



Tytuł oryginału: Hackish C++ Pranks & Tricks

Tłumaczenie: Przemysław Szeremiota

ISBN: 83-7361-801-5

Copyright © 2005 by A-LIST, LLC

Translation copyright © 2005 by Helion S.A. All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyły wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nic biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Chopina 6, 44-100 GLIWICE
tel. (32) 231-22-19, (32) 230-98-63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie?cpelha>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

| | |
|---|-----------|
| Wstęp..... | 7 |
| Wprowadzenie..... | 9 |
| O książce..... | 9 |
| Kim jest haker? Jak zostać hakerem?..... | 11 |
| Rozdział 1. Jak uczynić program zwartym, a najlepiej niewidzialnym? | 19 |
| 1.1. Kompresowanie plików wykonywalnych | 19 |
| 1.2. Ani okna, ani drzwi..... | 24 |
| 1.3. Wnętrze programu..... | 30 |
| 1.3.1. Zasoby projektu | 31 |
| 1.3.2. Kod źródłowy programu..... | 33 |
| 1.4. Optymalizacja programu..... | 43 |
| Zasada 1. Optymalizować można wszystko..... | 44 |
| Zasada 2. Szukaj wąskich gardel i słabych ogniw | 44 |
| Zasada 3. W pierwszej kolejności optymalizuj operacje często powtarzane | 45 |
| Zasada 4. Pomyśl dwa razy, zanim zoptymalizujesz operacje jednorazowe..... | 47 |
| Zasada 5. Poznaj wnętrze komputera i sposób jego działania..... | 48 |
| Zasada 6. Przygotuj tabele gotowych wyników obliczeń i korzystaj z nich w czasie działania programu | 49 |
| Zasada 7. Nie ma niepotrzebnych testów..... | 50 |
| Zasada 8. Nie bądź nadgorliwy..... | 50 |
| Podsumowanie..... | 51 |
| 1.5. Prawidłowe projektowanie okien | 51 |
| 1.5.1. Interfejs okna głównego..... | 54 |
| 1.5.2. Elementy sterujące..... | 55 |
| 1.5.3. Okna dialogowe | 55 |
| Rozdział 2. Tworzenie prostych programów-żartów | 61 |
| 2.1. Latający przycisk Start..... | 62 |
| 2.2. Zaczniij pracę od przycisku Start | 71 |
| 2.3. Zamieszanie z przyciskiem Start..... | 73 |
| 2.4. Więcej dowcipów z paskiem zadań | 76 |
| 2.5. Inne żarty | 83 |
| Jak „zgasić” monitor?..... | 83 |
| Jak uruchamiać systemowe pliki CPL? | 83 |
| Jak wysunąć tarcę napędu CD-ROM? | 84 |
| Jak usunąć zegar z paska zadań? | 86 |

| | |
|---|------------|
| Jak ukryć cudze okno?..... | 86 |
| Jak ustawić własną tapetę pulpitu?..... | 87 |
| 2.6. Berek z myszą..... | 88 |
| Szalona mysz..... | 88 |
| Latające obiekty..... | 89 |
| Mysz w klatce..... | 90 |
| Jak zmienić kształt wskaźnika myszy?..... | 91 |
| 2.7. Znajdź i zniszcz..... | 92 |
| 2.8. Pulpit..... | 93 |
| 2.9. Bomba sieciowa..... | 94 |
| Rozdział 3. Programowanie w systemie Windows..... | 97 |
| 3.1. Manipulowanie cudzymi oknami | 97 |
| 3.2. Gorączkowa drążeczka | 102 |
| 3.3. Przełączanie ekranów | 103 |
| 3.4. Niestandardowe okna | 107 |
| 3.5. Finezyjne kształty okien..... | 113 |
| 3.6. Sposoby chwytania nietypowego okna | 119 |
| 3.7. Ujawnianie haseł | 121 |
| 3.7.1. Biblioteka deszyfrowania haseł | 122 |
| 3.7.2. Deszyfrowanie hasła | 126 |
| 3.7.3. Obróćmy to w żart | 128 |
| 3.8. Monitorowanie plików wykonywalnych..... | 130 |
| 3.9. Zarządzanie ikonami pulpitu..... | 132 |
| 3.9.1. Animowanie tekstu | 133 |
| 3.9.2. Odświeżanie pulpitu | 134 |
| 3.10. Żarty z wykorzystaniem schowka | 134 |
| Rozdział 4. Sieci komputerowe | 139 |
| 4.1. Teoria sieci i protokołów sieciowych..... | 139 |
| 4.1.1. Protokoły sieciowe..... | 141 |
| Protokół IP..... | 142 |
| Protokół ARP a protokół RARP | 143 |
| 4.1.2. Protokoły transportowe..... | 143 |
| Protokół UDP — szybki | 143 |
| Protokół TCP — wolniejszy, ale solidniejszy | 144 |
| TCP — zagrożenia i słabości..... | 145 |
| 4.1.3. Protokoły warstwy aplikacji — tajemniczy NetBIOS..... | 145 |
| 4.1.4. NetBEUI | 146 |
| 4.1.5. Gniazda w Windows | 147 |
| 4.1.6. Protokoły IPX/SPX | 147 |
| 4.1.7. Porty | 148 |
| 4.2. Korzystanie z zasobów otoczenia sieciowego | 148 |
| 4.3. Struktura otoczenia sieciowego | 151 |
| 4.4. Obsługa sieci za pośrednictwem obiektów MFC | 158 |
| 4.5. Transmisja danych w sieci za pośrednictwem obiektu CSocket | 165 |
| 4.6. Bezpośrednie odwołania do biblioteki gniazd | 174 |
| 4.6.1. Obsługa błędów | 175 |
| 4.6.2. Wczytywanie biblioteki gniazd | 175 |
| 4.6.3. Tworzenie gniazda | 179 |
| 4.6.4. Funkcje strony serwera | 180 |
| 4.6.5. Funkcje strony klienta | 184 |
| 4.6.6. Wymiana danych | 186 |
| 4.6.7. Zamykanie połączenia | 191 |
| 4.6.8. Zasady stosowania protokołów bezpołączeniowych..... | 192 |

| | |
|---|------------|
| 4.7. Korzystanie z sieci za pośrednictwem protokołu TCP | 194 |
| 4.7.1. Przykładowy serwer TCP | 194 |
| 4.7.2. Przykładowy klient TCP | 199 |
| 4.7.3. Analiza przykładów | 202 |
| 4.8. Przykłady wykorzystania protokołu UDP | 204 |
| 4.8.1. Przykładowy serwer UDP | 204 |
| 4.8.2. Przykładowy klient UDP | 205 |
| 4.9. Przetwarzanie odebranych danych | 207 |
| 4.10. Wysyłanie i odbieranie danych | 209 |
| 4.10.1. Funkcja select | 210 |
| 4.10.2. Prosty przykład stosowania funkcji select | 211 |
| 4.10.3. Korzystanie z gniazd za pośrednictwem komunikatów systemowych | 213 |
| 4.10.4. Asynchroniczna wymiana danych z wykorzystaniem obiektów zdarzeń | 220 |
| Rozdział 5. Obsługa sprzętu | 223 |
| 5.1. Parametry podsystemu sieciowego | 223 |
| 5.2. Zmiana adresu IP komputera | 229 |
| 5.3. Obsługa portu szeregowego | 234 |
| 5.4. Pliki zawieszające system | 239 |
| Rozdział 6. Sztuczki, kruczki i ciekawostki | 241 |
| 6.1. Algorytm odbioru-wysyłania danych | 242 |
| 6.2. Szybki skaner portów | 245 |
| 6.3. Stan portów komputera lokalnego | 252 |
| 6.4. Serwer DHCP | 257 |
| 6.5. Protokół ICMP | 260 |
| 6.6. Śledzenie trasy wędrówki pakietu | 267 |
| 6.7. Protokół ARP | 273 |
| Podsumowanie | 283 |
| Skorowidz | 285 |

Podziękowania

Dziękuję:

Rodzicom, którzy dali mi życie. Gdyby nie oni, niniejsza książka nie powstałyby pewnie wcale; w najlepszym przypadku podpiszyałaby się pod nią zupełnie kto inny.

Żonie, która ciągle mnie wspierała, i dzieciom, które czasem pozwalały mi pracować, a czasem mnie od pracy odciagały.

Zespołom redakcyjnym magazynów, w których publikowane były moje artykuły. Dzięki tym publikacjom nabyłem doświadczenia, bardzo pomocnego przy pisaniu książki. Nie jestem pisarzem i swego czasu nie potrafiłem sklecić sensownego zdania, nie mówiąc już o całej magii redagowania tekstu za pomocą komputera. Dzięki praktyce w magazynie *Hacker* teraz to potrafię.

Zespołowi wydawnictwa A-LIST, który uwierzył we mnie jako autora i pomógł przygotować i opublikować tę książkę. Jestem wdzięczny redaktorom, którzy mojemu słowotokowi nadali ostateczną formę i poprawili znaczną liczbę popełnionych przeze mnie błędów gramatycznych.

Podziękowania mógłbym kontynuować jeszcze długo. Krótko dziękuję więc: przedszkolankom w moim przedszkolu, nauczycielom w mojej szkole, profesorom uniwersytetu, na którym studiowałem, caemu zespołowi redakcyjnemu magazynu *Hacker* i wszystkim moim przyjaciołom, którzy są ze mną, kochają mnie (mam nadzieję) i pomagają mi, jak umieją. Na koniec zaś chciałbym podziękować wszystkim moim Czytelnikom.

Specjalne podziękowania kieruję do nabywcy tej książki. Mam nadzieję, że nie odzuje rozczarowania i nie pożałuje czasu, który poświęci na lekturę, i pieniędzy, które wydał w księgarni.

Wstęp

Niniejsza książka prezentuje świat programowania w języku C++ widziany oczami haker'a. Kim jest jednak „haker”? Ja nadaję temu słowi znaczenie nieco inne od potocznego. Mówiąc o hakerze, mówię o specjalistie informatyku, a nie o wandalu i złodzieju informacji. Haker to ekspert komputerowy, który, korzystając ze swojej wiedzy, niekiedy staje się przyczyną zmartwienia innych ludzi. Słodem, książka ta traktuje o programowaniu w języku C++ z punktu widzenia zawodowego programisty sieciowego starającego się tworzyć ciekawe i przydatne programy, a nie komputerowego chuligana. Moje rozumienie pojęcia „haker” postaram się przedstawić szerzej w punkcie „Kim jest haker? Jak zostać hakerem?” kilka stron dalej.

W niniejszej książce Czytelnik będzie miał okazję obserwować zastosowania wielu niestandardowych technik programistycznych i oglądać przykłady wykorzystywania nieudokumentowanych funkcji języka C++. Co ważniejsze, zobaczy szereg ciekawych metod obsługi sieci w systemie Windows.

Książka ujawnia szereg sekretów hakerów, ucząc tworzenia programów sieciowych i robienia nieszkodliwych żartów programistycznych.

Czytelnik odkryje, jak pisać krótkie, zabawne programy, które można wykorzystać do irytowania albo rozśmieszania kolegów. Nauczy się pisać własne programy, które z pewnością zostaną docenione przez znajomych. Dzięki żartobliwym programom prezentowanym w książce można wykazać się zarówno poczuciem humoru, jak i całkiem poważną wiedzą o programowaniu z wykorzystaniem systemu operacyjnego.

Postaram się wyczerpująco opisać sposoby optymalizacji rozmiaru i wydajności programów. Powinno to pomóc w tworzeniu w przyszłości programów bardziej zwartych i szybszych. Kwestie optymalizacji — mimo ciągłego zwiększania mocy obliczeniowej komputerów i pojemności ich twardych dysków — są wciąż istotne; nie wszyscy dysponują szerokopasmowym dostępem do internetu, a więc i rozmiar programów jest wciąż ważny.

Większość niniejszej książki poświęcona jest właśnie programowaniu sieciowemu z wykorzystaniem internetu i intranetu. Czytelnik będzie miał okazję napisać własny szybki skaner portów oraz program konia trojańskiego — z pewnością nie każdy może pochwalić się takimi umiejętnościami.

Poza programistycznymi dowcipami i programami sieciowymi opisuję w książce algorytmy wykorzystywane w narzędziach hakerskich. Znając język przeciwnika, można lepiej przygotować się do obrony. Hakera można bowiem pokonać jedynie znajomością jego mocnych i słabych stron.

Opisuję też kilka technik obsługi sprzętu stanowiącego wyposażenie komputera. Zagadnienia komunikacji ze sprzętem są w książkach o programowaniu traktowane po maco- szemu. Postaram się polepszyć nieco sytuację Czytelników i zaprezentować sposoby obsługi najpopularniejszych i najczęściej wykorzystywanych urządzeń.

Materiał prezentowany w książce ma prosty format, z założenia ułatwiający zrozumienie. Czytelnik nie będzie potrzebował specjalistycznej wiedzy programistycznej, a większość przykładów będzie w stanie uruchomić na własnym komputerze. W niektórych zagadnieniach pomocna będzie co prawda znajomość języka programowania C++, ale sądzę, że i bez niej Czytelnik sobie poradzi.

Wprowadzenie

Z natury jestem niepoprawnym programistą i nie przyznaję się do jakichkolwiek talentów pisarskich. Spróbuję jednak w prostych słowach podzielić się z Tobą swoją wiedzą; mam nadzieję, że moje rewelacje są Ci jeszcze nieznane. Być może znasz już niektóre z żartów i sztuczek opisanych w niniejszej książce (może niektórych z nich doświadczyłeś na własnej skórze), ale nie wiesz, jak działa ich kod. Spróbuję podejść do tematu właśnie od tej strony i ujawnić niektóre z „sekretnych” trików programistycznych.

Oferuję Czytelnikowi całą masę przykładowych programów napisanych w języku C++. Wśród nich łatwo znaleźć szereg dowcipów i aplikacji sieciowych. Książka zawiera minimalną ilość teorii, skupiając się na stronie praktycznej. Wszystko, o czym mowa, można tu „dotknąć” i zobaczyć „na własne oczy”.

Każdy, kto w książce szuka kodu wirusów, mocno się rozczaruje. Nie chcę niczegoniszczyć. Uważam się raczej za twórcę. Żaden z programów-dowcipów nie naraża swoich ofiar na jakiekolwiek poważne szkody. Dowcip jest dobry tylko wtedy, kiedy śmieją się z niego wszyscy, nie tylko inicjator.

O książce

Aby jak najwięcej skorzystać z lektury książki, warto znać przynajmniej podstawy języka C++ i posiadać choćby minimalną wprawę w obsłudze komputera i myszy. Warto wiedzieć, jak tworzyć aplikacje i jak działają pętle. Atutem byłaby też znajomość wskaźników i kwestii adresowania obiektów w pamięci programu. Wszystko to pozwoli na pełniejsze czerpanie z przykładów prezentowanych w tej publikacji. Co do programowania sieciowego, to jest ono opisane dość wyczerpująco — od samych podstaw po całkiem złożone przykłady. Tutaj przydałaby się już choćby podstawowa wiedza o sieciach, choć i przy jej braku można sporo skorzystać na lekturze.

Próbowałem prezentować materiał w jak najprostszy sposób. Większość przykładowego kodu opatrywana jest wyczerpującymi opisami i komentarzami ułatwiającymi jego analizę. Struktura książki różni się w tym zakresie od książek typowych — nie

znajdziesz tu nudnych i rozwlekłych omówień teoretycznych, a jedynie przykłady i kod źródłowy. Można tę pracę uznać za tekst czysto praktyczny. Tylko praktyka pozwala bowiem w pełni przyswoić wiedzę teoretyczną.

Programiści mają wiele wspólnego z lekarzami. Lekarz, który nie potrafi odróżnić symptomów zatrucia pokarmowego od objawów ataku wyrostka robaczkowego, nie powinien zbliżać się do pacjenta. Programista, który zna protokoły sieciowe, ale nie potrafi z nich korzystać, nie będzie w stanie napisać nawet jednego działającego programu sieciowego.

To porównanie ma swoje korzenie w moim osobistym doświadczeniu. W roku 2002 wylądowałem w szpitalu z bólem brzucha i gorączką. Lekarz podejrzewał atak wyrostka robaczkowego. Przez trzy dni nie zdobył się jednak na decyzję o operowaniu, choć nikt nie potrafił wytłumaczyć mi, skąd taki ból i temperatura.

Trzeciego dnia opuściłem szpital ze względu na urodziny mojej matki. Na przyjęciu urodzinowym jeden z gości, który okazał się być lekarzem (nawiasem mówiąc, położnikiem), zbadał mnie, zpisał leki i kazał trzymać się z daleka od szpitala. Nie uwierzycie, ale temperatura spadła już po pierwszej tabletce, a po drugiej mogłem już tańczyć. Okazało się, że szpitalny lekarz pomylił z atakiem wyrostka robaczkowego zwykłe zatrucie. O mało co nie wyciągnąłem niepotrzebnie wyrostka; nie wiem, czy z niedbalstwa, niewiedzy, ciekawości czy po prostu rutyny.

Przekonało mnie to do znaczenia praktyki. Nie sposób inaczej zapoznać się z dowolną niemal dziedziną niż przez praktykę. Dotykając rzeczy samodzielnie, biorąc je pod lupa, lepiej zapamiętuje się również wiadomości teoretyczne, jakie się już posiada.

Mogę podać jeszcze jeden przykład. W roku 2000 uczestniczyłem w szkoleniach Microsoftu z zakresu administrowania serwerem baz danych i programowania baz danych. Szkolenie szło świetnie, a wykładowca dokładał starań, aby wyczerpująco zaprezentować materiał. Szkolenie miało jednak formę kursu teoretycznego z minimalną ilością ćwiczeń praktycznych. W efekcie jego uczestnicy (również ja) wiedzieliśmy dobrze, co serwer potrafi. Ale w czasie późniejszej pracy, kiedy napotykałem konkretne problemy, nie bardzo wiedziałem, jak zrobić to czy tamto. Na szkoleniu poznałem w całości podręcznik obsługi serwera, ale zaledwie kilka przykładów rozwiązań praktycznych problemów. Życzębym sobie, aby szkolenie obejmowało raczej prezentację sposobów, a nie teoretycznych możliwości. W mojej opinii szkolenie było stratą czasu.

Nie jestem przeciwnikiem nauczania teorii i nie mam zamiaru dowodzić, że jest ona zbędna. Uważam jednak, że uczniowie i studenci powinni być również zaopatrywani w umiejętności wykonywania konkretnych zadań, a przy tej okazji należy wyjaśniać, dlaczego rozwiązanie obejmuje takie, a nie inne czynności. Wtedy uczniowie czy studenci dowiedzą się, jak rozwiązywać praktyczne problemy i osiągać zamierzone efekty.

Trudno w księgarniach znaleźć książki z takim naciskiem na praktykę albo choćby uzupełnione solidną dawką konkretnych przykładów. Mam nadzieję, że mój tekst okaże się choć trochę pomocny w Twojej przyszłej pracy i w implementacjach zadań programistycznych.

Podsumowując, książka zawiera szereg przykładów i pomimo swego tytułu powinna okazać się przydatna szerokiemu gronu Czytelników, opisuje bowiem szeroki wachlarz metod programistycznych, które można z powodzeniem wykorzystywać w codziennej pracy.

Wszystkie przykłady prezentowane w książce zostały zamieszczone również w katalogu *Przykłady* na dołączonej do książki płycie CD-ROM. Zalecałbym jednak ograniczanie korzystania z ich gotowej postaci i odwoływanie się do gotowych plików jedynie w ostateczności, celem porównania kodów i wykrycia błędów. Samodzielna praca znacznie lepiej włącza informacje do pamięci i pozwala na ich skuteczniejsze zapamiętanie niż lektura setek stron opracowań teoretycznych. Właśnie dlatego w mojej książce jest tak wiele przykładów uzupełnionych kodem źródłowym.

Jeśli zdecydujesz się na skorzystanie z gotowego przykładu, upewnij się, że rozumiesz w pełni jego działanie. Spróbuj zmienić niektóre parametry programu i sprawdź ich wpływ na jego pracę. Spróbuj zmodyfikować przykład, dodając do niego własne funkcje. To jedyny sposób na opanowanie zasad działania funkcji i algorytmów stosowanych w przykładach.

Kim jest haker? Jak zostać hakerem?

Zanim zaczniemy, chciałbym „załadować” pamięć Czytelnika dawką podstawowych informacji teoretycznych. Konkretnie chciałbym, aby przed podjęciem lektury wiadomo było, кто to taki „haker”. Jeśli nadasz temu słowi znaczenie inne od mojego, nie zrozumiemy się.

Na wiele pytań ludzie nie potrafią odpowiedzieć poprawnie. Co więcej, ludzie po-wszechnie używają słów, których znaczenie nie jest dla nich do końca jasne. Wiele osób nie ma, na przykład, świadomości właściwego znaczenia słowa „haker”. Wielu uważa swoje pojmowanie za najwłaściwsze. Nie będę zmuszał nikogo do przyjęcia mojego punktu widzenia, moim zdaniem należy je jednak uznać za poprawne.

Przed zagłębiением się w niuanse znaczeniowe warto przypomnieć historię komputeryzacji sprzed pojawiения się internetu.

Słowo „haker” pojawiło się wraz z rozprzestrzenieniem się pierwszej publicznej sieci rozległej — ARPAnet. Wówczas słowo to odnosiło się do eksperta komputerowego. Niektórzy uważali, że „haker” to fanatyk komputerów. Słowo to zostało skojarzone z hobbystą, którego celem jest poznanie wszystkiego, co związane jest z jego pasją — komputerami. Owa chęć poznania wraz z przemożną potrzebą wymiany informacji spowodowała gwałtowny rozwój internetu. Hakerzy, zjednoczywszy siły, powołali do życia sieć FIDO. Stworzyli też podobne do uniksowych systemy operacyjne bazujące na otwartym kodzie, które dziś obsługują znaczną część serwerów internetowych.

W owych pionierskich czasach nikt nie słyszał o wirusach i próbach włamania do sieci czy pojedynczych komputerów. Ta rysa na wizerunku hakerów pojawiła się dopiero później. Jednak nie powinna ona przesłaniać właściwego obrazu. Prawdziwy haker nie ma interesu we włamywaniu się do cudzych zasobów, a jeśli jego celem jest pu-stoszenie, nie zyskuje uznania w społeczności hakerów.

Prawdziwy haker jest twórcą, nie niszczycielem. Ponieważ twórców było znacznie więcej niż niszczycieli, tych ostatnich hakerzy zaczęli nazywać krakerami albo wandalami czy chuliganami. Tak hakerzy, jak i krakerzy są w świecie komputerów prawdziwymi specjalistami. Obie grupy opowiadają się za wolnym dostępem do informacji i technologii, jednak tylko krakerzy włamują się do cudzych baz danych i innych zasobów dla pieniędzy albo sławy. Są zwykłymi kryminalistami.

Haker to ekspert w swojej dziedzinie. Mówię o specjaliste, który swoją branżę zna od podszewki, tworzy w niej rzeczy nowe, własne i niepowtarzalne, i dzieli się swoją wiedzą z innymi. Każdy, kto go posądza o vandalizm, jest w błędzie. Prawdziwi hakerzy nie wykorzystują swojej wiedzy w celu zaszkodzenia innym; ja zachęcam w swojej książce do hakerstwa konstruktywnego, dając Czytelnikom jedynie pomocne i ciekawe informacje, wzbogacające wiedzę i — jeśli są odpowiednio wykorzystane — zwiększącą jakość programów.

Przejdzmy zatem, jak zostać hakerem.

◆ Należy poznać swój komputer i nauczyć się efektywnie go kontrolować.

Dużym plusem będzie znajomość sprzętu. Co oznacza efektywna kontrola nad komputerem? Chodzi o znajomość wszystkich możliwych metod wykonania danej czynności i dobieranie ich odpowiednio do sytuacji. W szczególności warto znać stosowne skróty klawiaturowe i nie używać do wszystkiego myszy — z klawiatury można korzystać znacznie efektywniej. Warto się zmusić do korzystania z niej, to się naprawdę opłaca. Osobiście staram się korzystać wyłącznie z klawiatury, do myszy uciekając się tylko w ostateczności.

Prosty przykład: mój szef do kopiowania i wklejania danych do i ze schowka systemowego wykorzystuje zawsze przyciski paska narzędziowego albo pozycje menu rozwijanych (pojawiających się po kliknięciu kopiowanego obiektu prawym przyciskiem myszy). Jednak nie wszystkie aplikacje posiadają stosowne menu albo przyciski na pasku narzędziowym. W takich przypadkach mój szef przepisuje tekst ręcznie. Tymczasem mógłby skorzystać ze skrótów klawiaturowych *Ctrl+C* i *Ctrl+V*, ewentualnie *Ctrl+Ins* i *Shift+Ins*, które są o tyle uniwersalne, że działają praktycznie w każdej współczesnej aplikacji.

Kopiowanie i wklejanie danych w standardowych komponentach interfejsu Windows (np. polach tekstowych) to zadanie systemu operacyjnego, a implementacja tych operacji w programie nie wymaga od jego twórcy pisania żadnego kodu. Jeśli nawet autor programu nie przewidział stosownego przycisku czy pozycji menu, nie oznacza to, że zablokował możliwość wykonywania tych operacji. Są one ciągle dostępne za pośrednictwem skrótów klawiaturowych.

◆ Warto dowiedzieć się jak najwięcej o komputerach, przynajmniej o ich obecności w dziedzinie, która Cię interesuje. Osoby zainteresowane grafiką komputerową powinny zapoznać się z najlepszymi pakietami graficznymi, poznać techniki konstruowania prostych scen i złożonych światów. Osoby zainteresowane sieciami komputerowymi powinny zdobyć jak najwięcej informacji o sieciach. Ktoś, kto uważa, że wie już dosyć, powinien kupić jedną z grubszych książek ze swojej dziedziny, a przekona się, że jest w błędzie. Komputery i technologie informatyczne to branża, w której nikt nie wie wszystkiego.

Hakerzy są ekspertami w swoich dziedzinach. Nie muszą one być zresztą związane ściśle z komputerami czy programowaniem. Można być hakerem w dowolnej dziedzinie; my mówimy tu jednak o hakerach-komputerowcach.

- ◆ Warto nauczyć się programować. Każdy haker powinien znać przynajmniej jeden język programowania. Najlepiej byłoby znać ich kilka. Zalecałbym rozpoczęcie nauki programowania od języka Borland Delphi albo C++. Borland Delphi to środowisko bardzo proste, szybkie i efektywne. Co ważniejsze, bardzo bogate. C++ to z kolei powszechny światowy standard, choć jego opanowanie może okazać się trudniejsze. Zresztą nie trzeba ograniczać się do jednego z tych dwóch języków. Przydatna i pożądana będzie choćby znajomość języka Basic (nie zalecałbym go jako podstawowego języka programowania, ale jego znajomość w niczym nie przeszkodzi hakerowi, może jedynie pomóc).

Choć osobiście nie lubię Visual Basica, dostrzegając niewygodę jego stosowania i rozmaite inne słabości, widziałem już wiele znakomitych aplikacji napisanych w tym języku. Ich autorów, którzy wykonali znakomitą robotę, śmiało nazwałbym hakerami. Stworzenie dzieła z niczego to sztuka bliska jedynie hakerom.

Haker to osoba, która coś tworzy. W większości przypadków rolę dzieł pełni kod programów, ale również znaczącym efektem zaangażowania w danej dziedzinie może być kompozycja muzyczna albo graficzna. To również dzieła godne hakera. Zresztą nawet kompozytorowi korzystającemu z komputera przydałaby się umiejętność programowania. Współcześnie tworzenie programów to rzecz nieco prostsza niż niegdyś. Przy zastosowaniu języków takich jak Borland Delphi można szybko tworzyć użyteczne narzędzia. Nie bądź więc leniwy i naucz się programować.

W książce powiem Czytelnikom, co powinien umieć programista-haker. Pokażę też szereg interesujących technik programistycznych i przykładów ich wykorzystania w języku C++. Jeśli nie znasz jeszcze tego języka, ta książka pomoże Ci się go nauczyć.

- ◆ Nie stój w poprzek nurtu postępu, ale nie daj się wciągnąć wirom. Hakerzy od zawsze propagowali wolność dostępu do informacji. Jeśli chcesz być hakerem, powinieneś pomagać innym. Haker powinien promować postęp technologiczny. Może to czynić, pisząc programy i udostępniając je za darmo albo w inny sposób dzieląc się swoją wiedzą.

Powszechność dostępności informacji nie oznacza niemożności zarobkowania na jej przetwarzaniu. Nie można obwiniać hakera za jego ambicje finansowe. Hakerzy potrzebują pieniędzy na utrzymanie swoje i swoich rodzin. Pieniądze nie są jednak w ich życiu najważniejsze. Ważna jest kreatywność. To kolejna różnica pomiędzy hakerami a krakerami. Hakerzy tworzą informację, a krakerzy ją niszczą. Jeśli napiszesz własny żartobliwy program i podeślesz go dla kawału kolegom, będziesz hakerem. Jeśli napiszesz wirusa, który usunie informacje z dysków kolegów, będziesz krakerem — zwykłym (nie zawaham się przed tym określeniem) kryminalistą.

Optując za dostępem do informacji, haker może się włamywać, ale jedynie w sposób nieniszczący. W etyce hakerów dozwolone jest złamanie programu

celem poznania sposobu jego działania, niedozwolone jest natomiast usuwanie zabezpieczeń przed kopiowaniem. Należy szanować pracę innych i nie naruszać ich praw autorskich.

Wyobraź sobie, że ktoś ukradł ze sklepu telewizor. Jako złodziej powinien być ścigany i zostać ukarany. To powszechna norma i większość ludzi unika napiętowania i kary, powstrzymując się od kradzieży. Skąd więc taka zuchwałość u krakerów łamiących programy bez strachu przed konsekwencjami? To przecież również kradzież. Osobiście nie rozróżniam włamań do sieci i programów od włamań do sklepów. Obie czynności uważam za naganne.

Jestem programistą i sprzedaję swoje programy. Nigdy nie opatruję ich skomplikowanymi systemami zabezpieczeń przed kopiowaniem, ponieważ stanowią one utrudnienia dla zwykłych użytkowników, a krakerom i tak nie sprawiają większych trudności. Wielkie korporacje inwestują spore pieniądze w mechanizmy zabezpieczające ich majątkowe prawa autorskie, ale większość ich aplikacji jest łamana jeszcze przed ich oficjalnym wprowadzeniem na rynek. Uważam, że systemy zabezpieczeń oparte na kluczu aktywacyjnym nie mogą być skutecznym zabezpieczeniem przed kopiowaniem.

W cywilizowanym świecie program powinien udostępniać pole służące do wpisywania kodu potwierdzającego legalność nabycia programu. Program nie powinien zaś wymagać aktywacji czy brnięcia przez procedury rejestracyjne. Użytkownicy powinni być uczciwi, wiedząc, że za każdą pracę należy się płacić. Jeśli dobro (tu produkt programowy) może zostać zdobyte za darmo, nie oznacza to zniesienia obowiązku uiszczenia opłaty za jego wykorzystywanie.

◆ Nie wyważaj otwartych drzwi. Hakerzy nigdy nie stają w miejscu i zawsze dzielą się swoimi osiągnięciami i wiedzą. Jeśli więc napiszesz własny, niepowtarzalny kod, powinieneś udostępnić go innym, aby nie musieli męczyć się nad taką samą funkcją czy programem. Nie warto z własnych osiągnięć robić tajemnic, warto zaś pomagać innym.

Mając do dyspozycji kod cudzego autorstwa, czuj się upoważniony do jego wykorzystywania (jeśli jego autor na to zezwolił). Nie twórz od podstaw rzeczy gotowych. Gdyby wszyscy chcieli samodzielnie wynaleźć koło, nikt nie doszedłby do konstrukcji wózka.

◆ Hakerzy tworzą społeczność. Nie oznacza to, że wszyscy ubierają się czy wyglądają podobnie. Każdy jest indywidualistą. Nie próbuj naśladować innych osób. Nie uczyni Cię to hakerem. Markę wyrobisz sobie jedynie samodzielnie.

Jeśli w pewnych kręgach zostaniesz zauważony, traktuj to honorowo. Hakerzy to osoby sławne ze swych umiejętności i odpowiedniej postawy. Warto, aby wszyscy byli sławni.

W którym momencie można zaliczyć się do społeczności hakerów? Odpowiedź na to pytanie jest prosta — jeśli mówią o Tobie „haker”, jesteś hakerem. Tutaj pojawia się pewna trudność wynikająca z pomieszanego pojęć — wielu ludzi nazywa hakerami osoby łamiące oprogramowanie i włamujące się do cudzych

sieci. Należy jednak oprzeć się pokusie takiego pozyskania rozgłosu i poszukiwać raczej sławy „dobrej” niż „złej”. To co prawda bardzo trudne, ale nie sposób nic na to poradzić. Nikt nie obiecuje, że rzecz będzie prosta.

◆ Jak rozróżnić programistę, użytkownika i hakera? Programista piszący program ma świadomość celu i implementuje własne pomysły. Użytkownik nie zawsze wie, jakim celom służy program i używa go zgodnie z własnym pojęciem jego przeznaczenia.

Programista nie może przewidzieć poczynań klienta, a programy nie zawsze mogą być wszechstronnie przetestowane. Użytkownicy mogą wprowadzać do programu takie parametry działania, które czynią go niestabilnym, czynią to jednak przypadkowo.

Hakerzy świadomie wyszukują dziury w programach, zmuszając je do nieoprawnego albo innego od zamierzonego działania.

Wymaga to oryginalnego podejścia i sporej wyobraźni. Haker powinien wyczuwać konstrukcję programu i widzieć rzeczy ukryte przed innymi.

Jeśli chcesz być hakerem, powinieneś zdobyć umiejętność czynienia rzeczy prostych ciekawymi i zajmującymi. Musisz do tego zaangażować własną wyobraźnię. W tej książce będziesz miał okazję zobaczyć co najmniej kilka nietypowych perspektyw powszechnie znanych rzeczy, pomocnych w pisaniu własnych programów i projektowaniu efektywnych algorytmów.

Na koniec poleciłbym lekturę *How to Become a Hacker* Erica S. Raymonda — uznanego i szanowanego hakera. Z niektórymi jego tezami można dyskutować, ale jego artykuł z pewnością oddaje ducha hakingu.

Zastanawiasz się zapewne, dlaczego tworzenie dowcipów programistycznych i małych programów sieciowych jest dla mnie przyczynkiem hakingu. Cóż, po pierwsze, hakerzy zawsze demonstrują swoje umiejętności za pośrednictwem niebanalnych i przeważnie rozrywkowych programów. Z tej kategorii wykluczamy wirusy, ponieważ, choć zwyczaj ciekawe, są równocześnie destrukcyjne. Przez swoje nieszkodliwe żarty haker zdradza poziom wiedzy i znajomość systemu operacyjnego, równocześnie wywołując uśmiech na twarzach kolegów. Wielu hakerów cechuje się niezłym poczuciem humoru, które warto jakoś wykorzystać. Zalecam kanalizowanie potrzeby dowcipkowania właśnie w procesie tworzenia zabawnych programików.

Co do programów sieciowych, to nie sposób oddzielić obecności w sieci od bycia hakerem. Społeczność hakerów jest dziś tak liczna właśnie dzięki sieciom. Hakerzy wniesli przy tym spory wkład w rozwój technik komunikacji i internetu. To dlatego tak wiele miejsca poświęcam programowaniu sieciowemu, choć prezentuję jego zagadnienia z dość niestandardowego punktu widzenia.

I jeszcze słówko: wspomniałem już, że haker powinien legitymować się umiejętnością pisania programów w dowolnym, wybranym przez siebie języku programowania. Niektórzy sądzą, że haker koniecznie musi znać język asemblerowy. To nie jest wymóg konieczny. Znajomość tego języka byłaby z pewnością pożądana, ale nie jest obowiązkowa. Osobiście lubię programować w Delphi firmy Borland. Środowisko nie ogranicza moich poczynań, pozwalając na implementację wszystkiego, co sobie wymyślę. Mimo to korzystam również z C++ i języka asemblerowego.

Języków programowania należy używać roztropnie. Mimo swej atrakcyjności do Delphi nie skorzystałbym z niego przy pisaniu gry komputerowej. To od zawsze była i zawsze będzie domena języków C i C++. W programowaniu sieciowym czasem wygodniej korzystać z C++ niż Delphi, czasem poręczniejsze okazuje się Delphi. Na pewno zaś pisanie większych programów w języku asemblerowym nie jest ani wygodne, ani efektywne.

Należy też pamiętać o znaczeniu technologii. Nawet przed ukończeniem studiów wiedziałem, że klient ma zawsze rację. Co ciekawe, zasada ta nie obowiązuje w przemyśle komputerowym.

Oto prosty przykład: współcześnie wielu programistów implementuje w swoich aplikacjach obsługę języka XML niezależnie od tego, czy taka obsługa jest aby w danym zastosowaniu konieczna. W rzeczy samej mało kto korzysta z tego formatu i nie trzeba go obsługiwać w każdym programie. Podążanie w tym zakresie za zaleceniami firmy Microsoft nie jest oczywiste, ponieważ programy piszemy dla klientów, a nie Billa Gatesa. Programista powinien zawsze spełniać wymagania przyszłego użytkownika jego dzieła.

Firma Microsoft regularnie wtłacza na rynek nowości, a kiedy towarzyszą im nowe technologie, programiści spieszają się poprawiać swoje programy tak, aby były zgodne z nowymi cechami proponowanymi przez Microsoft. W efekcie znaczną część czasu programistów pochłaniają aktualizacje i modyfikacje oprogramowania. Jeśli Twój program odwołuje się do bazy danych za pośrednictwem interfejsu DAO, nie trzeba koniecznie uzupełniać go o obsługę ADO. Użytkownicy nie zwracają uwagi na sposób transferu danych z bazy danych — chodzi im tylko o to, aby transfer został dokonany.

Kolejny przykład dotyczy dziedziny interfejsu użytkownika. Interfejs pakietu MS Office wciąż ulega zmianom, przy czym kolejne jego wcielenia są reklamowane jako najwygodniejsze z możliwych. Programiści, naśladowując tak doskonałe interfejsy, spieszają się modyfikować swoje pomysły, choć np. Internet Explorer i inne aplikacje firmy Microsoft od lat wyglądają tak samo. W ich wyglądzie nic się nie zmienia, bo zmiany oznaczająby marnotrawstwo zasobów. W konkurencji mamy natomiast pęd do aktualizacji zgodnie z najnowszymi trendami, wymagającej niekiedy miesięcy zbędnej pracy.

Zgoda, wypada podążać za trendami, ponieważ moda obowiązuje również w świecie komputerów. Warto jednak postarać się o utrzymanie indywidualnego stylu.

Programiści i hakerzy swymi poglądami i wypowiedziami narzucają innym pogląd co do wyższości jednych języków programowania nad innymi. Zwykle dają radę przekonać odbiorców swoich programów co do swoich racji, opierając się na nieznajomości zagadnień programowania u tych ostatnich. A przecież to, w jakim języku powstała aplikacja, jest dla klienta najmniej ważne — liczy się dla niego jedynie jej jakość. Co do mnie, to przy ścisłych wymogach czasowych potrafię zachować wysoką jakość oprogramowania jedynie korzystając z Borland Delphi. Utrzymanie podobnej jakości oprogramowania w C++ wymaga od programisty (nie jestem tu wyjątkiem) znacznie większych nakładów.

W niniejszej książce wszystkie programy przykładowe napisane zostały w języku Visual C++, jako że jest to język najpopularniejszy i traktowany jako de facto standard programistyczny.

Kiedy odbiorca mojego programu kładzie nacisk na jego zwartość albo szybkość działania, korzystam z asemblera albo języka C (nie mylić z C++). Zdarza się to jednak rzadko, ponieważ współczesne komputery są milion razy szybsze niż pierwotne konstrukcje. Rozmiar i szybkość programów straciły na znaczeniu, wartości nabrala za to ich jakość.

Na tym zakończę wprowadzenie. Przejedziemy teraz do przeglądu praktycznych ćwiczeń w sztukach, w których priorytetem jest zachowanie tajności i wydatkowanie możliwie małych ilości energii.

Rozdział 1.

Jak uczynić program zwartym, a najlepiej niewidzialnym?

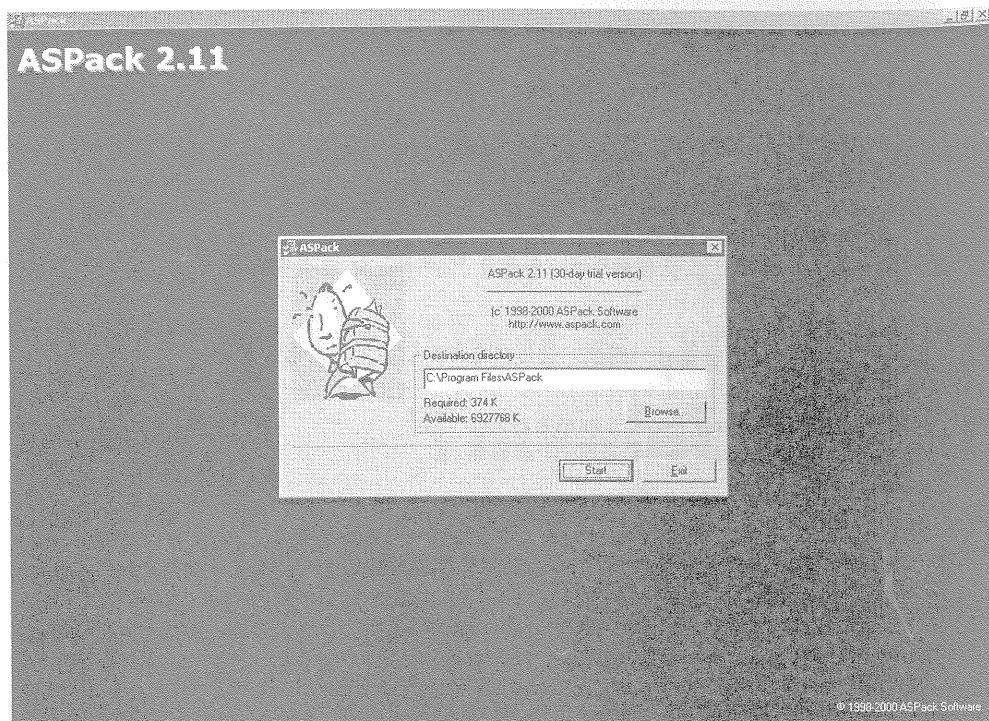
Co jest najważniejsze przy tworzeniu programów-dowcipów? Oczywiście ich niewidzialność. Programy tworzone w tym i następnych rozdziałach będą się ukrywały w systemie i ograniczą do wykonywania pewnych czynności w obliczu ściśle określonych warunków. Oznacza to, że program, choć uruchomiony, nie powinien się pojawić ani na pasku zadań, ani na liście pojawiającej się w oknie wyświetlanym po naciśnięciu kombinacji klawiszy *Ctrl+Alt+Del*. Zanim zaczniemy programy pisać, musimy się dowiedzieć, jak je ukryć w systemie.

Ponadto dowcipi programistyczne nie powinny przybierać zbyt dużych rozmiarów. Aplikacje tworzone w Visual C++ z wykorzystaniem zaawansowanych bibliotek, jak MFC, są raczej „ciężkawe”. Nawet prosty program, wyświetlający pojedyncze zaledwie okno, może zająć na dysku znaczną ilość miejsca. Gdyby taki żart miał być propagowany za pośrednictwem poczty elektronicznej, jego pobieranie i wysyłanie byłoby czynnością bardzo czasochłonną. Jak uniknąć przerostu programu pisaneego w Visual C++? Poświęcimy temu pierwszy rozdział.

1.1. Kompresowanie plików wykonywalnych

Najprostszym sposobem zmniejszenia rozmiaru pliku programu jest zastosowanie narzędzia kompresującego. Osobiście często korzystam i lubię program ASPack. Można go pobrać z witryny <http://www.aspack.com>; jego kopia znajduje się również na płycie CD-ROM dołączonej do książki w folderze *Programy* (nazwa pliku instalacyjnego to *ASPack.exe*). Program znakomicie kompresuje pliki wykonywalne EXE i DLL.

Uruchom plik *ASPack.exe*, a zobaczysz okno instalacji (jak na rysunku 1.1). W oknie tym określ ścieżkę folderu docelowego i kliknij przycisk *Start*. W ciągu kilku sekund program zostanie zainstalowany i uruchomiony.



Rysunek 1.1. Okno instalacji programu ASPack

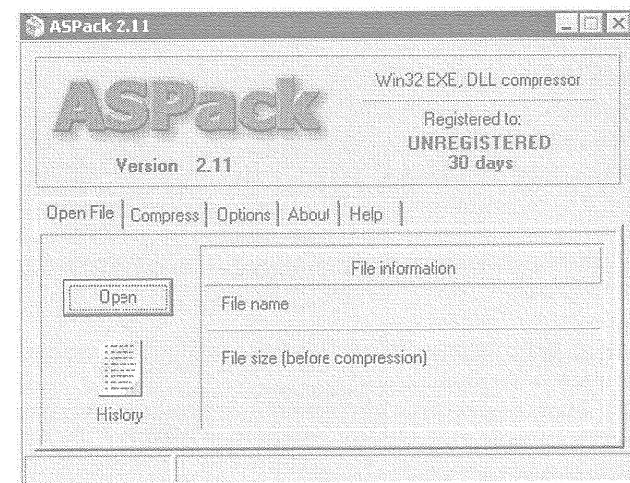
Główne okno programu (rysunek 1.2) zawiera następujące zakładki:

- ◆ *Open File*
- ◆ *Compress*
- ◆ *Options*
- ◆ *About*
- ◆ *Help*

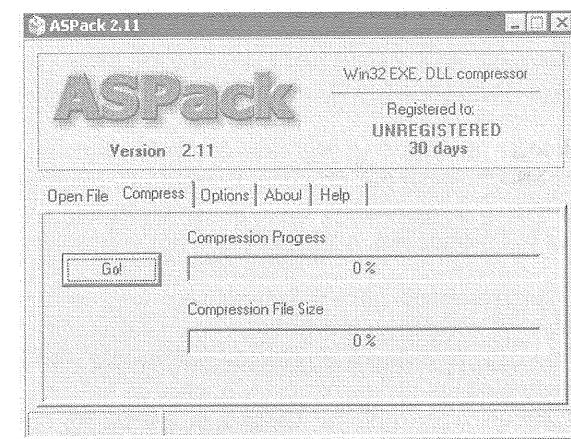
Na zakładce *Open File* znajduje się tylko jeden przycisk — *Open*. Po jego kliknięciu należy wskazać plik, który ma zostać skompresowany. Po wskazaniu pliku program automatycznie przejdzie na zakładkę *Compress* i rozpoczęcie proces kompresji (rysunek 1.3).

Skompresowany plik zastąpi oryginał, ale ten ostatni zostanie na wszelki wypadek zapisany z rozszerzeniem BAK. Można zresztą wyłączyć tworzenie kopii zapasowej, jednak nie zalecałbym tego. Za chwilę zajmiemy się szczegółowo opcjami programu.

Rysunek 1.2.
Główne okno
ASPack



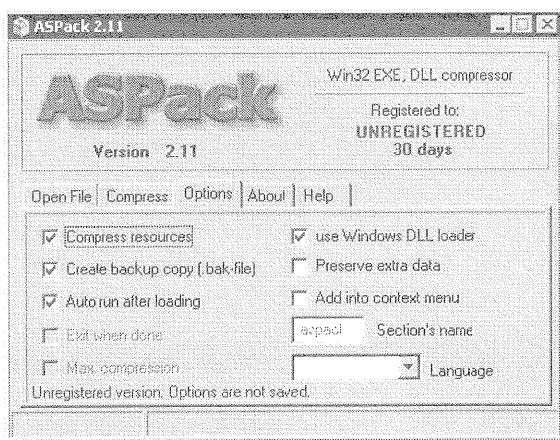
Rysunek 1.3.
Zakładka kompresji
pliku



Działaniem programu ASPack steruje zaledwie kilka parametrów; są one zgrupowane na zakładce *Options* (rysunek 1.4). Oto one:

- ◆ *Compress resources* (kompresuj zasoby) — jeśli program korzysta z MFC (Microsoft Foundation Classes) i przechowuje okna dialogowe i bitmapy jako zasoby, plik wykonywalny zawiera obszerną sekcję zasobów. Z mojego doświadczenia wynika, że większość sekcji zasobów zajmują obrazy — nie są one kompresowane. Po zaznaczeniu tego pola ASPack będzie je kompresował.
- ◆ *Create a backup copy* (twórz kopię zapasową) — ASPack przed przystąpieniem do kompresji utworzy kopię zapasową kompresowanego pliku. Kopią ta zapisywana jest w tym samym folderze i pod oryginalną nazwą, ale z rozszerzeniem BAK zamiast EXE. Na przykład kompresując plik o nazwie *myprogram.exe*, jego kopią zapasowa będzie nosiła nazwę *myprogram.bak*.

Rysunek 1.4.
Ustawienia
programu ASPack



Zalecałbym włączanie opcji wykonywania kopii zapasowej, ponieważ ASPack jest czasem niestabilny i po wprowadzeniu pewnych wartości parametrów może zniszczyć kompresowany plik. W takim przypadku będzie można wyeliminować uszkodzenia, zmieniając rozszerzenie BAK z kopii zapasowej na EXE. Można pokusić się o wyłączenie tej opcji; zalecam jednak wtedy wykonywanie kopii kompresowanego pliku we własnym zakresie.

Jeśli masz do dyspozycji pełny kod źródłowy programu, możesz przywrócić uszkodzony podczas kompresji plik do stanu używalności, inicjując komplikację kodu źródłowego. Jeśli jednak kompresujesz cudzy program, to bez kopii zapasowej jego odtworzenie może być niemożliwe. A nie warto przecież przysparzać sobie dodatkowych kłopotów.

Po skompresowaniu pliku wykonywalnego warto przetestować jego zdolność do uruchomienia. W większości przypadków uruchomienie przebiegnie bezproblemowo i można założyć, że żadnych problemów nie będzie również w przyszłości. Zdarzają się czasem błędy w niektórych oknach, choć są one rzadkie. Warto po skompresowaniu wyczerpująco przetestować działanie programu, aby nie udostępnić nabywcy uszkodzonej wersji. W projektach komercyjnych błędów takich należy unikać jak ognia.

- ◆ *Auto run after loading* (rozpoczynaj po załadowaniu) — jeśli to pole jest zaznaczone, ASPack będzie automatycznie rozpoczynał kompresowanie po wskazaniu pliku na zakładce *Open File*.
- ◆ *Exit when done* (wyjdź po zakończeniu) — nie trzeba objaśniać.
- ◆ *Max compression* (kompresja maksymalna) — za pośrednictwem tego parametru można zmniejszyć rozmiar pliku wynikowego, niestety, kosztem zwiększenia ryzyka wystąpienia błędów. Po włączeniu maksymalnej kompresji należy szczególnie starannie testować działanie skompresowanego pliku wykonywalnego i w razie jakichkolwiek błędów wycofać opcję maksymalnej kompresji.
- ◆ *Use a Windows DLL Loader* (zastosuj program ładujący DLL z Windows) — mamy do wyboru dwa programy ładujące (loadery): standardowy program ładujący Windows i program ładujący optymalizowany dla wcześniejszych kompilatorów Borland C++. Jako że piszemy programy w MS Visual C++, powinniśmy zaznaczyć przy ich kompresowaniu pole *Use Windows DLL Loader*.

- ◆ *Preserve extra data* (zachowaj dane dodatkowe) — niektóre programy zawierają dodatkowe dane umieszczone na końcu pliku wykonywalnego. Jeśli program ASPack spróbuje je skompresować, mogą się stać niedostępne w programie. Przykładem takich programów są pliki wykonywalne programów instalacyjnych. Zawierają one kod programu uzupełniony dodatkowymi danymi, które powinny zostać zainstalowane w komputerze. Przeważnie danych takich nie wolno kompresować.

Przyjrzyjmy się teraz samej kompresji. Na początku całość programu wykonywalnego podlega przetworzeniu odpowiednim kompresorem. Jeśli sądzisz, że stosowany kompresor to coś wyszukanego, jesteś w błędzie. W programie ASPack stosowany jest zwykły kompresor optymalizowany dla kodu binarnego. Po drugie, do spakowanego kodu, na jego końcu, dołączany jest kod dekompresora. Dekompresor w momencie uruchomienia programu zajmie się rozpakowaniem jego właściwego kodu. Na koniec ASPack modyfikuje nagłówek pliku wykonywalnego, wymuszając uruchomienie dekompresora.

Algorytm kompresji programu ASPack jest bardzo skuteczny, a dekompresor bardzo mały (zajmuje poniżej 1 kB). Dzięki temu kompresja daje dobre rezultaty, przy konieczności zwiększenia rozmiaru pliku wynikowego o zaledwie 1 kB. Przy 1,5-megabajtowym pliku wykonywalnym można dzięki temu osiągnąć rozmiar wynikowy rzędu 300 czy 400 kilobajtów.

Przy uruchomieniu skompresowanego programu jako pierwszy startuje dekompresor. Ten rozpakowuje kod binarny do pamięci komputera. Po zakończeniu rozpakowywania przekazuje sterowanie do właściwego programu.

Niektórzy sądzą, że aplikacja skompresowana będzie działać wolniej, z racji narzutu dekompresora. Tymczasem najczęściej nie daje się odczuć żadnej różnicy w prędkości działania programu skompresowanego i nieskompresowanego. Jeśli jakieś spowolnienie faktycznie występuje, jest raczej niezauważalne (przynajmniej na stosunkowo nowoczesnych komputerach). Algorytm pakujący jest optymalizowany dla kodu binarnego, a dekompresja odbywa się tylko raz, po załadowaniu programu do pamięci. Spowolnienie w żadnym razie nie może być więc odczuwalne po uruchomieniu właściwego programu.

Skompresowany czy nie, przed uruchomieniem każdy program musi zostać załadowany do pamięci. Jeśli jego kod jest spakowany, to podczas lądowania następuje jego rozpakowanie. Tutaj mamy do czynienia z dwiema kwestiami: czasem potrzebnym na rozpakowanie programu, ale z drugiej strony przyspieszeniem lądowania z racji mniejszego rozmiaru spakowanego kodu. Należy przy tym pamiętać, że dysk twardy to najwolniejszy element komputera. Im mniej jest danych do odczytania z dysku i załadowania do pamięci, tym szybciej program zostanie uruchomiony. Jak widać, kompresja może wcale nie spowalniać uruchamiania programu.

W zwykłym programowaniu aplikacji, czyli programowaniu z wykorzystaniem gotowych bibliotek komponentów wizualnych i obiektowych, plik wynikowy programu jest zazwyczaj spory. Po zastosowaniu specjalnego kompresora można go zmniejszyć do 60, 70 procent pierwotnego rozmiaru. Zachowujemy wtedy wygodę programowania, nie tracąc na zwiększym rozmiarze pliku wykonywalnego.

Kolejną zaletą kompresji programów wykonywalnych jest to, że skompresowany kod jest znacznie trudniejszy do złamania niż kod nieskompresowany. Mało który deassembler potrafi dekodować instrukcje maszynowe w pliku spakowanym. Kompresja daje więc — oprócz zmniejszenia rozmiaru pliku programu — pewne zabezpieczenie przed próbami złamania kodu programu. Szczerze mówiąc, profesjonalista da radę złamać program spakowany, ponieważ będzie miał do dyspozycji dostępne w internecie programy rozpoznające fakt spakowania pliku wykonywalnego i dekompresujące taki plik. Jednak włamywacz-amator będzie miał z takim programem spory kłopot i raczej szybko zniechęci się do deasemblieracji spakowanego kodu programu.



Na dołączonej do książki płyce CD-ROM, w folderze \Przykłady\Rozdział1\Rysunki1, znajdują się pliki rysunków z tego rozdziału w wersji barwnej.

1.2. Ani okna, ani drzwi...

Klucz do kolejnej metody zmniejszania rozmiaru programu tkwi w pytaniu: „Dlaczego programy Visual C++ są tak duże?”. Są w tym, że C++ jest językiem obiektowym. W tym języku każdy komponent programu jest obiektem z własnymi właściwościami, danymi, metodami i zdarzeniami. Każdy obiekt jest samodzielny, w pewnej mierze samowystarczalny i potrafi wiele działać pomimo minimalnej ingerencji z zewnątrz. Wystarczy skojarzyć taki obiekt z formularzem aplikacji, ustawić odpowiednio jego właściwości i skompilować gotowy już program! Program będzie działał, mimo że nie określmy szczegółów reakcji na rozmaite zdarzenia.

Mało jest wad takiego modelu programistycznego. Obiekty implementują wiele akcji, które mógłby wykonać użytkownik albo programista, jednak w praktycznym programie liczba faktycznie wykorzystywanych właściwości tych obiektów ma się nijak do ich potencjalnych możliwości. Są one wobec tego w pewnej mierze zbędnym balastem.

Jak stworzyć zwarty kod, taki, aby program zajmował na dysku i w pamięci komputera najmniejszą możliwą ilość miejsca? Mamy do dyspozycji dwie metody:

- ◆ Nie korzystać z upraszczającej programowanie biblioteki MFC (ani biblioteki VCL w Borland Delphi). Decydując się na taki krok, trzeba przygotować się na samodzielne oprogramowanie całości aplikacji za pośrednictwem wywołań systemowego interfejsu programistycznego WinAPI. Taki program będzie zajmował mało miejsca i szybko działał. Kod wynikowy powinien być mniejszy niż stworzony z wykorzystaniem MFC i maksymalną nawet kompresją. Stracimy jednak łatwość programowania wizualnego i odczujemy wszystkie niewygody związane z korzystaniem z „gołego” API Windows. Celem dalszego zmniejszenia rozmiaru można przejść na język asemblerowy. Jest to jednak na tyle skomplikowana sprawa, że program pisany w języku asemblerowym wymaga od programisty większych nakładów niż pisany nawet w czystym języku C. Dlatego też nie będziemy w książce zajmować się językiem asemblerowym.

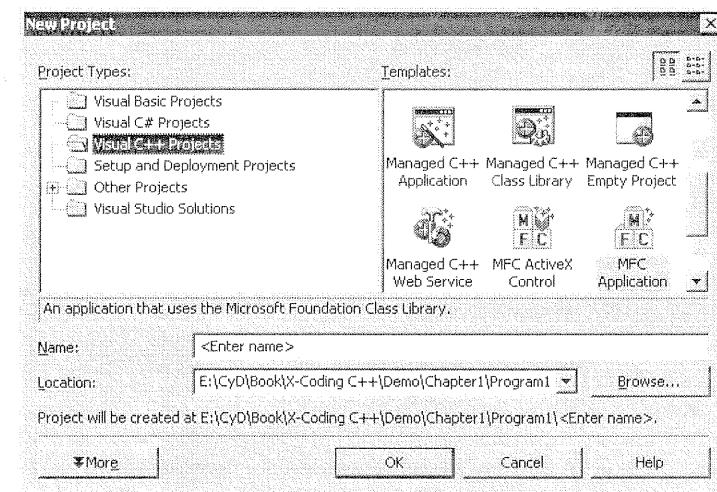
- ◆ Skompresować gotowy program. Kod obiektowy zostanie skompresowany kilkukrotnie, więc aplikacja korzystająca z MFC może z 300-kilobajtowego „potwora” zmienić się w 30- albo 50-kilobajtowy „drobiazg”. Główną zaletą tej metody jest zachowanie wygody programowania wspomaganej interfejsem graficznym i uniknięcie niewygód korzystania z WinAPI.

Drugą z metod mamy już rozpracowaną, przyjrzyjmy się więc bliżej pierwszej.

Chcąc utworzyć naprawdę mały i zwarty program, powinniśmy od razu odrzucić nadzieję na komfort programowania. Nie można w takim przypadku korzystać z graficznego wspomagania programowania i wielu użytecznych modułów powstałych z myślą o uproszczeniu życia programistom. Nie można korzystać z klas i komponentów ActiveX. Trzeba ograniczyć się do „gołego” interfejsu programistycznego Windows API.

Spróbujmy zasmakować tego na przykładzie. Uruchom Visual C++ i utwórz w nim nowy projekt. W tym wybierz z menu pozycję *File/New/Project*. Na ekranie pojawi się okno nowego projektu (jak na rysunku 1.5). Po lewej widoczna jest hierarchia typów projektu. Ponieważ piszemy w C++, wybieramy element *Visual C++ Projects*. Element ten będziemy wybierać dla każdego kolejnego przykładu omawianego w książce. W panelu *Templates* po prawej pojawią się ikony kreatorów różnych typów projektów. Wybierz *MFC Application*.

Rysunek 1.5.
Okno wyboru
rodzaju nowego
projektu

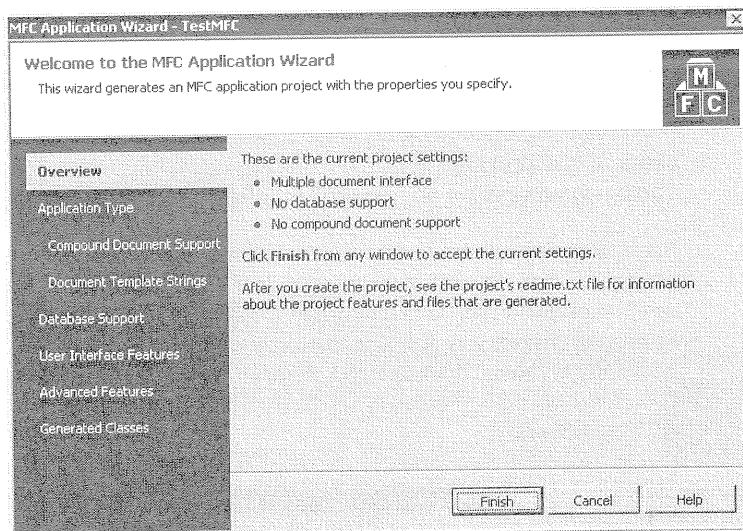


Poniżej panelu znajdują się dwa pola tekstowe. W pierwszym określa się nazwę tworzonego projektu. Będzie to nazwa ostatecznego pliku wykonywalnego oraz nazwa głównego pliku kodu źródłowego. Ustalmy ją jako *TestMFC*.

W polu *Location* należy wpisać ścieżkę dostępu do folderu, w którym mają zostać utworzone pliki projektu. Zalecamy utworzenie dla naszych celów specjalnego folderu, np. *Moje projekty C++*, i tworzenie w nim wszystkich kolejnych projektów. Wskaz ten folder i kliknij *OK*. Kiedy kreator zakończy pracę, folder *Moje projekty C++* zawierać będzie folder *TestMFC* z plikami projektu.

Po kliknięciu przycisku *OK* w oknie wyboru typu i nazwy projektu uruchamiany jest kreator nowej aplikacji MFC (rysunek 1.6). W jego oknie można nacisnąć od razu przycisk *Finish* (wtedy projekt otrzyma ustawienia domyślne), albo na kolejnych zakładkach określić własne parametry projektu. Ponieważ naszym zadaniem jest utworzenie jak najmniejszej aplikacji, spróbujemy zoptymalizować nieco ustawienia kreatora projektu.

Rysunek 1.6.
Okno kreatora
nowego projektu

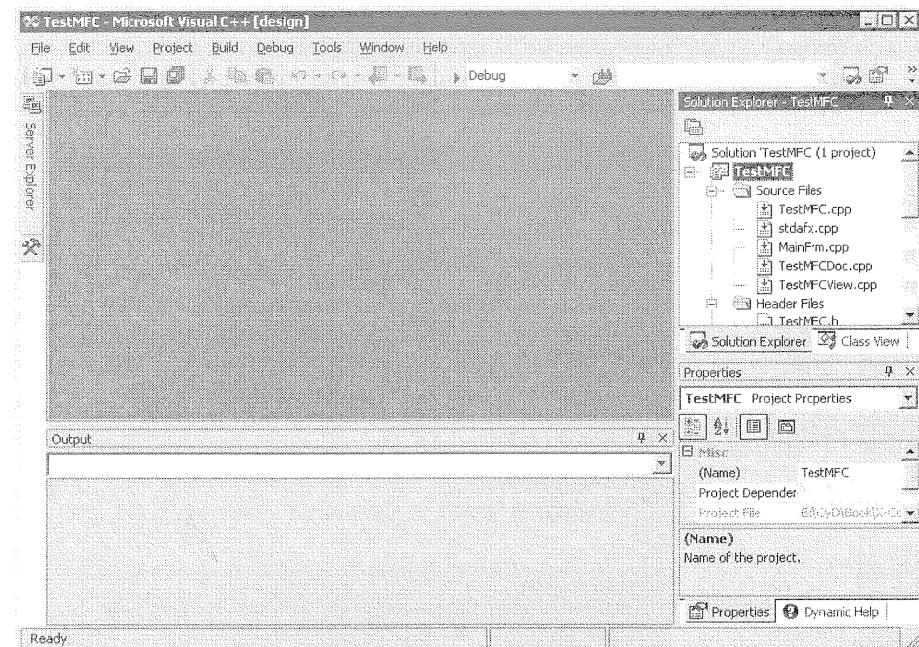


Po lewej stronie okna kreatora mamy szereg ułożonych w pionie etykiet. Wskazując kolejne etykiety, wybieramy grupy parametrów projektu. Będziemy teraz przełączać się do kolejnych grup parametrów i usuwać niepotrzebne funkcje projektu; podjęte teraz decyzje będą obowiązywać również dla przyszłych naszych projektów.

- ◆ *Application Type* (typ aplikacji) — tutaj definiujemy rodzaj tworzonej aplikacji. Ustawiamy:
 - ◆ *Single Document* (aplikacja z pojedynczym oknem dokumentu) — nasza aplikacja będzie składać się z jednego okna. Nie będziemy na razie zajmować się aplikcjami wielodokumentowymi i większość przykładów zastosowania MFC będzie bazować na aplikacji jednodokumentowej, ewentualnie na oknach dialogowych (opcja *Dialog based*).
 - ◆ *Project style* (styl projektu) — pozostawiamy styl domyślny, czyli MFC, we wszystkich aplikacjach.
 - ◆ *Document/View architecture support* (obsługa architektury dokument-widok) — to ustawienie nas nie dotyczy; pozostawiamy parametry domyślne.
- ◆ *Advanced Features* (właściwości zaawansowane) — tutaj w przyszłości ustawiać będziemy parametr *Windows sockets* (biblioteka gniazd sieciowych). Pozwoli to nam na programowanie przykładów wykorzystujących sieć.

Na pozostałych zakładkach pozostawiamy parametry domyślne — możemy sobie na to pozwolić, bo nie będziemy używać ani baz danych, ani okien dokumentów. W większości przypadków nasze programy będą się składać z pojedynczego okna i menu rozwijanego. W pierwszych przykładach spróbujemy uporać się bez MFC.

Kliknij przycisk *Finish*, aby zamknąć okno kreatora. Zobaczysz wtedy okno takie jak na rysunku 1.7. Będziemy z niego często korzystać, wypadałoby się więc z nim zapoznać. Byłoby jednak błędem omawianie teraz wszystkich elementów okna projektu, gdyż i tak nie dałoby się tego zapamiętać. Znaczenie i funkcja poszczególnych elementów stanie się oczywista w miarę, jak będziemy się do nich odwoływać w pracy.



Rysunek 1.7. Główne okno środowiska programistycznego MS Visual C++

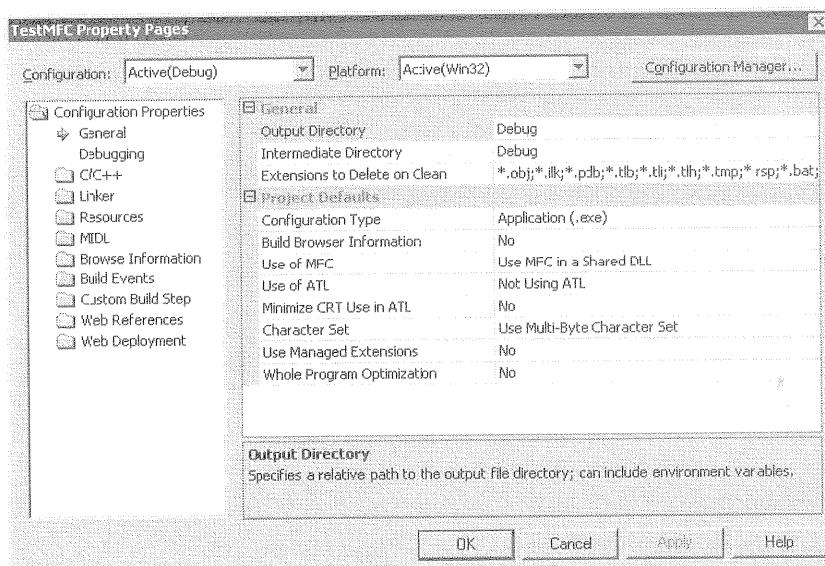
Póki co interesują nas parametry projektu. Powinniśmy wyłączyć wszystko, co mogłoby przeszkodzić w wygenerowaniu jak najmniejszego programu. Przy konsolidacji projektu pisanej w Visual C++ można wykorzystać dwa ustawienia konsolidacji: konsolidację dla celów diagnostycznych (ang. *debug*) i konsolidację ostateczną (ang. *release*). Pierwsza z nich przydaje się na etapie tworzenia programu. W tym trybie konsolidacji Visual C++ tworzy plik wykonywalny zawierający dużo dodatkowych informacji pomocnych w diagnostyce programu. W drugim trybie informacje te są z pliku wynikowego usuwane, co czyni plik wykonywalny znacznie mniejszym.

Na pasku narzędziowym mamy listę rozwijaną ustawioną na wartość *Debug*. Przełączmy ją na pozycję *Release*.

Środowisko programistyczne Visual C++ potrafi tworzyć pliki wykonywalne korzystające z jednego z dwóch rodzajów bibliotek MFC: statycznych i dynamicznych. Domyslnie włączona jest konsolidacja dynamiczna. Pozwala ona na zmniejszenie rozmiaru pliku wykonywalnego, ale tak utworzony program nie zadziała, jeśli w systemie nie będą zainstalowane biblioteki *mfcXXX.dll*, gdzie *XXX* to numer wersji MFC.

Po uaktywnieniu konsolidacji dynamicznej musimy odbiorcy programu, oprócz samego pliku wykonywalnego, udostępnić również dodatkowe pliki bibliotek. Jest to nielegantyczne i niewygodne. Przy konsolidacji statycznej plik wykonywalny będzie co prawda większy, ale za to kompletny i nadający się do samodzielnego uruchomienia bez warunku obecności dodatkowych bibliotek.

Aby zmienić typ konsolidacji biblioteki MFC, należy wskazać nazwę projektu w oknie *Solution Explorer*, a następnie wywołać prawym przyciskiem myszy menu podręczne i wybrać *Project/Properties* (właściwości). Na ekranie pojawi się okno karty właściwości projektu podobne do tego z rysunku 1.8.



Rysunek 1.8. Okno kart właściwości

Po lewej widzimy listę kategorii właściwości. Interesuje nas dział właściwości ogólnych — *General*. Zaznaczamy go i w głównym panelu okna pojawia się lista właściwości. Należy na niej odszukać pozycję *Use of MFC* i zmienić jej wartość na *Use MFC in a Static Library*. Żeby zapisać zmiany w projekcie, wystarczy kliknąć przycisk *OK*.

Spróbujmy skompilować nasz pierwszy projekt do postaci pliku wykonywalnego. W tym celu z menu *Build* wybierz *Build Solution*. U dołu głównego okna projektu, w panelu *Output* wyświetlane będą komunikaty opisujące postęp komplikacji. Zaczekaj, aż pojawi się tam komunikat:

```
----- Done -----
Build: 1 succeeded, 0 failed, 0 skipped
```

Otwórz folder, w którym utworzyłeś projekt, i znajdź w nim folder *TestMFC*. Powiniennego zawierać pliki kodu źródłowego projektu, wygenerowane przez kreator. Poza nimi powinieneś znaleźć folder *Release* z wygenerowanymi podczas komplikacji plikami kodu pośredniego i plikiem wykonywalnym. Wskaz w nim plik *TestMFC.exe* i obejrzyj jego właściwości (kliknij prawym przyciskiem myszy ikonę pliku i wybierz *Właściwości*). Rozmiar pustego programu to 368 kB. Dość dużo.

Spróbuj skompresować plik programem ASPack. W moim eksperymencie wersja skompresowana miała 187 kB. Przy kompresji rzędu 50% otrzymujemy plik o rozmiarze nadającym się już lepiej albo gorzej do płatania figli.



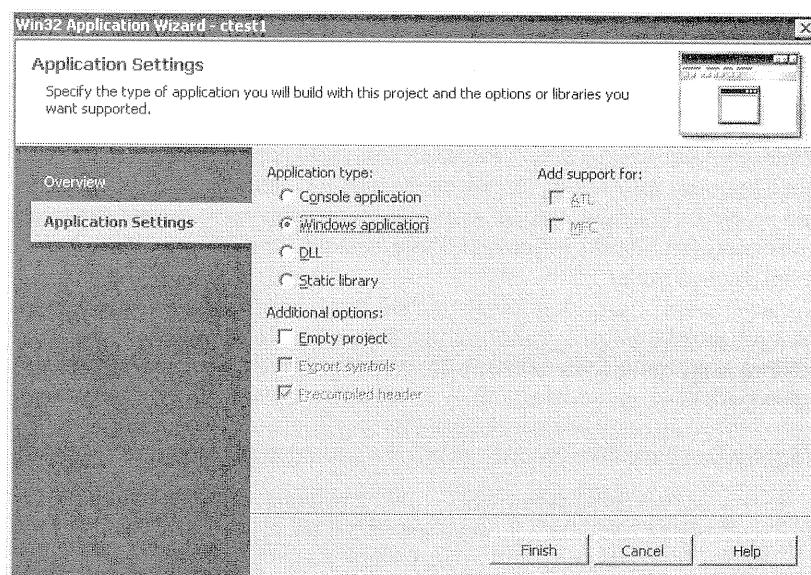
Pliki projektu przykładowego znajdują się na dołączonej do książki płyce CD-ROM, w jej podkatalogu *Przykłady/Rozdział1/TestMFC*. Aby otworzyć przykład w Visual C++, skorzystaj z opcji *Open solution* menu *File*. W wyświetlonym oknie wskaz po prostu katalog przykładu, a w nim plik o nazwie zgodnej z nazwą projektu i rozszerzeniu *vcproj* albo *.sh*.

Aby dalej zmniejszyć program, musimy porzucić MFC i oprogramować go za pomocą „gołego” interfejsu WinAPI dla języka C. To trochę bardziej i raczej niewygodne, ale przy mniejszych projektach niewygody te da się ścierpieć.

Aby utworzyć prosty program pozbawiony wsparcia MFC, wybierz z menu *File* pozycje *New/Project*, a następnie wskaz typ projektu *Win32 Project*. Nazwij projekt *CTest*, a ścieżki dostępu pozostaw bez zmian.

Jeśli poprzedni projekt wciąż jest otwarty, poniżej pola położenia plików projektu znajdziesz przycisk typu radio z dwoma pozycjami: *Add to solution* i *Close solution*. Jeśli wybierzesz pierwszą z nich, nowy projekt zostanie dodany do projektu już otwartego. Po wybraniu drugiej poprzedni projekt zostanie zamknięty, a na potrzeby nowego projektu utworzony zostanie nowy obszar roboczy.

Po naciśnięciu przycisku *OK* pojawi się okno kreatora. Pierwszy krok działania kreatora jest czysto informacyjny, można więc od razu przejść do etykiety *Application Settings*. Okno kreatora będzie wyglądać tak jak na rysunku 1.9.



Rysunek 1.9. Okno kreatora ustawień aplikacji

Jako że chcemy utworzyć jak najprostszą aplikację okienkową, wybieramy z grupy przycisków *Application type* pole *Windows application*. Jeśli kreator nie ma obciążać projektu dodatkami, nie zaznaczaj żadnych innych opcji. Chcemy otrzymać minimalną wersję aplikacji okienkowej. Od razu kliknij więc przycisk *Finish*, kończąc tworzenie nowego projektu.

Warto pamiętać o zmianie ustawienia rodzaju konsolidacji z *Debug* na *Release*. W ustawieniach projektu nie musimy niczego zmieniać, ponieważ tym razem nie korzystamy z MFC, więc nie musimy martwić się o tryb konsolidacji bibliotek MFC. Można to sprawdzić, otwierając okno właściwości projektu i upewniając się, że właściwość *Use of MFC* ma wartość *Standard Windows Libraries*. Oznacza to brak MFC w projekcie — program będzie działał z wykorzystaniem jedynie standardowych bibliotek systemu Windows.

Skompiluj projekt. W tym celu wybierz z menu *Build* pozycję *Build solution*. Kiedy komplikacja się zakończy, otwórz folder *ctest/release* w folderze projektów i sprawdź rozmiar pliku wykonywalnego. U mnie plik miał 81 kB. Po skompresowaniu udało się uzyskać rozmiar poniżej 70 kB. Taki program można już szybko przesłać siecią.

Oczywiście można go jeszcze bardziej zmniejszyć, usuwając kilka nieużywanych, a zajmujących miejsce komponentów. Na razie się od tego powstrzymamy.

1.3. Wnętrze programu

W pierwszej części tej książki będziemy często odwoływać się do szkieletu programu tworzonego przez kreator Visual C++ po wybraniu typu aplikacji *Win32 Application*. Dlatego warto zatrzymać się na chwilę i poznać kilka elementów tego szkieletu. Pełniejszego opisu należy szukać w książkach poświęconych w całości programowaniu w Visual C++ — tutaj ograniczymy się do minimum niezbędnych informacji.

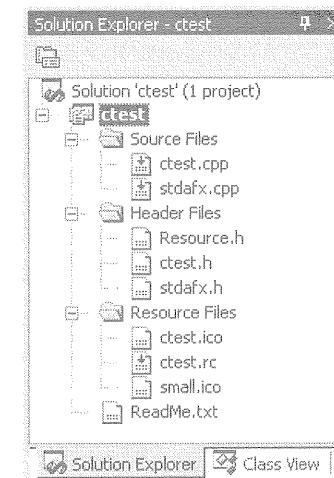
Będę wielokrotnie powtarzał opis aplikacji wygenerowanej przez kreator, więc będziesz miał okazję stopniowo poznać wszystkie elementy aplikacji. Wyjaśnianie wszystkich szczegółów od razu byłoby niecelowe, ponieważ zapamiętanie takiej „uderzeniowej” dawki informacji, które dodatkowo nie są w danej chwili użyteczne, mogłoby być trudne.

Otwórz projekt *CTest* i spójrz na panel *Solution Explorer*. Wyświetlane jest w nim drzewo reprezentujące wszystkie komponenty projektu. W moim przypadku drzewo to wyglądało jak na rysunku 1.10 — u siebie powinieneś zobaczyć podobną hierarchię.

Drzewo komponentów projektu składa się z następujących gałęzi:

- ◆ *Source files* (pliki źródłowe) — pliki zawierające kod źródłowy projektu. Główny plik kodu źródłowego ma nazwę taką jak nazwa projektu i opatrzony jest rozszerzeniem CPP. W naszym przypadku jest to *ctest.cpp*.

Rysunek 1.10.
Hierarchia komponentów projektu



- ◆ *Header files* (pliki nagłówkowe) — pliki zawierające mnóstwo informacji pomocniczych o włączonych do programu modułach i klasach.
- ◆ *Resource files* (pliki zasobów) — pliki ikon i plik zasobów projektu, u nas *ctest.rc*.

1.3.1. Zasoby projektu

Spójrzmy na nasze zasoby. Kliknij dwukrotnie nazwę pliku *ctest.rc*, a w tym samym panelu pojawi się zakładka widoku zasobów — *Resource View*. Wyświetla ona hierarchię zasobów.

Gałiąz *Accelerator* zawiera listę skrótów klawiszowych rozpoznawanych przez program. Tutaj może występować kilka zasobów, na razie jednak kreator utworzył jedynie jeden, o nazwie *IDC_CTEST*. Ponieważ nie zamierzamy wykorzystywać skrótów klawiszowych, można śmiało usunąć wszystkie pozycje z tej gałęzi. W tym celu należy kliknąć zasób prawym przyciskiem myszy i z wyświetlonego menu kontekstowego wybrać pozycję *Delete*. Z jednej strony program nie powinien zawierać żadnych nadmiarowych komponentów, z drugiej — nie zyskamy w ten sposób za wiele miejsca.

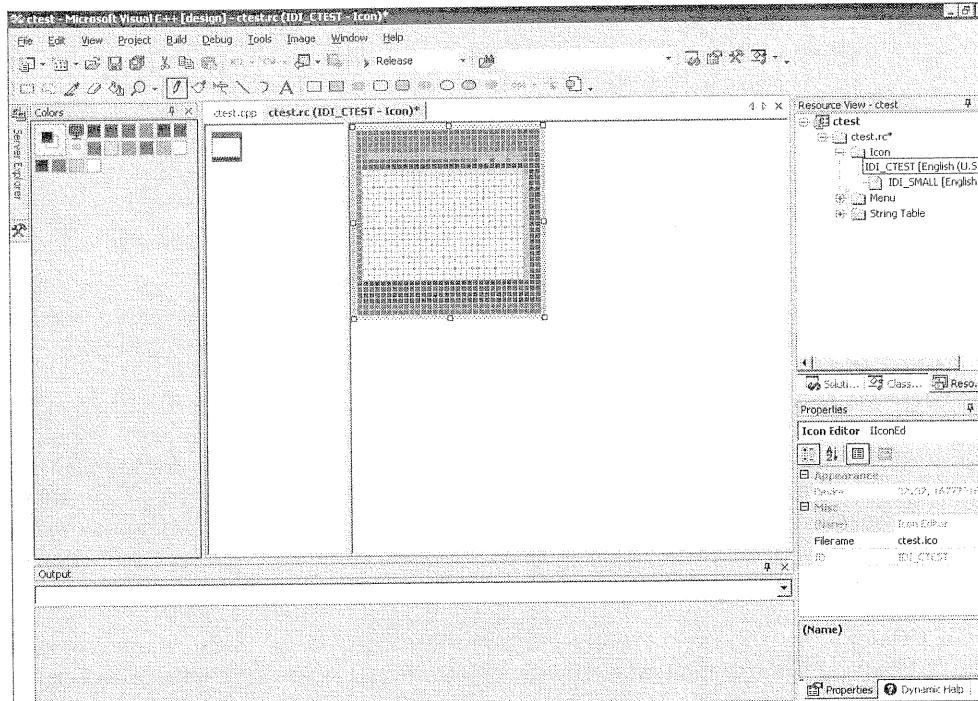
Gałiąz *Dialog* zawiera listę okien dialogowych. Jeśli masz zamiar tworzyć program działający w sposób niewidoczny dla użytkownika, nie powinien on z pewnością wyświetlać żadnych okien dialogowych. Domyślnie szkielet programu konstruowany przez kreator zawiera okno dialogowe *IDD_ABOUTBOX* wyświetlające informacje o programie. Aby je usunąć, wskaż je i kliknij prawym przyciskiem myszy, po czym wybierz z menu pozycję *Delete*.

Gałiąz *Icon* zawiera ikony programu. Może być ich kilka, różniących się rozmiarami i kolorami. W programach pełniących rolę żartów są one często potrzebne, ale powinny wyglądać jak najmniej podejrzanie, aby użytkownik uruchamiający program nie domyślił się dowcipu na podstawie ikony. Jeśli to Ty zasugerujesz mu uruchomienie

programu, ikona ma znaczenie drugorzędne. Inaczej jest w przypadku, kiedy ofiara uruchamia program z własnej inicjatywy — wtedy odpowiedni dobór ikony wprowadza dodatkowy element zaskoczenia.

Aby niedoszła ofiara uruchomiła program, jego ikona musi być jej znajoma. Można, na przykład, programowi-dowcipowi przypisać ikonę programu Microsoft Word. Jeśli w systemie użytkownika wyświetlanie znanych rozszerzeń plików jest wyłączone (jest to ustawienie domyślnie), ofiara weźmie nasz program za plik dokumentu MS Word. Do uruchomienia może też zachętać odpowiednio dobrana nazwa pliku.

Aby zmienić ikonę programu na jakąś efektowną i mało podejrzana, należy dwukrotnie kliknąć nazwę ikony; w głównym oknie projektu pojawi się okno edytora graficznego (patrz rysunek 1.11). Można za jego pomocą zmienić rysunek ikony. Edytor ten nie nadaje się co prawda do poważnej pracy graficznej, najlepiej więc wkleić do niego gotowy rysunek (na przykład ze schowka systemowego). Masę odpowiednich rysunków można znaleźć w internecie.



Rysunek 1.11. Edytor ikon

W gałęzi *Menu* przechowywane są zasoby menu programu. Omówimy je później; zresztą w naszych programach przykładowych z menu będziemy korzystać raczej rzadko.

Gałąź *String table* zawiera zasoby programu w postaci ciągów znaków. Domyślnie w skład tych zasobów wchodzi ciąg belki tytułowej okna. Ciagi nie zajmują dużo miejsca, można więc je zostawić.

1.3.2. Kod źródłowy programu

Przyjrzyjmy się teraz kodowi źródłowemu programu wygenerowanemu przez kreator. Kod ten zapisany jest w pliku *ctest.cpp*; tutaj prezentuję go na listingu 1.1. Oczywiście nie będziemy zajmować się wszystkimi wierszami kodu. Książka ta nie jest podręcznikiem programowania w języku C++, więc ograniczymy się do szczegółowego opisu najważniejszych elementów programu. Jeśli znasz już język C++, zrozumienie kodu zapisanego w pliku *ctest.cpp* nie będzie dla Ciebie problemem. Jeśli nie, musi wystarczyć Ci lektura komentarzy¹ i wyjaśnień.

Listing 1.1. Kod źródłowy z pliku *ctest.cpp*

```
#include "stdafx.h"
#include "ctest.h"
#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst; // Current instance
TCHAR szTitle[MAX_LOADSTRING]; // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING]; // The main window class name

// Forward declarations of functions included in this code module:
ATOM MyRegisterClass(HINSTANCE hInstance);
BOOL InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine,
    int nCmdShow)
{
    // TODO: Place code here.
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_CTEST, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_CTEST);
```

¹ Kreator aplikacji Visual C++ automatycznie generuje podstawowy szkielet programu, opatrując go własnymi komentarzami. Nie o te komentarze tu chodzi (choć i one spełniają ważną rolę informacyjną, ale, niestety, tylko dla osób władcących językiem angielskim) — mowa o takich komentarzach, które Autor wprowadził do kodu samodzielnie; są one przetłumaczone na język polski i odpowiednio wyróżniane — *przyp. tłum.*

```

// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

return (int) msg.wParam;
}

// FUNCTION: MyRegisterClass()
// PURPOSE: Registers the window class.
// COMMENTS:
// This function and its usage are only necessary if you want this code
// to be compatible with Win32 systems prior to the 'RegisterClassEx'
// function that was added to Windows 95. It is important to call this function
// so that the application will get 'well formed' small icons associated
// with it.
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEX wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);

    wcex.style = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = (WNDPROC)WndProc;
    wcex.cbClsExtra = 0;
    wcex.cbWndExtra = 0;
    wcex.hInstance = hInstance;
    wcex.hIcon = LoadIcon(hInstance, (LPCTSTR)IDI_CTEST);
    wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName = (LPCTSTR)IDC_CTEST;
    wcex.lpszClassName = szWindowClass;
    wcex.hIconSm = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);

    return RegisterClassEx(&wcex);
}

// FUNCTION: InitInstance(HANDLE, int)
// PURPOSE: Saves instance handle and creates main window
// COMMENTS:
// In this function, we save the instance handle in a global variable and
// create and display the main program window.
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;

    hInst = hInstance; // Store instance handle in our global variable

    hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

    if (!hWnd)
    {

```

```

        return FALSE;
    }

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return TRUE;
}

// FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
// PURPOSE: Processes messages for the main window.
// WM_COMMAND - process the application menu
// WM_PAINT - Paint the main window
// WM_DESTROY - post a quit message and return
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_COMMAND:
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
            {
                case IDM_ABOUT:
                    DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
                    break;
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
                default:
                    return DefWindowProc(hWnd, message, wParam, lParam);
            }
            break;
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            // TODO: Add any drawing code here...
            EndPaint(hWnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}

// Message handler for about box.
// Ponieważ usuwaliśmy okno dialogowe, możemy usunąć również poniższy kod.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{

```

```

switch (message)
{
    case WM_INITDIALOG:
        return TRUE;

    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
        break;
}
return FALSE;
}

```

Wiemy już, że nasz program nie powinien zawierać okna dialogowego informacji o programie i że takie okno usunęliśmy z zasobów. Kod programu wciąż zawiera jednak funkcję obsługi okna i przez to programu nie daje się uruchomić. Aby umożliwić uruchomienie projektu, usuń wszystko poniżej wiersza:

// Ponieważ usunęliśmy okno dialogowe, możemy usunąć również poniższy kod.

Kod ten wyświetla okno informacji o programie i możemy usunąć go w całości.

Teraz przejdź do funkcji WndProc i usuń z niej wywołanie funkcji About. W tym celu zlokalizuj w kodzie programu i usuń trzy następujące wiersze:

```

case IDM_ABOUT:
    DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
    break;

```

Wiersze te obsługują wywołanie pozycji *About* z menu *Help* naszego programu. Wszystkie procedury obsługi takich wywołań mają w kodzie źródłowym następującą postać:

```

case Identyfikator:
    akcje
    break;

```

Identyfikator jest tu stałą przypisaną do elementu sterującego (tutaj pozycji menu). Instrukcja case sprawdza, czy z elementu sterującego zgłoszono zdarzenie o danym identyfikatorze. Jeśli tak, wykonywany jest kod aż do instrukcji break. O zdarzeniach powiemy sobie więcej nieco później.

Usunęliśmy już wszystko, co było zbędne. Możemy jeszcze usunąć kilka elementów (jak menu czy nawet okno programu — wtedy będzie on zupełnie niewidzialny), ale zostawimy je, ponieważ są przydatne dla celów prezentacyjnych. Obłaskawmy więc nasz program, który i tak jest już niewielki i zawiera bardzo mało zbędnego kodu.

Przy tworzeniu programów, które mają być niewidzialne, należałoby jeszcze odszukać i usunąć dwa wiersze:

```

ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

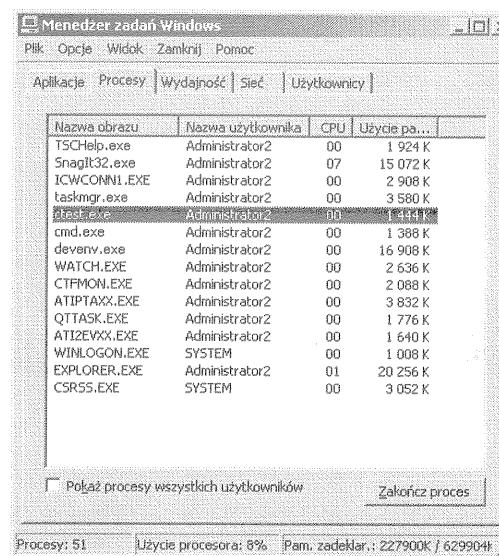
```

Znajdują się one w ciele funkcji InitInstance, która służy właściwie do tworzenia i wyświetlania okna głównego programu na pulpicie. Można pozostawić w niej kod tworzący okno, ukrywając program przez usunięcie wierszy wyświetlających okno.

Pierwszy z tych dwóch wierszy wyłącza wyświetlanie gotowego już okna programu, drugi powoduje odrysowanie zawartości okna. Można je usunąć z kodu, oznaczając je jako komentarz — służą do tego dwa znaki ukośnika umieszczone na początku wiersza. Skompiluj teraz program i spróbujmy go uruchomić. Jeśli naciśniemy *Ctrl+Alt+Del*, na liście nie zobaczymy naszego programu. W systemie Windows XP można odnaleźć ślad programu jedynie na zakładce *Procesy* (rysunek 1.12).

