić otrzymywanie komunikatów o zdarzeniach przez nasze okno, aby mogło ono poprawnie funkcjonować. Oba te zadania spoczywają na **pętli komunikatów**.

Pętla komunikatów (ang. *message loop*) odpowiada za odbieranie od systemu Windows komunikatów o zdarzeniach i przesyłanie ich do docelowych okien aplikacji.

Pętla ta wykonuje się przez cały czas trwania programu (chciałoby się powiedzieć, że jest nieskończona, lecz nie całkiem tak jest) i wytrwale troszczy się o jego właściwą interakcję z otoczeniem.

Kod pętli komunikatów może przedstawiać się następująco:

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

Jest to jej najprostszy wariant, ale dla naszych teraźniejszych potrzeb całkowicie wystarczający. Wyjaśnijmy sobie jego działanie.

Otóż nasza pętla komunikatów działa dopóty, dopóki program nie zamieru zostać zakończony. Przez cały ten czas wykonuje przy tym bardzo pożyteczną pracę: pobiera nadchodzące informacje o zdarzeniach z **kolejki komunikatów** (ang. *message queue*) Windows, a następnie wysyła je do właściwych im okien. Kolejka komunikatów jest zaś wewnętrzną strukturą danych systemu operacyjnego, istniejącą dla każdej uruchomionej w nim aplikacji. Na jeden koniec tej kolejki trafiają wszystkie komunikaty o zdarzeniach, jakie pochodzą ze wszystkich możliwych źródeł w systemie; z drugiego jej końca program pobiera te komunikaty i reaguje na nie zgodnie z intencją programisty. W ten sposób nadchodzące zdarzenia są przetwarzane w kolejności pojawiania się, a żadne z nich nie zostaje „zgubione”.

Nawet jeśli program przez chwilę zdaje się nie odpowiadać, zajęty swoimi czynnościami, kolejka komunikatów nadal rejestruje zdarzenia, „nie zapominając” o żadnym kliknięciu czy przyciśnięciu klawisza. Kiedy więc aplikacja „odwiesi” się, zareaguje na każde z owych zdarzeń, aczkolwiek z pewnym opóźnieniem. Wynika stąd na przykład, iż nie musimy przerywać pisania tekstu w edytorze nawet jeżeli przez jakiś czas nie pojawia się on na ekranie.

Za pobieranie komunikatów od systemu odpowiedzialna jest funkcja `GetMessage()`. Umieszcza ona uzyskany komunikat w strukturze specjalnego typu `MSG`, zawierającej między innymi cztery pola odpowiadające parametrom procedury zdarzeniowej. Adres tej struktury (u nas nazywa się ona `msgKomunikat`) podajemy w pierwszym parametrze funkcji `GetMessage()`; pozostałe trzy są parametry w większości przypadków wypełniane zerami.

Wartość zwrócona przez `GetMessage()` jest także bardzo ważna, skoro używamy jej jako warunku pętli `while` - pętli komunikatów. Omawiana funkcja zwraca bowiem zero (co

przerzywa pętlę), gdy odebrany komunikat jest `WM_QUIT`. Poznaliśmy ten specjalny komunikat, kiedy jeszcze pisaliśmy procedurę zdarzeniową okna, a teraz jego wyjątkowa rola potwierdziła się, skoro:

Odebranie komunikatu `WM_QUIT` powoduje zakończenie działania programu.

Gdy zatem `WM_QUIT` przerwie wykonywanie pętli komunikatów, funkcja `WinMain()` osiągnie swoją ostatnią instrukcję, czyli:

```
return static_cast<int>(msgKomunikat.wParam);
```

Zwraca ona ten kod wyjścia, który podaliśmy podówczas w funkcji `PostQuitMessage()` (czyli 0), jako rezultat działania `WinMain()` - a więc, co za tym idzie, wynik wykonywania całej aplikacji. Jest on więc zapisywany w parametrze `wParam` komunikatu `WM_QUIT`, skąd go teraz wydobywamy; cały komunikat jest bowiem przechowany w strukturze `msgKomunikat`, dokąd trafił po ostatnim (terminalnym) wywołaniu funkcji `GetMessage()`.

W ten zatem sposób kończy się funkcjonowanie naszego programu, lecz my chcemy jeszcze przeglądać zawartość bloku pętli komunikatów. Przedstawia się on w nader prostej formie przywołania dwóch funkcji:

```
TranslateMessage (&msgKomunikat);
DispatchMessage (&msgKomunikat);
```

Owe wywołania pełnią rolę swoistego folkloru wśród programistów Windows, ponieważ wszyscy oni doskonale wiedzą, że są to niezbędnie konieczne instrukcje, ale niewielu ma przy tym jakiegokolwiek pojęcie, co one właściwie robią ;)) Dzieje się tak chyba dlatego, że nie nastęrczają one nigdy żadnych problemów. Warto byłoby aczkolwiek znać ich zadania.