Rysunek 1.12.

Program *ctest* na liście procesów w Windows XP



Jeśli nie jesteś doświadczonym programistą Visual C++ i nie rozumiesz wielu elementów kodu źródłowego, nie przejmuj się. Wszystko się wkrótce wyjaśni. W dalszej części książki omówię większość kodu, który mieliśmy okazję przeglądać.

Przyjrzyjmy się bliżej kilku wybranym fragmentom kodu. Program rozpoczyna wykonanie od funkcji *_tWinMain*, której kod prezentowany jest na listingu 1.2.

Listing 1.2. Funkcja startowa programu

```

int APIENTRY _tWinMain(HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPTSTR lpCmdLine,
                        int nCmdShow)
{
    // Deklaracje dwóch zmiennych
    MSG msg;
    HACCEL hAccelTable;

    // Inicjalizacja zmiennych ciągów znaków
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_CTEST, szWindowClass, MAX_LOADSTRING);

```

```

MyRegisterClass(hInstance);

// Inicjalizacja programu
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}

hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_CTEST);

// Główna pętla przetwarzania komunikatów Windows
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

return (int) msg.wParam;
}

```

Ostatnio kilkukrotnie stosowałem pojęcie „funkcji” bez wyjaśniania jego znaczenia. Osoby mające doświadczenie w programowaniu w języku C++ z pewnością wiedzą, o co chodzi. Szczegółowe omówienie funkcji niestety wykracza poza tematyczny zakres tej książki — polecam lekturę jednej z publikacji poświęconych podstawom języka C++. Postaram się jednak wyposażyć Czytelnika w taką ilość informacji, która pozwoli przynajmniej zrozumieć przykłady.

W języku C++ wszystkie funkcje są deklarowane tak jak poniżej:

```

Typ Nazwa(Parametry)
{
}

```

Gdzie:

- ◆ Typ — typ wartości zwracanej przez funkcję. Typ int w funkcji _tWinMain oznacza liczbę całkowitą.
- ◆ Nazwa — nazwa funkcji (zasadniczo dowolna, ale główna funkcja programu okienkowego ma ściśle ustaloną i zawsze taką samą nazwę).
- ◆ Parametry — wartości przekazywane do funkcji, do których można się w niej odwoływać podczas przetwarzania.

Nasza funkcja główna posiada w deklaracji, za typem wartości zwracanej, słowo APIENTRY. Sygnalizuje ono, że funkcja jest punktem wejścia do programu.

Spójrz teraz ponownie na listing 1.1. Umieściłem na nim kilka komentarzy, które objaśniają znaczenie wybranych fragmentów kodu. Jak już wiemy, komentarze rozpoczynają się dwoma znakami ukośnika (//). Tekst znajdujący się za ukośnikami nie wpływa никак na wykonanie programu — ma jedynie wartość informacyjną i to tylko dla programisty.

Na początku funkcji deklarowane są dwie zmienne:

```

MSG msg;
HACCEL hAccelTable;

```

Deklaracja składa się z typu i nazwy. Typ instruuje kompilator, ile pamięci powinien zarezerwować dla zmiennej. Do wartości zmiennej można odwoływać się za pośrednictwem nazwy wymienionej w deklaracji. Typów w języku C++ jest cała masa — będziemy je poznawać w trakcie analizy kolejnych przykładów.

Jeśli zmienność jest typu prostego (na przykład liczbą, ciągiem znaków czy strukturą), po deklaracji można od razu z niej korzystać. Jeśli jest jednak wskaźnikiem albo obiektem, trzeba najpierw przypdzielić dla niej pamięć. Obiekty wykorzystywane są powszechnie w bibliotece MFC; wskaźniki są zmiennymi przechowującymi adresy pewnych obszarów pamięci („wskażującymi” te obszary). Jeśli wskaźnik ma wskazywać coś sensownego, należy najpierw przypdzielić pamięć, w której umieszczone zostaną wskazywane przezeń dane.

Po co nam wskaźniki? Wielu programistów nie docenia ich możliwości; wielu lęka się ich stosowania, obawiając się skutków wykraczania za pośrednictwem wskaźnika poza przypzieloną pamięć. Przypuszcmy, że do pamięci wczytaliśmy obrazek o wielkości 1 MB i chcemy umożliwić funkcji odczytanie danych tego obrazka. Moglibyśmy przekazać do funkcji cały megabajt danych, ale wywołanie funkcji z takim przekazaniem zajęłoby mnóstwo czasu i pamięci. Zamiast tego lepiej byłoby pokazać funkcji, w którym miejscu w pamięci znajduje się obrazek. Innymi słowy, najlepiej w takiej sytuacji przekazać do funkcji wskaźnik pamięci zawierającej dane obrazka.

Jeszcze inny sposób polega na wykorzystaniu zmiennych globalnych — tego należy jednak unikać. Zmienne globalne to takie, które są widziane z wnętrza każdej funkcji programu. Z reguły są one deklarowane w pliku nagłówkowym (pliku z rozszerzeniem H) albo na samym początku pliku kodu źródłowego, przed wszystkimi deklaracjami funkcji.

Zmienne lokalne z kolei to zmienne deklarowane wewnątrz funkcji i jako takie dostępne tylko dla tej funkcji. Zmienne takie są tworzone w momencie rozpoczęcia wykonywania funkcji. Są umieszczane w specjalnym obszarze pamięci zwanym stosem i usuwane z tej pamięci automatycznie w momencie zakończenia wykonania funkcji. Automatyczne tworzenie i usuwanie zmiennych dotyczy jednak znowu jedynie zmiennych typów prostych, nie zaś wskaźników — pamięć przypzieloną w takiej funkcji do wskaźnika trzeba zwolnić ręcznie.

Za deklaracjami zmiennych widzimy dwa wiersze:

```

LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
LoadString(hInstance, IDC_CTEST, szWindowClass, MAX_LOADSTRING);

```

Mamy tu dwa wywołania funkcji o nazwie LoadString. Jej działanie polega na wczytaniu ciągu znaków z pliku zasobów. Funkcja jest fragmentem kodu, który posiada nazwę i który można wywołać z innych miejsc programu. W tym przypadku wywołujemy kod wczytujący zasoby. Aby wywoływana funkcja „wiedziała”, o jakie zasoby nam chodzi i gdzie ma je załadować, w nawiasach uzupełniających nazwę funkcji podajemy parametry wywołania. Parametry oddzielamy przecinkami; w omawianym przypadku mamy cztery takie parametry:

- ◆ hInstance — wskaźnik egzemplarza programu.
- ◆ IDS_APP_TITLE — nazwa zasobu do wczytania. Gdybyśmy teraz kliknęli dwukrotnie gałąz *String Table* w przeglądarce zasobów, zobaczylibyśmy tabelę ciągów znaków, w której pierwsza kolumna zawiera nazwę ciągu, a druga właściwy ciąg (napis). Nazwę taką podajemy jako parametr wywołania funkcji wczytującej zasoby.
- ◆ szTitle — zmienna, do której funkcja ma wczytać wartość zasobu. Mamy tu w dwóch wywołaniach funkcji dwie różne zmienne, szTitle i szWindowClass, deklarowane na początku pliku:

```
TCHAR szTitle[MAX_LOADSTRING];           // ciąg belki tytułowej okna programu
TCHAR szWindowClass[MAX_LOADSTRING];      // nazwa klasy okna głównego programu
```

Jak już Ci wiadomo, deklaracja zmiennej rozpoczyna się od typu zmiennej. W tym przypadku typem jest TCHAR, co oznacza ciąg znaków. Za typem mamy nazwę zmiennej i dalej w nawiasach prostokątnych długość ciągu (tzn. maksymalną liczbę znaków w ciągu) wymaganą w deklaracjach ciągów. W naszym kodzie długość jest określona jako MAX_LOADSTRING. Wartość ta jest stała równą maksymalnej liczbie wczytywanych znaków. W nawiasach prostokątnych w deklaracji ciągów moglibyśmy zamiast stałej podać wprost jej wartość, ale najlepiej dla powtarzalnych wartości zdefiniować stałą i odwoływać się do niej w deklaracjach.

- ◆ MAX_LOADSTRING — ostatni parametr określa maksymalną liczbę wczytywanych znaków. Stosujemy tu tę samą stałą, co w deklaracjach zmiennych ciągów, w których funkcja ma umieścić zasoby. Dzięki temu długość wczytanego z pliku zasobów ciągu jest równa długości zmiennej, w której ma być zapisany — zabezpieczamy się tak przed przypadkową próbą zapisania w zmiennej ciągu dłuższego, niż to możliwe.

Dalej mamy wywołanie funkcji MyRegisterClass(hInstance). Funkcja ta wypełnia strukturę WNDCLASSEX. Cóż to takiego struktura? Struktura to zmienna, w której mieści się zbiór zmiennych dowolnych typów. Struktura może, na przykład, zawierać jeden ciąg o nazwie Nazwisko i jedną liczbę całkowitą o nazwie Wiek. Aby odczytać albo zapisać zmienną w strukturze, należy zapisać Struktura.Zmienna, gdzie Struktura jest nazwą struktury, a Zmienna nazwą jednej z jej zmiennych.

Struktura WNDCLASSEX wykorzystywana jest przy konstruowaniu nowej klasy okna. W najmniejszej nawet aplikacji trzeba wypełnić w niej przynajmniej następujące pola:

- ◆ style — styl okna.
- ◆ lpfnWndProc — wskaźnik funkcji wywoływanej w obliczu zdarzenia użytkownika.
- ◆ hInstance — uchwyt potrymany w wywołaniu funkcji _tWinMain.
- ◆ hbrBackground — kolor tła (zasadniczo nie trzeba ustawać tej zmiennej, ale przyjęty zostanie wtedy kolor domyślny dla okna).
- ◆ lpszClassName — nazwa tworzonej klasy okna.
- ◆ hCursor — kursor programu (należy załadować standardowy kursor w postaci strzałki).

Struktura jest gotowa i można przystąpić do rejestracji w systemie nowej klasy okna. W tym celu wywołujemy funkcję RegisterClassEx(&wcex). System uzyskuje w ten sposób opis przyszłego okna programu. Dlaczego „przyszłego”? Otóż jak dotychczas właściwe okno programu nie zostało jeszcze utworzone. Tworzy się je dopiero wywołaniem funkcji CreateWindowEx. Odbywa się to w funkcji InitInstance wywoływanej w _tWinMain zaraz po wywołaniu MyRegisterClass. Funkcja CreateWindowEx ma całkiem sporo parametrów, które wypadałoby opisać.

```
hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
```

- ◆ Nazwa klasy. Zarejestrowaliśmy wcześniej klasę i zapisaliśmy jej nazwę w zmiennej szWindowClass. Dlatego tę zmienną przekazujemy w wywołaniu CreateWindowEx.
- ◆ Nazwa okna. Nazwa okna to po prostu napis, który znajdzie się na belce tytułowej okna. Napis ten wczytaliśmy wcześniej funkcją LoadString do zmiennej szTitle.
- ◆ Styl okna — wybieramy najprostszy, WS_OVERLAPPEDWINDOW.
- ◆ Kolejne cztery parametry określają położenie lewego górnego narożnika okna, jak i jego rozmiary: szerokość i wysokość. Można przekazać zera albo wpisać CW_USEDEFAULT — okno będzie miało wtedy domyślne rozmiary i położenie.
- ◆ Główne okno dla tworzonego okna — nasze okno jest właśnie samo w sobie oknem głównym, więc przekazujemy NULL, czyli zero.

Pozostałe parametry chwilowo pominiemy. Po utworzeniu okna należałoby je wyświetlić. Służy do tego wywołanie funkcji ShowWindow, o której mówiliśmy wcześniej. Przyjmuje ona dwa parametry:

- ◆ Uchwyt utworzonego okna.
- ◆ Parametry wyświetlania okna — tutaj zastosowaliśmy nCmdShow. Jest to wartość przekazywana do programu i oparta na właściwościach określonych dla ikony wywołującej program. Znaczenie pozostałych parametrów można odczytać z plików pomocy funkcji WinAPI.

Ostatnią czynnością przygotowującą program do działania jest wywołanie UpdateWindow. Służy ono do odrysowania utworzonego właśnie na ekranie okna.

Jeśli program nie potrzebuje okna albo jeśli ma być niewidoczny dla użytkownika, dwa ostatnie wywołania funkcji trzeba oznaczyć jako komentarz albo po prostu usunąć z pliku kodu źródłowego.

Przyjrzyjmy się teraz cyklowi przetwarzania komunikatów:

```
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Funkcja GetMessage oczekuje na pojawienie się komunikatu systemowego albo komunikatu użytkownika i zwraca wartość TRUE, kiedy taki komunikat nadjeździe. Komunikat jest następnie konwertowany wywołaniem funkcji TranslateMessage i przekazywany do funkcji obsługi komunikatów, DispatchMessage.

Każdy program powinien dysponować własną funkcją obsługi komunikatów. Która to funkcja? Określiliśmy ją podczas tworzenia klasy okna wartością zapisywaną w zmiennej wcex.lpfnWndProc. W Visual C++ funkcję tę zwykle nazywać WndProc — taka nazwa jest wykorzystywana domyślnie. Treść samej funkcji można obejrzeć na listingu 1.1.

Funkcja obsługi komunikatów powinna zawsze zawierać wywołanie funkcji DefWindowProc. Funkcja ta poddaje dany komunikat przetwarzaniu za pośrednictwem domyślnej funkcji obsługi komunikatów. To bardzo ważne, ponieważ dzięki obecności funkcji DefWindowProc nie musimy samodzielnie obsługiwać wszystkich możliwych komunikatów systemowych, mogąc odwołać się do kodu systemu operacyjnego.

Warto jednak najpierw, przed przekazaniem do funkcji obsługi domyślnej, sprawdzić treść komunikatu. Jeśli to konieczne, można go obsługiwać samodzielnie. Nie trzeba obsługiwać wszystkich komunikatów — wystarczy ograniczyć się do tych, które są ważne z punktu widzenia funkcji programu. Pozostałe będą z powodzeniem obsługiwanie funkcją domyślną. Treść komunikatu sprawdzamy, porównując parametr message funkcji obsługi komunikatów z kodami komunikatów standardowych. Jeśli, na przykład, komunikat jest równy WM_DESTROY, program powinien zostać zakończony. W ramach obsługi komunikatu należałoby zwolnić pamięć przydzieloną dla programu.

Mając już pewne rozeznanie w kodzie przykładu, możemy go uruchomić. Na ekranie pojawi się puste okno. Aby je zamknąć, należy nacisnąć kombinację klawiszy Alt+F4 albo kliknąć lewym przyciskiem myszy przycisk zamknięcia okna.

Jeśli program ma być utajony, wystarczy usunąć z kodu programu wywołanie ShowWindow. Funkcja ta włącza wyświetlanie okna — bez niej okno programu (a więc i sam program) pozostanie niewidoczne. Można też zmienić drugi parametr wywołania funkcji na SW_HIDE — efekt będzie taki sam. Najwyraźniej przekazanie parametru SW_HIDE jest równoznaczne z pominięciem wywołania w ogóle. Funkcja ShowWindow jest wywoływana z parametrem SW_HIDE najczęściej wtedy, kiedy należy ukryć okno bez usuwania go z pamięci komputera.

Z funkcją ShowWindow spotkamy się jeszcze niejednokrotnie.

Aby skompilować projekt, wybierz z menu *Build* polecenie *Build Solution*. Spowoduje to komplikację projektu i utworzenie pliku wykonywalnego programu. Aby uruchomić gotowy program, wystarczy zaś z menu *Debug* wybrać *Start*.



Kod źródłowy przykładu znajduje się na dołączonej do książki płycie CD-ROM w podkatalogu *Przykłady\Rozdział 1\empt*.

1.4. Optymalizacja programu

Życie programisty to ciągła walka ze spowolnieniami i brakiem czasu. Programista codziennie spędza nieradko po kilka godzin na optymalizowaniu swoich programów. Każdy programista próbuje optymalizować wszystko, co dzieje się w programie. Czy na pewno robi to dobrze? Być może jest pole do ulepszeń.

Jestem świadom faktu, że ludzie są cokolwiek leniwi. Co do mnie, przyzwyczaiłem się, że mój komputer robi wszystko za mnie, a ja wszystko robię na komputerze i nie pamiętam już, jak wygląda zwykły długopis. Niedawno musiałem ręcznie wypełnić jakiś formularz. Okazało się, że zapomniałem, jak pisze się znaki angielskiego alfabetu — musiałem pomagać sobie, spoglądając na klawiaturę. Poważnie! To wina postępu, który pozwala mi na wszechstronne wykorzystanie komputera.

Żeby napisać najkrótszy choćby list, włącza się dziś komputer i uruchamia program Microsoft Word albo inny edytor tekstu, co jest oczywistą stratą czasu. Dlaczegoż ten czyta nie napiszą wiadomości odręcznie? Czyżby było to za trudne?

Szczególnie leniwi są programiści. Wielu z nich nie troszczy się o wygląd pisanej kodu źródłowego — i tak nikt poza nimi nie będzie go oglądał. Tu są w błędzie. Z tego punktu widzenia aplikacje open-source mają znaczną przewagę jako szybsze i bardziej przejrzyste. Programiści często z lenistwa zarzucają optymalizację rozmiaru i szybkości wykonania tworzonego przez siebie kodu. Kiedy widzę takie programy, ulegam frustracji. Niestety, moja frustracja nie ulepsza owych programów. Największymi leniami są — niestety — hakerzy.

Ta swego rodzaju degradacja jakości została chyba spowokowana przez firmę Microsoft. Bierzymy do ręki mysz i zaczynamy klikać tu i tam, zapominając o klawiaturze i skrótach klawiszowych. Jestem przekonany, że należy to powstrzymać. Ostatnio przyłapałem się na tym, że z lenistwa odłożyłem klawiaturę na bok i uruchomiłem klawiaturę ekranową obsługiwany myszą. Jeszcze tylko porosnę futrem i będę mógł spokojnie zamknąć się w klatce z małpami.

Nie wydawajcie pieniędzy na wieczne ulepszanie komputera! Ulepszcie się najpierw sami. Zoptymalizujcie swoją pracę, a i komputer okaże się szybszy niż dotychczas.

Pierwotnie ta część książki miała opisywać sposoby optymalizacji kodu programów. Później zdecydowałem się na dołączenie do książki jednego z moich artykułów publikowanych w internecie. Piszę w nim o tym, że optymalizować trzeba wszystko. Będę mówił o regułach optymalizacji pamięci. Są one na tyle uniwersalne, że można je wykorzystać tak do optymalizacji rozkładu dnia pracy, jak i optymalizacji i przyspieszenia systemu operacyjnego i komputera. Skupię się jednak mimo wszystko na optymalizacji kodu programu.

Jak zawsze spróbuję omówienie poprzez przykładami praktycznymi, żeby nie być posądzonym li tylko o snucie pięknych, ale nieprawdziwych opowieści.

Zacznę od zasad, które dają się zastosować nie tylko w programowaniu, ale również w życiu. O rzeczach przydatnych w optymalizacji kodu programu powiem natomiast przy okazji podsumowania.

Zasada 1. Optymalizować można wszystko

Nawet jeśli uważasz, że Twoja praca przebiega wyjątkowo sprawnie, nie lęk się — można ją usprawnić.

Zasada ta dotyczy zwłaszcza programowania. Nie ma czegoś takiego jak kod idealny. Optymalizować da się nawet najprostsze operacje, jak $2 + 2$. Aby otrzymać najlepsze wyniki, należy postępować w sposób opisany poniżej.

Pamiętaj, że każde zadanie może zostać wykonane na więcej niż jeden sposób, a zaczynając od wyboru tego, który zapewni najlepszą wydajność i będzie najbardziej uniwersalny.

Zasada 2. Szukaj wąskich gardł i słabych ogniw

Czy jest sens optymalizować mocne punkty i mocne ognia? Jeśli będziemy optymalizować je w pierwszej kolejności, natrafimy najprawdopodobniej na niespodziewane konflikty. A korzyści, jeśli jakiekolwiek, będą znikome.

Pamiętam taki przykład z własnej praktyki. Chyba w 1996 roku wpadłem na pomysł napisania własnej gry w stylu Doom. Nie miałem zamiaru na tym dziele zarabiać — traktowałem to wyłącznie jako wyzwanie. Po czterech miesiącach wytężonej pracy uzyskałem coś, co można by nazwać „silnikiem” gry. Stworzyłem też jeden poziom, po którym mógł poruszać się gracz. Dumny ze swojego dokonania zacząłem wałęsać się korytarzami poziomu.

W grze nie było jeszcze przeciwników, drzwi ani scen, a mimo to program działał bardzo powoli. Wyobraziłem sobie, co by było, gdybym dodał wszystkie brakujące elementy, uzupełnił całość o jako taką sztuczną inteligencję potworów i... zniechęciłem się. Komu potrzebny silnik gry, który nawet przy rozdzielczości 320×200 jest potwornie wolny? Dokładnie nikomu.

Mój wirtualny świat wymagał z pewnością optymalizacji. Przez miesiąc szlifowałem kod, biorąc pod lupę każdą instrukcję silnika gry. Efekt był następujący: świat odrysowywał się na ekranie o 10 procent szybciej, ale wciąż za wolno. Nagle zrozumiałem, że najsłabszym ogniwem i wąskim gardłem gry jest wyświetlanie gotowego obrazu na ekranie. Mój silnik obliczał kolejne ramki obrazu dostatecznie szybko, był jednak spowalniany samym wyświetlaniem. W owym czasie magistrala AGP była mrzonką przyszłości, a ja bazowałem na karcie graficznej S3 PCI z 1 MB pamięci graficznej.

Przez następne ileś godzin wydobyłem wszystko tak z karty graficznej, jak i z siebie. Skompilowałem ponownie program gry i wkroczyłem do wirtualnych Lochów. Naciśnałem klawisz „do przodu” i momentalnie znalazłem się przy przeciwległej ścianie.

Nie było żadnych spowolnień — gra działała z dużą prędkością, bez oczekiwania na odrysowanie obrazu.

Jak widać, popełniłem błąd na etapie określania wąskiego gardła w moim kodzie. Spędziłem blisko miesiąc na optymalizacji matematycznej części kodu, uzyskałem jednak poprawę rzędu 10 procent. Kiedy wreszcie znalazłem prawdziwe wąskie gardło, mogłem zwiększyć prędkość działania gry wielokrotnie.

Właśnie dlatego zalecam optymalizowanie w pierwszej kolejności najsłabszych ogniw programu. Być może ich ulepszenie da tak dobre rezultaty, że optymalizację reszty uznamy za zbędną. Z drugiej strony, jeśli zaczniemy optymalizację od mocnych punktów programu, namęczymy się mocno, a wyniki mogą okazać się mierne — jak u mnie, kiedy miesiąc poprawek dał dziesięcioprocentową tylko poprawę!

Wąskie gardła komputera

Nie rozumiem ludzi, którzy zmieniają procesor na szybszy, zostawiając w komputerze starą kartę graficzną S3, wiekowy dysk twardy o prędkości obrotowej 5400 obrotów na minutę i „aż” 32 MB pamięci. Przed udaniem się na zakupy otwórz obudowę komputera i przyjrzyj się jego elementom. Jeśli odkryjesz, że masz tylko 64 MB pamięci, powiedz głośno: „Drogi DIMM-ie, jesteś najsłabszym ogniwem i musisz opuścić mój komputer!”. Kup potem choćby 128 MB (albo 256, a najlepiej 512 MB) pamięci i ciesz się przyspieszeniem, którego doznają Visual C++, PhotoShop i inne wymagające aplikacje.

W takim układzie zwiększenie częstotliwości taktowania procesora o kilkaset nawet megaherców nie da porównywalnego przyspieszenia. Jeśli korzystasz z wymagających aplikacji i nie posiadasz odpowiedniej ilości pamięci, procesor, nawet najszybszy, większość czasu spędza na oczekiwaniu na odczyt i zapis danych w pliku wymiany. Kiedy zaś komputer posiada wystarczającą ilość pamięci, procesor może skoncentrować się na właściwych obliczeniach, nie tracąc czasu na zarządzanie pamięcią wirtualną.

To samo dotyczy się karty graficznej. Jeśli jest ona słaba, procesor będzie obliczał sceny szybciej, niż karta będzie w stanie je wyświetlić. Spowoduje to opóźnienia, a zmiana procesora na szybszy nie pomoże wcale albo pomoże niewiele. Dobra karta graficzna pozwoli na odpowiednio szybkie wyświetlanie efektów obliczeń procesora.

Zasada 3. W pierwszej kolejności optymalizuj operacje często powtarzane

Po wytypowaniu najsłabszych punktów programu należy znaleźć w nim te operacje, które są wielokrotnie powtarzane. Zasadę tę zilustruję na przykładzie programistycznym. Przypuśćmy, że analizujemy pewien kod, który można by zinterpretować następująco:

1. $A := A * 2$

2. $B := 1$

3. $X := X + B$

4. $B := B + 1$
5. Jeśli $B < 100$ idź do 3

Każdy programista powie, że wąskim gardłem tego kodu jest wiersz z numerem jeden, ponieważ występuje w nim mnożenie. To prawda. Mnożenie to operacja wybitnie czasochłonna — w tym przypadku można łatwo zaoszczędzić kilka czy kilkanaście cykli procesora, zastępując mnożenie dodawaniem ($A := A + A$), albo jeszcze lepiej przesunięciem bitowym. Zysk będzie jednak niemalże pomijalny.

Spójrzmy jeszcze raz na kod. Czy widzisz w nim coś specjalnego? Ja widzę. Mamy w nim pętlę: „dopóki B jest mniejsze od 100, wykonuj $X := X + B$ ”. Pętla zakłada wykonanie 100 skoków z wiersza 5. do wiersza 3. Sto skoków to sporo. Czy da się taką pętlę zoptymalizować? Jasne. Pętla obejmuje wiersze 3. i 4. Moglibyśmy na przykład powieść te wiersze wewnątrz pętli:

1. $B := 1$
2. $X := X + B$
3. $B := B + 1$
4. $X := X + B$
5. $B := B + 1$
6. Jeśli $B < 100$ idź do 3

Teraz mamy nieco krótszą pętlę (z mniejszą liczbą powtórzeń). Powtórzyliśmy w niej dwukrotnie operacje z pierwotnych wierszy 3. i 4. W efekcie są one wykonywane dwa razy w każdym przebiegu pętli. Pętla taka wykonuje się jedynie 50 razy (w jednym przebiegu mamy za to dwie operacje). Zaoszczędziliśmy 50 skoków. Nieźle. Zaoszczędziliśmy co najmniej kilkaset cykli procesora.

A gdyby, idąc za ciosem, powieść wiersze 2. i 3. w pętli dziesięciokrotnie? Pętla wykonywała by 10 operacji w jednym przebiegu i wykonałaby się tylko 10 razy. Zaoszczędziliśmy nawet 90 skoków.

Wadą takiej optymalizacji jest rozbudowa kodu. Zaletą za to znaczne przyspieszenie jego wykonania. Sposób ten jest dobry, nie wolno go jednak nadużywać. Zwiększa bowiem nie tylko szybkość wykonania kodu, ale i rozmiar programu wykonywalnego. A duży rozmiar programu nie sposób uznać za zaletę. Warto poszukać więc złotego środka w tej optymalizacji.

W każdym przedsięwzięciu ważna jest efektywność i umiarkowanie. Nie można przesadzać z optymalizacją szybkości, jeśli zaczyna to powodować znaczny rozrost programu.

Również w życiu mamy wiele przykładów tego rodzaju. Optymalizować można wszystkie cykliczne operacje. Weźmy choćby dostawcę usług internetowych, który obsługuje kilka numerów telefonów dostępu do sieci. Klient wybiera po kolej i wszystkie numery, szukając wolnej linii. Powinieneś powiedziać, że operator powinien zoptymalizować pulę modemów tak, aby klient mógł wybierać tylko jeden numer. Doświadczony

użytkownik powie jednak, że nie każdy klient ma dobre połączenie z każdą centralą. Dlatego musi posiadać swoje łączna na różnych centralach, aby można było zestawić jak najlepsze połączenie z każdym z klientów. Ci powinni zaś zainstalować program, który będzie automatycznie wybierał kolejne numery z puli numerów operatora.

Kolejny przykład: mamy jednogodzinną przedpłatę na dostęp do internetu u pewnego dostawcy. Byłoby bezcelowe definiowanie dla potrzeb otrzymanego numeru telefonu osobnego konta połączenia telefonicznego z internetem, ponieważ najprawdopodobniej połączenie wygaśnie po upływie godziny i numer nie będzie więcej odpowiadał. Ustawianie parametrów połączenia i ich późniejsze usuwanie zajmie więcej czasu niż skorzystanie ze standardowych narzędzi Windows. Wniosek: do każdego zadania należy starannie dobierać odpowiednie narzędzia.

Zasada 4. Pomyśl dwa razy, zanim zoptymalizujesz operacje jednorazowe

Zasada ta jest uzupełnieniem poprzedniej.

Optymalizowanie operacji, które realizowane są w programie tylko jednokrotnie, jest stratą czasu.

Jakieś pół roku temu czytałem w internecie historyjkę zatytuowaną „Z pamiętnika żony programisty” (<http://www.uic.nsu.ru/~exor/wife.htm>). To całkiem realistyczna i zabawna historia. Czytając ją, miałem wrażenie, że autorka mogłaby być moją własną żoną. Na szczęście nie jest aż tak złośliwa.

W owej historyjce pewna ładna dziewczyna ma niedługo wyjść za mąż za programistę. Zabierają się więc za rozsyłanie zaproszeń. Zamiast wypisać zaproszenia na maszynie programista postanawia, że jako ekspert w tej dziedzinie napisze stosowny program. Jeden dzień zajmuje mu jego napisanie, drugi — poprawianie.

Jego głównym błędem był brak właściwej optymalizacji. Byłoby przecież znacznie łatwiej utworzyć szablon zaproszenia w dowolnym edytorze tekstu i zmieniać w szablonie imiona i nazwiska gości tej szalonej stresującej imprezy (wiem z własnego doświadczenia). Nawet jeśli nie ma pod ręką odpowiedniego edytora tekstów, napisanie programu jest bezcelowe. Trwa to za długo jak na jednorazowy użytk. W rzeczy samej lepiej byłoby już zaprząć do pracy maszyny do pisania.

Wcześniej krytykowałem lenistwo. Jednak w pewnym sensie przyczynia się ono do optymalizacji. Prawdziwym ekspertem nie jest ten, który całymi dniami męczy się przy znikomych rezultatach, ale ten, który kończy swoje zadanie szybko i wykonuje je efektywnie, dostosowując środki do zadań. Warto o tym pamiętać.

Zasada 5. Poznaj wnętrze komputera i sposób jego działania

Im lepiej znasz swój komputer i sposób, w jaki wykonuje on pisany przez Ciebie kod, tym lepsze osiągniesz wyniki optymalizacji.

Tę zasadę musimy odnieść — z konieczności — jedynie do programowania. Trudno przy tym dać zestaw uniwersalnych wskazówek, można jednak opisać kilka technik optymalizacji:

- ◆ Unikaj obliczeń zmiennoprzecinkowych. Operacje na liczbach całkowitych wykonywane są wielokrotnie szybciej.
- ◆ Mnożenie to operacja powolna; jeszcze wolniejsze jest dzielenie. Jeśli musisz przemnożyć pewną liczbę przez np. 3, uzyskasz ten sam wynik szybciej, wykonując dodawanie trzech liczb. Co prawda w najnowszych procesorach mnożenie wykonywane jest stosunkowo szybko i zysk z takiej optymalizacji może być ledwieauważalny.

Jak w najefektywniejszy sposób zrealizować dzielenie? Warto znać matematykę. Procesor obsługuje operacje przesunięć bitowych. Warto pamiętać, że procesor „myśli” dwójkowo (binarnie) i wszystkie liczby są w pamięci komputera reprezentowane również dwójkowo. Na przykład liczba 198 to dla procesora 11000110. Weźmy się teraz za operacje przesunięć bitowych.

Przesunięcie bitowe w prawo — jeśli przesunąć bity liczby 11000110 o jedną pozycję w prawo, ostatni bit zniknie i zostanie nam 1100011. Wpisz taką wartość dwójkową do kalkulatora z konwersją systemów liczbowych. Zobaczysz, że w systemie dziesiętnym otrzymana po przesunięciu wartość to 99. Przypadkiem (a właściwie nieprzypadkowo) to dokładnie połowa 198.

Wniosek: przesunięcie bitowe liczby o jedną pozycję w prawo dzieli ją przez dwa.

Przesunięcie bitowe w lewo — weźmy znów tę samą liczbę, 11000110. Po przesunięciu jej cyfr o jedną pozycję w lewo na końcu liczby pojawi się niezajęta cyfra, zastępowana zerem. Otrzymamy 110001100. Wpisz taką wartość dwójkową do kalkulatora z konwersją systemów liczbowych. Zobaczysz, że w systemie dziesiętnym otrzymana po przesunięciu wartość to 396. Cóż to za liczba? To po prostu 198 razy 2.

Wniosek: przesunięcie bitowe liczby o jedną pozycję w lewo mnoży ją przez dwa.

Wszędzie tam, gdzie możemy, powinniśmy w miejsce mnożenia i dzielenia stosować operacje przesunięć bitowych, które wykonywane są przez procesor wielokrotnie szybciej niż mnożenie, dzielenie, a nawet dodawanie i odejmowanie.

- ◆ Definiując funkcję, unikaj przeładowania jej parametrami. Bezpośrednio przed wywołaniem funkcji procesor umieszcza wartości parametrów na stosie (specjalnym obszarze pamięci); funkcja odczytuje wartości parametrów, zdejmując je ze stosu. Im więcej parametrów, tym dłużej trwa wywołanie funkcji.

Warto tu zaznaczyć konieczność starannego stosowania samych parametrów. Nie wolno przekazywać do funkcji zmiennych zawierających znaczne ilości danych! Lepiej wtedy przekazać do funkcji adres pamięci, pod którym owe dane rezydują. Wyobraź sobie, że chcesz przekazać do funkcji ciąg znaków o objętości równej objętości *New York Timesa*. Przed wywołaniem funkcji program będzie musiał umieścić cały tekst w pamięci stosu. Jeśli nawet nie zdarzy się przepełnienie stosu, operacja ta zajmie bardzo dużo czasu.

- ◆ W najbardziej krytycznych miejscach kodu (na przykład w funkcji odrysowującej obraz na ekranie) używaj języka asemblerowego. Delphi i C++ pozwalały na umieszczenie w kodzie źródłowym wstawek asemblerowych, które są zazwyczaj wykonywane szybciej niż odpowiadające im instrukcje języków Delphi czy C++. Jeśli w danym fragmencie kodu szybkość wykonania ma znaczenie zasadnicze, można wyodrębnić ten fragment do osobnego modułu pisaneego w całości w języku asemblerowym i komplikowanego za pomocą TASM albo MASM, a następnie konsolidowanego z resztą programu.

Język asemblerowy daje szybkie i zwarte programy, ale bardzo trudno pisać w nim większe projekty. Zalecałbym wstrzemięźliwość w tym zakresie i uciekanie się do asemblera tylko tam, gdzie szybkość jest kluczowa dla programu.

Zasada 6. Przygotuj tabele gotowych wyników obliczeń i korzystaj z nich w czasie działania programu

Kiedy światło dzienne ujrzała pierwsza wersja gry Doom, społeczność graczy była oszołomiona jej szybkością i jakością grafiki. Było to prawdziwe dzieło sztuki programistycznej, ponieważ w owym czasie komputery nie były zdolne do generowania trójwymiarowej grafiki w czasie rzeczywistym. Nikt nie słyszał wtedy o akceleratorach grafiki trójwymiarowej, a karty graficzne wyświetlały jedynie dane o obrazie, nie wspomagając nijak procesu ich przygotowania.

W jaki sposób autorzy Dooma dali radę stworzyć trójwymiarowy świat gry? Sztuczka okazała się prosta. Gra wcale nie oblicza scen w czasie rzeczywistym. Wszystkie złożone obliczenia matematyczne zostały wykonane wcześniej, a ich wyniki zapisane w bazie danych ładowanych do pamięci gry przy jej uruchamianiu. Oczywiście taka baza danych nie mogła zawierać wszystkich informacji potrzebnych w przebiegu gry, a jedynie wyniki podstawowych, powtarzających się i długotrwałych obliczeń. Kiedy w grze potrzebna jest wartość, której nie ma akurat w tablicy gotowych wyników, nie przeprowadza się skomplikowanego obliczenia, a jedynie szacuje wartość przybliżoną. To właśnie dlatego Doom działał szybko mimo niesłychanej jak na swoje czasy jakości grafiki.

Kiedy na rynku gier pojawił się Quake, społeczność graczy znów została zaskoczona jakością oświetlenia i cieni w wirtualnym świecie gry. Obliczanie rozkładu oświetlenia to zadanie wyjątkowo złożone, nie mówiąc już o cieniach. Jak twórcy gry zdołali pogodzić tak wyśmienitą jakość grafiki z równie dobrą wydajnością? Odpowiedź brzmi podobnie jak poprzednio — przy użyciu tablic obliczonych wcześniej wyników.

Jakiś czas później przemysłem gier komputerowych wstrąsnęła jeszcze większa niespodzianka. Gra Quake 2 obliczała oświetlenie w czasie rzeczywistym, ale jej świat wyglądał odrobinę nienaturalnie. Nawet w grze Half-life, która bazowała na przestarzałym już silniku Quake'a świat wyglądał bardziej realnie. Sęk w tym, że komputery nie były wystarczająco wydajne, aby realizować obliczenia w czasie rzeczywistym wystarczająco dokładnie — obraz gry zepsuły nieco błędy zaokrągleń.

Mimo to Quake będzie zawsze cieszył się legendarną sławą. Nawet gdyby usunąć z niej oświetlenie i cieniowanie, gra pozostałyby znakomitą. To prawdziwy majstersztyk, dzieło prawdziwych hakerów.

Zasada 7. Nie ma niepotrzebnych testów

Bardzo często optymalizacja prowadzi do destabilizacji kodu. Dzieje się tak, kiedy programiści próbują zwiększyć wydajność, usuwając z kodu zabezpieczenia i testy wyników obliczeń, które uważają za zbędne. Zapamiętaj — nie ma niepotrzebnych zabezpieczeń! Jeśli nawet sądzisz, że wymagająca nietypowej obsługi sytuacja jest tak nieprawdopodobna, że można ją zignorować, nie oznacza to, że użytkownik programu będzie na nią równie odporny. Jemu bardzo często zdarza się pomylić klawisze albo wprowadzić złe dane.

Zawsze sprawdzaj dane wprowadzane przez użytkownika. Rób to zaraz po ich wprowadzeniu.

Nie umieszczaj testów wyjątkowych we wnętrzu pętli. Dodatkowe instrukcje if we wnętrzu pętli mają duży wpływ na wydajność. Wykonuj testy przed i za pętlami.

Pętle to wąskie gardła i słabe ognia wszystkich programów komputerowych. Optymalizację rozpocznij od nich. Nie umieszczaj testów wewnętrz pętli. Wszystkie operacje wykonywane w pętli są przeprowadzane wielokrotnie, więc każda zbędna instrukcja rozrasta się do wielu zbędnych instrukcji.

Zasada 8. Nie bądź nadgorliwy

Nadmierna gorliwość w optymalizowaniu może być bezwocna. Wyznacz sobie realistyczne cele i zadania. Jeśli nie podołasz optymalizacji kodu do pożądanego poziomu, nie marnuj czasu na niekończące się próby. Poszukaj lepiej innego rozwiązania problemu.

Często pożądany ideał jest niemożliwy do osiągnięcia, ponieważ optymalizacja czasu wykonania programu kłoci się zazwyczaj z optymalizacją jego jakości. Widać to szczególnie w programach przetwarzających grafikę. Aby przyspieszyć odrysowywanie sceny (np. w grze komputerowej), można wykonać obliczenia szybkie, ale przybliżone. Gra będzie działać szybko, ale obraz będzie pogorszony. Z drugiej strony można zwiększyć jakość obrazu, poświęcając na obliczenia więcej czasu. Trzeba wybrać pomiędzy tymi alternatywami.

Twórcy edytorów graficznych poświęcają szybkość na rzecz jakości, ponieważ od ich produktów nie wymaga się działania w czasie rzeczywistym. Twórcy gier muszą poświęcić jakość. Inaczej gra, choć piękna, nie nadawałaby się do grania — byłaby zbyt powolna.

Podsumowanie

Jeśli przeczytałeś ten podrozdział z odpowiednią uwagą, możesz powiedzieć, że znasz podstawy optymalizacji. Są to jednak jedynie podstawy i nie wolno Ci spocząć na laurach. Mógłbym napisać o optymalizacji więcej, ale byłoby to bezcelowe. Optymalizacja to proces twórczy, wymagający specjalnego podejścia w każdym odmiennym przypadku. Mimo to wymienione reguły obowiązują w 99,9% tych przypadków.

Jeśli chcesz poznać głębszą teorię optymalizacji oprogramowania, powinieneś zapoznać się z podstawami działania procesorów i systemów operacyjnych. Najważniejsze jednak, że znasz zasady. Reszta przyjdzie w końcu sama, wraz z praktyką.

1.5. Prawidłowe projektowanie okien

Przymierząc się do pisania własnych programów z myślą o ich sprzedaży, powinieneś starannie przemyśleć ich interfejs. Jak zwykle najważniejsze jest pierwsze wrażenie. Jeśli okno programu odrzuca swoim wyglądem, nikt nie doceni innych zalet programu. Jak stworzyć program na tyle atrakcyjny, żeby użytkownik poświęcił mu więcej niż 5 minut i zdołał docenić jego właściwe możliwości? To nie takie trudne. Cała sztuka we właściwym podejściu do projektowania.

Niegdyś próbowałem w swoich programach zawierać niestandardowe elementy interfejsu okna głównego, chcąc wyróżnić się na tle konkurentów, ale moje programy sprzedawały się kiepsko. Po trzech latach takiego postępowania dałem programowi standardowe okno z prostymi przyciskami i pozycjami menu — sprzedaż wzrosła trzykrotnie. Użytkownicy nie lubią ani pionierów, ani maruderów projektowania interfejsu, nie lubią komplikacji i dziwacznych elementów sterujących. Chcą prostych programów, które będą potrafiły obsługiwać zaraz po instalacji.

Jeśli tworzysz niewielki program z kilkoma zaledwie funkcjami, jego okno, przyciski i menu mogą mieć dowolne rozmiary, kształty i kolory. Okno programu-dialera może być na przykład owalne, zaokrąglone albo mieć kształt egzotycznego zwierzęcia, a jego interfejs ograniczać się do trzech pól tekstowych (numer telefonu, nazwa użytkownika i hasło) oraz przycisku „Wybierz numer”. Użytkownik od razu zorientuje się, jakie jest przeznaczenie tych elementów niezależnie od ich układu i wyglądu. Można puścić wodze wyobraźni i przyciągnąć użytkownika niestandardowym, ale ładnym i wciąż wygodnym interfejsem (patrz rysunek 1.13).

Dobrym przykładem niewielkiego narzędzia, które podbiło w ten sposób świat, jest WinAMP. Program jest bardzo prosty w obsłudze i użytkownik od razu wie, jak odtworzyć muzykę, niezależnie od wyglądu okna głównego. Takie rozwiązanie jest atypowe. Ładna powłoka graficzna pozwala programowi zająć właściwą dla niego niszę rynkową. Jeśli dodatkowo program posiada funkcję zmiany szaty graficznej (przez zmianę tzw. skór), będziesz w połowie drogi do sukcesu.

Rysunek 1.13.
Odtwarzacze dźwięku i wideo mogą mieć niemal dowolny wygląd



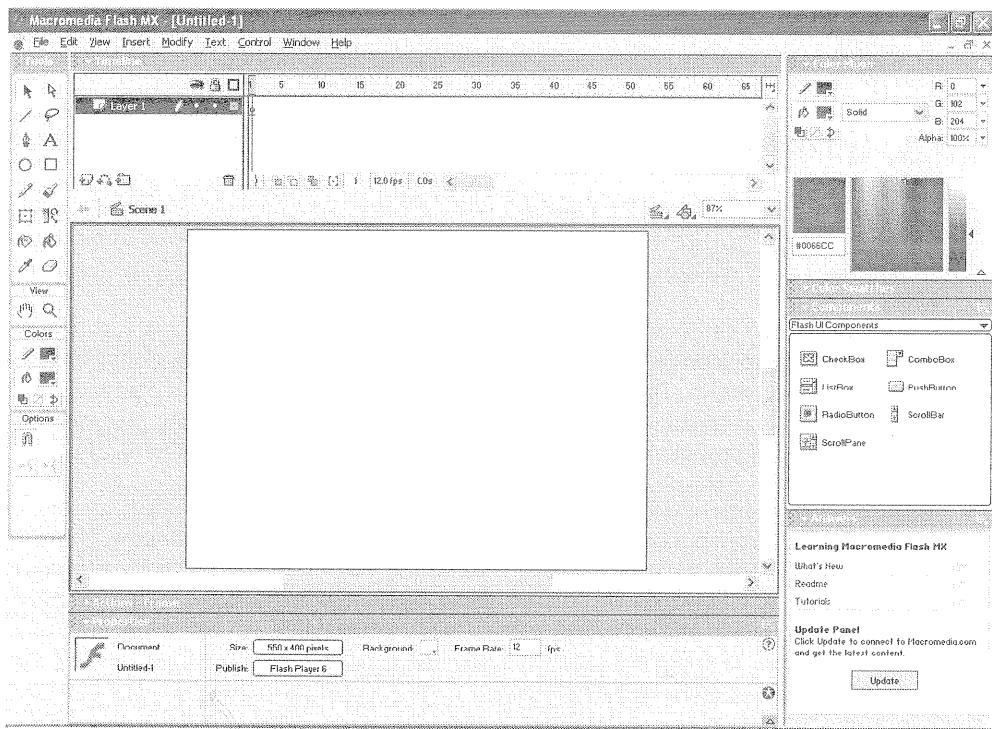
Kiedy tworysz program z wieloma funkcjami i o złożonej strukturze, w głównym oknie należy zachować standardowy układ elementów sterujących — okno powinno być prostokątne i zachowywać standardowe kolory Windows. Wyobraź sobie, że okno główne programu Word jest zaokrąglone albo wręcz okrągłe. Program byłby może i ciekawy ze względów estetycznych, ale ja bym go nie zainstalował.

Upewnij się, że w swoim programie spełniasz standardy de facto przyjęte w danej dziedzinie. Przykładem takiego standardu de facto dla edytorów grafiki jest interfejs programu PhotoShop. Wcześniej wszyscy producenci starali się wymyślić coś specjalnego, w końcu przekonali się jednak, że wygląd programu firmy Adobe jest po prostu modny i teraz z kolei starają się zachować podobieństwo.

Kiedy na rynku pojawił się Flash 5, programiści z firmy Macromedia próbowali upodobnić jego interfejs do interfejsu PhotoShopa. Mimo że jedna z tych aplikacji operuje obrazami rastrowymi, a druga wektorowymi, ich wygląd był bardzo zbliżony. Na przykład pasek narzędzi znikł, choć zwiększał funkcjonalność programu. W efekcie Macromedia Flash 5 stał się niezwykle popularnym narzędziem zwłaszcza wśród zawodowych grafików. W rzeczy samej możliwości graficzne nie zmieniły się wiele w 5. wersji w porównaniu z wersjami poprzednimi. Główną innowacją była funkcja ActionScript i nowy interfejs. Artyści nie są programistami i raczej nie zwracają sobie głowy możliwościami ActionScript. Spodobał się im jednak znany już im interfejs — nie musieli poświęcać miesięcy na zmianę przyzwyczajień i wyrobienie nowych nawyków (patrz rysunek 1.14).

Rozpoczynając nowy projekt, przyjrzyj się najpierw produktom konkurencji. Szczególną uwagę zwróć na produkt kontrolujący rynek i posiadający największy w nim udział. Trzymaj się wyznaczanych przezeń standardów. Jeśli lider rynku stosuje oryginalne, własne rozwiązania, staraj się zastosować podobne u siebie. W takim przypadku pójście własną drogą (np. powrót do rozwiązań klasycznych) może być zgubne. Zapewne uda się i tak znaleźć klientów, ale nie będzie ich wielu. Konkuruj w jakości, zestawie funkcji i wygodzie, nigdy w „wodotryskach”. Inaczej skażesz się na porażkę.

Rozdział 1. ♦ Jak uczynić program zwartym, a najlepiej niewidzialnym?



Rysunek 1.14. Choć Flash MX ma nowy wygląd, wciąż przypomina PhotoShopa

Jeśli urodziłeś się wynalazcą, możesz próbować własnych rozwiązań. Może okażą się najlepsze. Nie będziesz tego wiedział, jeśli nie spróbujesz. Ryzyko porażki jest wtedy wysokie, ale jeśli trafisz w dziesiątkę, będziesz dyktował modę na następny sezon i zbierzesz obfite plony. Gdyby programiści firmy Nullsoft nie zaryzykowali i nie stworzyli czegoś oryginalnego, WinAMP nigdy nie stałby się tak popularny mimo dziesiątek najbardziej nawet zaawansowanych funkcji.

Widziałem już wiele odtwarzaczy, które przewyższały WinAMPA liczbą funkcji, ale to WinAMP podbił serca miłośników muzyki. Inne programy są tylko naśladowcami, a dzisiejszy rynek zalany jest wprost odtwarzaczami z możliwością dowolnej zmiany interfejsu.

Negatywnym przykładem jest zaś 3D FTP. Jego programiści próbowali odnieść sukces, wyposażając klienta protokołu FTP w możliwość zmiany wyglądu interfejsu. Nigdy nie widziałem gorszej aplikacji. Wyobrażasz sobie Adobe PhotoShop czy Microsoft Word z funkcją zmiany „skóry”? 3D FTP to program o wielu funkcjach (wyprzedzający wielu konkurentów), który nie odniósł sukcesu. Jego autorzy powinni wzorować się raczej na Cute FTP albo CyD FTP i upodobić okna programu do okien tych dwóch aplikacji.

Systemy operacyjne z rodzin Windows podbiły rynek dzięki standardowemu interfejsowi i ujednoliceniu wyglądu aplikacji. Nawet nowicjusz wie, gdzie powinien szukać funkcji wczytania pliku albo wydruku zawartości dokumentu czy funkcji edycji. Po uruchomieniu aplikacji łatwo zgadnąć, które przyciski do czego służą.

Projektowanie interfejsu to rozległy temat, któremu poświęcono już wiele książek. Warto poznać choćby podstawy tej sztuki — może dzięki nim uda się odnieść sukces.

1.5.1. Interfejs okna głównego

Co jest dla każdego programu projektowane w pierwszej kolejności? Oczywiście jego okno główne. Jak już wiesz, okno to powinno być prostokątne i zawierać menu systemowe. Nigdy nie usuwaj z okna głównego ramek, chyba że jest to niezbędne.

Okno powinno zawierać menu i pasek narzędziowy z przyciskami reprezentującymi najważniejsze polecenia — elementy te powinny znajdować się u góry okna. Nie zakładaj, że użytkownicy Twojego programu domyślą się sami funkcji poszczególnych elementów. Każde polecenie powinno być opatrzone krótkim opisem, wyświetlonym na pasku stanu programu. Same polecenia w menu powinny być krótkie.

Jeśli w programie występuje pasek narzędziowy z przyciskami, niekiedy trudno jest odgadnąć funkcje poszczególnych przycisków na podstawie ich rysunków. Niektórzy programiści sądzą, że wystarczającą wskazówką jest umieszczenie takich samych rysunków obok odpowiednich poleceń menu. Wtedy jednak użytkownik, aby poznać funkcję przycisku, musi otwierać kolejne menu i przeglądać ich zawartość — tymczasem powinien móc poznać znaczenie przycisku, odczytując pojawiającą się w oknie wskazówkę albo napis na pasku stanu.

Pasek stanu powinien też zawierać informacje o bieżącym stanie wykonania programu albo postępie rozpoczętych operacji. Nie warto wyświetlać tych informacji w osobnych panelach czy oknach. Do tego służy właśnie pasek stanu.

Nazwy pozycji menu powinny być maksymalnie opisowe, ale nie powinny składać się z więcej niż trzech słów. Dłuższe opisy należy przerzucać na pasek stanu. W przypadku standardowych funkcji należy trzymać się standardowych nazw. Nie warto zamiast *Plik/Nowy* wprowadzać pozycji *Plik/Nowy moduł XXX*. To będzie zbyt rozwlekłe, a i tak zbyt mało mówiące.

Pasek narzędzi powinien również być zgodny z przyjętymi konwencjami. U samej góry okna należy umieścić pasek z poleceniami standardowymi (*Nowy*, *Otwórz*, *Drukuj* itd.). Nie warto przesuwać go na dół okna albo umieszczać na jego bokach. Najlepiej opatrzyć przyciski standardowymi ikonami, takimi jakie występują np. w pakiecie MS Office albo innych aplikacjach firmy Microsoft. Użytkownicy zdążyli do nich przywyknąć — będą mieli wrażenie znajomości interfejsu.

Jeśli nie potrafisz rysować, znajdź stosowne ikonki w internecie — w żadnym razie nie sil się na stworzenie własnego obrazka rozpoznawalnego tylko dla Ciebie. Jeśli jednak masz zdolności artystyczne, możesz spróbować narysować sam ikony podobne do ikon stosowanych w produktach konkurentów. Użytkownicy tych produktów będą z mniejszymi oporami przesiadać się na nowy program. To bardzo ważne, stąd ikonom należy poświęcić szczególnie dużo uwagi.

Ikonki powinny mieć przy tym oczywiste znaczenie, związane z funkcją, do której przypisany jest przycisk. Jeśli na przycisku wyboru palety kolorów będzie rysunek hipopotama, to do odgadnięcia zadania przycisku trzeba będzie współczesnego Nostradamusa.

Warto wyposażyć pasek narzędziowy w możliwość modyfikacji, aby użytkownik mógł z niego usuwać przyciski niepotrzebne i pozostawić jedynie te, które faktycznie wykorzystuje w pracy. Jeśli jednak pasek składa się zaledwie z 10 przycisków, taka funkcja będzie zbędna. Można wtedy ewentualnie pozwolić użytkownikowi na chowanie albo pokazywanie całego paska narzędziowego.

Przyciski paska narzędziowego powinny być pogrupowane wedle funkcji. Jeśli przycisków jest dużo, należy utworzyć kilka pasków narzędzi. Przy grupowaniu przycisków można kierować się kolejnością poleceń w menu głównym programu. Jeśli program wyposażony jest w więcej niż dwa paski narzędzi, użytkownik powinien mieć możliwość zamiany ich miejscami oraz pokazywania i chowania jednego albo obu pasków. Użytkownik będzie mógł w ten sposób decydować o wielkości obszaru roboczego pozostającego do jego dyspozycji.

1.5.2. Elementy sterujące

Wszystkie elementy sterujące w oknie powinny pochodzić ze zbioru standardowych elementów Windows. Nie warto tworzyć przycisków o nieregularnych kształtach dla samej przyjemności ich oglądania w programie. Kiedyś zaokrągliliem przyciski i wygładziłem krawędzie pól tekstowych. Nie zwiększyło to nijak sprzedaży programów, zwiększyło za to komplikacje w projektowaniu spójnego interfejsu.

W firmie Microsoft nad ułatwieniami interfejsu łamią głowy setki zawodowców. Nie myśl, że jesteś od nich lepszy. Możesz pozwolić sobie na projektowanie własnych elementów sterujących jedynie wtedy, kiedy w zbiorze elementów standardowych nie znajdziesz potrzebnej rzeczy. Poza wszystkim warto zaś upraszczać sobie zadanie, a nie je utrudniać.

1.5.3. Okna dialogowe

Bardzo ważne są okna dialogowe programu. To za ich pomocą użytkownicy wprowadzają dane i obserwują wyniki. Jeśli są one niewygodne i irytujące, użytkownik najpewniej uruchomi program *uninstall.exe*. Każdy element sterujący w oknie dialogowym powinien mieć swoje starannie przemyślane miejsce.

Jednym oknem, które może wyglądać zupełnie dowolnie, jest okno „O programie” (ang. *About*) — patrz rysunek 1.15. Najprawdopodobniej użytkownik nigdy go nie zobaczy. Jeśli w oknie tym pojawi się coś niespodziewanego, nie będzie to dla użytkownika żadnym problemem. We wszystkich pozostałych oknach jakiekolwiek dekoracje są niemile widziane. Nawet okno informacji o programie musi jednak być zaprojektowane z uwzględnieniem pewnych reguł — powinno zawierać przycisk *Zamknij*, *Close* albo *OK*, inaczej trudno będzie użytkownikowi zgadnąć, że żeby zamknąć okno, wystarczy kliknąć myszą w dowolnym miejscu w jego obszarze.

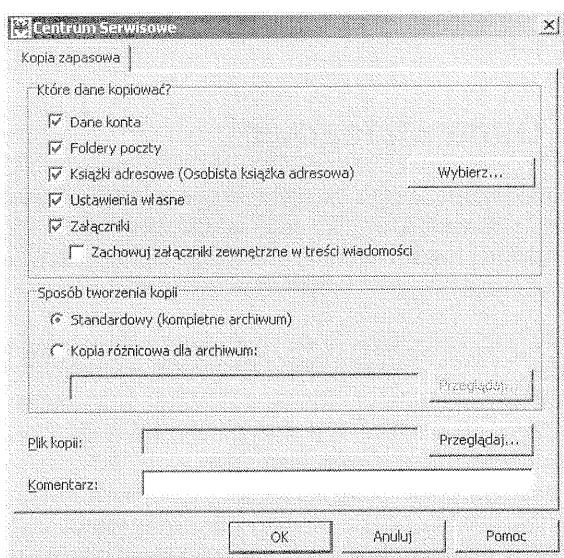
Rysunek 1.15.
Okno
„O programie”
programu The Bat!



Jak więc powinno wyglądać okno dialogowe? Niewątpliwie powinno być prostokątne. Poza tym byłoby dobrze, żeby jego szerokość była większa od wysokości. Takie okna dialogowe są lepiej odbierane, ponieważ człowiek jest przyzwyczajony do postrzegania wszystkiego w odniesieniu horyzontalnym. Oglądamy przecież filmy panoramiczne, a rozdzielczości poziome monitorów są większe, niż pionowe. Szerokie okno dialogowe łatwiej uczyńić „przyjemnym” dla oka.

Spójrz na okno dialogowe kopii zapasowych programu The Bat! (rysunek 1.16). Jest ono wyższe niż szersze, co sprawia, że jego odbiór jest negatywny. Rozumiem, że programiści próbowali udostępnić w pojedynczym oknie jak największą liczbę opcji, ale proporcje i brak wyrównania opcji w oknie są nieco denerwujące. Dodatkowo ostatnie z pól wyboru zostało przesunięte w prawo. To nieporozumienie. Wszystkie elementy sterujące powinny być wyrównane do lewej krawędzi okna, bo odchylenia zaburzają wygląd okna. Jeśli jedno ustawienie zależy od poprzedniego, najlepiej je blokować (właściwości Enable), a nie wysuwać z linii.

Rysunek 1.16.
Okno dialogowe
kopii zapasowych
w programie
The Bat!

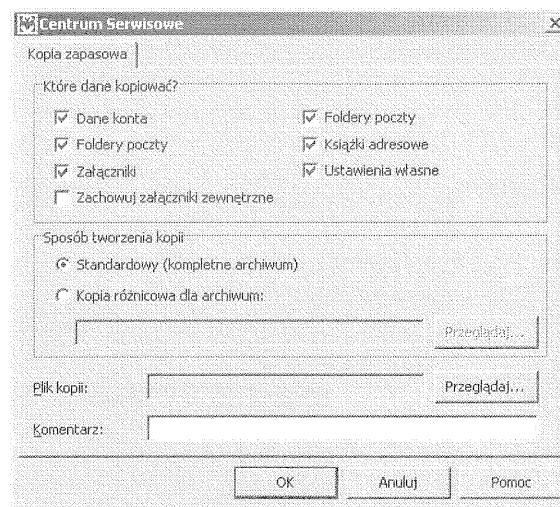


Rozdział 1. ♦ Jak uczynić program zwartym, a najlepiej niewidzialnym?

W omawianym oknie dobrze jest to, że elementy sterujące zostały właściwie pogrupowane. Pola wyboru znajdują się w jednej grupie, przyciski typu radio w innej.

Spójrz teraz na rysunek 1.17. Usunąłem z niego kilka niepotrzebnych elementów (można by je zostawić, ale i bez nich funkcjonalność programu nie jest wiele mniejsza). Wyrównałem też elementy i poszerzyłem okno dialogowe. Teraz wygląda zupełnie inaczej i — moim zdaniem — jest wygodniejsze w obsłudze.

Rysunek 1.17.
Poprawione okno
dialogowe The Bat!



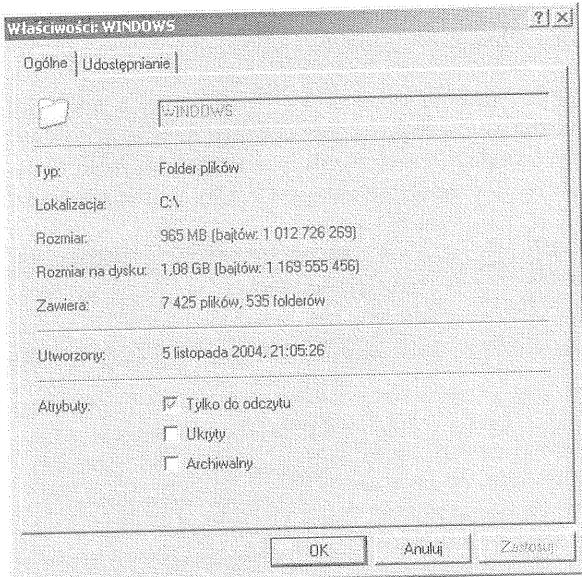
Jeśli chcesz wyświetlić właściwości obiektu (pliku czy dokumentu), powinieneś to zrobić w oknie rozciagniętym w pionie. To wyjątek od reguły i trzeba go bezwzględnie przestrzegać!

Do grupowania informacji o obiekcie powinny służyć zakładki. Dobrym przykładem takiego grupowania jest okno właściwości pliku (katalogu) w Eksploratorze Windows (rysunek 1.18) albo okno właściwości dokumentu w programie Microsoft Word (*Plik/Właściwości*).

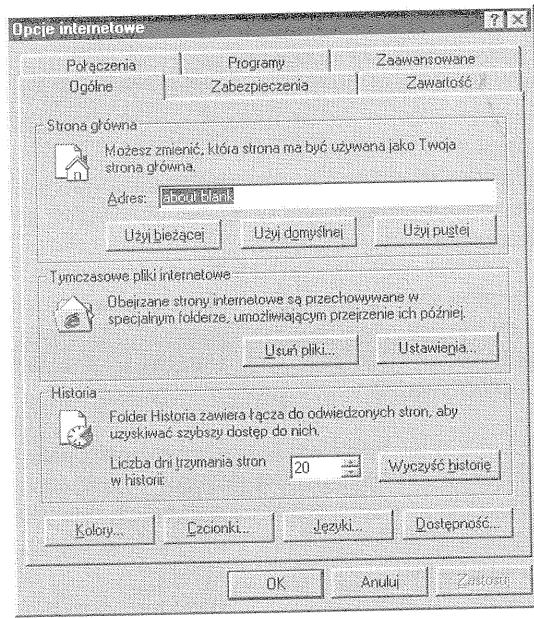
Żaden obiekt nie posiada tylu właściwości, aby nie dało się pogrupować ich na czterech czy pięciu zakładkach okna dialogowego. Jeśli liczba właściwości jest zbyt duża, aby pomieścić je na czterech zakładkach, należy pomyśleć o optymalizacji struktury obiektu. Spróbuj zdecydować, które z informacji są naprawdę ważne dla użytkownika, a które można pominąć.

Jeśli aplikacja ma sporo parametrów, można utworzyć coś w stylu okna właściwości Internet Explorera firmy Microsoft, czyli posłużyć się zestawem zakładek (rysunek 1.19). Tam, gdzie ustawień jest jeszcze więcej, warto zastosować pomysł znany z programu Netscape Navigator — w panelu po lewej stronie okna umieszcza się hierarchię kategorii opcji, a prawy panel wyświetla grupy parametrów związanego z daną gałęzią hierarchii.

Rysunek 1.18.
Okno właściwości katalogu



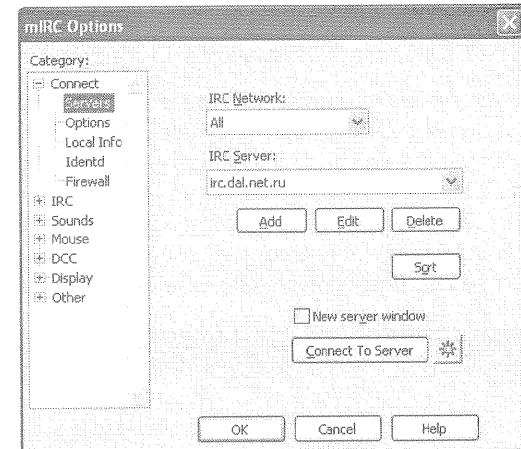
Rysunek 1.19.
Ustawienia internetowe



Nie warto próbować umieszczać zbyt wielu informacji na jednym panelu. Spada wtedy czytelność, a użytkownik ma trudności z lokalizowaniem niezbędnych ustawień. Warto zostawić odstępy pomiędzy elementami i stosować wcięcia tak, aby utrzymać spójność wyglądu okna dialogowego.

Spójrz na rysunek 1.20 — widać na nim okno ustawień programu mIRC. Ma ono wiele wad. Panel hierarchii opcji jest za blisko lewej krawędzi okna i jest zbyt wąski, a samo okno niewystarczająco szerokie. Elementy sterujące są ułożone na chyb il trafili.

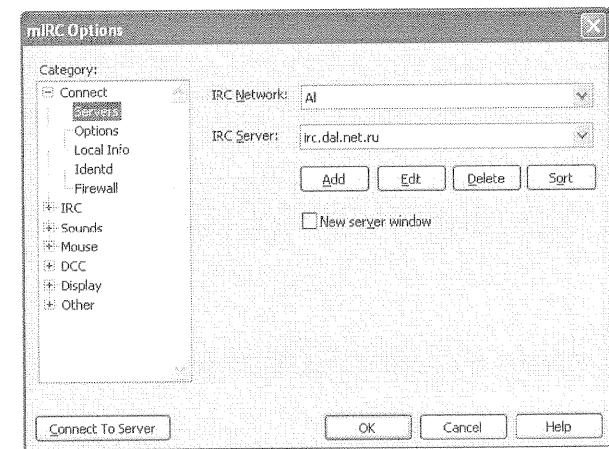
Rysunek 1.20.
Okno dialogowe ustawień programu mIRC



Są tam też przyciski zupełnie tam niepotrzebne, które powinny zostać przesunięte do innych okien dialogowych. Przycisk *Sort* jest dziwnie oddalony i umyka uwadze. Na domiar złego listy rozwijane mają różne szerokości.

Teraz popatrz na rysunek 1.21, na którym widnieje to samo okno dialogowe po małych przeróbkach w programie MS Paint. Okno zostało poszerzone, a znajdujące się w nim elementy wyglądają znacznie porządniej, ponieważ zostały wyrównane do lewej i mają równe szerokości. Przycisk ze „słoneczkiem” został usunięty, ponieważ był mylący, a jego jedynym zadaniem było przeniesienie użytkownika do okna ustawień gałęzi *Connect*.

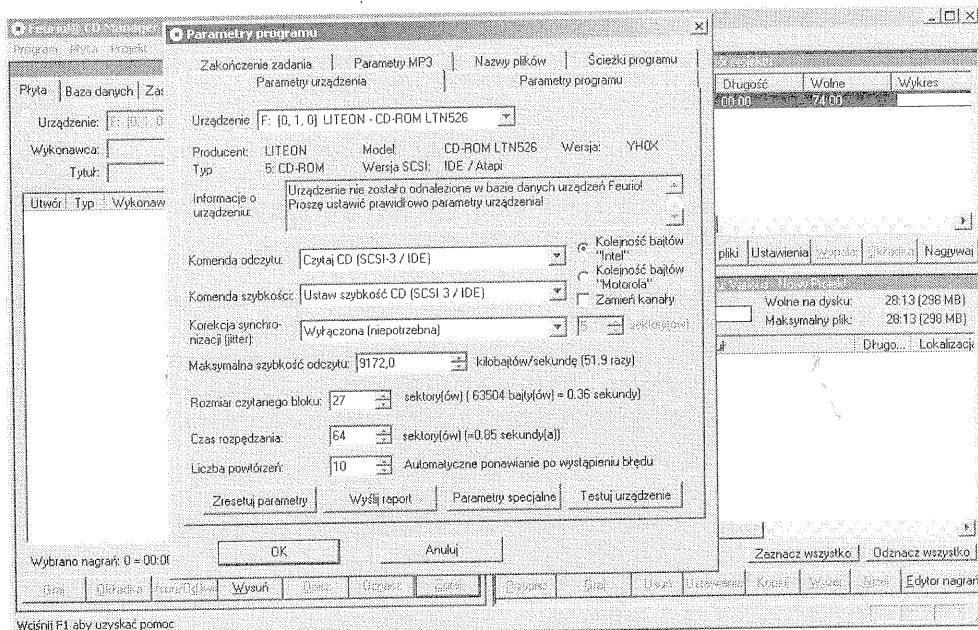
Rysunek 1.21.
Okno dialogowe ustawień programu mIRC — wersja poprawiona



W oknie dialogowym powstało trochę wolnej, niezagospodarowanej przestrzeni, nie oznacza to jednak, że należy zmniejszyć jego rozmiary. Być może panele pozostałych gałęzi hierarchii opcji wymagać będą więcej miejsca. W oknie głównym można umieścić dowolną liczbę elementów sterujących, aby aplikacja była jak najbardziej komunikatywna. W oknach dialogowych nie trzeba całkowicie wypełniać dostępnej przestrzeni. Ważniejszy jest wygląd.

Nie marnuj czasu na tworzenie wymyślonego interfejsu. Najlepszy nawet program, jeśli nie będzie wygodny, przegra z prostym, ale użytecznym i wygodnym narzędziem.

Pamiętam, jak dla magazynu *Hacker* opisywałem Feurio, bardzo efektywny program do nagrywania płyt CD. Spędziłem tydzień na nauce jego interfejsu (rysunek 1.22) i doszedłem do wniosku, że to bardzo dobry program, który powinien znaleźć się w komputerze każdego miłośnika muzyki. Ja jednak nie jestem miłośnikiem muzyki i jestem bardzo niezadowolony z jego niewygodów. Wolałbym korzystać z prostego i wygodnego WinOnCD niż z mocarnego, ale odstręczającego Feurio. Gdybym codziennie wypał płytę CD, może przekonałbym się do narzędzia bardziej zaawansowanego.



Rysunek 1.22. Przeładowany interfejs Feurio

Pamiętaj, jeśli nie jesteś pewny, jak postąpić, popatrz na konkurencję!

Rozdział 2.

Tworzenie prostych programów-żartów

Możemy teraz zacząć pisać żartobliwe programy dla systemu Windows. Ponieważ to najpopularniejszy system operacyjny, świetnie nadaje się do takich kawałów, gdyż łatwo będzie znaleźć ofiarę. Każdy programista jest zaś dumny, kiedy zdola w niewidocznny sposób podesłać swojemu koledze jakiś żarcik.

Ponieważ wszyscy pałamy wrodzoną żądzą bycia najlepszymi, nic dziwnego, że programiści również starają się demonstrować swoją wyższość, pisząc programy unikalne, ekscentryczne i ciekawe. Często owocem ich pracy są niezłe żarty.

Programy prezentowane w tej książce nie wyrządzają nikomu szkody, cała zabawa polega na ich utajonym przesłaniu do systemu kolegi. Będziemy nazywać go ofiarą żartu.

Większość żartów prezentowanych w tym rozdziale bazuje na podstawowych funkcjach WinAPI. Jak już powiedziałem, lekturę ułatwi na pewno umiejętność programowania. Spróbuje jednak tak przedstawić prezentowany kod, aby jego działanie było jasne dla wszystkich. Jeśli zresztą nawet na co dzień korzystasz z Visual C++, zapewne bardzo rzadko korzystasz z wywołań WinAPI. Nie spodziewam się więc, że je wszystkie już znasz.

Znam doskonałych programistów, którzy aplikacje bazodanowe potrafią pisać „z zamkniętymi oczami”. Nie potrafią jednak napisać programu, który przesuwa po ekranie wskaźnik myszy.

Aby zrozumieć, o co mi chodzi, spróbuj wykonać wszystkie żarty opisane w książce samodzielnie. To najlepszy sposób na naukę i opanowanie materiału. Lekcje praktyczne zapamiętujemy łatwiej i chętniej niż wykłady teoretyczne, będę więc podawał jak najwięcej przykładów.

2.1. Latający przycisk Start

Pamiętam, że kiedy pierwszy raz zobaczyłem Windows 95, szczególnie spodobał mi się przycisk *Start* oraz przycisk „Wyłącz komputer”. Wkrótce potem komputery w naszym instytucie zostały ulepszone i zainstalowano na nich właśnie Windows 95. Chciałem zażartować sobie ze studentów i zdecydowałem, że napiszę program, który będzie zmuszał przycisk *Start* do podskakiwania. Napisałem go i uruchomiłem na wszystkich komputerach. Zawsze, kiedy przycisk *Start* podfruwał do góry, wszyscy przed komputerami podskakiwali na krzesłach. Jakiś czas później podobny żart znalazłem w internecie.

Powtórzę teraz ten leciwy już dowcip i pokażę, jak napisać taki program. Choć pierwotnie pisałem go w Delphi, tym razem wykorzystamy do tego Visual C++. Usiądź wygodnie, bo niedługo będziesz gonił przycisk *Start* po ekranie!

W tym przykładzie wykorzystamy sztuczkę — nie będziemy podrzucać w górę samego przycisku, a jedynie okno z rysunkiem przycisku *Start*. Później pokażę Ci, jak odwołać się do właściwego przycisku, na razie udawajmy, że daliśmy się nabrać. Obiecuję, że i tak będzie ciekawie.

Zanim zaczniemy programować, powinniśmy przygotować sobie wizerunek przycisku *Start*. Możemy narysować go ręcznie w dowolnym edytorze grafiki. Jeśli jednak posiadasz klawiaturę z przyciskiem *Print Screen*, możesz za jego pomocą zapisać obraz pulpitu w pamięci schowka i wkleić go do programu graficznego. Wtedy wystarczy okroić obraz pulpitu i otrzymamy obrazek przycisku *Start*. Zapiszmy go w pliku.

Ja uzyskałem w ten sposób obrazek o rozmiarach 55 na 22 piksele; znajdziesz go na dołączonej do książki płytcie CD-ROM w katalogu \Przykłady\Rozdział2\Start Button\Start.bmp. Możesz go wykorzystać.

Utwórz w Visual C++ nowy projekt typu *Win32 Project* i nazwij go *Start Button*. Powinieneś dodać do niego rysunek przycisku. W tym celu otwórz okno zasobów, klikając dwukrotnie gałąź *Start Button.rc* w przeglądarce projektu. Zobaczysz okno z hierarchią zasobów, jak na rysunku 2.1.

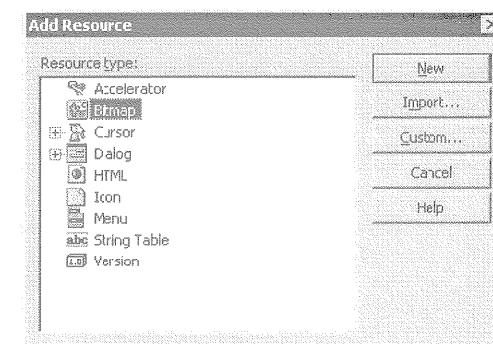
Rysunek 2.1.
Okno widoku
zasobów



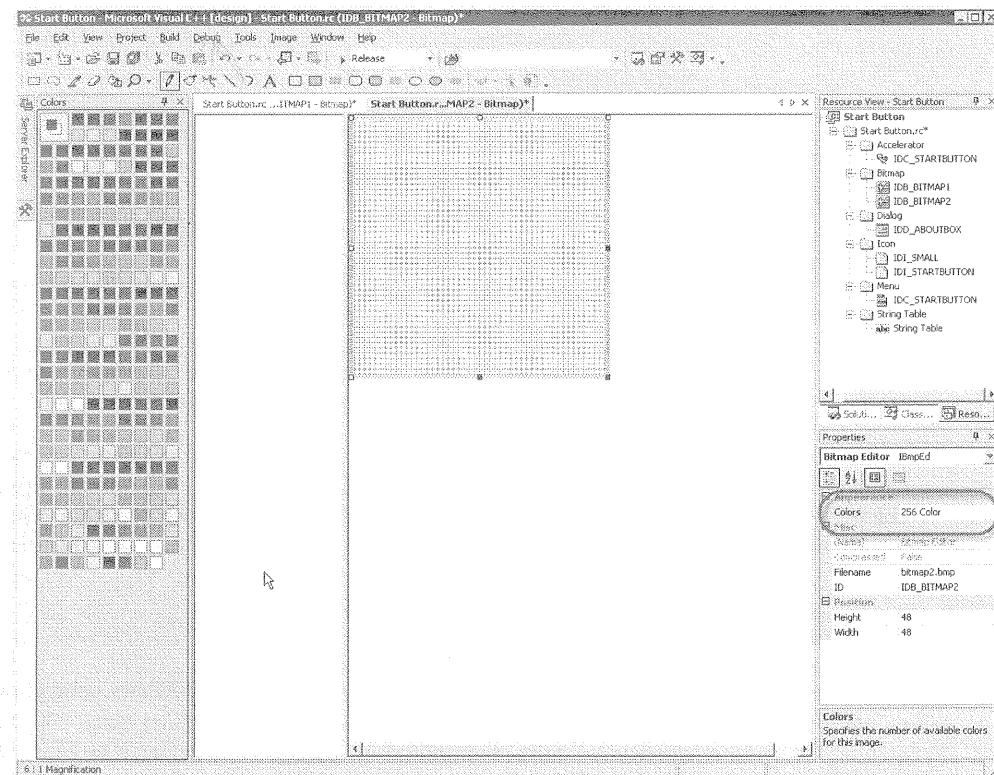
Rozdział 2. ♦ Tworzenie prostych programów-żartów

Kliknij okno prawym przyciskiem myszy i z menu podręcznego wybierz polecenie *Add*, a następnie *Add Resource* (dodaj zasób). Ujrzyś okno wyboru typu zasobu (rysunek 2.2). Wskaż pozycję *Bitmap* i kliknij przycisk *New*.

Rysunek 2.2.
Okno dodawania
zasobu

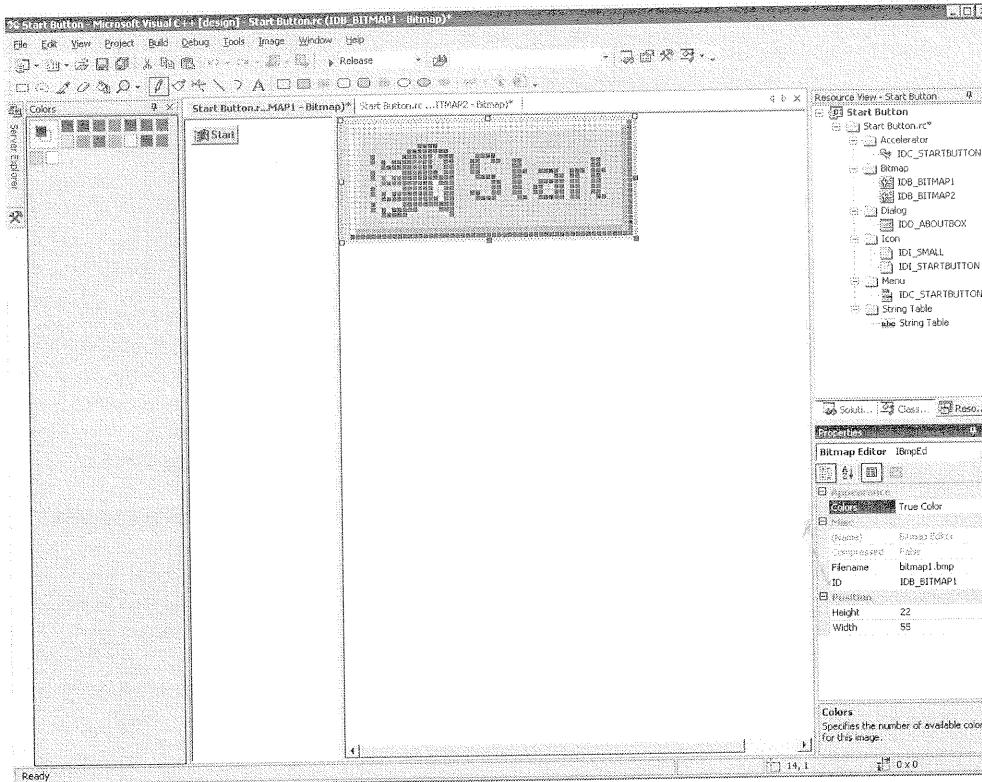


Jeśli gałąź *Bitmap* wcześniej nie istniała — zostanie teraz utworzona. Będzie ona zawierać zasób związanego z rysunkiem. Poniżej okna widoku zasobów zobaczysz okno *Properties* (właściwości, rysunek 2.3). Na samym początku powinieneś zmienić w tym oknie właściwość *Colors*, ustawiając wartość *256 Color* albo *True Color*. Określ też właściwości *Width* (szerokość) i *Height* (wysokość) zgodnie z rozmiarami rysunku.



Rysunek 2.3. Okno właściwości obrazka

Otwórz teraz rysunek przycisku *Start* w dowolnym edytorze zasobów (choćby w programie Paint) i skopiuj go do schowka (będziesz musiał posłużyć się w tym celu poleciennem *Copy* z menu *Edit* programu, do którego wczytałeś rysunek). Wróć teraz do edytora Visual C++ i wybierz z kolejи z menu *Edit* polecenie *Paste*. Obszar roboczy powinien wyglądać po tym tak, jak na rysunku 2.4.



Rysunek 2.4. Obraz przycisku *Start* w edytorze zasobów

Przejdzmy do właściwego programowania. Przykład ten będzie o tyle istotny, że wykorzystamy w nim elementy, z których później będziemy dość często korzystać.

Otwórz teraz plik *Start Button.cpp*. W tym celu w oknie przeglądarki projektu *Solution Explorer* rozwiń gałąź *Source Files* (pliki źródłowe) i kliknij dwukrotnie odpowiedni wpis. Odszukaj w pliku źródłowym zmienne globalne (są położone w okolicach początku pliku, w sekcji opatrzonej komentarzem „Global Variables”). Dodaj do pliku dwie zmienne:

```
// Global Variables
HWND hWnd;
HBITMAP startBitmap;
```

Pierwsza z tych zmiennych ma typ *HWND*. Typ ten służy do przechowywania uchwytów okien. W zmiennej tej będziemy składać uchwyt utworzonego później okna, tak aby móc się do niego odwoływać w dowolnym momencie. Druga ze zmiennych to zmienność typu *HBITMAP*. Typ ten służy jako uchwyt obrazków — zmienność tego typu posłuży nam do przechowywania obrazka przycisku *Start*.

Przejdz do funkcji *_tWinMain*. W jej wnętrzu dodaj (za wywołaniami funkcji wczytywających z zasobów ciąg znaków belki tytułowej i nazwy klasy okna programu) następujące wiersze:

```
startBitmap = (HBITMAP)::LoadImage(hInstance,
    MAKEINTRESOURCE(IDB_BITMAP1), IMAGE_BITMAP,
    0, 0, LR_DEFAULTCOLOR);
```

Kod ten przypisuje do zmiennej *startBitmap* obraz przycisku *Start* wczytany z zasobów. Wczytuje go funkcja *LoadImage* przyjmująca następujące parametry:

- ◆ Uchwyt egzemplarza programu. Tutaj przekazujemy zmienną *hInstance* przekazywaną pierwszym parametrem do funkcji *_tWinMain*. Zmienna ta zawiera identyfikator egzemplarza programu.
- ◆ Nazwa zasobu — nasz obrazek został w zasobach zapisany jako *IDB_BITMAP1*.
- ◆ Typ obrazka. Ponieważ mamy do czynienia z bitmapą, przekazujemy *IMAGE_BITMAP*.
- ◆ Wymiary obrazka (dwa kolejne parametry). Przekazując zera, godzimy się na bieżący rozmiar obrazka.
- ◆ Znacznik *LR_DEFAULTCOLOR* — wymuszający zastosowanie dla obrazka domyślnych kolorów.

Nie zmieniamy w funkcji *_tWinMain* niczego więcej — wróćmy do niej później. Na razie przejdźmy jednak do funkcji *InitInstance*. Jej kod widnieje na listingu 2.1.

Listing 2.1. Zaktualizowany kod funkcji *InitInstance*

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    hInst = hInstance; // Store instance handle in our global variable

    hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

    if (!hWnd)
    {
        return FALSE;
    }

    // Te linie dodaliśmy:
    int Style;
    Style = GetWindowLong(hWnd, GWL_STYLE);
    Style = Style || WS_CAPTION;
    Style = Style || WS_SYSMENU;
    SetWindowLong(hWnd, GWL_STYLE, Style);

    return TRUE;
}
```

Kod, który dodaliśmy, rozpoczyna się od deklaracji zmiennej Style typu int (liczba całkowita). W kolejnym wierszu zmiennej tej przypisywany jest wynik wywołania funkcji GetWindowLong zwracającej ustawienia okna. Funkcja ta przyjmuje dwa parametry:

- ◆ Okno, którego parametry mają zostać odczytane. My podaliśmy nowo utworzone okno programu.
- ◆ Rodzaj ustawień do odczytania. Ponieważ interesuje nas jedynie styl okna, podajemy `GWL_STYLE`.

Po co nam ów styl okna? Domyślnie okno posiada belkę tytułową, a na niej przyciski minimalizacji i maksymalizacji, których nie chcemy. Powinniśmy usunąć wszystkie elementy dekoracji okna, pozostawiając jedynie obszar roboczy, dlatego w kolejnych dwóch wierszach usuwamy ze stylu okna menu systemowe i belkę tytułową.

Dalej wywoływana jest funkcja SetWindowLong zapisująca nowe ustawienia okna. Jeśli teraz uruchomisz program, zobaczysz szary obszar roboczy okna bez belki tytułowej, przycisków ani ramek.

Czeka na nas funkcja WndProc obsługująca wszystkie komunikaty przewijające się przez aplikację. Interesuje nas zwłaszcza moment odrysowania okna na ekranie, więc dodajemy do funkcji kod obsługi komunikatu WM_PAINT:

```
case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    // TODO: Add any drawing code here...
    Rectangle(hdc, 1, 1, 10, 10);
    hdcBits = ::CreateCompatibleDC(hdc);
    SelectObject(hdcBits, startBitmap);
    BitBlt(hdc, 0, 0, 55, 22, hdcBits, 0, 0, SRCCOPY);
    DeleteDC(hdcBits);
    EndPaint(hWnd, &ps);
    break;
```

Funkcja WndProc w jej pełnej postaci prezentowana jest na listingu 2.2.

Listing 2.2. WndProc z obsługą komunikatu odrysowania okna

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;
    HDC hdcBits;

    switch (message)
    {
        case WM_COMMAND:
            wmId = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // Parse the menu selections
            switch (wmId)
            {
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
            }
            break;
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            // TODO: Add any drawing code here...
            Rectangle(hdc, 1, 1, 10, 10);
            hdcBits = ::CreateCompatibleDC(hdc);
            SelectObject(hdcBits, startBitmap);
            BitBlt(hdc, 0, 0, 55, 22, hdcBits, 0, 0, SRCCOPY);
            DeleteDC(hdcBits);
            EndPaint(hWnd, &ps);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

```
break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
break;
case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    // TODO: Add any drawing code here...
    Rectangle(hdc, 1, 1, 10, 10);
    hdcBits = ::CreateCompatibleDC(hdc);
    SelectObject(hdcBits, startBitmap);
    BitBlt(hdc, 0, 0, 55, 22, hdcBits, 0, 0, SRCCOPY);
    DeleteDC(hdcBits);
    EndPaint(hWnd, &ps);
    break;
case WM_DESTROY:
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
```

Aby rozpocząć rysowanie, powinniśmy wiedzieć, gdzie rysować. Każde okno posiada tzw. kontekst, po którym można rysować za pośrednictwem specjalnych systemowych narzędzi. Aby pobrać kontekst bieżącego okna, należy wywołać funkcję `BeginPaint`. Funkcja zwraca uchwyt kontekstu okna określonego pierwszym parametrem wywołania.

Aby wyświetlić obraz przycisku *Start*, musimy go najpierw przygotować. Wśród funkcji WinAPI nie ma funkcji, która odrysowałaby w kontekście bitmapę. Można jednak pobrać kontekst obrazka i wykonać kopiowanie zawartości kontekstów. W tym celu należy utworzyć kontekst rysowania zgodny z tym wykorzystywanym przez okno, żeby kopiowanie przebiegało bez problemów. Służy do tego funkcja `CreateCompatibleDC` z odpowiednim parametrem. Zwraca ona nowy kontekst, zgodny z kontekstem określonym na podstawie parametru wywołania.

Teraz trzeba w nowym kontekście osadzić obrazek. W tym celu skorzystamy z funkcji `SelectObject` przyjmującej dwa parametry:

- ◆ Kontekst, w którym wybierany jest obiekt. Tutaj podajemy nowo utworzony kontekst bazujący na kontekście okna.
- ◆ Obiekt, który ma być wybrany do kontekstu. Tutaj podajemy obrazek (jego uchwyt).

Teraz możemy skopiować obrazek pomiędzy kontekstami, odwołując się do funkcji `BitBlt`. Wymaga ona przekazania następujących parametrów:

- ◆ Kontekstu docelowego. Podajemy kontekst okna programu.
- ◆ Czterech parametrów w postaci liczb całkowitych określających prostokąt, do którego ma być kopowany kontekst źródłowy. Są to współrzędne lewego górnego narożnika obszaru kopiowania oraz jego szerokość i wysokość.

W naszym przypadku narożnik wypada w narożniku okna (dwa zera), a szerokość i wysokość są równe szerokości i wysokości okna (55 i 22).

- ◆ Kontekstu źródłowego. Podajemy hdcBits zawierający obrazek.
- ◆ Współrzędnych lewego górnego narożnika obszaru do skopiowania w kontekście źródłowym. Ponieważ chcemy skopiować cały obrazek przycisku, podajemy dwa zera.
- ◆ Rodzaju kopирования — tworzymy kopię kontekstu źródłowego w kontekście docelowym, podajemy więc SRC_CPY.

Po zakończeniu kopowania nie potrzebujemy już kontekstu obrazka. Należy wyrobić sobie nawyk usuwania kontekstu od razu po zakończeniu operacji na nim. Usunięcie kontekstu polega na wywołaniu funkcji DeleteDC z kontekstem obrazka w roli parametru.

Na koniec trzeba by zakończyć obsługę komunikatu odrysowania okna, wywołując funkcję EndPaint. Zakończymy w ten sposób proces zainicjowany wywołaniem BeginPaint.

Od tego momentu w oknie naszego programu, począwszy od jego lewego górnego narożnika, odrysowywany będzie przycisk *Start*. Pozostało już tylko dopasować rozmiar okna programu do rozmiaru obrazka przycisku (tak aby użytkownik nie widział okna spoza przycisku) i zmusić je do poruszania się po ekranie. Posłuży do tego funkcja DrawStartButton, taka jak na listingu 2.3.

Listing 2.3. Funkcja poruszająca oknem programu

```
void DrawStartButton()
{
    int i;
    HANDLE h;
    int toppos = GetSystemMetrics(SM_CSCREEN) - 24;

    // Ustaw pozycję okna na lewy dolny narożnik pulpitu:
    SetWindowPos(hWnd, HWND_TOPMOST, 1, toppos, 55, 22, SWP_SHOWWINDOW);
    UpdateWindow(hWnd);

    // Utwórz wskaźnik h wykorzystywany do opóźniania pętli:
    h = CreateEvent(0, TRUE, FALSE, "et");

    // Teraz podrzuć przycisk:
    for (i = 0; i < 50; i++)
    {
        toppos = toppos - 4;
        SetWindowPos(hWnd, HWND_TOPMOST, 1, toppos, 55, 22, SWP_SHOWWINDOW);
        WaitForSingleObject(h, 15); // 15-milisekundowe opóźnienie
    }

    // I opuść przycisk:
    for (i = 50; i > 0; i--)
    {
        toppos = toppos+4;
        SetWindowPos(hWnd, HWND_TOPMOST, 1, toppos, 55, 22, SWP_SHOWWINDOW);
        WaitForSingleObject(h, 15); // 15-milisekundowe opóźnienie
    }
}
```

Aby odpowiednio umieścić przycisk *Start* na ekranie, musimy znać pionową rozdzielcość pulpitu. Można ją odczytać następującym wierszem kodu:

```
int toppos = GetSystemMetrics(SM_CSCREEN) - 24;
```

Funkcja GetSystemMetrics zwraca wybrane ustawienie parametrów systemowych. Wybór dokonywany jest parametrem wywołania (tu mamy SM_CSCREEN, czyli wysokość ekranu). Od wyniku odejmujemy 24 (rozmiar obrazka plus 2 piksele) i zapisujemy w zmiennej toppos.

Obliczyliśmy w ten sposób współrzędną pionową okna przycisku. Teraz powinniśmy przesunąć okno na tę pozycję. Warto też sobie zapewnić, aby nasze okno pozostało zawsze ponad innymi oknami (zawsze „na wierzchu”). Obie te operacje można wykonać wywołaniem jednej funkcji: SetWindowPos. Przyjmuje ona siedem parametrów:

- ◆ Przesunięcie okna — określamy odpowiednio.
- ◆ Pozycjonowanie okna na pulpicie. Chcemy, aby nasze okno było na wierzchu, przekazujemy więc znacznik HWND_TOPMOST.
- ◆ Cztery parametry określające prostokąt, w którym okno ma być umieszczone. Współrzędna pozioma lewego górnego narożnika okna to 4, a pionowa równa jest wartości toppos. Szerokość i wysokość okna powinna pozostać równa rozmiarom obrazka. W zależności od posiadanego obrazka może zajść potrzeba dostosowania współrzędnej poziomej albo pionowej narożnika okna.
- ◆ Sposób wyświetlania okna — tu ma być wyświetlane, przekazujemy więc SWP_SHOWWINDOW.

Następnie odrysowujemy okno na nowej pozycji wywołaniem funkcji UpdateWindow(hWnd). Parametr wywołania wskazuje okno do odrysowania.

Ostatnią czynnością jest utworzenie zdarzenia pustego za pomocą funkcji CreateEvent. Zdarzenie to przyda nam się nieco później; jako że jest puste, nie sprowokuje żadnych czynności.

Nasze okno jest prawidłowo rozmieszczone i możemy przejść do animacji (wrażenia ruchu okna na ekranie). Służy do tego specjalna pętla:

```
for (i = 0; i < 50; i++)
{
    toppos = toppos - 4;
    SetWindowPos(hWnd, HWND_TOPMOST, 1, toppos, 55, 22, SWP_SHOWWINDOW);
    WaitForSingleObject(h, 15); // 15-milisekundowe opóźnienie
}
```

Na samym jej początku pozycja pionowa okna jest zmniejszana o 4 piksele. Ma to dać efekt przesunięcia okna w górę. Następnie okno jest faktycznie przemieszczane do nowej pozycji.

Najciekawszy fragment kodu tej pętli to jej ostatni wiersz, w którym mamy wywołanie funkcji WaitForSingleObject. Funkcja ta oczekuje na zajście zdarzenia określonego pierwszym parametrem wywołania. Drugi parametr określa maksymalny czas oczekiwania.

Ponieważ przekazaliśmy pusty opis zdarzenia, nigdy się takiego zdarzenia nie doczekaemy, więc funkcja po prostu odczeka zadany czas. W efekcie uzyskamy opóźnienie pomiędzy kolejnymi przesunięciami okna bez marnowania zasobów systemowych.

Największą zaletą funkcji `WaitForSingleObject` jest to, że w czasie oczekiwania nie zajmuje ona czasu procesora. Niektórzy programiści konstruują pętle opóźniające, wykorzystując w nich jakieś długotrwałe obliczenia. Jest to sposób nieefektywny, ponieważ przeciąża procesor zbędnymi operacjami. Z mojego doświadczenia wynika, że stosowanie funkcji `WaitForSingleObject` niemal w ogóle nie obciąża procesora, a działa znakomicie — znacznie dokładniej niż pętle zbędnych instrukcji.

Powyzsza pętla daje efekt podrzucania okna w górę. Po jej wykonaniu powinniśmy opuścić okno przycisku z powrotem. Służy do tego kolejna pętla, która, analogicznie jak poprzednia, zmienia pozycję pionową okna.

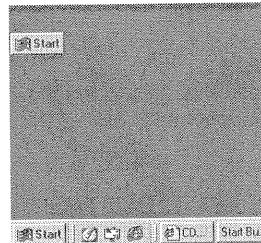
Wszystko jest już gotowe; wróćmy do funkcji `_tWinMain` i umieścmy w niej wywołanie funkcji `DrawStartButton`. Zalecam umieszczenie go zarówno przed pętlą komunikatów, jak i w jej wnętrzu:

```
DrawStartButton();

// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    DrawStartButton();
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Kiedy program zostanie uruchomiony, nasz fałszywy przycisk *Start* podfrunie i opadnie, zasłaniając właściwy przycisk. Jeśli użytkownik spróbuje przesunąć wskaźnik myszy nad rzeczywisty przycisk, umieści go nad fałszywką i zdarzenie przesunięcia myszy otrzyma nasz program. W efekcie nastąpi ponowne podrzucenie fałszywego przycisku (patrz rysunek 2.5).

Rysunek 2.5.
Program w działaniu



To bardzo przyjemny dowcip i nikomu nie szkodzi. Jest przy tym efektowny. Zalożejmy, że będzie z niego niezły ubaw.



Kod źródłowy programu oraz pliki wykonywalne znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział2\Start Button. Aby uruchomić program w Visual C++, wybierz polecenie *Start* z menu *Debug*.

Na bazie efektu zasłonięcia standardowych elementów systemu własnym obiektem z odpowiednio spreparowanym wyglądem można skonstruować jeszcze wiele dowcipów. Można wystraszyć użytkownika, wyświetlając fałszywe okno DOS-a z komunikatem w stylu „Formatowanie dysku C:”. Jeśli wyświetlisz dziesięć komunikatów „Uwaga, wirus!”, oszukasz niejednego zawodowca i wywołasz uśmiech na twarzach jego kolegów.

Pamiętam jeszcze stare, dobre lata dziewięćdziesiąte, kiedy nawet wirusy były nieszkodliwe. Wyświetlały zabawne komunikaty, odtwarzaly dźwięki za pomocą głośniczka PC albo wyświetlały jakieś rysunki komponowane ze znaków ASCII. Ich największym grzechem było powielanie siebie i przenoszenie pomiędzy komputerami. Oczywiście wirusy to wirusy — nawet nieszkodliwe są niepożądane — ale tamte były przynajmniej odkrywcze. Współczesne wirusy nie są ani ciekawe, ani też nie stanowią szczególnych osiągnięć programistycznych. Ich jedynym celem jest infekowanie jak największej liczby komputerów i sianie jak największych spustoszeń.

2.2. Zaczni j pracę od przycisku Start

Jeśli instalowałeś kiedyś Windows, zapewne miałeś okazję zobaczyć na pasku zadań komunikat w rodzaju „Zaczni j pracę, naciśkając przycisk *Start*” ze strzałką wskazującą przycisk *Start*. Swego czasu pracowałem jako administrator sieci komputerowej i byłem zasypywany pytaniemi użytkowników o położenie tej czy innej aplikacji w systemie. Wreszcie zirytowałam się tak mocno, że napisałem program, który na stałe otwierał menu wyświetlane po naciśnięciu przycisku *Start*. Pokażę Ci teraz podobny program.

Utwórz w Visual C++ nowy projekt, wybierając jako typ *Win32 Project*. Ja nazwałem swój *CrazyStart*, możesz jednak nadać mu dowolną inną nazwę. W tym przykładzie nazwa projektu nie jest никак wykorzystywana.

Otwórz główny plik kodu źródłowego projektu. Powinienn on mieć nazwę zgodną z nazwą projektu i rozszerzenie CPP (u mnie jest to *CrazyStart.cpp*). Znajdź w nim funkcję `_tWinMain` i zmień ją zgodnie z listingu 2.4. Skorzystaj z komentarzy.

Listing 2.4. Funkcja `_tWinMain`

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPTSTR lpCmdLine,
                        int nCmdShow)
{
    // TODO: Place code here.
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
```

```

LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
LoadString(hInstance, IDC_CRAZYSTART, szWindowClass, MAX_LOADSTRING);
MyRegisterClass(hInstance);

// Perform application initialization:
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}

hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_CRAZYSTART);

// Main message loop:
// dodaj do kodu poniższe trzy wiersze:
HWND hTaskBar, hButton;

hTaskBar = FindWindow("Shell_TrayWnd",NULL);
hButton = GetWindow(hTaskBar, GW_CHILD);

while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    // Naciśnięcie przycisku Start
    // dodaj do kodu poniższe wiersze:
    SendMessage(hButton, WM_LBUTTONDOWN, 0, 0);
    Sleep(1000);
}

return (int) msg.wParam;
}

```

Na początku deklarujemy dwie zmienne typu `HWND`: `hTaskBar` i `hButton`. Wiemy już, że typ `HWND` służy do przechowywania uchwytów okien. Następnie wywołujemy funkcję `FindWindow` wyszukującą w systemie okno o zadanych parametrach. Funkcja przyjmuje dwa parametry:

- ◆ Nazwę klasy okna — nazwę wykorzystywaną przy rejestraniu klasy okna w systemie.
- ◆ Nazwę okna — ciąg z belki tytułowej okna.

Przycisk `Start` umieszczony jest na pasku zadań, więc oknem, którego szukamy, jest okno paska zadań. Jego klasa to `Shell_TrayWnd` i taki napis przekazujemy pierwszym parametrem wywołania funkcji. Okno nie ma belki tytułowej, więc drugi parametr ustawiamy na `NULL`.

Okno to zawiera tylko jeden przycisk — `Start`. Jest to pierwszy przycisk okna paska zadań. Wiedząc to, możemy dobrać się do jego uchwytu funkcją `GetWindow`. Wymaga ona przekazania dwóch parametrów:

- ◆ Uchwytu okna.
- ◆ Relacji pomiędzy oknem a szukanym obiektem. Nasz przycisk znajduje się w oknie. Okno jest więc jego „rodzicem” (obiektem nadzorowanym) — przycisk jest z kolei dla okna „dzieckiem” (obiektem potomnym). Dlatego relację określamy jako `GW_CHILD`.

Otrzymujemy w ten sposób uchwyt przycisku, który zapisujemy w zmiennej `hButton`. W głównej pętli komunikatów przesyłamy do przycisku `Start` komunikat, wywołując funkcję `SendMessage`. Ta przyjmuje również dwa parametry:

- ◆ Okno, do którego kierowany jest komunikat. Przekazujemy uchwyt przycisku `Start`.
- ◆ Komunikat. Przesyłamy `WM_LBUTTONDOWN`, czyli komunikat naciśnięcia lewego przycisku myszy.

Po otrzymaniu takiego komunikatu przycisk `Start` „sądzi”, że został naciśnięty przez użytkownika i wyświetla menu programów.

Dalej wywoływana jest funkcja `Sleep`, która realizuje opóźnienie o zadanej liczbie milisekund. Jeśli parametrem wywołania jest 1 000, opóźnienie będzie dokładnie jednosekundowe. Funkcja `Sleep` zawiesza co prawda wykonanie programu, ale obciąża procesor w stopniu nieco większym niż `WaitForSingleObject`. Kiedy użytkownik chce zamknąć okno, musi przesunąć nad to okno wskaźnik myszy. Spowoduje to wygenerowanie kilku komunikatów myszy. Opóźnienie wprowadzone funkcją `Sleep` będzie tak duże, że użytkownikowi trudno będzie zamknąć okno.



Kod źródłowy programu oraz pliki wykonywalne znajdują się na dołączonej do książki płytcie CD-ROM, w podkatalogu \Przykłady\Rozdział2\CrazyStart.

2.3. Zamieszanie z przyciskiem Start

Z przyciskiem `Start` można bawić się naprawdę długo. Kolejny żart polegać będzie na schowaniu tego przycisku.

Na potrzeby następnego przykładu możemy albo utworzyć nową aplikację, albo skorzystać z kodu poprzedniego przykładu, zmieniając jedynie funkcję `_tWinMain` tak jak na listingu 2.5.

Listing 2.5. Nowa wersja funkcji `_tWinMain`

```

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR   lpCmdLine,
                       int      nCmdShow)
{
    // TODO: Place code here.
    MSG msg;

```

```

HACCEL hAccelTable;

// Initialize global strings
LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
LoadString(hInstance, IDC_CRAZYSTART, szWindowClass, MAX_LOADSTRING);
MyRegisterClass(hInstance);

// Perform application initialization:
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}

hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_CRAZYSTART);

// Main message loop:
HWND hTaskBar, hButton;

hTaskBar = FindWindow("Shell_TrayWnd", NULL);
hButton = FindWindowEx(hTaskBar, 0, "Button", NULL);

while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // Schowaj przycisk Start
    ShowWindow(hButton, SW_HIDE);
    // odczekaj 50 milisekund:
    Sleep(50);
    // Pokaż przycisk Start
    ShowWindow(hButton, SW_SHOW);
    Sleep(50);
}

return (int) msg.wParam;
}

```

W tym przykładzie również szukamy przycisku *Start* na pasku zadań. Istotna różnica w stosunku do poprzedniego przykładu tkwi w obsłudze komunikatów. Korzystamy tu z funkcji *ShowWindow*. Wiadomo już, że służy ona do oznaczania okna jako widoczne. Można jednak funkcję tę wykorzystać również do maksymalizacji, minimalizacji i ukrywania okien.

W systemie Windows przyciski również są oknami, więc tę samą funkcję możemy wykorzystać odnośnie przycisku *Start*. Wywołujemy ją dwukrotnie, za każdym razem przekazując pierwszym parametrem uchwyt odnalezionej wcześniej okna przycisku. Drugi parametr jest za pierwszym razem znacznikiem *SW_HIDE* (ukrywamy okno), a za drugim razem — znacznikiem *SW_SHOW* (pokazujemy okno). Wywołanie funkcji *Sleep* w międzyczasie ma za zadanie opóźnić wywołania funkcji, tak aby użytkownik był w stanie zauważyć efekt działania programu.

Uruchom program, a zobaczysz znikający i pojawiający się przycisk *Start*. Teraz już wiesz, jak pisać kod ukrywający inne ważne okna Windows, tak aby użytkownik nie mógł z nich korzystać.

Program ten różni się od poprzedniego również sposobem wyszukiwania okna przycisku w oknie paska zadań. Poprzednio korzystaliśmy z funkcji *GetWindow*, tu mamy zaś wywołanie funkcji *FindWindowEx*. Działa ona podobnie jak *FindWindow*, tyle że pozwala na dokładniejsze określenie szukanego obiektu. Można za jej pomocą wyszukiwać nie tylko okna główne programów, ale również ich okna potomne. Funkcja przyjmuje następujące parametry:

- ◆ Okno, w którym wyszukujemy obiekt. Pozwala to na ograniczenie poszukiwań do konkretnego okna.
- ◆ Obiekt, od którego mają się rozpocząć poszukiwania. Podanie 0 oznacza przeszukiwanie wszystkich obiektów potomnych okna od pierwszego elementu sterującego.
- ◆ Klasy obiektu. Szukamy przycisku, podajemy więc ciąg *Button*.
- ◆ Nazwę obiektu. Jeśli podamy *NULL*, przeszukane zostaną wszystkie obiekty danej klasy.



Kod źródłowy programu oraz pliki wykonywalne znajdują się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział2\StartMusic.

Modyfikując nieco kod, możemy wywołać na pasku zadań całkowity chaos, jak to robi program z listingu 2.6.

Listing 2.6. Zamieszanie na całości paska zadań

```

int APIENTRY _tWinMain(HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPTSTR lpCmdLine,
                        int nCmdShow)
{
    ...
    HWND hTaskBar, hButton;

    hTaskBar = FindWindow("Shell_TrayWnd", NULL);

    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        // Schowaj pasek zadań
        ShowWindow(hTaskBar, SW_HIDE);
        // odczekaj 100 milisekund:
        Sleep(100);
    }
}

```

```

    // Pokaż pasek zadań
    ShowWindow(hTaskBar, SW_SHOW);
    Sleep(100);
}

return (int) msg.wParam;
}

```



Kod źródłowy programu oraz pliki wykonywalne znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział2\Tasks.

2.4. Więcej dowcipów z paskiem zadań

Jak już Ci wiadomo, pasek zadań to zwykłe okno i możemy go modyfikować za pomocą wszystkich funkcji WinAPI manipulujących oknami. W pierwszym przykładzie z tego rozdziału podrzucaliśmy w góre fałszywy przycisk *Start*. Możemy jednak ulepszyć żart, podrzucając prawdziwy przycisk — wiemy już przecież, jak się do niego dobrać.

Nie będzie to jednak tak całkiem proste. Będziemy musieli poznać kilka sztuczek.

Utwórz nowy projekt i umieść w pliku kodu źródłowego, w sekcji zmiennych globalnych, następujące deklaracje zmiennych:

```

HWND hWnd;
HWND hTaskBar, hButton;
HWND MainMenuBar;

```

Deklarujemy tu cztery zmienne będące uchwytymi okien:

- ◆ hWnd — zmienna do przechowywania uchwytu okna głównego programu, tak by można się było odwołać do tego okna z dowolnego miejsca programu.
- ◆ hTaskBar i hButton — zmienne uchwytów okna paska zadań i okna przycisku *Start*.
- ◆ MainMenuBar — zmienna przeznaczona do przechowywania uchwytu menu naszego programu, zadeklarowana nieco na zapas. Jej zadanie wyjaśni się nieco później.

Przejdz do funkcji *_tWinMain* i dodaj do niej kod z listingu 2.7.

Listing 2.7. Kod, który powinien uzupełnić funkcję *_tWinMain*

```

hTaskBar = FindWindow("Shell_TrayWnd", NULL);
hButton = GetWindow(hTaskBar, GW_CHILD);
MainMenuBar = LoadMenu(hInstance, (LPCSTR)IDC_STARTENABLE);
SetParent(hButton, 0);

```

```

int i;
HANDLE h;
int topPos = GetSystemMetrics(SM_CYSCREEN) - 24;

// Ustaw okno w lewym dolnym narożniku pulpitu:
SetWindowPos(hButton, HWND_TOPMOST, 1, topPos, 55, 22, SWP_SHOWWINDOW);
UpdateWindow(hButton);
// Utwórz wskaźnik zdarzenia h wykorzystywany w implementacji opóźnienia:
h = CreateEvent(0, TRUE, FALSE, "et");

// Podrzuć przycisk:
for (i = 0; i < 50; i++)
{
    topPos = topPos - 4;
    SetWindowPos(hButton, HWND_TOPMOST, 1, topPos, 55, 22, SWP_SHOWWINDOW);
    WaitForSingleObject(h, 15); // 15-milisekundowe opóźnienie
}
for (i = 50; i > 0; i--)
{
    topPos = topPos + 4;
    SetWindowPos(hButton, HWND_TOPMOST, 1, topPos, 55, 22, SWP_SHOWWINDOW);
    WaitForSingleObject(h, 15); // 15-milisekundowe opóźnienie
}
SetParent(hButton, hTaskBar);

```

Pierwsze dwa wiersze nie powinny sprawić niespodzianki. Wyszukuję one w systemie okno paska zadań oraz okno przycisku *Start* i zwracając ich uchwyty, zapisywane w zmiennych globalnych. Dlaczego wykorzystujemy we wnętrzu funkcji zmienne globalne, zamiast lokalnych? Zamierzamy bowiem wykorzystywać je również w pozostałych żartach w programie, więc w ten sposób oszczędzimy sobie każdorazowego wyszukiwania uchwytów, skoro i tak zawsze będą one identyczne.

W trzecim wierszu wczytujemy do zmiennej *MainMenuBar* uchwyt menu; posługujemy się funkcją *LoadMenu*. Funkcja ta wymaga przekazania dwóch parametrów: uchwytu egzemplarza programu i nazwy wczytywanego menu. Menu przyda się nam później; tutaj tylko przygotowujemy je do użycia.

W funkcji *_tWinMain* ograniczamy się do podrzucenia przycisku *Start*. Zanim to zrobić, powinniśmy przypomnieć sobie, gdzie się on znajduje. Otóż przycisk należy do paska zadań i znajduje się na tymże pasku. Jeśli spróbujemy przesunąć przycisk w góre tak po prostu, przycisk nie drgnie. Dlaczego? Ponieważ nie możemy go zdjąć z paska zadań. Powinniśmy wcześniej zerwać połączenie pomiędzy przyciskiem a paskiem zadań. W tym celu wywołujemy funkcję *SetParent* nadającą przyciskowi nowego „rodzica”. Przyjmuje ona następujące parametry:

- ◆ Uchwyt okna, które ma zyskać nowego rodzica.
- ◆ Uchwyt okna, które ma zostać rodzicem.

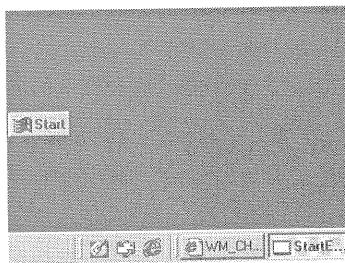
Jako pierwszy parametr podajemy uchwyt przycisku, a drugim przekazujemy 0. Po zakończeniu działania funkcji przycisk nie będzie miał żadnego rodzica, co spowoduje zerwanie więzi pomiędzy nim a oknem paska zadań.

Teraz można do woli przesuwać przycisk po ekranie. Kod podrzucający przycisk powinien być dla Czytelnika znajomy. Widzieliśmy go już na listingu 2.1; tam przesuwane było okno fałszywego przycisku. Tu przesuwamy przycisk, więc pierwszym parametrem wywołania SetWindowPos jest uchwyt tego przycisku.

Po podniesieniu przycisku i jego opuszczeniu zwracamy go jego pierwotnemu rodzicowi, kojarząc przycisk ponownie z oknem paska zadań. W tym celu znów wywołujemy funkcję SetParent.

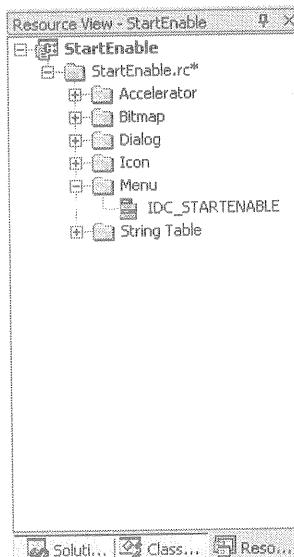
Możesz uruchomić przykład i sprawdzić jego działanie. Efekt powinien być podobny do tego z rysunku 2.6. Zwróć uwagę na puste miejsce na pasku zadań po przycisku *Start*.

Rysunek 2.6.
Efekt działania programu



Aby zebrać wszystkie dotychczasowe dowcipy w jednym programie, wyposażymy program w kilka pozycji menu i za ich pośrednictwem będziemy wywoływać poszczególne efekty. Aby utworzyć menu, otwórz w Visual C++ przeglądarkę zasobów projektu (rysunek 2.7). Wybierz z gałęzi *Menu* odpowiednią pozycję. Uruchomi się edytor menu.

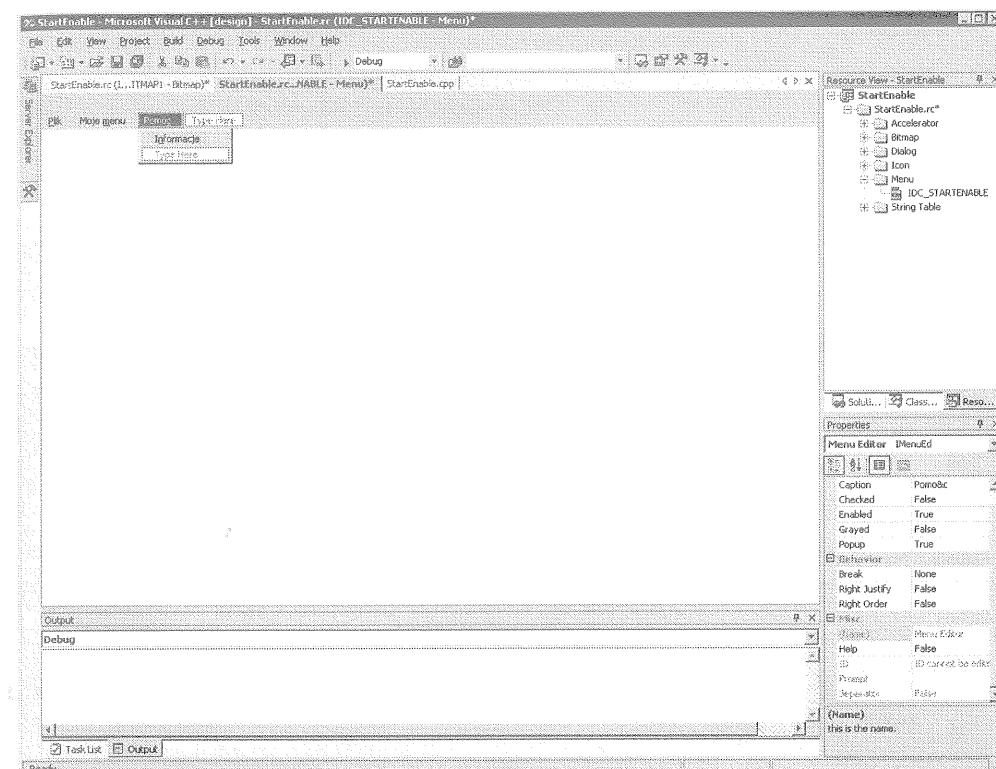
Rysunek 2.7.
Okno zasobów z elementem w gałęzi *Menu*



W oknie edytora menu zobaczysz menu, w którym możesz się poruszać. Pod koniec każdego menu rozwijanego znajduje się pozycja *Type Here* na jasnym tle. Wybierz taką pozycję znajdująca się na skraju menu i wpisz nazwę nowej pozycji menu, np.

Moje menu. Zmieni się napis i tło pozycji, a do zasobów dodana zostanie nowa pozycja menu. W podobny sposób można stworzyć kompletne menu wykorzystywane w programie¹.

Wybierz pozycję *Moje menu* i przeciągnij je do pozycji *Help*, jak na rysunku 2.8.

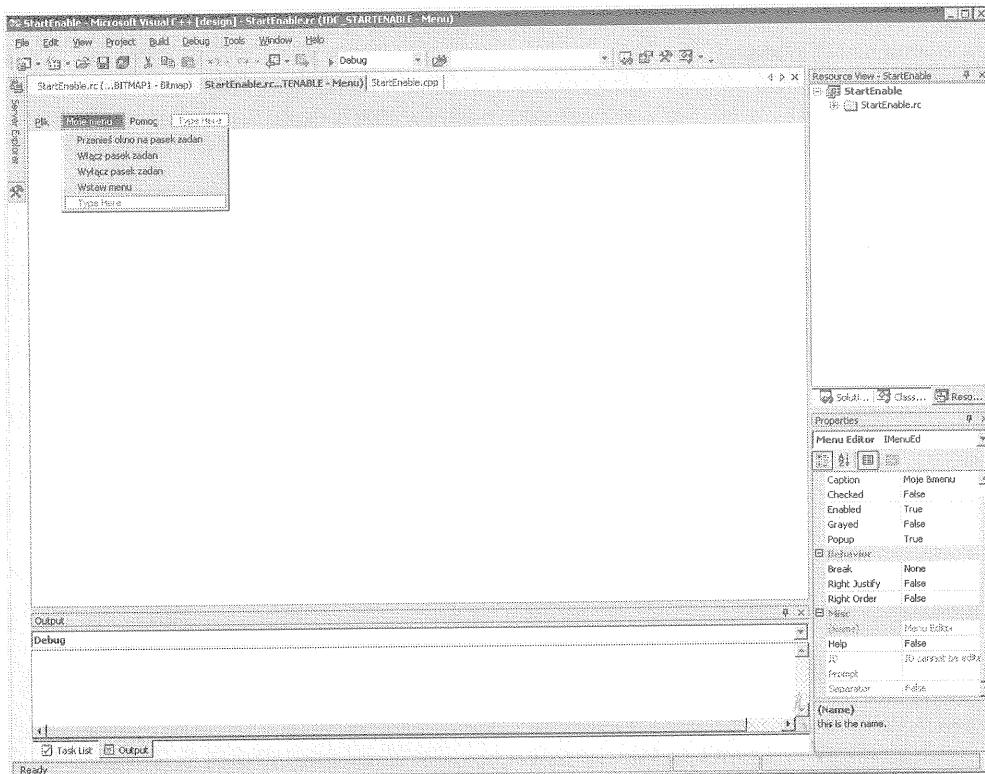


Rysunek 2.8. Edytor menu w oknie Visual C++

Przejdz do menu *Moje menu*, a w nim do pozycji *Type Here* i wpisz nową nazwę Przenieś okno na pasek zadań. W menu pojawi się nowa pozycja, poniżej której wciaż widoczna będzie pozycja *Type Here*. Przejdz do niej i wpisz kolejną nazwę — *Włącz pasek zadań*. Utwórz tak jeszcze dwie pozycje menu: *Włącz pasek zadań* i *Wstaw menu*. Powinieneś otrzymać menu takie jak na rysunku 2.9.

Wróćmy do programowania. Przejdz do pliku kodu źródłowego i znajdź w nim funkcję *WndProc*. Kiedy użytkownik wybiera w programie pozycję menu, system generuje komunikat. Obsługujemy go właśnie w funkcji *WndProc*.

¹ Jeśli Czytelnik chce konstruować (jak w tym przykładzie) menu polskojęzyczne, z polskimi znakami diakrytycznymi, powinien ustawić język zasobu menu na *Polski*. Wymaga to wskazania wpisu menu w przeglądarce zasobów i w oknie właściwości wybrania pozycji *Polski* z listy rozwijanej *Language* — przyp. tłum.



Rysunek 2.9. Nowo utworzone menu naszego programu

Kompletny kod funkcji WndProc jest pokazany na listingu 2.8. Uzgodnij z nim kod swojego programu.

Listing 2.8. Funkcja obsługi komunikatów — WndProc

```
HRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_COMMAND:
            wmId = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // Sprawdź wybór pozycji menu:
            switch (wmId)
            {
                // Obsługa menu
                // Przenieś okno na pasek zadań
                case ID_MOJEMENU_PRZENIESOKNONAPASEKZADAN:
                    SetParent(hWnd, hTaskBar);
                    break;
            }
    }
}
```

```
// Włącz pasek zadań
case ID_MOJEMENU_WLACZPASEKZADAN:
    EnableWindow(hTaskBar, true);
    break;
// Wyłącz pasek zadań
case ID_MOJEMENU_WYLACZPASEKZADAN:
    EnableWindow(hTaskBar, false);
    break;
// Wstaw menu
case ID_MOJEMENU_WSTAWMENU:
    SetMenu(hWnd, MainMenu);
    break;
case IDM_ABOUT:
    DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
    break;
case IDM_EXIT:
    DestroyWindow(hWnd);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
break;
case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    // TODO: Any drawing code here...
    EndPaint(hWnd, &ps);
    break;
case WM_DESTROY:
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
```

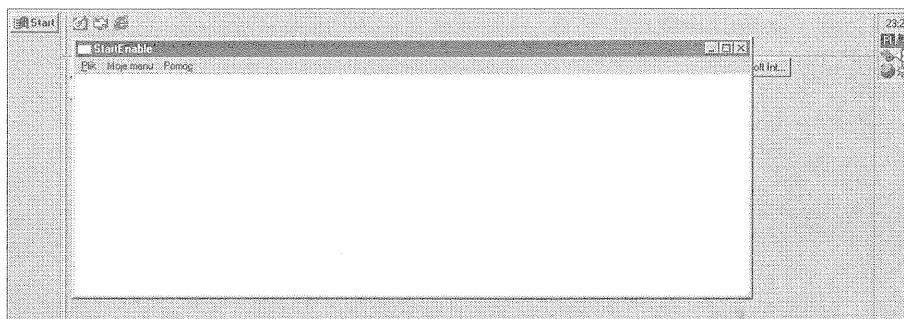
Większość powyższej funkcji została wygenerowana przez kreator kodu źródłowego projektu. My uzupełniliśmy ją jedynie o kilka drobnych elementów. Przyjrzyjmy się im z bliska. Wywołanie pierwszej pozycji menu (*Przenieś okno na pasek zadań*) jest przetwarzane następującym kodem:

```
// Przenieś okno na pasek zadań
case ID_MOJEMENU_PRZENIESOKNONAPASEKZADAN:
    SetParent(hWnd, hTaskBar);
    break;
```

Instrukcja case porównuje wartość komunikatu ze stałą `ID_MOJEMENU_PRZENIESOKNONAPASEKZADAN`. Jeśli otworzysz okno edytora zasobów i zaznaczysz w nim utworzoną wcześniej pozycję menu *Przenieś okno na pasek zadań*, zobaczysz, że pozycji tej została przypisana właśnie taka stała. Jeśli będzie tam inna stała (Visual C++ mógł wygenerować inną stałą, np. w innej wersji środowiska programistycznego), wystarczy poprawić kod albo stałą tak, aby odwołania były zgodne.

Jeśli porównanie wypadnie pomyślnie, procesor przechodzi do wykonania kolejnych wierszy kodu aż do instrukcji break. Kod ten składa się z pojedynczego wywołania funkcji SetParent. Wiemy już, że funkcja ta służy do zmiany rodzica okna wskazywanego pierwszym parametrem wywołania — nowym rodzicem zostaje okno wskazane drugim parametrem wywołania. W naszym przypadku pierwszy parametr to uchwyt okna głównego programu, a drugi — uchwyt okna paska zadań. W ten sposób okno naszego programu staje się oknem potomnym okna paska zadań.

Spójrz na rysunek 2.10. Widać na nim efekt wybrania z menu programu właśnie tej pozycji. Specjalnie powiększyłem pasek zadań, tak aby było jasne, że okno programu jest ograniczone do okna paska zadań. Okno programu nie może zostać przeniesione poza pasek zadań, dopóki pozostaje jego oknem potomnym.



Rysunek 2.10. Głównie okno programu staje się oknem potomnym paska zadań

Po kliknięciu w menu pozycji *Wyłącz pasek zadań* program wykonuje następujący kod:

```
// Włącz pasek zadań
case ID_MOLEMENU_WYLACZPASEKZADAN:
    EnableWindow(hTaskBar, true);
    break;
```

Kod ten wywołuje funkcję EnableWindow, która włącza albo wyłącza okno. Jej pierwszym parametrem jest uchwyt okna, a drugi to wartość logiczna — true, jeśli okno ma być włączone, albo false, kiedy ma być wyłączone. Tutaj wyłączamy okno, które przestaje być dostępne dla użytkownika. Można w nieskończoność kliknąć elementy wyłączonego okna, jedynym efektem będzie jednak dźwięk — brzęczyk błędu. Moglibyśmy alternatywnie, zamiast całego paska zadań, zablokować jedynie przycisk *Start* (przekazując w wywołaniu uchwyt hButton).

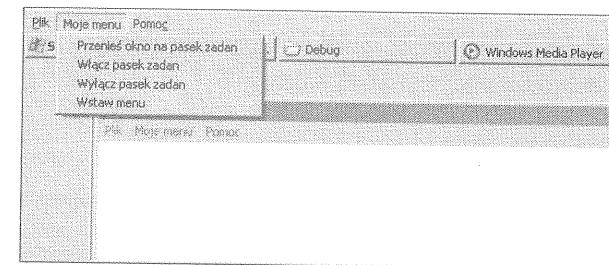
Po wybraniu z menu pozycji *Włącz pasek zadań* wywołana zostanie ta sama funkcja EnableWindow, tym razem jednak okno paska zadań zostanie przez nią włączone.

Jeśli wybierzesz pozycję *Wstaw menu*, w ramach obsługi komunikatu wywołana zostanie funkcja SetMenu. Przypisuje ona do okna wskazanego pierwszym parametrem wywołania menu określone drugim parametrem. Tutaj przydaje się wreszcie pozyskany wcześniej uchwyt menu zapisany w zmiennej MainMenu.

Rzuć okiem na rysunek 2.11, na którym prezentowany jest efekt działania programu po wywołaniu polecenia *Wstaw menu*. Co ciekawe, do tego menu nie można odwołać się za pośrednictwem myszy. Trzeba to robić za pomocą klawiatury. W tym celu należy uczynić okno paska zadań aktywnym (kliknąć pasek zadań) i nacisnąć przycisk *Alt*. Podświetlone zostanie pierwsze menu rozwijane i dopiero wtedy będzie można wędrować po menu czy to za pomocą klawiatury, czy myszy.

Rysunek 2.11.

Menu na pasku zadań — java czy sen?



Kod źródłowy programu oraz pliki wykonywalne znajdują się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział2\StartEnable.

2.5. Inne żarty

Weźmy się za inne żarciki. Będą one tak drobne, że nie będziemy nawet tworzyć dla nich osobnych projektów przykładowych. Od razu połączymy wszystkie w jeden większy program. Będzie go można wykorzystać przy tworzeniu własnych dowcipów albo poważnych programów. Niektóre z wykorzystywanych w nim funkcji mogą być bowiem wykorzystane z powodzeniem również w oprogramowaniu komercyjnym.

Jak „zgasić” monitor?

Najlepiej chwycić gaśnicę. Pamiętam jak dziś — w dzieciństwie byłem członkiem drużyny młodych gaśnicowych... tzn. strażaków. Ale żarty na bok. Aby wygasnąć monitor, wystarczy wywołać funkcję SendMessage(hWnd, WM_SYSCOMMAND, SC_MONITORPOWER, 0). Aby ponownie załączyć wyświetlanie, należy wywołać tę samą funkcję, ale ostatni parametr zastąpić wartością -1.

Jak uruchamiać systemowe pliki CPL?

Przede wszystkim trzeba włączyć do programu plik nagłówkowy *shellapi.h*, tak aby można było wywołać w programie funkcję *ShellExecute*:

```
#include <shellapi.h>
```

Potem wystarczy zaopatrzyć program w następujące wywołanie:

```
ShellExecute(hWnd, "Open", "Rundll32.exe",
    "shell32,Control_RunDLL nazwapliku.cpl", "",
    SW_SHOWNORMAL);
```

Funkcja ShellExecute uruchamia program. Wymaga przekazania następujących parametrów:

- ◆ Okna, które inicjuje uruchomienie programu.
- ◆ Rodzaju czynności do wykonania. Chcemy uruchomić program, więc przekazujemy "Open" (otwórz).
- ◆ Programu do uruchomienia.
- ◆ Wiersza polecenia uruchomienia programu (wiersz parametrów uruchomienia).
- ◆ Katalogu uruchomionego programu. Jeśli podamy ciąg pusty, program będzie korzystał ze ścieżki domyślnej.
- ◆ Typu uruchomienia. Parametr ten określa, jak należy zainicjować program. Znacznik SW_SHOWNORMAL wymusza uruchomienie okna programu w zwykłym trybie (znaczenie znacznika jest identyczne, jak w funkcji ShowWindow).

W naszym przypadku wywołaniem ShellExecute wyrażamy chęć uruchomienia pliku *Rundll32.exe* (programu uruchomieniowego plików DLL i CPL). Do tego programu przekazywany jest ciąg zadany czwartym parametrem: shell32,Control_RunDLL nazwapliku.cpl.

Okno ustawień internetowych wywołuje się na przykład tak:

```
ShellExecute(hWnd, "Open", "Rundll32.exe",
    "shell32,Control_RunDLL inetcpl.cpl", "",
    SW_SHOWNORMAL);
```

A okno właściwości ekranu tak:

```
ShellExecute(hWnd, "Open", "Rundll32.exe",
    "shell32,Control_RunDLL desk.cpl", "",
    SW_SHOWNORMAL);
```

Jak wysunąć tarcę napędu CD-ROM?

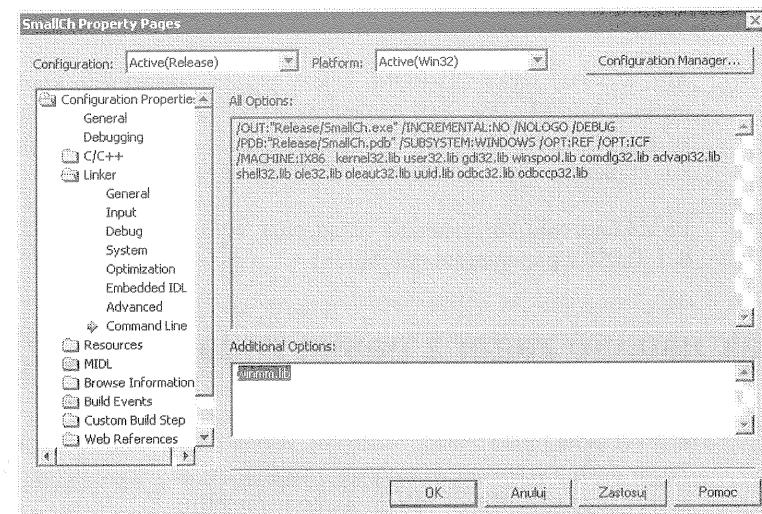
Niespodziewane wysunięcie lub schowanie tarczy napędu CD-ROM to świetny dowcip. Można powtarzać go na okrągło, wysuwając i chowając tarczę w pętli. W tym przykładzie poprzestaniemy na jednorazowym wybryku.

Na początek musimy włączyć do programu plik nagłówkowy *mmsystem.h* bądź *stdafx.h*:

```
#include <mmsystem.h>
```

W oknie przeglądarki projektu *Solution Explorer* wybierz nazwę swojego projektu i wybierz polecenie *Properties* z menu *Project*. W hierarchii z lewej strony wybierz *Configuration Properties/Linker/Command Line*. Funkcje, których będziemy potrzebować, są skompilowane w pliku bibliotecznym *winmm.lib*, który domyślnie nie jest konsolidowany z projektem w czasie generowania pliku wykonywalnego. Dlatego trzeba ręcznie dodać tę bibliotekę do projektu. W tym celu wprowadź w polu *Additional Options* (rysunek 2.12) nazwę *winmm.lib*.

Rysunek 2.12.
Ustawienia
konsolidacji projektu



Będziemy też potrzebować kilku zmiennych:

```
MCI_OPEN_PARMS OpenParm;
MCI_SET_PARMS SetParm;
MCIDEVICEID dID;
```

Oto kod otwierający i zamkijający tarcę CD-ROM:

```
OpenParm.lpszDeviceType = "CDAudio";
mciSendCommand(0, MCI_OPEN, MCI_OPEN_TYPE, (DWORD_PTR)&OpenParm);
dID = OpenParm.wDeviceID;
mciSendCommand(dID, MCI_SET, MCI_SET_DOOR_OPEN, (DWORD_PTR)&SetParm);
mciSendCommand(dID, MCI_SET, MCI_SET_DOOR_CLOSED, (DWORD_PTR)&SetParm);
mciSendCommand(dID, MCI_CLOSE, MCI_NOTIFY, (DWORD_PTR)&SetParm);
```

Na samym początku należy wypełnić zmienną *lpszDeviceType* struktury *OpenParm*. Trzeba jej przypisać ciąg *CDAudio*, sygnalizując, że chodzi nam o sterowanie napędem CD-ROM.

Do obsługi urządzeń multimedialnych, do których zalicza się też napęd CD-ROM, służy funkcja *mciSendCommand*. Wysyła ona do urządzenia odpowiednio spreparowany komunikat. Wymaga przekazania czterech parametrów:

- ◆ Identyfikatora urządzenia, do którego kierowany jest komunikat. Identyfikator ten otrzymujemy przy otwieraniu urządzenia. Jeśli więc drugim parametrem wywołania jest *MCI_OPEN*, parametr pierwszy jest ignorowany — urządzenie nie zostało jeszcze otwarte, więc parametr na pewno nie zawiera poprawnego identyfikatora.

- ◆ Rodzaju komunikatu.
- ◆ Znacznika komunikatu grającego rolę parametru polecenia.
- ◆ Wskaźnika struktury przechowującej parametry komunikatu.

Za pierwszym razem wysyłamy komunikat MCI_OPEN otwierający urządzenie. W wyniku wywołania pole wDeviceID struktury OpenParm będzie zawierać identyfikator otwartego urządzenia. Jego wartość będziemy wykorzystywać w kolejnych wywołaniach w roli identyfikatora urządzenia.

Aby wysunąć tarcę CD-ROM, należy przesłać do napędu komunikat MCI_SET z parametrem MCI_SET_DOOR_OPEN. Nie martw się na razie wartością ostatniego parametru. Tarcę wsuwa się z powrotem prawie tak samo jak się ją wysuwa — inny jest jedynie parametr komunikatu mający postać MCI_SET_DOOR_CLOSED.

Po zakończeniu korzystania z urządzenia należy je zamknąć. Służy do tego komunikat MCI_CLOSE z parametrem MCI_NOTIFY.

Jak usunąć zegar z paska zadań?

To żart podobny do chowania przycisku *Start*. Na początek wypadałoby odszukać uchwyt okna paska zadań. Następnie w tym oknie okno zasobnika i dalej okno zegara. Zegar można schować wywołaniem funkcji ShowWindow z uchwytem okna zegara w roli pierwszego i SW_HIDE w roli drugiego parametru:

```
HWND Wnd;
Wnd = FindWindow("Shell_TrayWnd", NULL);
Wnd = FindWindowEx(Wnd, HWND(0), "TrayNotifyWnd", NULL);
Wnd = FindWindowEx(Wnd, HWND(0), "TrayClockWClass", NULL);
ShowWindow(Wnd, SW_HIDE);
```

Można też schować cały zasobnik systemowy (obszar z ikonami aplikacji, po prawej stronie paska zadań) od razu. Wystarczy usunąć z powyższego kodu czwarty wiersz.

Jak ukryć cudze okno?

Manipulowanie cudzymi oknami będzie omawiane jeszcze wielokrotnie w dalszych częściach książki. Na razie chciałbym zainteresować Czytelnika przykładem, w którym schowamy okno czegoś programu:

```
HWND Wnd;
while (TRUE)
{
    Wnd = GetForegroundWindow();
    if (Wnd > 0)
        ShowWindow(Wnd, SW_HIDE);
    Sleep(1000);
}
```

W powyższym kodzie uruchamiamy nieskończoną pętlę while, w której wykonywane są następujące czynności:

- ◆ Pobieramy uchwyt aktywnego okna (wywołaniem funkcji GetForegroundWindow()).
- ◆ Uchwyt jest poprawny, jeśli jest niezerowy. Wtedy ukrywamy okno wywołaniem ShowWindow.
- ◆ Oczekujemy jedną sekundę, aby dodatkowo zmylić użytkownika.

Po uruchomieniu tego kodu każde okno, które stanie się aktywne, zniknie w ciągu najdalej sekundy i użytkownik nie będzie w stanie korzystać z systemu. Jeśli nawet spróbuje wywołać okno menedżera zadań, aby przerwać działanie złośliwego programu, nie będzie mógł skorzystać z tego narzędzia, ponieważ jego okno zniknie, zanim użytkownik zdąży wybrać w nim kłopotliwy program i kliknąć przycisk *Zakończ zadanie*. Jeśli użytkownik zechce wyłączyć system i posłuży się przyciskiem *Start*, uaktywni pasek zadań, który zaraz potem zniknie.

Dlatego przed uruchomieniem tego programu zalecałbym zapisanie wszystkich otwartych dokumentów na dysku — inaczej jego działanie może doprowadzić do utraty niezapisanych informacji. Dodatkowo warto wyposażyć program w opcję przerywającą pętlę.

Jak ustawić własną tapetę pulpitu?

To całkiem proste:

```
SystemParametersInfo(SPI_SETDESKWALLPAPER, 0, "c:\\\\1.bmp",
    SPIF_UPDATEINIFILE);
```

Funkcja SystemParametersInfo przyjmuje cztery parametry:

- ◆ Kod czynności do wykonania. Kodów takich jest mnóstwo i nie byłoby sensu opisywać wszystkich. Oto kilka najprzydatniejszych:
 - ◆ SPI_SETDESKWALLPAPER — ustawianie tapety pulpitu. Ścieżka do pliku tapety podawana jest trzecim parametrem wywołania.
 - ◆ SPI_SETDOUBLECLICKTIME — ustawia dopuszczalny odstęp czasowy dwukrotnego kliknięcia przyciskiem myszy. Drugi parametr określa wtedy liczbę milisekund, jaka może upływać pomiędzy pierwszym a drugim naciśnięciem przyciskiem myszy, aby zostały one zaliczone jako dwukrotne kliknięcie. Spróbuj podać liczbę mniejszą od dziesięciu. Ofiara nie będzie w stanie w żaden sposób wykonać dwukrotnego kliknięcia w tak krótkim czasie. Spowodujesz więc de facto zablokowanie połowy funkcji myszy.
 - ◆ SPI_SETKEYBOARDDELAY — ustawia opóźnienie samopowtarzania klawiatury. Czas opóźnienia przekazywany jest drugim parametrem.

- ◆ SPI_SETMOUSEBUTTONSWAP — jeśli drugi parametr wywołania ma wartość 0, wywołanie ustawia standardowe funkcje przycisków myszy. Dla wszystkich innych wartości wywołanie powoduje zamianę funkcji przycisków myszy, przystosowując ją do potrzeb osób leworęcznych.
- ◆ Znaczenie drugiego parametru zależy od wartości pierwszego.
- ◆ Znaczenie trzeciego parametru zależy od wartości pierwszego.
- ◆ Czwarty parametr określa czynności do wykonania po zakończeniu czynności określonej pierwszym parametrem. Dopuszczalne są następujące wartości:
 - ◆ SPIF_UPDATEINIFILE — po wykonaniu czynności nastąpi aktualizacja profilu użytkownika.
 - ◆ SPIF_SENDCHANGE — po wykonaniu czynności nastąpi wygenerowanie komunikatu WM_SETTINGCHANGE.
 - ◆ SPIF_SENDWININICHANGE — jak poprzedni.

Jeśli funkcja zakończy pomyślnie swoje działanie, zwróci wartość różną od zera. W przypadku błędu zasygnalizuje go zerową wartością zwracaną. Oto przykład kodu, który zamienia funkcje przycisków myszy:

```
// Ustaw przyciski myszy dla osób leworęcznych:  
SystemParametersInfo(SPI_SETMOUSEBUTTONSWAP, 1, 0,  
                      SPIF_SENDWININICHANGE);  
  
// Przywróć standardowe ustawienia:  
SystemParametersInfo(SPI_SETMOUSEBUTTONSWAP, 1, 0,  
                      SPIF_SENDWININICHANGE);
```

Kod źródłowy programu oraz pliki wykonywalne z tego podrozdziału znajdują się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział2\SmallCh.



2.6. Berek z myszą

Wymarzonym obiektem świątecznych i efektownych dowcipów jest mysz. Można, na przykład, spowodować, aby wskaźnik myszy poruszał się w przypadkowy z pozoru sposób po ekranie albo aby zamarł w pewnym punkcie. Jak pokazuje doświadczenie, żarty tego rodzaju wprowadzają ofiary w osłupienie, zwłaszcza jeśli ofiarą jest użytkownik niedoświadczony, który korzysta z myszy częściej niż z klawiatury.

Szalona mysz

Jak doprowadzić myszkę do szalu? To proste:

```
for (int i = 0; i < 20; i++)  
{  
    SetCursorPos(rand() % 640, rand() % 480);  
}
```

```
    Sleep(100);  
}
```

Powyższa pętla wykonuje się dwadzieścia razy (tzn. wymusza dwudziestokrotne powtórzenie instrukcji znajdujących się pomiędzy nawiasami klamrowymi). Pierwszy wiersz pętli zmienia pozycję wskaźnika myszy. Służy do tego funkcja SetCursorPos przyjmująca dwa parametry: nową współrzędną poziomą oraz nową współrzędną pionową. Wartości tych parametrów określamy za pośrednictwem funkcji rand(), która zwraca losową liczbę całkowitą, i ograniczamy zakres wartości parametru do wartości określonej za znakiem %. W prawdziwym programie powinniśmy wartości ograniczające zakres współrzędnych wyznaczyć wywołaniami funkcji GetSystemMetrics z parametrami SM_CXSCREEN i SM_CYSCREEN. Dla uproszczenia w tym przykładzie przyjęliśmy zakres do 640 w poziomie i do 480 w pionie.

Po przesunięciu wskaźnika myszy na nową pozycję następuje krótka przerwa, tak aby użytkownik mógł zobaczyć wskaźnik na nowym miejscu. Wskaźnik „wygląda” się w ten sposób dwadzieścia razy.

Latające obiekty

W podrozdziale 2.2 mieliśmy okazję analizować przykład, w którym „programowo” klikaliśmy przycisk *Start*. Wtedy wiedzieliśmy, którego okna szukamy, więc zadanie było proste. Spróbujmy czegoś bardziej skomplikowanego — „programowego” klikania w przypadkowych miejscach pulpitu. W tym celu po ustawieniu kurSORA myszy musimy sprawdzić, nad jakim oknem się znalazł. Nie jest to trudne:

```
for (int i = 0; i < 20; i++)  
{  
    POINT pt = {rand()&800, rand()%600};  
    SetCursorPos(pt.x, pt.y);  
    Sleep(100);  
  
    HWND hPointWnd = WindowFromPoint(pt);  
    SendMessage(hPointWnd, WM_LBUTTONDOWN, MK_LBUTTON,  
                MAKELONG(pt.x, pt.y));  
    SendMessage(hPointWnd, WM_LBUTTONUP, 0,  
                MAKELONG(pt.x, pt.y));  
}
```

Tak jak w poprzednim przykładzie, generujemy dwie przypadkowe współrzędne i zapisujemy je w polach x i y struktury pt. Następnie przesuwamy wskaźnik myszy do tak wylosowanej pozycji.

Dalej sprawdzamy, jakie okno znajduje się pod wskaźnikiem myszy. W tym celu wywołujemy funkcję WindowFromPoint. Jej parametrem jest struktura typu POINT przechowująca współrzędne punktu, wokół którego szukane będzie okno. Funkcja zwraca uchwyt znalezionej okna.

Do tak wytypowanego okna przesyłamy dwa komunikaty, których znaczenie już znamy. Pierwszy z nich niesie komunikat WM_LBUTTONDOWN (naciśnięcie lewego przycisku myszy), drugi — WM_LBUTTONUP (zwolnienie lewego przycisku myszy). Po co nam

oba te komunikaty? Otóż przycisk *Start* reaguje na zdarzenie naciśnięcia przycisku myszy, ale większość programów podejmuje stosowne czynności dopiero po zwolnieniu przycisku. Dlatego powinniśmy na wszelki wypadek przesłać oba komunikaty.

Ostatnim parametrem funkcji *SendMessage* są współrzędne punktu, w którym znajduje się wskaźnik myszy. Aby przekazać parę liczb za pośrednictwem jednego parametru, tworzymy z tej pary jedną długą liczbę, do czego wykorzystujemy makrodefinicję *MAKELONG* (makrodefinicję możemy uważać za odpowiednik funkcji).

Możemy nieco zmodyfikować program:

```
for (int i = 0; i < 20; i++)
{
    // Ustaw wskaźnik na losowej pozycji
    POINT pt = {rand()&800, rand()%600};
    SetCursorPos(pt.x, pt.y);
    Sleep(100);

    // Wyślij komunikat naciśnięcia klawisza myszy
    HWND hPointWnd = WindowFromPoint(pt);
    SendMessage(hPointWnd, WM_LBUTTONDOWN, MK_LBUTTON,
                MAKELONG(pt.x, pt.y));

    // Zmień pozycję wskaźnika
    POINT pt1 = {rand()&800, rand()%600};
    SetCursorPos(pt1.x, pt1.y);

    SendMessage(hPointWnd, WM_MOUSEMOVE, 0,
                MAKELONG(pt1.x, pt1.y));

    // Zwolnij przycisk myszy
    SendMessage(hPointWnd, WM_LBUTTONUP, 0,
                MAKELONG(pt1.x, pt1.y));
}
```

Pomiędzy „wciśnięciem” a „zwolnieniem” (w cudzysłowach, bo odbywa się to programowo) przycisku myszy generujemy nowe współrzędne wskaźnika myszy i przesuwamy go. Innymi słowy, przed nadaniem komunikatu zwolnienia przycisku wysyłamy komunikat *WM_MOUSEMOVE* informujący o przesunięciu wskaźnika myszy. Przycisk jest więc naciskany w jednym miejscu i zwalniany w innym. Jeśli akurat pod wskaźnikiem znajduje się obiekt, który da się przesuwać, zostanie on przeciągnięty na nową pozycję. Symulujemy w ten sposób technikę „przeciągania i upuszczania”.

Mysz w klatce

Oto ciekawy przykład ograniczania zakresu ruchów wskaźnika myszy. Spójrz na następujący kod:

```
RECT r;
r.left = 10;
r.top = 10;
r.bottom = 100;
r.right = 100;
ClipCursor(&r);
```

Potrzebujemy zmiennej typu *RECT*. Jest to struktura zawierająca cztery zmienne liczbowe określające prostokąt. Mają one następujące nazwy, *left*, *top*, *bottom* oraz *right*, i reprezentują wierzchołki prostokąta: lewy górny i prawy dolny.

W następnych czterech wierszach przypisujemy do pól struktury wartości. Po wywołaniu funkcji *ClipCursor* z taką strukturą ograniczymy możliwości przesuwania wskaźnika myszy do wnętrza zadanego obszaru prostokątnego.

Spróbuj też uruchomić taki kod:

```
RECT r;
r.left = 0;
r.top = 0;
r.bottom = 1;
r.right = 1;
ClipCursor(&r);
```

Tutaj obszar prostokątny to obszar o boku jednego piksela, więc wskaźnik myszy nie ma żadnej swobody ruchu w jego wnętrzu. Można w ten sposób zablokować wskaźnik — użytkownik będzie go widział, nie będzie jednak mógł nim poruszać.

Jak zmienić kształt wskaźnika myszy?

W zestawie funkcji WinAPI dostępna jest ciekawa funkcja *SetSystemCursor*. Wymaga przekazania w wywołaniu dwóch parametrów:

- ◆ Uchwytu wskaźnika do zmiany. Można go uzyskać wywołaniem *GetCursor* zwracającym uchwyt bieżącego wskaźnika myszy.
- ◆ Uchwytu nowego wskaźnika. Dla tego parametru można określić jedną z następujących wartości:
 - ◆ *OCR_NORMAL* — zwykły wskaźnik (wartość domyślna).
 - ◆ *OCR_IBEAM* — wskaźnik zaznaczania tekstu.
 - ◆ *OCR_WAIT* — wskaźnik oczekiwania (z klepsydrą).
 - ◆ *OCR_CROSS* — wskaźnik zaznaczania grafiki (krzyżyk).
 - ◆ *OCR_UP* — strzałka w góre.
 - ◆ *OCR_SIZE* — wskaźnik zmiany rozmiaru.
 - ◆ *OCR_ICON* — ikona.
 - ◆ *OCR_SIZENWSE* albo *OCR_SIZENESW* — wskaźnik rozciągania obiektu.
 - ◆ *OCR_SIZEWE* — wskaźnik rozciągania obiektu w poziomie.
 - ◆ *OCR_SIZENS* — wskaźnik rozciągania obiektu w pionie.
 - ◆ *OCR_SIZEALL* — wskaźnik rozciągania obiektu w poziomie i pionie.
 - ◆ *OCR_SIZENO* — wskaźnik operacji niedozwolonej.
 - ◆ *OCR_APPSTARTING* — wskaźnik oczekiwania na uruchomienie aplikacji.

Oto prosty przykład zmiany bieżącego kształtu wskaźnika myszy:

```
SetSystemCursor(GetCursor(), OCR_CROSS);
```

Kod ten zmienia kursor, nadając mu kształt krzyżka (celownika) charakterystycznego dla operacji graficznych.



Kod źródłowy programu oraz pliki wykonywalne z tego podrozdziału znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział2\JokesWithMouse.

2.7. Znajdź i niszcz

Chciałbym teraz pokazać program, który będzie wyszukiwał konkretne okno i je zamknął. Będzie do tego potrzebny nowy projekt *Win32 Project* i pozycja menu, której wybór będzie uruchamiało właściwą funkcję programu.

Przejdź do funkcji WndProc obsługującej wszystkie komunikaty. Dodaj do niej (na początek kodu funkcji) zmienną h typu HWND:

```
HRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;
    HWND h;
```

oraz kod obsługi komunikatu wybrania pozycji menu:

```
case ID_MOLEMENU_ZNAJDZIZNISZCZ:
    h = FindWindow(0, "1 - Notatnik");
    if (h != 0)
        SendMessage(h, WM_DESTROY, 0, 0);
    break;
```

Powyższy kod wyszukuje w zasobach systemowych okno zatytułowane „1 — Notatnik”. Wyszukiwanie realizowane jest znaną już nam dobrze funkcją *FindWindow*. Wynik wyszukiwania zapisywany jest w zmiennej h. Jedynym kryterium wyszukiwania jest tytuł okna, klasa okna pozostaje nieokreślona. Zwiększa to złożoność programu. Większość aplikacji (choćby Microsoft Word) wstawia na belkę tytułową napisy w rozdaju „Microsoft Word — Nazwa dokumentu”, np. „Microsoft Word — Dokument1” — tytuł okna zawiera więc nazwę dokumentu, jeśli w programie jakiś dokument jest otwarty. W takim przypadku funkcja *FindWindow* nie może w sposób pewny wyszukać okna, o które nam chodzi. W naszym przykładzie funkcja może zwrócić zero, jeśli w systemie nie działa akurat Notatnik z dokumentem o nazwie „1”. Jak widać wyszukiwanie w oparciu o belkę tytułową okna nie jest pewne, ale niekiedy konieczne — gdy nie można określić klasy okna.

Dalej kod sprawdza wartość zmiennej h. Jeśli jest ona różna od zera, zawiera uchwyt znalezionej okna. Możemy wtedy wysłać do okna komunikat WM_DESTROY zamkujący okno.

Powyższy przykład zamyka wybrane okno po wybraniu odpowiedniej pozycji z menu. Można jednak jego właściwy kod przenieść do głównej pętli komunikatów:

```
// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    h = FindWindow(0, "Notatnik - 1");
    if (h != 0)
        SendMessage(h, WM_DESTROY, 0, 0);

    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Teraz program będzie wyszukiwał i niszczył okno od razu po uruchomieniu. Po przeniesieniu właściwego kodu do pętli nieskończonej obsługi komunikatów program nie będzie oczekiwał na zdarzenie wyboru polecenia menu, ale bez przerwy próbował odnaleźć okno i je zamknąć. Można w ten sposób zablokować użytkownikowi dostęp do wybranej aplikacji.



Kod źródłowy programu oraz pliki wykonywalne z tego podrozdziału znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział2\FindAndDestroy.

2.8. Pulpit

Pulpit jest oknem ze wszystkimi tego konsekwencjami. Aby uzyskać jego uchwyty, należy wywołać funkcję *GetDesktopWindow*. Spójrzmy na kod prostego żartu z wykorzystaniem pulpitu:

```
HWND h = GetDesktopWindow();
EnableWindow(h, FALSE);
```

W pierwszym wierszu pobieramy uchwyty okna pulpitu, w drugim czynimy to okno nieaktywnym. Spróbuj uruchomić taki kod, a zablokujesz pulpit Windows. Niestety, blokada nie jest zupełna — naciskając kombinację klawiszy *Ctr+Alt+Del*, użytkownik może wywołać okno menedżera zadań i system zostanie odblokowany. Jeśli jednak umieścimy ten kod w głównej pętli komunikatów, efekt blokady będzie całkiem solidny.

Nie ma sensu zmieniać pozycji okna pulpitu, choć jest to możliwe. Niebawem pokażę kolejny efektowny przykład żartu z pulpitem.

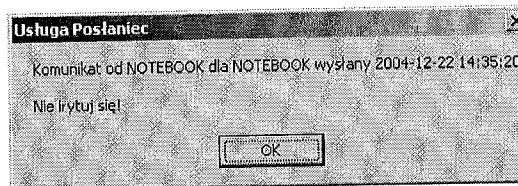


Kod źródłowy programu oraz pliki wykonywalne z tego podrozdziału znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział2\DesktopWindow.

2.9. Bomba sieciowa

W systemach z rodziny Windows NT (NT, 2000, XP i 2003) użytkownicy mają do dyspozycji polecenie NET SEND. Pozwala ono na przesyłanie z poziomu wiersza polecenia komunikatów do innych komputerów w sieci. Wystarczy wpisać polecenie, adres odbiorcy i tekst komunikatu. Po wykonaniu polecenia adresat zobaczy na ekranie okno z komunikatem. Przykład takiego okna widać na rysunku 2.13.

Rysunek 2.13.
Komunikat przesyłany poleceniem NET SEND



Składnia polecenia NET SEND jest następująca:

NET SEND adres treść

W miejsce adresu można przekazać albo nazwę NETBIOS komputera, albo jego adres IP. Aby, na przykład, przesłać komunikat „Cześć, Dany” do komputera o nazwie Dany, wystarczy wpisać:

NET SEND Dany Cześć. Dany

Co ciekawe, Windows 2000 i Windows XP nie mają zabezpieczenia przed bombardowaniem użytkownika komunikatami inicjowanymi poleceniem NET SEND. Można nadawać dowolną liczbę komunikatów i wszystkie one zostaną dostarczone do komputera adresata. Jeśli znudzi Ci się ich ręczne wpisywanie, napisz program, który zautomatyzuje wysyłanie.

Utwórz w programie Visual C++ nowy projekt typu *Win32 Project* i umieść poniższy kod tuż przed główną pętlą komunikatów:

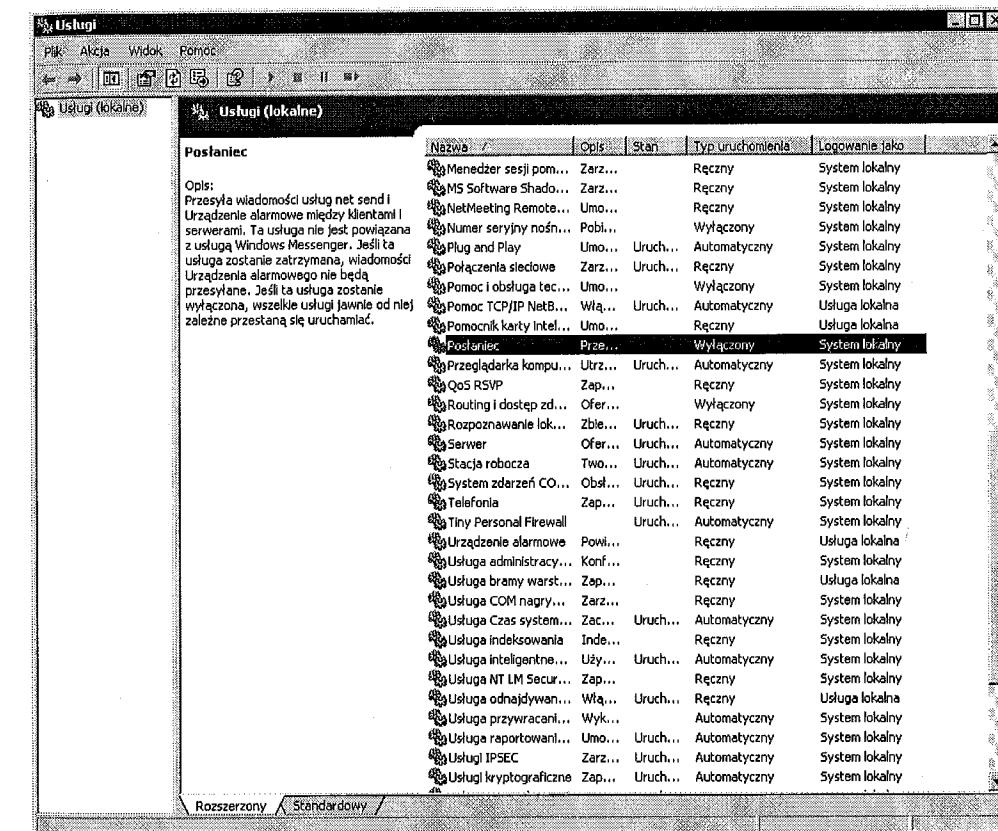
```
for (int i = 0; i < 10; i++)
{
    WinExec("NET SEND 192.168.1.121 Nie irytuj sie!", SW_SHOW);
    Sleep(1000);
}
```

Pętla ta zostanie powtórzona dziesięciokrotnie, co spowoduje dziesięciokrotne wywołanie funkcji WinExec. Funkcja ta uruchamia program określony pierwszym parametrem wywołania, interpretując go jako wiersz polecenia systemu Windows. U nas parametr ten ma postać "NET SEND 192.168.1.121 Nie irytuj się!". Wykonanie powyższego kodu spowoduje więc wysłanie do komputera o adresie 192.168.1.121 komunikatu o treści Nie irytuj się!.

W każdym przebiegu pętli funkcja Sleep wprowadza jednosekundowe opóźnienie pomiędzy kolejnymi komunikatami.

Jeśli ktoś zacznie bombardować Ciebie komunikatami NET SEND, nie próbuj zamknąć pojawiających się na ekranie okien. Zamiast tego wykonaj dwie czynności:

1. Wyjmij wtyczkę kabla łączącego Twój komputer z siecią (jeśli to niemożliwe, przejdź od razu do drugiej czynności).
2. Wybierz Start/Ustawienia/Panel Sterowania/Narzędzia administracyjne/Usługi i znajdź w wyświetlonym oknie (rysunek 2.14) wpis usługi *Postaniec* (ang. *Messenger*). Kliknij ją prawym przyciskiem myszy i wybierz z menu podręcznego polecenie *Zatrzymaj*.



Rysunek 2.14. Okno Usługi

Jeśli nie korzystasz z posłańca, powinieneś wykonać drugą z tych czynności zaraz po zainstalowaniu systemu — unikniesz uciążliwego niekiedy bombardowania.



Kod źródłowy programu oraz pliki wykonywalne z tego podrozdziału znajdują się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział 2\NetBomb.

Rozdział 3.

Programowanie w systemie Windows

W niniejszym rozdziale przyjrzymy się różnym narzędziom systemowym. Zobaczymy przykłady programów, które pomagają podglądać przebieg pracy komputera. Nie służą one tylko do zabawy. Będziemy w istocie pracować z systemem operacyjnym, choć w wielu przykładach będziemy się uciekać do żartów. Pisałem już, że każdy haker to profesjonalista — powinien więc znać tajniki systemu operacyjnego, z którego korzysta.

Zakładam tutaj, że Czytelnik korzysta z systemu Windows i pisze bądź ma zamiar pisać programy przeznaczone do uruchamiania w tym właśnie systemie. Rozdział ten ma Czytelnikowi pomóc lepiej zrozumieć system. Jak zwykle, zamiast przeciągać pamięć teorią, będziemy się uczyć przez praktykę. Czytelnicy moich poprzednich książek z pewnością pamiętają to podejście. Zawsze twierdziłem, że jedynie praktyka daje prawdziwą wiedzę. Dlatego wszystkie moje książki są wprost przeładowane przykładami. Nie inaczej będzie tym razem.

Zabierzemy się za chwilę za analizę kilku ciekawych przykładów. Będą one ilustrować techniki pracy w systemie operacyjnym Windows i uczyć praktycznego ich zastosowania. Mam nadzieję, że choć niektóre z nich przydadzą się Czytelnikowi w pracy.

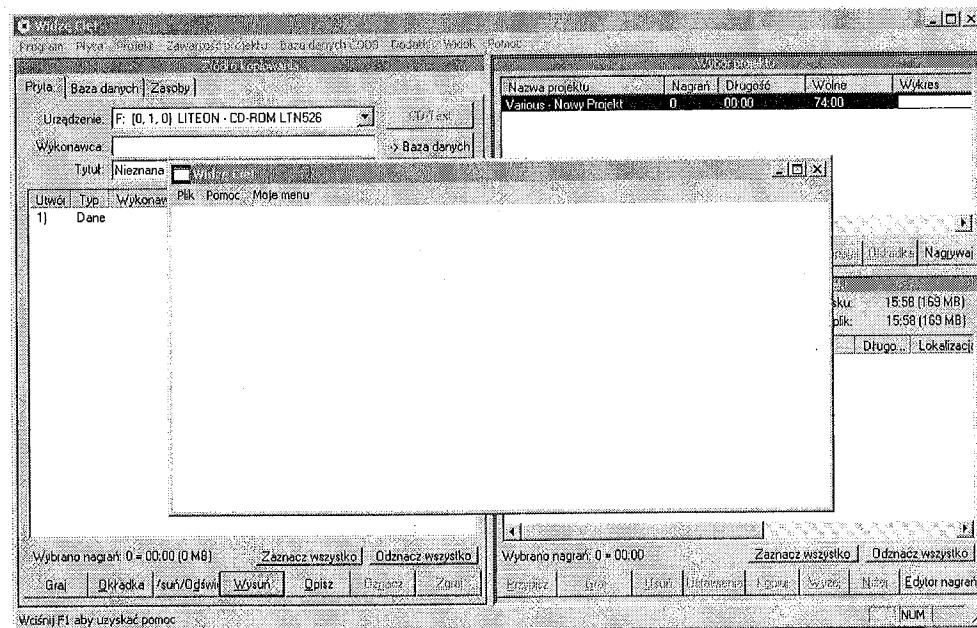
Będę starał się w tym rozdziale stopniować poziom zaawansowania przykładów, tak aby kolejne prezentacje wnosiły coś nowego i Czytelnik nie stracił zainteresowania tematem.

3.1. Manipulowanie cudzymi oknami

W mojej skrzynce poczty elektronicznej często lądują pytania: „Jak zamknąć czyjeś okno albo coś w nim zmienić?”. Zasadniczo zadanie to można łatwo zrealizować funkcją `FindWindow`, z którą zdążyliśmy się już zapoznać. Jeśli jednak zachodzi potrzeba

ingerowania w kilka (albo wszystkie) okien, należałoby skorzystać z innej niż propo-nowana wcześniej metody wyszukiwania. Napiszmy na początek program, który wy-szuka wszystkie okna na pulpicie i zmieni ich tytuły.

Rysunek 3.1 prezentuje wpływ naszego programu na okno innego programu. Jak widać, to ostatnie ma na belce tytułowej napis: „Widzę Cię!”.



Rysunek 3.1. Efekt działania programu

Utwórz w Visual C++ nowy projekt *Win32 Project* i wyposaż go w menu, za pomocą którego będziesz uruchamiał właściwą funkcję programu.

Do funkcji *WndProc* dodaj następujący kod obsługi komunikatu wywołania menu:

```
case ID_MOJEMENU_WIDZECIE:
    while (TRUE)
    {
        EnumWindows(&EnumWindowsWnd, 0);
    }
```

ID_MOJEMENU_WIDZECIE jest tu oczywiście identyfikatorem utworzonej pozycji menu.

Obsługująca ją pętla *while* jest pętlą nieskończoną wywołującą wciąż funkcję *Enum-Windows*(&*EnumWindowsWnd*, 0). To funkcja WinAPI wykorzystywana do wyliczania wszystkich otwartych okien. Jej pierwszym parametrem jest adres innej funkcji (tzw. funkcji zwrotnej), która będzie wywoływana za każdym razem, kiedy wykryte zosta-nie działające okno. Drugi parametr to liczba przekazywana do owej funkcji zwrotnej.

W roli funkcji zwrotnej występuje tu *EnumWindowsWnd*. Dla każdego okna, które znaj-dzie funkcja *EnumWindows*, wywołana zostanie funkcja *EnumWindowsWnd*. Oto jej kod:

```
BOOL CALLBACK EnumWindowsWnd(
    HWND hwnd,           // uchwyt okna-rodzica
    LPARAM lParam        // parametr własny funkcji zwrotnej
)
{
    SendMessage(hwnd, WM_SETTEXT, 0, LPARAM(LPCTSTR("Widzę Cię!")));
    return TRUE;
}
```

Liczba i typy parametrów funkcji zwrotnej oraz typ wartości zwracanej powinny być następujące:

- ◆ Pierwszy parametr — uchwyt znalezionej okna (typu *HWND*).
- ◆ Drugi parametr — wartość typu *LPARAM* wykorzystywana wedle uznania programisty funkcji zwrotnej.
- ◆ Wartość zwracana — wartość logiczna (typu *BOOL*).

Jeśli zmienisz typy bądź kolejność parametrów albo typ wartości zwracanej, funkcja stanie się niezgodna z funkcją *EnumWindows*. Aby uniknąć pomyłek, skopiowałem na-zwę funkcji i jej parametry z plików pomocy opisujących interfejs WinAPI. Wolę to od późniejszego szukania literówek w kodzie źródłowym i zalecam wszystkim takie samo postępowanie. W tym celu należy odszukać w pomocy hasło *EnumWindow* i zna-leźć w opisie odsyłacz do opisu formatu funkcji zwrotnej.

W momencie wywołania funkcji zwrotnej mamy do dyspozycji uchwyt następnego znalezionej okna. Wykorzystywaliśmy już takie uchwyty do chowania okien. Teraz spróbujemy za pośrednictwem uchwytu zmienić tytuł okna. Posłuży do tego znana nam już funkcja *SendMessage* służąca do wysyłania komunikatów systemu Windows. Oto jej parametry:

- ◆ Uchwyt okna adresata komunikatu. Otrzymaliśmy go w postaci parametru wywołania funkcji zwrotnej.
- ◆ Typ komunikatu — *WM_SETTEXT* to komunikat zmieniający tytuł okna.
- ◆ Parametr komunikatu — tutaj 0.
- ◆ Ciąg nowego tytułu okna.

Aby program kontynuował wyszukiwanie kolejnych okien, funkcja zwrotna powinna zwrócić wartość *TRUE*.



Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział3\ISeeYou.

Skomplikujmy nieco program, zmieniając na początek funkcję *EnumWindowsWnd* w sposób następujący:

```
BOOL CALLBACK EnumWindowsWnd(
    HWND hwnd,           // uchwyt okna-rodzica
    LPARAM lParam        // parametr własny funkcji zwrotnej
)
```

```

    {
        SendMessage(hwnd, WM_SETTEXT, 0, LPARAM(LPCSTR("Widzę Cię!")));
        EnumChildWindows(hwnd, &EnumChildWnd, 0);
        return TRUE;
    }
}

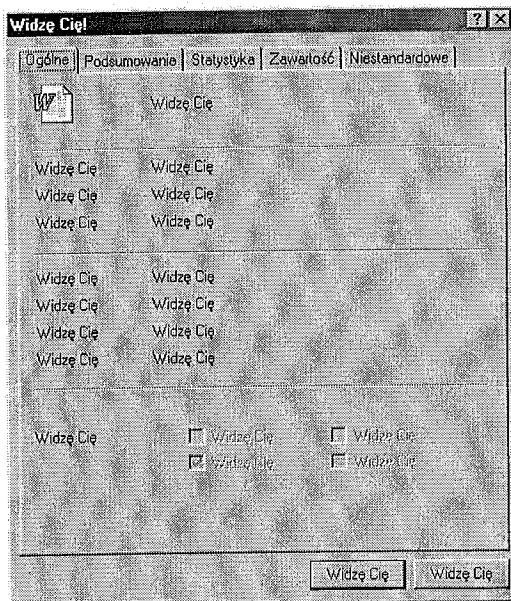
```

Tym razem po wysłaniu do znalezionej okna komunikatu zmiany tytułu wywoływana jest funkcja `EnumChildWindows`, która wyszukuje wszystkie okna potomne danego okna. Przyjmuje ona trzy parametry:

- ◆ Uchwyty okna rodzica, którego okna potomne mają być wyszukane. Podajemy tu właśnie odnalezione okno, którego uchwyty przekazany zostały w wywołaniu `EnumWindowsWnd`.
- ◆ Adres funkcji zwrotnej wywoływanej dla kolejnych okien potomnych.
- ◆ Liczba przekazywana jako własny parametr funkcji zwrotnej.

Łatwo zauważyc, że funkcja `EnumChildWnd` powinna działać podobnie jak `EnumWindowsWnd`. Ta ostatnia zmienia tytuły wszystkich okien w systemie, pierwsza zaś będzie zmieniać nazwy okien potomnych. Przykładem efektu jej działania jest rysunek 3.2.

Rysunek 3.2.
Okno
ze zmienionymi
podpisami okien
potomnych



Wiemy już, że również `EnumChildWnd` ma zmieniać podpis czy tytuł okna. Aby po zmianie można było kontynuować wyszukiwanie okien potomnych, funkcja powinna zwrócić wartość `TRUE`.

Program można uznać za kompletny, ale ma on jeszcze pewną słabość, którą trzeba wyeliminować. Przypuśćmy, że program znajdzie okno i rozpoczęcie wyliczanie jego okien potomnych, ale użytkownik nagle zamknie okno-rodzica. Program będzie próbował wysłać komunikat do znalezionej okna potomnego, które nie będzie już istniało, co doprowadzi do błędu wykonania programu. Aby tego uniknąć, należałoby za każdym razem sprawdzać poprawność otrzymanego uchwytu okna:

```

if (hwnd == 0)
    return TRUE;

```

Teraz program można uznać za gotowy.

Pamiętaj, że nie ma czegoś takiego, jak nadmierna ostrożność w programowaniu. Jeśli program ma być niezawodny, należy wziąć pod uwagę wszystkie możliwe okoliczności jego działania, które mogą doprowadzić do problemów. W niektórych przykładach ignoruję tę regułę, chcąc uniknąć komplikowania i gmatwania kodu przykładowego. Będę jednak wskazywał te miejsca w kodzie, które wymagają szczególnej rozwagi.



Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział3\ISeeYou2.

W przykładach z rozdziału 2. próbowałem zazwyczaj umieścić cały kod w głównej pętli komunikatów. Program wykonywał swoje zadanie, a potem przechodził do obsługi komunikatów systemowych. Aby zakończyć takie programy, wystarczyło zamknąć ich okna. W programie, który teraz omawiamy, występuje nieskończona pętla poza pętlą obsługi komunikatów. Oznacza to, że w czasie działania programu, kiedy wkroczy on już do naszej pętli nieskończonej, obsługa komunikatów zostanie zawieszona. Taki program bardzo trudno byłoby zamknąć. Użytkownik programu od razu zrozumie, w czym sek, ponieważ okno programu po wybraniu pozycji menu uruchamiającej zmianę tytułów przestanie odpowiadać. Będzie można pozbyć się programu jedynie przez jego zatrzymanie z poziomu menedżera zadań.

Taki efekt uboczny jest korzystny w przypadku okien niewidocznych. Jeśli kod wyświetlający okno główne zostałby usunięty z programu, również główna pętla komunikatów stałaby się zbędna i można by się jej pozbyć.

Dodajmy do naszego programu jeszcze jeden prosty, acz interesujący efekt. Spróbujmy zminimalizować wszystkie okna. Funkcja zwrotna `EnumWindowsWnd` (wywoływana dla każdego znalezionego okna) powinna wyglądać tak:

```

BOOL CALLBACK EnumWindowsWnd(
    HWND hwnd,           // uchwyty okna-rodzica
    LPARAM lParam        // parametr własny funkcji zwrotnej
)
{
    ShowWindow(hwnd, SW_MINIMIZE);
    return TRUE;
}

```

W powyższym kodzie zastosowaliśmy nową wartość drugiego parametru funkcji `ShowWindow`. Wymusza ona minimalizację okna. Uruchamiając ten program, należy zachować ostrożność. Funkcja `FindWindow` wylicza bowiem wszystkie okna, również te, które są niewidoczne.



Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział3\RandMinimize.

3.2. Gorączkowa drążczka

Skomplikujmy przykład z poprzedniego podrozdziału, pisząc program, który będzie zmieniał pozycje i rozmiary wszystkich okien tak, aby system wyglądał jak w febrze.

Stwórz nowy projekt typu *Win32 Project* w Visual C++. Dodaj do projektu pozycję menu, za pośrednictwem której będziesz wywoływać rzeczony efekt. W kodzie źródłowym projektu znajdź funkcję *WndProc*, obsługującą komunikaty kierowane do okna programu. Ruchy okien muszą być opóźniane tak, aby były dobrze widoczne. Do tego celu przyda się zmienna typu *HANDLE* inicjalizowana wywołaniem funkcji *CreateEvent*:

```
HANDLE h;
h = CreateEvent(0, TRUE, FALSE, "et");
```

Kod obsługi komunikatu wywołania polecenia menu powinien wyglądać następująco:

```
case ID_MOJEMENU_FEBRA:
    while (TRUE)
    {
        EnumWindows(&EnumWindowsWnd, 0);
        WaitForSingleObject(h, 10); // opóźnienie
    }
```

Jak w poprzednim przykładzie inicjujemy pętlę nieskończoną. Wewnątrz pętli wyliczamy wszystkie okna, opóźniając kolejny przebieg pętli za pomocą znanej już Czytelnikowi funkcji *WaitForSingleObject*.

Najciekawszy jest w tym przykładzie kod funkcji zwrotnej *EnumWindowsWnd* prezentowany na listingu 3.1.

Listing 3.1. Funkcja zwrotna *EnumWindowsWnd*

```
BOOL CALLBACK EnumWindowsWnd(
    HWND hwnd,           // uchwyt okna-rodzica
    LPARAM lParam        // parametr własny funkcji zwrotnej
)
{
    if (IsWindowVisible(hwnd) == FALSE)
        return TRUE;

    RECT rect;
    GetWindowRect(hwnd, &rect);

    int index = rand() % 2;
    if (index == 0)
    {
        rect.top = rect.top + 3;
        rect.left = rect.left + 3;
    }
    else
    {
        rect.top = rect.top - 3;
        rect.left = rect.left - 3;
    }
}
```

```
MoveWindow(hwnd, rect.left, rect.top, rect.right - rect.left,
           rect.bottom - rect.top, TRUE);
```

```
return TRUE;
```

Spójrzmy, co takiego robi funkcja *EnumWindowsWnd* wywoływana dla każdego znalezionego w systemie okna. Po pierwsze, wywołuje ona funkcję *IsWindowVisible*, która sprawdza, czy znalezione okno jest w tej chwili widoczne. Jeśli nie jest, funkcja zwrotna zwraca wartość *TRUE*, umożliwiając jej wywołanie dla kolejnego okna. Jeśli okno jest niewidoczne, nie warto go przesuwać czy rozciągać.

Następnie wywoływana jest funkcja *GetWindowRect*. Przyjmuje za pośrednictwem pierwszego parametru uchwyt okna docelowego, zwracając w drugim parametrze rozmiary okna opisane strukturą *RECT* (opisującą współrzędne prostokątnego obszaru okna polami *left*, *top*, *right* i *bottom*).

Po określeniu rozmiarów okien wyznaczamy funkcją *rand* liczbę losową z zakresu od zera do jeden. Jeśli wynik jest równy zero, wartości pól *top* i *left* struktury wymiarów okna są zwiększone o 3. Jeśli wypadnie jeden, wartości tych pól są zmniejszane o 3.

Po zmianie parametrów prostokąta przesuwamy okno funkcją *MoveWindow*. Przyjmuje ona następujące parametry:

- ◆ Uchwyt okna, którego pozycja ma zostać zmieniona (*h*).
- ◆ Nową pozycję lewej krawędzi okna (*rect.left*).
- ◆ Nową pozycję górnej krawędzi okna (*rect.top*).
- ◆ Nową szerokość okna (*rect.right - rect.left*).
- ◆ Nową wysokość okna (*rect.bottom - rect.top*).

Na koniec funkcja zwrotna zwraca *TRUE*, umożliwiając dalsze poszukiwanie okien.

Po uruchomieniu programu zobaczysz, jak wyświetlane na pulpicie okna zaczynają drżeć. Program zmienia losowo ich pozycje. Sam sprawdź. Efekt jest... wstrząsający.



Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział3\Vibration.

3.3. Przełączanie ekranów

Pamiętam, kiedy na rynku pojawiła się pierwsza wersja programu Dashboard (dla Windows 3.1). Zainteresowałem się przełączaniem ekranów i próbowałem znaleźć funkcję WinAPI, która przyjmowałaby poprzez parametr pożądany ekran. Nie udało się, nie było takiej funkcji.

Później odkryłem, że ta funkcja została „zapożyczona” z Linuksa, w którym jądro systemu implementuje konsole (ekrany) wirtualne. Taka implementacja była jednak skomplikowana. Zdołałem napisać jednak własne narzędzie do przełączania ekranów w systemach Windows 9x. Spróbuję pokazać, jak wykorzystać zastosowane w nim techniki w prostym programie-żarciku.

Jak działa przełączanie pomiędzy ekranami? Zdradzę Ci sekret — w rzeczywistości nie odbywa się żadne przełączanie. Wszystkie widoczne okna są po prostu usuwane z pulpitu, tak aby nie było ich widać. Użytkownik otrzymuje pusty pulpit. Kiedy zechce powrócić do pierwotnego ekranu, wszystko jest po prostu układane z powrotem na swoim miejscu. Jak widać, czasem najprostsze pomysły są najlepsze.

Przy przełączaniu ekranów trzeba momentalnie usunąć okna poza ekran. Będziemy to jednak robić powoli, tak aby dało się zaobserwować sposób działania programu. Będzie to wyglądać tak, jakby okna „uciekały”. Sam program będzie niewidoczny, a jedynym sposobem jego przerwania będzie jego zakończenie z poziomu okna menedżera zadań. Tu ciekawostka: jeśli nie zamknijemy programu w ciągu kilku sekund, to również okno menedżera zadań ucieknie z ekranu i trzeba będzie zaczynać zamazywanie od początku.

Nie będziemy jednak do przesuwania okien wykorzystywać dotychczasowych funkcji. Funkcje, które ustawiają pozycję okna, nie sprawdzą się w tym zadaniu, ponieważ przesuwają i odrysowują one każde okno z osobna, co zajmuje sporo czasu procesora. Jeśli na pulpicie będzie 20 okien, ich przeniesienie funkcją SetWindowPos będzie trwało zdecydowanie za dugo.

Aby szybko zaimplementować symulację przełączania ekranów, powinniśmy skorzystać ze specjalnych funkcji, które przesuwają całą grupę okien jednocześnie. Spójrzmy na przykład wykorzystujący te funkcje.

Utwórz w Visual C++ nowy projekt *Win32 Project* i przejdź do funkcji _tWinMain. Skorzystaj z listingu 3.2, aby uzupełnić ją (przed główną pętlą komunikatów) o kod przeprowadzający okna.

Listing 3.2. Kod przemieszczający okna

```
HANDLE h = CreateEvent(0, TRUE, FALSE, "et");

// Nieskończona pętla:
while (TRUE)
{
    int windowCount;
    int index;
    HWND winlist[10000];
    HWND w;
    RECT WRct;

    for (int i = 0; i < GetSystemMetrics(SM_CXSCREEN); i++)
    {
        // Zlicz okna:
        windowCount = 0;
        w = GetWindow(GetDesktopWindow(), GW_CHILD);

```

```
        while (w != 0)
        {
            if (IsWindowVisible(w))
            {
                winlist[windowCount] = w;
                windowCount++;
            }
            w = GetWindow(w, GW_HWNDNEXT); // Szukaj okna
        }
        HDWP MWStruct = BeginDeferWindowPos(windowCount);

        for (int index = 0; index < windowCount; index++)
        {
            GetWindowRect(winlist[index], &WRct);
            MWStruct = DeferWindowPos(MWStruct, winlist[index], HWND_BOTTOM,
                                      WRct.left - 1, WRct.top,
                                      WRct.right - WRct.left,
                                      WRct.bottom - WRct.top,
                                      SWP_NOACTIVATE || SWP_NOZORDER);
        }

        EndDeferWindowPos(MWStruct); // Właściwe przenosiny
    }
    WaitForSingleObject(h, 3000); // Trzysekundowe opóźnienie
}
```

Na początku tego kodu tworzymy zdarzenie puste, które później wykorzystamy w implementacji opóźnienia.

Następnie uruchamiamy pętlę nieskończoną (`while (TRUE)`). Kod wewnętrznej pętli składa się z trzech części: pozyskiwania uchwytów widocznych okien, zbiorowego przesunięcia okien do nowej pozycji i opóźnienia. Opóźnienie wprowadzaliśmy już wielokrotnie, więc nie powinieneś mieć kłopotów z jego zrozumieniem.

Wyszukiwanie widocznych okien realizowane jest następująco:

```
// Zlicz okna:
windowCount = 0;
w = GetWindow(GetDesktopWindow(), GW_CHILD);
while (w != 0)
{
    if (IsWindowVisible(w))
    {
        winlist[windowCount] = w;
        windowCount++;
    }
    w = GetWindow(w, GW_HWNDNEXT); // Szukaj okna
}
```

W pierwszym wierszu tego kodu pozyskujemy uchwyty pierwszego okna z pulpitu i zapisujemy go w zmiennej `w`. Następnie uruchamiamy pętlę, w której pozyskujemy kolejne uchwyty okien aż do momentu pozyskania uchwytu zerowego.

Wewnątrz pętli sprawdzamy widoczność okna, korzystając z funkcji IsWindowVisible wywoływanej z parametrem *w*. Jeśli okno jest niewidoczne albo zminimalizowane (funkcja IsWindowVisible zwraca wtedy FALSE), nie trzeba go przesuwać. W przeciwnym przypadku dodajemy bieżący uchwyt do tablicy uchwytów okien do przesunięcia *winlist* i zwiększamy licznik okien *windowCount*.

Funkcja GetWindow zwraca uchwyty wszystkich widocznych okien, nie rozróżniając okien nadzędnych i okien potomnych. Uchwyt znalezionego okna jest zachowywany w zmiennej *w*.

W tym przykładzie uchwyty okien są przechowywane w tablicy o z góry określonym rozmiarze (*winlist[10000]*). Ustaliłem ten rozmiar na 10 000 elementów, co powinno wystarczyć do przechowywania uchwytów okien wszystkich działających aplikacji. W rzeczy samej chyba nikt nie uruchomi naraz więcej niż 100 programów.

Najlepiej byłoby zaprząć do przechowywania uchwytów tablice dynamiczne (takie, których rozmiar da się zmieniać w czasie działania programu wedle potrzeb). Zdecydowałem jednak o wykorzystaniu tablicy statycznej, żeby nie komplikować programu. Moim celem było pokazanie ciekawego algorytmu i efektu — możesz samodzielnie ulepszyć program.

Po wykonaniu tego kodu tablica *winlist* będzie wypełniona uchwytemi wszystkich uruchomionych i widocznych okien, a zmienna *windowCount* zawierać będzie liczbę tych uchwytów. Weźmy się teraz za zbiorowe przenoszenie okien. Proces rozpoczyna się wywołaniem funkcji WinAPI BeginDeferWindowPos. Funkcja ta przydziela pamięć dla nowego okna pulpitu, do którego przeniesione zostaną wszystkie widoczne okna. Liczba okien do przeniesienia zadawana jest parametrem wywołania.

Aby przenieść okna do przydzielonej pamięci, należy wywołać funkcję DeferWindowPos. Nie przenosi ona tak naprawdę okien, a jedynie zmienia przypisane do nich informacje o pozycjach i rozmiarach okien. Funkcja ta przyjmuje następujące parametry:

- ◆ Wynik działania funkcji BeginDeferWindowPos.
- ◆ Uchwyt przenoszonego okna, czyli następny element tablicy *winlist*.
- ◆ Liczbę porządkową informującą o pozycji, którą powinno zająć dane okno względem pozostałych.
- ◆ Cztery parametry określające współrzędne okna (zmniejszamy tu współrzędną poziomą o 10) i jego rozmiary (szerokość i wysokość). Zostały one wcześniej pozyskane wywołaniem funkcji GetWindowPos.
- ◆ Znaczniki sterujące aktywnością i pozycją okna względem innych okien.

Po przeniesieniu wszystkich okien wywołujemy funkcję EndDeferWindowPos. W tym momencie wszystkie okna „przeskakują” do nowych pozycji. Odbywa się to błyskawicznie. Gdybyśmy do przesuwania wykorzystali w pętli instrukcję SetWindowPos, odrysowywanie i przesuwanie kolejnych okien trwałoby znacznie dłużej.

Dobrą praktyką jest inicjalizowanie i zwalnianie wszystkich zmiennych wymagających znaczących ilości pamięci (np. obiektów i tablic). Inicjalizacja oznacza przydział pamięci,

a zwolnienie — jej zwrócenie do dyspozycji systemu. Jeśli nie zwolnimy zajmowanych zasobów, komputer być może później odczuje ich niedostatek, co może spowolić jego działanie albo nawet wymusić przeładowanie systemu.

W tym przykładzie utworzyliśmy obiekt, ale go nie zwolniliśmy, a to dlatego, że program jest przeznaczony do działania w pętli nieskończonej, którą można przerwać jedynie na dwa sposoby:

- ◆ Odłączeniem zasilania komputera — w takim przypadku żadna z aplikacji nie zdoła zwolnić pamięci, bo cały system przestanie nagle działać.
- ◆ Zakończeniem procesu programu — jeśli nawet użytkownik będzie na tyle sprytny, żeby to zrobić, program zostanie zatrzymany w trybie natychmiastowym, więc i tak nie udałoby mu się zwolnić pamięci, nawet, gdyby był wyposażony w stosowny kod — system bezwzględnie przerwie działanie programu.

Okazuje się więc, że zwalnianie obiektu jest tu bezcelowe. Nie znaczy to, że można darować sobie zwalnianie obiektów w pozostałych programach. Jeden dodatkowy wiersz kodu nikomu nie zaszkodzi, a pozwoli na zachowanie stabilności i efektywności systemu.

Jeśli uruchomisz program, wszystkie otwarte okna zaczyną uciekać na lewo. Spróbuj choćby wywołać menu podrzenne pulpitu (prawym przyciskiem myszy) — nawet ono po chwili ucieknie. W ten sposób z ekranu zniknie każdy uruchomiony program.

Bardzo polubiłem ten program. Bawiłem się nim przeszło pół godziny. Zaciekałem mniej na tyle, że nie potrafiłem sobie odmówić takiego marnotrawstwa czasu. Szczególnie spodobał mi się przynimus szybkiego przerywania programu — to naprawdę nie jest proste. Na początku ustawiłem opóźnienie na 5 sekund, potem na cztery. Ćwiczyłem intensywnie naciskanie kombinacji *Ctrl+Alt+Del*, wyszukiwanie programu na liście procesów i naciskanie przycisku *Zakończ zadanie*. Trudność polega na tym, że okno z listą procesów również przesuwa się po ekranie. Jeśli nie zdążysz na czas wykonać wszystkich czynności, będziesz musiał powtarzać próbę.

W podobny do pokazanego sposób implementowanych jest większość aplikacji przełączających pulpty. W każdym razie ja nie znam innej metody i nie znalazłem żadnych innych przydatnych w takim zadaniu funkcji systemowych.



Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział3\DesktopSwitch.

3.4. Niestandardowe okna

W zamierzchłych czasach, w połowie lat dziesiątych, wszystkie okna były prostokątne i nikomu to nie przeszkadzało. W ciągu ostatnich kilku lat modne stały się jednak okna o kształtach nieregularnych. Każdy szanujący się programista czuje się więc w obowiązku stworzyć program z takimi oknami, aby bez wstydu i skutecznie konkurować z oknami innych programistów.

Osobiście jestem przeciwny udziwnieniom interfejsu i okna o nieregularnych kształtach, wykorzystuję jedynie sporadycznie. Pisalem o tym wcześniej i będę się powtarzał, ponieważ temat jest istotny dla całego rynku oprogramowania komercyjnego. Programista musi jednak niekiedy utworzyć okno o nieregularnych kształtach. Poza tym projektowanie takich okien jest dobrym ćwiczeniem wyobraźni, a niniejsza książka ma na celu między innymi pobudzenie kreatywności Czytelników. Dlatego zajmiemy się teraz kwestią kształtów okien.

Na początek stwórzmy okno ovalne. Przyda się do tego nowy projekt *Win32 Project* w Visual C++. Zmień jego funkcję *InitInstance* tak jak na listingu 3.3. Kod, który należy dodać do funkcji, jest na listingu oznaczony komentarzami.

Listing 3.3. Tworzenie ovalnego okna

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    hInst = hInstance; // Store instance handle in our global variable
    hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

    if (!hWnd)
    {
        return FALSE;
    }

    // Początek kodu do dodania
    Hrgn FormRgn;
    RECT WRct;
    GetWindowRect(hWnd, &WRct);
    FormRgn = CreateEllipticRgn(0, 0, WRct.right - WRct.left,
        WRct.bottom - WRct.top);
    SetWindowRgn(hWnd, FormRgn, TRUE);
    // Koniec dodanego kodu

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return TRUE;
}
```

W dodanym fragmencie kodu po pierwsze deklarujemy dwie zmienne:

- ◆ *FormRgn* typu *Hrgn* służącą do przechowywania tzw. regionów opisujących wygląd okna.
- ◆ *WRct* typu *RECT* służącą do przechowywania rozmiaru i pozycji okna. Określają one obszar, wewnątrz którego zamknięty będzie oval okna.