Oto więc `TranslateMessage()` dokonuje „przetłumaczenia” co niektórych komunikatów, zmieniając je w razie potrzeby w inne. Dotyczy to w szczególności zdarzeń związanych z klawiaturą - przykładowo, następujące po sobie komunikaty o wciśnięciu (i przytrzymaniu) oraz puszczeniu tego samego klawisza mogą (a nawet powinny) być zinterpretowane jako pojedyncze wciśnięcie tegoż klawisza. `TranslateMessage()` dba więc, aby faktycznie tak się tutaj działo¹¹⁰.

Z kolei `DispatchMessage()` ma bardziej klarowne zadanie do wykonania. Ta funkcja wysyła bowiem podany komunikat do jego docelowego okna, któremu jest on przeznaczony. Ni mniej, ni więcej, jak tylko dokonuje tej nieodzownej czynności, nie robiąc nic oprócz niej (bo i czy to nie wystarczy?...). Ta funkcja jest zatem podstawą działania całego windowsowego mechanizmu zdarzeń, opartego na komunikatach.

Pętla komunikatów (zwana też czasem, z racji wykonywanej pracy, **pompą komunikatów**) jest więc witalną częścią tego systemu. Przez nią przechodzą wszystkie zdarzenia, kierowane do właściwych sobie okien, które dzięki temu mogą interaktywnie współpracować z użytkownikiem, tworząc graficzny interfejs sterowany zdarzeniami.

Tą konkluzją kończymy swoje pierwsze spotkanie z oknami w Windows. Zaznajomiliśmy się w nim najpierw z podstawową funkcją `WinMain()` i wyświetlaniem komunikatu poprzez `MessageBox()`. Dalej zaliczyliśmy bliższy kontakt z rozwiązaniem zdarzeniowego modelu funkcjonowania aplikacji w Windows, a więc z procedurą zdarzeniową i pętlą

¹¹⁰ W cytowanym przykładzie znaczy to, że kolejno następujące komunikaty `WM_KEYDOWN` oraz `WM_KEYUP` zostaną uzupełnione o jeszcze jeden - `WM_CHAR`.

komunikatów. Jednocześnie też stworzyliśmy i pokazaliśmy nasze pierwsze prawdziwe okno.

Wszystko to mogło ci się wydać, oględnie się wyrażając, trochę tajemnicze. Rzeczywiście, trudno nie być przytłoczonym dziesiątkami nazw, jakie musiałem zaserwować w tym podrozdziale, i nie zadawać sobie rozpaczliwego pytania: „Czy ja to muszę znać na pamięć?” Należałoby coś w tej kwestii powiedzieć.

Otóż zasadniczo możesz odetchnąć z ulgą, chociaż może zaraz będziesz chciał złapać oddech z powrotem :) Przede wszystkim musisz jednak wiedzieć, że obecnie nawet największe biblioteki programistyczne nie są straszne koderom, jeżeli mogą z nich wygodnie korzystać. Wygodnie - to również znaczy w sposób, który częściowo odciążałby ich od konieczności pamiętania wszystkich niuansów. Nowoczesne środowisko programistyczne (na przykład Visual Studio .NET) znakomicie ułatwia bowiem programowanie z użyciem Windows API (i nie tylko), ciągle dając piszącemu kod niezwykle przydatne wskazówki. Dotyczą one w szczególności parametrów funkcji oraz pól struktur: w odpowiednich momentach pojawiają się mianowicie wszelkiego rodzaju „dymki” oraz listy, przypominające programiście prototypy funkcji, których właśnie używa, i definicje struktur, którymi się w danej chwili posługuje. Z tymi elementami biblioteki nie powinno być wszelako żadnych większych problemów.

Co do znajomości nazw funkcji i typów, to jak wszystko przychodzi ona z czasem i doświadczeniem. Na początku będziesz może tylko kopiował, wklejał i przerabiał gotowe kody, ale już wkrótce nabierzesz wystarczającej wprawy, by samodzielnie konstruować programy okienkowe - szczególnie, że przecież na tym jednym rozdziale nie kończy się nasze spotkanie z nimi.

Podsumowanie

Tworzenie aplikacji okienkowych jest w Windows całkiem proste, prawda? ;) No, może niezupełnie. Wielu programistów uważa nawet, że to bardzo, bardzo trudne zajęcie, do którego lepiej nie podchodzić zbyt blisko. My jednak podeszliśmy do niego odważnie i chyba przekonaliśmy się, że nie taki Windows straszny, jak go co niektórzy malują.

Nie obyło się oczywiście bez odpowiedniego, łagodnego wprowadzenia: najpierw poznaliśmy więc ideę graficznego interfejsu użytkownika, rozpowszechnionego we wszystkich nowoczesnych systemach operacyjnych. Dalej powiedzieliśmy sobie, czym różnią się programy pracujące w konsoli od tych wykorzystujących GUI, jeżeli chodzi o ich sposób działania - dowiedzieliśmy się tutaj o trzech modelach funkcjonowania aplikacji, ze szczególnym uwzględnieniem modelu zdarzeniowego.