Dalej wywoływana jest znana już nam dobrze funkcja *GetWindowRect* wypełniająca zmienną *WRct* wymiarami i pozycją okna programu. Jesteśmy już gotowi do skonstruowania ovalnego okna. Będą nam do tego potrzebne dwie funkcje: *CreateEllipticRgn* i *SetWindowRgn*. Przyjrzyjmy się im bliżej:

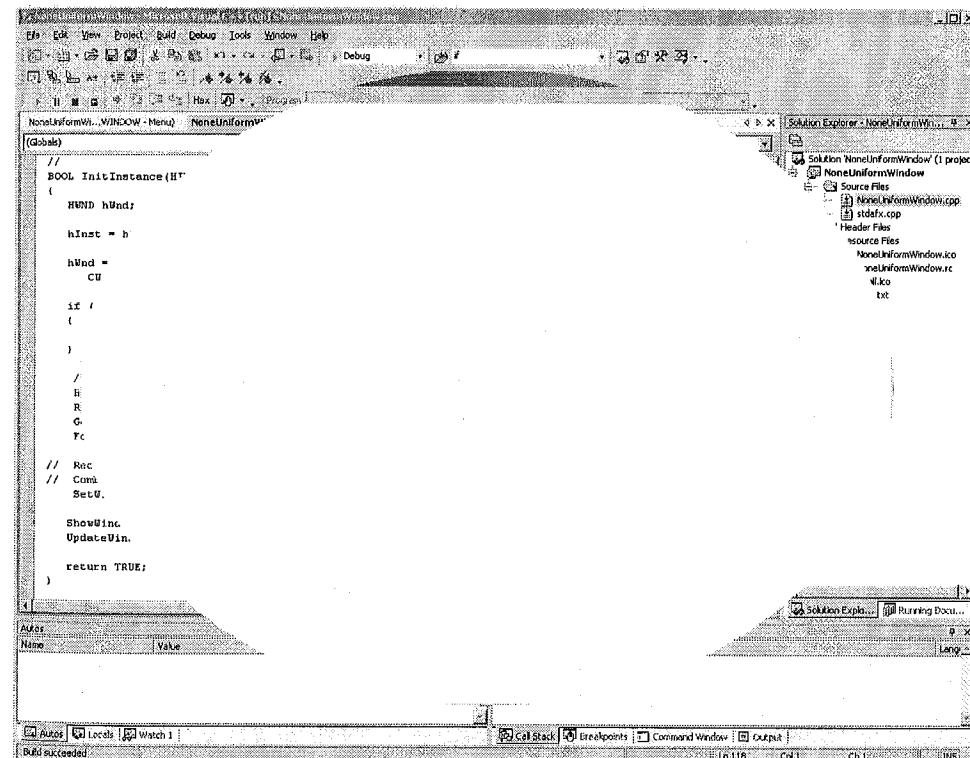
```
HRGN CreateEllipticRgn(
    int nLeftRect,           // współrzędna x górnego lewego narożnika obszaru elipsy
    int nTopRect,            // współrzędna y górnego lewego narożnika obszaru elipsy
    int nRightRect,          // współrzędna x dolnego prawego narożnika obszaru elipsy
    int nBottomRect          // współrzędna y dolnego prawego narożnika obszaru elipsy
);
```

Funkcja ta tworzy ovalny (eliptyczny) region okna. Robi to na podstawie zadanych w wywołaniu wymiarów prostokąta ograniczającego oval.

```
int SetWindowRgn(
    HWND hWnd,               // uchwyt okna
    Hrgn hRgn,              // uchwyt regionu
    BOOL bRedraw             // znacznik odrysowania okna po zmianie regionu
);
```

Ta funkcja przypisuje do okna określonego pierwszym parametrem wskazany drugim parametrem region. Jeśli trzeci parametr ma wartość *TRUE*, okno zostanie po zmianie regionu odrysowane. W przeciwnym razie trzeba będzie odrysowanie wymusić samodzielnie. W powyższym kodzie po ustaleniu regionu wywoływana jest funkcja *UpdateWindow*. Okno zostanie i tak odrysowane — trzeci parametr wywołania *SetWindowRgn* mógłby więc mieć równie dobrze wartość *FALSE*.

Uruchom program, a zobacysz na ekranie okno ovalne, jak na rysunku 3.3.



Rysunek 3.3. Ovalne okno programu

Pójdzmy nieco dalej i utwórzmy ovalne okno z prostokątną „dziurą” we wnętrzu elipsy. Zmień kod następująco:

```
HRGN FormRgn, RectRgn;
RECT WRct;
GetWindowRect(hWnd, &WRct);
FormRgn = CreateEllipticRgn(0, 0, WRct.right - WRct.left,
                           WRct.bottom - WRct.top);

RectRgn = CreateRectRgn(100, 100, WRct.right - WRct.left - 100,
                       WRct.bottom - WRct.top - 100);
CombineRgn(FormRgn, FormRgn, RectRgn, RGN_DIFF);
SetWindowRgn(hWnd, FormRgn, TRUE);
```

Mamy tu deklaracje dwóch zmiennych typu Hrgn. Pierwsza z nich, FormRgn, będzie przechowywać region ovalny, utworzony funkcją CreateEllipticRgn. Druga ma przechowywać region prostokątny utworzony funkcją CreateRectRgn. Tak jak przy tworzeniu regionu ovalnego funkcja ta wymaga określenia współrzędnych i rozmiarów prostokąta. Wynik działania funkcji zapisywany jest w zmiennej RectRgn.

Po utworzeniu obu regionów składamy je funkcją CombineRgn:

```
int CombineRgn(
    Hrgn hrgnDest,           // uchwyt regionu wynikowego
    Hrgn hrgnSrc1,          // uchwyt pierwszego regionu źródłowego
    Hrgn hrgnSrc2,          // uchwyt drugiego regionu źródłowego
    int fnCombineMode        // tryb składania
);
```

Funkcja ta składa dwa regiony źródłowe (przekazane parametrami hrgnSrc1 i hrgnSrc2) i zapisuje wynik złożenia w regionie hrgnDest.

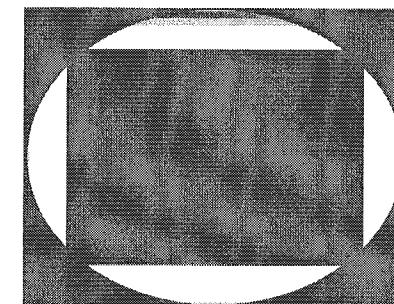
Tryb składania można określić, nadając czwartemu parametrowi wywołania funkcji (fnCombineMode) jedną z następujących wartości:

- ◆ RND_AND — region wynikowy będzie iloczynem regionów źródłowych.
- ◆ RND_COPY — region wynikowy będzie kopią regionu pierwszego.
- ◆ RND_DIFF — region wynikowy będzie zawierał te obszary, które są w regionie pierwszym, z wyjątkiem tych, które określa region drugi.
- ◆ RGN_OR — region wynikowy będzie sumą regionów źródłowych.
- ◆ RGN_XOR — region wynikowy będzie sumą wyłączającą regionów źródłowych.

Wynik działania programu widać na rysunku 3.4. Celowo w tle okna programu umieszcilem jednolity barwnie podkład, żeby można było na jego tle zobaczyć właściwy kształt okna.

Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział3\NoneUniformWindow.

Rysunek 3.4.
Owalne okno
z prostokątną
„dziurą”

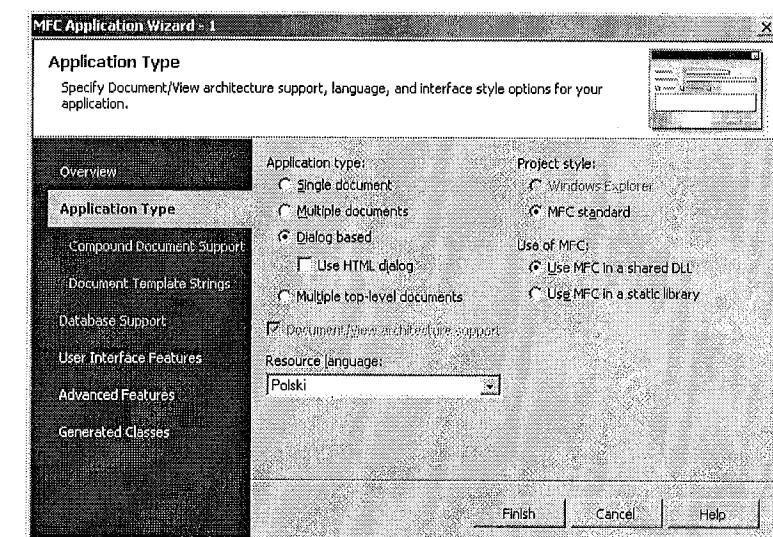


Możesz zmieniać nie tylko kształty okien, ale również kształty niektórych ich elementów sterujących. Zobaczmy to na przykładzie.

Utwórz projekt typu *MFC Application*. Nie potrzebujemy tym razem zwartości programu, możemy więc uprosić sobie programowanie, korzystając z biblioteki MFC.

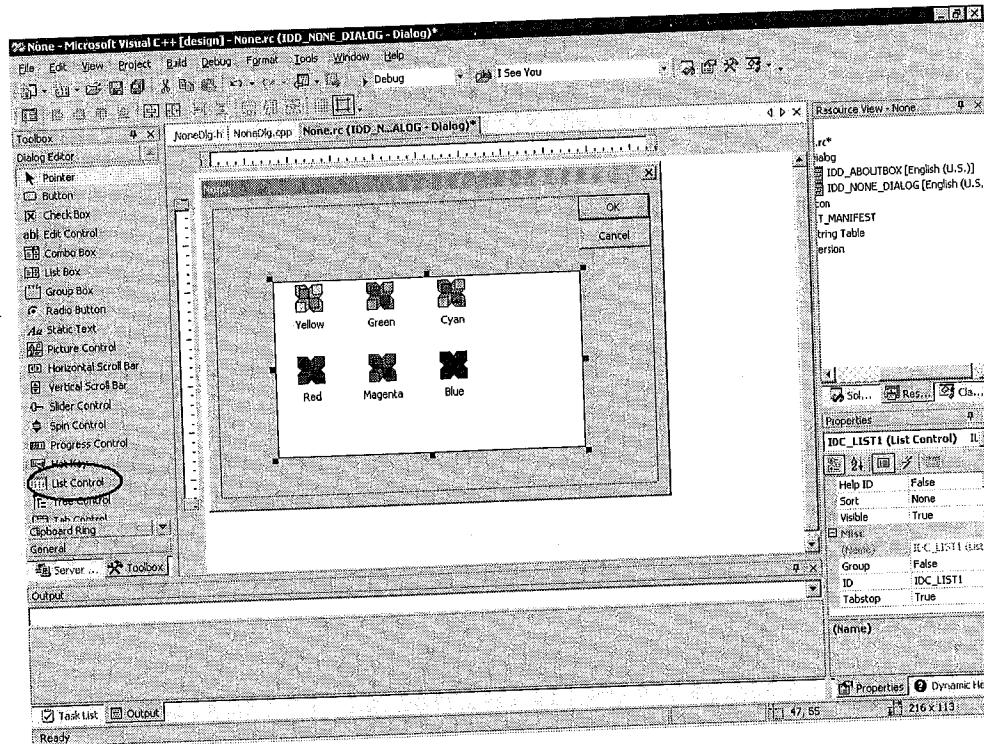
W kreatorze aplikacji przejdź na zakładkę *Application Type* i zaznacz pozycję *Dialog based* (patrz rysunek 3.5). Pozostałe parametry zostaw bez zmian. Ja swój projekt nazywałem *None*.

Rysunek 3.5.
Wybór typu aplikacji
w oknie kreatora
aplikacji MFC



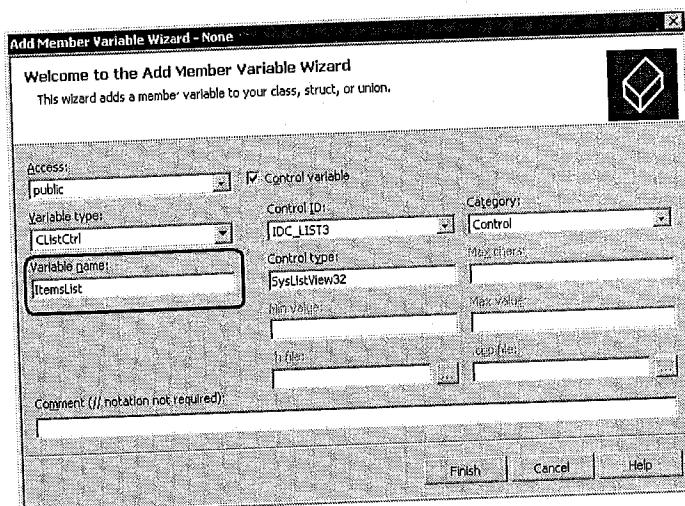
Otwórz przeglądarkę zasobów i kliknij dwukrotnie pozycję *IDD_NONE_DIALOG* w gałęzi *Dialog*. Umieść na formularzu programu jeden komponent *ListControl* (jak na rysunku 3.6).

Aby móc obsługiwać nowy element sterujący, kliknij go prawym przyciskiem myszy i wybierz z menu podrzędnego polecenie *Add Variable...*. W oknie, które się pojawi, wpisz w polu *Variable Name* nazwę zmiennej. Nazwijmy ją *ItemList* (patrz rysunek 3.7). Możesz już kliknąć przycisk *Finish* kończący definiowanie zmiennej.



Rysunek 3.6. Formularz okna tworzonego programu

Rysunek 3.7.
Okno definiowania
zmiennych uchwytów
elementów sterujących



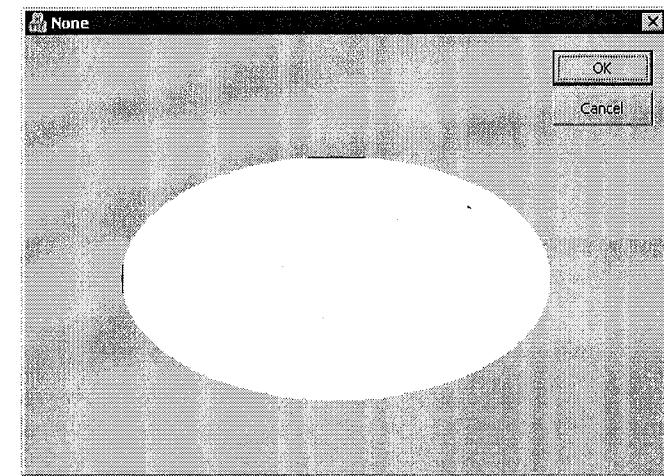
Otwórz teraz plik kodu źródłowego *NoneDlg.cpp* i znajdź w nim funkcję *CNoneDlg::OnInitDialog*. Dodaj do niej poniższy kod, umieszczając go na końcu funkcji za komentarzem // TODO: Add extra initialization here:

```
// TODO: Add extra initialization here
RECT WRct;
HRGN FormRgn;
```

```
::GetWindowRect(ItemsList, &WRct);
FormRgn = CreateEllipticalRgn(0, 0, WRct.right - WRct.left, WRct.bottom -
WRct.top);
::SetWindowRgn(ItemsList, FormRgn, TRUE);
```

Powyższy kod powinien być Ci znajomy; zmienna *ItemsList* występuje tu w roli uchwytu okna. Funkcje *GetWindowRect* i *SetWindowRgn* są poprzedzane znakami `::` wskazującemi, że funkcje te są wywoływanie z biblioteki WinAPI, a nie z MFC. Efekt działania programu widać na rysunku 3.8.

Rysunek 3.8.
Efekt działania
programu *None*



Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział3\None.

3.5. Finezyjne kształty okien

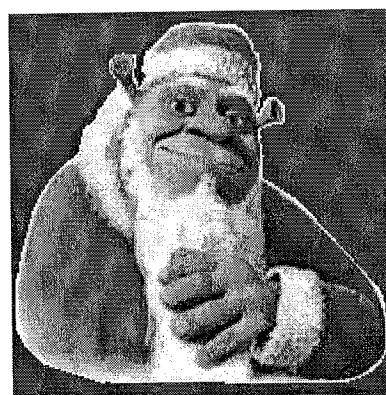
Wiemy już, jak tworzyć okna o prostych kształtach geometrycznych (owali i prostokątów) i ich kombinacji. Pora na tworzenie okien o dosłownie dowolnych kształtach. Jest to — to zrozumiałe — zadanie znacznie bardziej skomplikowane niż kombinacja dwóch prostych figur geometrycznych.

Na rysunku 3.9 możesz zobaczyć obrazek z czerwonym tłem. Spróbujemy utworzyć okno, które będzie zawierać ten obrazek i mieć przezroczyste tło (a nie czerwone, jak na obrazku), tak by kształt okna miał formę obrazka. Efekt taki byłoby niezwykle trudno uzyskać, gdybyśmy chcieli kombinować regiony w sposób przypadkowy.

WinAPI pozwala co prawda na konstruowanie wielokątnych regionów, ale ich zastosowanie bynajmniej nie ułatwia zadania.

Cóż, spróbujmy utworzyć region o kształcie obrazka. Zaprezentuję tu sposób na tyle uniwersalny, że można go stosować z dowolnymi obrazkami. Jest on przy tym prosty.

Rysunek 3.9.
Maska kształtu okna



Najpierw zastanówmy się, co to jest obrazek. To po prostu dwuwymiarowa tablica pikseli. Możemy każdy z wierszy tej tablicy potraktować jako osobny region. Innymi słowy, dla każdego wiersza pikseli obrazka utworzymy jeden region, a potem połączymy wszystkie regiony w jeden, wyznaczający kształt okna. Algorytm ten można rozpisać następująco:

1. Przejrzyj bieżący wiersz obrazka i odszukaj w nim pierwszy piksel niebędący piksemem tła. Zapamiętaj współzędną początku prostokątnego regionu w zmiennej X_1 .
2. Przejrzyj resztę wiersza obrazka w poszukiwaniu przeciwległej granicy tła. Pozycję ostatniego nieprzezroczystego piksela zapamiętaj jako X_2 . Jeśli do końca wiersza nie znajdziesz pikseli tła, rozciagnij region do końca wiersza.
3. Ustaw współzędną Y_1 na numer wiersza, a Y_2 na Y_1+1 (wysokość regionu prostokątnego obejmującego pojedynczy wiersz obrazka to jeden piksel).
4. Skonstruj region na bazie otrzymanych współzędnnych.
5. Przejdź do następnego wiersza i powtórz kroki od 1. do 4.
6. Połącz otrzymane regiony i skojarz je z oknem.

To algorytm uproszczony, ponieważ niekiedy jeden wiersz wymagać będzie więcej niż jednego regionu, jeśli właściwy obrazek będzie w danym wierszu przerwany tłem.

Powyższy algorytm, zaimplementowany w języku C++, prezentowany jest na listingu 3.4. Przeanalizujemy go nieco później.

Na razie chciałbym skupić się na obrazku. Może być nim dowolna bitmapa systemu Windows. Rozmiar pliku obrazka zależy od rozmiaru obrazka. W naszym przykładzie obrazek ma 200 na 200 pikseli i takie rozmiary zostały ustawione w kodzie. Możesz jednak spróbować uniezależnić kod od rozmiaru obrazka.

Zakładam, że piksel znajdujący się na pozycji $(0, 0)$ jest piksemem tła. Przygotowując obrazek, upewnij się, że wszystkie piksele tła mają ten sam kolor co piksel z narożnika. Takie założenie zwiększa elastyczność algorytmu, ponieważ nie blokuje żadnego określonego koloru jako koloru tła. Przy tym w narożniku obrazka rzadko znajduje się

istotny element obrazka i zawsze można wstawić tam jeden piksel tła. Nie zniszczy to zapewne estetyki obrazka.

Utwórz nowy projekt typu *Win32 Project* i znajdź w kodzie źródłowym funkcję *InitInstance*. Zmień funkcję tworzącą okno:

```
hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, 200, 200, NULL, NULL,
    hInstance, NULL);
```

Dwie kolejne liczby 200 odnoszą się do wymiarów okna dopasowanych do rozmiarów obrazka. Jeśli masz zamiar wykorzystać inny obrazek, powinieneś odpowiednio zmienić wartości parametrów.

Z okna usuniemy też menu, bo nie będzie nam potrzebne. W tym celu znajdź funkcję *MyRegisterClass* i wiersz, w którym ustawiane jest pole *wcex.lpszMenuName*. Przypisz do niej zero:

```
wcex.lpszMenuName = 0;
```

W sekcji zmiennych globalnych dodaj dwie nowe zmienne:

```
HBITMAP maskBitmap;
HWND hWnd;
```

Pierwsza z nich będzie przechowywać uchwyt obrazka, a druga to znana nam już dobrze zmienna uchwytu okna programu. Jej zadeklarowanie jako globalnej wymusza usunięcie lokalnej zmiennej *hWnd* z funkcji *InitInstance*.

Zmień funkcję *_tWinMain* zgodnie z listingiem 3.4. Program jest gotowy.

Zanim uruchomisz program, skompiluj go i otwórz katalog, w którym znajduje się kod źródłowy. Jeśli w czasie pracy włączony był tryb komplikacji *Debug*, zobaczysz podkatalog *Debug*. W innym przypadku znajdziesz tam podkatalog *Release*. Aby uniknąć błędu uruchomienia, powinieneś skopiować do tego katalogu plik obrazka.

Listing 3.4. Tworzenie okna o dowolnych rozmiarach na bazie obrazka-maski

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine,
    int nCmdShow)
{
    // TODO: Place code here
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_MASKWINDOW, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Perform application initialization
    if (!InitInstance (hInstance, nCmdShow))
    {
```

```

        return FALSE;
    }

    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_MASKWINDOW);

    // Dodaj poniższy kod
    // Na początek pozbaw okna belki tytułowej i menu systemowego
    int Style;
    Style = GetWindowLong(hWnd, GWL_STYLE);
    Style = Style || WS_CAPTION;
    Style = Style || WS_SYSMENU;
    SetWindowLong(hWnd, GWL_STYLE, Style);
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    // Wczytaj rysunek
    maskBitmap = (HBITMAP)LoadImage(NULL, "mask.bmp",
                                    IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE);
    if (!maskBitmap) return NULL;

    // Deklaracje niezbędnych zmiennych
    BITMAP bi;
    BYTE bpp;
    DWORD TransPixel;
    DWORD pixel;
    int startx;
    INT i, j;
    HRGN Rgn, ResRgn = CreateRectRgn(0, 0, 0, 0);

    GetObject(maskBitmap, sizeof(BITMAP), &bi);

    bpp = bi.bmBitsPixel >> 3;
    BYTE *pBits = new BYTE[bi.bmWidth * bi.bmHeight * bpp];

    // Skopiuj pamięć rysunku
    int p = GetBitmapBits(maskBitmap, bi.bmWidth * bi.bmHeight * bpp, pBits);

    // Znajdź kolor przezroczystości
    TransPixel = *(DWORD*)pBits;

    TransPixel <= 32 - bi.bmBitsPixel;

    // Pętla przeglądająca linie rysunku
    for (i = 0; i < bi.bmHeight; i++)
    {
        startx = -1;
        for (j = 0; j < bi.bmWidth; j++)
        {
            pixel = *(DWORD*)(pBits + (i * bi.bmWidth + j) * bpp)
                  << (32 - bi.bmBitsPixel);
            if (pixel != TransPixel)
            {
                if (startx < 0)
                {
                    startx = j;
                }
                else if (j == (bi.bmWidth - 1))

```

```

            {
                Rgn = CreateRectRgn(startx, i, j, i + 1);
                CombineRgn(ResRgn, ResRgn, Rgn, RGN_OR);
                startx = -1;
            }
        } else if (startx >= 0)
        {
            Rgn = CreateRectRgn(startx, i, j, i + 1);
            CombineRgn(ResRgn, ResRgn, Rgn, RGN_OR);
            startx = -1;
        }
    }
}

delete pBits;
SetWindowRgn(hWnd, ResRgn, TRUE);
InvalidateRect(hWnd, 0, false);
// Koniec dodanego kodu

// Main message loop:
while (GetMessage(&msg, NULL, 0, false))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

return (int) msg.wParam;
}

```

W pierwszej kolejności usuwamy z okna belkę tytułową i menu systemowe.

Następnie wczytujemy do pamięci programu bitmapę, korzystając z funkcji LoadImage. Obrazek jest wczytywany z pliku, więc pierwszy parametr ma wartość NULL, drugi to nazwa pliku obrazka, a ostatni — znacznik LR_LOADFROMFILE. Ponieważ określamy samą tylko nazwę pliku obrazka (bez ścieżki dostępu) program będzie szukał pliku w tym samym katalogu, z którego został uruchomiony. Dlatego właśnie przed uruchomieniem skompilowanego programu trzeba ręcznie skopiować plik obrazka do podkatalogu z plikiem wykonywalnym programu (*Debug* albo *Release*).

Program powinien sprawdzać, czy plik znajduje się w bieżącym katalogu. Jeśli maskBitmap ma wartość zero, należy uznać, że wczytanie obrazka się nie powiodło (najprawdopodobniej z powodu jego braku) i program powiniśmy zakończyć.

```
if (!maskBitmap) return NULL;
```

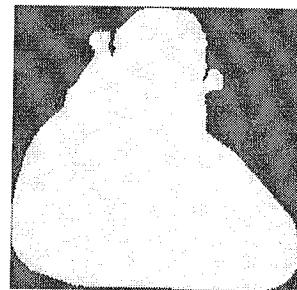
Test ten jest niezbędny, ponieważ próba odwołania się do pamięci nienależącej faktycznie do obrazka spowodowałaby natychmiastowe załamanie programu.

Dalej zaczyna się dość skomplikowany kod. Aby go zrozumieć, musisz wiedzieć, jak korzystać ze wskaźników. Nie będę jednak tego wyjaśniał — to temat na inną książkę.

Jeśli uruchomisz przykład, zobaczyś okno takie jak na rysunku 3.10. Okno przyjmuje kształt obrazka, ale będzie puste. Utworzyliśmy bowiem tylko kształt okna. Aby wypełnić okno obrazkiem, będziemy musieli uzupełnić program o następujący kod obsługujący komunikat WM_PAINT:

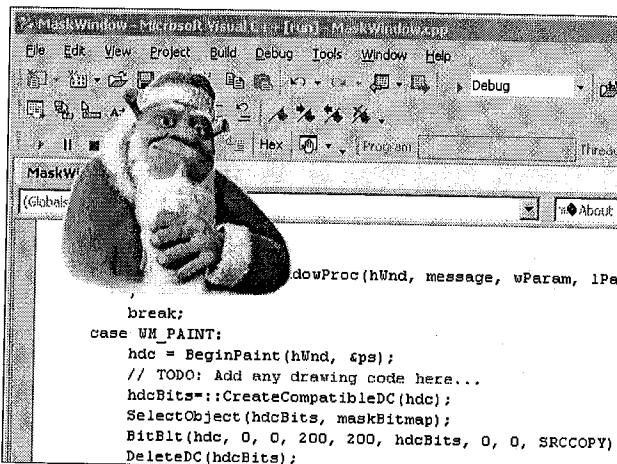
```
case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    // TODO: Any drawing code here...
    hdcBits = ::CreateCompatibleDC(hdc);
    SelectObject(hdcBits, maskBitmap);
    BitBlt(hdc, 0, 0, 200, 200, hdcBits, 0, 0, SRCCOPY);
    DeleteDC(hdcBits);
    EndPaint(hWnd, &ps);
    break;
```

Rysunek 3.10.
Pusty kształt okna



Odrysowujemy zawartość okna tak samo jak swego czasu odrysowywaliśmy w oknie obrazu przycisku Start. Efekt możesz podziwiać na swoim ekranie i rysunku 3.11.

Rysunek 3.11.
Kompletne okno
o dowolnym
kształcie



Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział3\MaskWindow.



3.6. Sposoby chwytania nietypowego okna

Korzystając z kodu z podrozdziału 3.5, możemy uzyskać okno o dowolnym niemal kształcie. Ma ono jednak pewną dotkliwą wadę — okna nie można przesuwać po ekranie, gdyż nie ma go po prostu za co chwycić myszą! Okno nie posiada belki tytułowej ani menu systemowego, za pomocą których można przesuwać zwykłe okna. Posiada za to prostokątny obszar roboczy, co dodatkowo utrudnia zadanie.

Aby pozbyć się tej niedogodności, powinniśmy „nauczyć” program, jak przesuwać okno po kliknięciu myszą w dowolnym punkcie kształtu okna. Mamy do wyboru dwa sposoby:

- ◆ Kiedy użytkownik kliknie w obszarze roboczym okna, możemy oszukać system operacyjny, „udając”, że użytkownik kliknął w obszarze (nieistniejącej) belki tytułowej. To najprostsze rozwiązanie i wymaga zaledwie jednego wiersza kodu. Jest jednak w praktyce mało wygodne, dlatego zainteresujemy się sposobem drugim.
- ◆ Możemy samodzielnie przesuwać okno. Wymaga to większej ilości kodu, ale daje rozwiązanie uniwersalne i elastyczne.

Aby zaimplementować drugie z proponowanych rozwiązań, powinniśmy oprogramować obsługę następujących zdarzeń:

- ◆ Zdarzenia naciśnięcia przycisku myszy — należy wtedy zapisać bieżącą pozycję wskaźnika myszy i sygnal zdarzenia w odpowiedniej zmiennej. W naszym przykładzie zastosujemy zmienną dragging typu bool. Dodatkowo powinniśmy przechwycić mysz tak, aby po kliknięciu okna wszystkie komunikaty o ruchu myszy były kierowane do naszego okna. Służy do tego funkcja SetCapture wymagająca przekazania uchwytu okna-odbiorcy komunikatów.
- ◆ Zdarzenia przesunięcia wskaźnika myszy — jeśli zmienna dragging ma wartość TRUE, oznacza to, że użytkownik ostatnio kliknął w obszarze okna i obecne przesunięcia powinny poruszać oknem. W tym przypadku powinniśmy aktualizować pozycję okna zgodnie z nowymi współrzędnymi wskaźnika myszy. Jeśli dragging ma wartość FALSE, nie trzeba przesuwać okna.
- ◆ Zdarzenia zwolnienia przycisku myszy — należy przypisać zmiennej dragging wartość FALSE, aby zablokować dalsze przesuwanie okna wraz z przesunięciami wskaźnika myszy.

Mogemy wykorzystać kod poprzedniego przykładu, wystarczy znaleźć w nim funkcję WndProc i dodać do niej oznaczony odpowiednimi komentarzami kod z listingu 3.5. Wcześniej w sekcji zmiennych globalnych należałoby zadeklarować dwie zmienne:

```
bool dragging = FALSE;
POINT MousePnt;
```

Listing 3.5. Kod przeciągający okno za wskaźnikiem myszy

```

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;
    HDC hdcBits;

    RECT wndrect;
    POINT point;

    switch (message)
    {
        case WM_COMMAND:
            wmId = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // Parse the menu selections
            switch (wmId)
            {
                case IDM_ABOUT:
                    DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
                    break;
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
                default:
                    return DefWindowProc(hWnd, message, wParam, lParam);
            }
            break;
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            // TODO: Any drawing code here...
            hdcBits = ::CreateCompatibleDC(hdc);
            SelectObject(hdcBits, maskBitmap);
            BitBlt(hdc, 0, 0, 200, 200, hdcBits, 0, 0, SRCCOPY);
            DeleteDC(hdcBits);
            EndPaint(hWnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        // Kod obsługi interesujących nas zdarzeń
        // Naciśnięcie lewego przycisku myszy:
        case WM_LBUTTONDOWN:
            GetCursorPos(&MousePnt);
            dragging = true;
            SetCapture(hWnd);
            break;
        // Przesuwanie wskaźnika myszy:
        case WM_MOUSEMOVE:
            if (dragging) // jeśli wcześniej naciśnięto przycisk myszy
            {
                // Pobierz bieżącą pozycję wskaźnika:
                GetCursorPos(&point);
                // Pobierz bieżący obszar okna:

```

```

GetWindowRect(hWnd, &wndrect);

// Dostosuj pozycję okna
wndrect.left = wndrect.left + (point.x - MousePnt.x);
wndrect.top = wndrect.top + (point.y - MousePnt.y);

// Ustaw nowy obszar okna:
SetWindowPos(hWnd, NULL, wndrect.left, wndrect.top, 0, 0,
              SWP_NOZORDER | SWP_NOSIZE);

// Zapisz bieżącą pozycję wskaźnika myszy w zmiennej:
MousePnt = point;
}
break;
// Zwolnienie lewego przycisku myszy:
case WM_LBUTTONUP:
    if (dragging)
    {
        dragging=false;
        ReleaseCapture();
    }
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}

```

Wszystkie funkcje wykorzystane w tym przykładzie są już Czytelnikowi znane. Program jest jednak dość rozległy, więc opatrzyłem go większą niż zwykle liczbą komentarzy, które powinny pomóc w analizie jego działania.



Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział3\MaskWindow2.

3.7. Ujawnianie haseł

W większości aplikacji wprowadzane do nich hasła są wyświetlane w postaci szeregu gwiazdek. Ma to zapobiec podejrzeniu hasła przez osoby niepowołane. Ale co, jeśli zapomniemy hasła? Jak je odczytać, jeśli w polu hasła widać tylko gwiazdki? Istnieje kilka narzędzi, które na to pozwalają. Jestem jednak daleki od odsyłania do nich Czytelnika.

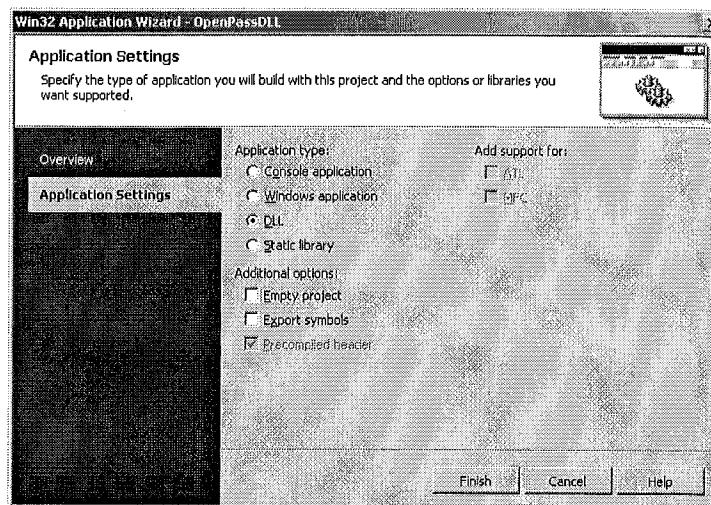
Zamiast iść na łatwiznę, sami napiszemy potrzebny program.

Program będzie się składać z dwóch plików. Pierwszy z nich — plik wykonywalny — będzie wczytywał do pamięci programu inny plik (plik biblioteki DLL). Kod z tej biblioteki zostanie zarejestrowany w systemie w roli kodu obsługi komunikatów naciśnięcia prawego przycisku myszy w konkretnym oknie. W tym momencie tekst z tegoż okna zostanie zamieniony z postaci ciągu gwiazdek na postać zwykłego ciągu znaków. Brzmi to bardzo uczenie, ale da się oprogramować w parę minut.

3.7.1. Biblioteka deszyfrowania haseł

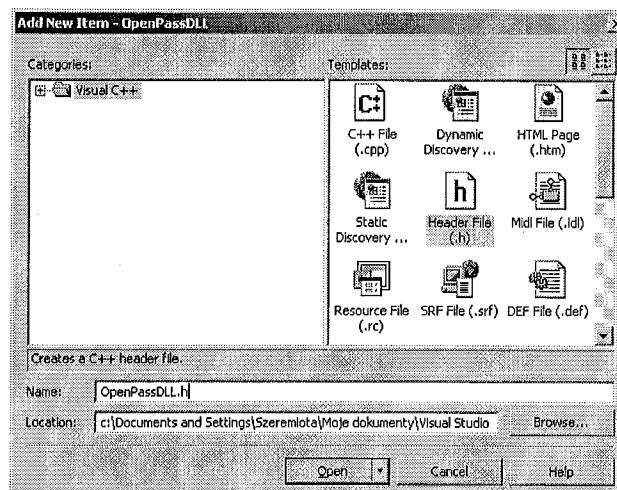
Dla potrzeb tego przykładu oprogramowałem specjalną bibliotekę DLL. Teraz zrobimy to samo wspólnie. Utwórz w Visual C++ nowy projekt *Win32 Project* i nazwij go *OpenPassDLL*. W kreatorze aplikacji wybierz *DLL* jako typ aplikacji (jak na rysunku 3.12).

Rysunek 3.12.
Ustawienia kreatora aplikacji dla biblioteki DLL



Nowy projekt będzie się składał z jednego tylko (poza standardowym plikiem *stdafx.cpp*) pliku — *OpenPassDLL.cpp* — nie będzie miał jednak żadnego pliku nagłówkowego. Pliki nagłówkowe zawierają zwykle deklaracje, a my będziemy kilku potrzebować. Musimy więc plik nagłówkowy dodać do projektu własnoręcznie. W tym celu w oknie *Solution Explorer* kliknij prawym przyciskiem myszy pozycję *Header Files*. Z menu podręcznego wybierz *Add*, a następnie *Add New Item*. Zobaczysz okno podobne do tego z rysunku 3.13. W prawej części okna zaznacz *Header File (.h)* i wpisz w polu *Name* nazwę *OpenPassDLL.h*. Kliknij przycisk *Open*, a projekt zostanie uzupełniony o nowy plik.

Rysunek 3.13.
Okno dodawania pliku projektu



Kliknij dwukrotnie nowo dodany plik. Otworzy się edytor tekstu. Wpisz następujący kod:

```
// Makrodefinicja eksportu DLL w Win32, zastępuje __export z Win16
#define DllExport extern "C" __declspec(dllexport)

// Prototyp
DllExport void RunStopHook(bool State, HINSTANCE hInstance);
```

Makrodefinicja *DllExport* pozwala funkcjom, których deklaracje są nią opatrzone, na eksportowanie — tzn. umożliwia im wywoływanie z innych aplikacji.

Drugi z wierszy kodu pliku nagłówkowego deklaruje eksportowaną funkcję. Jak widać, deklaracja przypomina implementację pozbawioną kodu funkcji — mamy tu jedynie nazwę funkcji i typu i nazwy parametrów. Właściwa definicja funkcji powinna wylądować w pliku *OpenPassDLL.cpp*.

Przejdzmy do pliku *OpenPassDLL.cpp*. Jego zawartość prezentowana jest na listingu 3.6. Skopiuj ten kod do swojego pliku i sprawdź go.

Listing 3.6. Plik *OpenPassDLL.cpp*

```
// OpenPassDLL.cpp : Defines the entry point for the DLL application.
//

#include <windows.h>
#include "stdafx.h"
#include "OpenPassDLL.h"

HHOOK SysHook;
HWND Wnd;
HINSTANCE hInst;

BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
                      )
{
    hInst = (HINSTANCE)hModule;
    return TRUE;
}

HRESULT CALLBACK SysMsgProc(
    int code,           // Kod zaczepu
    WPARAM wParam,     // Znacznik usuwania
    LPARAM lParam,     // Adres struktury komunikatu
)
{
    // Prześlij komunikat do pozostałych zaczepów w systemie
    CallNextHookEx(SysHook, code, wParam, lParam);

    // Sprawdź komunikat
    if (code == HC_ACTION)
    {
        // Pobierz uchwyty okna, które wygenerowało komunikat
        Wnd = ((tagMSG*)lParam)->hwnd;
```

```

// Sprawdzenie typu komunikatu.
// Czy użytkownik nacisnął prawy przycisk myszy?
if (((tagMSG*)lParam)->message == WM_RBUTTONDOWN)
{
    SendMessage(hWnd, EM_SETPASSWORDCHAR, 0, 0);
    InvalidateRect(hWnd, 0, true);
}

return 0;
}

///////////////////////////////
DllExport void RunStopHook(bool State, HINSTANCE hInstance)
{
    if (true)
        SysHook = SetWindowsHookEx(WH_GETMESSAGE, &SysMsgProc, hInst, 0);
    else
        UnhookWindowsHookEx(SysHook);
}

```

Przyjrzymy się bliżej kodowi biblioteki DLL. Na początku kodu włączane są pliki nagłówkowe. Jednym z nich jest plik *OpenPassDLL.cpp* zawierający makrodefinicję eksportującą funkcję naszej biblioteki.

Dalej deklarowane są trzy zmienne globalne:

- ◆ SysHook — uchwyt zaczepu komunikatów systemowych.
- ◆ Wnd — uchwyt okna (tego z gwiazdkami) klikniętego prawym przyciskiem myszy.
- ◆ hInst — uchwyt działającego egzemplarza DLL.

Deklaracje mamy z głowy. W tej części programu jako główna występuje funkcja *DllMain*. Jest to standardowa funkcja wykonywana podczas uruchomiania biblioteki DLL. Przeprawdza się w niej czynności inicjalizacyjne. W naszym przypadku nie mamy czego inicjalizować poza zachowaniem pierwszego parametru wywołania funkcji (uchwytu egzemplarza biblioteki) w zmiennej *hInst*.

Przejdzmy do funkcji *RunStopHook*. Jej zadanie polega na zakładaniu i zwalnianiu zaczepu systemowego. Przyjmuje ona dwa parametry:

- ◆ Wartość logiczną (typu *bool*) TRUE, jeśli zaczep jest zakładany, i FALSE, kiedy ma być zwolniony.
- ◆ Uchwyt egzemplarza aplikacji wywołującej tę funkcję. Nie będziemy na razie wykorzystywać tego parametru.

Jeśli pierwszym parametrem przekazywaną jest wartość TRUE, rejestrujemy zaczep, za pośrednictwem którego będziemy przechwytywać komunikaty systemu Windows. Służy do tego funkcja *SetWindowsHookEx*. Wymaga ona przekazania czterech parametrów:

- ◆ Typu zaczepu (tutaj WH_GETMESSAGE).
- ◆ Wskaźnika funkcji, która ma otrzymać przechwycony komunikat.
- ◆ Uchwytu egzemplarza aplikacji — przekazujemy zachowany wcześniej uchwyt egzemplarza biblioteki DLL.
- ◆ Identyfikatora wątku. Wartość zero obejmuje wszystkie wątki.

W roli drugiego parametru przekazujemy adres funkcji *SysMsgProc*. Jest ona deklarowana w tej samej bibliotece — jej kodem zajmiemy się później.

Wartość zwracaną przez funkcję *SetWindowsHookEx* zachowujemy w zmiennej *SysHook*. Będzie ona potrzebna do zwolnienia zaczepu.

Jeśli do funkcji *RunStopHook* przekazana zostanie wartość FALSE, zwalniamy zaczep. Podega to na wywołaniu funkcji *UnhookWindowsHookEx* i przekazaniu do niej zmiennej *SysHook*. Wartość *SysHook* uzyskaliśmy wcześniej przy zakładaniu zaczepu.

Przyjrzyjmy się teraz funkcji *SysMsgProc*, która będzie wywoływana w momencie przechwycenia komunikatu o zdarzeniu.

W pierwszym wierszu kodu funkcji przechwycony komunikat jest funkcją *CallNextHookEx* przesyłany do pozostałych zaczepów systemowych. Krok ten jest niezbędny, aby obsługa komunikatu była kompletna — komunikat interesuje nie tylko nas, ale i, być może, wykorzystywany jest w innych zaczepach.

Następnie sprawdzamy typ komunikatu. Interesują nas jedynie zdarzenia naciśnięcia przycisków myszy. Porównujemy więc kod zdarzenia (*code*) z *HC_ACTION*. Obsługę pozostałych komunikatów możemy sobie darować.

Dalej określamy okno, dla którego przeznaczony był pierwotnie komunikat, i sprawdzamy typ tego komunikatu. Uchwyt okna jest pozyskiwany instrukcją: *((tagMSG*)lParam)->hwnd*. Na pierwszy rzut okna jest ona完全nie nieczytelna. Spróbujmy ją rozszyfrować. Wyrażenie to opiera się na zmiennej typu *lParam*, którą pozyskaliśmy za pośrednictwem ostatniego parametru wywołania funkcji *SysMsgProc*. Zapis *((tagMSG*)lParam)* oznacza, że pod adresem wskazywanym przez przekazany do funkcji parametr *lParam* znajduje się struktura typu *tagMSG*. Struktura ta posiada pole *hwnd*, które przechowuje uchwyt okna, dla którego wygenerowano pierwotnie komunikat.

Dalej sprawdzamy rodzaj zdarzenia. Jeśli komunikat reprezentuje naciśnięcie prawego przycisku myszy, powinniśmy usunąć gwiazdki z okna. Warunek ten sprawdzamy przez test wartości pola *message* struktury *((tagMSG*)lParam)*.

Jeśli wartość ta jest równa *WM_RBUTTONDOWN*, oznacza to, że naciśnięty został prawy przycisk myszy, więc powinniśmy przystąpić do ujawnienia zaslonietego gwiazdkami tekstu. W tym celu należy przesłać do okna komunikat. Korzystamy z pośrednictwa funkcji *SendMessage* z następującymi parametrami:

- ◆ *Wnd* — uchwytem okna, do którego kierujemy komunikat.
- ◆ *EM_SETPASSWORDCHAR* — typem komunikatu. Wartość ta sygnalizuje konieczność zmiany znaków wykorzystywanych do ukrywania ciągu hasła.

- ◆ 0 — nowy znak wykorzystywany do usunięcia maski i przywrócenia właściwego tekstu hasła.
- ◆ 0 — parametr zarezerwowany.

Na koniec wywołujemy funkcję InvalidateRect, która wymusza odrysowanie okna określonego pierwszym parametrem wywołania. Drugi parametr określa obszar, który powinien zostać odrysowany — jeśli będzie miał wartość zero, odrysowaniem objęte zostanie całe okno. Jeśli ostatni parametr wywołania ma wartość TRUE, odrysowane zostanie również tło.



Kod źródłowy tego przykładu oraz pliki biblioteki znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział3\OpenPassDLL.

3.7.2. Deszyfrowanie hasła

Napiszmy program, który będzie wczytywał utworzoną przed chwilą bibliotekę DLL i zakładał zaczep. Utwórz nowy projekt typu *Win32 Project*, zaznaczając jako typ aplikacji *Windows application*. Do kodu pliku źródłowego w funkcji _tWinMain wprowadź zmiany zgodnie z listingiem 3.7.

Listing 3.7. Wczytywanie biblioteki DLL i zakładanie zaczepu

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine,
    int nCmdShow)
{
    // TODO: Place code here.
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_OPENPASSTEST, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_OPENPASSTEST);

    /////////////////////////////////
    // Dodaj poniższy kod:
    LONG lResult;
    HINSTANCE hModule;

    // Synonim typu dla wskaźnika funkcji:
    typedef void (RunStopHookProc)(bool, HINSTANCE);
```

```
RunStopHookProc* RunStopHook = 0;

// Wczytaj plik DLL:
hModule = ::LoadLibrary("OpenPassDLL.dll");

// Pobierz adres funkcji z biblioteki:
RunStopHook = (RunStopHookProc*)::GetProcAddress((HMODULE) hModule,
"RunStopHook");

// Wywołaj funkcję:
(*RunStopHook)(TRUE, hInstance);

// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
(*RunStopHook)(FALSE, hInstance);
FreeLibrary(hModule);

return (int) msg.wParam;
```

Ponieważ funkcja zadeklarowana jest w bibliotece DLL, a wywoływana jest w innym programie, trzeba w tym programie określić typ funkcji. Jeśli tego nie zrobimy, kompilator nie będzie w stanie zrealizować poprawnie wywołania. Typ funkcji, o której mowa, opisany jest następująco:

```
typedef void (RunStopHookProc)(bool, HINSTANCE);
```

Instrukcja ta deklaruje typ RunStopHookProc jako funkcję, która nie zwraca żadnych wartości i przyjmuje dwa parametry (pierwszy typu bool, drugi typu HINSTANCE). W następnym wierszu deklarujemy funkcję tego typu i przypisujemy do niej chwilowo zero.

Teraz powinniśmy wczytać do pamięci bibliotekę DLL. Służy do tego specjalna funkcja o nazwie LoadLibrary — przyjmuje ona za pośrednictwem parametrów nazwę pliku i ścieżkę dostępu. My przekazujemy tylko nazwę pliku biblioteki, co powoduje, że przed uruchomieniem programu powinniśmy skopiować plik biblioteki do katalogu programu, ewentualnie do jednego z katalogów bibliotek systemu Windows.

Po wczytaniu biblioteki określamy adres funkcji RunStopHook w pamięci, tak aby można było ją spod tego adresu wywołać. Wykorzystujemy do tego funkcję GetProcAddress, która wymaga przekazania wskaźnika uchwytu egzemplarza biblioteki i nazwy funkcji. Wynik wywołania zapisujemy w zmiennej RunStopHook.

Jesteśmy już gotowi do wywołania funkcji zakładającej zaczep. Wywołanie wygląda dość niecodziennie:

```
(*RunStopHook)(TRUE, hInstance);
```

Zaraz po wywołaniu startuje główna pętla komunikatów, w której nie musimy nic zmieniać. Pod koniec programu musimy jedynie zwolnić zaczep i usunąć bibliotekę DLL z pamięci. Realizują to dwa wiersze:

```
(*RunStopHook) (FALSE, hInstance);
FreeLibrary(hModule);
```

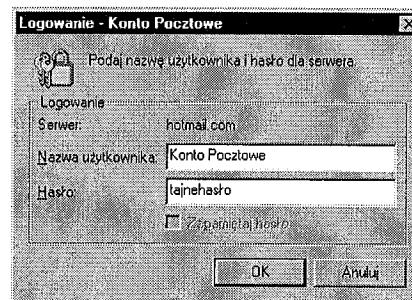


Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział3\OpenPassTest.

Aby przetestować działanie całości, powinieneś umieścić plik biblioteki DLL *OpenPasDLL.dll* w katalogu pliku wykonywalnego projektu *OpenPasTest*. Po uruchomieniu programu kliknij prawym przyciskiem myszy dowolne okno tekstowe z hasłem. Gwiazdki (albo inne znaki maskujące hasło) zamienią się w zwykłe litery.

Przykład efektów działania programu ilustruje rysunek 3.14. Widać na nim okno dialogowe logowania wyświetlane przez klienta poczty elektronicznej. Zauważ, że pole hasła, zazwyczaj maskujące treść gwiazdkami, zawiera jawną tekst hasła.

Rysunek 3.14.
Program
OpenPassTest
w działaniu



3.7.3. Obróćmy to w żart

Ten przykład może być łatwo przerobiony na żart programowy. Aby zmienić zachowanie programu, wystarczy zmienić kilka parametrów biblioteki DLL. Weźmy się więc za obsługę również kliknięcia lewym przyciskiem myszy — tym razem zamiast ujawniać, będziemy maskować znaki w polu tekstowym. Jeśli użytkownik zechce przełączyć się myszą na pole zawierające tekst, zobaczy tylko ciąg liter „d”. Zmodyfikowany kod przykładu widnieje na listingu 3.8.

Listing 3.8. Zaczep komunikatów zastępujący wszelkie litery literą „d”

```
HRESULT CALLBACK SysMsgProc(
    int code,           // Kod zaczepu
    WPARAM wParam,     // Znacznik usuwania
    LPARAM lParam      // Adres struktury komunikatu
)
{
    // Prześlij komunikat do pozostałych zaczepów systemu
    CallNextHookEx(SysHook, code, wParam, lParam);
}
```

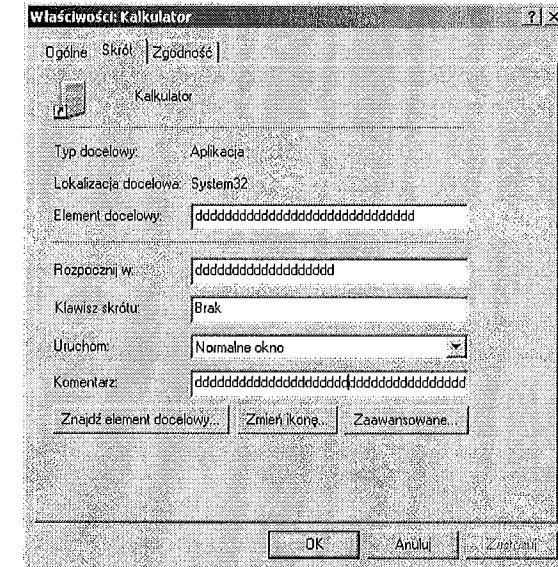
```
// Sprawdź komunikat
if (code == HC_ACTION)
{
    // Pobierz uchwyty okna, które wygenerowało komunikat
    Wnd = ((tagMSG*)lParam)->hwnd;

    // Sprawdzenie typu komunikatu.
    // Czy użytkownik naciągnął lewy przycisk myszy?
    if (((tagMSG*)lParam)->message == WM_LBUTTONDOWN)
    {
        SendMessage(Wnd, EM_SETPASSWORDCHAR, 100, 0);
        InvalidateRect(Wnd, 0, true);
    }
}

return 0;
```

Tym razem sprawdzamy, czy komunikat dotyczy zdarzenia naciśnięcia lewego przycisku myszy. Jeśli tak, to za pośrednictwem funkcji *SendMessage* wysyłamy do okna, na rzecz którego pierwotnie wygenerowano komunikat, komunikat zmiany znaku maskowania z trzecim parametrem o wartości 100 (100 to kod litery „d”). W efekcie, kiedy w czasie działania programu użytkownik kliknie jakiekolwiek pole tekstowe, jego zawartość zastąpiona zostanie ciągiem liter „d”. Przykład działania programu mamy na rysunku 3.15, na którym widać okno *Właściwości skrótu do Kalkulatora* — pola tekstowe zawierają zamaskowane ciągi znaków.

Rysunek 3.15.
„Zmiana”
właściwości skrótu



Kod źródłowy tego przykładu znajduje się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział3\SetPassDLL.

3.8. Monitorowanie plików wykonywalnych

Czasem chcielibyśmy wiedzieć, jakie programy uruchamiał użytkownik i jak długo ich używał. Takie informacje przydają się nie tylko hakerom, ale również opiekunom systemów informatycznych i kierownictwu działów.

Haker może, na przykład, oczekwać na uruchomienie konkretnego programu, aby wykonać na nim jakieś czynności. Administrator sieci może z kolei chcieć wiedzieć, co użytkownik robił, kiedy system uległ awarii. Przełożeni chętnie zaś dowiedzieliby się, czy ich pracownicy zajmują się w pracy tym, czym powinni.

Próbuje odpowiedzieć na te pytania musiałem swego czasu dowiedzieć się, jak monitorować uruchamianie i czas działania programów w systemie operacyjnym. Okazuje się, że jest to całkiem proste, przy czym program monitorujący nie różni się wiele od poprzednio omawianego programu ujawniającego zamaskowane hasła. Powiniśmy bowiem również w tym przypadku założyć zaczep, za pośrednictwem którego monitorowalibyśmy wybrane komunikaty systemowe. Poprzednio zaczep zakładaliśmy wywołaniem SetWindowsHookEx, a przechwytywanie podlegała komunikaty typu WH_GETMESSAGE. Jeśli zmienimy ten parametr na WH_CBT, przechwytywane będą następujące komunikaty:

- ◆ HCBT_ACTIVATE — sygnalizujący aktywację aplikacji.
- ◆ HCBT_CREATEWND — sygnalizujący utworzenie nowego okna.
- ◆ HCBT_DESTROYWND — sygnalizujący usunięcie jednego z istniejących okien.
- ◆ HCBT_MINMAX — sygnalizujący minimalizację albo maksymalizację jednego z istniejących okien.
- ◆ HCBT_MOVESIZE — sygnalizujący przesunięcie albo zmianę rozmiaru jednego z istniejących okien.

Kod biblioteki DLL monitorującej działające programy prezentowany jest na listingu 3.9. Na razie opowiemy sobie jedynie o rozpoznawaniu zdarzeń; ich obsługą zajmiemy się później.

Listing 3.9. Kod biblioteki monitorującej pliki wykonywalne

```
// FileMonitor.cpp : Defines the entry point for the DLL application.
//

#include <windows.h>
#include "stdafx.h"
#include "FileMonitor.h"

HHOOK SysHook;
HINSTANCE hInst;

BOOL APIENTRY DllMain( HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
                      )
```

```
{
    hInst = (HINSTANCE)hModule;
    return TRUE;
}

HRESULT CALLBACK SysMsgProc(
    int code,           // Kod zaczepu
    WPARAM wParam,     // Znacznik usuwania
    LPARAM lParam      // Adres struktury komunikatu
)
{
    // Prześlij komunikat do pozostałych zaczepów systemu
    CallNextHookEx(SysHook, code, wParam, lParam);

    if (code == HCBT_ACTIVATE)
    {
        char windtext[255];
        HWND Wnd = ((tagMSG*)lParam)->hwnd;
        GetWindowText(Wnd, windtext, 255);

        // Tu możesz zapisać tytuł aktywnego pliku
    }

    if (code == HCBT_CREATEWND)
    {
        char windtext[255];
        HWND Wnd = ((tagMSG*)lParam)->hwnd;
        GetWindowText(Wnd, windtext, 255);

        // Tu możesz zapisać nowy tytuł pliku
    }
    return 0;
}

///////////////////////////////////////////////////////////////////
```

```
DllExport void RunStopHook(bool State, HINSTANCE hInstance)
{
    if (true)
        SysHook = SetWindowsHookEx(WH_CBT, &SysMsgProc, hInst, 0);
    else
        UnhookWindowsHookEx(SysHook);
}
```

Nasz zaczep będzie wywoływany zawsze przy tworzeniu nowego okna albo aktywowaniu jednego z istniejących. W tej chwili funkcja zaczepu zawiera kod określający nazwę okna, które wygenerowało komunikat. Możesz uzupełnić ten kod własnymi czynnościami (na przykład zapisaniem daty i czasu utworzenia czy aktywowania okna). Zostawię tą kwestię otwartą, ponieważ kod tej części należy dopasować do konkretnych potrzeb.

Za pomocą tej prostej metody możemy uzyskiwać dostęp do komunikatów o zdarzeniach dotyczących okien innych programów i monitorować działalność użytkownika w systemie.



Kod źródłowy tego przykładu znajduje się w katalogu \Przykłady\Rozdział3\FileMonitor dołączonej do książki płyty CD-ROM, a kod źródłowy programu testującego działanie biblioteki znajduje się w katalogu \Przykłady\Rozdział3\FileMonitorTest. Zanim uruchomisz plik wykonywalny tego ostatniego, upewnij się, że w jego katalogu znajduje się plik biblioteki DLL.

3.9. Zarządzanie ikonami pulpitu

Ikony pulpitu są w rzeczywistości elementami obiektu sterującego typu *List View* (widok listy). Dzięki temu bardzo łatwo nimi zarządzać. Wystarczy znaleźć okno klasy ProgMan. Z tego okna można pobrać uchwyty widoku listy przechowującego ikony pulpitu.

Wskazówki te bardzo łatwo zaimplementować w kodzie źródłowym:

```
HWND DesktopHandle = FindWindow("ProgMan", 0);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
```

Powyższy kod najpierw wyszukuje w systemie okno, które zarejestrowano z klasą o nazwie ProgMan. Choć okna tego nie widać na ekranie, istnieje w systemie od czasów Windows 3.0, a program okna nosi nazwę *Menedżera programów*. W kolejnym wierszu mamy wywołanie funkcji pozyskującej uchwyty okna potomnego, a w kolejnym — okna potomnego okna uzyskanego poprzednio. W ten sposób otrzymujemy uchwyty obiektu systemowego klasy SysListView32. Przechowuje on wszystkie ikony pulpitu.

Ikony te możemy teraz kontrolować, przesyłając do owego obiektu komunikaty (funkcją SendMessage). Możemy, na przykład, wyrównać wszystkie ikony do lewej krawędzi ekranu:

```
SendMessage(DesktopHandle, LVM_ARRANGE, LVA_ALIGNLEFT, 0);
```

Przyjrzyjmy się parametrom tego wywołania:

- ◆ DesktopHandle — to uchwyty obiektu, do którego wysyłamy komunikat.
- ◆ Drugi parametr wywołania to typ komunikatu. LVM_ARRANGE sygnalizuje konieczność rozmieszczenia ikon.
- ◆ Trzeci parametr wywołania to pierwszy parametr komunikatu — LVA_ALIGNLEFT określa wyrównanie ikon do lewej.
- ◆ Czwarty parametr wywołania to drugi parametr komunikatu. Podajemy 0.

Jeśli zmienisz LVA_ALIGNLEFT na LVA_ALIGNTOP, ikony zostaną rozmieszczone wzdłuż górnej krawędzi pulpitu.

Poniższy wiersz usuwa wszystkie ikony z pulpitu:

```
SendMessage(DesktopHandle, LVM_DELETEALLITEMS, 0, 0);
```

Wywołanie jest podobne do poprzedniego, tyle że tu mamy komunikat LVM_DELETEALLITEMS wymuszający usunięcie wszystkich ikon z pulpitu. Uruchomienie takiego programu spowoduje wyczyszczenie pulpitu. Można co prawda odzyskać usunięte ikony — wystarczy przeładować system operacyjny. Osiągniesz jednak dobry efekt zaskoczenia, jeśli uruchomisz w systemie niewidzialny program, który od czasu do czasu będzie czyścić pulpit.

Teraz najciekawsze — przesuwanie ikon po pulpicie. Służy do tego następujący kod:

```
HWND DesktopHandle = FindWindow("ProgMan", 0);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
for (int i = 0; i < 200; i++)
    SendMessage(DesktopHandle, LVM_SETITEMPOSITION, 0, MAKELPARAM(10, i));
```

Podobnie jak poprzednio, na początku odnajdujemy uchwyty okna-obiektu zawierającego ikony. Następnie inicjujemy pętlę wykonywaną dla wartości i od 0 do 199, wywołującą funkcję SendMessage z następującymi parametrami:

- ◆ Uchwytem okna, do którego kierowany jest komunikat (tutaj jest to uchwyty obiektu zarządzającego ikonami pulpitu).
- ◆ Komunikatem. LVM_SETITEMPOSITION wymusza zmianę pozycji ikony.
- ◆ Pierwszym parametrem komunikatu — numerem ikony do przesunięcia.
- ◆ Drugim parametrem komunikatu — nową pozycję ikony. Parametr ten składa się z dwóch zmiennych: współrzędnej poziomej i pionowej ikony. Aby zmieścić te zmienne w jednym parametrze, są one kombinowane wywołaniem MAKELPARAM.

Powyższy kod pozwala na dowolne niemal zabawy z pulpitem. Jedyną jego wadą jest dziwaczne przesuwanie ikon na pulpicie Windows XP. W pozostałych systemach operacyjnych z rodziny Windows przesuwanie działa gładko i daje odpowiedni efekt.

Co jeszcze można zrobić z ikonami na pulpicie? Cóż, wszystko to, co da się zrobić z elementem sterującym widoku listy (*list view*). Zobaczmy.

3.9.1. Animowanie tekstu

Bardzo ciekawym efektem jest animowanie podpisów ikon. Wystarczy wiedzieć, jak zmienić kolor tekstu w podpisach ikon. Mając tę wiedzę, można zaprogramować pętlę animacji, która zmieniałaby kolor podpisów zgodnie z pewnym algorytmem. Jedyną trudnością jest konieczność odrysowywania pulpitu — zmiany kolorów będą widoczne jedynie po odświeżeniu obrazu pulpitu.

Aby zmienić kolor podpisów pod ikonami, należy przesyłać do zarządcy ikon pulpitu komunikat LVM_SETITEMTEXT. Pierwszy parametr komunikatu powinien mieć wartość zero, a drugi — być ustawiony na pożądany kolor podpisów. Aby, na przykład, zmienić kolor na czarny, wystarczy uruchomić taki kod:

```
HWND DesktopHandle = FindWindow("ProgMan", 0);
DesktopHandle = GetWindow(DesktopHandle, GW_CHILD);
```

```
DesktopHandle = GetWindow/DesktopHandle, GW_CHILD);
SendMessage/DesktopHandle, LVM_SETITEMTEXT, 0, (LPARAM) (COLORREF)0);
```

Wyzwaniem pozostaje tylko sposób odświeżenia pulpitu, tak aby można było zobaczyć zmiany.

3.9.2. Odświeżanie pulpitu

Kod przesuwający ikony pulpitu, prezentowany w poprzednim podrozdziale, jest mało efektywny, ponieważ na pulpicie nie widać właściwie animacji. Użytkownik zobaczy jedynie pozycję początkową i końcową, więc najciekawsze mu umknie. Można tę sytuację poprawić, odświeżając ikonę po zmianie jej pozycji. Służy do tego komunikat LVM_UPDATE:

```
HWND DesktopHandle = FindWindow("ProgMan", 0);
DesktopHandle = GetWindow/DesktopHandle, GW_CHILD);
DesktopHandle = GetWindow/DesktopHandle, GW_CHILD);
for (int i = 0; i < 200; i++)
{
    SendMessage/DesktopHandle, LVM_SETITEMPOSITION, 0, MAKELPARAM(10, i));
    SendMessage/DesktopHandle, LVM_UPDATE, 0, 0);
    Sleep(10);
}
```

Wewnątrz pętli zmieniamy pozycję pierwszej ikony pulpitu (ikony o numerze 0) i wywołujemy jej odrysowanie komunikatem LVM_UPDATE. Trzeci parametr wywołania SendMessage to numer ikony do odświeżenia. Jeśli zachodziłyby potrzeba odświeżenia obrazu drugiej ikony pulpitu, trzeba by zainicjować wywołanie:

```
SendMessage/DesktopHandle, LVM_UPDATE, 1, 0);
```



Kod źródłowy tego przykładu umieszczony jest w podkatalogu \Przykłady\Rozdział3\Arrangelcons dołączonej do książki płyty CD-ROM.

3.10. Żarty z wykorzystaniem schowka

Dowcipkować można z użyciem dowolnego niemal komponentu systemu, nie ujdzie więc naszej uwadze również pozytyczny schowek systemowy. Ten pozornie niewinny składnik Windows może być w ręku hakera wydajnym narzędziem. Wystarczy puścić wodze wyobraźni.

Schowek jest zwykle wykorzystywany do przenoszenia danych pomiędzy aplikacjami — najczęściej chodzi o kopowanie bloków tekstu. Czego użytkownik spodziewa się po schowku? Że wklejone z niego dane będą tymi samymi danymi, które wcześniej do niego skopiował. Zaskoczymy go.

W systemie Windows dostępna jest funkcja oraz zestaw komunikatów o zdarzeniach, które pozwalają na monitorowanie stanu schowka. Komunikaty te i funkcje są niezbędne

w działaniu aplikacji korzystających ze schowka, które powinny udostępniać funkcję wklejania ze schowka tylko wtedy, kiedy zawiera on dane o odpowiednim dla aplikacji formacie. Wykorzystajmy to do własnych celów.

Spróbujemy napisać program, który będzie monitorował stan schowka i „psuł” to, co zostanie do schowka skopiowane. Utwórz nową aplikację MFC (może ona wykorzystywać dialogi) i nazwij ją *ClipboardChange*.

Dodaj do projektu dwa zdarzenia, które będziemy wykorzystywać w programie do monitorowania schowka: ON_WM_CHANGECHAIN i ON_WM_DRAWCLIPBOARD. W tym celu otwórz plik kodu źródłowego *ClipboardChangeDlg.cpp*, znajdź w nim mapę komunikatów (MESSAGE_MAP) i uzupełnij ją następująco:

```
BEGIN_MESSAGE_MAP(CClipboardChangeDlg, CDialog)
    ON_WM_CHANGECHAIN()
    ON_WM_DRAWCLIPBOARD()
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    //}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Teraz otwórz plik nagłówkowy *ClipboardChangeDlg.h* i znajdź w nim deklaracje funkcji odpowiedzialnych za obsługę zdarzeń deklarowanych w mapie komunikatów. Powinny znajdować się w sekcji protected klasy okna dialogowego. Dodaj do nich dwie deklaracje:

```
afx_msg void OnChangeCbChain(HWND hWndRemove, HWND hWndAfter);
afx_msg void OnDrawClipboard();
```

Będziemy też potrzebować zmiennej typu HWND, w której będziemy przechowywać uchwyt okna przeglądarki schowka. Nazwij zmienną *ClipboardViewer*.

Wróć do pliku kodu źródłowego *ClipboardChangeDlg.cpp* i dodaj do niego kod obu funkcji. Nie będą one co prawda wywołane, zanim faktycznie nie uczynimy naszego programu przeglądarką zawartości schowka. Można to zrobić, uzupełniając funkcję OnInitDialog następującym wierszem:

```
ClipboardViewer = SetClipboardViewer();
```

Przyjrzyjmy się teraz funkcjom wywoływanym w reakcji na zdarzenia związane ze schowkiem. Ich kod prezentowany jest na listingu 3.10.

Listing 3.10. Przeglądarka schowka

```
void CClipboardChangeDlg::OnChangeCbChain(HWND hWndRemove, HWND hWndAfter)
{
    if (ClipboardViewer == hWndRemove)
        ClipboardViewer = hWndAfter;

    if (NULL != ClipboardViewer)
    {
        ::SendMessage(ClipboardViewer, WM_CHANGECHAIN,
                     (WPARAM)hWndRemove, (LPARAM)hWndAfter);
    }
}
```

```

    CClipboardChangeDlg::OnChangeCbChain(hWndRemove, hWndAfter);

}

void CClipboardChangeDlg::OnDrawClipboard()
{
    if (!OpenClipboard())
    {
        MessageBox("Schowek jest czasowo niedostępny");
        return;
    }
    if (!EmptyClipboard())
    {
        CloseClipboard();
        MessageBox("Nie można opróżnić schowka");
        return;
    }

    CString Text = "Coś nie tak? ";
    HGLOBAL hGlobal = GlobalAlloc(GMEM_MOVEABLE, Text.GetLength() + 1);

    if (!hGlobal)
    {
        CloseClipboard();
        MessageBox(CString("Błąd przydziału pamięci"));
        return;
    }

    strcpy((char *)GlobalLock(hGlobal), Text);
    GlobalUnlock(hGlobal);
    if (!SetClipboardData(CF_TEXT, hGlobal))
    {
        MessageBox("Błąd wpisu do schowka");
    }
    CloseClipboard();
}

```

Najciekawsze rzeczy dzieją się w funkcji OnDrawClipboard, która jest wywoływana za każdym razem, kiedy w schowku lądują nowe dane. W takim przypadku czyścimy zawartość schowka i wstawiamy do niego własne dane (napis „Coś nie tak?”), przez co użytkownik nie może korzystać z operacji kopiowania i wklejania.

Zanim będziemy mogli skorzystać ze schowka, musimy go otworzyć wywołaniem funkcji OpenClipboard. Jeśli schowek zostanie pomyślnie otwarty, funkcja zwróci wartość TRUE.

Następnie opróżniamy schowek funkcją EmptyClipboard. Jeśli operacja się powiedzie, funkcja zwróci TRUE. W przeciwnym przypadku schowek zostanie zamknięty, a nasz program wyświetli komunikat o błędzie. Do zamykania schowka służy funkcja CloseClipboard.

Możemy teraz wstawić do schowka własne dane. W tym celu musimy przydzielić dla nich odpowiedni blok pamięci globalnej i umieścić w niej np. własny napis. W naszym przykładzie jest to napis „Coś nie tak?”. Następnie wstawiamy tak spreparowane dane do schowka, wywołując w tym celu funkcję SetClipboardData. Funkcja przyjmuje dwa parametry:

- ◆ Stałą definiującą typ danych. Podajemy CF_TEXT, czyli dane tekstowe.
- ◆ Wskaźnik pamięci, w której znajdują się dane przeznaczone do umieszczenia w schowku.

Po zakończeniu tych manipulacji należy schowek zamknąć, wywołując CloseClipboard.

Jak widać, schowek, choć tak przydatny, może kiedyś zacząć być kłopotliwy. Uruchom program i spróbuj skorzystać ze schowka — czego byś do niego nie wkleiał, za każdym razem wyjdzie z niego napis „Coś nie tak?».



Kod źródłowy tego przykładu umieszczony jest w podkatalogu *Przykłady\Rozdział3\ClipboardChange* dołączonej do książki płyty CD-ROM.

Rozdział 4.

Sieci komputerowe

Pierwotne znaczenie słowa haker zawierało w sobie pojęcie osoby biegłej w programowaniu, znającej tajniki działania systemów operacyjnych i sieci komputerowych. Co do programowania, to już się nim zajęliśmy i nie odpuścimy aż do końca książki. W poprzednich rozdziałach ilustrowaliśmy ciekawymi i żartobliwymi przykładami sposób działania systemu operacyjnego Windows. Pora na zainteresowanie się również programowaniem sieciowym.

W niniejszym rozdziale przedstawię Czytelnikowi sieciowe możliwości programowania w C++. Pokażę, jak tworzyć proste, acz efektowne i efektywne narzędzia przy wykorzystaniu obiektów Visual C++ i biblioteki sieciowej WinSock.

Z początku omówienie programowania sieciowego będzie ograniczone do modelu obiektowego udostępnianego w wybranym przez nas środowisku programistycznym. Później zajmiemy się już niskopoziomowym programowaniem sieciowym.

Nie chciałbym przy tym przeciągać pamięci Czytelnika kwestiami programowania niskopoziomowego, od razu omawiając poszczególne funkcje API biblioteki sieciowej. Taka nauka byłaby nieefektywna. Zaczniemy więc jak zwykle od rzeczy najprostszych, powoli przesuwając się do zagadnień bardziej złożonych.

4.1. Teoria sieci i protokołów sieciowych

Zanim zobaczymy pierwszy przykład, chciałbym zapoznać Czytelnika z pewną ilością teoretycznych informacji o sieciach. Nie zajmie to wiele czasu, a pozwoli na lepsze zrozumienie działania przykładów. Lektura niniejszego rozdziału wymaga bowiem pewnej wiedzy o podstawach funkcjonowania sieci komputerowych i stosowanych w nich protokołach.

Za każdym razem, kiedy za pośrednictwem sieci komputerowej dokonywany jest transfer danych, następuje przepływ tych danych pomiędzy dwoma komputerami. Jak to się dzieje? Owszem, umożliwiają to specjalne protokoły sieciowe. Jest ich jednak całkiem sporo. Który z nich jest najodpowiedniejszy w danej sytuacji? Jak wybrać najwydajniejszy protokół? Jak taki protokół działa? Spróbujmy odpowiedzieć sobie na te pytania.

Zanim zaczniemy omawiać protokoły, chciałbym opowiedzieć o modelu połączeń systemów komputerowych OSI (ang. *Open Systems Interconnection*) opracowanym przez międzynarodową organizację standaryzacyjną ISO. Zgodnie z tym modelem w połączeniu sieciowym wyróżnia się siedem warstw:

1. Warstwę fizyczną odpowiedzialną za transfer bitów w kanale fizycznym (np. w kablu koncentrycznym, skrętce czy światłowodzie). Tutaj definiowane są charakterystyki otoczenia fizycznego i parametry sygnałów elektrycznych.
2. Warstwę łączą danych odpowiedzialną za transfer ramek danych pomiędzy dwoma dowolnymi węzłami sieci w topologii standardowej (np. magistrali) albo pomiędzy dwoma sąsiadującymi węzłami sieci w topologii dowolnej. W tej warstwie wprowadzane jest pojęcie adresu fizycznego węzła — MAC.
3. Warstwę sieciową odpowiedzialną za transfer pakietów pomiędzy węzłami sieci o dowolnej topologii. Protokoły tej warstwy nie gwarantują doręczenia pakietów.
4. Warstwę transportową odpowiedzialną za transfer pakietów pomiędzy dowolnymi węzłami sieci o dowolnej topologii i gwarantującą określony poziom niezawodności transferu. Warstwa ta obejmuje również narzędzia i mechanizmy nawiązywania połączenia i buforowania, numerowania oraz porządkowania pakietów.
5. Warstwę sesji kontrolującą interakcję pomiędzy dwoma węzłami.
6. Warstwę prezentacji odpowiedzialną za transformacje danych (szifrowanie czy kompresję).
7. Warstwę aplikacji składającą się z zestawu usług sieciowych (FTP, poczta elektroniczna itd.) dostępnych użytkownikom i aplikacjom.

Jeśli przejrzałeś tę listę z odpowiednią uwagą, zapewne zdążyłeś dojść do wniosku, że pierwsze trzy warstwy implementowane są w sprzęcie, np. w kartach sieciowych, routeraх, koncentratorach, mostach itd. Ostatnie trzy warstwy implementowane są na poziomie systemu operacyjnego i działających w nim aplikacji. Warstwę pośrednią (łącznikową) stanowi zaś warstwa czwarta — transportowa.

Jak, w oparciu o taki model, działa protokół sieciowy? W pierwszym rzędzie występuje warstwa aplikacji. Ona generuje pakiet danych, opatrując go odpowiednim nagłówkiem. Następnie pakiet taki jest z warstwy aplikacji przekazywany do następnej warstwy (prezentacji). Tutaj jest uzupełniany kolejnym nagłówkiem i przekazywany do kolejnej warstwy. W ten sposób dociera wreszcie do warstwy fizycznej, która nadaje pakiet za pośrednictwem medium transmisyjnego.

Kiedy komputer odbiera pakiet, realizuje procedurę odwrotną. Pakiet jest odbierany w warstwie fizycznej i przesyłany do warstwy łączącej danych, która usuwa z niego właściwe dla swojej warstwy nagłówki i przekazuje do warstwy sieciowej. Ta ogałaça pakiet z nagłówków warstwy sieciowej i przekazuje jeszcze dalej. Ostatecznie pakiet ląduje w warstwie aplikacji, w której nie zawiera już żadnych informacji poza tymi umieszczonymi w nim przez aplikację-nadawcę przed wysłaniem pakietu siecią.

Transfer danych nie musi koniecznie być inicjowany w warstwie siódmej. Jeśli wykorzystywany do transmisji protokół operuje w warstwie czwartej, wędrówka pakietu w dół rozpocznie się właśnie od warstwy czwartej. Liczba warstw protokołu determinuje możliwości transmisji i wymagania co do sposobu transmisji danych.

Im protokół bliższy jest warstwy aplikacji, tym więcej może mieć możliwości, ale również tym większy wprowadza narzut (w postaci dłuższego i bardziej złożonego nagłówka pakietu warstwy aplikacji). Protokoły omawiane w niniejszej książce operują w różnych warstwach, stąd różnice w zakresie ich stosowania i możliwościach transmisyjnych.

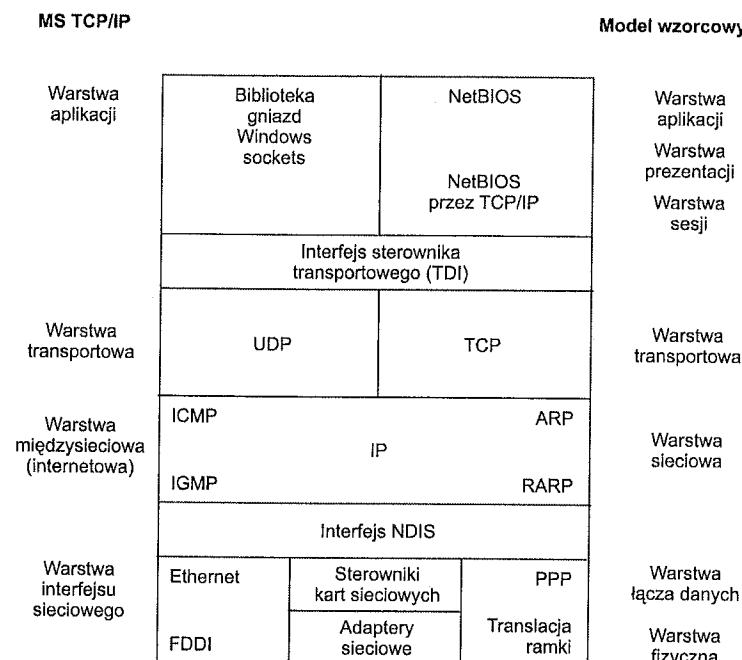
Firma Microsoft zaimplementowała protokół TCP/IP w modelu OSI na swoją własną modłę (z pewnymi odchyleniami od standardu). Oczywiście model OSI należy traktować jako wzorcowy, ale nie jako obowiązujący — jego wytyczne są jedynie zaleceniami. Mimo to firma Microsoft nie powinna go zmieniać, nawet jeśli zmiana nie dotyczy kwestii zasadniczych, a jedynie nazw i numerowania warstw.

Implementacja TCP/IP w wydaniu Microsoftu składa się z czterech, a nie siedmiu warstw. Nie oznacza to, że warstw zabrakło. Po prostu pojedyncza warstwa przejęła zadania trzech warstw modelu OSI. Chodzi o warstwę aplikacji, która w modelu firmy Microsoft łączy cechy warstwy aplikacji, reprezentacji i sesji.

Różnice pomiędzy modelem wzorcowym a modelem Microsoftu ilustruje rysunek 4.1. Nazwy wprowadzone w modelu Microsoftu znajdują się po lewej stronie; po prawej mamy nazwy warstw z modelu OSI. Środek rysunku zajmują konkretne protokoły obsługujące poszczególne warstwy. Próbowałem przypisać je do tych warstw, w których faktycznie operują. Warto przyjrzeć się dobrze rysunkowi, przyda się nam w przyszłości.

4.1.1. Protokoły sieciowe

Zanim zaczniesz pisać programy sieciowe, powinieneś poznać sieciowe protokoły i zrozumieć podstawy ich działania. Z tego względu w niniejszym punkcie przedstawię najważniejsze elementy tych protokołów, które mają największe znaczenie dla programisty sieciowego. Czytelnik będzie mógł zaobserwować różnice pomiędzy poszczególnymi protokołami i dowie się, dlaczego wybór protokołu przesyłania danych pomiędzy komputerami nie jest obojętny. Tymczasem decyzja co do wyboru protokołu jest bardzo trudna, bo determinuje przyszły kształt programu.



Rysunek 4.1. Model OSI i jego implementacja w wydaniu Microsoft

Protokół IP

Jeśli spojrzyesz na diagram modelu sieciowego (rysunek 4.1), zobaczyysz, że protokół IP jest protokołem warstwy sieciowej. Można wysnuć z tego wniosek, że protokół IP implementuje funkcje sieciowe, czyli realizuje transmisję pakietów pomiędzy dowolnymi węzłami sieci o dowolnej topologii.

Protokół IP nie ustanawia na potrzeby transmisji danych wirtualnego połączenia. Informacje są w nim przesyłane pomiędzy komputerami w pakietach (zwanych tu datagramami). Oznacza to, że IP przesyła pakiety siecią bez żadnych potwierdzeń ich doręczenia, a więc nie gwarantuje również poprawności transmisji. Tymczasem, jeśli choć jeden ze stu pakietów składających się na transmisję zostanie utracony i nie osiągnie węzła odbiorcy, integralność transmitowanych danych zostanie naruszona, uniemożliwiając odtworzenie danych po stronie odbiorcy.

Wszystkie czynności niezbędne do uzyskania pewności doręczenia i utrzymania spójności danych muszą być implementowane w protokołach warstw wyższych.

Każdy pakiet IP zawiera adres nadawcy i adres odbiorcy, identyfikator protokołu, czas życia pakietu (TTL) oraz sumę kontrolną służącą do sprawdzania spójności pakietu. Jednak czynność wykrycia utraty spójności na podstawie tej sumy kontrolnej może wykonać jedynie odbiorca pakietu. Kiedy węzeł docelowy odbiera pakiet, sprawdza jego spójność. Jeśli wszystko jest w porządku, przetwarza pakiet, przekazując go do

wyższych warstw sieciowych. W przeciwnym przypadku pakiet jest ignorowany. Jeśli więc odbiorca wykryje błąd, nie zgłosi tego faktu nadawcy z żądaniem powtórzenia transmisji. Z tego punktu widzenia protokół IP jest protokołem zawodnym.

Protokół ARP a protokół RARP

Protokół Address Resolution Protocol (ARP) wykorzystywany jest do kojarzenia z adresami MAC węzłów sieci komputerowej adresów IP. Przed wysłaniem danych do komputera w sieci lokalnej nadawca musi określić adres MAC odbiorcy, znając często jedynie jego IP. Wtedy musi skorzystać z usług protokołu ARP.

Kiedy nadawca wysyła żądanego ARP celem odszukania adresu MAC, implementacja protokołu przegląda najpierw lokalną pamięć podręczną wyników poprzednich żądań. Jeśli ostatnio ktoś odwoływał się do danego adresu IP w poszukiwaniu adresu MAC, oba adresy powinny być przechowywane w lokalnej pamięci i na jej podstawie można zrealizować żądanego ARP. Jeśli pamięć podręczna nie zawiera potrzebnych informacji, inicjowane jest rozgłoszeniowe żądanego ARP. Żądanie to trafia do wszystkich komputerów danej sieci. Każdy z tych komputerów odbiera pakiet ARP i porównuje zawarty w nim adres IP ze swoim adresem IP. Ten komputer, którego adres IP zgadza się z poszukiwanym, odpowiada na żądanie, nadając własny adres MAC. Adres ten powinien być niepowtarzalny (jest zaszywany w urządzeniu sieciowym — np. karcie sieciowej — w momencie jej produkcji), więc na żądanie ARP powinna napływać tylko jedna odpowiedź. Warto jednak mieć świadomość, że adres MAC może zostać zafałszowany, co może spowodować pojawienie się dwóch odpowiedzi.

Protokół RARP działa odwrotnie, tzn. określa adres IP na podstawie znanego adresu MAC. Procedura wyszukiwania adresu jest analogiczna do tej stosowanej w ARP.

4.1.2. Protokoły transportowe

W warstwie transportowej królują dwa protokoły: UDP i TCP; oba wykorzystywane są w połączeniu z IP — kiedy w sieci ma być przesłany pakiet protokołu TCP albo UDP, pakiet trafia do warstwy sieciowej i jest umieszczany w pakiecie protokołu IP. Jest przy tym uzupełniany nagłówkiem IP z adresami nadawcy i odbiorcy, parametrem TTL i paroma innymi atrybutami protokołu IP. Dopiero wtedy pakiet jest przekazywany dalej w dół, do warstwy fizycznej. „Goły” pakiet TCP czy UDP nie nadaje się do przesyłania siecią komputerową, ponieważ nie zawiera informacji o odbiorcy pakietu — informacje te są umieszczane dopiero w nagłówku pakietu IP w warstwie sieciowej.

Przyjrzyjmy się bliżej obu protokołom warstwy transportowej.

Protokół UDP — szybki

Podobnie jak w protokole IP, protokół UDP nie ustanawia żadnego rodzaju połączenia pomiędzy węzłami wymieniającymi dane. Dane są po prostu nadawane w sieci, a protokół nie dysponuje żadnymi mechanizmami potwierdzenia odbioru czy wykrywania uszkodzeń transmisji. Jeśli dane zostaną utracone albo zniekształcone, nadawca

nigdy się o tym nie dowie (przynajmniej nie za pośrednictwem mechanizmów protokołu). Z tego względu ważnych informacji nie należy przesyłać protokołem UDP ani „gólym” protokołem IP.

Ponieważ w UDP nie ma pojęcia połączenia, protokół działa bardzo szybko (kilukrotnie szybciej niż opisywany dalej TCP) choćby dlatego, że nie realizuje kosztownej czasowej procedury nawiązywania połączenia. Z racji swej szybkości protokół UDP wykorzystywany jest chętnie w sytuacjach, w których spójność danych ma mniejsze znaczenie. Wykorzystują go, na przykład, internetowe rozgłoszenie radiowe. Emitują one swoje audycje do internetu, a jeśli słuchacz utraci jeden czy kilka pakietów, w najgorszym przypadku zauważa „przeskok” dźwięku. W tym zastosowaniu taka zawodność jest całkowicie akceptowalna.

Wysoka prędkość transmisji osiągnięta została kosztem bezpieczeństwa. Z racji braku połączenia logicznego pomiędzy nadawcą a odbiorcą nie ma żadnej gwarancji, że odbierane dane pochodzą od zaufanej strony. Protokół UDP jest podatny na fałszowanie źródła transmisji — eliminuje to go z zastosowań kładących nacisk na bezpieczeństwo wymiany danych.

Podsumowując, protokół UDP jest szybki, ale może być wykorzystywany jedynie do transmisji danych mało wartościowych (z racji możliwości ich utraty) albo jawnych (dane poufne mogłyby zostać przechwycone).

Protokół TCP — wolniejszy, ale solidniejszy

Jak już wspomniałem, protokół TCP operuje w tej samej warstwie co UDP i również bazuje na protokole IP, który służy mu do transmisji międzysieciowej. Ścisłe powiązanie tych protokołów powoduje, że sposób transmisji z ich wykorzystaniem określa się mianem protokołu TCP/IP.

W przeciwieństwie do UDP, TCP rekompensuje wady swojego woła roboczego, którym jest tu IP. Protokół TCP zawiera mechanizmy nawiązywania połączenia pomiędzy nadawcą a odbiorcą, wykrywania utraty spójności danych i gwarancji dostarczenia.

Kiedy nadawca przesyła dane za pośrednictwem protokołu TCP, włącza zegar. Jeśli odbiorca nie odpowie na wysłane dane w ciągu określonego czasu, nadawca próbuje ponownej transmisji danych. Jeśli odbiorca odbierze dane uszkodzone (zniekształcone) powinien powiadomić o tym nadawcę i zażądać powtórzenia transmisji zniekształconego albo utraconego pakietu.

Kiedy zachodzi potrzeba przesłania większej ilości danych, których nie sposób pomieścić w pojedynczym pakiecie TCP, transmisja jest realizowana w wielu takich pakietach. Są one wysypane grupami po kilka pakietów (ich liczba zależy od parametrów stosu TCP/IP). Kiedy serwer odbiera taki zestaw pakietów, powinien odpowiednio je uporządkować (jeśli kolejność ich odbierania nie zgadza się z numerami pakietów).

Protokół TCP jest ze względu na wymienione funkcje wolniejszy niż UDP. Z drugiej strony zastosowanie protokołu TCP daje pewność transmisji, przez co protokół ten nadaje się do wykorzystania w transmisjach informacji wartościowych i wrażliwych.

Protokół nie jest co prawda w pełni bezpieczny (brak mu mechanizmów szyfrowania), jest bez porównania solidniejszy niż UDP. Jest też co prawda podatny na fałszowanie adresów, ale fałszerstwo takie nie jest tak proste jak w przypadku UDP. Czytelnik sam się o tym przekona, kiedy dotrzymy do omówienia sposobu nawiązywania połączeń. Mimo wszystko nawet TCP nie jest całkiem bezpieczny — hakerzy wiedzą, jak go złamać.

TCP — zagrożenia i słabości

W jaki sposób w TCP zabezpieczane jest połączenie pomiędzy węzłami transmisji? Transmisja rozpoczyna się od procedury nawiązywania połączenia pomiędzy dwoma komputerami. Procedura ta przebiega następująco:

1. Klient, który chce nawiązać połączenie, wysyła do odbiorcy pakiet ze znacznikiem SYN, numerem portu, z którym chce się połączyć, i specjalnym numerem pakietu (jest on zazwyczaj generowany losowo).
2. Odbiorca odpowiada pakietem z ustawionymi znacznikami SYN i ACK oraz własnym numerem pakietu. Dodatkowo potwierdza odbiór pakietu inicjującego połączenie, przesyłając w nagłówku numer pakietu SYN zwiększyony o jeden.
3. Inicjator potwierdza odbiór pakietu SYN+ACK odbiorcy, wysyłając pakiet z ustawionym znacznikiem ACK i numerem pakietu odbiorcy zwiększyonym o jeden.

Jak widać, w czasie nawiązywania połączenia pomiędzy stronami transmisji wymieniane są specjalne numery. Następnie numery te służą do wykrywania spójności połączenia (utraty pakietów), stanowią też zabezpieczenie połączenia przed ingerencją stron trzecich. Jeśli taka strona spróbowałaby podszyć się pod jedną ze stron połączenia, musiałaby odgadnąć owe specjalne numery. Ponieważ są one wielkie i losowe, zadanie to nie jest proste. Jednak nie jest też niemożliwe — udało się to między innymi niejakiemu Kevinowi Mitnickowi. Ale to już całkiem inna historia.

Każdy pakiet TCP potwierdzany jest specjalnym pakietem ACK — mamy więc mechanizm pewności doręczenia.

4.1.3. Protokoły warstwy aplikacji — tajemniczy NetBIOS

NetBIOS (ang. *Network Basic Input Output System*) to standardowy interfejs programowania aplikacji. Innymi słowy, to po prostu zestaw funkcji API implementujący obsługę sieci (w istocie NetBIOS składa się z tylko jednej funkcji!). NetBIOS został opracowany dla IBM-a przez firmę Sytek Corporation w roku 1983.

NetBIOS definiuje jedynie programowy standard transmisji danych. Określa więc, w jaki sposób powinien zachowywać się program transmitujący dane za pośrednictwem sieci. Standard nie odnosi się w ogóle do kwestii fizycznej transmisji danych.

Na rysunku 4.1 NetBIOS znajduje się na szczycie diagramu, obejmując warstwę aplikacji, reprezentacji i sesji.

NetBIOS reguluje wyższe kwestie formatowania danych przesyłanych siecią, nie mówi jednak nic na temat protokołu transportowego — może więc bazować tak na parze TCP/IP, jak i na IPX/SPX i innych. NetBIOS jest więc protokołem niezależnym od warstwy transportowej. Podczas, gdy inne protokoły wyższych warstw modelu OSI są przypisane do konkretnych protokołów transportowych, pakiety NetBIOS mogą być przesyłane dowolnymi protokołami warstwy transportowej. Cóż za elastyczność! Wyobraź sobie, że napisaliśmy aplikację sieciową wykorzystującą protokół NetBIOS. Będzie ona mogła działać zarówno w sieciach uniksowych i windowsowych, bazujących zazwyczaj na TCP, jak i w sieciach Novell, korzystających z protokołu IPX.

Z drugiej strony, kiedy dwa komputery próbują nawiązać połączenie NetBIOS, muszą uzgodnić wcześniej ten sam protokół transportowy. Jeśli jeden z nich będzie osadzał pakiety NetBIOS w pakietach TCP, a drugi w pakietach IPX, nie dojdzie do skutecznej komunikacji. Protokół transportowy może być co prawda dowolny, ale taki sam dla obu stron połączenia.

Powiniem wspomnieć też, że nie wszystkie protokoły transportowe mogą bezproblemowo przenosić pakiety NetBIOS. Na przykład para IPX/SPX nie jest do tego przystosowana. Aby umożliwić transport NetBIOS za pośrednictwem tej pary, należy najpierw zainstalować w systemie protokół transportowy opisywany jako „NWLink IPX/SPX/NetBIOS Compatible Transport Protocol”.

Ponieważ najczęściej wykorzystywanym protokołem transportowym dla NetBIOS jest TCP, a ten posiada mechanizmy ustanawiania połączenia, w pakietach NetBIOS można zazwyczaj przesyłać dane wrażliwe. W takim układzie dla TCP/IP odpowiedzialna jest za spójność i niezawodność dostarczenia pakietów, a NetBIOS tworzy wygodne środowisko obsługi sieci i programowania aplikacji sieciowych. Jeśli, na przykład, masz zamiar napisać program przesyłający przez sieć pliki, możesz polegać na protokole NetBIOS.

4.1.4. NetBEUI

W roku 1985 IBM zdecydował o uczynieniu protokołu NetBIOS samodzielnym protokołem, zdolnym również do obsługi fizycznego transferu danych w sieciach komputerowych. W ten sposób powstał NetBEUI (ang. *NetBIOS Extended User Interface*). Definiuje on transmisję na poziomie fizycznym jako podbudowę protokołu NetBIOS.

NetBEUI nie jest protokołem routowalnym, więc pierwszy router na drodze pakietu NetBEUI odbije pakiet do sieci wewnętrznej, jak bramkarz piłkę celującą w „okienko”. Innymi słowy, jeśli na drodze pakietu pomiędzy dwoma komputerami znajduje się router, pakiet nie ma możliwości dotarcia do celu, a ustanowienie połączenia jest niemożliwe.

4.1.5. Gniazda w Windows

Gniazda to interfejs programistyczny ułatwiający interakcję różnych aplikacji. Współczesne gniazda różnią się od mechanizmów programowania sieciowego zaimplementowanych pierwotnie pod tą nazwą w systemie BSD Unix. Interfejs ten ułatwiał programistom wykorzystywanie protokołu TCP/IP w wyższych warstwach modelu OSI.

Za pośrednictwem gniazd można w prosty sposób zaimplementować większość protokołów wykorzystywanych w internecie, w tym HTTP, FTP, POP3, SMTP i tak dalej. Wszystkie one wykorzystują w roli protokołu transportowego albo UDP, albo TCP i są zazwyczaj oprogramowywane przy użyciu bibliotek gniazd.

4.1.6. Protokoły IPX/SPX

Nie powinniśmy zapominać o kilku kolejnych protokołach, które co prawda są wykorzystywane rzadziej, ale również bywają przydatne. Pora na parę IPX/SPX.

Współczesnie protokół IPX (ang. *Internet Packet eXchange*) jest wykorzystywany chyba tylko w sieciach Novell. Nasze ulubione systemy Windows mają specjalne narzędzie w postaci klienta sieci Novell, które pozwala na włączenie systemu do takich sieci. Podobnie jak IP i UDP, IPX nie ustanawia połączenia pomiędzy komunikującymi się stronami, nie daje więc pewności doręczenia; z wymienionymi protokołami dzieli też pozostałe wady i zalety.

SPX (ang. *Sequence Packet eXchange*) to protokół transportowy dla IPX ustanawiający połączenie i kontrolujący spójność danych. Jeśli więc przy przesyłaniu danych protokołem IPX potrzebne jest zapewnienie niezawodności, należy skorzystać z kombinacji protokołów IPX/SPX albo IPX/SPX11.

Czasy popularności protokołu IPX już minęły, pamiętam jednak czasy królowania systemu DOS — w owych czasach wszystkie gry sieciowe bazowały na IPX.

Okazuje się, że w wymianie danych w internecie uczestniczy wiele protokołów, jednak nie są one zupełnie niezależne od siebie (np. HTTP wykorzystuje w warstwie transportowej protokół TCP, a w warstwie sieciowej — protokół IP). Przy tym protokół sprawdzający się w jednym zastosowaniu nie musi nadawać się do innego — jak widać, nie ma czegoś takiego jak protokół idealny. Każdy z nich ma swoje zalety i specyficzne dla siebie wady.

Mimo to model OSI, przyjęty w zarodku sieci internet, do dziś stanowi jej fundament. Każdy z jego elementów sprawdza się również obecnie. Jego główną zaletą jest za to, że ukrywa przed użytkownikiem szczegółowe, skomplikowane zagadnienia komunikacji w sieciach komputerowych. Stary, sprawdzony OSI wciąż wykonuje dobrą robotę.

4.1.7. Porty

Zanim zaczniemy pisać własne programy sieciowe, powinniśmy powiedzieć sobie jeszcze o portach. Wyobraź sobie, że karta sieciowa Twojego komputera odebrała pakiet danych. Jak system operacyjny ma określić, dla którego z uruchomionych programów jest on przeznaczony? Może przecież być częścią transmisji do przeglądarki Internet Explorer, ale równie dobrze fragmentem poczty elektronicznej albo transmisji Twojego własnego programu. Rozróżnienia takiego dokonuje się za pośrednictwem numerów portów.

Kiedy program nawiązuje połączenie z innym programem (serwerem), otwiera w swoim systemie operacyjnym port sieciowy i przekazuje jego numer serwerowi. Serwer może wtedy opatrzyć przesyłane do programu (klienta) pakiety numerem portu, pod który są adresowane. Pełny adres składa się więc z adresu IP komputera i numeru portu odbiorcy. W takim układzie system operacyjny nie ma żadnych trudności z rozsyłaniem pakietów sieciowych pomiędzy uruchomionymi aplikacjami — dostarcza je po prostu do określonych portów.

Aby z kolei połączyć się z serwerem, musimy znać jego adres IP i oczywiście numer portu wykorzystywanego przez program serwera — na jednym komputerze (węźle sieciowym) może działać więcej niż jedna aplikacja sieciowa, ale każda z nich zajmować będzie osobny port.

Z tego, co powiedzieliśmy powyżej, wynika, że dany port może otworzyć tylko jedna aplikacja. Jeśli dwa różne programy spróbowią otworzyć ten sam port (np. port numer 21), system (nie tylko Windows, ale każdy sieciowy system operacyjny) nie będzie w stanie określić właściwego adresata pakietu.

Porty są numerowane liczbami z zakresu od 1 do 65535. Zapis takiego numeru wymaga jedynie dwóch bajtów, więc jego stosowanie nie wprowadza narzutu transmisyjnego. Zalecam stosowanie numerów portów powyżej 1024, ponieważ wśród numerów niższych znaczna część jest zarezerwowana, co może doprowadzić do konfliktu Twojej aplikacji z innymi aplikacjami sieciowymi.

Przyjrzymy się teraz bliżej kilku protokołom i funkcjom sieciowym stosowanym w systemie Windows. Nie mam jednak zamiaru wszystkiego szczegółowo opisywać — zamiast tego postaram się zaprezentować kilka ciekawych przykładów ilustrujących zastosowanie najważniejszych elementów sieciowego interfejsu programistycznego Windows.

4.2. Korzystanie z zasobów otoczenia sieciowego

System Windows wyposażony jest w bardzo ciekawą funkcję pozwalającą użytkownikom na wymienianie informacji pomiędzy komputerami za pośrednictwem współużytkowanych zasobów. Użytkownik może, na przykład, udostępnić folder użytkownikom

sieciowym i będą oni mogli (dysponując odpowiednimi uprawnieniami i znając stosowne hasło) korzystać z tego folderu jak z folderu znajdującego się na dysku lokalnym — owszem, folder taki można zamontować w systemie jako kolejny dysk. Korzystanie z tak udostępnionych zasobów odbywa się przy tym za pośrednictwem standardowych narzędzi Windows (głównie okna Eksploratora).

Kiedy aplikacja odwołuje się do pliku, system operacyjny wykrywa urządzenie, na którym przechowywany jest dany plik (zasób). Jeśli zasób zlokalizowany jest na zdalnym węźle sieci, żądanie wejścia-wyjścia jest przesyłane do tegoż węzła za pośrednictwem sieci komputerowej. Kiedy więc użytkownik ma zamiar skorzystać z zasobu udostępnianego w sieci, system operacyjny dokonuje przekierowania żądania wejścia-wyjścia z węzła lokalnego do zdalnego.

Przypuśćmy, że dysponujemy w systemie dyskiem Z, który jest w istocie folderem dysku komputera zdalnego, podłączonego do sieci komputerowej. Przy każdym odwołaniu do tegoż dysku system operacyjny przekierowuje operacje wejścia-wyjścia do mechanizmu ustanawiającego połączenie sieciowe ze zdalnym komputerem zarządzającym danym zasobem. Dzięki temu użytkownik może korzystać z zasobów lokalnych i zdalnych za pośrednictwem tych samych narzędzi systemowych. Znakomicie ułatwia to tworzenie aplikacji działających w lokalnych sieciach komputerowych. Program nie musi być specjalnie przygotowywany do obsługi sieci komputerowej — odwołania do folderu zdalnego realizowane są tak samo jak do folderu lokalnego.

Bardziej szczegółowych informacji o mechanizmie przekierowania żądań wejścia-wyjścia w systemach z rodziną Windows należałoby szukać w dokumentacji systemu i książkach poświęconych tej tematyce. Informacje te są jednak zbędne nie tylko szeregowemu użytkownikowi systemu, ale również części programistów — ci i tak nie muszą samodzielnie oprogramowywać tego mechanizmu, bo przekierowanie jest realizowane na poziomie systemu operacyjnego, a nie ich programów.

Aby udostępnić użytkownikom zasoby komputera sieciowego, nie trzeba montować foldera zdalnego jako dysku sieciowego. Wystarczy w odwołaniu właściwie określić ścieżkę dostępu do zasobu. Ścieżka taka powinna być skonstruowana zgodnie z wytycznymi uniwersalnej konwencji nazewniczej UNC (ang. *Universal Naming Convention*), umożliwiającej jednoznaczne identyfikowanie zasobów w sieci komputerowej, nie tylko plików i katalogów, ale również np. drukarek, bez konieczności przypisywania im liter dysków lokalnych. Wtedy można opierać się nie na nazwach dysków lokalnych, a na nazwach komputerów przechowujących dane zasób.

UNC w ogólnej postaci ma następującą składnię:

\\\komputer\nazwa\ścieżka

Pełna nazwa zasobu sieciowego rozpoczyna się od dwóch lewych ukośników (\\\). Za nimi występować powinna nazwa komputera, który zarządza pożdanym zasobem. Dalej mamy nazwę folderu sieciowego, a na końcu ścieżkę (względem tegoż folderu) dostępu do zasobu.

Załóżmy, że w sieci działa komputer o nazwie *Tomek* ze wspólnie wykorzystywanym sieciowym folderem o nazwie *Audio*. Folder ten zawiera plik o nazwie *MySound.wav*. Aby odwołać się do tego pliku, należy skonstruować UNC w postaci:

```
\Tomek\Audio\MySound.wav
```

Listing 4.1 ilustruje sposób programowego utworzenia pliku we wspólnie wykorzystywanym folderze sieciowym przechowywanym w komputerze o nazwie *Notebook*:

Listing 4.1. Tworzenie pliku w folderze udostępnianym przez komputer zdalny

```
void CreateNetFile()
{
    HANDLE FileHandle;
    DWORD BWritten;

    // Utwórz plik \notebook\temp\myfile.txt
    if ((FileHandle = CreateFile("\\\\notebook\\\\temp\\\\myfile.txt",
        GENERIC_WRITE | GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
    )) == INVALID_HANDLE_VALUE)
    {
        MessageBox(0, "Błąd utworzenia pliku", "Błąd", 0);
        return;
    }

    // Zapisz w pliku 12 znaków
    if (WriteFile(FileHandle, "Zapis próbny", 12, &BWritten, NULL) == 0)
    {
        MessageBox(0, "Błąd zapisu w pliku", "Błąd", 0);
        return;
    }

    // Zamknij plik
    CloseHandle(FileHandle);
}
```

Na samym początku mamy wywołanie funkcji WinAPI o nazwie *CreateFile* tworzącej plik. Funkcja ta wymaga przekazania następujących parametrów:

- ◆ Ścieżki do tworzonego pliku.
- ◆ Trybu dostępu — czy plik po utworzeniu ma być dostępny do odczytu (GENERIC_READ) i zapisu (GENERIC_WRITE).
- ◆ Trybu dostępu dla aplikacji zewnętrznych — czy mogą one odczytywać zawartość pliku (FILE_SHARE_READ) i zapisywać w pliku (SHARE_FILE_WRITE).
- ◆ Atrybutów bezpieczeństwa (tu nie określamy żadnych — przekazujemy NULL).
- ◆ Metody otwierania pliku — przekazujemy CREATE_ALWAYS, czyli bezwzględne tworzenie nowego pliku — jeśli plik o takiej nazwie i ścieżce dostępu już istnieje, jego zawartość zostanie zamazana.

- ◆ Atrybutów tworzonego pliku — wymuszymy zwykłe atrybuty (FILE_ATTRIBUTE_NORMAL).
- ◆ Uchwytu szablonu tworzenia pliku.

Funkcja *CreateFile* zwraca uchwyt utworzonego i natychmiast otwartego pliku. Wartość *INVALID_HANDLE_VALUE* oznacza, że operacja tworzenia pliku była nieskuteczna.

Do zapisu danych w pliku służy funkcja *WriteFile*. Przyjmuje ona następujące parametry:

- ◆ Deskryptor (uchwyt) otwartego pliku.
- ◆ Dane do zapisania.
- ◆ Liczbę bajtów danych do zapisania.
- ◆ Liczbę bajtów zapisanych w pliku — to parametr wyjściowy, należy przekazać w jego miejsce zmienną typu *DWORD*.
- ◆ Strukturę wymaganą, kiedy plik było otwierany w trybie nakładania operacji wejścia-wyjścia.

Jeśli zapis był skuteczny, funkcja powinna zwrócić zero.

Po zakończeniu manipulowania plikiem należy plik zamknąć. Służy do tego funkcja *CloseHandle*, której parametrem jest uchwyt zamkniętego pliku.



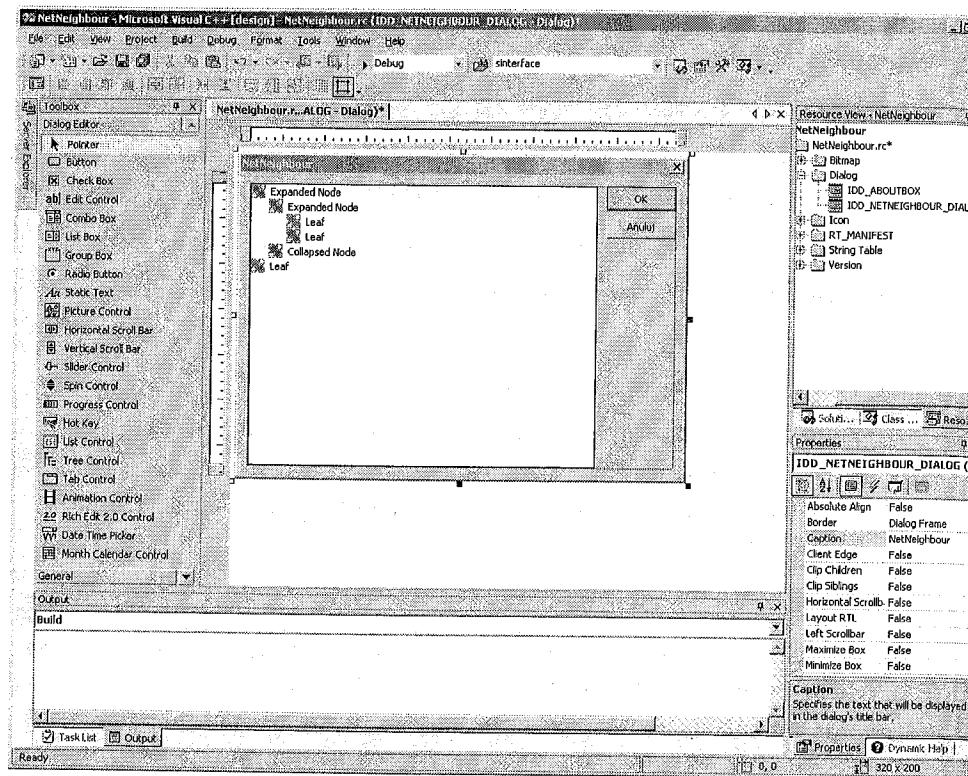
Kod źródłowy tego przykładu znajduje się w podkatalogu \Przykłady\Rozdział 4\Network na dołączonej do książki płytcie CD-ROM

4.3. Struktura otoczenia sieciowego

Aby sprawdzić, jakie komputery są przyłączone do sieci, należy zainstallować do otoczenia sieciowego. Jak zrobić to nie z poziomu pulpitu czy Eksploratora, ale z poziomu własnego programu? To całkiem proste. Gotowy program wyświetlający hierarchiczną listę wszystkich komputerów w sieci wraz z udostępnianymi przez nie zasobami prezentowany jest poniżej.

Aby go utworzyć, otwórz w Visual C++ nowy projekt i nazwij go *NetNeighbour*. W kreatorze aplikacji, na zakładce typu aplikacji (*Application Type*), wybierz aplikację opartą na oknach dialogowych (*Dialog based*), a na zakładce cech zaawansowanych (*Advanced Features*) zaznacz gniazda systemu Windows (*Windows sockets*). Kliknij przycisk *Finish*, a kreator utworzy gotowy szkielet aplikacji sieciowej.

Zanim przystapiemy do właściwego programowania, powinniśmy dopasować do własnych potrzeb okno główne programu. W edytorze zasobów należy w tym celu rozwinąć gałąź *Dialog* i kliknąć dwukrotnie pozycję *IDD_NETNEIGHBOUR_DIALOG*. Potem w oknie roboczym należy rozciągnąć okno komponentu listy hierarchicznej (*Tree Control*) na jak największą powierzchnię okna głównego programu (patrz rysunek 4.2).



Rysunek 4.2. Dopasowanie rozmiarów komponentu Tree Control

Aby móc sterować tym komponentem, musimy skojarzyć go ze zmienną programu. W tym celu należy kliknąć go prawym przyciskiem myszy i z menu podręcznego wybrać pozycję *Add* i *Add Variable*, a w wyświetlonym oknie w polu nazwy zmiennej (*Variable name*) wpisać `m_NetTree` (nazwa może być inna, taką przyjęłem po prostu w swoim projekcie).

Jesteśmy już gotowi do omówienia kodu źródłowego programu. Otwórz plik *NetNeighbourDlg.cpp*. Znajdź w nim funkcję *OnInitDialog* wywoływaną w czasie inicjalizacji okna dialogowego. W tej funkcji trzeba utworzyć korzeń drzewa elementów sieciowych; robi się to następująco:

```
m_hNetworkRoot = InsertTreeItem(TVI_ROOT, NULL, "Moja sieć", DRIVE_RAMDISK + 1);
```

W powyższym wierszu zmiennej `m_hNetworkRoot` przypisywany jest wynik działania funkcji *InsertTreeItem*.

Następny fragment kodu będziemy wykorzystywać stosunkowo często, najlepiej więc unikać jego wielokrotnego powtarzania, pisząc osobną funkcję (jej kod prezentowany jest na listingu 4.2).

Listing 4.2. Dodawanie elementu do listy hierarchicznej elementów otoczenia sieciowego

```
HTREEITEM CNetNeighbourDlg::InsertTreeItem(HTREEITEM hParent, NETRESOURCE *const pNetResource, CString sText, int iImage)
{
    TVINSERTSTRUCT InsertStruct;
    InsertStruct.hParent      = hParent;
    InsertStruct.hInsertAfter = TVI_LAST;
    InsertStruct.itemex.mask  = TVIF_IMAGE | TVIF_TEXT | TVIF_CHILDREN | TVIF_PARAM;
    InsertStruct.itemex.pszText = sText.GetBuffer(sText.GetLength());
    InsertStruct.itemex.iImage = iImage;
    InsertStruct.itemex.cChildren = 1;
    InsertStruct.itemex.lParam = (LPARAM)pNetResource;
    sText.ReleaseBuffer();
    return m_NetTree.InsertItem( &InsertStruct );
}
```

Teraz program i wygląda odpowiednio, i odpowiednio zakorzenia hierarchię widoku otoczenia sieciowego. Kiedy jednak program się uruchomi i użytkownik kliknie element hierarchii, program powinien automatycznie odszukać wszystkie elementy otoczenia sieciowego związane z tymże elementem.

Aby to oprogramować powinniśmy uzupełnić kod o podprogram obsługi zdarzenia *ITEMEXPANDING*, w którym realizowalibyśmy przeszukiwanie otoczenia sieciowego. Przejdz do edytora zasobów i zaznacz komponent *Tree Control*. W oknie właściwości (*Properties*) kliknij przycisk *Control Events*. Zobaczyś listę wszystkich zdarzeń związanych z obsługą elementu interfejsu. Kliknij pole obok zdarzenia *TVN_ITEMEXPANDING* i z listy rozwijanej wybierz *Add* w celu dodania kodu obsługi zdarzenia. Kod ten prezentowany jest na listingu 4.3.

Listing 4.3. Kod obsługi zdarzenia *TVN_ITEMEXPANDING*

```
void CNetNeighbourDlg::OnTvnItemexpandingTree1(NMHDR *pNMHDR, LRESULT *pResult)
{
    LPNMTREEVIEW pNMTreeView = reinterpret_cast<LPNMTREEVIEW>(pNMHDR);
    // TODO: Add your control notification handler code here

    CWaitCursor CursorWaiting;
    ASSERT(pNMTreeView);
    ASSERT(pResult);

    if (pNMTreeView->action == 2)
    {
        CString sPath = GetItemPath(pNMTreeView->itemNew.hItem);

        if (!m_NetTree.GetChildItem(pNMTreeView->itemNew.hItem))
        {
            EnumNetwork(pNMTreeView->itemNew.hItem);
            if (m_NetTree.GetSelectedItem() != pNMTreeView->itemNew.hItem)
                m_NetTree.SelectItem(pNMTreeView->itemNew.hItem);
        }
    }
    *pResult = 0;
}
```

Powyższy kod przegląda elementy hierarchii należące do rozwiniętej gałęzi. Służy do tego wywołanie funkcji `EnumNetwork`, której kod prezentowany jest na listingu 4.4.

Listing 4.4. Funkcja przeszukująca zasoby sieciowe — `EnumNetwork`

```
bool CNetNeighbourDlg::EnumNetwork(HTREEITEM hParent)
{
    bool bGotChildren = false;

    NETRESOURCE *const pNetResource = (NETRESOURCE
*)m_NetTree.GetItemData(hParent);

    DWORD dwResult;
    HANDLE hEnum;
    DWORD cbBuffer = 16384;
    DWORD cEntries = 0xFFFFFFF;
    LPNETRESOURCE lpNetDrv;
    DWORD i;

    dwResult = WNetOpenEnum(pNetResource ? RESOURCE_GLOBALNET : RESOURCE_CONTEXT,
                           RESOURCETYPE_ANY, 0,
                           pNetResource ? pNetResource : NULL,
                           &hEnum );

    if (dwResult != NO_ERROR)
        return false;

    do
    {
        lpNetDrv = (LPNETRESOURCE) GlobalAlloc(GPTR, cbBuffer);
        dwResult = WNetEnumResource(hEnum, &cEntries, lpNetDrv, &cbBuffer);
        if (dwResult == NO_ERROR)
        {
            for(i = 0; i<cEntries; i++)
            {
                CString sNameRemote = lpNetDrv[i].lpRemoteName;
                int nType = 9;
                if (sNameRemote.IsEmpty())
                {
                    sNameRemote = lpNetDrv[i].lpComment;
                    nType = 8;
                }
                if (sNameRemote.GetLength() > 0 && sNameRemote[0] == _T('\\'))
                    sNameRemote = sNameRemote.Mid(1);
                if (sNameRemote.GetLength() > 0 && sNameRemote[0] == _T('\\'))
                    sNameRemote = sNameRemote.Mid(1);

                if (lpNetDrv[i].dwDisplayType == RESOURCEDISPLAYTYPE_SHARE)
                {
                    int nPos = sNameRemote.Find(_T('\\'));
                    if(nPos >= 0)
                        sNameRemote = sNameRemote.Mid(nPos+1);
                    InsertTreeItem(hParent, NULL, sNameRemote, DRIVE_NO_ROOT_DIR);
                }
                else
                {

```

Rozdział 4. ♦ Sieci komputerowe

```
NETRESOURCE* pResource = new NETRESOURCE;
ASSERT(pResource);
*pResource = lpNetDrv[i];
pResource->lpLocalName = MakeDynamic(pResource->lpLocalName);
pResource->lpRemoteName = MakeDynamic(pResource->lpRemoteName);
pResource->lpComment = MakeDynamic(pResource->lpComment);
pResource->lpProvider = MakeDynamic(pResource->lpProvider);
InsertTreeItem(hParent, pResource, sNameRemote,
               pResource->dwDisplayType + 7);

}
bGotChildren = true;
}

}
GlobalFree((HGLOBAL)lpNetDrv);
if (dwResult != ERROR_NO_MORE_ITEMS)
    break;
}
while (dwResult != ERROR_NO_MORE_ITEMS);

WNetCloseEnum(hEnum);
return bGotChildren;
}
```

Logika kryjąca się za tym pozornie skomplikowanym kodem jest całkiem prosta. Na początku inicjowane jest wyliczanie zasobów sieciowych. Służy do tego funkcja `WNetOpenEnum`:

```
DWORD WNetOpenEnum(
    DWORD dwScope,           // Zasięg wyliczania
    DWORD dwType,            // Typ zasobów objęty wyliczaniem
    DWORD dwUsage,           // Kategoria używalności zasobów do wyliczania
    LPNETRESOURCE lpNetResource, // Wskaźnik struktury zasobu
    LPHANDLE phEnum          // Wskaźnik uchwytu bufora wyliczania
);
```

Funkcja uruchamia wyliczanie urządzeń i zasobów sieciowych dostępnych w sieci lokalnej. Jej parametry mają następujące znaczenie:

- ◆ `dwScope` — zasięg wyliczania. Może być kombinacją wartości:
 - ◆ `RESOURCE_GLOBALNET` — wszystkie zasoby sieciowe,
 - ◆ `RESOURCE_CONNECTED` — podłączone zasoby sieciowe,
 - ◆ `RESOURCE_REMEMBERED` — zapamiętane zasoby sieciowe.
- ◆ `dwType` — typ zasobów objętych wyliczaniem — kombinacja poniższych wartości:
 - ◆ `RESOURCETYPE_ANY` — dowolne zasoby sieci komputerowej,
 - ◆ `RESOURCETYPE_DISK` — zasoby dyskowe sieci.
 - ◆ `RESOURCETYPE_PRINT` — drukarki sieciowe.
- ◆ `dwUsage` — kategoria zastosowań zasobów objętych wyliczaniem:
 - ◆ 0 — wszystkie zasoby sieci,

- ◆ RESOURCEUSAGE_CONNECTABLE — zasoby, z którymi można się połączyć,
- ◆ RESOURCEUSAGE_CONTAINER — zasoby kontenerowe.
- ◆ *lpNetresource* — wskaźnik struktury NETRESOURCE. Jeśli parametr ten ma wartość 0 (NULL), wyliczanie rozpoczyna się od głównego elementu sieciowej hierarchii zasobów. Przy takiej wartości wyliczanie rozpocznie się od pierwszego zasobu sieciowego. Następnie można zwrócić tym parametrem wskaźnik przekazać w kolejnym wywołaniu, a wyliczanie rozpocznie się od wskazanego zasobu.
- ◆ *lphEnum* — wskaźnik, który zostanie wykorzystany w wywołaniu *WNetEnumResource*.

Przyjrzyjmy się strukturze NETRESOURCE.

```
typedef struct NETRESOURCE {
    DWORD dwScope;
    DWORD dwType;
    DWORD dwDisplayType;
    DWORD dwUsage;
    LPTSTR lpLocalName;
    LPTSTR lpRemoteName;
    LPTSTR lpComment;
    LPTSTR lpProvider;
} NETRESOURCE;
```

Wiemy już, jakie znaczenie przypisywać polom *dwScope*, *dwType* i *dwUsage*. Resztę elementów należy interpretować następująco:

- ◆ *dwDisplayType* — sposób, w jaki zasób jest wyświetlany:
 - ◆ RESOURCEDISPLAYTYPE_DOMAIN — domena,
 - ◆ RESOURCEDISPLAYTYPE_GENERIC — brak wartości,
 - ◆ RESOURCEDISPLAYTYPE_SERVER — serwer,
 - ◆ RESOURCEDISPLAYTYPE_SHARE — zasób współużytkowany.
- ◆ *lpLocalName* — nazwa lokalna zasobu.
- ◆ *lpRemoteName* — nazwa zdalna zasobu.
- ◆ *lpComment* — komentarz.
- ◆ *lpProvider* — wskaźnik opisu udostępniającego zasób (jeśli pole ma wartość 0, udostępniający jest nieznany).

Weźmy na warsztat kolejną funkcję:

```
DWORD WNetEnumResource(
    HANDLE hEnum,           // Uchwyt bufora wyliczania
    LPDWORD lpcCount,        // Wskaźnik liczby wpisów do wyliczenia
    LPVOID lpBuffer,          // Wskaźnik bufora wyników wyliczenia
    LPDWORD lpBufferSize       // Wskaźnik zmiennej rozmiaru bufora
);
```

Funkcja *WNetEnumResource* wymaga przekazania następujących parametrów:

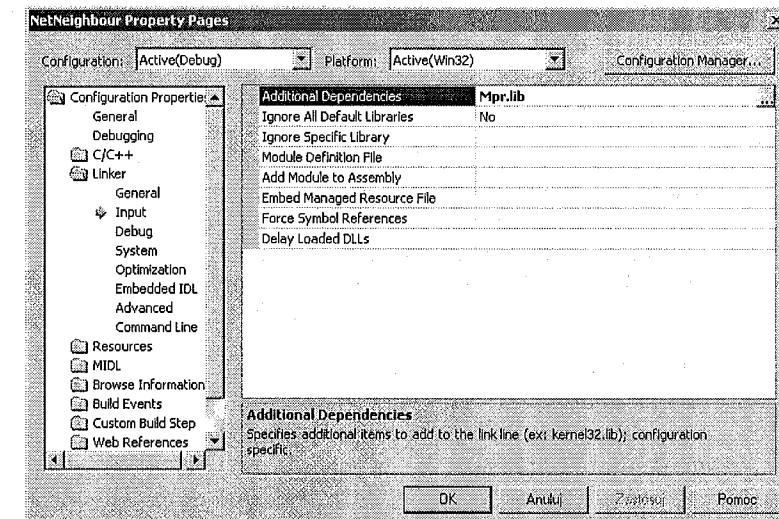
- ◆ *hEnum* — uchwytu wartości zwróconej przez funkcję *WNetOpenEnum*.
- ◆ *lpcCount* — limitu liczby zwróconych wartości. Można tu podać nawet i 2000. Jeśli przekazana zostanie wartość 0xFFFFFFFF, lista wynikowa obejmować będzie wszystkie wyliczone zasoby. Kiedy funkcja zakończy działanie, zwróci za pośrednictwem tego parametru faktyczną liczbę znalezionych zasobów.
- ◆ *lpBuffer* — wskaźnik bufora wyników.
- ◆ *lpBufferSize* — wskaźnik rozmiaru bufora wyników.

Po zakończeniu wyliczania zasobów sieciowych należy wywołać funkcję *WNetCloseEnum*, aby zakończyć wyliczanie zainicjowane wywołaniem *WNetOpenEnum*. Jedynym parametrem wywołania *WNetCloseEnum* jest uchwyt wyliczania zwrócony przez *WNetOpenEnum*.

To byłoby na tyle w kwestii wyszukiwania współużytkowanych zasobów w sieci. Jeszcze tylko jedno: funkcja *WNetOpenEnum* i związane z nią struktury i funkcje implementowane są w bibliotece *mpr.lib*, która nie jest domyślnie konsolidowana z projektami Visual C++. Aby uniknąć błędów konsolidacji, trzeba jawnie wymusićłączenie tej biblioteki do projektu. W tym celu należy kliknąć prawym przyciskiem myszy w oknie przeglądarki projektu (*Solution Explorer*) nazwę projektu i z menu podręcznego wybrać polecenie *Properties*. Na ekranie wyświetcone zostanie okno dialogowe właściwości projektu. Należy w nim odnaleźć pozycję *Configuration Properties/Linker/Input* i w polu tekstowym opisanym jako *Additional Libraries* wpisać *mpr.lib*. (jak na rysunku 4.3).

Rysunek 4.3.

Dodawanie do projektu biblioteki zawierającej implementację *WNetOpenEnum*



Kod źródłowy tego przykładu znajduje się w podkatalogu \Przykłady\Rozdział4\NetNeighbour na dołączonej do książki płytcie CD-ROM.

4.4. Obsługa sieci za pośrednictwem obiektów MFC

Aby z poziomu własnych programów wygodnie posługiwać się siecią, można wykorzystać obiektowe elementy środowiska programistycznego Visual C++. Obiekty ułatwiają programowanie, ukrywając niektóre szczegóły związane z implementacją protokołów sieciowych.

Zastosowanie tych projektów powoduje, niestety, znaczny rozrost programu, nie możemy bowiem dłużej korzystać z aplikacji typu *Win32 Project* — zamiast tego każdy projekt musi być inicjowany jako *MFC Application*. Na początku możemy jednak pogodzić się ze zwiększeniem objętości programu, jeśli pomoże to w szybkiej nauce tworzenia programów sieciowych. Będę za to konsekwentnie przedstawiał kolejne funkcje WinAPI, którymi z czasem Czytelnik będzie mógł zastąpić obiekty Visual C++ i pogodzić programowanie sieciowe z małymi rozmiarami plików wykonywalnych.

Biblioteka MFC zawiera bardzo wygodną klasę obiektów obsługi gniazd sieciowych — CSocket. Jej klasą bazową jest CAsyncSocket. Cóż to oznacza? Otóż obiekt CAsyncSocket obsługuje sieć w sposób asynchroniczny (nieblokujący). Po wysłaniu pakietu nie oczekuje na potwierdzenie jego dostarczenia — od razu kontynuuje własne, dalsze zadania. O efekcie zainicjowanej transmisji aplikacja dowiaduje się za pośrednictwem zdarzeń, których generowaniem zajmuje się obiekt. Programista aplikacji musi jedynie oprogramować obsługę tych zdarzeń.

W trybie synchronicznym za każdym razem, kiedy użytkownik, bądź aplikacja, inicjuje wysyłanie pakietu przez sieć, inicjując np. połączenie z serwerem, wykonanie aplikacji jest zawieszane na czas zakończenia operacji (np. do momentu uzyskania połączenia). Nie pozwala to na optymalne wykorzystanie czasu procesora.

Obiekt typu CSocket jest obiektem potomnym obiektu CAsyncSocket, co oznacza, że dziedziczy po tym ostatnim komplet właściwości, zdarzeń i metod (funkcji składowych obiektu). Jego działanie opiera się na architekturze klient-serwer — jeden z komunikujących się ze sobą obiektów jest „serwerem” przetwarzającym żądania inicjowane przez „klienta”. Aplikacje, które mają transmitować dane, powinny więc zawierać dwa obiekty: CServerSocket (obiekt gniazda serwera) i CCClientSocket (obiekt gniazda klienta).

Obiekt CServerSocket jest bardzo podobny do CCClientSocket. Serwer oczekuje na nawiązanie połączenia, nasłuchując pod określonym numerem portu. Kiedy klient nawiąże połączenie, po stronie serwera tworzona jest gniazdo CCClientSocket służące do odbierania danych przez serwer.

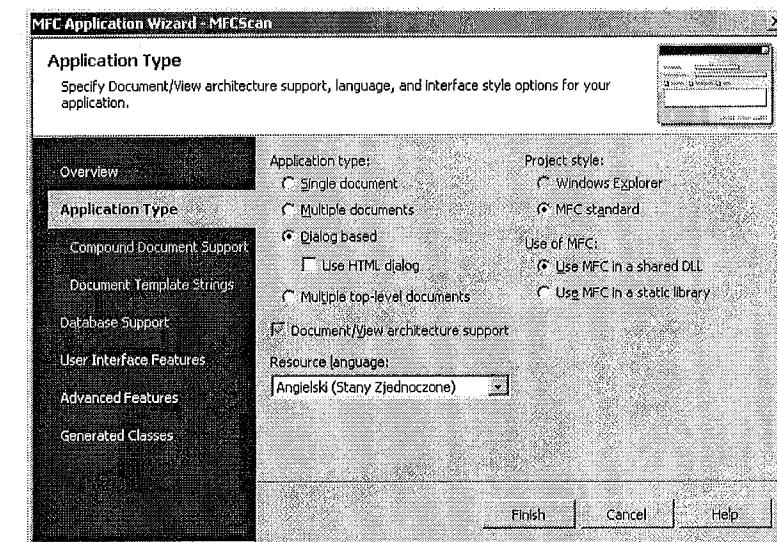
Aby sprawdzić, jak całość działa w sieci, napiszemy aplikację skanującą wybrany komputer w poszukiwaniu otwartych portów i nasłuchujących serwerów (będzie to tzw. skaner portów). Jak działa taki program? Aby odnaleźć otwarte do nasłuchu porty, wystarczy spróbować się z nimi połączyć. Jeśli próba będzie skuteczna, oznacza to, że na węźle zdalnym działa aplikacja, która otwarła port.

Niektórzy programiści sądzą, że jeśli program serwera wymaga autoryzacji połączenia, to nie można połączyć się z jego portem nasłuchu. To nie tak, ponieważ autoryzacja następuje zawsze po nawiązaniu połączenia. Dlatego można próbować łączyć się z dowolnymi portami pod warunkiem, że pomiędzy naszym a skanowanym komputerem nie działa żaden system ochrony (jak zapora sieciowa).

Przejedźmy od słów do czynów. Utwórzmy w Visual C++ nowy projekt, wybierając jako typ projektu *MFC Application*. Nazwijmy projekt *MFCScan*. W kreatorze projektu ustaw następujące parametry:

- ♦ Na zakładce *Application Type* zaznacz *Dialog based* (rysunek 4.4).

Rysunek 4.4.
Okno dialogowe typu aplikacji



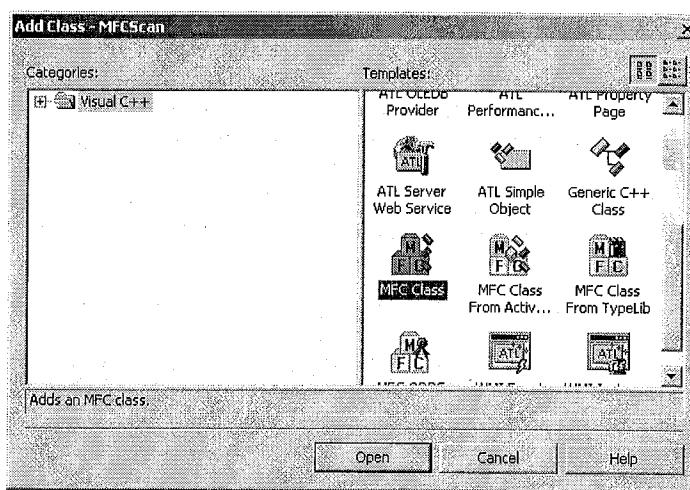
- ♦ Na zakładce *Advanced Features* zaznacz *Windows Sockets*.

Naciśnij przycisk *Finish* — kreator utworzy niezbędne pliki projektu.

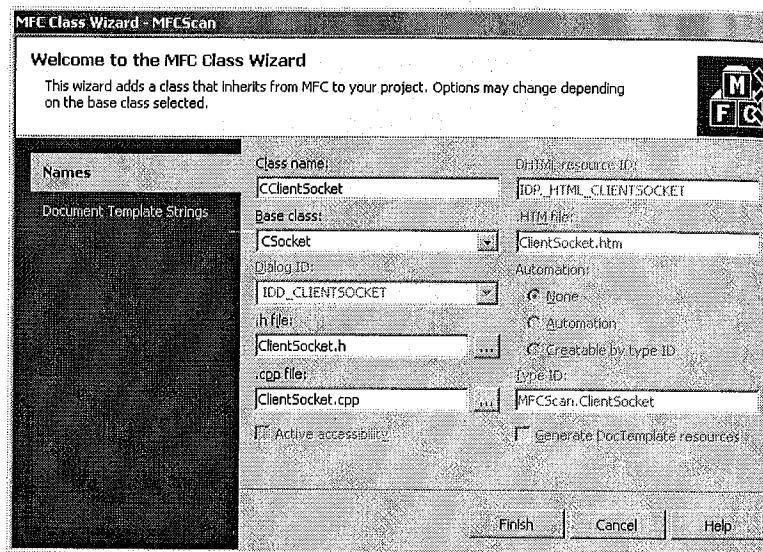
Szkielek aplikacji jest gotowy, brakuje w nim jednak obiektów, które obsługująby połączenia sieciowe. Możemy się bez nich chwilowo obejść, ale gdyby w przyszłości zaszła potrzeba rozbudowania skanera, warto byłoby zadbać już teraz o taką możliwość. Kliknij prawym przyciskiem myszy nazwę projektu w oknie *Solution Explorer* i z menu podręcznego wybierz polecenia *Add, Add Class...*. Otwarte zostanie okno dialogowe uzupełnienia projektu o nową klasę (rysunek 4.5). Zaznacz element *MFC Class* i kliknij przycisk *Open*.

Jeśli wykonasz wszystko poprawnie, zobaczysz okno takie jak na rysunku 4.6. Możesz w nim określić nazwę klasy dodawanej do projektu wraz z jej klasą bazową, z której Twoja klasa ma zostać wyprowadzona. W tym celu w polu *Base Class* wybierz *CSocket*, a w polu *Class Name* wpisz *CCClientSocket*.

Rysunek 4.5.
Okno dialogowe uzupełniania projektu



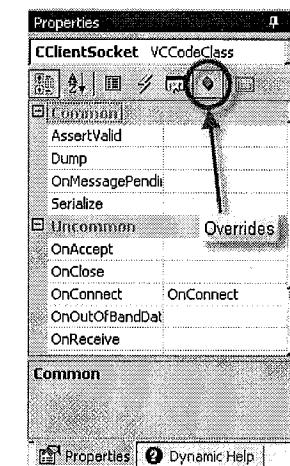
Rysunek 4.6.
Okno ustawień klasy



Do projektu dodane zostaną dwa nowe pliki: *ClientSocket.cpp* oraz *ClientSocket.h*. Ich kod służy do kontrolowania połączenia. Przyjrzymy się mu bliżej.

Otwórz plik *ClientSocket.cpp* i kliknij w oknie właściwości przycisk *Overrides*. W oknie pojawią się metody i zdarzenia (rysunek 4.7), których obsługę można przesłonić własnym kodem po to, aby obiekt klasy działał zgodnie z naszymi oczekiwaniami. Przesłonięcie polega na kliknięciu listy rozwijanej i wybraniu z niej pożąanej metody czy zdarzenia i wybraniu z menu polecenia *Add <nazwa metody>*. Na razie nie będziemy tu jednak niczego zmieniać.

Rysunek 4.7.
Okno właściwości klasy



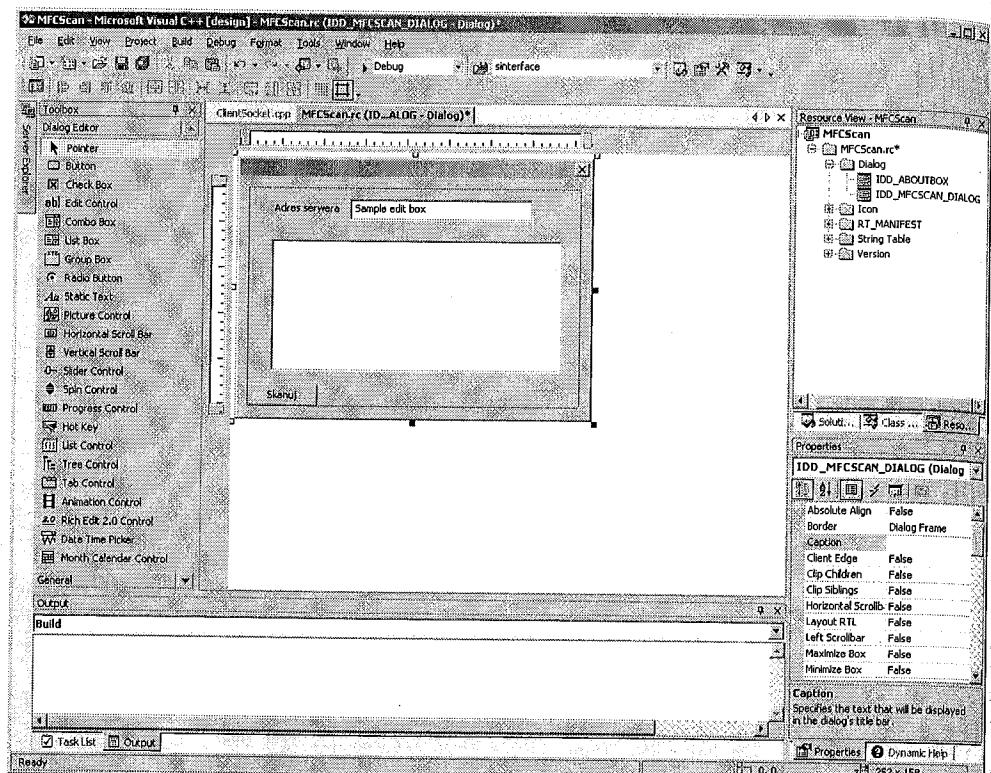
Teraz otwórz przeglądarkę zasobów i znajdź pośród nich gałąź okien dialogowych, a w niej pozycję *IDD_MFCSCAN_DIALOG*. Kliknij ją dwukrotnie. W edytorze zasobu usuń z okna dialogowego przyciski *OK* i *Cancel* i umieść w oknie następujące komponenty:

- ◆ Komponent etykiety (*Static Text*) z napisem *Adres serwera*.
- ◆ Komponent pola tekstowego (*Edit Control*) — służący do wpisywania adresu serwera do przeskanowania.
- ◆ Komponent okna listy (*List Box*) — do przechowywania listy otwartych portów.
- ◆ Komponent przycisku (*Button*) z napisem *Skanuj* — inicjującego proces skanowania wskazanego komputera.

Powinieneś uzyskać okno dialogowe mniej więcej takie jak na rysunku 4.8.

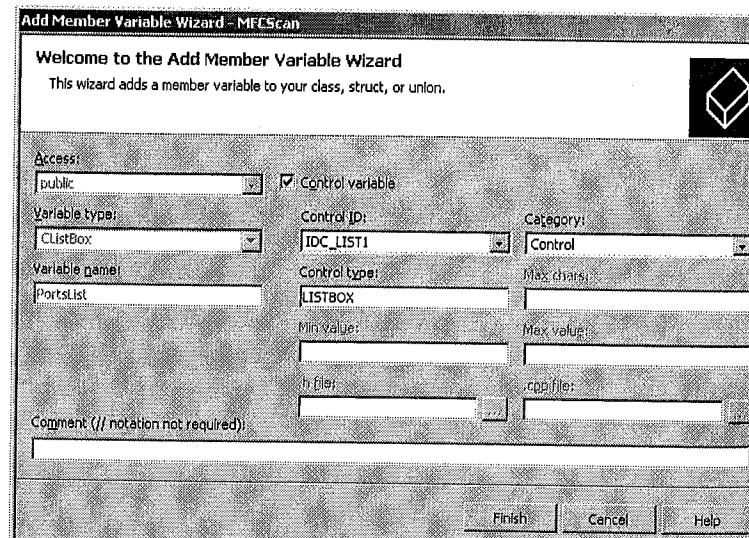
Teraz przypisz zmienne do komponentów. W tym celu kliknij komponent *List Box* prawym przyciskiem myszy i z menu podręcznego wybierz polecenie *Add Variable*. W wyświetlonym oknie (rysunek 4.9) wprowadź w polu *Variable Name* nazwę zmiennej. Przyjmijmy, że będzie to nazwa *PortsList*.

Czynności przygotowawcze mamy już za sobą. Możemy przystąpić do pisania właściwego kodu skanera portów. W tym celu powinniśmy zdefiniować kod obsługi zdarzenia generowanego w reakcji na kliknięcie przez użytkownika przycisku *Skanuj*. W tym celu należy kliknąć komponent przycisku prawym przyciskiem myszy i z menu podręcznego wybrać polecenie *Add Event Handler*. Otworzy się okno kreatora procedury obsługi zdarzenia (*Add Event Handler Wizard*, rysunek 4.10). Nie zmieniaj żadnych ustawień i kliknij przycisk *Add and Edit*.

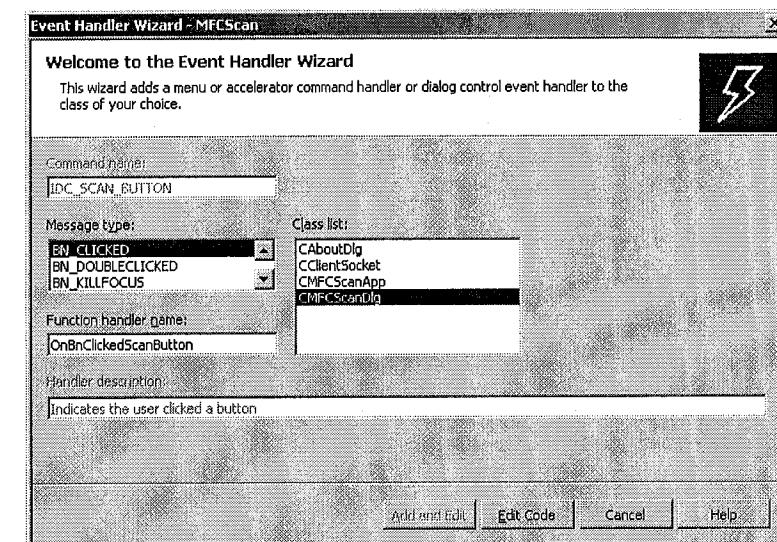


Rysunek 4.8. Głównie okno przyszłej aplikacji

Rysunek 4.9.
Okno dialogowe
przypisania zmiennej
do komponentu
interfejsu



Rysunek 4.10.
Kreator procedury
obsługi zdarzenia



Kreator utworzy szablon kodu obsługi zdarzenia. Uzupełnij go kodem z listingu 4.5.

Listing 4.5. Kod skanera portów

```
void CMFCSanDlg::OnBnClickedScanButton()
{
    // TODO: Add your control notification handler code here
    CCClientSocket *pSocket;
    CString      ip;
    CString      messtr;
    int         port;

    pSocket = new CCClientSocket();
    pSocket->Create();

    GetDlgItemText(IDC_EDIT1, ip);
    port = 1;
    while (port < 100)
    {
        if(pSocket->Connect(ip, port))
        {
            messtr.Format("Port=%d otwarty", port);
            PortsList.AddString(messtr);
            pSocket->Close();
            pSocket->Create();
        }
        port++;
    }
}
```

Przeanalizujmy powyższy kod. Deklaruje on na początku zmienną pSocket typu CCClientSocket. Będziemy ją wykorzystywać w roli obiektu obsługującego sieć opartą na protokołach TCP/IP. Zanim zaczniemy, musimy jednak przydzielić dla obiektu odpowiednią ilość pamięci i zainicjalizować obiekt. Służą do tego dwa wiersze kodu:

```
pSocket = new CClientSocket();
pSocket->Create();
```

Teraz należałoby się dowiedzieć, jaki adres wpisał użytkownik w polu edycyjnym przed kliknięciem przycisku *Skanuj*. Ciąg zawarty w tym polu zwraca funkcja *GetDlgItemText*. Wymaga ona przekazania dwóch parametrów: identyfikatora komponentu i zmiennej, w której zapisana ma być jego zawartość.

Moglibyśmy na potrzeby funkcji *GetDlgItemText* utworzyć specjalną zmienną — wystarczyłoby kliknąć komponent w edytorze zasobów i utworzyć zmienną. Ale ponieważ wartość umieszczoną w polu tekstowym wykorzystamy tylko raz, nie musimy tworzyć żadnej zmiennej.

Następnie inicjalizujemy zmienną port wartością 1. To numer portu, od którego rozpoczniemy skanowanie węzła zdalnego. Zaraz potem inicjowana jest pętla wykonywana do momentu osiągnięcia przez zmienną port wartości 100.

Wewnątrz pętli próbujemy nawiązać połączenie z węzłem zdalnym (serwerem):

```
pSocket->Connect(ip, port);
```

Ten pojedynczy wiersz kodu inicjuje procedurę nawiązywania połączenia za pośrednictwem obiektu wskazywanego przez zmienną *pSocket*. Wywoływana na rzecz tego obiektu metoda przyjmuje dwa parametry: adres komputera, z którym obiekt ma się połączyć, oraz numer portu. Jeśli uda się nawiązać połączenie, wywołanie da wartość zero, a w oknie listy *PortsList* pojawi się wpis z informacją o otwartym porcie. Bardzo ważny moment to zamknięcie połączenia i ponowna inicjalizacja obiektu gniazda. Jeżeli te czynności zostały pominięte, następne próby połączeń byłyby bezskuteczne, a program nie działałby tak jak powinien. Zamknięcie połączenia i ponowna inicjalizacja obiektu odbywa się za pośrednictwem metod *Close* i *Create*:

```
pSocket->Close();
pSocket->Create();
```

Pod koniec pętli wartość zmiennej port jest zwiększana o jeden, tak aby w kolejnym przebiegu pętli sondowany był kolejny port serwera.

Możemy już przystąpić do komplikacji programu. Aby uniknąć błędów, powinniśmy jednak na początku pliku, tam, gdzie wymieniane są moduły włączane do programu, umieścić następujący wiersz:

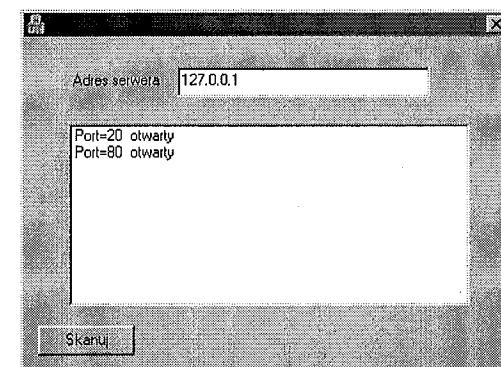
```
#include "ClientSocket.h"
```

Chodzi o to, że w programie wykorzystywany jest obiekt klasy *CClientSocket*, którego definicja znajduje się w pliku nagłówkowym *CCliensocket.h* — bez włączenia tego pliku do programu nie da się skompilować projektu.

Efekt działania programu prezentowany jest na rysunku 4.11. Aby uzyskać taki widok na swoim ekranie, musisz jedynie uruchomić program, wpisać w polu adresu serwera 127.0.0.1 i nacisnąć przycisk *Skanuj* — program rozpoczęcie sondowanie portów 0 do 99 wskazanego komputera. Dlaczego zakres portów skanowania jest tak mocno ograniczony? Cóż, skanowanie tysięcy portów w systemie Windows mogłoby zająć dobrych kilka minut, najlepiej więc przykład (który i tak ma przecież charakter poglądowy) ograniczyć do węższego zakresu portów.

Rysunek 4.11.

Efekt skanowania portów



Niedługo przyjrzymy się ulepszonemu programowi skanowania portów. Co do tego przykładowu, to ma on wartość czysto treningową. Pozwolił jednak (mam nadzieję) zrozumieć algorytm skanowania portów wykorzystywane w tego rodzaju narzędziach. Nawiąsem mówiąc, jeśli nawet masz doświadczenie programistyczne w Visual C++ i potrafisz tworzyć aplikacje wielowątkowe, nie zalecałbym tworzenia osobnych wątków do skanowania pojedynczych portów. Być może przyspieszyłoby to nawet odrobinę program, ale obciążałoby mocno system operacyjny. Jeśli czekasz na ulepszenia, poczekaj na następną wersję prezentowanego programu.

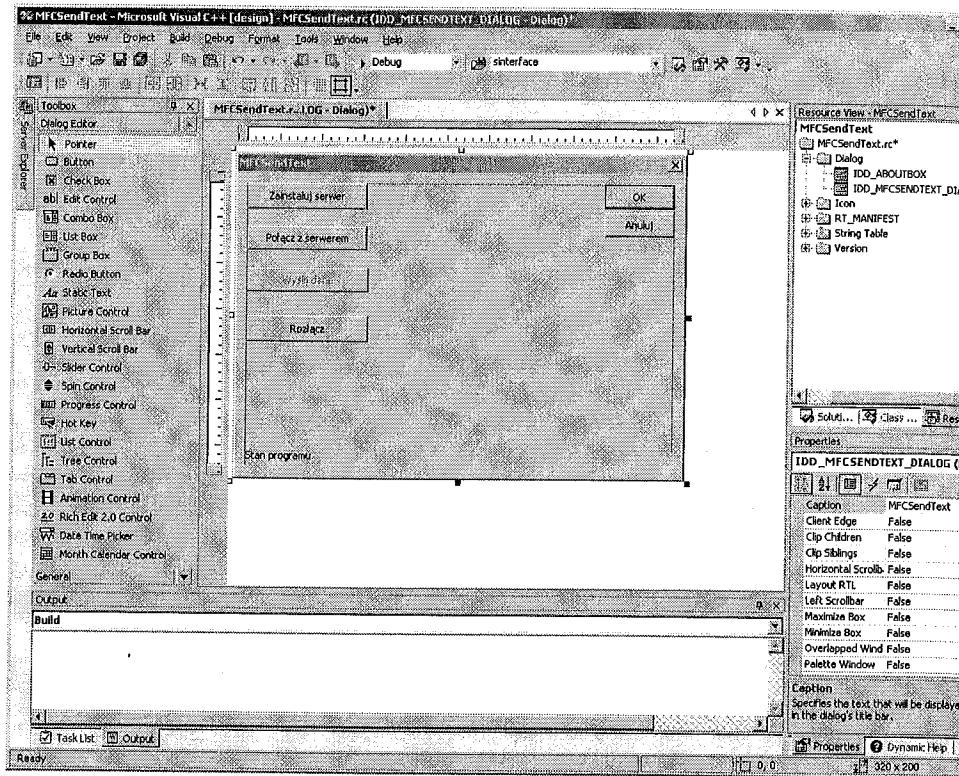
 Kod źródłowy tego przykładu znajduje się w podkatalogu \Przykłady\Rozdział4\Scan na dołączonej do książki płytcie CD-ROM.

4.5. Transmisja danych w sieci za pośrednictwem obiektu CSocket

Jak już wspomniałem, obsługa gniazd opiera się na technologii klient-serwer. Uruchamiany w systemie serwer otwiera do nasłuchu port o wybranym numerze i oczekuje na nawiązanie połączenia przez klienta. Kiedy ten połączy się z odpowiednim portem, może rozpocząć wymianę danych z serwerem.

Przyjrzyjmy się takiej transmisji danych. Zaczniemy od utworzenia projektu typu *MFC Application* i opatrzenia go nazwą *MFCSendText*. Jeszcze w oknie kreatora aplikacji wypadałoby zmienić parametry projektu, podobnie jak w poprzednim przykładzie (patrz podrozdział 4.4). Do projektu dodaj dwie klasy wyprowadzone z klas *CSocket* — jedną do obsługi klienta i jedną do obsługi serwera. Nazwij je (odpowiednio) *CClientSocket* oraz *CServerSocket*. Jak widać, z pojedynczej klasy bazowej można wyprowadzać wiele klas pochodnych.

Pora na dostosowanie głównego okna aplikacji (rysunek 4.12). W tym celu otwórz element *IDD_MFCSENDDTEXT_DIALOG* w edytorze zasobów i umieść na formularzu okna aplikacji cztery przyciski: *Zainstaluj serwer* (*IDC_BUTTON1*), *Połącz z serwerem* (*IDC_BUTTON2*),



Rysunek 4.12. Edytor głównego okna aplikacji

Wyślij dane (IDC_BUTTON3) i Rozłącz (IDC_BUTTON4). U dołu okna umieść pole etykiety (Static Control, IDC_STATIC), które wykorzystamy do wyświetlania komunikatów programu.

Przypisz zmienną do przycisku *Wyślij dane* (kliknij przycisk prawym przyciskiem myszy i z menu podręcznego wybierz polecenie *Add Variable*; w oknie dodawania zmiennej podaj jako nazwę zmiennej *m_SendButton*).

Przejdźmy do programowania. Na pierwszy ogień pójdzie plik nagłówkowy *ServerSocket.h*, który zawiera deklarację klasy *CServerSocket* (patrz listing 4.6).

Listing 4.6. Zawartość pliku ServerSocket.h

```
#pragma once

#include "MFCSendTextDlg.h"

// CServerSocket command target

class CServerSocket : public CSocket
{
public:
    CServerSocket(CMFCSendTextDlg* Dlg);
```

```
virtual ~CServerSocket();
virtual void OnAccept(int nErrorCode);
virtual void OnConnect(int nErrorCode);

protected:
    CMFCSendTextDlg* m_Dlg;

public:
    virtual void OnClose(int nErrorCode);
};
```

Pierwszą rzeczą, którą trzeba było zmienić, był konstruktor klasy. W obecnej wersji *CServerSocket* przyjmuje jeden parametr (*Dlg*) typu *CMFCSendTextDlg*. Parametr ten będzie później wykorzystywany do przekazywania wskaźnika okna głównego programu, tak aby można się do niej było odwoływać w klasie *CServerSocket*. W sekcji składowych zabezpieczonych klasy (*protected*) zadeklarowana została zmenna wskaźnika okna głównego.

Poza tym uzupełniłem klasę o metodę *OnAccept*. Metoda ta będzie wywoływana, kiedy serwer przyjmie żądanie nawiązania połączenia.

Na listingu 4.7 prezentowana jest zawartość pliku *ServerSocket.cpp*. Zawiera on implementację klasy *CServerSocket*.

Listing 4.7. Implementacja klasy CServerSocket

```
// ServerSocket.cpp : implementation file
//

#include "stdafx.h"
#include "MFCSendText.h"
#include "ServerSocket.h"

// CServerSocket

CServerSocket::CServerSocket(CMFCSendTextDlg* Dlg)
{
    m_Dlg = Dlg;
}

CServerSocket::~CServerSocket()
{
}

// CServerSocket member functions
void CServerSocket::OnAccept(int nErrorCode)
{
    // TODO: Add your specialized code here and/or call the base class
    AfxMessageBox("Przyjęto nowe połączenie");
    m_Dlg->AddConnection();

    CSocket::OnAccept(nErrorCode);
}

void CServerSocket::OnConnect(int nErrorCode)
{
```

```

// TODO: Add your specialized code here and/or call the base class
CSocket::OnConnect(nErrorCode);
}

void CServerSocket::OnClose(int nErrorCode)
{
    // TODO: Add your specialized code here and/or call the base class
    CSocket::OnClose(nErrorCode);
}

```

Konstruktor obiektu klasy CServerSocket zapisuje po prostu wartość przekazaną parametrem wywołania w składowej `m_Dlg`.

Metoda `OnAccept` jest wywoływana za każdym razem, kiedy serwer zaakceptuje żądanie nawiązania połączenia zainicjowane przez klienta. W ramach obsługi tego zdarzenia wywoływana jest najpierw funkcja `AfxMessageBox` wyświetlająca komunikat o przyjęciu nowego połączenia. Następne wywołanie to metoda `AddConnection` klasy okna wskazywanego zmienną `m_Dlg`.

W tym kodzie zakładam, że `m_Dlg` zawiera poprawne dane, tzn. wskazuje istniejący obiekt. Jeśli przewidujesz, że wartość tej zmiennej będzie się zmieniać, albo że obiekt, który wskazuje, może zostać przedwcześnie usunięty, powinieneś każdorazowo przed odwołaniem się do zmiennej w wywołaniu `AddConnection` sprawdzać poprawność jej wartości. To ważne, ponieważ w innym przypadku aplikacja może naruszyć stabilność systemu operacyjnego.

Spójrzmy teraz na deklarację klasy klienta `CCClientSocket`, czyli zawartość pliku `ClientSocket.h` (listing 4.8).

Listing 4.8. Plik `ClientSocket.h`

```

#pragma once

#include "MFCSendTextDlg.h"

// CCClientSocket command target

class CCClientSocket : public CSocket
{
public:
    CCClientSocket(CMFCSendTextDlg* Dlg);
    virtual ~CCClientSocket();
    virtual void OnReceive(int nErrorCode);
    virtual void OnClose(int nErrorCode);
protected:
    CMFCSendTextDlg* m_Dlg;
};

```

W tym pliku również zmieniony został konstruktor, tak aby można w nim zapisać w składowej obiektu wskaźnik obiektu okna programu, który jest wobec obiektu gniazda klienta obiektem nadziednym. Do tego służy zadeklarowana we wnętrzu klasy `CCClientSocket` zmienna `m_Dlg`.

Deklaracja klasy została też uzupełniona o dwie metody: `OnReceive` (wywoływaną w momencie odebrania pakietu danych) oraz `OnClose` (wywoływaną w reakcji na zdarzenie zamknięcia połączenia).

Spójrzmy teraz na implementację klasy, czyli zawartość pliku `ClientSocket.cpp` (listing 4.9).

Listing 4.9. Zawartość pliku `ClientSocket.cpp`

```

// ClientSocket.cpp : implementation file

#include "stdafx.h"
#include "MFCSendText.h"
#include "ClientSocket.h"

// CCClientSocket

CCClientSocket::CCClientSocket(CMFCSendTextDlg* Dlg)
{
    m_Dlg = Dlg;
}

CCClientSocket::~CCClientSocket()
{
}

void CCClientSocket::OnReceive(int nErrorCode)
{
    char recstr[1000];
    int r = Receive(recstr, 1000);
    recstr[r] = '\0';
    m_Dlg->SetDlgItemText(IDC_STATIC, recstr);

    CSocket::OnReceive(nErrorCode);
}

void CCClientSocket::OnClose(int nErrorCode)
{
    m_Dlg->m_SendButton.EnableWindow(FALSE);

    CSocket::OnClose(nErrorCode);
}

```

Tutaj najciekawsza jest metoda `OnReceive`. Jest ona wywoływana za każdym razem, kiedy obiekt gniazda klienta odbierze z sieci nowy pakiet danych. Aby odczytać ich treść, należy skorzystać z metody `Receive` obiektu. Wymaga ona przekazania dwóch parametrów:

- ◆ wskaźnika bufora, w którym dane mają zostać umieszczone (zmienna recstr),
- ◆ rozmiaru bufora.

Metoda zwraca liczbę bajtów odebranych z sieci. Wartość ta jest zachowywana w zmiennej r. Po powrocie z funkcji dane znajdują się w zmiennej recstr. Składnią wiemy, że to tekst — ciąg znaków. Ciąg taki powinien się jednak kończyć tzw. znakiem pustym (zerem). Stąd za ostatnim znakiem w buforze powinniśmy samodzielnie umieścić znak pusty:

```
recstr[r] = '\0';
```

Teraz można otrzymany tekst skopiować do komponentu etykiety komunikatów w oknie głównym programie. Realizuje to następny wiersz:

```
m_Dlg->SetDlgItemText(IDC_STATIC, recstr);
```

Metoda OnClose jest wywoływana w reakcji na zamknięcie połączenia. W ramach jej kodu czynimy przycisk *Wyślij dane* nieaktywnym — bez aktywnego połączenia użytkownik nie powinien móc wysyłać danych do serwera:

```
m_Dlg->m_SendButton.EnableWindow(FALSE);
```

Przedźmy do omówienia głównego modułu naszego programu — MFCSendTextDlg. Zaczniemy od pliku nagłówkowego (listing 4.10).

Listing 4.10. Plik nagłówkowy MFCSendTextDlg.h

```
// MFCSendTextDlg.h : header file

#pragma once
#include "afxwin.h"

class CServerSocket;
class CCClientSocket;

class CMFCSendTextDlg : public CDialog
{
// Construction
public:
    CMFCSendTextDlg(CWnd* pParent = NULL); // standard constructor

// Dialog Data
    enum { IDD = IDD_MFCSENDTEXT_DIALOG };

protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support

// Implementation
protected:
    HICON m_hIcon;
    CServerSocket* m_sSocket;
    CCClientSocket* m_cSocket;
    CCClientSocket* m_scSocket;
```

```
// Generated message map functions
virtual BOOL OnInitDialog();
afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
DECLARE_MESSAGE_MAP()

public:
    afx_msg void OnBnClickedButton1();
    afx_msg void OnBnClickedButton2();
    afx_msg void OnBnClickedButton3();
    CButton m_SendButton;
    afx_msg void OnBnClickedButton4();
    void AddConnection();
};
```

W tym pliku zadeklarowaliśmy dodatkowo trzy składowe w sekcji protected:

- ◆ m_sSocket — wskaźnik obiektu klasy CServerSocket,
- ◆ m_cSocket i m_scSocket — wskaźniki obiektów klasy CCClientSocket.

W sekcji składowych publicznych wylądowała zaś dodatkowa metoda, AddConnection.

Pora na zdefiniowanie kodu obsługi zdarzeń dla wszystkich przycisków okna głównego. Jak się do tego zabrać? Otóż należy kliknąć w edytorze zasobów każdy z przycisków prawym przyciskiem myszy i z menu podręcznego wybrać polecenie *Add Event Handler*. Przyjrzymy się z osobna wszystkim czterem procedurom obsługi przycisków.

Dla przycisku *Zainstaluj serwer* odpowiedni będzie następujący kod:

```
void CMFCSendTextDlg::OnBnClickedButton1()
{
    // TODO: Add your control notification handler code here
    m_sSocket = new CServerSocket(this);
    m_sSocket->Create(22345);
    m_sSocket->Listen();
    SetDlgItemText(IDC_STATIC, "Serwer zainstalowany");
}
```

Obsługa przycisku polega na utworzeniu obiektu gniazda serwera i rozpoczęciu nasłuchu (oczekiwania na połączenie) na porcie o numerze 22345. Po kolej: w pierwszym wierszu mamy inicjalizację zmiennej m_sSocket. Jest ona obiektem typu CServerSocket, więc parametrem wywołania konstruktora obiektu jest wskaźnik obiektu bieżącego (obiektu okna głównego programu będącego wobec obiektu gniazda obiektem nadziednym — jego „rodzicem”). Wskaźnik te jest wyrażany jako this.

W dalszej kolejności wywoływana jest metoda Create obiektu gniazda. Jej jedynym parametrem jest numer portu nasłuchu. Sam nasłuch jest inicjowany wywołaniem metody Listen.

Po uruchomieniu nasłuchu wysyłany jest jeszcze komunikat o stanie serwera, kierowany do odpowiedniego komponentu okna.

Kod obsługi zdarzenia kliknięcia przycisku *Połącz z serwerem* powinien wyglądać następująco:

```
void CMFCSendTextDlg::OnBnClickedButton2()
{
    // TODO: Add your control notification handler code here
    m_cSocket = new CCClientSocket(this);
    m_cSocket->Create();
    if (m_cSocket->Connect("127.0.0.1", 22345))
        m_SendButton.EnableWindow(TRUE);
}
```

W pierwszym wierszu tworzony jest obiekt zmiennej `m_cSocket`. W kolejnym jest on inicjalizowany. Zaraz potem można już wykorzystać go do nawiązania połączenia z serwrem — służy do tego metoda `Connect` obiektu. Metoda ta ma kilka wersji różniących się liczbą i typami parametrów wywołania. W naszym przykładzie wywoływana jest wersja dwuparametrowa wymagająca przekazania:

- ◆ adresu IP serwera (w postaci ciągu znaków),
- ◆ numeru portu.

Jeśli uda się nawiązać połączenie, metoda zwróci wartość niezerową. Po sprawdzeniu wartości zwracanej, jeśli połączenie okaże się skuteczne, aktywowany jest przycisk *Wyślij dane*, umożliwiając użytkownikowi wysyłanie danych do serwera. Kod obsługi zdarzenia naciśnięcia tego przycisku powinien mieć następującą treść:

```
void CMFCSendTextDlg::OnBnClickedButton3()
{
    // TODO: Add your control notification handler code here
    m_cSocket->Send("Halo", 100);

    int err = m_cSocket->GetLastError();
    if(err > 0)
    {
        CString ErrStr;
        ErrStr.Format("błąd=%d", err);
        AfxMessageBox(ErrStr);
    }
}
```

Przesył danych następuje w ramach wywołania metody `Send` obiektu gniazda klienta `m_cSocket`. Metoda wymaga dwóch parametrów:

- ◆ danych do wysłania (tu stanowi je ciąg „Halo”),
- ◆ rozmiaru danych — powinniśmy tu podać 4 (ciąg ma przecież tylko 4 litery), w wywołaniu figuruje jednak wartość 100. Nie spowoduje to żadnego błędu, a pozwoli w przyszłości dość swobodnie zmieniać wysyłane dane bez konieczności każdorazowego obliczania ich rozmiaru. W późniejszych zastosowaniach należałoby jednak zadbać o obliczenie poprawnego rozmiaru danych.

Jak dotąd nie przejmowaliśmy się specjalnie wynikami inicjowanych operacji. Przy uruchamianiu serwera kontrola taka jest koniecznością, ponieważ jeśli serwer został już wcześniej zainstalowany, kolejne próby otwarcia portu do nasłuchu spowodują błąd wykonania programu. Może też się zdarzyć, że komputer docelowy nie obsługuje protokołu TCP. W takim przypadku po stronie klienta wykryty zostanie błąd połączenia.

Stosowny sprawdzian następuje po wysłaniu danych. Ewentualny kod błędu jest tu odczytywany metodą `GetLastError` — zwraca ona kod błędu ostatniej operacji inicjowanej na rzecz obiektu gniazda. Jeśli wartość zwracana przez tę metodę jest większa od zera, mamy do czynienia z błędem transmisji.

Kod obsługi zdarzenia naciśnięcia przycisku *Rozłącz* jest najkrótszy:

```
void CMFCSendTextDlg::OnBnClickedButton4()
{
    // TODO: Add your control notification handler code here
    SetDlgItemText(IDC_STATIC, "Rozłączony");
    m_cSocket->Close();
}
```

Pierwszy wiersz wyświetla w oknie stosowny komunikat o przerwaniu połączenia. W drugim wierszu następuje zaś wywołanie metody `Close` zamkijającej połączenie z serwrem.

Teraz najciekawsze: metoda `AddConnection` wykorzystywana już przy okazji nawiązania połączenia z serwerem:

```
void CMFCSendTextDlg::AddConnection()
{
    m_scSocket = new CCClientSocket(this);
    m_sSocket->Accept(*m_scSocket);
}
```

Jak widać, w reakcji na nowe połączenie tworzony jest nowy obiekt typu `CCClientSocket`. Następnie nawiązuje on połączenie z gniazdem serwera `m_sSocket` za pośrednictwem metody `Accept`. W ten sposób z nowo nawiązanym połączeniem kojarzony jest obiekt gniazda typu `CCClientSocket`. Za jego pośrednictwem serwer może — nie przerywając nasłuchu — komunikować się z klientem.

Okazuje się, że do realizacji transmisji danych po stronie serwera i klienta wykorzystywane są obiekty tego samego typu — `CCClientSocket`. Obiekt typu `CServerSocket` służy jedynie do prowadzenia nasłuchu i kojarzenia kolejno napływających połączeń z obiektami typu `CCClientSocket`.

Niniejszy przykład działa prawidłowo jedynie wtedy, kiedy z serwerem łączy się jeden klient. Kiedy próbę nawiązania połączenia zainicjuje kolejny klient, zmienna `m_scSocket` zostanie skojarzona z nowym połączeniem, przez co poprzedni klient utraci możliwość komunikacji. W tym miejscu przydałaby się dynamicznie zarządzana tablica gniazd typu `CCClientSocket`, stanowiąca swego rodzaju pulę połączeń, z którymi kojarzone byłyby połączenia kolejnych klientów. Dla każdego nowego klienta należałoby utworzyć nowy, osobny obiekt typu `CCClientSocket` i umieścić go w tablicy. Przy rozłączaniu połączenia należałoby natomiast usunąć obiekt gniazda z tablicy.

Nie powiedzieliśmy jeszcze ani słowa o protokole wymiany informacji pomiędzy klientem a serwerem. Tymczasem domyślnie gniazda klas pochodnych względem CSocket korzystają z protokołów TCP/IP.



Kod źródłowy tego przykładu znajduje się w podkatalogu \Przykłady\Rozdział 4\FCSendText na dołączonej do książki płytcie CD-ROM.

4.6. Bezpośrednie odwołania do biblioteki gniazd

Przekonaliśmy się już, jak prosta jest obsługa połączeń sieciowych za pośrednictwem obiektów i klas biblioteki MFC. Jednak wciąż dążyliśmy do tworzenia jak najbardziej zwartych programów, a nie ma lepszego sposobu na zmniejszenie rozmiaru programu niż korzystanie wprost (bez pośrednictwa obiektów MFC) z bibliotek standardowych systemu Widnows.

W systemie Windows za obsługę sieci odpowiedzialna jest biblioteka gniazd — Winsock. Biblioteka ta występuje w dwóch wersjach. Pierwsza z nich bazowała na modelu biblioteki gniazd wg Berkeley implementowanego powszechnie w systemach uniksowych. Począwszy od Windows 98 „okienka” wyposażane są w drugą wersję biblioteki wbudowaną w system operacyjny.

Nowa wersja biblioteki Winsock jest zgodna wstępco. Oznacza to, że w programach wciąż można wykorzystywać funkcje pierwszej wersji biblioteki i że programy napisane z myślą o tej wersji powinno dać się uruchomić i powinny poprawnie działać w systemach z zainstalowaną nowszą wersją. Nowa wersja biblioteki gniazd zawiera też nowe funkcje niezgodne z ich starszymi odpowiednikami. Wśród owych nowych funkcji wchodzących w skład biblioteki gniazd od wersji 1.1 mamy WSASStartup, WSACleanup, WSAGetLastError i WSARcvEx (wyróżnikiem tych funkcji jest wspólny przedrostek — WSA). W kolejnych wersjach pojawiło się jeszcze więcej nowych funkcji.

Jeśli nawet masz do dyspozycji bibliotekę Winsock2, nie musisz jej używać. Zalecałbym najpierw sprawdzenie, czy nie da się oprzeć programu na pierwszej wersji biblioteki gniazd. Jeśli okaże się to możliwe, nie będziesz miał większych trudności z ewentualnym przystosowaniem programu do komplikacji w systemach uniksowych.

Komputery z systemami Windows 95 są współcześnie prawdziwymi rodzinami, ale wbrew pozorom system ten wciąż jest wykorzystywany. Jeśli posiadasz taki system, możesz pobrać odpowiednią dla niego wersję biblioteki gniazd z witryny www.microsoft.com.

Jeśli zdecydujesz o wykorzystaniu we własnych programach oryginalnej biblioteki gniazd, powinieneś włączyć do kodu źródłowego plik nagłówkowy *winsock.h*. Odpowiednikiem tego pliku dla nowszej wersji jest *windows2.h*.

Czytelnikowi należy się tu ostrzeżenie: w przykładach wykorzystywana będzie zarówno starsza, jak i nowsza wersja biblioteki gniazd.

4.6.1. Obsługa błędów

Na początek powinieneś wiedzieć, jak wykrywać błędy, które mogą wystąpić przy wywołaniach funkcji sieciowych. Właściwa obsługa błędów jest bardzo ważna w każdej aplikacji. Choć funkcje sieciowe nie są najważniejszymi funkcjami systemu operacyjnego, mogą istotnie wpływać na działanie programów. Te z kolei mogą niepoprawnym działaniem degradować bezpieczeństwo systemu.

Programy sieciowe wymieniają dane z innymi komputerami, przy czym w roli drugiej strony komunikacji często występuje nieznana, być może złośliwa osoba. Jeśli zanieśbasz obsługę błędów, może ona uzyskać dostęp do funkcji systemowych albo istotnych danych przechowywanych w komputerze.

Weźmy najprostszy przykład. Założymy, że za każdym razem, kiedy program otrzymuje dane za pośrednictwem sieci, wywołujemy funkcję sprawdzającą poprawność danych i uprawnienia klienta. Jeśli wszystko wygląda tak jak trzeba, funkcja wykonuje pewien kod, który nie powinien być dostępny intruzowi. Obsługa błędów powinna tu występować na każdym etapie działania programu: w momencie otrzymania danych, w momencie kontroli ich poprawności, kontroli uprawnień dostępu zdalnej strony i wreszcie po każdym wywołaniu funkcji sieciowych. Testy takie czynią aplikację — a wraz z nią cały system — bezpieczniejszą i bardziej niezawodną.

Jeśli w ramach wywołania funkcji sieciowej dojdzie do błędu, funkcja zwróci wartość SOCKET_ERROR, czyli -1. Jeśli wykryjesz taką wartość zwracaną funkcji, powinieneś skorzystać z wywołania WSAGetLastError. Nie trzeba przekazywać w nim żadnych parametrów — wywołanie zwraca po prostu kod błędu, który wystąpił w czasie wykonania ostatniej funkcji sieciowej. Kodów błędów zdefiniowano całkiem sporo — ich wartości zależą od rodzaju wywołanej funkcji. Będziemy je omawiać w miarę potrzeb.

4.6.2. Wczytywanie biblioteki gniazd

Zanim zaczniesz wykorzystywać bibliotekę gniazd powinieneś najpierw wczytać do pamięci programu odpowiednią wersję biblioteki. Od tej operacji zależy zestaw funkcji sieciowych dostępnych w programie. Jeśli wczytasz bibliotekę starszą, a w kodzie programu odwołasz się do funkcji wprowadzonych dopiero w późniejszych wersjach, doprowadzisz do błędu wykonania programu. Jeśli nie wczytasz biblioteki w ogóle, wszelkie wywołania implementowanych w tej bibliotece funkcji sieciowych będą zwracać błąd o kodzie WSANOTINITIALIZED.

Wczytanie biblioteki polega na wywołaniu funkcji WSASStartup. Funkcja ta deklarowana jest następująco:

```
int WSASStartup(
    WORD   wVersionRequested,
    LPWSADATA lpWSAData
);
```

Pierwszy parametr wywołania (wVersionRequested) określa pożądaną wersję biblioteki. Mniej znaczący bajt tego parametru (zajmującego dwa bajty) określa numer główny, bajt bardziej znaczący koduje zaś numer poboczny wersji biblioteki. Dla ułatwienia

zalecałbym konstruowanie wartości tego parametru za pośrednictwem makrodefinicji MAKEWORD(i, j), gdzie i to wartość bajta bardziej znaczącego, a j — bajta mniej znaczącego.

Drugi parametr funkcji WSASStartup to wskaźnik struktury WSADATA, w której funkcja zapisze informacje o wczytanej bibliotece.

Jeśli załadowanie biblioteki powiedzie się, wynikiem wywołania będzie wartość zero. Każda inna wartość sygnalizuje błąd.

Oto przykład wywołania funkcji WSASStartup celem wczytania biblioteki Winsock 2.0:

WSADATA wsaData;

```
int err = WSASStartup(MAKEWORD(2, 0), &wsaData);
if (err != 0)
{
    // Poinformuj użytkownika o niemożności wczytania biblioteki Winsock 2.0
    return;
}
```

Zauważ, że zwracana wartość jest porównywana z zerem od razu po wywołaniu funkcji wczytującej bibliotekę gniazd. Jeśli funkcja wykona się prawidłowo, zwróci właśnie zero. W innym przypadku zasygnalizuje błąd. Oto najważniejsze kody błędów:

- ◆ WSASYSNOTREADY — podsystem sieciowy systemu operacyjnego nie jest gotowy do realizowania połączeń.
- ◆ WSAVERNOTSUPPROTED — żądana wersja biblioteki nie jest obsługiwana w systemie.
- ◆ WSAEPROCLIM — liczba zadań poza określonym limitem.
- ◆ WSAEFAULT — błędny wskaźnik struktury WSADATA.

Struktura WSADATA ma następującą definicję:

```
typedef struct WSADATA {
    WORD        wVersion;
    WORD        wHighVersion;
    char        szDescription[WSADESCRIPTION_LEN + 1];
    char        szSystemStatus[WSASYS_STATUS_LEN + 1];
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR *   lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

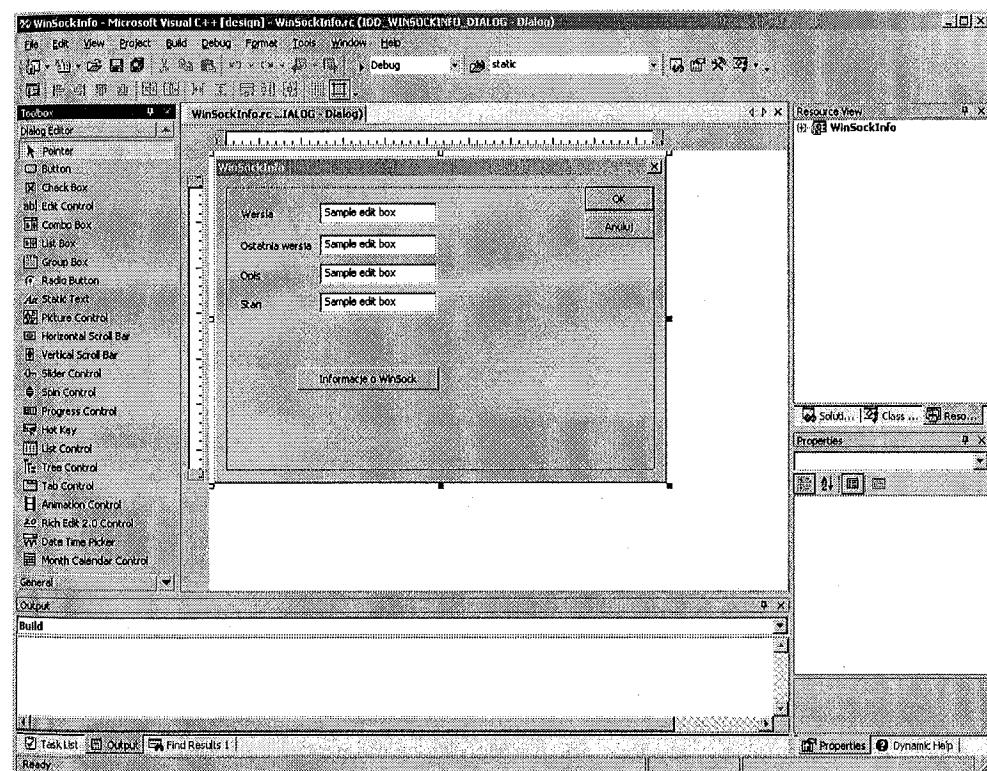
Jakie jest znaczenie jej pól?