Następnie przyglądaliśmy się bliżej samemu już systemowi Windows oraz narzędziom, dzięki którym możemy tworzyć aplikacje działające w tym środowisku - czyli Windows API. Uświadomiliśmy sobie tutaj znaczenie bibliotek łączonych dynamicznie, plików nagłówkowych, zadeklarowanych w nich funkcji i typów danych oraz dokumentacji MSDN, stanowiącej przewodnik po całym tym niezmiernym bogactwie.

Wreszcie zabraliśmy się do prawdziwego kodowania. Napisaliśmy więc swoją pierwszą aplikację dla Windows, wyświetlającą okno komunikatu, i poznaliśmy przy okazji rolę funkcji `WinMain()` i `MessageBox()`.

Potem zajęliśmy się już poważniejszym programem, tworzącym prawdziwe, w pełni funkcjonalne okno systemu Windows. Zapoznaliśmy się tutaj z komunikatami o zdarzeniach i sposobem reagowania na nie przy pomocy procedury zdarzeniowej; zajęliśmy się rejestracją najprostszej klasy okna; w końcu stworzyliśmy i wyświetliliśmy samo okno na ekranie komputera. Wszystko to zrobiliśmy po to, by na koniec zaobserwować pracę pętli komunikatów, czyniącej nasz program całkowicie interaktywnym.

W ten oto sposób zakosztowaliśmy przedsmaku uroków programowania dla Windows. W następnych rozdziałach będziemy poszerzać swoje wiadomości i umiejętności w tym zakresie, zyskując nowy programistyczny potencjał do działania.

Pytania i zadania

Zupełnie nowy rodzaj programów, jakie zaczęliśmy tworzyć, i zupełnie nowe środowisko, w jakim one funkcjonują, wymaga... zupełnie nowego zestawu pytań i ćwiczeń kontrolnych :D Wykonaj je zatem skwapliwie.

Pytania

1. Czym charakteryzuje się graficzny interfejs użytkownika (GUI)? Omów jego wady i zalety w porównaniu z interfejsem tekstowym i wydawaniem poleceń w konsoli. Uwzględnij sposób pracy początkującego i zaawansowanego użytkownika.
2. Wymień i scharakteryzuj trzy modele funkcjonowania programów.
3. W jaki sposób systemy operacyjne praktycznie implementują zdarzeniowy model działania programów?
4. (**Trudniejsze**) Czym jest programowanie sterowane zdarzeniami?
5. Czym jest okno w systemie Windows?
6. Co nazywamy instancją programu?
7. Jak system gospodaruje pamięcią operacyjną procesów?
8. Jakie zalety mają dynamicznie dołączane biblioteki (DLL)?
9. Co to jest Windows API?
10. Jaki plik nagłówkowy należy dołączyć do programu, aby móc korzystać z symboli Windows API?
11. Czym są i jaką rolę w Windows API odgrywają uchwyty?
12. Jak nazywa się główna funkcja programu okienkowego w Windows?
13. Do czego służy funkcja `MessageBox()` i jakie możliwości oferuje?
14. Jakie są dwa etapy utworzenia głównego okna aplikacji?
15. Jakie informacje musimy podać, rejestrując klasę okna?
16. Czym dla okna jest jego procedura zdarzeniowa?
17. (**Bardzo trudne**) Czy dwa okna należące do tej samej klasy mogą mieć różne procedury zdarzeniowe?
Wskazówka: poczytaj w MSDN o [subclassingu](#) i funkcji [SetWindowLongPtr\(\)](#).
18. Czym jest komunikat Windows?
19. Jak funkcjonuje pętla komunikatów i dlaczego jest tak ważna?

Ćwiczenia

1. Napisz okienkową wersję programu `Random` z rozdziału 1.4. Niech wyświetla ona w oknie komunikatu losową liczbę z przedziału `<1; 6>`.
2. Stwórz program, który poprosi użytkownika (poprzez okno komunikatu) o podjęcie jakiejś decyzji i zareaguje na nią w pewien sposób.
3. Napisz aplikację, która po kliknięciu myszą w swoje okno (komunikat `WM_LBUTTONDOWN`) pokaże na ekranie informację o tym.
4. (**Trudne**) Zmodyfikuj przykład `Window` tak, aby przy próbie zamknięcia okna programu użytkownik otrzymywał pytanie, czy aby na pewno chce to zrobić. Aplikacja powinna się oczywiście zakończyć tylko wtedy, gdy odpowiedź na to pytanie będzie pozytywna.
Wskazówka: zajrzyj do MSDN po [opis komunikatu WM_CLOSE](#).