- ◆ wVersion — numer wersji wczytanej biblioteki gniazd.
- ◆ wHighVersion — numer ostatniej wersji.
- ◆ szDescription — ciąg znaków opisu biblioteki występujący w niektórych wersjach.

- ◆ szSystemStatus — ciąg znaków opisu stanu biblioteki występujący w niektórych wersjach.
- ◆ iMaxSockets — maksymalna liczba otwartych połączeń. Informacja ta nie odpowiada rzeczywistości o tyle, że faktyczna liczba połączeń zależy od ilości dostępnych zasobów systemowych. Parametr pozostał w definicji struktury gwoli zgodności z jej pierwotną specyfikacją.
- ◆ iMaxUdpDg — maksymalny rozmiar datagramu (pakietu). Informacja ta nie odpowiada rzeczywistości o tyle, że faktyczny rozmiar pakietu zależy od protokołu.
- ◆ lpVendorInfo — informacje o dostawcy biblioteki.

Przyjrzyjmy się teraz prostemu przykładowi wczytania biblioteki gniazd i obsługi struktury WSADATA. Utwórzmy w Visual C++ nowy projekt typu *MFC Application*. W kreatorze aplikacji, na zakładce *Application Type*, zaznaczmy *Dialog based*, a na zakładce *Advanced Features* — *Windows sockets*. Ustawienia te powinny być Ci już znane.

W edytorze zasobów otwórz główne okno aplikacji i zmodyfikuj je zgodnie z rysunkiem 4.13. Formularz okna głównego aplikacji powinien ostatecznie zawierać cztery komponenty pól edycyjnych (*Edit Control*) służące do wyświetlania informacji o bibliotece i pojedynczy przycisk *Informacje o WinSock*, którego kliknięcie będzie wypełniało owe pola.



Rysunek 4.13. Okno główne programu WinSockInfo

Zadeklaruj w programie następujące zmienne:

- ◆ `mVersion` — numer wersji biblioteki,
- ◆ `mHighVersion` — numer ostatniej wersji,
- ◆ `mDescription` — opis biblioteki,
- ◆ `mSystemStatus` — opis stanu.

Utwórz procedurę obsługi zdarzenia kliknięcia przycisku *Informacje o WinSock* i skopiuj do niej kod z listingu 4.11.

Listing 4.11. Pobieranie informacji o bibliotece gniazd

```
void CWinSockInfoDlg::OnBnClickedButton1()
{
    // TODO: Add your control notification handler code here
    WSADATA wsaData;

    int err = WSAStartup(MAKEWORD(2, 0), &wsaData);
    if (err != 0)
    {
        // Poinformuj użytkownika o niemożności wczytania biblioteki Winsock 2.0
        return;
    }

    char mText[255];
    mVersion.SetWindowText(itoa(wsaData.wVersion, mText, 10));
    mHighVersion.SetWindowText(itoa(wsaData.wHighVersion, mText, 10));
    if (wsaData.szDescription)
        mDescription.SetWindowText(wsaData.szDescription);
    if (wsaData.szSystemStatus)
        mSystemStatus.SetWindowText(wsaData.szSystemStatus);
}
```

Na samym początku kodu obsługi zdarzenia wczytywana jest biblioteka gniazd (ten fragment kodu już znamy). Następnie uzyskane w ten sposób informacje są umieszczane w polach edycyjnych okna programu.

Efekt działania programu widać na rysunku 4.14.



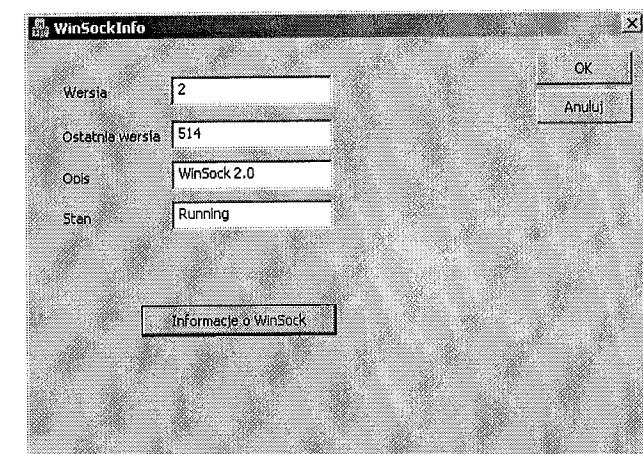
Kod źródłowy tego przykładu znajduje się w podkatalogu \Przykłady\Rozdział\inSockInfo na dołączonej do książki płycie CD-ROM.

Przykład ten obarczony jest jedną wadą — nie usuwa biblioteki z pamięci. Można ów brak uzupełnić, wywołując funkcję `WSACleanup` deklarowaną następująco:

```
int WSACleanup(void);
```

Funkcja ta nie przyjmuje żadnych parametrów. Po zakończeniu wykonania jej kodu funkcje sieciowe stają się po prostu niedostępne dla programu.

Rysunek 4.14.
Informacje
o bibliotece gniazd
— program
WinSockInfo



4.6.3. Tworzenie gniazda

Po wczytaniu biblioteki gniazd należy utworzyć gniazdo pośredniczące w komunikacji sieciowej. W pierwszej wersji biblioteki służy do tego funkcja `socket`:

```
SOCKET socket(
    int af,
    int type,
    int protocol
);
```

W WinSock2 analogiczne zadanie realizuje funkcja `WSASocket`:

```
SOCKET WSASocket(
    int af,
    int type,
    int protocol
    LPWSAPROTOCOL_INFOP lpProtocolInfo,
    GROUP g,
    DWORD dwFlags
);
```

Pierwsze trzy parametry oraz wartość zwracana są w obu funkcjach identyczne. W obu przypadkach wynikiem wywołania funkcji jest nowo utworzone gniazdo. Można je od tego momentu wykorzystywać do komunikacji sieciowej. Przyjrzyjmy się bliżej wspólnym parametrom funkcji:

- ◆ `af` — rodzina protokołów do obsługi gniazda:
 - ◆ `AF_UNSPEC` — brak określenia rodziny protokołów,
 - ◆ `AF_INET` — rodzina protokołów internetowych, jak TCP, UDP itd. (są to protokoły najpopularniejsze i jako takie wykorzystywane najczęściej również w tej książce),
 - ◆ `AF_IPX` — protokół IPX i SPX,

- ◆ AF_APPLETALK — protokół Appletalk,
- ◆ AF_NETBIOS — protokół NetBIOS.
- ◆ type — typ tworzonego gniazda. Dopuszczalne są następujące wartości:
 - ◆ SOCK_STREAM — transfer danych z ustanowionym połączeniem (dla rodziny protokołów internetowych odpowiedni jest tu protokół TCP),
 - ◆ SOCK_DGRAM — bezpołączeniowy transfer danych (dla rodziny protokołów internetowych odpowiedni jest tu protokół TCP).
- ◆ protocol — protokół. Do wyboru jest mnóstwo protokołów. Informacji o nich i reprezentujących je wartościach należy szukać w pomocy podręcznej Visual C++. Osobiście najczęściej stosuję wartość IPPROTO_TCP reprezentującą protokół TCP.

W funkcji WSASocket doszły dodatkowo następujące parametry wywołania:

- ◆ lpProtocolInfo — wskaźnik struktury WSAPROTOCOL_INFO definiującej charakterystykę tworzonego gniazda,
- ◆ g — identyfikator grupy gniazd,
- ◆ dwFlags — atrybuty gniazda.

O znaczeniu i wartościach parametrów dowiemy się więcej przy okazji analizy przykładów prezentowanych w kolejnych punktach. Jak zwykle najlepiej zapamiętać wpływ parametrów na działanie programu, obserwując wprost efekt jego wykonania.

4.6.4. Funkcje strony serwera

Jak już wspomniałem, protokół TCP oparty jest na modelu klient-serwer. Jeśli mamy dwa komputery i jeśli mają one nawiązać połączenie, jeden z nich musi rozpoczęć nasłuch pod określonym numerem portu. Czyni go to serwerem. Dopiero potem klient może spróbować nawiązać połączenie z nasłuchującym komputerem.

Przyjrzymy się więc funkcjom niezbędnym do zainstalowania serwera na wybranym porcie. Po pierwsze należy skojarzyć nowo utworzone gniazdo z pulą adresów, które będą uwzględniane przy nasłuchu. Służy do tego funkcja bind:

```
int bind(
    SOCKET s,
    const struct sockaddr FAR* name,
    int namelen
);
```

Parametry wywołania mają następujące znaczenie:

- ◆ utworzone wcześniej gniazdo,
- ◆ wskaźnik struktury typu sockaddr,
- ◆ rozmiar struktury sockaddr wskazanej drugim parametrem.

Struktura sockaddr służy do przechowywania adresu, a różne protokoły wymagają różnych reprezentacji adresów sieciowych, dlatego też struktura socketaddr jest strukturą dość zmienną. W przypadku protokołów internetowych należy korzystać z jej wersji sockaddr_in, która jest zdefiniowana następująco:

```
struct socketaddr_in {
    short   sin_family;
    u_short sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
};
```

A oto znaczenie jej składowych:

- ◆ sin_family — rodzina protokołów. Element ten należy interpretować podobnie jak pierwszy parametr wywołania funkcji socket. W przypadku protokołów internetowych pole to ma wartość AF_INET.
- ◆ sin_port — numer portu wykorzystywany do transmisji danych.
- ◆ sin_addr — struktura SOCKADDR_IN przechowująca adres IP.
- ◆ sin_zero — pole wykorzystywane do wyrównania rozmiaru struktury; rozmiar struktury SOCKADDR_IN powinien być równy rozmiarowi struktury SOCKADDR.

Chciałbym nieco więcej czasu poświęcić portom. Wybierając port, należy zachować szczególną rozwagę, ponieważ jeśli port o wybranym przez nas numerze jest już okupowany przez inny program, próba zainstalowania nasłuchu będzie nieskuteczna i doprowadzi do błędu. Warto wiedzieć, że niektóre porty są zarezerwowane dla konkretnych (najpopularniejszych) usług. Rezerwacją zarządza organizacja *Internet Assigned Numbers Authority* (IANA). Wyróżnia ona trzy kategorie portów:

- ◆ 0 do 1023 — porty pozostające pod „opieką” IANA i zarezerwowane dla usług standardowych. Nie powinieneś wybierać portu z tego zakresu.
- ◆ 1024 do 49151 — zarezerwowane przez IANA, ale z możliwością wykorzystywania przez procesy i programy. Większość z nich można swobodnie używać.
- ◆ 49152 do 65535 — porty do użytku własnego, nierezerveowane.

Jeśli funkcja bind wykryje, że zadany port jest już wykorzystywany przez inny program, zwróci błąd o kodzie WSAEADDRINUSE.

Spójrzmy na przykład tworzący gniazdo i kojarzący je z lokalnym adresem sieciowym:

```
SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(4888);
addr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(s, (SOCKADDR*)&addr, sizeof(addr));
```

W tym przykładzie tworzymy gniazdo o następujących parametrach:

- ◆ AF_INET — określa zastosowanie gniazda do obsługi protokołów z rodziny protokołów internetowych,
- ◆ SOCK_STREAM — wymusza korzystanie z protokołu połączniowego,
- ◆ IPPROTO_TCP — wyznacza protokół TCP.

Następnie konstruujemy strukturę addr typu sockaddr_in. Jej pole sin_family ustawiamy identycznie jak pierwszy parametr wywołania funkcji tworzącej gniazdo (AF_INET). Pole sin_port przypisujemy numer portu. Bajty w owym numerze (numer portu składa się z dwóch bajtów) powinny być uporządkowane inaczej niż bajty typowych dwubajtowych wartości liczbowych w językach C i C++. Do konwersji do wymaganej reprezentacji sieciowej służy funkcja htons.

Do pola sin_addr.s_addr przypisujemy specjalną wartość INADDR_ANY, dzięki czemu aplikacja korzystająca z gniazda będzie przyjmować połączenia z dowolnego interfejsu sieciowego. Oznacza to, że jeśli w komputerze zainstalowane są dwie karty przyłączone do dwóch różnych sieci, program będzie nasłuchiwał w oczekiwaniu na połączenia z obu sieci. Można by też określić tu inną wartość specjalną, jak INADDR_BROADCAST — to pozwalałoby na realizację transmisji rozgłoszeniowej kierowanej do wszystkich komputerów w danej sieci.

Po skojarzeniu z gniazdem portu i adresu lokalnego, można rozpoczęć nasłuch w oczekiwaniu na połączenia inicjowane przez klientów. Służy do tego funkcja listen:

```
int listen(
    SOCKET s,
    int backlog
);
```

Pierwszy parametr jej wywołania to utworzona wcześniej struktura opisująca gniazdo. Na podstawie zawartości owej struktury funkcja listen może rozpoczęć nasłuch na określonym numerze portu.

Drugi parametr to limit połączeń oczekujących na przetworzenie. Jeśli określmy tam np. trzy, a do portu zostanie skierowanych w krótkim czasie pięć żądań nawiązania połączenia, zrealizowane zostaną jedynie trzy pierwsze z kolejki. Reszta połączeń zostanie odrzucona, a inicjujący je klienci otrzymają kod błędu WSAECONNREFUSED. Wiemy zatem, że pisząc kod klienta, powinniśmy sprawdzić wynik funkcji nawiązującej połączenie pod kątem tego kodu.

Wywołanie funkcji listen może sprowokować następujące błędy:

- ◆ WSAEINVAL — kiedy dla gniazda nie wywołano wcześniej funkcji bind.
- ◆ WSANOTINITIALIZED — kiedy w programie zaniedbalismy wczytanie biblioteki gniazd.
- ◆ WSAENETDOWN — kiedy funkcja wykryje załamanie podsystemu sieciowego.
- ◆ WSAEISCONN — kiedy istnieje już połączenie z gniazdem.

Możliwe jest też wystąpienie innych błędów, te są jednak najczęstsze.

Jeśli żądanie nawiązania połączenia trafi do kolejki połączeń oczekujących serwera, ten powinien przyjąć połączenie, wywołując funkcję accept. Jest ona deklarowana następująco:

```
SOCKET accept(
    SOCKET s,
    struct sockaddr FAR* addr,
    int FAR* addrlen
);
```

W nowszej wersji biblioteki to samo zadanie realizuje funkcja WSAAccept. Jej trzy pierwsze parametry są identyczne jak w funkcji accept. Prototyp funkcji WSAAccept prezentuje się następująco:

```
SOCKET WSAAccept(
    SOCKET s,
    struct sockaddr FAR* addr,
    LPINT addrlen,
    LPCCONDITIONPROC lpCondition,
    DWORD dwCallbackData
);
```

Przyjrzyjmy się wspólnym parametrom obu funkcji. Są nimi:

- ◆ Utworzone wcześniej i uruchomione w nasłuchu gniazdo.
- ◆ Wskaźnik struktury typu sockaddr.
- ◆ Rozmiar struktury przekazanej drugim parametrem.

Po powrocie z funkcji accept drugi parametr jej wywołania (addr) będzie zawierał informacje o adresie IP klienta, który nawiązał połączenie. Informacja ta może zostać wykorzystana do kontrolowania dostępu do serwera na podstawie adresów sieciowych klientów. Trzeba jednak przy tym pamiętać, że ewentualny intruz może potrafić fałszować adresy źródłowe połączenia, więc takie zabezpieczenie nie jest bynajmniej wystarczające.

Funkcja accept zwraca wskaźnik nowego gniazda, które można wykorzystać do komunikacji z klientem. Gniazdo przekazane w wywołaniu pozostaje w stanie nasłuchu, oczekując na kolejne połączenia, nie moglibyśmy więc za jego pośrednictwem realizować komunikacji z klientem. Dla każdego z klientów, którym uda się nawiązać połączenie, tworzone jest więc osobne gniazdo służące do komunikacji wyłącznie z tym klientem.

Jeśli przypominasz sobie przykład wykorzystujący w roli gniazda obiekty MFC (w odróżnieniu 4.5) pamiętasz też zapewne, że zastosowaliśmy tam podobną metodę. Jak tylko klient połączył się z serwerem, utworzyliśmy nowe gniazdo, za pośrednictwem którego realizowaliśmy właściwą komunikację z klientem. Gniazdo to służyło dalej zarówno do odbierania danych wysyłanych przez klienta, jak i wysyłania danych programu do tegoż klienta.

4.6.5. Funkcje strony klienta

W punkcie 4.6.4 poznaliśmy kilka funkcji WinAPI, za pośrednictwem których możliwe jest zainstalowanie w systemie serwera. Nie możemy jednak od razu przystąpić do wykorzystania nowych wiadomości w praktyce, ponieważ nie wiemy jeszcze, jak za pomocą WinAPI oprogramować stronę klienta — nie byłoby więc jak testować program serwera. Pora więc na przegląd funkcji umożliwiających implementację klienta wymiany danych.

Połączenie klienta z serwerem przebiega ze strony tego pierwszego w dwóch fazach: utworzenia gniazda i nawiązania połączenia za pośrednictwem tego gniazda. Tak wygląda to w przypadku idealnym. Zazwyczaj konieczna jest jeszcze faza trzecia. Na czym ona polega? Otóż nie wszyscy potrafią zapamiętać adresy IP serwerów, korzystają więc z prostszych do zapamiętania nazw symbolicznych. W takim układzie potrzebna jest jeszcze faza konwersji adresu z postaci symbolicznej na postać liczbową.

Wiemy już, jak utworzyć gniazdo. Po stronie klienta wygląda to tak samo jak po stronie serwera. Przejdzmy więc do translacji nazwy serwera na postać jego adresu. W zależności od wykorzystywanej biblioteki gniazd należy do tego wykorzystać albo funkcję `gethostbyname`, albo `WSAAAsyncGetHostByName`. Przyjrzyjmy się najpierw funkcji `gethostbyname`:

```
struct hostent FAR* gethostbyname(
    const char FAR * name
);
```

Funkcja przyjmuje zaledwie jeden parametr reprezentujący symboliczną nazwę serwera. Funkcja zwraca strukturę typu `hostent`, o której powiemy sobie więcej nieco później.

Teraz funkcja `WSAAAsyncGetHostByName`:

```
HANDLE WSAAAsyncGetHostByName(
    HWND hWnd,
    unsigned int wMsg,
    const char FAR * name,
    char FAR* buf,
    int buflen
);
```

Funkcja ta działa asynchronicznie. Oznacza to, że program nie zostaje zawieszony w oczekiwaniu na zakończenie jej wykonania. Program może pracować dalej, a o wyniku działania funkcji dowiaduje się za pośrednictwem komunikatu określonego drugim parametrem wywołania funkcji. To bardzo wygodne, ponieważ proces konwersji nazwy do adresu IP trwa z punktu widzenia przydziału czasu procesora bardzo długo — można ten czas wykorzystać w programie na pozostałe operacje poprzedzające nawiązanie połączenia.

Oto znaczenie parametrów wywołania `WSAAAsyncGetHostByName`:

- ◆ `hWnd` — uchwyt okna, które ma odebrać komunikat sygnalizujący zakończenie asynchronicznej operacji translacji adresu.
- ◆ `wMsg` — komunikat, który zostanie przesłany do programu w momencie rozstrzygnięcia adresu IP.

- ◆ `name` — symboliczna nazwa komputera, którego adres ma zostać pozyskany.
- ◆ `buf` — wskaźnik bufora, w którym ma być przechowywana struktura `hostent`. Bufor ten powinien być odpowiednio duży. Jego maksymalny rozmiar określa makrodefinicja `MAXGETHOSTSTRUCT`.
- ◆ `buflen` — rozmiar bufora wskazywanego czwartym parametrem wywołania.

A oto i sama struktura `hostent` zawierająca wyniki poszukiwania adresu IP:

```
struct hostent {
    char FAR * h_name;
    char FAR * FAR * h_aliases;
    short h_addrtype;
    short h_length;
    char FAR * FAR * h_addr_list;
};
```

Jej elementy mają następujące znaczenie:

- ◆ `h_name` — pełna nazwa węzła. Jeśli w sieci stosowany jest system nazw domenowych, pole to zawierać będzie pełną nazwę domenową komputera (węzła), którego adres był poszukiwany.
- ◆ `h_aliases` — dodatkowe nazwy węzła.
- ◆ `h_addrtype` — typ adresu.
- ◆ `h_length` — długość każdego z adresów na liście.
- ◆ `h_addr_list` — lista adresów węzła.

Pojedynczy komputer (węzeł sieci) może mieć więcej niż jeden adres, więc struktura przechowuje pełną listę takich adresów. W większości przypadków wystarczy odwołać się do pierwszego z brzegu adresu. Jeśli funkcja `gethostbyname` (albo `WSAAAsyncGetHostByName`) odnajdzie więcej niż jeden adres, do nawiązania połączenia można wykorzystać dowolny z nich.

Do nawiązania właściwego połączenia służy funkcja `connect`:

```
int connect(
    SOCKET s,
    const struct sockaddr FAR * name,
    int namelen
);
```

Wymaga ona przekazania w wywołaniu trzech parametrów:

- ◆ `s` — gniazda, które ma obsługiwać połączenie,
- ◆ `name` — struktury `SOCKADDR` przechowującej adres serwera,
- ◆ `namelen` — rozmiaru struktury `SOCKADDR` przekazanej drugim parametrem wywołania.

W drugiej wersji biblioteki gniazd to samo zadanie zostało oddelegowane do funkcji `WSAConnect`:

```

int WSACConnect(
    SOCKET s,
    const struct sockaddr FAR * name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
);

```

Znaczenie pierwszych trzech parametrów jest identyczne jak w funkcji connect. Interesują nas jeszcze dwa parametry:

- ◆ lpCallerData — wskaźnik bufora danych, które zostaną przesłane do serwera w czasie nawiązywania połączenia.
- ◆ lpCalleeData — wskaźnik bufora danych, które zostaną odebrane od serwera w czasie nawiązywania połączenia.

Oba parametry są wskaźnikami struktury typu WSABUF, której skład prezentuje się następująco:

```

typedef struct _WSABUF {
    u_long    len;
    char FAR * buf;
} WSABUF, FAR * LPWSABUF;

```

Pierwsze pole struktury to rozmiar bufora, a drugi to jego wskaźnik. Ostatnie dwa parametry wywołania WSACConnect (lpSQOS i lpGQOS) to wskaźniki struktur typu QOS. Struktury te opisują wymagania co do przepustowości kanału przy wysyłaniu i odbieraniu danych. Przekazując w wywołaniu zero, sygnalizuje się brak szczególnych wymagań co do jakości połączenia.

Przy próbie nawiązania połączenia mogą wystąpić następujące błędy:

- ◆ WSAE TIMEOUT — niedostępność serwera (prawdopodobny problem na trasie pakietów).
- ◆ WSAECONNREFUSED — serwer nie rozpoczął nasłuchu na określonym porcie.
- ◆ WSAEADDRINUSE — określony adres jest już wykorzystywany w innym połączeniu.
- ◆ WSAEAFNOSUPPORT — adres jest nieodpowiedni dla danego gniazda. Błąd ten występuje, kiedy adres został określony zgodnie z regułami jednego protokołu, a połączenie wykorzystuje inny protokół.

4.6.6. Wymiana danych

Wiemy już, jak zainstalować serwer i jak zainicjować połączenie. Teraz wypadałoby dowiedzieć się czegoś o sposobach właściwej transmisji danych — to jest przecież ostateczny cel korzystania z funkcji sieciowych. Wszystkie poznane dotychczas funkcje mają nam po prostu umożliwić wymianę danych pomiędzy komputerami.

Na początek chciałbym zaznaczyć, że funkcje, które omawialiśmy powstały, zanim na arenę wkroczył standard UNICODE (to standard kodowania znaków na tyle uniwersalny, że pozwala na kodowanie znaków alfabetów wszystkich języków). Aby w transmisji danych skorzystać z dobrodziejstw owego standardu kodowania, należy dokonać konwersji ciągu znaków z typu char*; konwersja powoduje dwukrotne wydłużenie ciągu — pojedynczy znak UNICODE zajmuje 2 bajty, podczas gdy pojedynczy znak ASCII to dokładnie jeden bajt.

Aby ktoś mógł odebrać dane, ktoś inny musi je najpierw wysłać. Omówienie wymiany danych rozpoczęć więc od ich wysyłania. Aby przesyłać dane do serwera, należy skorzystać z jednej z dwóch funkcji: send albo WSASend (ta ostatnia należy do biblioteki WinSock2). Funkcja send deklarowana jest jak poniżej:

```

int send(
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags
);

```

Oto znaczenie parametrów wywołania funkcji:

- ◆ s — gniazdo pośredniczące w wysyłaniu danych. Program może utworzyć więcej niż jedno połączenie, komunikując się z kilkoma różnymi serwerami, trzeba więc dokładnie określić tym parametrem, o które połączenie chodzi.
- ◆ buf — wskaźnik bufora zawierającego dane przeznaczone do wysyłania.
- ◆ len — rozmiar bufora danych.
- ◆ flags — znaczek określający metodę wysyłania. Można tu określić kombinację następujących wartości:
 - ◆ 0 — brak znaczników,
 - ◆ MSG_DONTROUTE — wysyłane pakiety nie będą routowane. Jeśli protokół transportowy wykorzystywany do transmisji danych nie jest protokołem routowanym, znaczek ten będzie ignorowany,
 - ◆ MSG_OOB — wysyłanie danych „pozapasmowo”, czyli w trybie pilnym.

Funkcja WSASend deklarowana jest następująco:

```

int WSASend(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);

```

Przyjrzyjmy się jej parametrom:

- ◆ s — gniazdo obsługujące połączenie.
- ◆ lpBuffers — struktura w postaci tablicy struktur typu WSABUF. Strukturę WSABUF już znamy — była omawiana przy okazji funkcji connect. Identyczna struktura wykorzystywana jest w roli bufora danych do wysłania.
- ◆ dwBufferCount — liczba struktur SWABUF wskazywanych parametrem lpBuffers.
- ◆ lpNumberOfBytesSent — liczba bajtów wysłanych w ramach operacji wejścia-wyjścia.
- ◆ dwFlags — określa metodę transmisji i przyjmuje wartości identyczne jak parametr dwFlags funkcji connect.
- ◆ lpOverlapped i lpCompletionRoutine — określone w przypadku wykorzystywania nakładających się operacji wejścia-wyjścia. To jeden z modeli operacji asynchronicznych obsługiwanych przez bibliotekę gniazd.

Jeśli funkcja send (albo WSASend) pomyślnie wykona swoje zadanie, zwróci liczbę wysłanych bajtów. W przeciwnym razie wartością funkcji będzie -1 (albo równa wartości -1 stała SOCKET_ERROR). Po wykryciu takiej wartości należałoby przeanalizować przyczynę błędu, wywołując funkcję WSAGetLastError. Może ona wtedy zwrócić jeden z następujących kodów:

- ◆ WSAECONNABORTED — połączenie z serwerem zostało przerwane, czyli albo doszło do wyczerpania limitu czasu oczekiwania na potwierdzenie, albo funkcja wykryła inny błąd.
- ◆ WSAECONNRESET — węzeł zdalny zerwał połączenie; należy zamknąć gniazdo.
- ◆ WSAENOTCONN — połączenie nie zostało nawiązane.
- ◆ WSAETIMEOUT — upływ czasu oczekiwania na potwierdzenie.

Do odbierania danych służy funkcja recv albo WSARecv (ta ostatnia dostępna jest w nowszej wersji biblioteki gniazd). Oto deklaracja funkcji recv:

```
int recv(
    SOCKET s,
    char FAR * buf,
    int len,
    int flags
);
```

Znaczenie jej parametrów nie różni się wiele od znaczenia analogicznych parametrów wywołania funkcji send:

- ◆ s — gniazdo pośredniczące w odbieraniu danych. Program może utworzyć więcej niż jedno połączenie, komunikując się z kilkoma różnymi serwerami, trzeba więc dokładnie określić tym parametrem, o które połączenie chodzi.
- ◆ buf — wskaźnik bufora na odbierane dane.
- ◆ len — rozmiar bufora danych.

- ◆ flags — znacznik określający metodę transmisji. Można tu określić kombinację następujących wartości:
 - ◆ 0 — brak znaczników,
 - ◆ MSG_PEEK — podgląd pakietu w systemowym buforze odbiorczym bez jego usuwania z tego bufora (domyślnie dane są usuwane z bufora po przekazaniu do programu),
 - ◆ MSG_OOB — przetwarzanie danych „pozapasmowych”.

Nie zalecam stosowania znacznika MSG_PEEK, ponieważ może to doprowadzić do wielu problemów. W takich przypadkach każdorazowo trzeba dla podglądzanych danych wywołać funkcję recv ponownie, tym razem celem ostatecznego usunięcia danych z systemowego bufora odbiorczego. Przy następnym odczycie danych bufor może jednak zawierać więcej danych niż poprzednio (w międzyczasie do danego portu mogą napływać dodatkowe pakiety), ryzykujemy więc z jednej strony dwukrotne przetwarzanie tych samych danych, z drugiej zaś pominięcie części danych w przetwarzaniu.

Kolejny problem, który można w ten sposób sprowokować, polega na niezwolnieniu pamięci systemowej i zmniejszenie rozmiarów buforów dla nowych danych. Dlatego znacznik MSG_PEEK powinien być stosowany z zachowaniem szczególnej rozwag i tylko w razie istotnej konieczności.

Weźmy na warsztat funkcję WSARecv:

```
int WSARecv(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRevd,
    LPDWORD lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Podobieństwo parametrów do parametrów wywołania funkcji WSASend jest oczywiste:

- ◆ s — gniazdo obsługujące połączenie.
- ◆ lpBuffers — struktura w postaci tablicy struktur typu WSABUF, czyli buforów odbiorczych.
- ◆ dwBufferCount — liczba struktur SWABUF wskazywanych parametrem lpBuffers.
- ◆ lpNumberOfBytesSent — liczba bajtów odebranych w ramach operacji wejścia-wyjścia.
- ◆ dwFlags — metoda transmisji; może przyjmować wartości identyczne jak parametr dwFlags funkcji recv oraz jeden znacznik dodatkowy — MSG_PARTIAL. Ten ostatni można stosować, jeśli protokół obsługuje odbiór danych w wielu etapach. Określenie w wywołaniu znacznika MSG_PARTIAL spowoduje odbieranie w ramach pojedynczej operacji odczytu jedynie części danych.

- ◆ `lpOverlapped` i `lpCompletionRoutine` — określane w przypadku wykorzystywania nakładających się operacji wejścia-wyjścia. To jeden z modeli operacji asynchronicznych obsługiwanych przez bibliotekę gniazd.

Powiniem wspomnieć, że w przypadku wykorzystywania protokołów zorientowanych na komunikaty (jak protokół UDP) przy określeniu nieprawidłowego rozmiaru bufora (mniejszego od rozmiaru komunikatu) funkcje odbiorcze będą generować błąd o kodzie `WSAEMSGSIZE`. W przypadku protokołów strumieniowych (jak choćby TCP) błąd ten nie pojawi się, ponieważ „nadmiarowe” dane są buforowane przez system i program odbiera je porcjami. Jeśli rozmiar bufora okaże się niewystarczający, reszta danych zostanie udostępniona w kolejnych operacjach odczytu.

W bibliotece WinSock2 implementowana jest pewna ciekawa funkcja. Funkcje starszej wersji biblioteki gniazd (te bez przedrostka WSA) działają i są stosowane podobnie tak w systemie Windows, jak i w systemach uniksowych. Biblioteka WinSock2 zawiera jednak charakterystyczne tylko dla Windows rozszerzenie zestawu funkcji sieciowych o funkcję `TransmitFile`.

Funkcja `TransmitFile` przesyła za pośrednictwem sieci całą zawartość wskazanego pliku. Transmisja odbywa się szybko, ponieważ wysyłaniem zajmuje się jądro biblioteki. Programista aplikacji sieciowej nie musi się więc zajmować kontrolowaniem rozmiaru dotychczas wysłanych danych i wczytywaniem kolejnych porcji pliku. Wszystkim tym zajmuje się biblioteka gniazd.

Oto rzeczona funkcja:

```
BOOL TransmitFile(
    SOCKET hSocket,
    HANDLE hFile,
    DWORD nNumberOfBytesToWrite,
    DWORD nNumberOfBytesPerSend,
    LPOVERLAPPED lpOverlapped,
    LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers
    DWORD dwFlags
);
```

Jej parametry to:

- ◆ `hSocket` — gniazdo pośredniczące w transmisji.
- ◆ `hFile` — uchwyt otwartego pliku do wysłania.
- ◆ `nNumberOfBytesToWrite` — liczba bajtów pliku, które mają zostać wysłane. Dla wartości zero wysłany zostanie cały plik.
- ◆ `nNumberOfBytesPerSend` — rozmiar pojedynczego pakietu transmisji. Dla wartości 1024 plik będzie wysyłany w pakietach po 1024 bajty. Dla wartości zero zastosowany zostanie domyślny rozmiar pakietów.
- ◆ `lpOverlapped` — parametr używany w operacjach nakładających się.
- ◆ `lpTransmitBuffers` — parametr niosący dodatkowe informacje wysypane przed i po wysłaniu właściwej zawartości pliku. Dane te są wykorzystywane przez stronę odbiorczą do wykrywania początku i końca transmisji.

- ◆ `dwFlags` — znaczniki. Dopuszczalne są następujące wartości:
 - ◆ `TF_DISCONNECT` — wymusza zamknięcie gniazda po zakończeniu transmisji,
 - ◆ `TF_REUSE_SOCKET` — przygotowuje gniazdo do ponownego użycia,
 - ◆ `TF_WRITE_BEHIND` — wymusza zakończenie działania bez oczekiwania na potwierdzenie odbioru danych przez drugą stronę połączenia.

Parametr `lpTransmitBuffers` to wskaźnik następującej struktury:

```
typedef struct _TRANSMIT_FILE_BUFFERS {
    PVOID Head;
    DWORD HeadLength;
    PVOID Tail;
    DWORD TailLength;
} TRANSMIT_FILE_BUFFERS;
```

Oto jej elementy składowe:

- ◆ `Head` — wskaźnik bufora zawierającego dane przeznaczone do przesłania przed właściwą zawartością pliku.
- ◆ `HeadLength` — rozmiar bufora wskazywanego parametrem `Head`.
- ◆ `Tail` — wskaźnik bufora zawierającego dane przeznaczone do przesłania po właściwej zawartości pliku.
- ◆ `TailLength` — rozmiar bufora wskazywanego parametrem `Tail`.

4.6.7. Zamykanie połączenia

Aby zakończyć sesję transmisji danych, należy przede wszystkim poinformować drugą stronę połączenia o zakończeniu wymiany danych. Służy do tego funkcja `shutdown`:

```
int shutdown(
    SOCKET s,
    int how
);
```

Pierwszy parametr wywołania funkcji to gniazdo połączenia, które ma zostać zamknięte. Drugi parametr może przyjąć jedną z poniższych wartości:

- ◆ `SD_RECEIVE` — blokuje wszelkie funkcje odbioru danych. Parametr ten nie ma wpływu na gniazda obsługujące protokoły niższych warstw. W przypadku protokołu strumieniowego (takiego jak TCP), kiedy w buforach systemowych na wywołanie funkcji `recv` oczekują nieodebrane jeszcze przez program dane, czyli dane, które napłynęły do węzła po ostatnim wywołaniu `recv`, połączenie zostanie wyzerowane. W przypadku protokołu UDP nastąpi odbiór komunikatów.
- ◆ `SD_SEND` — blokuje wszelkie funkcje nadawcze.
- ◆ `SD_BOTH` — blokuje wysyłanie i odbiór danych.

Po zasygnalizowaniu drugiej stronie konieczności zakończenia połączenia można zamknąć gniazdo. Służy do tego funkcja closesocket:

```
int closesocket(
    SOCKET s
);
```

Funkcja zamknięcia gniazda określone parametrem wywołania. Jeśli gniazdo poddane działaniu tej funkcji zostałoby później wykorzystane w jednej z funkcji sieciowej, funkcja ta wygenerowałaby błąd WSAENOTSOCK (przekazany deskryptor nie jest gniazdem). Wszystkie pakiety oczekujące na wysłanie bądź odebranie (przechowywane w buforach systemowych) zostaną w ramach zamknięcia gniazda odrzucone.

4.6.8. Zasady stosowania protokołów bezpołączeniowych

Wszystko, co powiedziałem wcześniej, dotyczyło protokołów, które na potrzeby transmisji danych ustanawiają połączenia pomiędzy stronami wymiany (klientem a serwerem) — np. protokołu TCP. Istnieją jednak protokoły (jak choćby UDP), które nie ustanawiają połączeń. W takim przypadku zbędne są wywołania funkcji connect, a sama wymiana odbywa się nieco inaczej niż w przypadku protokołów połączeniowych. Celowo unikałem wcześniej tego tematu, aby nie wprowadzać zamieszania. Przy korzystaniu z protokołów bezpołączeniowych wystarczy po stronie serwera wywołać funkcje socket i bind, aby utworzyć gniazdo i skojarzyć je z lokalnym portem nasłuchu. Nie trzeba potem wywoływać funkcji listen ani accept, ponieważ te służyły do nawiązania połączenia pomiędzy nasłuchującym serwerem a inicjującym połączenie klientem. Zamiast tego serwer po prostu oczekuje na napływające dane, które odbiera z gniazda za pośrednictwem funkcji recvfrom:

```
int recvfrom(
    SOCKET s,
    char FAR * buf,
    int len,
    int flags,
    struct sockaddr FAR * from,
    int FAR * fromlen
);
```

Pierwsze cztery parametry wywołania nie różnią się od parametrów funkcji recv. Parametr from wskazuje strukturę sockaddr przechowującą adres IP komputera, który wysyłał dane. Parametr fromlen przekazuje z kolei rozmiar owej struktury.

W nowszej wersji biblioteki gniazd (WinSock2) dostępny jest odpowiednik tej funkcji w postaci WSARecvFrom. Również ona różni się od WSARecv dwoma dodatkowymi parametrami: lpRecvFrom i fromlen:

```
int WSARecvFrom (
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
```

```
LPDWORD lpFlags,
    struct sockaddr FAR * lpFrom;
    LPINT lpFromLen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Po stronie klienta sprawia przedstawia się równie nieskomplikowanie. Wystarczy utworzyć gniazdo i można już wysyłać dane. Służy do tego funkcja sendto:

```
int send(
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags
    const struct sockaddr FAR * to,
    int FAR * tolen
);
```

Pierwsze cztery parametry wywołania należy interpretować identycznie jak w funkcji send. Dalej, parametr to wskazuje strukturę typu sockaddr. Zawiera on adres węzła docelowego transmisji i numer portu, do którego dane mają być transmitowane. Z racji braku połączenia pomiędzy klientem a serwerem informacje te powinny być przekazane funkcji wysyłającej. Ostatni parametr to po prostu rozmiar struktury wskazywanej parametrem to.

Począwszy od drugiej wersji biblioteki gniazd programiści pracujący w Windows mogą korzystać z funkcji WSASendTo. Przyjmuje ona w wywołaniu parametry identyczne z WSASend, z dwoma dodatkowymi: lpTo i iTolLen, pośredniczące w przekazaniu do funkcji wskaźnika struktury z adresem odbiorcy transmisji i rozmiaru tej struktury.

```
int WSASend(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    const struct sockaddr FAR * lpTo;
    int iTolLen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Jak widać, korzystanie z protokołów bezparametrowych jest znacznie prostsze niż korzystanie z protokołów wymagających ustanawiania połączenia. Nie trzeba instalować nasłuchu na porcie po stronie serwera i wywoływać funkcji inicjujących połączenie po stronie klienta. Jeśli więc rozumiesz zasadę wymiany danych za pośrednictwem protokołu TCP, nie będziesz miał zupełnie żadnych problemów z opanowaniem transmisji opartych na UDP.

4.7. Korzystanie z sieci za pośrednictwem protokołu TCP

Czas na wypróbowanie opisanych wcześniej funkcji biblioteki gniazd. Pokażę teraz prosty program, w ramach którego klient wysyła do serwera żądanie, a serwer na to żądanie odpowiada. Przykład ten pomoże Czytelnikowi zrozumieć, jak hakerzy programują konie trojańskie wykradające dane z komputerów zdalnych.

4.7.1. Przykładowy serwer TCP

Zacznijemy od programowania serwera. W tym celu powinieneś utworzyć nowy projekt typu *Win32 Project* i nazwać go np. *TCPServer*. Otwórz teraz pliku *TCPServer.cpp* i wstaw do niego dwie funkcje z listingu 4.12, umieszczając je za deklaracjami zmiennych globalnych, ale przed funkcją *_tWinMain*. Pamiętaj o kolejności funkcji: pierwsza powinna być *ClientThread*, druga — *NetThread*.

Listing 4.12. Funkcje obsługi połączenia

```
DWORD WINAPI ClientThread(LPVOID lpParam)
{
    SOCKET sock = (SOCKET)lpParam;
    char szRecvBuff[1024];
    char szSendBuff[1024];
    int ret;

    // Rozpocznij nieskończoną pętlę
    while(1)
    {
        // Odbierz dane
        ret = recv(sock, szRecvBuff, 1024, 0);
        // Sprawdź odebrane dane
        if (ret == 0)
            break;
        else if (ret == SOCKET_ERROR)
        {
            MessageBox(0, "Błąd odbioru danych", "Błąd", 0);
            break;
        }
        szRecvBuff[ret] = '\0';

        // Sprawdź odebrany ciąg przechowywany w buforze szRecvBuffer
        // Przygotuj ciąg do odesłania klientowi
        strcpy(szSendBuff, "Polecenie GET przyjęte");

        // Wyślij klientowi zawartość bufora szSendBuff
        ret = send(sock, szSendBuff, sizeof(szSendBuff), 0);
        if (ret == SOCKET_ERROR)
        {
            break;
        }
    }
}
```

```

    }

    return 0;
}

DWORD WINAPI NetThread(LPVOID lpParam)
{
    SOCKET sServerListen,
    SOCKET sClient;
    struct sockaddr_in localaddr,
    struct sockaddr_in clientaddr;
    HANDLE hThread;
    DWORD dwThreadId;
    int iSize;

    // Utwórz gniazdo
    sServerListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (sServerListen == SOCKET_ERROR)
    {
        MessageBox(0, "Nie można utworzyć gniazda", "Błąd", 0);
        return 0;
    }
    // Wypełnij strukturę localaddr informacjami o lokalnym adresie serwera
    // i numerze portu nasłuchu
    localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    localaddr.sin_family = AF_INET;
    localaddr.sin_port = htons(5050);

    // Skojarz gniazdo z adresem i portem nasłuchu określonym w localaddr
    if (bind(sServerListen, (struct sockaddr *)&localaddr,
        sizeof(localaddr)) == SOCKET_ERROR)
    {
        MessageBox(0, "Nie można zainstalować serwera", "Błąd", 0);
        return 1;
    }
    // Poinformuj użytkownika o stanie instalacji
    MessageBox(0, "Serwer zainstalowany", "Błąd", 0);

    // Rozpocznij nasłuch
    listen(sServerListen, 4);

    // Poinformuj użytkownika o stanie nasłuchu
    MessageBox(0, "Nasłuch rozpoczęty", "Błąd", 0);

    // Rozpocznij nieskończoną pętlę nasłuchu
    while (1)
    {
        iSize = sizeof(clientaddr);
        // Przyjmij następne żądanie nawiązania połączenia z kolejki.
        // Jeśli kolejka jest pusta, poczekaj na żądania.
        sClient = accept(sServerListen, (struct sockaddr *)&clientaddr,
            &iSize);
        // Sprawdź identyfikator gniazda klienta
        if (sClient == INVALID_SOCKET)
        {
            MessageBox(0, "Błąd funkcji accept", "Błąd", 0);
            break;
        }
    }
}
```

```

// Utwórz nowy wątek do obsługi połączenia z klientem
hThread = CreateThread(NULL, 0, ClientThread,
    (LPVOID)sClient, 0, &dwThreadId);
if (hThread == NULL)
{
    MessageBox(0, "Nie można utworzyć wątku", "Błąd", 0);
    break;
}
CloseHandle(hThread);
}

// Zamknij gniazdo po zakończeniu wątku nasłuchu
closesocket(sServerListen);
return 0;
}

```

Teraz jeszcze w funkcji `_tWinMain`, przed główną pętlą komunikatów, trzeba umieścić taki kod:

```

WSADATA      ws;
if (WSAStartup(MAKEWORD(2,2), &ws) != 0)
{
    MessageBox(0, "Nie można wczytać biblioteki WinSock", "Błąd", 0);
    return 0;
}

HANDLE      hNetThread;
DWORD       dwNetThreadId;
hNetThread = CreateThread(NULL, 0, NetThread, 0, 0, &dwNetThreadId);

```

Spójrzmy, cośmy najlepszego narobili. Funkcja `_tWinMain` wczytuje bibliotekę WinSock w wersji 2.2.

Następnie, za pomocą funkcji `CreateThread`, tworzony jest nowy wątek programu. Chodzi o to, że wywołanie funkcji `accept` blokuje program serwera. Gdyby funkcję tę wywołać w głównym wątku programu, okno programu przestałoby reagować na komunikaty. Z tego właśnie względu tworzony jest osobny wątek serwera. W efekcie w jednym wątku wykonywany jest program główny, a w drugim wątku — współbieżnie — prowadzony jest nasłuch w oczekiwaniu na napływanie połączeń.

Z punktu widzenia programisty wątek jest funkcją, która jest wykonywana równolegle (współbieżnie) do pozostałych wątków systemu operacyjnego. Tak właśnie działa wielozadaniowość w systemie Windows. Więcej informacji na temat wątków należałoby jednak szukać w dokumentacji systemu albo książkach poświęconych w całości środowisku Visual C++.

Trzeci parametr funkcji `CreateThread` to wskaźnik funkcji, która będzie wykonywana w ramach nowego wątku.

Najciekawsze rzeczy dzieją się właśnie w tej funkcji — `NetThread`. Co do jej treści, to wszystkie wywoływane w niej funkcje już znamy. Tu mamy do czynienia jedynie z ich pierwszym praktycznym zastosowaniem.

Na samym początku funkcja `NetThread` tworzy — za pomocą funkcji `socket` — gniazdo. Następnie wypełnia odpowiednimi wartościami strukturę `localaddr` typu `sockaddr_in`. Dla danego serwera wypełniane są trzy pola struktury:

- ◆ `localaddr.sin_addr.s_addr` — wartość `INADDR_ANY` w tym polu wymusza przyjmowanie połączeń z dowolnych interfejsów zainstalowanych w systemie.
- ◆ `localaddr.sin_family` — wartość `AF_INET` sygnalizuje wykorzystanie rodzin protokołów internetowych.
- ◆ `localaddr.sin_port` — w większości systemów port o numerze 5050 powinien być wolny.

Następnie tak wypełniona struktura jest przekazywana do funkcji `bind` kojarzącej gniazdo z adresem lokalnym.

Od tego momentu gniazdo jest gotowe do rozpoczęcia nasłuchu. Nasłuch inicjuje się wywołaniem funkcji `listen`. Drugim parametrem jej wywołania jest czwórka, co oznacza, że serwer będzie dysponował kolejką dla czterech połączeń. Jeśli w krótkim przedziale czasu z serwerem połączy się więcej niż czterech klientów, obsłużona zostanie jedynie pierwsza czwórka. Reszta spotka się z odmową nawiązania połączenia.

Na potrzeby odbierania połączeń klientów inicjowana jest pętla nieskończona. Ma ona obsługiwać wszystkie połączenia. Dlaczego pętla ma trwać w nieskończoność? Cóż, serwer powinien być w każdej chwili i wciąż gotowy do obsługi żądań klientów.

Wewnątrz pętli wywoływana jest funkcja `accept`, która wybiera z kolejki żądanie nawiązania połączenia. Kiedy połączenie zostanie nawiązane, funkcja ta utworzy gniazdo i zwróci wskaźnik tego gniazda — zostanie ono przypisane do zmiennej `sClient`. Przed skorzystaniem z nowo otwartego gniazda należy jeszcze sprawdzić poprawność jego utworzenia. Jeśli zmienna `sClient` ma wartość `INVALID_SOCKET`, z gniazda nie należy korzystać w ogóle.

Jeśli gniazdo jest poprawne, tworzony jest kolejny wątek. Będzie on obsługiwał właściwą wymianę danych pomiędzy serwerem a klientem, to znaczy odbierał dane od klienta i reagował na jego żądania. Wątek tworzony jest znaną nam już funkcją `CreateThread`. Trzecim parametrem jej wywołania jest wskaźnik funkcji implementującej nowy wątek — `ClientThread`. Ona również zostanie uruchomiona współbieżnie z programem głównym i wątkiem nasłuchu serwera.

Czwartym parametrem funkcji `CreateThread` można przekazać wartość, która zostanie przekazana do funkcji wątku jako jej parametr wywołania. Możemy ów parametr wykorzystać do przekazania wątkowi gniazda, za pośrednictwem którego ma się on komunikować z klientem.

Funkcja wątku, `ClientThread`, przyjmuje tylko jeden parametr — ten, który przekazujemy jako czwarty parametr wywołania funkcji tworzącej wątek, `CreateThread`. W pierwszym wierszu kodu funkcji wątku wskaźnik ten jest przepisywany do lokalnej zmiennej `sock` typu `SOCKET`:

```
SOCKET sock = (SOCKET)Param;
```

Również ten wątek uruchamia pętlę nieskończoną, w której realizuje kolejno funkcje odbioru danych i wysyłania odpowiedzi do klienta.

Pierwszą operacją pętli jest odebranie danych od klienta. Dane te należy sprawdzić pod kątem poprawności odbioru.

Jeśli test wypadnie pomyślnie, należałoby sprawdzić, jakieś to polecenie wysłał do nas klient. Gdybyśmy pisali program konia trojańskiego, moglibyśmy obsługiwać polecenia przesyłania przechwyconych hasel, przeładowania systemu operacyjnego czy uruchomienia programu-żartu. Wtedy też trzeba byłoby porównać otrzymane polecenie z wzorcami znanych poleceń i odpowiednio zareagować.

Polecenie mogłoby przybrać postać prostego ciągu, takiego jak „restart” czy „wyślijhasła”. Ponieważ na razie przyglądamy się jedynie podstawowym działaniom koni trojańskich i nie silimy się na oprogramowanie takiego konia, ograniczymy listę rozpoznawanych poleceń do polecenia „GET”. Serwer powinien na takie polecenie odpowiedzieć potwierdzeniem w postaci „Polecenie GET przyjęte”. Potwierdzenie to jest umieszczane w buforze, którego zawartość jest następnie przesyłana do klienta funkcją send:

```
strcpy(szSendBuff, "Polecenie GET przyjęte");

ret = send(sock, szSendBuff, sizeof(szSendBuff), 0);
if (ret == SOCKET_ERROR)
{
    break;
}
```

Następnie wykonywany jest kolejny przebieg pętli, w której serwer powinien odbierać, rozpoznawać i reagować na kolejne polecenia. Błąd rozpoznania polecenia albo błąd odbioru przerywa pętlę.

Jak już wspomniałem, wszystkie funkcje sieciowe są zadeklarowane w pliku *winsock2.h* — plik ten należy włączyć do projektu, inaczej pojawią się błędy komplikacji. Wystarczy odnaleźć w pliku kodu źródłowego programu następujący wiersz:

```
#include "stdafx.h"

I dodać za nim wiersz:
#include <winsock2.h>
```

Aby skompilować projekt bez błędów, trzeba by jeszcze dodać do niego bibliotekę *ws2_32.lib*. W tym celu należy w przeglądarce projektu (*Solution Explorer*) kliknąć prawym przyciskiem myszy nazwę projektu i z menu podręcznego wybrać polecenie *Properties*.

W wyświetlonym oknie trzeba przejść do *Configuration Properties/Linker/Input* i w polu *Additional Dependencies* wpisać *ws2_32.lib*.

I to już wszystko! Po uruchomieniu programu na ekranie powinny zostać wyświetlane dwa komunikaty: „Serwer zainstalowany” i „Nasłuch rozpoczęty”. Jeśli zobaczysz te komunikaty, będziesz miał pewność, że serwer pracuje poprawnie i oczekuje na połączenia klientów.



Kod źródłowy tego przykładu znajduje się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział4\TCPserver.

Nie uzyskamy jednak ostatecznej pewności co do poprawności działania serwera, jeśli nie napiszemy programu klienta, który będzie się z nim mógł połączyć i przesyłać odpowiednie polecenie. Zajmiemy się tym w następnym punkcie.

4.7.2. Przykładowy klient TCP

Serwer jest gotowy do obsługi połączeń, pora więc przymierzyć się do programowania klienta. Utwórzmy nowy projekt typu *Win32 Project* o nazwie *TCPClient*.

Odszukaj w pliku kodu źródłowego funkcję *_tWinMain* i umieść przed główną pętlą komunikatów poniższy kod:

```
WSADATA      wsd;
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    MessageBox(0, "Nie można wczytać biblioteki WinSock", "Błąd", 0);
    return 0;
}

HANDLE      hNetThread;
DWORD       dwNetThreadId;
hNetThread = CreateThread(NULL, 0, NetThread, 0, 0, &dwNetThreadId);
```

Kod ten wczytuje bibliotekę gniazd (w wersji 2.2). W tym przykładzie wykorzystywane są co prawda wyłącznie funkcje pierwotnej wersji biblioteki, więc wystarczające byłoby wczytanie pierwszej wersji WinSock. Jednak w programach przykładowych wolę — mimo ograniczania się do starych wywołań — stosować nowszą implementację biblioteki, licząc na to, że nowsze wersje eliminują ewentualne błędy wykryte w międzyczasie w poprzednich wersjach bibliotek.

Tak jak w przypadku serwera, do obsługi samego połączenia sieciowego tworzony jest osobny wątek — dla klienta zupełnie wystarczający będzie jednak wątek pojedynczy. Jest on tworzony wywołaniem funkcji *CreateThread*, w którym trzeci parametr to nazwa funkcji (*NetThread*), która ma być uruchomiona do wspólnie wykonania. Nie znamy jeszcze kodu tej funkcji, więc do robót. Uzupełnij plik kodu źródłowego programu klienta o kod z listingu 4.13, umieszczając go przed funkcją *_tWinMain*.

Listing 4.13. Wątek obsługi połączenia

```
DWORD WINAPI NetThread(LPVOID lpParam)
{
    SOCKET      sClient;
    char        szBuffer[1024];
    int         ret, i;
    struct sockaddr_in server;
    struct hostent *host = NULL;
    char szServerName[1024], szMessage[1024];
```

```

strcpy(szMessage, "GET");
strcpy(szServerName, "127.0.0.1");

// Utwórz gniazdo
sClient = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sClient == INVALID_SOCKET)
{
    MessageBox(0, "Nie można utworzyć gniazda", "Błąd", 0);
    return 1;
}

// Wypełnij strukturę adresem serwera i numerem portu
server.sin_family = AF_INET;
server.sin_port = htons(5050);
server.sin_addr.s_addr = inet_addr(szServerName);

// Jeśli adres został podany w postaci nazwy, wykonaj translację na adres IP
if (server.sin_addr.s_addr == INADDR_NONE)
{
    host = gethostbyname(szServerName);
    if (host == NULL)
    {
        MessageBox(0, "Nie można określić adresu serwera", "Błąd", 0);
        return 1;
    }
    CopyMemory(&server.sin_addr, host->h_addr_list[0],
               host->h_length);
}

// Nawiąż połączenie z serwerem
if (connect(sClient, (struct sockaddr *)&server,
            sizeof(server)) == SOCKET_ERROR)
{
    MessageBox(0, "Błąd nawiązywania połączenia", "Błąd", 0);
    return 1;
}

// Wyślij polecenie
ret = send(sClient, szMessage, strlen(szMessage), 0);
if (ret == SOCKET_ERROR)
{
    MessageBox(0, "Błąd wysyłania", "Błąd", 0);
}

// Oczekaj...
Sleep(1000);

// Odbierz odpowiedź
char szRecvBuff[1024];
ret = recv(sClient, szRecvBuff, 1024, 0);
if (ret == SOCKET_ERROR)
{
    MessageBox(0, "Błąd odbioru", "Błąd", 0);
}
MessageBox(0, szRecvBuff, "Odebrane dane", 0);
closesocket(sClient);
}

```

Warto przyjrzeć się bliżej temu programowi. Ciąg polecenia, który ma być przesłany do serwera, przechowuje zmienna szMessage. W naszym przykładzie jest to ciąg „GET”. Adres serwera zapisany jest w zmiennej szServerName. Tutaj jest to adres 127.0.0.1, czyli adres komputera lokalnego. Oznacza to, że klient i serwer działają na tym samym komputerze. Następnie — tak jak w poprzednim przykładzie — tworzone jest gniazdo.

Kolejna faza wykonania programu polega na wypełnieniu struktury adresowej typu sockaddr_in (o nazwie server), w której ma zostać zapisany kod rodziny protokołów, numer portu (5050, taki sam jak port nasłuchu serwera) oraz adres serwera.

W tym przykładzie adres jest już podany jako adres IP, jednak program przygotowany jest również do samodzielnego określenia adresu IP komputera zdalnego na podstawie jego nazwy domenowej. Właśnie po to adres jest porównywany ze stałą INADDR_NONE:

```
if (server.sin_addr.s_addr == INADDR_NONE)
```

Jeśli powyższy warunek jest spełniony, adres został podany w postaci symbolicznej i powinien zostać przekonwertowany na adres IP wywołaniem funkcji gethostbyname. Efekt wywołania tej funkcji jest zapisywany w zmiennej (host) typu hostent. Wcześniej nadmieniałem, że komputer może mieć więcej niż jeden adres IP. W takim przypadku wynikiem wywołania będzie tablica struktur typu hostent. Dla uproszczenia kodu programu odwołujemy się po prostu do pierwszego elementu tej tablicy, indeksując ją tak: host->h_addr_list[0].

Wszystko jest już gotowe do nawiązania połączenia z serwerem. Samo połączenie zainicjuje oczywiście funkcja connect. Jej parametrami będą nowo utworzone gniazdo, struktura zawierająca adres serwera i rozmiar tejże struktury. Jeśli funkcja zwróci wartość inną niż SOCKET_ERROR, połączenie można uznać za otwarte. W przeciwnym razie mamy do czynienia z błędem połączenia.

W dalszej kolejności inicjujemy transmisję danych do serwera, wywołując w tym celu funkcję send. Po wysłaniu danych spodziewamy się jakiejś reakcji ze strony serwera. Nie należy jednak oczekwać tej reakcji natychmiast, ponieważ sama transmisja (przemieszczenie danych pomiędzy buforami obu programów i buforami systemowymi) oraz reakcja serwera może trochę potrwać. Gdybyśmy wywołali od razu funkcję recv, najprawdopodobniej otrzymalibyśmy kod błędu, ponieważ odpowiedź jeszcze nie nadeszła. Dlatego po funkcji send program jest na sekundę zawieszany wywołaniem Sleep.

W „prawdziwym” (a nie przykładowym) programie nie trzeba wprowadzać opóźnienia. Zamiast niego można uruchomić pętlę aktywnego oczekiwania na odpowiedź, wywołując w niej funkcję recv aż do momentu odebrania danych. Byłaby to metoda znacznie bardziej niezawodna.

Podobnie jak program serwera, program klienta wymaga włączenia do kodu źródłowego pliku nagłówkowego *winsock2.h* i włączenia do projektu biblioteki *ws2_32.lib* — bez tych elementów projektu nie da się skompilować.

 Kod źródłowy tego przykładu znajduje się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział4\TCPClient.

4.7.3. Analiza przykładów

Gdyby uczynić program serwera niewidocznym i wyposażyć go w możliwość przechwytywania haseł albo przeładowywania komputera w reakcji na polecenia nego, program stałby się typowym koniem trojańskim. Nie będziemy jednak rozbudowywać klienta zdał go w tym kierunku, ponieważ od początku moim celem nie było bynajmniej zachęcanie do włamań komputerowych. Wszystkie przykłady prezentowane w tej książce mają charakter wyłącznie edukacyjny i tego się trzymajmy.

Warto też zwrócić uwagę na fakt kontrolowania stanu wykonania każdej funkcji sieciowej. Jest to o tyle uzasadnione, że powodzenie kolejnych operacji zależne jest od wyników poprzednich — nie ma sensu instalować serwera, jeśli nie uda się utworzyć gniazda.

W jaki sposób można by zwiększyć elastyczność kodu obu ostatnich przykładów? Są one bowiem obarczone wadą polegającą na tym, że jeśli użytkownik zechce przesyłać większe ilości danych, to albo nie zostaną one przesłane wcale, albo nie uda się ich przesyłać w całości. Są w tym, że dane są transmitowane w pakietach, a pojemność systemowych buforów nadawczych i odbiorczych jest ograniczona.

Przypuśćmy, że po stronie serwera mamy do dyspozycji bufor nadawczy o rozmiarze 64 kB. Jeśli spróbujemy jednorazowo przesłać więcej niż 64 kB danych, do klienta dotrze jedynie pierwsze 64 kB — dlatego po wysłaniu danych należy sprawdzać liczbę wysłanych bajtów i odpowiednio do tej liczby reagować (np. powtarzając wysyłanie reszty).

Przykład tak uelastycznionego kodu prezentowany jest na listingu 4.14. Algorytm wysyłania jest tu całkiem prosty, pozwoli sobie jednak poświęcić trochę czasu na jego dokładne omówienie.

Listing 4.14. Algorytm wysyłania większych ilości danych

```
char szBuf[4096];
szBuff = "Dane do wysłania...";
int nSendSize = sizeof(szBuff);
int iCurrPos = 0;

while (nSendSize > 0)
{
    ret = send(sock, &szBuff[iCurrPos], nSendSize, 0);
    if (ret == 0)
        break;
    else if (ret == SOCKET_ERROR)
    {
        // Wykryty błąd wysyłania
        MessageBox(0, "Błąd wysyłania", "Błąd", 0);
        break;
    }
    nSendSize -= ret;
    iCurrPos += ret;
};
```

W tym kodzie zmienna nSendSize kontroluje rozmiar danych do wysłania. Zmienna ta wykorzystywana jest w pętli kontynuowanej dopóki zmienna ta ma wartość większą od zera, czyli dopóki nie uda się wysłać całości danych. Zmienna iCurrPos wskazuje z kolei pozycję w buforze danych do wysłania, od której należy kontynuować wysyłanie. Pierwotnie ma ona wartość zero.

Drugi parametr funkcji send to wskaźnik bufora danych, a liczba w nawiasach kwadratowych to przesunięcie w buforze, od którego funkcja powinna zacząć pobieranie danych.

Funkcja ta zwraca liczbę wysłanych bajtów. Po sprawdzeniu wartości zwracanej należy odpowiednio zmniejszyć zmienną określającą liczbę bajtów do wysłania i zwiększyć przesunięcie bieżącej pozycji w buforze danych.

Jeśli nie uda się jednorazowo przesłać całości danych, funkcja podejmie próbę przesłania reszty w następnym przebiegu pętli.

Podobnie jest po stronie klienta, który również nie może niekiedy odebrać jednorazowo całości oczekiwanych danych. Dlatego program klienta powinien również uruchamiać pętlę odbiorczą. Jednak tutaj sprawa jest o tyle bardziej skomplikowana, że klient nie wie z góry, jakiego rozmiaru danych oczekivać i nie wiadomo, jak miałyby skonstruować warunek zakończenia pętli.

Na szczęście to całkiem proste. Zanim program zacznie wysyłanie danych powinien poinformować drugą stronę transmisji o rozmiarze danych do wysłania. Trzeba więc na potrzeby komunikacji zdefiniować najprostszy choćby protokół. Można, na przykład, uzupełnić polecenie „GET” kilkoma bajtami niosącymi właśnie informacje o ilości oczekiwanych danych, a przed właściwą transmisją danych można by zmusić serwer do oczekiwania na polecenie, np. „DATA”. Pozwoliłoby to obu stronom komunikacji na uzgodnienie ilości przesyłanych danych. Kod odbioru większych ilości danych po stronie klienta prezentowałby się wtedy następująco:

Listing 4.15. Algorytm odbierania większych ilości danych

```
char szBuf[4096];
int nSendSize = 1000000; // uzgodniony wcześniej rozmiar danych
int iCurrPos = 0;

while (nSendSize > 0)
{
    ret = recv(sock, &szBuf[iCurrPos], nSendSize, 0);
    if (ret == 0)
        break;
    else if (ret == SOCKET_ERROR)
    {
        // Wykryty błąd odbioru
        MessageBox(0, "Błąd odbioru", "Błąd", 0);
        break;
    }
    nSendSize -= ret;
    iCurrPos += ret;
};
```

4.8. Przykłady wykorzystania protokołu UDP

Jak już Czytelnikowi wiadomo, korzystanie z protokołu UDP różni się nieco od korzystania z protokołu TCP, opisywanego w ostatnim podrozdziale. Brak połączenia pomiędzy stronami komunikacji upraszcza sprawę i zwalnia z obowiązku wywoływania kilku funkcji sieciowych.

Funkcje niezbędne do obsługi transmisji na bazie protokołu UDP zostały omówione w punkcie 4.6.8.

Pora na wykorzystanie zawartych tam informacji w praktyce.

4.8.1. Przykładowy serwer UDP

Utwórz nowy projekt typu *Win32 Project* i opatrz go nazwą *UDPServer*. Otwórz plik kodu źródłowego *UDPServer.cpp* i dodaj do funkcji *_tWinMain* (przed główną pętlą komunikatów) następujący kod:

```
WSADATA      wsd;
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    MessageBox(0, "Nie można wczytać biblioteki WinSock", "Błąd", 0);
    return 0;
}

HANDLE      hNetThread;
DWORD       dwNetThreadId;
hNetThread = CreateThread(NULL, 0, NetThread, 0, 0, &dwNetThreadId);
```

Tak jak w serwerze TCP konieczne jest załadowanie biblioteki gniazd i uruchomienie osobnego wątku programu, zajmującego się obsługą właściwej transmisji. Kod funkcji wątku prezentowany jest na listingu 4.16.

Listing 4.16. Funkcja wątku obsługi transmisji UDP

```
DWORD WINAPI NetThread(LPVOID lpParam)
{
    SOCKET      sServerListen;
    struct sockaddr_in localaddr,
                  clientaddr;
    int         iSize;

    sServerListen = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sServerListen == INVALID_SOCKET)
    {
        MessageBox(0, "Nie można utworzyć gniazda", "Błąd", 0);
        return 0;
    }
    localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
localaddr.sin_family = AF_INET;
localaddr.sin_port = htons(5050);

if (bind(sServerListen, (struct sockaddr *)&localaddr,
         sizeof(localaddr)) == SOCKET_ERROR)
{
    MessageBox(0, "Nie można zainstalować serwera", "Błąd", 0);
    return 1;
}

MessageBox(0, "Serwer Zainstalowany", "Uwaga", 0);

char buf[1024];

while (1)
{
    iSize = sizeof(clientaddr);
    int ret = recvfrom(sServerListen, buf, 1024, 0,
                       (struct sockaddr *)&clientaddr, &iSize);
    MessageBox(0, buf, "Uwaga", 0);
}
closesocket(sServerListen);
return 0;
```

Przy tworzeniu gniazda funkcją *socket* drugim parametrem wywołania jest wartość *SOCK_DGRAM*. Sygnalizuje on konieczność zastosowania protokołu opartego na komunikatach (datagramowego). Ostatni parametr powinien być stałą określającą dokładnie protokół transmisji. W tym przykładzie mamy zamiar komunikować się za pośrednictwem protokołu UDP, podajemy więc jawnie *IPPROTO_UDP* (jako że jest to dla protokołów datagramowych wartość domyślna, moglibyśmy przekazać zero).

Reszta kodu nie powinna być Czytelnikowi obca. Po utworzeniu gniazda serwer jest instalowany wywołaniem *bind*. Na tym kończy się instalacja serwera UDP — nie trzeba rozpoczynać nasłuchu. Następnie serwer uruchamia nieskończoną pętlę, w ramach której odbiera komunikaty klientów (funkcją *recvfrom*).

Kiedy serwer odbierze dane, wyświetli okno z odebranymi danymi. Adres nadawcy komunikatu zapisywany jest w zmiennej *clientaddr*, serwer może więc odpowiedzieć klientowi, przesyłając do niego stosowny komunikat.



Kod źródłowy tego przykładu znajduje się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział4\UDPServer.

4.8.2. Przykładowy klient UDP

W tym punkcie zajmiemy się programem klienta wysyłającym dane do serwera UDP. Utwórz więc nowy projekt typu *Win32 Project* i nazwij go *UDPClient*. Tym razem możemy się obejść bez osobnych wątków wykonania programu i rozpoczęć wysyłanie danych od razu w funkcji *_tWinMain*. Wysyłanie danych za pośrednictwem protokołu UDP

nie zawiesza bowiem programu do czasu otrzymania potwierdzenia odbioru, więc program będzie — mimo obsługi operacji sieciowych — reagował na komunikaty systemowe. Z tego względu nie ma potrzeby tworzenia aplikacji wielowątkowej.

Otwórz plik kodu źródłowego *UDPCClient.cpp* i w funkcji *_tWinMain*, gdzieś przed główną pętlą komunikatów, umieść kod z listingu 4.17.

Listing 4.17. Wysyłanie danych do serwera — protokół UDP

```
WSADATA      wsd;
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    MessageBox(0, "Nie można wczytać biblioteki WinSock", "Błąd", 0);
    return 0;
}

SOCKET      sSocket;
struct sockaddr_in servaddr;
char szServerName[1024], szMessage[1024];
struct hostent *host = NULL;

strcpy(szMessage, "To jest komunikat od klienta");
strcpy(szServerName, "127.0.0.1");

sSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (sSocket == INVALID_SOCKET)
{
    MessageBox(0, "Nie można utworzyć gniazda", "Błąd", 0);
    return 0;
}
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(5050);
servaddr.sin_addr.s_addr = inet_addr(szServerName);

if (servaddr.sin_addr.s_addr == INADDR_NONE)
{
    host = gethostbyname(szServerName);
    if (host == NULL)
    {
        MessageBox(0, "Nie można określić adresu serwera", "Błąd", 0);
        return 1;
    }
    CopyMemory(&servaddr.sin_addr, host->h_addr_list[0],
               host->h_length);
}

sendto(sSocket, szMessage, 30, 0, (struct sockaddr *)&servaddr,
       sizeof(servaddr));
```

Tak jak w przypadku serwera UDP, przy tworzeniu gniazda drugi parametr wywołania funkcji *socket* ustawiany jest na *SOCK_DGRAM*, a protokół komunikacji określany jest jako *IPPROTO_UDP*.

Następnie program wypełnia strukturę adresową, wpisując do niej adres serwera i numer portu transmisji. Jeśli adres serwera zostanie podany symbolicznie, program skonwertuje go do postaci adresu IP, tak samo jak w programie klienta TCP.

Teraz można rozpoczęć przesyłanie danych, wywołując od razu — bez nawiązywania połączenia z serwerem — funkcję *sendto*. W naszym przykładzie do serwera wysyłany jest ciąg zapisany w zmiennej *szMessage*.

Jak poprzednio komplikacja programu uzależniona jest od włączenia do kodu programu odpowiednich plików nagłówkowych i włączenia do projektu pliku biblioteki *ws2_32.lib* (jak w punkcie 4.7.1).



Kod źródłowy tego przykładu znajduje się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział4\UDPCClient.

4.9. Przetwarzanie odebranych danych

Wszystkie dane odebrane z sieci należy poddać starannej weryfikacji. Jeśli dostęp do pewnych funkcji serwera powinien być kontrolowany hasłem, zalecałbym na samym początku sprawdzać uprawnienia klienta do wykonania polecenia. Dopiero potem należy sprawdzić poprawność samego polecenia i jego ewentualnych parametrów.

Przypuśćmy, że klient zażądał przesłania z serwera określonego pliku. Jeśli najpierw sprawdzimy ścieżkę dostępu do pliku i nazwę pliku i w przypadku jego braku odeślemy do klienta (nieautoryzowanego jeszcze) komunikat o braku pliku, haker będzie wiedział już choćby to, że w danym systemie nie istnieje konkretny plik. Niekiedy zaś taka informacja może ułatwić mu próby włamania do systemu. Dlatego zawsze na samym początku wymiany danych należy potwierdzić uprawnienia klienta i dopiero potem weryfikować poprawność wydawanych przez niego poleceń.

Jeśli to możliwe, należy zaimplementować po stronie serwera jak najściślejsze kontrole poprawności komunikacji. Na przykład dla polecenia o składni „GET nazwapliku” należy skrupulatnie sprawdzać, czy pierwsze trzy litery polecenia układają się w słowo „GET”. Nie można ograniczać się do wyszukania w poleceniu podciągu „GET”, ponieważ pozwoli to na podszywanie serwerowi niepoprawnych, spreparowanych danych. Większość włamań do systemów komputerowych udaje się właśnie dzięki niewłaściwej analizie odpowiednio spreparowanych danych.

Opracowując protokół wymiany poleceń pomiędzy klientem a serwerem, należy zawsze pamiętać o pierwszeństwie nazwy polecenia przed jego parametrami. Weźmy na przykład polecenie „GET”. Mogłoby ono (hipotetycznie) działać w dwóch trybach:

- ♦ pobrania pliku z serwera (GET nazwapliku FROM adres),
- ♦ pobrania pliku od połączonego z serwerem klienta (GET nazwapliku).

Pierwsza postać polecenia jest niebezpieczna. Aby wyszukać słowo kluczowe FROM, trzeba przeszukać cały串 polecenia. Wszystkie słowa kluczowe poleceń powinny znajdować się na ścisłe określonych pozycjach komunikatu. W naszym przykładzie pierwszą wersję polecenia należałoby raczej zdefiniować jako:

GET FROM nazwapliku adres

Tym razem wszystkie słowa kluczowe mają swoje pozycje i można łatwo sprawdzić ich obecność. Znacznie utrudni to hakerowi przemykanie niepoprawnych parametrów poleceń.

Jeśli przesyłane dane mają zmienne rozmiary, ale powinny pasować do pewnego wzorca, warto kontrolować ich zgodność z owym wzorcem. Wymaga to dodatkowych czynności kontrolnych, ale pozwala na dodatkowe zabezpieczenie serwera przed próbami ataków. Programiści często niepoprawnie kontrolują zgodność ze wzorcem. Im bardziej skomplikowany sam wzorzec i sposób kontrolowania zgodności, tym trudniej uwzględnić podczas kontroli wszystkie ograniczenia, jakie protokół nakłada na przesyłane dane. Zanim więc program serwera zostanie wdrożony do działania, należy jak najwięcej czasu poświęcić na testowanie tej właśnie części kodu. Byłoby najlepiej, gdyby jej skuteczność przetestowała osoba trzecia — w przeciwieństwie do programisty, który jako twórca protokołu sam sobie podświadomy narzuca (jako oczywiste) ograniczenia jego testowania, użytkownicy postronni potrafią wprowadzać do programu takie dane, które twórcy programu nie przyszłyby na myśl.

Zadanie kontroli danych otrzymywanych siecią komplikuje się jeszcze bardziej, kiedy dane te sterują odwołaniami do systemu plików. W efekcie może dojść do uzyskania przez klienta nieautoryzowanego dostępu do dysku ze wszystkimi tego konsekwencjami. Kiedy parametrem polecenia jest ścieżka dostępu, łatwo dopasować ją do wzorca, ale również łatwo o pomyłkę. Większość programistów ogranicza się do sprawdzania początku ścieżki — to typowy błąd.

Załóżmy, że jedynym udostępnionym przez serwer folderem na dysku C jest folder *interpub*. Jeśli w odwołaniach do tego folderu ograniczymy się do sprawdzania jedynie początku ścieżki, intruz może przekazać ścieżkę w następującej postaci:

c:\interpub\..\winnt\system32\cmd.exe

Dwie niewinne z pozoru kropki w ścieżce dostępu pozwalają na wyjście w hierarchii folderów na zewnątrz folderu *interpub* i odwołanie się do dowolnego pliku na dysku C, z plikami systemowymi włącznie.

Zanim więc oprogramujesz kontrolę parametrów przez dopasowanie ich do wzorca, powinieneś zastanowić się nad wszystkimi wartościami wyjątkowymi. Nie wolno też zapominać o konieczności wyczerpującego testowania poprawności kontroli przez przesyłanie do serwera najróżniejszych, również zupełnie nielogicznych z punktu widzenia protokołu danych. Świętne do takich testów nadają się użytkownicy, zwłaszcza ci niedoświadczeni, oraz hakerzy. Ci pierwsi naruszają wzorce z powodu niewiedzy, ci drudzy podobnie niespodziewane efekty osiągają sprytem i wiedzą.

4.10. Wysyłanie i odbieranie danych

Znamy już podstawy teoretyczne wymiany danych pomiędzy dwoma komputerami, po-dodaliśmy też implementacji przykładów praktycznych. Szanujący się haker nie poprzestałby na tym, dając do jak najpełniejszego poznania wszystkich możliwych metod i trybów wymiany danych. Otóż gniazda mają dwa tryby robocze i warto wiedzieć, jak je wykorzystywać, aby zwiększyć efektywność komunikacji i szybkość wykonywania programów.

Wymiana danych za pośrednictwem sieci i gniazd może odbywać się w dwóch trybach:

- ◆ Synchronicznym (blokującym) — wywołanie funkcji nadawczej blokuje program wysyłający do czasu zakończenia operacji.
- ◆ Asynchronicznym (nieblokującym) — wywołanie funkcji nadawczej nie blokuje programu wysyłającego, który nie musi biernie oczekивать na zakończenie operacji.

Pojęcie operacji asynchronicznych wprowadziłem w punktach 4.6.5 i 4.6.6, przy okazji omawiania funkcji sieciowych. Z operacjami asynchronicznymi warto jednak zapoznać się bliżej, ponieważ pozwalały na przyspieszenie wykonania programów i zwiększenie efektywności wykorzystania zasobów systemu.

Przez domniemanie przy tworzeniu gniazd wybierany jest tryb blokujący (synchroniczny). Z tego względu we wszystkich dotychczasowych przykładach (dla ich uproszczenia) wykorzystywany był właśnie tryb blokujący. W takim układzie zachowanie reaktywności programu na komunikaty systemowe wymagało utworzenia na potrzeby komunikacji osobnych wątków wykonania.

Jednak utrata reaktywności okna programu na czas operacji sieciowych nie jest największym problemem. Ważniejsze jest raczej bezpieczeństwo programów, które nie idzie w parze z ich prostotą. Założymy, na przykład, że wywołaliśmy funkcję *recv*, ale z jakichś względów funkcja ta nie zwróciła żadnych danych. W takim przypadku program zostałaby zablokowany na dobre, a serwer przestałby odpowiadać na polecenia klientów. Aby uniknąć takich sytuacji, stosuje się zwykle podgląd odebranych danych w buforze systemowym realizowany za pośrednictwem funkcji *recv* ze znacznikiem *MSG_PEEK*. Wiemy jednak, że metoda ta jest o tyle ryzykowna, że naraża nas na wielokrotne przetwarzanie tych samych danych albo utratę części danych. Ponadto metoda ta przeciąża system niepotrzebnymi testami obecności danych w buforach odbiorczych.

Programowanie gniazd nieblokujących jest trudniejsze, ale za to eliminuje opisane zagrożenia. Aby przełączyć gniazdo w tryb asynchroniczny, należy skorzystać z funkcji *ioctlsocket*:

```
int ioctlsocket(
    SOCKET s,
    long cmd,
    u_long FAR * argp
);
```

Funkcja ta przyjmuje trzy parametry:

- ◆ gniazdo, którego tryb ma zostać zmieniony,
- ◆ polecenie do wykonania,
- ◆ parametr polecenia.

Zmiana trybu polega na wykonaniu odnośnie gniazda polecenia kodowanego stałą FIONBIO. Jeśli jej parametr ma wartość zero, gniazdo jest przełączane do trybu blokującego. Dla innych wartości gniazdo przełączane jest do trybu nieblokującego.

Oto jak można utworzyć gniazdo i przełączyć je do działania w trybie nieblokującym:

```
SOCKET s;
unsigned long ulMode;

s = socket(AF_INET, SOCK_STREAM, 0);
ulMode = 1;
ioctlsocket(s, FIONBIO, (unsigned long*)&ulMode);
```

Od tego momentu wszystkie funkcje wysyłające i odbierające dane za pośrednictwem gniazda będą zwracały kody błędów. To całkiem normalne i należy wziąć to pod uwagę przy tworzeniu aplikacji działającej w trybie nieblokującym. Są w tym, że mimo że funkcja zwraca WSAEWOULDBLOCK, nie oznacza to błędnej transmisji danych. Wszystko jest w najlepszym porządku. W obliczu prawdziwego błędu zwrócony zostałby raczej kod inny niż WSAEWOULDBLOCK.

W trybie asynchronicznym (nieblokującym) funkcja recv nie oczekuje na nadanie danych, od razu zwracając WSAEWOULDBLOCK. Skąd więc będzie wiadomo, że do portu napłynęły jakieś dane? Można uruchomić pętlę powtarzającą wywołanie funkcji recv aż do momentu zwrócenia przez nią danych, nie jest to jednak najlepszy pomysł, ponieważ daje efekt podobny do wywołania recv w trybie blokującym, a dodatkowo program (z racji wykonywania pętli) przeciąża procesor.

Co prawda pomiędzy kolejnymi wywołaniami recv można realizować inne użyteczne z punktu widzenia programu operacje, co zwiększy nieco efektywność wykorzystania procesora. Dajmy jednak spokój tej metodzie, znam bowiem znacznie lepszą.

4.10.1. Funkcja select

Od pierwszej już wersji biblioteka gniazd udostępnia ciekawą funkcję pozwalającą na kontrolowanie stanu gniazd. Mowa o funkcji select:

```
int select(
    int nfds,
    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout
);
```

Funkcja zwraca liczbę uchwytów gniazd gotowych do użycia. Oto jej parametry:

- ◆ nfds — ignorowany (obecny wyłącznie gwoli zgodności z implementacją gniazd w wydaniu Berkeley).
- ◆ readfds — opcja odczytu (wskaźnik struktury typu fd_set).
- ◆ writefds — opcja zapisu (wskaźnik struktury typu fd_set).
- ◆ exceptfds — priorytet komunikatu (wskaźnik struktury typu fd_set).
- ◆ timeout — limit czasu oczekiwania (dla wartości NULL oczekивание bez limitu czasu).

Struktura fd_set to zestaw gniazd, które mogą przesyłać do aplikacji zezwolenie na wykonanie danej operacji. Jeśli, na przykład, zachodzi potrzeba oczekiwania na napływ danych do jednego z dwóch gniazd, należy postąpić następująco:

1. Dodać oba utworzone zawsze gniazda do struktury fd_set.
2. Wywołać funkcję select i przekazać wskaźnik struktury drugim parametrem wywołania.

Funkcja select będzie przez określony ostatnim parametrem czas oczekiwania, po czym można będzie przystąpić do odczytu danych. Jednak może się okazać, że tylko jedno z dwóch gniazd odebrało dane. Skąd mamy wiedzieć, które to gniazdo? Przede wszystkim należy sprawdzić, czy dane gniazdo wchodzi w skład obserwowanego przez select zestawu. Służy do tego funkcja FD_ISSET.

W obsłudze struktur typu fd_set przydają się następujące funkcje:

- ◆ FD_ZERO — czyści zestaw. Funkcję tę należy wywoływać przed dodaniem nowych gniazd, w ramach inicjalizacji zestawu. Przyjmuje ona tylko jeden parametr — wskaźnik zmiennej typu fd_set.
- ◆ FD_SET — dodaje gniazdo do zestawu. Przyjmuje dwa parametry: gniazdo i wskaźnik zmiennej typu fd_set.
- ◆ FD_CLR — usuwa gniazdo z zestawu. Przyjmuje dwa parametry: gniazdo i wskaźnik zmiennej typu fd_set.
- ◆ FD_ISSET — sprawdza, czy określone gniazdo przekazane pierwszym parametrem wywołania wchodzi w skład zestawu wskazywanego drugim parametrem.

4.10.2. Prosty przykład stosowania funkcji select

Spróbujmy praktycznie wykorzystać informacje z poprzedniego punktu. Otwórz projekt *TCPServer* (z punktu 4.7.1) i dodaj za miejscem utworzenia gniazda następujący fragment kodu:

```
ULONG ulBlock;
ulBlock = 1;
if (ioctlsocket(sServerListen, FIONBIO, &ulBlock) == SOCKET_ERROR)
```

```

    {
        return 0;
    }
}

```

Przełączyc to gniazdo w tryb asynchroniczny. Spróbuj uruchomić przykład. Powinieneś zobaczyć dwa komunikaty: „Serwer zainstalowany” i „Nasłuch rozpoczęty”, a następnie komunikat „Błąd funkcji accept”. W trybie asynchronicznym funkcja accept nie może blokować programu, nie może więc również oczekiwania naadejście połączenia. Jeśli w chwili jej wywołania kolejka żądań nawiązania połączenia jest pusta, funkcja kończy działanie z kodem WSAEWOULDBLOCK.

Aby rozwiązać ten problem, musimy zmienić pętlę oczekiwania na połączenie (nieukończoną pętlę while rozpoczynaną za wywołaniem funkcji listen). Nieblokująca wersja programu powinna w tym zakresie prezentować się tak jak na listingu 4.18.

Listing 4.18. Pętla oczekiwania na połączenie

```

FD_SET ReadSet;
int ReadySock;

while (1)
{
    FD_ZERO(&ReadSet);
    FD_SET(sServerListen, &ReadSet);

    if ((ReadySock = select(0, &ReadSet, NULL, NULL, NULL)) == SOCKET_ERROR)
    {
        MessageBox(0, "Błąd funkcji select", "Błąd", 0);
    }
    if (FD_ISSET(sServerListen, &ReadSet))
    {
        iSize = sizeof(clientaddr);
        sClient = accept(sServerListen, (struct sockaddr *)&clientaddr,
                         &iSize);
        if (sClient == INVALID_SOCKET)
        {
            MessageBox(0, "Błąd funkcji accept", "Błąd", 0);
            break;
        }

        hThread = CreateThread(NULL, 0, ClientThread,
                               (LPVOID)sClient, 0, &dwThreadId);
        if (hThread == NULL)
        {
            MessageBox(0, "Błąd utworzenia wątku", "Błąd", 0);
            break;
        }
        CloseHandle(hThread);
    }
}

```

Przed rozpoczęciem pętli deklarowane są dwie zmienne: ReadSet typu FD_SET, przechowującą docelowo zestaw gniazd, oraz ReadySock typu int, do przechowywania liczby gniazd gotowych do użycia (zwracanej przez funkcję select). Na razie mamy do dyspozycji tylko jedno gniazdo, więc nie będziemy korzystać z wartości drugiej zmiennej.

Na samym początku pętli zestaw gniazd jest zerowany wywołaniem FD_ZERO, następnie zaś uzupełniany utworzonym wcześniej gniazdem serwera. Zaraz potem następuje wywołanie funkcji select. Ustawiany jest tylko drugi z jej parametrów, pozostały przypisywane są wartości NULL. Drugi parametr sygnalizuje, że funkcja powinna oczekiwania na możliwość odczytu danych za pośrednictwem gniazd wchodzących w skład przekazanego zestawu. Czas oczekiwania jest nieograniczony, bo ostatni parametr wywołania ma wartość NULL.

Gniazdo serwera oczekuje na zainicjowanie połączenia przez klienta, a funkcja select poinformuje nas o momencie gotowości do odczytu danych. Kiedy w gnieździe pojawi się żądanie nawiązania połączenia, przed podjęciem jakichkolwiek czynności należało by sprawdzić, czy gniazdo aby znajduje się w zestawie gniazd gotowych do użycia; służy do tego funkcja FD_ISSET.

Reszta kodu pozostała bez zmian. Akceptujemy połączenie wywołaniem funkcji accept, tworzymy nowe gniazdo do komunikacji z klientem (w zmiennej sClient) i tworzymy nowy wątek do obsługi nawiązanego połączenia.

Uruchom przykład i sprawdź, czy działa poprawnie. Tym razem nie powinno być żadnych błędów, a program serwera powinien oczekiwania na połączenia klientów.

Czytelnik zastanawia się pewnie, w jakim trybie działa gniazdo sClient utworzone po zaakceptowaniu połączenia funkcją accept i służące do wymiany danych z klientem. Wcześniej wspomniałem, że domyślnym trybem dla gniazd jest tryb blokujący, co powinno wyjaśnić wątpliwości — po tworzeniu gniazda nie zmieniamy przecież trybu jego działania. Gdybyśmy z kodu programu klienta usunęli kod wysyłający dane i zamkający połączenie z serwerem, serwer po zaakceptowaniu połączenia „zasnąłby” w oczekiwaniu na dane. Dowodziły to niezbicie, że gniazdo wykorzystywane do komunikacji z klientem jest otwarte w trybie blokującym i że funkcja recv zawiesza wątek serwera w oczekiwaniu na dane. Mimo że gniazdo nasłuchu działa w trybie nieblokującym, gniazda tworzone na potrzeby funkcji accept są już gniazdam blokującymi.

Funkcja select pozwala na rezygnację z drugiego wątku programu serwera obsługującego właściwą wymianę danych pomiędzy serwerem a klientem. Poprzednia wersja przykładu zmuszała program serwera do tworzenia wielu wątków. Dzięki funkcji select można uprościć program i zwiększyć efektywność wykorzystania zasobów systemowych. Do zadania tego wróćmy w rozdziale 6. przy okazji omawiania kilku ciekawych algorytmów.



Kod źródłowy tego przykładu znajduje się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział4\Select.

4.10.3. Korzystanie z gniazd za pośrednictwem komunikatów systemowych

Funkcja select została włączona do biblioteki gniazd systemu Windows ze względu na zgodność z podobnymi bibliotekami wykorzystywany na innych platformach. W systemie Windows programiści mają jednak do dyspozycji funkcję jeszcze elastyczniejszą,

która pozwala na monitorowanie stanu gotowości gniazd za pośrednictwem komunikatów systemowych. Mowa o funkcji WSAAAsyncSelect. Dzięki niej nie trzeba wcale blokować programu w oczekiwaniu na pojawienie się danych na porcie, obsługując to zdarzenie w funkcji WndProc w reakcji na określony komunikat.

Funkcja prezentuje się następująco:

```
int WSAAAsyncSelect(
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent
);
```

Oto znaczenie jej parametrów:

- ◆ s — gniazdo, które ma być monitorowane.
- ◆ hWnd — okno, do którego ma być przesłany komunikat. Okno to (albo jego okno nadziedne) powinno dysponować funkcją obsługi komunikatów WndProc.
- ◆ wMsg — komunikat przesyłany do okna. Jego typ determinuje rodzaj zdarzenia sieciowego.
- ◆ lEvents — maska bitowa interesującego nas zdarzenia sieciowego. Parametr ten może być kombinacją następujących wartości:
 - ◆ FD_READ — gotowość do odczytu,
 - ◆ FD_WRITE — gotowość do zapisu,
 - ◆ FD_OOB — odbiór danych pozapasmowych,
 - ◆ FD_ACCEPT — przyjęcie połączenia klienta,
 - ◆ FD_CONNECT — połączenie z serwerem,
 - ◆ FD_CLOSE — zamknięcie połączenia,
 - ◆ FD_QOS — zmiana poziomu obsługi QoS (ang. *Quality of Service*),
 - ◆ FD_GROUP_QOS — zmiana grupy poziomu obsługi QoS.

Jeśli wykonanie funkcji przebiegnie pomyślnie, zwróci wartość większą od zera; w innym przypadku wartością zwracaną będzie SOCKET_ERROR.

Funkcja WSAAAsyncSelect automatycznie przełącza gniazdo w tryb nieblokujący, nie trzeba więc samodzielnie wywoływać na rzecz gniazda funkcji ioctlsocket.

Oto prosty przykład zastosowania funkcji WSAAAsyncSelect:

```
WSAAAsyncSelect(s, hWnd, wMsg, FD_READ | RD_WRITE);
```

Po wykonaniu powyższego wiersza okno wskazywane uchwytem hWnd za każdym razem, kiedy gniazdo będzie gotowe do odbioru danych albo ich nadawania, będzie otrzymywać komunikat wMsg. Aby odwołać monitorowanie gniazda, wystarczy wywołać tę samą funkcję z zerem w roli czwartego parametru:

```
WSAAAsyncSelect(s, hWnd, 0, 0);
```

W tym przypadku ważne jest prawidłowe określenie dwóch pierwszych parametrów i przekazania zera za pośrednictwem ostatniego parametru. Wartość третьego parametru nie ma większego znaczenia — po zakończeniu funkcji do okna nie będą wysyłane żadne komunikaty zdarzeń sieciowych — można więc ustawić go na zero. W przypadku chęci zmiany rodzaju zdarzeń inicjujących wysyłanie komunikatów należałoby po prostu wywołać tę samą funkcję z odpowiednio dobraną wartością czwartego parametru. Warto przy tym wcześniej przekazać tym parametrem zero, żeby wyzerować kryteria monitorowania gniazda.

Wszystkie zdarzenia pojedynczego gniazda są sygnaлизowane jednym typem komunikatu. Innymi słowy, nie można zdefiniować osobnych komunikatów dla zdarzenia gotowości gniazda do odczytu i dla zdarzenia gotowości tego samego gniazda do zapisu.

Zanim przejdziemy do kolejnego przykładu, powinniśmy jeszcze zastanowić się nad komunikatem przesyłanym do funkcji WndProc. Pamiętamy tę funkcję:

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

Parametry wParam i lParam niosą informacje dodatkowe i ich wartość jest zależna od rodzaju zdarzenia. W przypadku zdarzeń sieciowych parametr wParam przenosi uchwyt gniazda, którego zdarzenie dotyczy. Nie trzeba więc nawet w programie utrzymywać globalnej tablicy gniazd — odpowiednie gniazdo zostanie wskazane procedurze obsługi zdarzenia automatycznie.

Parametr lParam składa się z dwóch dwubajtowych słów. Słowo mniejsze znaczące określa rodzaj zdarzenia gniazda, a słowo bardziej znaczące koduje numer błędu. Wyposażeni w te informacje możemy przyjrzeć się wreszcie przykładowi. Utwórz nowy projekt typu *Win32 Project* o nazwie *WSASel*. Otwórz plik kodu źródłowego *WSASel.h* i wprowadź kilka zmian w funkcji _tWinMain. Jak zwykle uzupełnienie powinno znaleźć się przed główną pętlą komunikatów. Kod uzupełnienia widać na listingu 4.19.

Listing 4.19. Funkcja _tWinMain

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPTSTR lpCmdLine,
                        int nCmdShow)
{
    // TODO: Place code here.
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_WSASEL, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Perform application initialization:
```

```

if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}

hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_WSASEL);

WSADATA      wsd;
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    MessageBox(0, "Nie można wczytać biblioteki WinSock", "Błąd", 0);
    return 0;
}

SOCKET      sServerListen,
            sClient;
struct sockaddr_in localaddr,
                  clientaddr;
HANDLE       hThread;
DWORD        dwThreadId;
int          iSize;

sServerListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
if (sServerListen == SOCKET_ERROR)
{
    MessageBox(0, "Nie można utworzyć gniazda", "Błąd", 0);
    return 0;
}

ULONG ulBlock;
ulBlock = 1;
if (ioctlsocket(sServerListen, FIONBIO, &ulBlock) == SOCKET_ERROR)
{
    return 0;
}

localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
localaddr.sin_family = AF_INET;
localaddr.sin_port = htons(5050);

if (bind(sServerListen, (struct sockaddr *)&localaddr,
         sizeof(localaddr)) == SOCKET_ERROR)
{
    MessageBox(0, "Nie można zainstalować serwera", "Błąd", 0);
    return 1;
}

WSAAAsyncSelect(sServerListen, hWnd, WM_USER + 1, FD_ACCEPT);
listen(sServerListen, 4);

// Main message loop:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

```

```

    }

    closesocket(sServerListen);
    WSACleanup();

    return (int) msg.wParam;
}

```

Z racji zastosowania wywołania WSAAAsyncSelect cały kod serwera może zostać umieszczony w funkcji _tWinMain i nie ma potrzeby tworzenia odrębnych wątków programu.

Kod jest niemal identyczny jak w programie *TCPSever* (z punktu 4.7.1). Jedyna różnica polega na tym, że w miejsce funkcji listen wywoływana jest funkcja WSAAAsyncSelect przełączająca gniazdo w tryb asynchronousny. W wywoaniu przekazywane są następujące parametry:

- ◆ sServerListen — zmienna wskazująca nowo utworzone gniazdo.
- ◆ hWnd — uchwyt okna głównego programu (do którego mają być przesyłane zdarzenia z monitorowania gniazda).
- ◆ WM_USER+1 — wszystkie komunikaty do użytku własnego programisty (inne niż systemowe) powinny być kodowane wartościami nie mniejszymi od stałej WM_USER. Mniejsze wartości są zarezerwowane dla systemu, więc korzystanie z nich może prowokować konflikty. Specjalnie zastosowałem taką konstrukcję parametru, aby było to jasne. W prawdziwym kodzie zalecałbym jednak wcześniejsze zdefiniowanie dla komunikatu stałej o rozpoznawalnej nazwie, np. #define WM_NETMESSAGE WM_USER+1.
- ◆ FD_ACCEPT — kategoria zdarzeń do monitorowania. Cóż może robić gniazdo? Oczywiście przyjmować żądania połączenia z serwerem. To jedyne zdarzenie, jakie nas interesuje.

Najciekawsze jest tym razem w funkcji WndProc. Początek tej funkcji (zawierający kod obsługi zdarzenia sieciowego) prezentowany jest na listingu 4.20.

Listing 4.20. Obsługa zdarzeń sieciowych w funkcji WndProc

```

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    SOCKET ClientSocket;
    int ret;
    char szRecvBuff[1024], szSendBuff[1024];

    switch (message)
    {
        case WM_USER+1:
            switch (WSAGETSELECTEVENT(lParam))
            {

```

```

case FD_ACCEPT:
    ClientSocket = accept(wParam, 0, 0);
    WSAAAsyncSelect(ClientSocket, hWnd, WM_USER+1,
                    FD_READ | FD_WRITE | FD_CLOSE);
    break;

case FD_READ:
    ret = recv(wParam, szRecvBuff, 1024, 0);
    if (ret == 0)
        break;
    else if (ret == SOCKET_ERROR)
    {
        MessageBox(0, "Błąd odbioru danych", "Błąd", 0);
        break;
    }
    szRecvBuff[ret] = '\0';
    strcpy(szSendBuff, "Polecenie GET przyjęte");

    ret = send(wParam, szSendBuff, sizeof(szSendBuff), 0);
    break;

case FD_WRITE:
    // Gotowość do odbioru danych
    break;

case FD_CLOSE:
    closesocket(wParam);
    break;
}
case WM_COMMAND:
    ...

```

W funkcji tej dodaliśmy nową klauzulę case. Porównuje ona odebrany komunikat z wartością WM_USER+1. Jeśli porównanie wypadnie pomyślnie, funkcja rozpoczyna analizę komunikatu celem rozpoznania zdarzenia, które go spowodowało. Analiza w instrukcji switch polega na porównywaniu wartości parametru przesłanego komunikatu z kolejnymi stałymi w klauzulach case:

```
switch (WSAGETSELECTEVENT(1Param))
```

Jak już nam wiadomo, parametr 1Param reprezentuje ewentualny kod błędu i typ zdarzenia. Do wyłuskania z niego kodu zdarzenia służy wywołanie WSAGETSELECTEVENT. Następnie kod ten jest porównywany z kodami różnych obsługiwanych zdarzeń. Jeśli rozpoznane zostanie żądanie nawiązania połączenia, wykonany będzie następujący kod:

```

case FD_ACCEPT:
    ClientSocket = accept(wParam, 0, 0);
    WSAAAsyncSelect(ClientSocket, hWnd, WM_USER+1,
                    FD_READ | FD_WRITE | FD_CLOSE);
    break;

```

Na początku połączenie jest przyjmowane wywołaniem accept. Jego wynikiem jest gniazdo, które można wykorzystać do wymiany danych z klientami. Ponieważ zdarzenia tego nowego gniazda powinny również być monitorowane, ponownie wywoływana

jest metoda WSAAAsyncSelect. Aby uniknąć definiowania zbyt wielu komunikatów do własnego użytku, przydzielimy do monitorowania tego gniazda również komunikat WM_USER+1. Nie będzie to powodować żadnych konfliktów, ponieważ dla gniazda serwera funkcja WndProc obsługuje jedynie zdarzenie przyjęcia połączenia (FD_ACCEPT), a wśród zdarzeń klienta obsługiwane są zdarzenia gotowości do odbioru, wysyłania danych oraz zdarzenie zamknięcia połączenia.

Kiedy serwer odbierze dane, sterowanie zostanie przekazane do funkcji WndProc z komunikatem WM_USER+1, w której rozpoznane zostanie zdarzenie FD_READ. Dane zostają odczytane z buforów systemowych, a klientowi odesłany zostanie ciąg „Polecenie GET przyjęte”:

```

case FD_READ:
    ret = recv(wParam, szRecvBuff, 1024, 0);
    if (ret == 0)
        break;
    else if (ret == SOCKET_ERROR)
    {
        MessageBox(0, "Błąd odbioru danych", "Błąd", 0);
        break;
    }
    szRecvBuff[ret] = '\0';
    strcpy(szSendBuff, "Polecenie GET przyjęte");
    ret = send(wParam, szSendBuff, sizeof(szSendBuff), 0);
    break;

```

Taki sam kod występował w programie TCPServer w obsłudze wymiany danych pomiędzy klientem i serwerem. Celowo pozostawiłem go w jego pierwotnej postaci, aby serwer nadawał się do testowania programem TCPClient.

Reakcja na zdarzenie FD_WRITE jest nieokreślona — w kodzie występuje tam jedynie komentarz. Obsługa zdarzenia FD_CLOSE polega zaś na zamknięciu gniazda.

W przykładzie wykorzystującym funkcję select obsługiwany był tylko jeden klient. Obsługa więcej niż jednego połączenia wymagałaby utworzenia puli wątków obsługujących współbieżnie osobne połączenia i wysyłających i odbierających dane. W rozdziale 6. zaprezentowany zostanie przykład zastosowania funkcji select bez takiej puli.

Funkcja WSAAAsyncSelect łatwo stosuje się w programach, ponadto pozwala ona na obsługę większej liczby połączeń w pojedynczym wątku programu. Podstawową jej zaletą jest właśnie brak konieczności rozwidlania programu na wiele wątków.

Testowanie nowego przykładu polega na uruchomieniu w pierwszej kolejności programu WSASel, a następnie programu klienta TCPClient.



Kod źródłowy tego przykładu znajduje się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział 4\WSASel.

Chciałbym podkreślić, że tym razem również właściwa wymiana danych odbywa się asynchronicznie. Należy jednak pamiętać o konieczności podziału większych ilości danych na mniejsze porcje.

Wyobraźmy sobie, że klient zamierza przesłać do serwera 1-megabajtowy plik danych. Oczywiście nie może takiej ilości danych wysłać jednorazowo, dlatego po stronie serwera należałoby zorganizować odbiór w sposób następujący:

1. Serwer jest informowany o ilości danych do odebrania (powiadamia go o tym klient).
2. Serwer tworzy bufor i przydziela dla niego wymaganą ilość pamięci albo — jeśli ilość danych jest zbyt duża — zakłada plik tymczasowy.
3. Po rozpoznaniu zdarzenia FD_READ serwer zapisuje dane z bufora odbiorczego w buforze danych albo w pliku tymczasowym. Serwer obsługuje zdarzenia gotowości gniazda odczytu do momentu wyczerpania danych albo odebrania od klienta uzgodnionej wcześniej sekwencji bajtów sygnalizującej koniec transmisji.

Klient powinien zaś zachowywać się następująco:

1. Poinformować serwer o ilości danych do przesłania.
2. Otworzyć plik, z którego będzie odczytywały dane.
3. Wysyłać zawartość pliku w porcjach. Kolejne porcje wysyłać w reakcji na zdarzenie FD_WRITE.
4. Po wyczerpaniu zawartości pliku przesłać do serwera szczególną sekwencję bajtów.

Zaangażowanie do takiej transmisji mechanizmu komunikatów systemu Windows jest bardzo wygodne, nieważek jednak zgodność programu z systemami uniksowymi, w których mechanizmy komunikacji międzyprocesowej są realizowane odmiennie niż w Windows. Przeniesienie takiego programu na inną platformę może okazać się procesem żmudnym i wymagającym przepisania znacznych części kodu. Ponieważ jednak osobiście nie zamierzam tworzyć aplikacji wieloplatformowych, pozwalam sobie na wykorzystywanie w nich wywołań WSAAsyncSelect tak z racji wygody programowania, jak i efektywności programu.

4.10.4. Asynchroniczna wymiana danych z wykorzystaniem obiektów zdarzeń

Jeśli w programie nie masz zamiaru wykorzystywać funkcji obsługi komunikatów, możesz zorganizować sygnalizację zdarzeń za pośrednictwem specjalnych obiektów zdarzeń. W takim układzie komunikację należy zaimplementować następująco:

1. Utworzyć obiekt zdarzenia funkcją WSACreateEvent.
2. Wybrać gniazdo funkcją WSAEventSelect.
3. Zawiesić program w oczekiwaniu na zdarzenie — funkcją WSAWaitForMultipleEvents.

Przyjrzyjmy się bliżej wszystkim funkcjom zaangażowanym w taką wymianę danych.

Na początku potrzebna będzie nam funkcja WSACreateEvent tworząca obiekt zdarzenia. Nie trzeba przekazywać jej żadnych parametrów — funkcja po prostu zwraca nowy obiekt zdarzenia typu WSAEVENT:

```
WSAEVENT WSACreateEvent(void);
```

Teraz należy z tym obiektem skojarzyć gniazdo i określić interesujące nas zdarzenia, które mają być monitorowane. Służy do tego funkcja WSAEventSelect:

```
int WSAEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents
);
```

Pierwszy parametr wywołania to gniazdo, które ma być monitorowane. Drugi parametr to obiekt zdarzenia, za pośrednictwem którego sygnalizowane będą zmiany stanu gniazda. Ostatni parametr wyznacza zakres obserwowanych zdarzeń. Można za jego pośrednictwem przekazywać wartości identyczne jak w funkcji WSAAsyncSelect (czyli wszystkie stałe zaczynające się od przedrostka FD_).

Spotkaliśmy się już z funkcjami WaitForSingleObject i WaitForMultipleObject zawierającymi program w oczekiwaniu na zdarzenie typu HANDLE. Do oczekiwania na zdarzenia sieciowe wykorzystamy bardzo podobną funkcję, o nazwie WSAWaitForMultipleEvents:

```
WSAWaitForMultipleEvents(
    DWORD cEvents,
    const WSAEVENT FAR * lphEvents,
    BOOL fWaitAll,
    DWORD dwTimeout,
    BOOL fAlertable
);
```

Spójrzmy na jej parametry:

- ◆ **cEvents** — liczba obiektów zdarzeń, których zmiany mają być monitorowane. Maksymalna wartość tego parametru równa się stałej **WSA_MAXIMUM_WAIT_EVENTS**.
- ◆ **lphEvents** — tablica obiektów zdarzeń, na które funkcja ma oczekiwania.
- ◆ **fWaitAll** — tryb oczekiwania. Jeśli parametr ma wartość TRUE, funkcja będzie oczekiwania aż do zaobserwowania wszystkich zdarzeń. W przeciwnym razie sterowanie zostanie przekazane do programu po zajściu dowolnego ze zdarzeń.
- ◆ **dwTimeout** — limit czasu oczekiwania wyrażany w milisekundach. Jeśli w owym czasie nie zajdzie oczekiwane zdarzenie, funkcja zwróci **WSA_WAIT_TIMEOUT**. Jeśli powinna oczekiwania bez ograniczenia czasowego, należy przekazać tym parametrem stałą **WSA_INFINITE**.
- ◆ **fAlertable** — parametr wykorzystywany w nakładających się operacjach wejścia-wyjścia, których omówienie wykracza poza zakres tematyczny tej książki. Z tego względu przekazujemy za jego pośrednictwem wartość FALSE.

Aby określić, które ze zdarzeń wyznaczonych do monitorowania zostało zaobserwowane, należy odjąć od wartości zwracanej przez funkcję WSAWaitForMultipleEvents stałą **WSA_WAIT_EVENT_0**.

Przed przekazaniem tablicy obiektów zdarzeń do wywołania funkcji `WSAWaitForMultipleEvents` wszystkie obiekty powinny być puste. Jeśli choć jeden z nich nie jest pusty, funkcja natychmiast zwróci sterowanie do programu. Po zakończeniu działania funkcji obiekty zaobserwowanych zdarzeń będą obiektami niepustymi i po zakończeniu obsługi reprezentowanych przez nie zdarzeń należy je wyzerować. Służy do tego funkcja `WSAResetEvent`:

```
BOOL WSAResetEvent(
    WSAEVENT hEvent
);
```

Parametrem wywołania funkcji jest obiekt zdarzenia do wyzerowania.

Jeśli obiekt zdarzenia nie jest już potrzebny, należy go zamknąć. Zadanie to realizuje funkcja `WSACloseEvent`. Jej jedynym parametrem jest obiekt zdarzenia do zamknięcia:

```
BOOL WSACloseEvent(
    WSAEVENT hEvent
);
```

Jeśli zamknięcie będzie skuteczne, funkcja zwróci wartość `TRUE`. W przypadku niepowodzenia wartością zwracaną będzie zaś `FALSE`.

Rozdział 5.

Obsługa sprzętu

W niniejszym rozdziale zajmiemy się zagadnieniami związanymi ze sprzętem komputerowym. Haker powinien przecież wiedzieć, jak obsługiwac programowo sprzęt i jak odczytywać parametry jego działania.

Ponieważ w znacznej części książki poświęcona jest programowaniu aplikacji sieciowych, tematyka ta powróci również w tym rozdziale. Pisząc oprogramowanie sieciowe, często trzeba znać charakterystyki komputera lokalnego, czasem zaś trzeba umieć ją modyfikować. Pokażę więc, w jaki sposób odczytywać ustawienia robocze karty sieciowej i podsystemu sieciowego. W rozdziale padnie też odpowiedź na jakże często zadawane przez początkujących programistów pytanie o adres IP komputera lokalnego.

Dodatkowo postaram się wprowadzić Czytelnika w zagadnienia związane z korzystaniem z portów szeregowych komputera, które wciąż są wykorzystywane do połączania najrozmaitszych urządzeń. Kiedy pracowałem jako programista urządzeń automatyki przemysłowej, napisałem masę programów zbierających dane z urządzeń monitorujących działanie rozmaitych maszyn połączonych do komputera właśnie za pośrednictwem portu szeregowego.

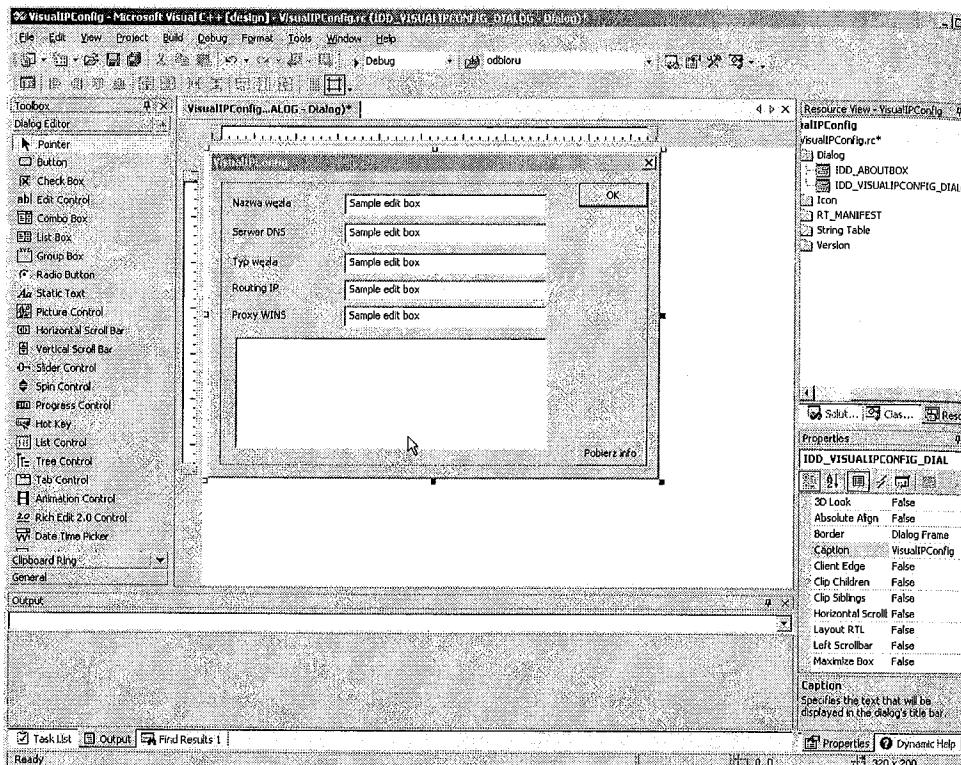
5.1. Parametry podsystemu sieciowego

W systemach z rodziny Windows 9x użytkownik ma do swojej dyspozycji bardzo wygodne narzędzie o nazwie *Konfiguracja IP* (*winipcfg*). Dzięki temu prostemu narzędziu może on łatwo określić adres IP komputera oraz adresy MAC zainstalowanych w nim kart sieciowych.

Adres MAC karty sieciowej jest niepowtarzalnym numerem tej karty i jest zapisany w jej pamięci ROM. Owa niepowtarzalność numerów kart sieciowych stała się jednym z mechanizmów zabezpieczających aplikacje sieciowe — jeśli komputer posiada na stałe pewną kartę sieciową, można rozpoznać (w sieci lokalnej) transmisje z tego komputera na podstawie ich adresu MAC.

Do manipulowania parametrami podsystemu sieciowego służy biblioteka *IPHlpApi.lib*. Spróbujmy zobaczyć, jak wykorzystać jej najciekawsze funkcje.

Utwórz w Visual C++ nowy, oparty na oknach dialogowych projekt typu *MFC Application*. Na formularzu głównego okna programu umieść pięć pól edycyjnych (*Edit Control*), jedno okno listy (*List Box*) i jeden przycisk opatrzony napisem *Pobierz info*. Okno powinno wyglądać mniej więcej tak, jak to na rysunku 5.1.



Rysunek 5.1. Okno główne programu *VisualIPConfig*

Dla pól edycyjnych utwórz następujące zmienne: dla pola opatrzonego etykietą *Nazwa węzła* zmienną *eHostName*, dla pola *Serwer DNS* zmienną *DNSServers*, dla pola *Typ węzła* zmienną *eNodeType*, dla pola *Routing IP* zmienną *eIPRouting* oraz dla pola *Proxy WINS* zmienną *eWinsProxy*.

Komputer może być wyposażony w więcej niż jedno urządzenie sieciowe, stąd potrzeba okna widoku listy, w którym owe urządzenia zostaną opisane. W polach edycyjnych będą natomiast wyświetlane informacje o interfejsie podstawowym.

Utwórz teraz procedurę obsługi zdarzenia BN_CLICKED (naciśnięcia przycisku) — w tym celu wystarczy dwukrotnie kliknąć przycisk na formularzu. Do funkcji obsługi zdarzenia wpisz kod z listingu 5.1. Zalecałbym przy tym samodzielne przepisanie kodu, bez uciekania się do skopiowania fragmentu pliku kodu źródłowego z płyty CD-ROM dołączonej do książki.

Listing 5.1. Pobieranie parametrów karty sieciowej

```
void CVisualIPConfigDlg::OnBnClickedButton1()
{
    PFIXED_INFO pFixedInfo;
    ULONG iFixedInfo = 0;

    PIP_ADAPTER_INFO pAdapterInfo, pAdapter;
    ULONG iAdapterInfo;
    PIP_ADDR_STRING chAddr;

    CString Str;
    TCHAR lpszText[1024];
    int iErr;

    if ((iErr = GetNetworkParams(NULL, &iFixedInfo)) != 0)
    {
        if (iErr != ERROR_BUFFER_OVERFLOW)
        {
            AfxMessageBox("Błąd funkcji GetNetworkParams");
            return;
        }
    }

    if ((pFixedInfo = (PFIXED_INFO)GlobalAlloc(GPTR, iFixedInfo)) == NULL)
    {
        AfxMessageBox("Błąd przydziału pamięci");
        return;
    }

    if (GetNetworkParams(pFixedInfo, &iFixedInfo) != 0)
    {
        AfxMessageBox("Błąd funkcji GetNetworkParams");
        return;
    }

    eHostName.SetWindowText(pFixedInfo->HostName);

    CString s=pFixedInfo->DnsServerListIpAddress.String;
    chAddr = pFixedInfo->DnsServerList.Next;
    while(chAddr)
    {
        s = s + " " + chAddr->IpAddress.String;
        chAddr = chAddr->Next;
    }
    DNServers.SetWindowText(s);

    switch (pFixedInfo->NodeType)
    {
        case 1:
            eNodeType.SetWindowText("Rozgłoszeniowy");
            break;
        case 2:
            eNodeType.SetWindowText("Transmisja");
            break;
        case 4:
            eNodeType.SetWindowText("Mieszany");
            break;
    }
}
```

```

        break;
    case 8:
        eNodeType.SetWindowText("Hybrydowy");
        break;
    default:
        eNodeType.SetWindowText("Nieznany");
    }

eIPRouting.SetWindowText(pFixedInfo->EnableRouting ? "Włączony" : "Wyłączony");
eWinsProxy.SetWindowText(pFixedInfo->EnableProxy ? "Włączony" : "Wyłączony");

iAdapterInfo = 0;
iErr=GetAdaptersInfo(NULL, &iAdapterInfo);
if ((iErr!= 0) && (iErr != ERROR_BUFFER_OVERFLOW))
{
    AfxMessageBox("Błąd funkcji GetAdaptersInfo");
    return;
}

if ((pAdapterInfo = (PIP_ADAPTER_INFO) GlobalAlloc(GPTR, iAdapterInfo)) == NULL)
{
    AfxMessageBox("Błąd przydziału pamięci");
    return;
}

if (GetAdaptersInfo(pAdapterInfo, &iAdapterInfo) != 0)
{
    AfxMessageBox("Błąd funkcji GetAdaptersInfo");
    return;
}

pAdapter = pAdapterInfo;
eAdaptersInfo.AddString("=====");
while (pAdapter)
{
    switch (pAdapter->Type)
    {
        case MIB_IF_TYPE_ETHERNET:
            Str = "Karta Ethernet: "; break;
        case MIB_IF_TYPE PPP:
            Str = "Karta PPP: "; break;
        case MIB_IF_TYPE_LOOPBACK:
            Str = "Sterownik pseudosieciowy: "; break;
        case MIB_IF_TYPE_TOKENRING:
            Str = "Karta Token Ring: "; break;
        case MIB_IF_TYPE_FDDI:
            Str = "Karta FDDI: "; break;
        case MIB_IF_TYPE_SLIP:
            Str = "Karta SLIP: "; break;
        case MIB_IF_TYPE_OTHER:
            default: Str = "Inna: ";
    }
    eAdaptersInfo.AddString(Str + pAdapter->AdapterName);
    Str = "Opis: ";
}

```

```

eAdaptersInfo.AddString(Str + pAdapter->Description);

Str="Adres fizyczny: ";
for (UINT i = 0; i < pAdapter->AddressLength; i++)
{
    if (i == (pAdapter->AddressLength - 1))
        sprintf(lpszText, "%.2X", (int)pAdapter->Address[i]);
    else
        sprintf(lpszText, "%.2X", (int)pAdapter->Address[i]);
    Str = Str + lpszText;
}
eAdaptersInfo.AddString(Str);

sprintf(lpszText, "DHCP: %s", (pAdapter->DhcpEnabled ? "tak" : "nie"));
eAdaptersInfo.AddString(lpszText);

chAddr = &(pAdapter->IpAddressList);
while(chAddr)
{
    Str = "Adres IP: ";
    eAdaptersInfo.AddString(Str + chAddr->IpAddress.String);

    Str = "Maska podsieci: ";
    eAdaptersInfo.AddString(Str + chAddr->IpMask.String);

    chAddr = chAddr->Next;
}

Str = "Brama domyślna: ";
eAdaptersInfo.AddString(Str + pAdapter->GatewayList.IpAddress.String);

chAddr = pAdapter->GatewayList.Next;
while(chAddr)
{
    // Wypisz następną bramę
    chAddr = chAddr->Next;
}

Str = "Serwer DHCP: ";
eAdaptersInfo.AddString(Str + pAdapter->DhcpServer.IpAddress.String);

Str = "Podstawowy serwer WINS: ";
eAdaptersInfo.AddString(Str + pAdapter->PrimaryWinsServer.IpAddress.String);

Str = "Pomocniczy serwer WINS: ";
eAdaptersInfo.AddString(Str + pAdapter->SecondaryWinsServer.IpAddress.String);

eAdaptersInfo.AddString("=====");
pAdapter = pAdapter->Next;
}

```

Do pobierania ogólnych informacji o interfejsie sieciowym służy funkcja GetNetworkParams. Przyjmuje ona dwa parametry: wskaźnik struktury typu `PFIXED_INFO` i rozmiar tej struktury.

Jeśli w miejsce pierwszego parametru podamy NULL, a drugim parametrem przekażemy adres zmiennej liczbowej (typu całkowitoliczbowego), w zmiennej tej funkcja zapisze rozmiar pamięci niezbędny do przechowywania struktury. Dokładnie tę operację wykonujemy przy okazji pierwszego wywołania funkcji GetNetworkParams. Po określeniu rozmiaru struktury opisującej parametry sieciowe przydzielimy dla niej pamięć w obszarze pamięci globalnej; ważne, aby wykorzystać do tego funkcję GlobalAlloc — inaczej możemy z funkcji GetNetworkParam uzyskać nieprawidłowe dane.

Po przydzieleniu pamięci do struktury wywołujemy GetNetworkParams po raz drugi, tym razem z właściwym parametrem. Jeśli uda się pozyskać informacje o urządzeniu sieciowym, funkcjawróci zero.

Pora na przegląd składowych struktury PFIXED_INFO:

- ◆ HostName — nazwa węzła (komputera).
- ◆ DnsServerList.IpAddress — lista adresów IP serwerów DNS.
- ◆ NodeType — typ węzła.
- ◆ EnableRouting — jeśli TRUE, dla urządzenia działa routing pakietów IP.
- ◆ EnableProxy — jeśli TRUE, urządzenia objęte jest buforowaniem pakietów.

Po uzyskaniu informacji ogólnych możemy przystąpić do pozyskiwania informacji o wszystkich zainstalowanych w systemie adapterach sieciowych. Tym razem posłużymy się funkcją GetAdaptersInfo. Również ta funkcja przyjmuje dwa parametry: zmienną typu PIP_ADAPTER_INFO i rozmiar struktury wskazywanej tą zmienną. I tym razem, jeśli pierwszy parametr ma wartość 0 (albo NULL), funkcja zwróci za pośrednictwem drugiego parametru ilość pamięci potrzebnej do przechowywania struktury PIP_ADAPTER_INFO.

Oto elementy rzeczonej struktury:

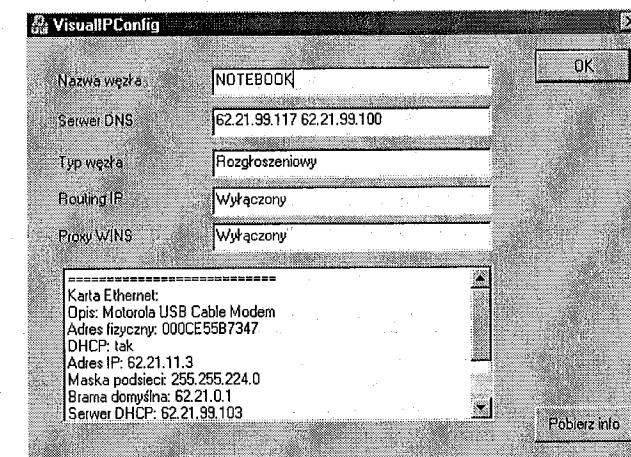
- ◆ Type — rodzaj karty (adAPTERA). Może przyjąć jedną z poniższych wartości (nie trzeba ich chyba komentować):
 - ◆ MIB_IF_TYPE_ETHERNET
 - ◆ MIB_IF_TYPE_TOKENRING
 - ◆ MIB_IF_TYPE_FDDI
 - ◆ MIB_IF_TYPE_PPP
 - ◆ MIB_IF_TYPE_LOOPBACK
 - ◆ MIB_IF_TYPE_SLIP
 - ◆ MIB_IF_TYPE_OTHER
- ◆ AdapterName — systemowa nazwa karty.
- ◆ Description — opis adAPTERA zawierający zazwyczaj nazwę producenta i przeznaczenie urządzenia.
- ◆ AddressLength — rozmiar adresu MAC.
- ◆ Address — adres MAC.

- ◆ DhcpEnabled — opcja korzystania z dynamicznego przydziału adresów (DHCP).
- ◆ IpAddressList — lista adresów IP i masek sieciowych (każdy z adapterów może mieć po kilka adresów).
- ◆ GatewayList — lista bram.
- ◆ DhcpServer — adresy serwerów DHCP.
- ◆ PrimaryWinsServer — adres podstawowego serwera WINS.
- ◆ SecondaryWinsServer — adres pomocniczego serwera WINS.

Aby omawiany przykład dał się bezbłędnie skompilować, należałoby jeszcze we właściwościach projektu w sekcji *Linker/Input* wpisać w polu *Additional Dependencies* nazwę *IPHlpApi.lib*. Dodatkowo w pliku kodu źródłowego, na samym jego początku, należałoby dodać wiersz włączający do kodu plik nagłówkowy *iphlpapi.h*.

Po skompilowaniu projektu uruchom plik *VisualIPConfig.exe*. Kliknij przycisk *Pobierz info*. Elementy okna powinny zostać wypełnione pobranymi z systemu informacjami o karcie sieciowej, jak na rysunku 5.2.

Rysunek 5.2.
Efekt działania
programu
VisualIPConfig



Kod źródłowy niniejszego przykładu znajduje się na dołączonej do książki płytcie CD-ROM w podkatalogu \Przykłady\Rozdział5\VisualIPConfig.

5.2. Zmiana adresu IP komputera

W niniejszym podrozdziale spróbujemy programowo zmienić adres IP komputera. Powoli to nam pisać programy, które mogą okresowo zmieniać adres sieciowy. Zmiany takie mogą zwiększyć bezpieczeństwo komputera i zapobiec niektórym atakom sieciowym.

Karta sieciowa może mieć więcej niż jeden adres IP, stąd potrzeba funkcji przypisującej do niej kolejne adresy oraz funkcji te adresy usuwającej. Aby dodać adres, należy posłużyć się funkcją AddIPAddress przyjmującą pięć parametrów:

- ◆ Adres IP.
- ◆ Maskę sieciową.
- ◆ Identyfikator karty sieciowej, która ma otrzymać nowy adres.
- ◆ Kontekst adresu. Jak pokazuje moje osobiste doświadczenie, najlepiej podać zero. Kontekst zostanie automatycznie ustawiony przez system.
- ◆ Wskaźnik instancji NTE adresu, zwykle ustawiany na zero.

Każdy z adresów sieciowych komputera jest skojarzony z konkretnym urządzeniem (adapterem) sieciowym. Jeśli, na przykład, w systemie zainstalowane są dwie karty sieciowe, utworzone zostaną dwa różne wpisy adresowe. Do rozróżnienia wpisów w systemie służy kontekst. Nie mogą istnieć dwa wpisy o takim samym kontekście, niezależnie od liczby kart sieciowych.

Znając kontekst adresu można łatwo ów adres usunąć, wywołując funkcję DeleteIPAddress z jednym tylko parametrem — kontekstem wpisu adresowego.

Spróbuję zilustrować rzecz na przykładzie. Utwórz nowy, oparty na oknach dialogowych projekt typu *MFC Application*. Opatrz projekt nazwą *ChangeIPAddress*. Docelowy wygląd okna głównego aplikacji ilustruje rysunek 5.3.

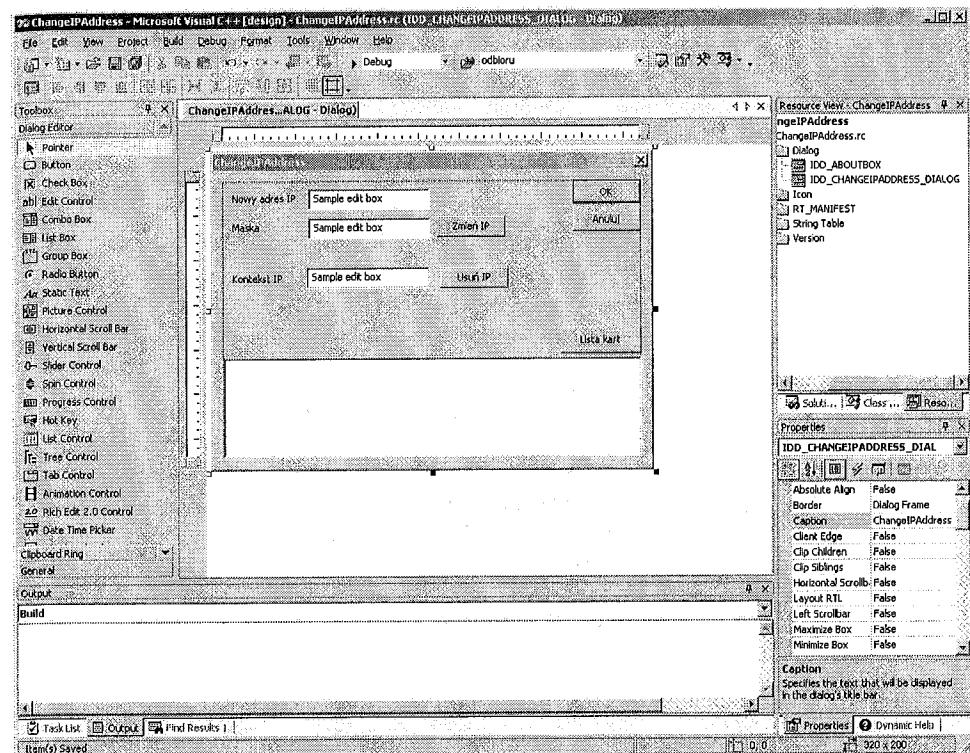
Po kliknięciu przycisku *Lista kart* w oknie widoku listy (*List Box*) rozciagniętego w dolnej części okna powinny zostać wypisane informacje o adresach sieciowych. Kod obsługi przycisku prezentowany jest na listingu 5.2.

Listing 5.2. Wyświetlanie informacji o adresach

```
void CChangeIPAddressD1g::OnBnClickedButton3()
{
    PIP_ADAPTER_INFO pAdapterInfo, pAdapter;
    ULONG iAdapterInfo;
    int iErr;
    CString Str;

    iAdapterInfo = 0;
    iErr = GetAdaptersInfo(NULL, &iAdapterInfo);
    if ((iErr != 0) && (iErr != ERROR_BUFFER_OVERFLOW))
    {
        AfxMessageBox("Błąd funkcji GetAdaptersInfo");
        return;
    }

    if ((pAdapterInfo = (PIP_ADAPTER_INFO) GlobalAlloc(GPTR, iAdapterInfo)) == NULL)
    {
        AfxMessageBox("Błąd przydziału pamięci");
        return;
    }
}
```



Rysunek 5.3. Okno programu *ChangeIPAddress*

```
if (GetAdaptersInfo(pAdapterInfo, &iAdapterInfo) != 0)
{
    AfxMessageBox("Błąd funkcji GetAdaptersInfo");
    return;
}

pAdapter = pAdapterInfo;
1Adapters.AddString("-----");
while (pAdapter)
{
    Str = "Adapter: ";
    1Adapters.AddString(Str + pAdapter->AdapterName);

    char s[20];
    Str = itoa(pAdapter->Index, s, 10);
    Str = "Numer: " + Str;
    1Adapters.AddString(Str);

    PIP_ADDR_STRING chAddr = &(pAdapter->IpAddressList);
    while(chAddr)
    {
        1Adapters.AddString("-----");

        Str = itoa(chAddr->Context, s, 10);
        Str = "Kontekst: " + Str;
        1Adapters.AddString(Str);
    }
}
```

```

        Str = "Adres IP: ";
        !Adapters.AddString(Str + chAddr->IpAddress.String);

        Str = "Maska podsieci: ";
        !Adapters.AddString(Str + chAddr->IpMask.String);

        chAddr = chAddr->Next;
    }
    pAdapter = pAdapter->Next;
}

}

```

Tak jak w podrozdziale 5.1, informacje o adresach pobieramy za pośrednictwem funkcji GetAdaptersInfo. Jak pamiętamy, jej pierwszym parametrem powinna być struktura PIP_ADAPTER_INFO. W tejże strukturze składowa Index przechowuje numer urządzenia sieciowego, który należy następnie przekazać trzecim parametrem wywołania funkcji AddIPAddress przy nadawaniu adapterowi nowego adresu IP. Składowa IpAddressList to tablica struktur typu PIP_ADDR_STRING. Z tej struktury potrzebujemy składowej Context, przechowującej kontekst wpisu adresowego.

Składowa IpAddress to adres IP urządzenia, a IpMask to skojarzona z tym adresem maska podsieci.

Po kliknięciu przycisku *Zmień IP* do listy adresów dopisany zostanie nowy adres IP. Przed jego dodaniem do listy można z niej usunąć wszystkie poprzednie adresy. Kod wykonywany po kliknięciu przycisku prezentowany jest na listingu 5.3.

Listing 5.3. Dodawanie nowego adresu do listy adresów adaptera sieciowego

```

void CChangeIPAddressDlg::OnBnClickedButton1()
{
    PIP_ADAPTER_INFO pAdapterInfo, pAdapter;
    ULONG iAdapterInfo;
    int iErr;
    ULONG iInst, iContext;
    iInst = iContext = 0;

    iAdapterInfo = 0;
    iErr = GetAdaptersInfo(NULL, &iAdapterInfo);
    if ((iErr != 0) && (iErr != ERROR_BUFFER_OVERFLOW))
    {
        AfxMessageBox("Błąd funkcji GetAdaptersInfo");
        return;
    }

    if ((pAdapterInfo = (PIP_ADAPTER_INFO) GlobalAlloc(GPTR, iAdapterInfo)) == NULL)
    {
        AfxMessageBox("Błąd przydziału pamięci");
        return;
    }

    if (GetAdaptersInfo(pAdapterInfo, &iAdapterInfo) != 0)
    {

```

```

        AfxMessageBox("Błąd funkcji GetAdaptersInfo");
        return;
    }

    pAdapter = pAdapterInfo;

    char sNewAddr[20], sNewMask[20];
    eIPEdit.GetWindowText(sNewAddr, 20);
    eMaskEdit.GetWindowText(sNewMask, 20);

    iErr = AddIPAddress(inet_addr(sNewAddr), inet_addr(sNewMask),
                        pAdapter->Index, &iContext, &iInst);
    if (iErr != 0)
        AfxMessageBox("Nie można zmienić adresu");
}

```

Dodanie nowego adresu wymaga przede wszystkim znajomości indeksu (numeru) adaptera sieciowego. Numer ten można określić za pośrednictwem funkcji GetAdaptersInfo. Następnie wystarczy wywołać funkcję AddIPAddress.

Kiedy użytkownik programu kliknie przycisk *Usuń IP*, program powinien usunąć adres o kontekście określonym elementem interfejsu opisanym etykietą *Kontekst IP*. Kod obsługi tego przycisku widnieje na listingu 5.4.

Listing 5.4. Usuwanie adresu IP

```

void CChangeIPAddressDlg::OnBnClickedButton2()
{
    char sContext[20];
    eContext.GetWindowText(sContext, 20);

    int Context = atoi(sContext);
    if (DeleteIPAddress(Context) != 0)
    {
        AfxMessageBox("Nie można usunąć adresu");
    }
}

```

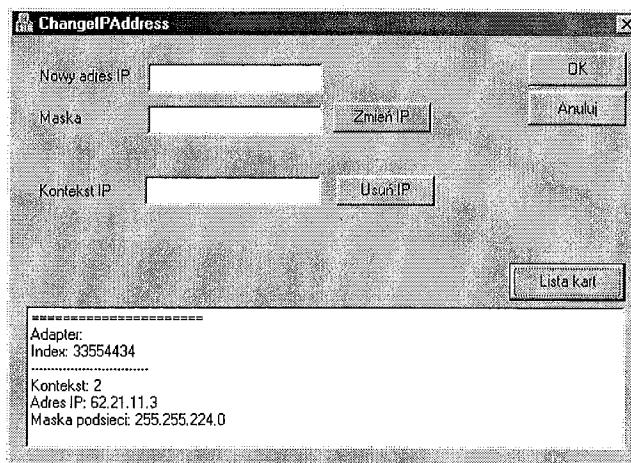
Ciekawy efekt daje usunięcie wszystkich adresów. W tym momencie komputer „znika” z sieci i nie może z nią współpracować. Byłby z tego niezły żart.

Podsumowując niniejszy podrozdział, chciałbym zaznaczyć, że opisywane funkcje będą działać prawidłowo jedynie przy prawidłowej konfiguracji. Funkcje nie zadziałają, na przykład, jeśli wyjmijemy z karty sieciowej wtyczkę kabla sieciowego. Efekt działania programu na moim komputerze przenośnym ilustruje rysunek 5.4. Przed kliknięciem przycisku *Lista kart* odłączylem kabel sieciowy. W efekcie adres IP i maska podsieci to adresy zerowe (0.0.0.0).



Kod źródłowy niniejszego przykładu znajduje się na dołączonej do książki płyce CD-ROM w podkatalogu \Przykłady\Rozdział5\ChangeIPAddress.

Rysunek 5.4.
Efekt działania
ChangeIPAddress
na komputerze
fizycznie odłączonym
od sieci



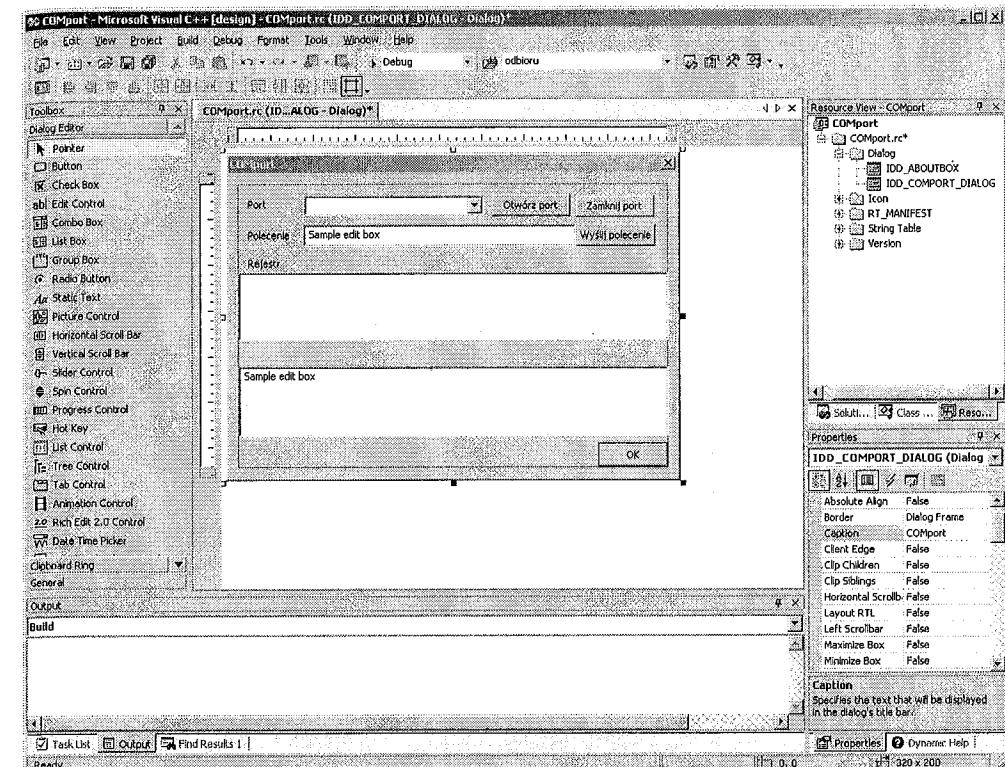
5.3. Obsługa portu szeregowego

W mojej pracy zawodowej miałem niejednokrotnie do czynienia z interfejsem RS-232. To oficjalna nazwa portu szeregowego (COM). Współczesne urządzenia kontrolno-pomiarowe (kontrolery, mierniki, urządzenia monitorujące itd.) często komunikują się z komputerem właśnie za pośrednictwem portu szeregowego. Port ten pośredniczy też w komunikacji z każdym modemem, nawet wewnętrznym. Nie sposób wręcz wyliczyć wszystkich urządzeń przyłączanych do tego portu.

Obsługa portów jest w systemie Windows podobna do obsługi plików. Spójrzmy na prosty przykład. Będzie on wymagać utworzenia nowego projektu typu *MFC Application* (o nazwie np. *COMport*) i zaprojektowania interfejsu programu na wzór tego z rysunku 5.5.

W górnej części okna powinna wylądować rozwijana lista (*Combo Box*), z której można będzie wybrać nazwę portu szeregowego. Obok listy powinny znaleźć się dwa przyciski służące do otwierania i zamknięcia portu. Poniżej warto umieścić pole edycyjne do wprowadzania polecenia oraz przycisk wysyłający polecenie do portu. Pośrodku okna powinien zostać rozciagnięty element widoku listy, za pośrednictwem którego program będzie wyświetlał postęp operacji na porcie; jeszcze niżej zalecam umieszczenie podobnej wielkości wielowierszowego pola edycyjnego, w którym wyświetlane będą dane odbierane z portu.

Po kliknięciu przycisku *Otwórz port* wykonany powinien zostać kod z listingu 5.5.



Rysunek 5.5. Okno programu *COMport*

Listing 5.5. Otwieranie portu

```
void CComportDlg::OnBnClickedOpenportButton()
{
    if (hCom != INVALID_HANDLE_VALUE)
    {
        OnBnClickedButton1();
        Sleep(300);
    }

    char sPortName[10];
    cbPorts.GetWindowText(sPortName, 10);

    hCom = CreateFile(sPortName, GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_EXISTING, 0, NULL);

    if (hCom == INVALID_HANDLE_VALUE)
        lLogList.AddString("Błąd podczas otwierania portu");
    else
        lLogList.AddString("Port otwarty.");

    DCB dcb;
    GetCommState(hCom, &dcb);
    dcb.BaudRate = CBR_57600;
```

```

dcb.ByteSize = 8;
dcb.Parity = NOPARITY;
dcb.StopBits = ONESTOPBIT;
if (SetCommState(hCom, &dcb))
    tLogList.AddString("Konfiguracja zakończona");
else
    tLogList.AddString("Błąd konfiguracji portu");
}
hThread = CreateThread(0, 0, ReadThread, (LPVOID)this, 0, 0);
}

```

Gdybyśmy spróbowali otworzyć otwarty już port, sprowokowalibyśmy komunikat o błędzie. Dlatego właśnie przed przystąpieniem do operacji otwierania powinien zostać wykonany test stanu portu. Jeśli port jest otwarty, powinien zostać zamknięty. Test taki jest przeprowadzany w funkcji OnBnClickedButton1 obsługującej również zdarzenie kliknięcia przycisku *Zamknij port*.

W dalszej kolejności program pobiera nazwę portu i otwiera port. Otwarcie polega na wywołaniu funkcji CreateFile służącej do tworzenia plików, z tym, że zamiast nazwy pliku przekazywana jest do niej nazwa portu.

Jeśli uda się port otworzyć, użytkownik zobaczy stosowny komunikat, a program przystapi do konfiguracji parametrów portu. Konfiguracja rozpoczyna się od pobrania bieżących ustawień portu za pośrednictwem funkcji GetCommState. Wymaga ona przekazania dwóch parametrów: uchwytu otwartego portu oraz wskaźnika struktury typu DCB. Struktura ta zawiera komplet informacji o połączeniu szeregowym i prezentuje się następująco:

```

typedef struct _DCB {
    DWORD DCBLength;           // Rozmiar struktury DBC
    DWORD BaudRate;            // Prędkość transmisji w bodach
    DWORD fBinary:1;           // Tryb bitowy
    DWORD fParity:1;            // Załączanie trybu kontroli parzystości
    DWORD fOutxCtsFlow:1;       // Kontrola nadawania metodą CTS
    DWORD fOutxDsrlFlow:1;      // Kontrola nadawania metodą DSR
    DWORD fDtrControl:2;        // Kontrola prędkości transmisji metodą DTR
    DWORD fDsrsensitivity:1;    // Czułość DSR
    DWORD fTxContinueOnXoff:1;   // Kontynuacja transmisji po znaku stopu
    DWORD fOutX:1;              // Sterowanie przepływem nadawania metodą XON/XOFF
    DWORD fInX:1;                // Sterowanie przepływem odbioru metodą XON/XOFF
    DWORD fErrorChar:1;          // Zastępowanie błędnych znaków ustalonym znakiem
    specjalnym
    DWORD fNull:1;              // Odrzucanie bajtów pustych
    DWORD fRtsControl:2;         // Sterowanie przepływem metodą RTS
    DWORD fAbortOnError:1;       // Zawieszanie operacji w obliczu błędu
    DWORD fDummy2:17;             // Zarezerwowane
    DWORD wReserved;             // Zarezerowane — zawsze zero
    WORD XonLim;                  // Próg inicjacji kontroli przepływu
    WORD XoffLim;                 // Próg zatrzymania kontroli przepływu
    BYTE ByteSize;                // Rozmiar znaku (najczęściej siedem albo osiem)
    BYTE Parity;                  // Schemat kontroli parzystości
    BYTE StopBits;                 // Liczba bitów stopu
    char XonChar;                  // Znak XON inicjujący kontrolę przepływu
}

```

```

char XoffChar;           // Znak XOFF zatrzymujący kontrolę przepływu
char ErrorChar;           // Wyróżniony znak dla znaków błędnych

char EofChar;             // Znak końca transmisji
char EvtChar;              // Zarezerwowany
WORD wReserved;            // Zarezerwowany
} DCB;

```

Niepoprawne wypełnienie pól struktury uniemożliwia komunikację za pośrednictwem portu szeregowego. Najważniejsze pola to:

- ◆ BaudRate — prędkość transmisji danych w bodach, czyli bitach na sekundę. Należy ustawić wartość CBR_średnia, gdzie średnia musi być średnią obsługiwana przez urządzenie podłączone do portu, np. CBR_56000.
- ◆ ByteSize — rozmiar znaku w bitach (siedem albo osiem).
- ◆ Parity — tryb kontroli parzystości.
- ◆ StopBits — liczba bitów stopu. Może przyjąć wartości ONESTOPBIT (jeden bit stopu), ONE5STOPBITS (półtora bita stopu) albo TWOSTOPBITS (dwa bity stopu).

W pozostałych polach można pozostawić wartości domyślne (czyli zwracane przez system w ramach odczytu bieżącej konfiguracji). Przed przypisaniem jakiegokolwiek wartości do jakiegokolwiek elementu tej struktury należy się upewnić co do jej poprawności, odczytując wartości dopuszczalnych parametrów w instrukcji obsługi i programowania urządzenia podłączonego do portu szeregowego. Nie widziałem jeszcze urządzenia, które obsługiwałoby wszystkie możliwe tryby transmisji. Na przykład modem ZyXel Omni 56K obsługuje prędkości transmisji od 2 400 do 56 000 (w bodach). W komunikacji z takim modelem nie należy wykraczać poza te wartości.

Dalej, oba uczestniczące w komunikacji urządzenia powinny być skonfigurowane tak samo (pracować z takimi samymi prędkościami transmisji, rozmiarami znaków itd.), w przeciwnym razie próby komunikacji będą skazane na niepowodzenie.

Po zakończeniu konfiguracji portu program tworzy nowy wątek wykonania. W ramach tego wątku próbuje (w nieskończoność) odczytać dane z portu. Oczywiście takie rozwiązanie oczekiwania na dane nie jest szczególnie efektywne — lepiej byłoby skorzystać z możliwości sygnaлизacji stanu portu za pośrednictwem komunikatów systemu Windows. W tak prostym przykładzie efektywność programu nie ma jednak większego znaczenia. Funkcja kodu wątku prezentuje się więc następująco:

```

DWORD __stdcall ReadThread(LPVOID hwnd)
{
    DWORD iSize;
    char sReceivedChar;
    while(true)
    {
        ReadFile(hCom, &sReceivedChar, 1, &iSize, 0);
        SendDlgItemMessage((HWND)hwnd, IDC_EDIT2, WM_CHAR, sReceivedChar, 0);
    }
}

```

Odczyt danych z portu odbywa się tu za pośrednictwem standardowej funkcji obsługi systemu plików — ReadFile.

Przyjrzyjmy się teraz funkcji zamkającej port, a wyzwalanej kliknięciem przycisku *Zamknij port*:

```
void CComPortDlg::OnBnClickedButton1()
{
    if (hCom == INVALID_HANDLE_VALUE)
        return;

    if (MessageBox("Zamknąć port?", "Uwaga", MB_YESNO) == IDYES)
    {
        TerminateThread(hThread, 0);
        CloseHandle(hCom);
        hCom = INVALID_HANDLE_VALUE;
    }
}
```

Przed właściwą operacją zamknięcia portu funkcja sprawdza stan zmiennej hCom. Jest bowiem prawdopodobne, że w momencie naciśnięcia przycisku port pozostaje zamknięty (bo, na przykład, nie został nigdy wcześniej otwarty). Jeśli zmienna zawiera niepoprawną wartość uchwytu portu, funkcja kończy działanie.

Jeśli jednak port jest faktycznie otwarty, funkcja wyświetla monit z żądaniem potwierdzenia chęci zamknięcia portu. Jeśli użytkownik potwierdzi operację, w pierwszej kolejności zamkany jest wątek obsługujący transmisję, potem zamkany jest sam port, a następnie do zmiennej hCom przypisywana jest wartość INVALID_HANDLE_VALUE (kodująca niepoprawną wartość uchwytu).

Programowi brakuje już jedynie opcji inicjowania własnych transmisji w reakcji na naciśnięcie przycisku *Wyślij polecenie*. Zdarzenie kliknięcia tego przycisku obsługuje funkcja z listingu 5.6.

Listing 5.6. Funkcja wysyłająca dane do portu

```
void CComPortDlg::OnBnClickedSendCommandButton()
{
    if (hCom == INVALID_HANDLE_VALUE)
    {
        AfxMessageBox("Przed wysłaniem polecenia otwórz port");
        return;
    }

    char sSend[1024];
    eSendCommand.GetWindowText(sSend, 1024);

    if (strlen(sSend) > 0)
    {
        lLogList.AddString(sSend);

        sSend[strlen(sSend)] = '\r';
        sSend[strlen(sSend)] = '\0';

        TerminateThread(hThread, 0);
        DWORD iSize;
```

```
        WriteFile(hCom, sSend, strlen(sSend), &iSize, 0);
        hThread = CreateThread(0, 0, ReadThread, (LPVOID)this, 0, 0);
    }
}
```

Funkcja na samym początku sprawdza, czy port jest otwarty. Jeśli nie jest, podejmowanie prób wysłania danych byłoby bezcelowe. Następnie sprawdzany jest rozmiar danych do wysłania. Jeśli rozmiar jest niezerowy, funkcja uzupełnia dane znakiem końca ciągu (znakiem pustym), wstawiając przed nim znak nowego wiersza. Większość urządzeń, które miałem okazję obsługiwać programowo, wymagała sygnalizowania końca polecenia znakiem nowego wiersza składającym się z kodów wysuwu wiersza i powrotu karetki; niektóre zadowalały się samym znakiem wysuwu wiersza.

Po skonstruowaniu ciągu danych do wysłania przerywany jest wątek obsługujący odbiór danych, po czym za pośrednictwem funkcji WriteFile następuje zapis danych do portu. Po zakończeniu zapisu wątek odbiorczy jest wznowiany.

Jeśli posiadasz modem, możesz uruchomić program, wybrać port, do którego podłączony jest modem, i wpisać w polu polecień polecenie ATDTxxxxxx, gdzie xxxxxxxx to numer telefonu. Modem powinien rozpoczęć wybieranie numeru.



Kod źródłowy niniejszego przykładu znajduje się na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział5\ComPort.

5.4. Pliki zawieszające system

Wiemy już z grubsza, jak programowo obsługiwać pliki. W podrozdziale 4.2 próbowaliśmy utworzyć plik na komputerze zdalnym i zapisać w nim jakieś dane. Przypomniało mi to o jeszcze jednej ciekawostce.

Jak zapewne Czytelnik pamięta, ścieżka dostępu do zasobu sieciowego ma postać:

nazwa-komputera\disk\ścieżka

Jeśli masz zamiar odwołać się do dysku lokalnego jako sieciowego, ale dysk ten nie jest udostępniony w sieci, możesz za nazwą dysku umieścić znak \$. Aby, na przykład, odwołać się do pliku *mójplik.txt*, przechowywanego na dysku C, powinieneś skonstruować następującą ścieżkę:

\\\MojKomputer\C\$\mójplik.txt

Teraz najciekawsze. W systemie Windows nie można tworzyć plików, które mają w nazwach znaki zapytania; system operacyjny kontroluje wprowadzane nazwy pod tym kątem, uniemożliwiając użytkownikowi utworzenie nieprawidłowo nazwanego pliku. Jednak przy odwołaniach inicjowanych za pośrednictwem sieci kontrola taka jest pomijana. Jeśli więc Twój komputer podłączony jest do sieci, możesz przejść do dowolnego udziału sieciowego innego komputera. Założmy, że jego nazwa to e:. Jeśli wykonasz teraz kod z listingu 5.7, program zostanie zawieszony i nie będzie żadnej możliwości jego zakończenia.

Listing 5.7. Ominięcie zabezpieczeń nazw systemu plików za pośrednictwem sieci

```

if ((FileHandle = CreateFile("\\\\notebook\\\\temp\\\\?myfile.txt",
    GENERIC_WRITE | GENERIC_READ,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
)) == INVALID_HANDLE_VALUE)
{
    MessageBox(0, "Błąd utworzenia pliku", "Error", 0);
    return;
}
// Zapis w pliku 12 znaków
if (WriteFile(FileHandle, "Zapis próbny", 12, &Written, NULL) == 0)
{
    MessageBox(0, "Błąd zapisu w pliku", "Błąd", 0);
    return;
}
// Zamknij plik
CloseHandle(FileHandle);

```

W powyższym kodzie próbujemy utworzyć plik i zapisać w nim dwanaście znaków, jednak z powodu błędnej nazwy pliku (zawiera ona znak ?) pliku nie można utworzyć. Z drugiej strony z powodu braku stosownej kontroli parametrów polecenia program będzie mimo wszystko czekał na odpowiedź systemu operacyjnego. Nie doczeka się jej nigdy.



Kod źródłowy niniejszego przykładu można zaczerpnąć z podkatalogu \Przykłady\Rozdział4\Network znajdującego się na dołączonej do książki płycie CD-ROM.

Rozdział 6.

Sztuczki, kruczki i ciekawostki

W poprzednich rozdziałach udało się nam przeanalizować szereg programów, często zartobliwych, i przedyskutować teoretyczne aspekty programowania systemowego (w Windows) i sieciowego. W niniejszym rozdziale chciałbym skoncentrować się na kilku przydatnych algorytmach. Będą one ilustrowały metody, jakimi posługują się hakerzy. Postaram się też skonsolidować wiedzę teoretyczną Czytelnika.

Omawiając funkcje sieciowe w rozdziałach 4. i 5., próbowaliśmy owe funkcje wykorzystywać w prostych programach przykładowych. Programy te były jednak dalekie od doskonałości. Na przykład skaner portów (z podrozdziału 4.4) był powolny, więc musielibyśmy zawęzić zakres skanowania do zaledwie stu portów. Powiedziałem przy tamtej okazji, jak przyspieszyć skanowanie. W tym rozdziale napiszemy więc najszybszy skaner, jaki da się stworzyć przy zachowaniu jego elastyczności i uniwersalności.

Postaram się też pokazać kilka ulepszeń związanych z obsługą transmisji (wysyłania i odbierania) danych. Transmisja danych to częste wąskie gardło oprogramowania, zwłaszcza w tych zastosowaniach, w których ważne jest pogodzenie wysokiej wydajności programu z jak najmniejszym obciążeniem procesora.

Programowanie to prawdziwa sztuka i rozległa dziedzina umiejętności, przy czym różne cele osiąga się w niej rozmaitymi środkami. Nie będziemy w stanie omówić sposobu zachowania i doboru środków do każdego możliwego problemu programistycznego — nie starczyłoby na to paru tysięcy stron, a i po drodze nie obyło by się bez wyższej matematyki. Dlatego znów skupię się na stosunkowo wąskiej dziedzinie zadań, czyli zagadnień sieciowych.

6.1. Algorytm odbioru-wysyłania danych

W punkcie 4.10.2 mieliśmy okazję obserwować przykład, w ramach którego serwer w sposób asynchroniczny akceptował połączenia klientów. Zaraz po dokończeniu procedury nawiązania połączenia tworzony był nowy wątek wymieniający dane z klientem.

Już wtedy zaznaczyłem, że asynchroniczne funkcje sieciowe pozwoliłyby na obsługę wielu połączeń za pośrednictwem pojedynczego wątku. W takim układzie tworzenie osobnych wątków do obsługi poszczególnych klientów jest marnotrawstwem zasobów systemowych. Stąd na listingu 6.1 prezentuję kod programu, w którym w ramach jednej funkcji serwer najpierw oczekuje na nawiązanie połączenia, a następnie obsługuje to połączenie.

Listing 6.1. Algorytm współbieżnej obsługi klientów w jednym wątku

```
DWORD WINAPI NetThread(LPVOID lpParam)
{
    SOCKET sServerListen;
    SOCKET ClientSockets[50];
    int TotalSocket=0;

    struct sockaddr_in localaddr,
                           clientaddr;
    int iSize;

    sServerListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (sServerListen == SOCKET_ERROR)
    {
        MessageBox(0, "Nie można wczytać biblioteki WinSock", "Błąd", 0);
        return 0;
    }

    ULONG ulBlock;
    ulBlock = 1;
    if (ioctlsocket(sServerListen, FIONBIO, &ulBlock) == SOCKET_ERROR)
    {
        return 0;
    }

    localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    localaddr.sin_family = AF_INET;
    localaddr.sin_port = htons(5050);

    if (bind(sServerListen, (struct sockaddr *)&localaddr,
             sizeof(localaddr)) == SOCKET_ERROR)
    {
        MessageBox(0, "Nie można zainstalować serwera", "Błąd", 0);
        return 1;
    }

    MessageBox(0, "Serwer zainstalowany", "Uwaga", 0);
}
```

```
listen(sServerListen, 4);

MessageBox(0, "Nasłuch rozpoczęty", "Uwaga", 0);

FD_SET ReadSet;
int ReadySock;

while (1)
{
    FD_ZERO(&ReadSet);
    FD_SET(sServerListen, &ReadSet);

    for (int i = 0; i < TotalSocket; i++)
        if (ClientSockets[i] != INVALID_SOCKET)
            FD_SET(ClientSockets[i], &ReadSet);

    if ((ReadySock = select(0, &ReadSet, NULL, NULL, NULL)) == SOCKET_ERROR)
    {
        MessageBox(0, "Błąd funkcji select", "Błąd", 0);
    }

    // Mamy nowe połączenie
    if (FD_ISSET(sServerListen, &ReadSet))
    {
        iSize = sizeof(clientaddr);
        ClientSockets[TotalSocket] = accept(sServerListen,
                                           (struct sockaddr *)&clientaddr, &iSize);
        if (ClientSockets[TotalSocket] == INVALID_SOCKET)
        {
            MessageBox(0, "Błąd funkcji accept", "Błąd", 0);
            break;
        }
        TotalSocket++;
    }

    // Mamy dane od klienta
    for (int i = 0; i < TotalSocket; i++)
    {
        if (ClientSockets[i] == INVALID_SOCKET)
            continue;
        if (FD_ISSET(ClientSockets[i], &ReadSet))
        {
            char szRecvBuff[1024],
                 szSendBuff[1024];

            int ret = recv(ClientSockets[i], szRecvBuff, 1024, 0);
            if (ret == 0)
            {
                closesocket(ClientSockets[i]);
                ClientSockets[i] = INVALID_SOCKET;
                break;
            }
            else if (ret == SOCKET_ERROR)
            {
                MessageBox(0, "Błąd odbioru danych", "Błąd", 0);
                break;
            }
        }
    }
}
```

```

szRecvBuff[ret] = '\0';

strcpy(szSendBuff, "Polecenie GET przyjęte");

ret = send(ClientSockets[i], szSendBuff, sizeof(szSendBuff), 0);
if (ret == SOCKET_ERROR)
{
    break;
}

closesocket(sServerListen);
return 0;
}

```

Jak działa powyższy kod? Na początku deklarowane są dwie zmienne:

- ◆ sServerListen — zmienna gniazda nasłuchu serwera pośredniczącego w przyjmowaniu połączeń inicjowanych przez klientów.
- ◆ ClientSockets — tablica pięćdziesięciu elementów typu SOCKET z przeznaczeniem do obsługi wymiany danych z klientami. Taki rozmiar tablicy gniazd pozwala na współbieżną obsługę do 50 połączeń. Dane są wymieniane z kolejno łączącymi się klientami za pośrednictwem kolejnych gniazd aż do klienta 51. z rzędu, którego połączenie zostanie odrzucone. W prawdziwym (nie przykładowym) programie należałoby zastąpić tę tablicę tablicą dynamiczną i dodatkowo zadbać o usuwanie niewykorzystywanych już gniazd po zakończeniu połączenia.

Następnie funkcja wątku serwera tworzy gniazdo nasłuchu, przełącza je w tryb asynchroniczny, instaluje dla lokalnego adresu wywołaniem funkcji bind i wreszcie rozpoczyna na nasłuch. Ten fragment kodu nie powinien budzić wątpliwości.

Znacznie ciekawszym fragmentem jest ten w pętli while, która w poprzednich przykładach służyła wyłącznie do oczekiwania na połączenie i do utworzenia kolejnego wątku obsługującego komunikację z nowym klientem. Tutaj posługujemy się jednak funkcją select, z tym że do zestawu obserwowanych gniazd włączane jest nie tylko gniazdo nasłuchu serwera, ale również aktywne już gniazda wymiany danych z klientami. Dzięki temu możemy reagować na zmiany stanu wszystkich gniazd wykorzystywanych w programie.

Im głębszej w las, tym więcej drzew, a im dalej w kod, tym kod ciekawszy. Po powrocie z funkcji select w pierwszej kolejności sprawdzany jest stan gniazda nasłuchu serwera. Jeśli gniazdo jest gotowe do odczytu, oznacza to pojawienie się kolejnego żądania nawiązania połączenia. W takim przypadku należy przyjąć połączenie wywołaniem funkcji accept i utworzyć dla potrzeb komunikacji z nowym klientem nowe gniazdo sieciowe. Gniazdo to przechowywane jest w kolejnym elemencie tablicy gniazd komunikacji z klientami. Następnie jest też włączane do zestawu gniazd obserwowanych przez funkcję select.

Po obsłudze ewentualnego żądania nawiązania połączenia program przystępuje do przeglądu gniazd klientów celem sprawdzenia gotowości do odczytu danych z tych gniazd. Jeśli dane gniazdo jest gotowe, program odczytuje przesłane dane i, ewentualnie, wysyła odpowiedź. Jeśli do gniazda nie dotarła jeszcze żadna transmisja, czyli jeśli funkcja recv zwróci 0, klient jest rozłączany.

Zaprezentowany algorytm jest nie tylko szybki, ale i elastyczny. Co ważniejsze, pozwala na obsługiwanie nasłuchu i komunikacji z klientami w ramach pojedynczej pętli. To metoda bardzo wygodna i efektywna. Jeśli program ma docelowo wymieniać niewielkie komunikaty, można w nim wykorzystać kod w tej postaci. Jeśli zaś transmisje miałyby obejmować większe ilości danych, należałoby rozbudować kod o obsługę transmisji danych porcjami.

Warto też pamiętać o możliwości zastosowania dynamicznej tablicy gniazd klientów. Jeśli chcesz tego jednak uniknąć, możesz skorzystać z prostszej metody zarządzania gniazdam — przed wypełnieniem struktury FD_SET przejrzyj elementy tablicy w poszukiwaniu takich gniazd, które mają wartość równą INVALID_SOCKET. Usuń je, a pozostałe przemieśc na miejsca usuniętych. Potem możesz dostosować wartość zmiennej TotalSocket tak, aby zawierała numer pierwszego wolnego elementu.



Kod źródłowy tego przykładu dostępny jest na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział6\AdvancedTCPserver.

6.2. Szybki skaner portów

Wątki to efektywne narzędzia zrównoleglenia przetwarzania pozwalające na implementację wielozadaniowości w obrębie pojedynczej aplikacji. Mają one jednak również wady. Otóż programiści mający doświadczenie w programowaniu wątków mają tendencję do ich nadużywania.

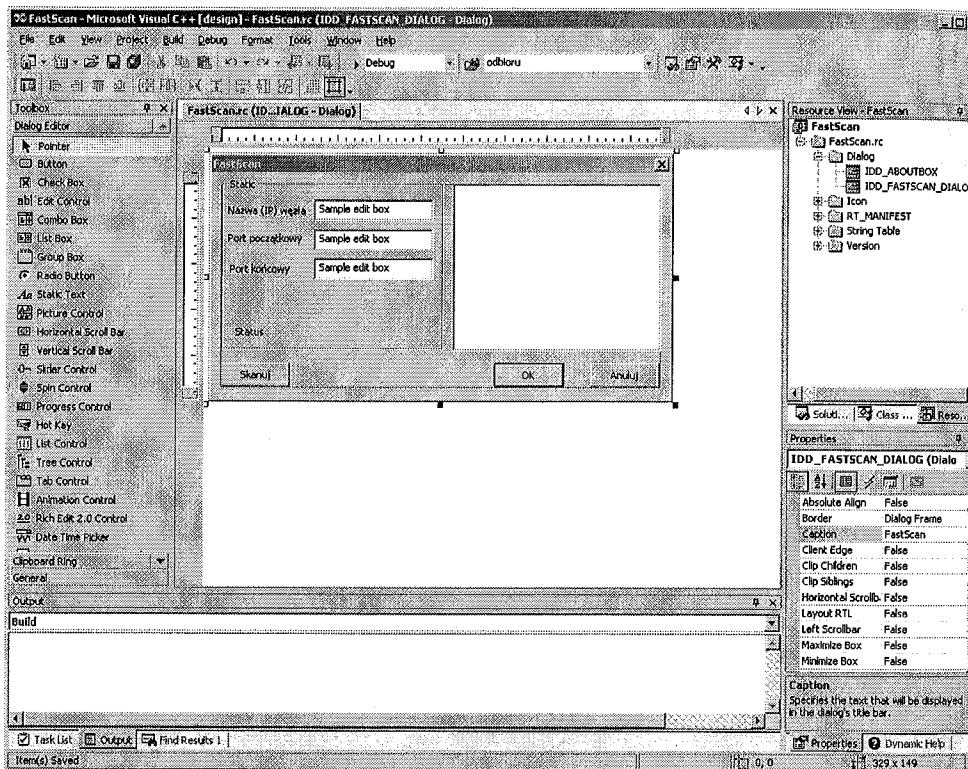
Widziałem już wiele skanerów portów, które wykorzystywały do skanowania dwadzieścia, a nierzadko i pięćdziesiąt wątków. Zdaje sobie sprawę, że program skanera portów z rozdziału 4. stanowi skrajność przeciwną, ponieważ jako jednowątkowy działa bardzo powoli. Można go jednak przyspieszyć, nie tylko obciążając system dodatkowymi wątkami wykonania. Zresztą Czytelnik może spróbować skanowania za pomocą kilku współbieżnych wątków. Przekona się, że nie jest to takie proste, przekona się też, że duża liczba wątków zwiększa dużą ilość zasobów systemowych.

Chciałbym pokazać, jak można napisać naprawdę szybki skaner portów bez korzystania z wielu wątków. Łatwo się domyślić, że będzie to związane z asynchroniczną obsługą operacji sieciowych. Można bowiem utworzyć kilka gniazd, przełączyć je w tryb asynchroniczny, zainicjować próbę nawiązania połączenia i dopisać je do wspólnego zestawu gniazd obserwowanych funkcją select. Kiedy funkcja zakończy działanie, należy tylko sprawdzić wszystkie gniazda i wyświetlić wyniki.

Spróbuj to oprogramować. Utwórz nowy projekt typu *MFC Application* (oparty na oknach dialogowych) o nazwie *FastScan*. Nie zaznaczaj jednak na zakładce opcji zaawansowanych (*Advanced Features*) kreatora aplikacji pola *WinSock*. W tym przykładzie będziemy korzystać z kilku zaledwie funkcji biblioteki gniazd (WinSock2), możemy więc ręcznie dołączyć do kodu źródłowego plik nagłówkowy *winsock2.h*, a do całego projektu plik biblioteki *ws2_32.dll*. Nie muszę chyba już powtarzać, jak się to robi.

Teraz otwórz w edytorze zasobów główne okno programu. Umieść na nim elementy sterujące zgodnie z rysunkiem 6.1. Na formularzu okna głównego powinny wylądować trzy pola edycyjne (*Edit Box*) i jedno pole listy (*List Box*) oraz przycisk, który posłuży do uruchamiania procesu skanowania. Dla pól edycyjnych utwórz następujące zmienne:

- ◆ dla pola opatrzonego etykietą *Nazwa (IP) węzła* — zmienną *chHostName*,
- ◆ dla pola opatrzonego etykietą *Port początkowy* — zmienną *chStartPort*,
- ◆ dla pola opatrzonego etykietą *Port końcowy* — zmienną *chEndPort*.



Rysunek 6.1. Projekt okna programu *FastScan*

Ogólna liczba portów jest tak duża, że nawet najszybszy skaner musiałby poświęcić na ich wysondowanie dość długi czas.

Przejdzmy do programowania. Utwórz zdarzenie kliknięcia przycisku uruchamiającego skanowanie (w tym celu wystarczy dwukrotnie kliknąć przycisk *Skanuj* w edytorze

zasobów). Kod obslugi zdarzenia jest dość długi (patrz listing 6.2), zalecałbym jednak mimo wszystko jego ręczne przepisanie bez uciekania się do kopiowania zawartości pliku z dołączonej do książki płyty CD-ROM. Włsnoręczne przepisywanie da Ci szansę zapoznania się z każdym kolejnym wierszem, co na pewno ułatwi zrozumienie działania całości. Za chwilę zresztą działanie to obszerne wyjaśnie.

Listing 6.2. Kod szybkiego skanera portów

```
void CFastScanDlg::OnBnClickedButton1()
{
    char tStr[255];
    SOCKET sock[MAX_SOCKETS];
    int busy[MAX_SOCKETS], port[MAX_SOCKETS];
    int iStartPort, iEndPort, iBusySocks = 0;
    struct sockaddr_in addr;
    fd_set fdWaitSet;

    WSADATA      wsdd;
    if (WSAStartup(MAKEWORD(2,2), &wsdd) != 0)
    {
        SetDlgItemText(IDC_STATUSTEXT, "Nie można wczytać biblioteki WinSock");
        return;
    }

    SetDlgItemText(IDC_STATUSTEXT, "Określanie adresu węzła");

    chStartPort.GetWindowText(tStr, 255);
    iStartPort = atoi(tStr);
    chEndPort.GetWindowText(tStr, 255);
    iEndPort = atoi(tStr);

    chHostName.GetWindowText(tStr, 255);

    struct hostent *host = NULL;
    host = gethostbyname(tStr);
    if (host == NULL)
    {
        SetDlgItemText(IDC_STATUSTEXT, "Nie można określić adresu węzła");
        return;
    }

    for (int i = 0; i < MAX_SOCKETS; i++)
        busy[i] = 0;

    SetDlgItemText(IDC_STATUSTEXT, "Skanowanie");

    while (((iBusySocks) || (iStartPort <= iEndPort)))
    {
        for (int i = 0; i < MAX_SOCKETS; i++)
        {
            if (busy[i] == 0 && iStartPort <= iEndPort)
            {
                sock[i] = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
                if (sock[i] < 0)
                {
                    SetDlgItemText(IDC_STATUSTEXT, "Błąd funkcji socket");
                }
            }
        }
    }
}
```

```

        return;
    }
    iBusySocks++;
    addr.sin_family = AF_INET;
    addr.sin_port = htons (iStartPort);
    CopyMemory(&addr.sin_addr, host->h_addr_list[0],
               host->h_length);

    ULONG ulBlock;
    ulBlock = 1;
    if (ioctlsocket(sock[i], FIONBIO, &ulBlock) == SOCKET_ERROR)
    {
        return;
    }

    connect(sock[i], (struct sockaddr *) &addr, sizeof (addr));
    if (WSAGetLastError() == WSAEINPROGRESS)
    {
        closesocket (sock[i]);
        iBusySocks--;
    }
    else
    {
        busy[i] = 1;
        port[i] = iStartPort;
    }
    iStartPort++;
}
FD_ZERO (&fdWaitSet);
for (int i = 0; i < MAX_SOCKETS; i++)
{
    if (busy[i] == 1)
        FD_SET (sock[i], &fdWaitSet);
}

struct timeval tv;
tv.tv_sec = 1;
tv.tv_usec = 0;

if (select (1, NULL, &fdWaitSet, NULL, &tv) == SOCKET_ERROR)
{
    SetDlgItemText(IDC_STATUSTEXT, "Błąd funkcji select");
    return;
}

for (int i = 0; i < MAX_SOCKETS; i++)
{
    if (busy[i] == 1)
    {
        if (FD_ISSET (sock[i], &fdWaitSet))
        {
            int opt;
            int Len = sizeof(opt);
            if (getsockopt(sock[i], SOL_SOCKET, SO_ERROR,
                           (char*)&opt, &Len) == SOCKET_ERROR)
                SetDlgItemText(IDC_STATUSTEXT, "Błąd funkcji getsockopt");
        }
    }
}

```

```

    if (opt == 0)
    {
        struct servent *tec;
        itoa(port[i], tStr, 10);
        strcat(tStr, " (");
        tec = getservbyport(htons (port[i]), "tcp");
        if (tec == NULL)
            strcat(tStr, "Nieznanego");
        else
            strcat(tStr, tec->s_name);

        strcat(tStr, ") - otwarty");
        m_PortList.AddString(tStr);
        busy[i] = 0;
        shutdown(sock[i], SD_BOTH);
        closesocket(sock[i]);
    }
    busy[i] = 0;
    shutdown (sock[i], SD_BOTH);
    closesocket (sock[i]);
    iBusySocks--;
}
else
{
    busy[i] = 0;
    closesocket(sock[i]);
    iBusySocks--;
}
}
}
WSACleanup();
SetDlgItemText(IDC_STATUSTEXT, "Skanowanie zakończone");
return;
}

```

Podczas skanowania program odwołuje się do trzech tablic:

- ◆ **sock** — tablicy uchwytów gniazd oczekujących na nawiązanie połączenia.
- ◆ **busy** — tablica stanów skanowanych portów. Każdy z portów może być portem zarezerwowanym, a próba nawiązania połączenia zakończy się błędem. W pliku pomocy biblioteki WinSock stwierdzono zresztą wprost, że nie każdy otwarty port musi nadawać się do użycia. Dlatego te z elementów tablicy, których numery odpowiadają portom zarezerwowanym, mają wartość 1; pozostały przypisywana jest wartość 0.
- ◆ **port** — tablica skanowanych portów. Program obyłby się i bez niej, ale jej zastosowanie pozwala na uproszczenie kodu.

Ten przykład korzysta z funkcji, której nie mieliśmy okazji dotychczas poznać — `getservbyport`. Jej deklaracja prezentuje się następująco:

```

struct serverent FAR * getservbyport(
    int port,
    const char FAR * proto
);

```

Funkcja ta zwraca informacje o usłudze udostępnianej za pośrednictwem portu o zadanym w wywoaniu numerze. Drugi parametr określa protokół. Wartością zwracaną jest wskaźnik struktury typu servent; jej składowa `s_name` zawiera słowny opis usługi. Zerową wartość zwracaną należy interpretować tak, że funkcja nie zdołała określić rodzaju usługi dla danego numeru portu.

Funkcję `getservbyport` bardzo łatwo oszukać, więc zwracane przez nią informacje nie mogą być traktowane jako całe wiarygodne. Na przykład dla portu o numerze 21 funkcja zawsze zwraca informację o usłudze FTP. Tymczasem nasłuch na tym porcie może co prawda prowadzić serwer FTP, ale równie dobrze port ten może być okupowany przez serwer WWW. Tego funkcja `getservbyport` już nie sprawdza.

Pozostałe czynności w ramach obsługi przycisku uruchamiającego skanowanie nie powinny budzić wątpliwości. Oto ogólny algorytm skanowania:

1. Wczytaj bibliotekę gniazd.
2. Określ adres IP komputera, który ma zostać poddany skanowaniu. Adres ten zostanie zapisany w strukturze `sockaddr_in`, której składowa numeru portu będzie zmieniana w kolejnych przebiegach pętli. Ponieważ adres skanowanego komputera pozostaje stały w czasie wykonania pętli skanującej, jest określany tylko jeden raz, przed rozpoczęciem pętli. Byłoby bezcelowe powtarzanie operacji określenia adresu IP w każdym przebiegu pętli, zwłaszcza jeśli wziąć pod uwagę czasochlonność procesu konwersji nazwy symbolicznej na postać adresu IP.
3. Uruchom pętlę kontynuowaną dopóty, dopóki początkowy numer portu jest mniejszy od numeru portu końcowego. Wewnątrz tejże pętli powtarzaj następujące operacje:
 - ◆ Rozpocznij pętlę powtarzaną `MAX_SOCKETS` razy. W tej pętli utwórz gniazdo, przełącz je w tryb asynchroniczny i wywołaj funkcję `connect`. Ponieważ gniazda działają w trybie asynchronicznym, program nie będzie oczekiwany na zakończenie operacji nawiązywania połączenia i będzie mógł kontynuować działanie. Nie będzie jednak wiadomo od razu, czy połączenie miało miejsce, czy też było niemożliwe.
 - ◆ Wyzeruj zbiór obserwowanych gniazd (`fdWaitSet`).
 - ◆ Ponownie rozpocznij pętlę powtarzaną `MAX_SOCKET` razy. W kolejnych przebiegach tej pętli dodaj kolejne gniazda do zbioru gniazd obserwowanych.
 - ◆ Zaczekaj na zmianę stanu gniazd, wywołując funkcję `select`.
 - ◆ Uruchom jeszcze raz pętlę powtarzaną `MAX_SOCKETS` razy. Tym razem sprawdź w jej kolejnych przebiegach poszczególne gniazda ze zbioru gniazd obserwowanych. Dla tych gniazd, które dokończyły nawiązywanie połączenia, pobierz symboliczną nazwę portu (funkcją `getsockopt`). Zamknij gniazdo, zamkając tym samym połączenie z serwerem.
4. Usuń z pamięci bibliotekę gniazd.

Co oznacza `MAX_SOCKETS`? To stała określająca liczbę numerów portów skanowanych w pojedynczym przebiegu pętli. W tym przykładzie została ona ustalona na 40. To opty-

malna wartość dla większości środowisk. Im większa liczba jednorazowo skanowanych portów, tym szybciej przebiega całe skanowanie.

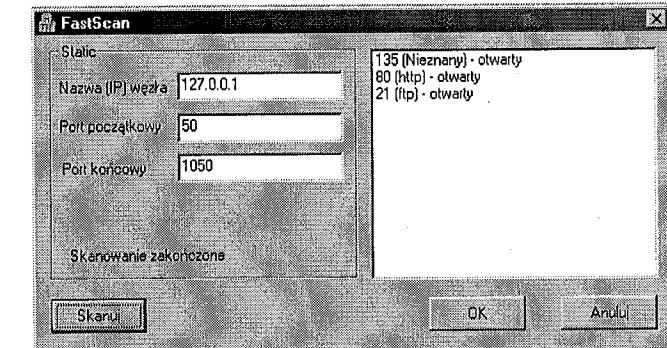
Program wciąż nie jest doskonały. Skanowanie blokuje go, więc otwarte porty można zobaczyć dopiero po zakończeniu skanowania, kiedy program przystępuje do aktualizacji okna głównego. Niekorzystnemu efektowi braku reaktywności można zapobiec, pisząc następującą funkcję:

```
void ProcessMessages()
{
    MSG msg;
    while (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
    {
        if (GetMessage(&msg, NULL, 0, 0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
            return;
    }
}
```

Powyższa funkcja zawiera prostą pętlę obsługi komunikatów. Podobne pętle widywaliśmy wielokrotnie w aplikacjach Win32. Pętla nie jest tym razem nieskończona i nie oczekuje na napływ komunikatów, ograniczając się do przetwarzania tych, które wylądowały już w kolejce komunikatów programu. Po wyczerpaniu kolejki pętla zostanie zakończona i program wróci do skanowania.

Funkcję tę należy umieścić w pliku kodu źródłowego (w pobliżu jego początku) i wywoływać pod koniec głównej pętli skanującej. Pozwoli to na uniknięcie „zawieszenia” programu i podglądanie na bieżąco wyników skanowania (jak na rysunku 6.2).

Rysunek 6.2.
Efekt skanowania komputera



Chciałbym też wspomnieć, że w tej postaci program określa nazwy usług dla danych portów jedynie dla protokołu TCP. Skanowaniu podlegają również wyłącznie porty TCP. W celu skanowania portów UDP należałoby utworzyć gniazda datagramowe.



Kod źródłowy tego przykładu dostępny jest na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział 6\FastScan.

6.3. Stan portów komputera lokalnego

Jeśli chcesz wiedzieć, które porty komputera lokalnego są w danej chwili otwarte, nie musisz uciekać się do skanowania wszystkich jego portów. Stan wszystkich aktywnych portów pobiera się funkcją `GetTcpTable`. Funkcja ta zwraca obszerne informacje, które można zaprezentować użytkownikowi programu w gustownej tabeli. Tabela taka składałaby się z następujących kolumn:

- ◆ Adres lokalny — interfejs, z którego korzysta port.
 - ◆ Port lokalny — numer otwartego portu.
 - ◆ Adres zdalny — adres klienta utrzymującego połączenie z portem.
 - ◆ Port zdalny — port, za pośrednictwem którego klient komunikuje się z komputerem lokalnym.
 - ◆ Stan portu — port może pozostawać w stanie nasłuchu, zamkniętym, może przyjmować połączenie itd.

Największą zaletą korzystania z lokalnej tablicy aktywnych portów jest szybkość pozyskiwania kompletu informacji. Trwa to zaledwie parę milisekund, niezależnie od liczby aktywnych w systemie portów.

Aby sprawdzić działanie tej funkcji dla portów TCP, utwórz nowy projekt typu *MFC Application* o nazwie *IPState*. Umieść na formularzu okna głównego programu pole listy oraz przycisk z napisem *Tabela TCP*. Okno powinno docelowo wyglądać mniej więcej tak jak na rysunku 6.3.

Zdarzenie kliknięcia przycisku obsługiwane będzie kodem z listingu 6.3.

Listing 6.3. Pobieranie informacji o stanie portów TCP

```

void CIPStateDlg::OnBnClickedButton1()
{
    DWORD dwStatus = NO_ERROR;
    PMIB_TCPTABLE pTcpTable = NULL;
    DWORD dwActualSize = 0;

    dwStatus = GetTcpTable(pTcpTable, &dwActualSize, TRUE);

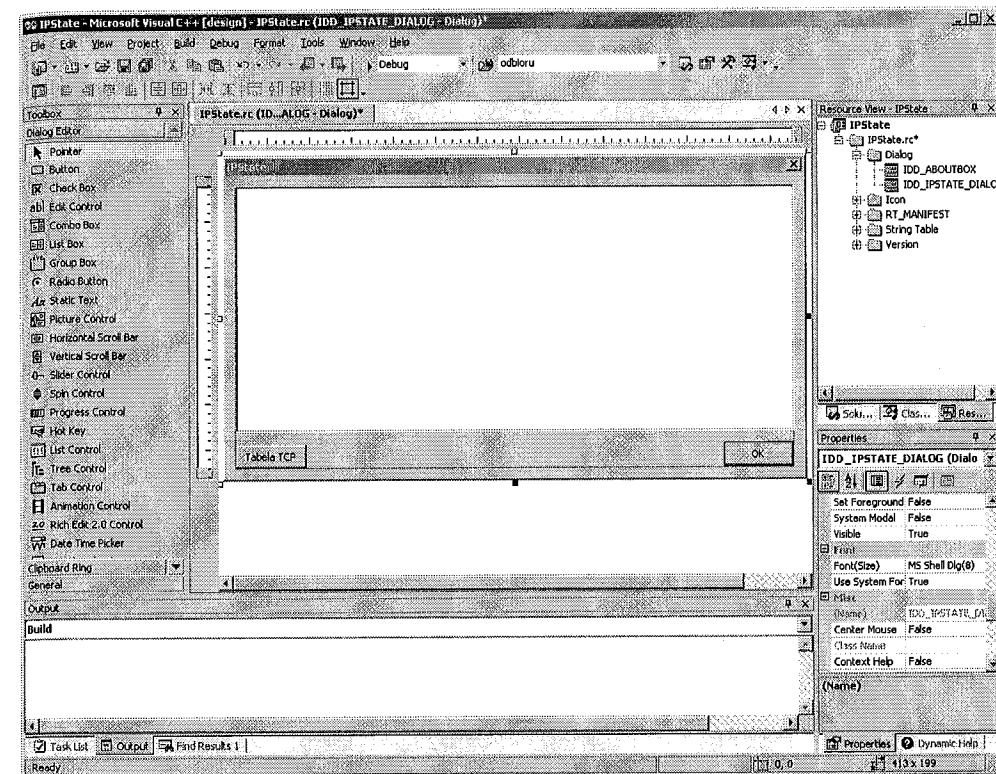
    pTcpTable = (PMIB_TCPTABLE) malloc(dwActualSize);
    assert(pTcpTable);

    dwStatus = GetTcpTable(pTcpTable, &dwActualSize, TRUE);
    if (dwStatus != NO_ERROR)
    {
        AfxMessageBox("Nie można pobrać tablicy połączeń TCP");
        free(pTcpTable);
        return;
    }

    CString strState;
    struct in_addr inadLocal, inadRemote;

```

Rozdział 6. ♦ Sztuczki, kruczki i ciekawostki



Rysunek 6.3. Projekt okna programu IPState

```
DWORD dwRemotePort = 0;
char szLocalIp[1000];
char szRemIp[1000];

if (pTcpTable != NULL)
{
    lList.AddString("=====");
    lList.AddString("Tabela po\u0144cze\u0144\u0107 TCP:");
    for (int i = 0; i < pTcpTable->dwNumEntries; i++)
    {
        dwRemotePort = 0;
        switch (pTcpTable->table[i].dwState)
        {
            case MIB_TCP_STATE_LISTEN:
                strState = "Nas\u0142uch...";
                dwRemotePort = pTcpTable->table[i].dwRemotePort;
                break;
            case MIB_TCP_STATE_CLOSED:
                strState = "Zamkni\u0144ty";
                break;
            case MIB_TCP_STATE_TIME_WAIT:
                strState = "Oczekiwani...";
```

```

        case MIB_TCP_STATE_LAST_ACK:
            strState = "Ostatni ACK...";
            break;
        case MIB_TCP_STATE_CLOSING:
            strState = "Zamykanie...";
            break;
        case MIB_TCP_STATE_CLOSE_WAIT:
            strState = "Oczekiwanie na zamknięcie...";
            break;
        case MIB_TCP_STATE_FIN_WAIT1:
            strState = "Oczekiwanie na FIN";
            break;
        case MIB_TCP_STATE_ESTAB:
            strState = "Połączony";
            break;
        case MIB_TCP_STATE_SYN_RECV:
            strState = "Odebrano SYN";
            break;
        case MIB_TCP_STATE_SYN_SENT:
            strState = "Wysłano SYN";
            break;
        case MIB_TCP_STATE_DELETE_TCB:
            strState = "Usunięty";
            break;
    }
    inadLocal.s_addr = pTcpTable->table[i].dwLocalAddr;
    inadRemote.s_addr = pTcpTable->table[i].dwRemoteAddr;
    strcpy(szLocalIp, inet_ntoa(inadLocal));
    strcpy(szRemIp, inet_ntoa(inadRemote));

    char prtStr[1000];
    sprintf(prtStr, "Adres lokalny %s; Port lokalny %u; Adres zdalny %s;
    Port zdalny %u; Stan %s",
            szLocalIp, ntohs((unsigned short)(0x0000FFFF & pTcpTable-
            >table[i].dwLocalPort)),
            szRemIp, ntohs((unsigned short)(0x0000FFFF & dwRemotePort)),
            strState);
    lList.AddString(prtStr);
}
free(pTcpTable);
}

```

Funkcja GetTcpTable przyjmuje trzy parametry:

- ◆ Wskaźnik struktury (zmienna typu PMIB_TCPTABLE).
- ◆ Rozmiar struktury wskazywanej pierwszym parametrem wywołania.
- ◆ Znacznik porządkowania wpisów tabeli — tabela może być uporządkowana według nazw portów albo pozostać nieuporządkowana.

Jeśli pierwsze dwa parametry zostaną ustalone jako zera, funkcja zwróci rozmiar pamięci niezbędnej do przechowywania struktury MIB_TCPTABLE. Podobną technikę określania rozmiaru potrzebnej pamięci stosowaliśmy w rozdziale 5.

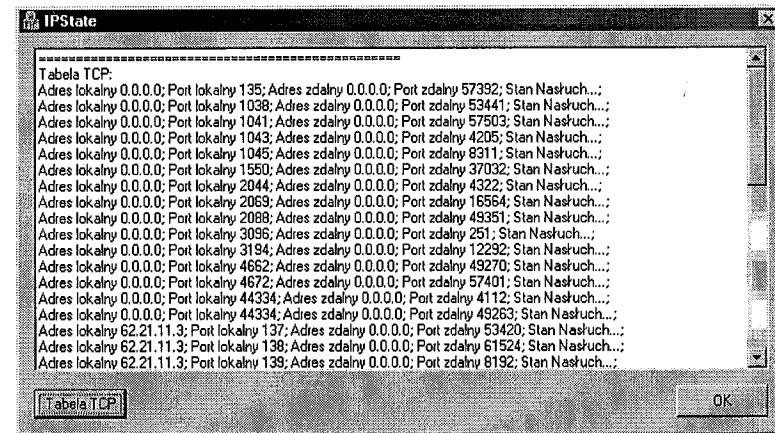
Po ustaleniu rozmiaru potrzebnej pamięci program przydziela sobie tę pamięć wywołaniem funkcji malloc. W tym przypadku nie ma potrzeby przydzielania pamięci globalnej.

W następnym wywołaniu funkcji GetTcpTable za pośrednictwem pierwszego parametru uzyskujemy informację o komplecie portów TCP (w programie jest to zmienna pTcpTable typu PMIB_TCPTABLE). Liczba opisanych w tej strukturze portów zapisywana jest w składowej dwNumEntries zwracanej struktury. Informacje o poszczególnych portach można zaś pobrać wyrażeniem table[i], gdzie table to składowa zwróconej struktury, a i to numer kolejny portu. Wyrażenie to zwraca jeszcze głębszą strukturę, w której interesują nas następujące składowe:

- ◆ dwState — stan portu. Może przyjąć rozmaite wartości (w tym MIB_TCP_STATE_LISTEN czy MIB_TCP_STATE_CLOSED). Pełnej listy stałych należy szukać w kodzie programu albo dokumentacji funkcji GetTcpTable. Ich znaczenie łatwo określić na podstawie kodu z listingu 6.3.
- ◆ dwLocalPort — port lokalny.
- ◆ dwRemotePort — port zdalny.
- ◆ dwLocalAddr — adres lokalny.
- ◆ dwRemoteAddr — adres zdalny.

Po pozyskaniu pełnej tablicy informacji o portach program przepisuje z niej dane do okna głównego, a konkretne do pola widoku listy, jak na rysunku 6.4.

Rysunek 6.4.
Efekt działania
programu IPState



Aby uniknąć błędów komplikacji, należy uzupełnić plik kodu źródłowego programu o następujące wiersze:

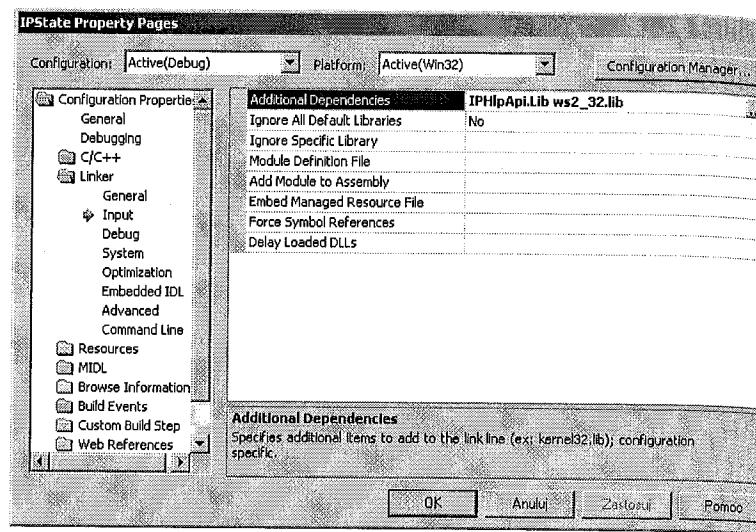
```

#include <iphlpapi.h>
#include <assert.h>
#include <winsock2.h>

```

Konieczne jest też włączenie do projektu plików bibliotek *IPHlpApi.lib* i *ws2_32.lib*. Można to zrobić za pośrednictwem pola *Additional Dependencies* w oknie właściwości projektu, w gałęzi *Linker/Input* (patrz rysunek 6.5):

Rysunek 6.5.
Włączanie do projektu potrzebnych bibliotek



Aby zaprezentować użytkownikowi tabelę portów wykorzystywanych obecnie w transmisjach UDP, należy skorzystać z funkcji GetUdpTable. Działa ona podobnie jak GetTcpTable, jednak zwracane przez nie informacje ograniczają się do adresów i portów lokalnych. Z braku połączenia w protokole UDP określenie adresu drugiej strony połączenia jest niemożliwe.

Umieść w oknie głównym programu kolejny przycisk, opatrzony etyktą *Tabela UDP*. Kod obsługi tego przycisku prezentowany jest na listingu 6.4.

Listing 6.4. Pobieranie informacji o stanie portów UDP

```
void CIPStateDlg::OnBnClickedButton2()
{
    DWORD dwStatus = NO_ERROR;
    PMIB_UDP TABLE pUdpTable = NULL;
    DWORD dwActualSize = 0;

    dwStatus = GetUdpTable(pUdpTable, &dwActualSize, TRUE);
    pUdpTable = (PMIB_UDP TABLE) malloc(dwActualSize);
    assert(pUdpTable);

    dwStatus = GetUdpTable(pUdpTable, &dwActualSize, TRUE);

    if (dwStatus != NO_ERROR)
    {
        AfxMessageBox("Nie można pobrać tabeli połączeń UDP");
        free(pUdpTable);
        return;
    }

    struct in_addr inadLocal;
    if (pUdpTable != NULL)
    {
```

```
    }List.AddString("=====");
    }List.AddString("Tabela połączeń UDP:");
    for (UINT i = 0; i < pUdpTable->dwNumEntries; ++i)
    {
        inadLocal.s_addr = pUdpTable->table[i].dwLocalAddr;

        char prtStr[1000];
        sprintf(prtStr, "Adres lokalny %s; Port lokalny %u",
            inet_ntoa(inadLocal),
            ntohs((unsigned short)(0x0000FFFF & pUdpTable-
            >table[i].dwLocalPort)));
        }List.AddString(prtStr);
    }
    free(pUdpTable);
}
```

Z racji podobieństwa (i pewnego uproszczenia) kodu pobierającego tabelę połączeń UDP do analogicznego kodu dla tabeli TCP objaśnianie tego pierwszego nie ma większego sensu.

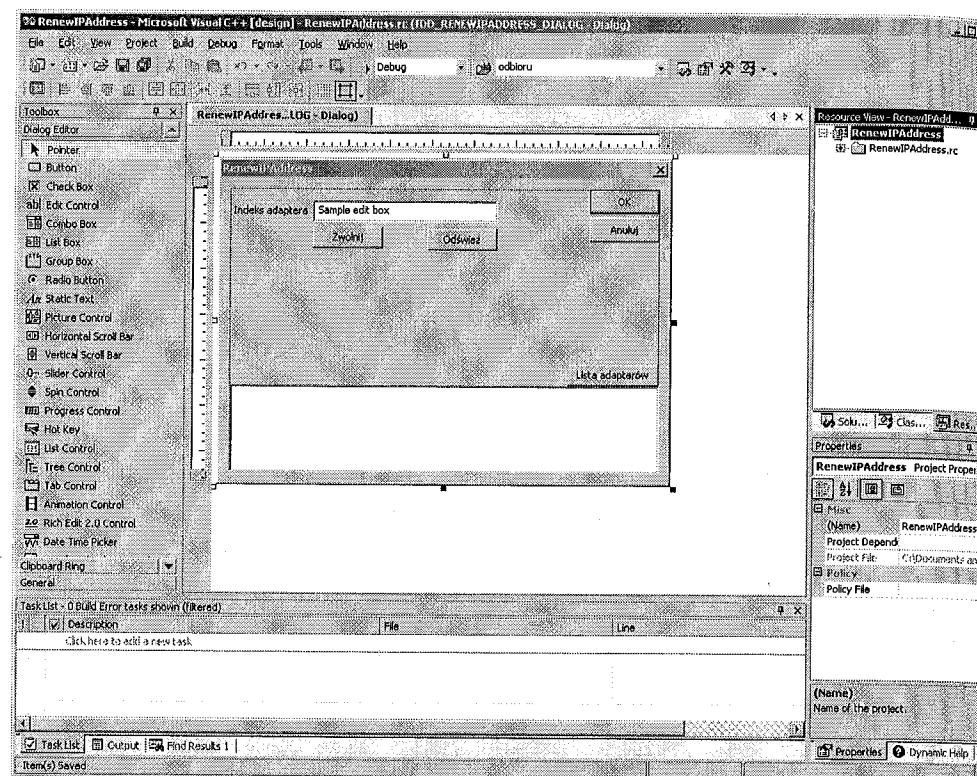
Kod źródłowy tego przykładu dostępny jest na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział6\IPState.

6.4. Serwer DHCP

Jeśli w Twojej sieci przydziałem adresów IP do komputerów zajmuje się serwer DHCP, nie możesz samowolnie nadawać komputerowi wybranych przez siebie adresów, jak pokazaliśmy to w rozdziale 5. W takim układzie adresy są zwalniane i przydzielane wyłącznie przez serwer DHCP, a ingerencji ręcznych należy zaniechać, bo powodują konflikty adresowe.

Po wyczerpaniu okresu, na jaki danemu systemowi przydzielony został adres IP (tzw. czasu dzierżawy), adres ten nie jest usuwany z systemu. System rezygnuje jedynie z jego wykorzystywania, aby serwer DHCP mógł przydzielić go innemu komputerowi. Do zwolnienia adresu przydzielonego przez serwer DHCP służy funkcja IpReleaseAddress wymagająca przekazania parametrem numeru interfejsu sieciowego. Odświeżenie adresu realizowane jest z kolei wywołaniem funkcji IpRenewAddress. Również jej parametrem jest numer interfejsu sieciowego.

Chciałbym zilustrować działanie tych funkcji na przykładzie. W tym celu powinniśmy utworzyć nowy projekt typu *MFC Application*, rozmieszczając na formularzu okna głównego elementy takie jak na rysunku 6.6. Aby zwolnić lub odświeżyć dzierżawę adresu, będziemy musieli znać indeks interfejsu sieciowego. Do jego określenia posłuży nam pole widoku listy zajmujące dolną część okna. Będzie ono — po kliknięciu przez użytkownika przycisku *Lista adapterów* — wyświetlało listę interfejsów sieciowych komputera. Kod wykonywany w reakcji na kliknięcie tego przycisku powinien być podobny do tego z listingu 5.2 — tam również program wyświetlał listę interfejsów.



Rysunek 6.6. Projekt okna głównego programu RenewIPAddress

W reakcji na kliknięcie przycisku *Zwolnij* program powinien zwolnić adres IP interfejsu. Kod obsługi tego działania widnieje na listingu 6.5.

Listing 6.5. Zwalnianie adresu IP

```
void CRenewIPAddressDlg::OnBnClickedButton2()
{
    char sAdaptIndex[20];
    int iIndex;
    sAdaptIndex.GetWindowText(sAdaptIndex, 20);
    iIndex = atoi(sAdaptIndex);

    DWORD InterfaceInfoSize = 0;
    PIP_INTERFACE_INFO pInterfaceInfo;

    if (GetInterfaceInfo(NULL, &InterfaceInfoSize) != ERROR_INSUFFICIENT_BUFFER)
    {
        AfxMessageBox("Błąd rozmiaru bufora");
        return;
    }

    if ((pInterfaceInfo
        = (PIP_INTERFACE_INFO) GlobalAlloc(GPTR, InterfaceInfoSize)) == NULL)
```

```
AfxMessageBox("Nie można przydzielić pamięci");
return;
}

if (GetInterfaceInfo(pInterfaceInfo, &InterfaceInfoSize) != 0)
{
    AfxMessageBox("Błąd funkcji GetInterfaceInfo");
    return;
}

for (int i = 0; i < pInterfaceInfo->NumAdapters; i++)
{
    if (iIndex == pInterfaceInfo->Adapter[i].Index)
    {
        if (IpReleaseAddress(&pInterfaceInfo->Adapter[i]) != 0)
        {
            AfxMessageBox("Błąd funkcji IpReleaseAddress");
            return;
        }
        break;
    }
}
```

Użytkownik zaznacza wybrany interfejs na liście w oknie głównym, a program wyodrębnia z zazначенego wiersza numer interfejsu. Potem program przegląda wszystkie interfejsy i porównuje ich indeksy z uzyskanym numerem. Jeśli porównanie wypadnie pomyślnie, następuje wywołanie funkcji zwalniającej adres IP dla danego interfejsu.

Pobranie listy interfejsów odbywa się za pośrednictwem funkcji GetInterfaceInfo. Działa ona podobnie jak znana nam już funkcja GetAdaptersInfo. Przy pierwszym wywołaniu, z wyzerowanym pierwszym parametrem, funkcja zwraca rozmiar pamięci potrzebnej do przechowywania listy. Program przydziela sobie odpowiednią ilość pamięci i drugim wywołaniem pobiera właściwą listę.

Lista jest przeglądana w pętli. Jeśli w którymś z jej przebiegów uda się znaleźć zaznaczony przez użytkownika interfejs, program wywołuje dla tego interfejsu funkcję zwalniającą adres IP.

Podobnie realizowane jest odnowienie dzierżawy adresu IP. Kod obsługi przycisku *Odnów* pokazany jest na listingu 6.6.

Listing 6.6. Odnawianie adresu IP

```
void CRenewIPAddressDlg::OnBnClickedButton3()
{
    char sAdaptIndex[20];
    int iIndex;
    sAdaptIndex.GetWindowText(sAdaptIndex, 20);
    iIndex = atoi(sAdaptIndex);

    DWORD InterfaceInfoSize = 0;
    PIP_INTERFACE_INFO pInterfaceInfo;
```

```

if (GetInterfaceInfo(NULL, &InterfaceInfoSize) !=  

    ERROR_INSUFFICIENT_BUFFER)  

{  

    AfxMessageBox("Błąd rozmiaru bufora");  

    return;  

}  

if ((pInterfaceInfo  

    = (PIP_INTERFACE_INFO) GlobalAlloc(GPTR, InterfaceInfoSize)) == NULL)  

{  

    AfxMessageBox("Nie można przydzielić pamięci");  

    return;  

}  

if (GetInterfaceInfo(pInterfaceInfo, &InterfaceInfoSize) != 0)  

{  

    AfxMessageBox("Błąd funkcji GetInterfaceInfo");  

    return;  

}  

for (int i = 0; i < pInterfaceInfo->NumAdapters; i++)  

    if (iIndex == pInterfaceInfo->Adapter[i].Index)  

    {  

        if (IpRenewAddress(&pInterfaceInfo->Adapter[i]) != 0)  

        {  

            AfxMessageBox("Błąd funkcji IpRenewAddress");  

            return;  

        }
        break;
    }
}

```

Kod odnawiający dzierżawę adresu IP różni się od kodu zwalniającego adres wyłącznie nazwą funkcji wywoływanej w pętli — tutaj jest to `IpRenewAddress`, poprzednio wywoływaliśmy zaś `IpReleaseAddress`.



Kod źródłowy tego przykładu dostępny jest na dołączonej do książki płycie CD-ROM w podkatalogu `\Przykłady\Rozdział6\RenewIPAddress`.

6.5. Protokół ICMP

Wspominałem już, że protokół IP nie gwarantuje niezawodnego dostarczania pakietów pomiędzy węzłami, nie dysponuje też mechanizmami kontroli spójności transmitowanych danych. Można jednak łatwo sprawdzić, czy pakiet dotarł do węzła przeznaczenia, korzystając z protokołu kontrolnego o nazwie ICMP (ang. *Internet Control Message Protocol*). Pakiety protokołu ICMP są wysyłane automatycznie, kiedy węzeł docelowy jest niedostępny, bufor bramy ulegnie przepchnieniu albo jest zbyt mały dla transmisji lub kiedy węzeł docelowy wymaga przesłania pakietu krótszą trasą.

Protokół ICMP został opracowany dla potrzeb informowania uczestników transmisji o problemach pojawiających się w trakcie nadawania albo odbioru danych; jego zastosowanie zwiększa niezawodność transmisji protokołem IP. Nie należy jednak przeceniać protokołu ICMP, ponieważ w przypadku utraty danych (zagubienia pakietu w sieci) nie otrzymamy o tym żadnej informacji. Celem zagwarantowania pewności doręczenia i kontroli spójności danych należy wdrożyć do transmisji protokoły wyższych warstw, jak choćby TCP.

Jeśli, bazując na IP, chcesz opracować własny protokół komunikacyjny, możesz zwiększyć jego niezawodność w stosunku do „gołego” IP, odpowiednio korzystając z komunikatorów protokołu ICMP. Warto jednak pamiętać, że nie będzie to zabezpieczenie wystarczające. Pakiety ICMP sygnalizują sytuacje takie jak niemożność przetworzenia transmisji przez bramę albo adresata. Jednak jeśli pakiet nie dotrze do odbiorcy z powodu np. zerwania fizycznego połączenia sieciowego, nadawca nie otrzyma żadnych sygnałów, ponieważ protokół IP nie korzysta z potwierdzeń.

Z racji niezupełnej niezawodności protokołu ICMP programiści rzadko korzystają z niego bezpośrednio (na przykład do sprawdzania doręczenia pakietu do adresata). Protokół ten jest jednak wykorzystywany powszechnie w innym celu. Mianowicie, jeśli wysłany do adresata pakiet ICMP osiągnie węzeł docelowy, węzeł ten powinien odpowiedzieć kolejnym pakietem ICMP. To z kolei pozwala na sprawdzanie jakości połączenia pomiędzy dwoma komputerami. Test taki implementuje popularny program `ping`.

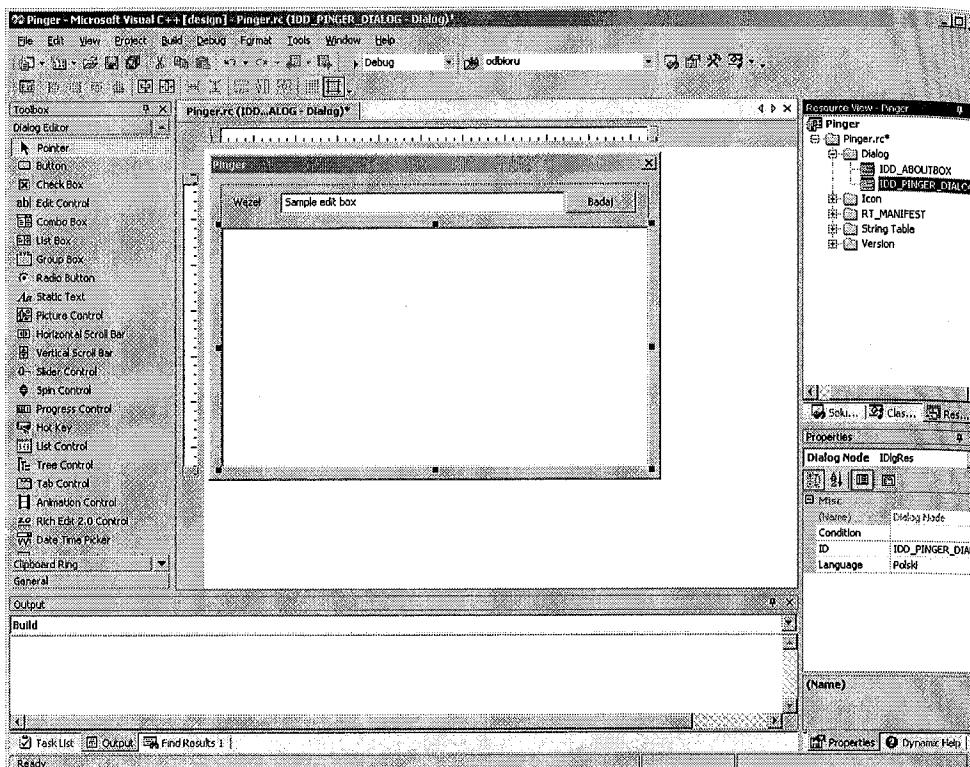
Jeśli znamy numer jednego z otwartych portów komputera docelowego, możemy przetestować możliwość ustanowienia połączenia, wysyłając żądanie jego nawiązania. Jeśli uda się je nawiązać, uznajemy komputer za dostępny. W przypadku nieznajomości numeru portu możemy uciec się do skanowania komputera, jest to jednak czynność czasochłonna. Można tego uniknąć, korzystając z funkcji protokołu ICMP.

W niektórych sieciach komputerowych wszystkie komputery są chronione zaparami sieciowymi, które blokują pakiety protokołu ICMP. Administratorzy tych sieci, aby nie blokować całkowicie możliwości testowania dostępności komputera, udostępniają zazwyczaj pojedynczy port z tzw. serwerem „echo” (serwerem, który odsyła do klienta wszystkie dane, które od niego odbierze). To bardzo dobra metoda, może się jednak zdarzyć, że program serwera „echo” zostanie zawieszony albo zamknięty — wtedy, mimo że komputer jest w sieci dostępny, nie ma jak tego stwierdzić. Można wtedy dojść do mylnego wniosku o fizycznym uszkodzeniu połączenia sieciowego. Takie konfiguracje sieci są jednak dość rzadkie i w większości przypadków protokołem ICMP można skutecznie testować możliwość ustanowienia połączenia.

Theoretycznie rzecz wygląda nieskomplikowanie, w praktyce pojawia się jednak problem. Otóż pakiety protokołu ICMP różnią się od pakietów TCP i datagramów UDP obsługiwanych przez bibliotekę WinSock. Jak więc skonstruować i wysłać pakiet protokołu kontrolnego? Cóż, trzeba zrobić to samodzielnie.

Pierwsza wersja biblioteki gniazd nie pozwalała programiście na odwoływanie się wprost do danych pakietu. Drugi parametr funkcji `select` mógł wtedy przyjmować jedynie jedną z dwóch wartości: `SOCK_STREAM` (dla protokołu TCP) i `SOCK_DGRAM` (dla protokołu UDP). W Winsock2 wprowadzono jednak obsługę gniazd „surowych”, dających niskopoziomowy dostęp do pakietów. Gniazdo takie tworzy się wywołaniem funkcji `socket` z drugim parametrem o wartości `SOCK_RAW`.

Weźmy się więc za implementację programu *Pinger*. Będzie to znakomitą ilustracją działania i obsługi gniazd „surowych” i kontroli połączeń za pośrednictwem protokołu ICMP. Utwórz w tym celu nowy projekt (*MFC Application*). Okno główne aplikacji skomponuj na wzór tego z rysunku 6.7.



Rysunek 6.7. Projekt okna głównego programu *Pinger*

Pole edycyjne opatrzone etykietą *Węzeł* będzie wyświetlać adres IP albo nazwę komputera docelowego. Po kliknięciu przycisku *Badaj* program powinien najpierw wysłać, a potem odebrać pakiet ICMP, a wynik jego analizy wyświetlić w polu listy zajmującego dolną część okna.

Za reakcję na naciśnięcie przycisku *Badaj* odpowiedzialny jest kod z listingu 6.7.

Listing 6.7. Korzystanie z pakietów ICMP

```
void CPingerDlg::OnBnClickedButton1()
{
    SOCKET rawSocket;
    LPHOSTENT lpHost;
    struct sockaddr_in sDest;
    struct sockaddr_in sSrc;
    DWORD dwElapsed;
    int iRet;
    CString str;
```

```
WSADATA wsd;
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    AfxMessageBox("Nie można wczytać biblioteki WinSock");
    return;
}

// Utwórz surowe gniazdo
rawSocket = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
if (rawSocket == SOCKET_ERROR)
{
    AfxMessageBox("Błąd tworzenia gniazda");
    return;
}

// Wyszukiwanie węzła
char strHost[255];
edHost.GetWindowText(strHost, 255);
lpHost = gethostbyname(strHost);
if (lpHost == NULL)
{
    AfxMessageBox("Nie znaleziono węzła");
    return;
}

// Adres gniazda
sDest.sin_addr.s_addr = *(u_long FAR *) (lpHost->h_addr);
sDest.sin_family = AF_INET;
sDest.sin_port = 0;

str.Format("Badanie %s [%s]",
           strHost, inet_ntoa(sDest.sin_addr));

lMessages.AddString(str);

// Skonstruuj komunikat echo protokołu ICMP
static ECHOREQUEST echoReq;

echoReq.icmpHdr.Type      = ICMP_ECHOREQ;
echoReq.icmpHdr.Code       = 0;
echoReq.icmpHdr.ID         = 0;
echoReq.icmpHdr.Seq        = 0;
echoReq.dwTime = GetTickCount();
FillMemory(echoReq.cData, 64, 80);
echoReq.icmpHdr.Checksum = CheckSum((u_short *)echoReq, sizeof(ECHOREQUEST));

// Wyślij komunikat echo
sendto(rawSocket, (LPSTR)&echoReq, sizeof(ECHOREQUEST),
       0, (LPSOCKADDR)&sDest, sizeof(SOCKADDR_IN));

struct timeval tVal;
fd_set readfds;
readfds.fd_count = 1;
readfds.fd_array[0] = rawSocket;
tVal.tv_sec = 1;
tVal.tv_usec = 0;
```

```

iRet=select(1, &readfds, NULL, NULL, &tVal);

if (!iRet)
{
    tMessages.AddString("Upłynął czas oczekiwania");
}
else
{
    // Odbierz odpowiedź
    ECHOREPLY echoReply;
    int nRet;
    int nAddrLen = sizeof(struct sockaddr_in);

    iRet = recvfrom(rawSocket, (LPSTR)&echoReply,
                    sizeof(ECHOREPLY), 0, (LPSOCKADDR)&sSrc, &nAddrLen);

    if (iRet == SOCKET_ERROR)
        AfxMessageBox("Błąd funkcji recvfrom");

    // Oblicz czas wędrówki pakietów
    dwElapsed = GetTickCount() - echoReply.echoRequest.dwTime;
    str.Format("Odpowiedź z: %s bajtów=%d czas=%ldms TTL=%d",
              inet_ntoa(sSrc.sin_addr), 64, dwElapsed, echoReply.ipHdr.TTL);
    tMessages.AddString(str);
}

iRet = closesocket(rawSocket);
if (iRet == SOCKET_ERROR)
    AfxMessageBox("Błąd zamknięcia gniazda");

WSACleanup();
}

```

Przed odwołaniem się do pierwszej funkcji sieciowej należy wczytać do programu bibliotekę gniazd. Służy do tego, tradycyjnie, funkcja WSASStartup. Biblioteka ta jest potrzebna również przy korzystaniu z gniazd „surowych”. Warto jedynie pamiętać, że tego rodzaju gniazda obsługiwane są dopiero w drugiej wersji biblioteki gniazd.

Po wczytaniu biblioteki można wreszcie utworzyć gniazdo, wywołując funkcję socket i przekazując do niej następujące parametry:

- ◆ Identyfikator rodziny protokołów — AF_INET (jak zawsze).
- ◆ Typ gniazda — SOCK_RAW dla gniazda „surowego”.
- ◆ Identyfikator protokołu — IPPROTO_ICMP.

Aby wysłać pakiet do innego komputera, trzeba znać jego adres IP. Jeśli użytkownik określi adresata za pośrednictwem nazwy symbolicznej, trzeba ją skonwertować na postać adresu IP wywołaniem funkcji gethostbyname.

Po ustaleniu adresu IP można przystąpić do wypełniania struktury adresowej sockaddr_in. Ponieważ protokół ICMP nie korzysta z portów, składową sin_port należy ustawić na zero.

Następnie należy wypełnić strukturę ECHOREQUEST reprezentującą pakiet ICMP. Zawartość tej struktury zostanie przesłana siecią. Podczas gdy protokoły TCP czy UDP zwalniają programistę od konstruowania nagłówków pakietów, wymagając jedynie podania właściwych danych do przesłania, korzystanie z protokołu ICMP za pośrednictwem surowego gniazda wymaga własnoręcznego skonstruowania całego pakietu wraz z właściwymi dla protokołu ICMP nagłówkami. Struktura ECHOREQUEST prezentuje się następująco:

```

typedef struct ECHOREQUEST
{
    ICMPHDR icmpHdr;
    DWORD dwTime;
    char cData[64];
} ECHOREQUEST, *ECHOREQUEST;

```

Składowa icmpHdr to nagłówek pakietu, który trzeba wypełnić samodzielnie; cData to właściwe dane pakietu. W naszym przykładzie będziemy przesyłać pakiet o rozmiarze 64 bajtów, więc na potrzeby struktury deklarowanej jest 64-elementowa tablica znaków. Tablica ta jest przez funkcję FillChar w całości wypełniana znakami o wartości 80. W przypadku programu badającego dostępność węzła dane niesione pakietem są bowiem zupełnie nieistotne — adresat nie będzie ich никак przetwarzał, a jego reakcją powinna być jedynie natychmiastowa odpowiedź na pakiet.

Pole dwTime będzie przechowywać informacje o czasie. Można je wykorzystać dowolnie. Większość programistów korzysta z jego wartości podczas analizy czasu odpowiedzi badanego węzła, określając na jego podstawie czas wędrówki pakietu do adresata.

Zawartość nagłówka pakietu ICMP jest zależna od rodzaju komunikatu. Ponieważ analizujemy program badający dostępność węzła zdalnego, nie będę omawiał wszystkich dostępnych funkcji ICMP. Czytelników zainteresowanych pełnym opisem odsyłam do najpewniejszego źródła, którym jest dokument RFC 792 (dostępny np. pod adresem <http://www.faqs.org/rfcs/rfc792.html>). Nagłówek pakietu ICMP to kolejna struktura:

```

typedef struct tagICMPHDR
{
    u_char Type;
    u_char Code;
    u_short Checksum;
    u_short ID;
    u_short Seq;
    char Data;
} ICMPHDR, *PICMPHDR;

```

Przyjrzyjmy się bliżej jej składowym:

- ◆ Type — typ pakietu. W naszym przykładzie jest to ICMP_ECHOREQ, czyli pakiet wymuszający na odbiorcy odesłanie otrzymanych danych. W nagłówku odpowiedzi składowa ta powinna być wyzerowana.
- ◆ Code — niewykorzystywana w komunikatach „echo” i jako taka zerowana.
- ◆ Checksum — suma kontrolna. Dokument RFC nie określa ścisłych wymogów co do algorytmu obliczania sumy kontrolnej. W naszym programie zastosowaliśmy algorytm mocno uproszczony, prezentowany na listingu 6.8.

Listing 6.8. Funkcja obliczająca sumę kontrolną

```
u_short CheckSum(u_short *addr, int len)
{
    register int nleft = len;
    register u_short answer;
    register int sum = 0;

    while(nleft > 1) {
        sum += *addr++;
        nleft -= 1;
    }

    sum += (sum >> 16);
    answer = ~sum;
    return (answer);
}
```

- ◆ ID — w komunikacie „echo” należy wyzerować.
- ◆ Seq — numer kolejki. Dla pola wyzerowanego pola Code powinna tu być wartość zero.

Po skonstruowaniu pakietu wysyłamy go do adresata, korzystając z wywołania funkcji sendto. Stosujemy sendto zamiast send, ponieważ protokół IP jest protokołem bezpołączniowym — podobnie jak UDP i odwrotnie niż TCP.

Na odpowiedź oczekuje funkcja select. Jeśli w ciągu pierwszej sekundy oczekiwania nie doczekamy się odpowiedzi, możemy uznać badany komputer za niedostępny. W innym przypadku odebrany pakiet jest odczytywany. Odczyt nie jest co prawda absolutnie konieczny, ponieważ samo jego dotarcie do komputera lokalnego jest dostatecznym dowodem dostępności komputera zdalnego. Dokonamy jednak odbioru choćby po to, aby zaprezentować sposób wymiany danych za pośrednictwem gniazd surowych. W prawdziwym programie należałoby zresztą odczytać pakiet choćby po to, aby przekonać się, że został przesyłany z komputera, do którego wysyłaliśmy pakiet ICMP (jest bowiem całkiem prawdopodobne, że w międzyczasie to nasz komputer stanie się obiektem badania pakietami ICMP). Odczyt pakietu jest też konieczny, jeśli program dokonuje badania kilku węzłów równocześnie.

Do odczytu pakietu ICMP służy funkcja recvfrom — dokładnie tak jak przy odczytce datagramów protokołu UDP. Choć wysyłaliśmy pakiet ECHOREQUEST, w odpowiedzi powinniśmy otrzymać pakiet o strukturze zgodnej ze strukturą ECHOREPLY, której skład jest następujący:

```
typedef struct tagECHO_REPLY
{
    IPHDR ipHdr;
    ECHOREQUEST echoRequest;
    char cFiller[256];
} ECHO_REPLY, *PECHO_REPLY;
```

Pierwsza składowa struktury to nagłówek pakietu odebranego, druga to powielona w pakiecie odebranym struktura nagłówka pakietu wysłanego.

Nagłówek odebranego pakietu różni się od nagłówka pakietu wysłanego:

```
typedef struct tagIPHDR
{
    u_char VHL;
    u_char TOS;
    short TotLen;
    short ID;
    short FlagOff;
    u_char TTL;
    u_char Protocol;
    u_short Checksum;
    struct in_addr iaSrc;
    struct in_addr iaDst;
} IPHDR, *PIPHDR;
```

To nic innego jak nagłówek protokołu IP.

Wszystkie wymagane do działania programu struktury należy zadeklarować w pliku nagłówkowym. Na potrzeby tego programu umieściłem te deklaracje w pliku *PingerDlg.h*.



Kod źródłowy tego przykładu dostępny jest na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział6\Pinger.

6.6. Śledzenie trasy wędrówki pakietu

Czy można prześledzić drogę pakietu pomiędzy nadawcą a adresatem? Można, o ile wykorzysta się funkcje protokołu kontrolnego ICMP. Otóż każdy pakiet przesyłany w sieci posiada parametr TTL (ang. *Time To Live* — czas życia pakietu). Każdy router na drodze pakietu zmniejsza wartość tego pola, a kiedy zostanie ona wyzerowana, router, który tego dokona, wysyła do nadawcy (za pośrednictwem protokołu ICMP) komunikat o błędzie, uznając pakiet za utracony.

Mogą więc spróbować wysłać do adresata pakiet, którego parametr TTL ma wartość 1. Pierwszy router na jego drodze zmniejszy TTL do zera. Zauważyszy to wyśle do nas komunikat ICMP. W ten sposób poznamy adres pierwszego routera na drodze pakietu do adresata. Potem możemy spróbować wysłać pakiet z TTL równym 2, próbując określić adres drugiego routera na drodze pakietu (pierwszy router prześle pakiet dalej, drugi zaś zwróci komunikat ICMP). W ten sposób, zwiększając stopniowo parametr czasu życia pakietu, można wyznaczyć kompletną trasę pakietu od nadawcy do adresata. Przy okazji można zmierzyć czasy odpowiedzi poszczególnych routerów i próbować na podstawie pomiarów wyznaczyć najslabsze ogniwo na trasie pakietu. Trzeba jednak wtedy pamiętać, że na takie pomiary spory wpływ ma chwilowe obciążenie routera, należałoby więc powtórzyć testy kilkukrotnie i wyznaczyć średni czas odpowiedzi.

Jest też prawdopodobne, że pierwszy wysłany pakiet podąży inną trasą niż drugi. Sytuacja taka jest jednak dość rzadka.

W tym podrozdziale zajmiemy się więc implementacją prostego programu śledzącego trasę pakietu w sieci. Tym razem jednak komunikaty ICMP będziemy wysyłać nie za pośrednictwem surowych gniazd, ale za pośrednictwem funkcji biblioteki *icmp.dll*. Zawiera ona komplet funkcji niezbędnych do tworzenia gniazda nadającego się do transmisji ICMP i funkcji konstruujących pakietu. Pozwoli to na ograniczenie konstrukcji pakietu do określenia adresu odbiorcy i podstawowych parametrów pakietu — resztą zajmie się kod biblioteki *icmp.dll*.

Utwórz nowy projekt typu *MFC Application* o nazwie *TraceRoute*. Formularz jego okna głównego powinien zawierać jedno pole edycyjne *Nazwa węzła* (do wpisywania adresów), jedno pole listy (do wypisywania wyników śledzenia) oraz przycisk *Badaj* (uruchamiający badanie trasy). W reakcji na kliknięcie tego ostatniego program powinien wykonać kod z *listingu 6.9*.

Listing 6.9. Śledzenie trasy wędrówki pakietu

```
void CTraceRouteDlg::OnBnClickedButton1()
{
    WSADATA wsa;
    if (WSAStartup(MAKEWORD(1, 1), &wsa) != 0)
    {
        AfxMessageBox("Nie można wczytać właściwej wersji biblioteki WinSock");
        return;
    }

    hIcmp = LoadLibrary("ICMP.DLL");
    if (hIcmp == NULL)
    {
        AfxMessageBox("Nie można wczytać biblioteki ICMP");
        return;
    }

    pIcmpCreateFile = (lpIcmpCreateFile) GetProcAddress(hIcmp, "IcmpCreateFile");
    pIcmpSendEcho = (lpIcmpSendEcho) GetProcAddress(hIcmp, "IcmpSendEcho");
    pIcmpCloseHandle = (lpIcmpCloseHandle) GetProcAddress(hIcmp, "IcmpCloseHandle");

    in_addr Address;
    if (pIcmpCreateFile == NULL)
    {
        AfxMessageBox("Błąd biblioteki ICMP");
        return;
    }

    char chHostName[255];
    edHostName.GetWindowText(chHostName, 255);
    LPHOSTENT hp = gethostbyname(chHostName);
    if (hp == NULL)
    {
        AfxMessageBox("Nie znaleziono węzła");
        return;
    }
    unsigned long addr;
    memcpy(&addr, hp->h_addr, hp->h_length);

    BOOL bReachedHost = FALSE;
```

```
for (UCHAR i = 1; i <= 50 && !bReachedHost; i++)
{
    Address.S_un.S_addr = 0;

    int iPacketSize=32;
    int iRTT;

    HANDLE hIP = pIcmpCreateFile();
    if (hIP == INVALID_HANDLE_VALUE)
    {
        AfxMessageBox("Nie można utworzyć uchwytu ICMP");
        return;
    }

    unsigned char* pBuf = new unsigned char[iPacketSize];
    FillMemory(pBuf, iPacketSize, 80);

    int iReplySize = sizeof(ICMP_ECHO_REPLY) + iPacketSize;
    unsigned char* pReplyBuf = new unsigned char[iReplySize];
    ICMP_ECHO_REPLY* pEchoReply = (ICMP_ECHO_REPLY*) pReplyBuf;

    IP_OPTION_INFORMATION ipOptionInfo;
    ZeroMemory(&ipOptionInfo, sizeof(IP_OPTION_INFORMATION));
    ipOptionInfo.Ttl = i;

    DWORD nRecvPackets = pIcmpSendEcho(hIP, addr, pBuf, iPacketSize,
                                         &ipOptionInfo, pReplyBuf,
                                         iReplySize, 30000);

    if (nRecvPackets != 1)
    {
        AfxMessageBox("Nie można zbadać węzła");
        return;
    }
    Address.S_un.S_addr = pEchoReply->Address;
    iRTT = pEchoReply->RoundTripTime;

    pIcmpCloseHandle(hIP);

    delete [] pReplyBuf;
    delete [] pBuf;

    char lpszText[255];

    hostent* phostent = NULL;
    phostent = gethostbyaddr((char *)&Address.S_un.S_addr, 4, PF_INET);

    if (phostent)
        sprintf(lpszText, "%d: %d ms [%s] (%d.%d.%d.%d)",
                i, iRTT,
                phostent->h_name, Address.S_un.S_un_b.s_b1,
                Address.S_un.S_un_b.s_b2,
                Address.S_un.S_un_b.s_b3, Address.S_un.S_un_b.s_b4);
    else
        sprintf(lpszText, "%d - %d ms (%d.%d.%d.%d)",
                i, iRTT,
```

```

Address.S_un.S_un_b.s_b1, Address.S_un.S_un_b.s_b2,
Address.S_un.S_un_b.s_b3, Address.S_un.S_un_b.s_b4);

    1bMessages.AddString(1pszText);

    if (addr == Address.S_un.S_addr)
        bReachedHost = TRUE;
    }

    if (hIcmp)
    {
        FreeLibrary(hIcmp);
        hIcmp = NULL;
    }

    WSACleanup();
}

```

Mimo zastosowania biblioteki *icmp.dll* nie można zaniedbać wczytania biblioteki gniazd. Będzie ona potrzebna choćby do realizacji wywołania funkcji *gethostbyname*, które określi adres IP badanego węzła, jeśli użytkownik określi ten węzeł nazwą symboliczną. Całkiem wystarczająca będzie przy tym pierwsza wersja tej biblioteki, ponieważ tym razem nie będziemy korzystać z gniazd typu RAW. Pozwoli to też na uruchomienie programu w systemie Windows 98 (który nie dysponuje domyślnie biblioteką Winsock2).

Po wczytaniu biblioteki gniazd należy za pomocą funkcji *LoadLibrary* wczytać do pamięci programu również bibliotekę *icmp.dll*. Biblioteka, o której nam chodzi, jest przechowywana w katalogu bibliotek systemu Windows (*Windows\System* albo *Windows\System32*), nie trzeba więc podawać w wywołaniu pełnej ścieżki do pliku biblioteki. Program sam znajdzie i wczyta bibliotekę.

Biblioteka ta interesuje nas ze względu na następujące funkcje:

- ◆ *IcmpCreateFile* — funkcja inicjalizująca „gniazdo” ICMP.
- ◆ *IcmpSendEcho* — funkcja wysyłająca pakiet ICMP z poleceniem „echo”.
- ◆ *IcmpCloseHandle* — funkcja zamkająca „gniazdo” ICMP.

Przed wysłaniem pierwszego pakietu należy zainicjalizować gniazdo ICMP wywołaniem funkcji *IcmpCreateFile*. Po zakończeniu korzystania z biblioteki należy zaś je zamknąć, uciekając się do wywołania *IcmpCloseHandle*.

Następnie warto pozyskać do uprzednio zadeklarowanych zmiennych lokalnych adresy tych funkcji w bibliotece:

```

pIcmpCreateFile = (1pIcmpCreateFile) GetProcAddress(hIcmp, "IcmpCreateFile");
pIcmpSendEcho = (1pIcmpSendEcho) GetProcAddress(hIcmp, "IcmpSendEcho" );
pIcmpCloseHandle = (1pIcmpCloseHandle) GetProcAddress(hIcmp, "IcmpCloseHandle");

```

Aby upewnić się co do poprawności wykonania programu, należy jeszcze sprawdzić, czy aby żaden ze wskaźników funkcji biblioteki nie ma wartości NULL. Jeśli którykolwiek z nich jest pomimo powyższych wywołań wskaźnikiem pustym, program nie będzie

w stanie odwołać się do danej funkcji. Błąd tego rodzaju najczęściej wynika z podania nieprawidłowej funkcji w wywołaniu *GetProcAddress*. Może się też jednak zdarzyć, że nazwy są poprawne, za to do pamięci została wczytana biblioteka o takiej jak podana nazwie, ale pozbawiona szukanych funkcji. Aby wykryć taką sytuację, należy porównać wskaźnik *pIcmpCreateFile* (który powinien zawierać już adres funkcji *IcmpCreateFile*) z wartością NULL. Jeśli wartości te będą równe, można wnioskować o wczytaniu niewłaściwej biblioteki. Program wyświetli stosowny komunikat i nie będzie już sprawdzał pozostałych dwóch wskaźników funkcji (zakładając, że przy właściwej bibliotece ich wartości będą poprawne).

Następny etap badania węzła polega na odczytaniu wprowadzonego przez użytkownika adresu węzła i jego konwersji na format IP. Jeśli przy tej okazji dojdzie do błędu, to z braku poprawnego adresu nie będzie można kontynuować badania.

Dysponując adresem, można przystąpić do badania węzła zdalnego. Ponieważ badanie polega na wysyłaniu pakietów o różnych wartościach TTL, jest realizowane w pętli, której zmienne sterująca (licznik pętli) zmienia się od 1 do 50. Nie zalecam tym razem stosowania pętli *while*, kontynuowanej aż do osiągnięcia przez któryś z kolejnych pakietów adresu docelowego, ponieważ może się zdarzyć, że adresat jest niedostępny i program będzie wykonywał pętlę w nieskończoność.

Wewnątrz pętli następuje inicjalizacja „gniazda” ICMP. Służy do tego funkcja *IcmpCreateFile*. Zwraca ona uchwyt nowo utworzonego obiektu, za pośrednictwem którego będziemy później wysyłać pakiet z poleceniem „echo”. Uchwyt zapisujemy w zmiennej *hIP* typu HANDLE:

```

HANDLE hIP = pIcmpCreateFile();
if (hIP == INVALID_HANDLE_VALUE)
{
    AfxMessageBox("Nie można utworzyć uchwytu ICMP");
    return;
}

```

Jeśli wartością funkcji jest *INVALID_HANDLE_VALUE*, mamy błąd inicjalizacji i kontynuowanie badania jest bezcelowe.

Kolejną czynnością jest przydział pamięci dla bufora danych. Podobnie jak w programie *Pinger*, bufor ten jest wypełniany znakami o wartości 80. Następnie tworzony jest pakiet typu *ICMP_ECHO_REPLY*. Będzie on wykorzystywany do reprezentowania odpowiedzi otrzymanej od routera pośredniczącego w transmisji do węzła zdalnego. Tworzona jest też struktura *IP_OPTION_INFORMATION*, w której interesuje nas składowa *Ttl* — będziemy tu ustawać parametr TTL pakietu.

Kiedy wszystko jest gotowe, można wysłać pakiet ICMP, wywołując funkcję *IcmpSendEcho*. Przyjmuje ona następujące parametry:

- ◆ Uchwyt ICMP (uzyskany jako wartość zwracana funkcji *IcmpCreateFile*).
- ◆ Adres węzła zdalnego.
- ◆ Bufor z danymi pakietu.
- ◆ Rozmiar pakietu (z rozmiarem danych włącznie).

- ◆ Pakiet IP z ustawionym parametrem TTL (o wartości jeden w pierwszym przebiegu pętli, dwa w drugim itd.).
- ◆ Bufor struktury ICMP_ECHO_REPLY dla odpowiedzi routera.
- ◆ Rozmiar bufora odpowiedzi.
- ◆ Limit czasu oczekiwania na odpowiedź.

Funkcja zwraca liczbę odebranych pakietów. W naszym przypadku odpowiedź powinna składać się z jednego pakietu. Gdyby wartością zwracaną okazało się zero, oznaczałoby to, że jeden z routerów na drodze pakietu (albo sam węzeł docelowy) nie odpowiedział na komunikat ICMP. Nie będzie więc można wyznaczyć parametrów czasowych odpowiedzi od tego routera.

Czas odpowiedzi można określić na podstawie składowej RoundTripTime struktury ICMP_ECHO_REPLY. Adres urządzenia sieciowego, które odpowiedziało na pakiet, zapisany jest w składowej Address.

Po zakończeniu badania nie wolno zapominać o zamknięciu uchwytu „gniazda” ICMP przez wywołanie funkcji IcmpCloseHandle.

Teraz można przystąpić do formatowania i wyświetlania pozyskanych informacji. Aby wyniki były bardziej zrozumiałe dla użytkownika, można dokonać konwersji otrzymanych adresów IP na postać nazw symbolicznych. Służy do tego funkcja get-hostbyaddr. Wymaga ona przekazania w wywołaniu trzech parametrów:

- ◆ Adresu IP komputera, którego nazwę chcemy określić.
- ◆ Długości przekazanego adresu.
- ◆ Identyfikatora rodziny protokołów (zależy od niego format adresu).

Jeśli program wykryje odpowiedź od węzła docelowego, pętla jest przerywana. W przeciwnym razie następuje kolejny przebieg pętli, zwiększenie parametru TTL i wysłanie kolejnego komunikatu ICMP:

```
if (addr == Address.S_un.S_addr)
    bReachedHost = TRUE;
```

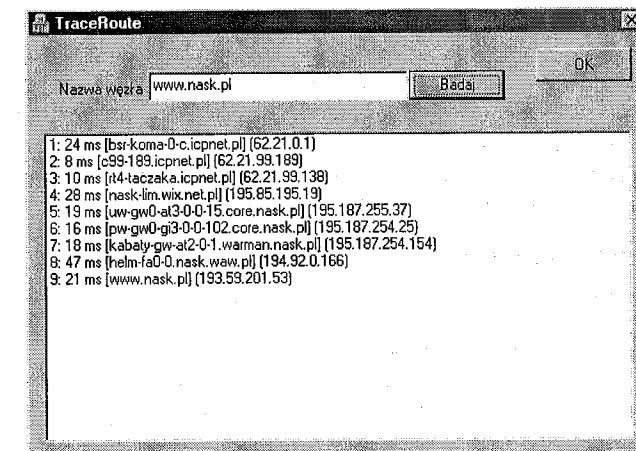
Po wyjściu z pętli należy jeszcze usunąć z pamięci wykorzystywane biblioteki:

```
{
    FreeLibrary(hIcmp);
    hIcmp = NULL;
}

WSACleanup();
```

Po uruchomieniu programu *TraceRoute* i kliknięciu przycisku *Badaj* powinieneś zobaczyć okno z opisem trasy pakietów podobne do tego z rysunku 6.8.

Rysunek 6.8.
Efekt działania programu *TraceRoute*



Dołożyłem starań, aby przykład był jak najprostszy i aby można było go bez problemu przeanalizować. W wyniku tego program przykładowu ma pewną słabość.

Otoż, jeśli w czasie wykonania programu dojdzie do błędu, program może zostać przedwcześnie zakończony bez usuwania z pamięci biblioteki *icmp.dll* i biblioteki gniazd. Można by tego uniknąć, wczytując biblioteki nie w momencie kliknięcia przycisku *Badaj*, ale w ramach inicjalizacji programu; usuwać je zaś można na końcu programu. W międzyczasie, jeśli biblioteki nie byłyby załadowane, należałoby po prostu blokować możliwość naciskania przycisku *Badaj*.

W internecie można znaleźć pliki nagłówkowe dla biblioteki *icmp.dll*, dzięki którym można ją konsolidować z programem i uniknąć opisywanych kłopotów. Zrezygnowałem z tego celowo, aby nie wprowadzać dodatkowego zamieszania.



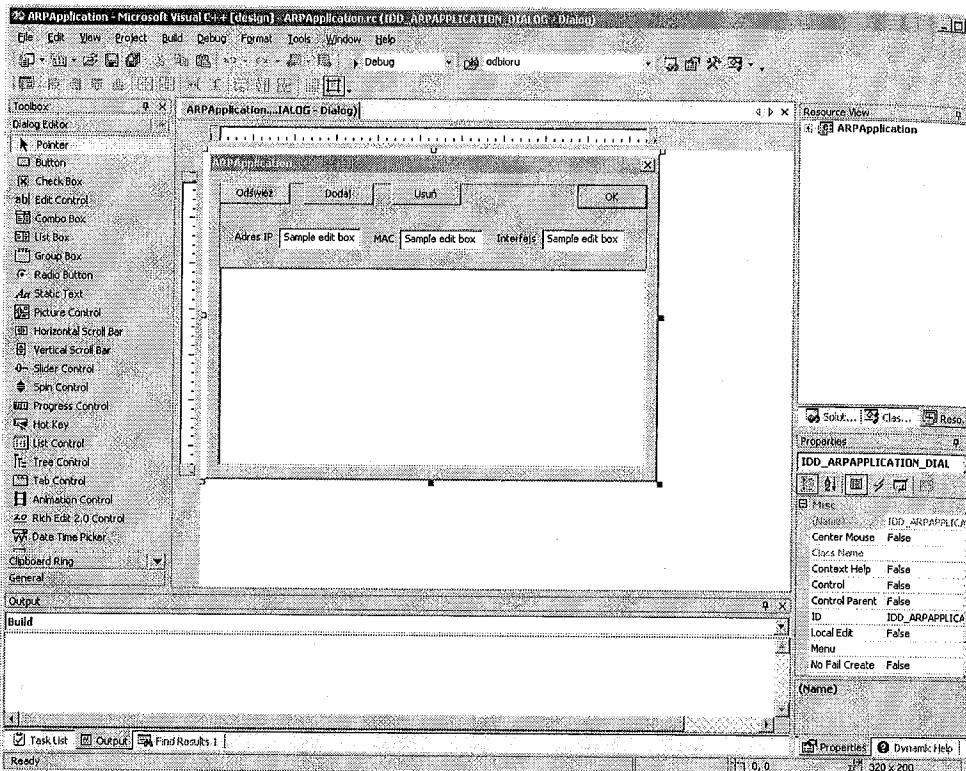
Kod źródłowy tego przykładu dostępny jest na dołączonej do książki płycie CD-ROM w podkatalogu \Przykłady\Rozdział6\TraceRoute.

6.7. Protokół ARP

Wspominałem już, że komputer jest w sieci lokalnej rozpoznawany na podstawie adresu MAC i adres ten jest mu niezbędny do przyłączenia się do tej sieci. W sieci stosowany jest też protokół ARP, który na podstawie adresu IP odnajduje adres MAC adresata. Dzieje się to automatycznie i użytkownik najczęściej nic o tym nie wie. Nie oznacza to, że nie może samodzielnie kontrolować tabeli wpisów ARP komputera lokalnego.

W systemie Windows użytkownik ma co prawda do dyspozycji program *arp*, ale jest to program działający w wierszu poleceń i jako taki jest mało wygodny w obsłudze. Możemy się jednak pokusić o napisanie jego graficznego odpowiednika, poznając przy okazji sposób obsługi protokołu ARP.

Utwórz nowy projekt typu *MFC Application*. Formularz jego okna głównego dostosuj do tego z rysunku 6.9. Okno powinno docelowo zawierać trzy pola tekstowe do wpisywania adresów IP i MAC oraz indeksu interfejsu, a także trzy przyciski. Pierwszy z nich (*Dodaj*) ma powodować dodanie wpisu ARP uzupełniającego tabelę ARP o adres MAC dla zadanego adresu IP. Drugi przycisk, *Odwieź*, powinien wyświetlać bieżącą zawartość tabeli ARP w zajmującym dolną część okna polu listy. Przycisk *Usuń* powinien zaś usuwać zaznaczony wpis z tabeli.



Rysunek 6.9. Projekt okna głównego nowego programu

Procedura obsługi zdarzenia naciśnięcia przycisku *Odwieź* pokazana jest na listingu 6.10.

Listing 6.10. Odświeżenie tabeli wpisów ARP

```
void CARPApplicationDlg::OnBnClickedButton1()
{
    DWORD dwStatus;
    PMIB_IPNETTABLE pIpArpTab=NULL;

    DWORD dwActualSize = 0;
    GetIpNetTable(pIpArpTab, &dwActualSize, true);

    pIpArpTab = (PMIB_IPNETTABLE) malloc(dwActualSize);
    if (GetIpNetTable(pIpArpTab, &dwActualSize, true) != NO_ERROR)
    {

```

```
        if (pIpArpTab)
            free (pIpArpTab);
        return;
    }

    DWORD i, dwCurrIndex;
    char sPhysAddr[256], sType[256], sAddr[256];
    PMIB_IPADDRTABLE pIpAddrTable = NULL;
    char Str[255];

    dwActualSize = 0;
    GetIpAddrTable(pIpAddrTable, &dwActualSize, true);
    pIpAddrTable = (PMIB_IPADDRTABLE) malloc(dwActualSize);
    GetIpAddrTable(pIpAddrTable, &dwActualSize, true);

    dwCurrIndex = -100;

    for (i = 0; i < pIpArpTab->dwNumEntries; ++i)
    {
        if (pIpArpTab->table[i].dwIndex != dwCurrIndex)
        {
            dwCurrIndex = pIpArpTab->table[i].dwIndex;

            struct in_addr in_ad;
            sAddr[0] = '\n';
            for (int i = 0; i < pIpAddrTable->dwNumEntries; i++)
            {
                if (dwCurrIndex != pIpAddrTable->table[i].dwIndex)
                    continue;

                in_ad.s_addr = pIpAddrTable->table[i].dwAddr;
                strcpy(sAddr, inet_ntoa(in_ad));
            }

            sprintf(Str, "Interfejs: %s o indeksie 0x%X", sAddr, dwCurrIndex);
            lbMessages.AddString(Str);
            lbMessages.AddString(" Adres internetowy | Adres fizyczny | Typ");
        }
    }

    AddrToStr(pIpArpTab->table[i].bPhysAddr,
              pIpArpTab->table[i].dwPhysAddrLen, sPhysAddr);

    switch (pIpArpTab->table[i].dwType)
    {
        case 1:
            strcpy(sType, "Inny");
            break;
        case 2:
            strcpy(sType, "Unieważniony");
            break;
        case 3:
            strcpy(sType, "Dynamiczny");
            break;
        case 4:
            strcpy(sType, "Statyczny");
            break;
    }
}
```

```

default:
    strcpy(sType, "");
}

struct in_addr in_ad;
in_ad.s_addr = pIpArpTab->table[i].dwAddr;
sprintf(Str, "%-22s | %-17s | %-12s", inet_ntoa(in_ad), sPhysAddr, sType);
lbtMessages.AddString(Str);
}

free(pIpArpTab);
}

```

Przyjrzyj się tej funkcji uważnie. Spostrzeżesz, że nie wczytujemy tu biblioteki gniazd. Plik *winsock.h* jest włączany do kodu źródłowego programu wyłącznie ze względu na zawarte w nim deklaracje potrzebnych nam typów danych; są one potrzebne jednak tylko w czasie komplikacji. Sam plik kodu biblioteki nie jest zaś potrzebny w czasie wykonania programu.

Gdyby funkcję obsługi zdarzenia naciśnięcia przycisku uzupełnić o choćby jedno wywołanie funkcji biblioteki gniazd (na przykład określanie adresów IP na podstawie nazw symbolicznych), trzeba by oczywiście wczytać bibliotekę do pamięci.

W ramach obsługi przycisku na samym początku pozyskiwana jest tabela odwzorowań adresów IP do adresów fizycznych, czyli tabela ARP. Zwraca ją funkcja:

```

DWORD GetIpNetTable(
    PMIB_IPNETTABLE pIpNetTable,
    PULONG pdwSize,
    BOOL bOrder
);

```

Tabela ARP opisywana jest następującymi parametrami:

- ◆ Wskaźnikiem struktury MIB_IPNETTABLE zawierającej właściwą tabelę.
- ◆ Rozmiarem struktury. Dla wartości zerowej tego parametru funkcja zwraca rozmiar pamięci potrzebnej do przechowywania tabeli.
- ◆ Znacznikiem porządkowania wpisów. Wartość TRUE oznacza, że wynikowa tabela będzie uporządkowana.

Struktura MIB_IPNETTABLE prezentuje się następująco:

```

typedef struct _MIB_IPNETTABLE {
    DWORD dwNumEntries;
    MIB_IPNETROW table[ANY_SIZE];
} MIB_IPNETTABLE, * PMIB_IPNETTABLE;

```

Pierwsza składowa struktury określa liczbę wpisów właściwej tabeli przechowywanej w drugiej składowej. Wiersze tej tabeli mają następującą strukturę:

```

typedef struct _MIB_IPNETROW {
    DWORD dwIndex;
    DWORD dwPhysAddrLen;
}

```

```

BYTE bPhysAddr[MAXLEN_PHYSADDR];
DWORD dwAddr;
DWORD dwType;
) MIB_IPNETROW, * PMIB_IPNETROW;

```

Sam pojedynczy wiersz tabeli wpisów ARP składa się z pięciu pól:

- ◆ Indeksu adaptera (dwIndex).
- ◆ Rozmiaru adresu fizycznego (dwPhysAddrLen).
- ◆ Samego adresu fizycznego (bPhysAddr).
- ◆ Adresu IP (dwAddr).
- ◆ Typu wpisu (dwType). Pole może przyjąć następujące wartości:
 - ◆ 4 — wpis statyczny (dodany ręcznie za pośrednictwem funkcji, które omówimy później),
 - ◆ 3 — wpis dynamiczny (wpis zawierający adres uzyskany automatycznie za pośrednictwem protokołu ARP, prawidłowy wyłącznie w pewnym ograniczonym czasie, po którym wpis jest automatycznie usuwany z tabeli),
 - ◆ 2 — wpis unieważniony, niezawierający poprawnych danych,
 - ◆ 1 — inny.

Jeśli w komputerze zainstalowana jest jedna tylko karta sieciowa, wszystkie wpisy w tabeli ARP będą odnosić się do jednego interfejsu. Jeśli jednak w systemie działa więcej niż jeden interfejs, niektóre z wpisów będą odnosiły się do jednego, reszta zaś do pozostałych interfejsów.

Aby więc użytkownik miał pełną информацию o rozdysponowaniu wpisów ARP, należałoby wyświetlać indeks interfejsu, którego wpis dotyczy. W strukturze MIB_IPNETROW znajduje się składowa przechowująca ten indeks, tyle że jej wartość nie ma dla użytkownika jasnego znaczenia. Indeks nabralby znaczenia, gdyby skojarzyć go z adresem IP karty interfejsu.

Tyle że jeszcze nie znamy adresu IP interfejsu. Aby go określić, musimy skojarzyć jeden z obowiązujących w systemie adresów IP z wybranym z zainstalowanych interfejsów. Można to zrobić za pośrednictwem funkcji GetIpAddrTable. Nie różni się ona wiele od GetIpNetTable:

```

DWORD GetIpAddrTable(
    PMIB_IPADDRTABLE pIpAddrTable,
    PULONG pdwSize,
    BOOL bOrder
);

```

Również przyjmuje trzy parametry: wskaźnik struktury typu MIB_IPADDRTABLE (parametr pIpAddrTable), rozmiar struktury (pdwSize) i znacznik sposobu porządkowania (bOrder).

Pierwszy parametr odnosi się do następującej struktury:

```

typedef struct _MIB_IPADDRTABLE {
    DWORD dwNumEntries;
}

```

```
MIB_IPADDRROW table[ANY_SIZE];
} MIB_IPADDRTABLE, * PMIB_IPADDRTABLE;
```

Struktura ta składa się z dwóch elementów:

- ◆ dwNumEntries — liczba elementów tablicy drugiej składowej.
- ◆ table — tablica struktur typu MIB_IPADDRROW.

Struktura MIB_IPADDRROW to:

```
typedef struct _MIB_IPNETROW {
    DWORD dwAddr;
    DWORDIF_INDEX dwIndex;
    DWORD dwMask;
    DWORD dwBCastAddr;
    DWORD dwReasmSize;
    unsigned short unused1;
    unsigned short wType;
} MIB_IPADDRROW, * PMIB_IPADDRROW;
```

Znaczenie składowych tej struktury jest następujące:

- ◆ dwAddr — adres IP.
- ◆ dwIndex — indeks adaptera skojarzonego z adresem IP.
- ◆ dwMask — maska podsieci dla adresu IP.
- ◆ dwBCastAddr — adres rozgłoszeniowy adaptera. Najczęściej jest to adres zgodny z adresem IP komputera — w adresie rozgłoszeniowym zeruje się jedynie numer węzła podsieci. Jeśli np. adres IP ma postać 192.168.4.7, adres rozgłoszeniowy może mieć postać 192.168.4.0.
- ◆ dwReamsSize — maksymalny rozmiar odbieranych pakietów.
- ◆ unused1 — zarezerwowane.
- ◆ wType — typ adresu. Może przyjąć jedną z pięciu wartości:
 - ◆ MIB_IPADDR_PRIMARY — podstawowy adres IP,
 - ◆ MIB_IPADDR_DYNAMIC — adres dynamiczny,
 - ◆ MIB_IPADDR_DISCONNECTED — adres interfejsu przypisywany dla okresu pozostawania systemu poza siecią (kiedy np. odłączy się kabel sieciowy od komputera),
 - ◆ MIB_IPADDR_DELETED — adres usunięto,
 - ◆ MIB_IPADDR_TRANSIENT — adres przejściowy.

Po pozyskaniu wszystkich potrzebnych informacji rozpoczyna się pętla przeglądająca kolejne wpisy tabeli ARP. Przed ich wyświetleniem musi jednak określić przynależność wpisów do interfejsów.

Przy pozyskiwaniu danych wymusiliśmy porządkowanie wpisów, możemy więc mieć nadzieję, że pierwszymi wpisami tabeli ARP będą wpisy pierwszego interfejsu. Dlatego przed rozpoczęciem pętli przypisujemy do dwCurrIndex wartość -100. Nie ma szans,

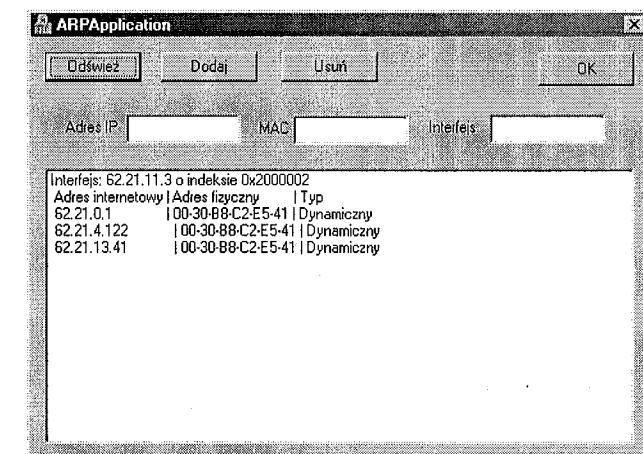
żeby w systemie działał interfejs o takim indeksie i już w pierwszym przebiegu pętli okaże się z pewnością, że wpis tabeli ARP nie odnosi się do interfejsu o numerze -100. Należy więc wyświetlić adres IP karty, do której odnosi się wpis. W tym celu trzeba przeszukać tabelę odwzorowań adresów IP do interfejsów, szukając w niej wpisu o indeksie równym dwIndex. Jeśli uda się taki wpis znaleźć, można wyświetlić nagłówek tabeli. Będzie wyglądał mniej więcej tak:

| | | |
|--|----------------|-----|
| Interfejs: 192.168.1.100 o indeksie 0x10000003 | | |
| Adres internetowy | Adres fizyczny | Typ |

Pod takim nagłówkiem można już wyświetlić zawartość tabeli ARP, zatrzymując się dopiero przy wpisie odnoszącym się do interfejsu o innym indeksie. Trzeba wtedy wygenerować kolejny nagłówek i dopiero potem kontynuować wypisywanie zawartości tabeli ARP.

Efekt działania programu widać na rysunku 6.10. Zauważ, że nie na każdym komputerze tablica ARP musi zawierać jakiekolwiek wpisy — protokół ARP stosowany jest jedynie w sieciach lokalnych. Jeśli łączysz się z internetem przez modem, tabela będzie pusta.

Rysunek 6.10.
Efekt działania programu ARPApplication



Jeśli nie wiesz, jak można praktycznie wykorzystać taką aplikację, postaram się podsunąć kilka zastosowań. Pamiętam ze swojej pracy momenty, w których potrzebowałem adresu MAC jednego z komputerów sieci lokalnej, ale komputer ten był zlokalizowany w innym pomieszczeniu. Można by oczywiście poświęcić chwilę na krótki spacer do tegoż pomieszczenia i uruchomienie polecenia *ipconfig* na danym komputerze, można jednak również dobrze wdrożyć następującą procedurę:

1. Uruchomić program *Pinger* albo *ping* celem sprawdzenia połączenia z komputerem. Wymusi to wysłanie pakietów echo do adresata, co z kolei wymagać będzie określenia jego adresu MAC, co spowoduje zaangażowanie do zadania protokołu ARP.
2. Uruchomić program wyświetlający tabelę ARP i odszukać w niej adres MAC komputera zdalnego.

Spójrzmy teraz, jak można dodawać wpisy do tabeli ARP. Jak pamiętamy, wszystkie wpisy dodawane ręcznie (programowo) będą wpisami statycznymi i nie będą automatycznie usuwane przez system. Po kliknięciu przycisku *Dodaj* program przechodzi do wykonania kodu z [listingu 6.11](#).

Listing 6.11. Dodawanie wpisów do tabeli ARP

```
void CARPApplicationDlg::OnBnClickedButton2()
{
    char sPhysAddr[255],
        sInetAddr[255],
        sMacAddr[255],
        sInterface[255];

    edIPAddress.GetWindowText(sInetAddr, 255);
    edMacAddress.GetWindowText(sMacAddr, 255);
    edInterface.GetWindowText(sInterface, 255);

    if (sInetAddr == NULL || sMacAddr == NULL || sInterface == NULL)
    {
        AfxMessageBox("Wpisz adres IP, adres MAC i numer interfejsu");
        return;
    }

    DWORD dwInetAddr;
    dwInetAddr = inet_addr(sInetAddr);
    if (dwInetAddr == INADDR_NONE)
    {
        AfxMessageBox("Niepoprawny adres IP");
        return;
    }

    StrToMACAddr(sMacAddr, sPhysAddr);

    MIB_IPNETROW arpRow;
    sscanf(sInterface, "%X", &(arpRow.dwIndex));

    arpRow.dwPhysAddrLen = 6;
    memcpy(arpRow.bPhysAddr, sPhysAddr, 6);
    arpRow.dwAddr = dwInetAddr;
    arpRow.dwType = MIB_IPNET_TYPE_STATIC;

    if (SetIpNetEntry(&arpRow) != NO_ERROR)
        AfxMessageBox("Nie można dodać wpisu ARP");
}
```

Najważniejszy komponent tego kodu to funkcja `SepIpNetEntry` dodająca nowy wpis ARP do tabeli:

```
DWORD SetIpNetEntry(
    PMIB_IPNETROW pArpEntry
);
```

Jednym parametrem jej wywołania jest struktura `MIB_IPNETROW`, z której korzystaliśmy już przy pozyskiwaniu danych z tabeli ARP. W ramach tej struktury należy na potrzeby funkcji `SetIpNetEntry` wypełnić komplet pól: `dwIndex` (numer interfejsu), rozmiar adresu MAC (`bPhysAddrLen`) adres MAC (`bPhysAddr`), adres IP (`dwInetAddr`) i typ wpisu (`dwType`).

Przyjrzyjmy się teraz funkcji usuwającej adresy statyczne. Jest ona wywoływana w reakcji na naciśnięcie przycisku *Usuń*, a jej kod widnieje na [listingu 6.12](#).

Listing 6.12. Usuwanie wpisów z tabeli ARP

```
void CARPApplicationDlg::OnBnClickedButton3()
{
    char sInetAddr[255],
        sInterface[255];

    edIPAddress.GetWindowText(sInetAddr, 255);
    edInterface.GetWindowText(sInterface, 255);

    if (sInetAddr == NULL || sInterface == NULL)
    {
        AfxMessageBox("Wprowadź adres IP i numer interfejsu");
        return;
    }

    DWORD dwInetAddr;
    dwInetAddr = inet_addr(sInetAddr);
    if (dwInetAddr == INADDR_NONE)
    {
        printf("IpArp: niepoprawny argument %s\n", sInetAddr);
        return;
    }

    MIB_IPNETROW arpEntry;

    sscanf(sInterface, "%X", &(arpEntry.dwIndex));
    arpEntry.dwAddr = dwInetAddr;

    if (DeleteIpNetEntry(&arpEntry) != NO_ERROR)
        AfxMessageBox("Nie można usunąć wpisu ARP");
}
```

Do usuwania wpisu ARP służy funkcja `DeleteIpNetEntry`:

```
DWORD DeleteIpNetEntry(
    PMIB_IPNETROW pArpEntry
);
```

Przyjmuje ona tylko jeden parametr — wskaźnik struktury `MIB_IPNETROW` opisującej wpis do usunięcia. W strukturze wystarczy określić indeks interfejsu i adres IP wpisu, który ma zostać usunięty.

 Kod źródłowy tego przykładu dostępny jest na dołączonej do książki płycie CD-ROM w podkatalogu `\Przykłady\Rozdział6\ARPAplication`.

Podsumowanie

Starałem się zamieścić w tej książce jak najwięcej przykładów, nie sposób jednak uznać, że wyczerpują one tematykę metod stosowanych przez hakerów. Programowanie jest sztuką wymagającą ciągłego doskonalenia i nauki — przygoda taka może potrwać całe życie. Przy tym rozwój technologii komputerowych i informatyki jest tak szybki, że nie sposób za nim w pojedynkę nadążyć.

Swego czasu byłem cały rok zajęty programowaniem w pewnej firmie. Musiałem na ten okres zrezygnować z intensywnego samodokształcania się. W efekcie po tym roku zostałem w tyle. Wciąż pisałem programy dla systemu MS-DOS, podczas gdy inni programiści studiowali już Windows. Zrozumiałem wtedy, że najwyższy czas zmienić pracę, bo wkrótce może się okazać, że moja wiedza jest nieaktualna, a umiejętności niepotrzebne. Odszedłem więc z firmy i przez trzy miesiące intensywnie studiowałem interfejs programistyczny systemu Windows, tak aby być przygotowanym do tworzenia w tym systemie projektów komercyjnych i móc znaleźć pracę, która pozwoli mi na dotrzymanie kroku technologiom.

Moi Czytelnicy często pytają mnie, co jeszcze powinni przeczytać. Odpowiedź brzmi: wszystko! Osobiście czytam, co popadnie: wszelkie dokumentacje, pliki pomocy, artykuły w magazynach komputerowych itd. Książki specjalistyczne są niewystarczające, nigdy też nie wiadomo, w której książce znajduje się to, czego akurat szukamy na potrzeby bieżącego projektu.

Nawet jednak poświęcanie czasu na samokształcenie nie gwarantuje nikomu kariery hakera. W okresie studiów spędzałem przy komputerze od 10 do 16 godzin dziennie i starczało mi czasu na realizację wszystkich (prawie) obowiązkowych projektów laboratoryjnych i własnych pomysłów. Teraz jestem żonaty i mam dwójkę dzieci i choć staram się spędzać przy komputerze przynajmniej 9 godzin dziennie, nie mam już czasu na wszystko. Zawsze chcę zrobić więcej, poznać nowe rzeczy, żeby rozwój informatyki nie zostawił mnie zbytnio w tyle. Jest to jednak niemożliwe również dlatego, że informatyka wkracza dziś w zbyt wiele całkowicie odmiennych zastosowań.

Tak czy inaczej mam nadzieję, że niniejsza książka pomoże odpowiedzieć Czytelnikowi na choćby niektóre z pytań, które chodzą mu po głowie.

Skorowidz

_tWinMain(), 37, 38, 65

A

About(), 36
accept(), 183, 192, 244
ActiveX, 25
AddIPAddress(), 230
ADO, 16
adres
 IP, 143, 229, 257
 MAC, 143, 223
algorytm odbioru-wysyłania danych, 242
animowanie tekstu, 133
API Windows, 24
APIENTRY, 38
aplikacje
 MFC, 26
 okienkowe, 30
 Win32 Application, 30
architektura klient-serwer, 158
ARP, 143, 273
 dodawanie wpisów, 280
 odświeżanie tabeli, 274
 usuwanie wpisów, 281
ARPAnet, 11
ASCII, 71, 187
asembler, 49
ASPack, 19, 21
 projekty komercyjne, 22
asynchroniczna wymiana danych, 220
autoryzacja połączenia, 159

B

BAK, 22
BeginDeferWindowPos(), 106
BeginPaint(), 67

biblioteki

deszyfrowania haseł, 122
DLL, 122, 124
gniazd, 174
icmp.dll, 270
IPHlpApi.lib, 224
tworzenie, 124
wczytywanie, 126
WinSock, 174
WinSock2, 174
bind(), 181, 205
bitmapa, 63
bomba sieciowa, 94
Borland Delphi, 13
break, 36, 82

C

C++, 9
CALLBACK, 99
CallNextHookEx(), 125
case, 36, 81, 218
CAsyncSocket, 158
CCClientSocket, 158, 168
ChangeIPAddress(), 231
chwytyanie okna, 119
ClipboardChange, 135
CloseClipboard(), 136
closesocket(), 192
COM, 234
Combo Box, 234
connect(), 192
CPP, 30
CrazyStart, 71
CreateCompatibleDC(), 67
CreateEllipticRgn(), 108, 110
CreateEvent(), 69, 102

CreateFile(), 150, 236
 CreateRectRgn(), 110
 CreateThread(), 196, 197
 CreateWindowEx(), 41
 CServerSocket, 158, 166, 171
 CSocket, 158, 165
 CW_USEDEFAULT, 41
 czas życia pakietu, 142

D

DAO, 16
 DCB, 236
 DeferWindowPos(), 106
 DefWindowProc(), 42
 DeleteDC(), 68
 DeleteIPAddress(), 230
 Delphi, 16
 destabilizacja kodu, 50
 deszyfrowanie haseł, 122, 126
 DHCP, 257
 odnawianie adresu IP, 259
 zwalnianie adresu IP, 258
 diagramy, 142
 DispatchMessage(), 42
 DLL, 19
 DllExport, 123
 DllMain(), 123
 DNS, 224
 DrawStartButton(), 70

E

ECHOREPLY, 266
 ECHOREQUEST, 265
 edytor
 menu, 79
 zasobów, 64
 efekt podrzucania okna w góre, 70
 elementy sterujące, 55
 EnableWindow(), 82
 EndDeferWindowPos(), 106
 EndPaint(), 68
 EnumChildWindows(), 100
 EnumChildWnd(), 100
 EnumNetwork(), 154
 EnumWindows(), 98
 EnumWindowsWnd(), 98, 100
 FD_ISSET(), 211, 213
 FindWindow(), 72, 92, 97
 FindWindowEx(), 75
 GetAdaptersInfo(), 228
 GetCommState(), 236
 GetCursor(), 91

F

FastScan, 246
 FD_CLOSE, 219
 FD_ISSET(), 211, 213
 FD_READ, 219, 220
 FD_WRITE, 219
 FIDO, 11
 FindWindow(), 72, 92, 97
 FindWindowEx(), 75
 Flash 5, 52
 FTP, 147
 funkcje, 38
 _tWinMain(), 65
 accept(), 183, 192, 244
 AddIPAddress(), 230
 BeginDeferWindowPos(), 106
 BeginPaint(), 67
 bind(), 181, 205
 CallNextHookEx(), 125
 ChangeIPAddress(), 231
 CloseClipboard(), 136
 closesocket(), 192
 connect(), 192
 CreateCompatibleDC(), 67
 CreateEllipticRgn(), 108, 110
 CreateEvent(), 69, 102
 CreateFile(), 150, 236
 CreateRectRgn(), 110
 CreateThread(), 196, 197
 CreateWindowEx(), 41
 DeferWindowPos(), 106
 DefWindowProc(), 42
 deklaracja, 38
 DeleteDC(), 68
 DeleteIPAddress(), 230
 DispatchMessage(), 42
 DllMain(), 123
 DrawStartButton(), 70
 EnableWindow(), 82
 EndDeferWindowPos(), 106
 EndPaint(), 68
 EnumChildWindows(), 100
 EnumChildWnd(), 100
 EnumNetwork(), 154
 EnumWindows(), 98
 EnumWindowsWnd(), 98, 100
 FD_ISSET(), 211, 213
 FindWindow(), 72, 92, 97
 FindWindowEx(), 75
 GetAdaptersInfo(), 228
 GetCommState(), 236
 GetCursor(), 91

GetDesktopWindow(), 93
 GetDlgItemText(), 164
 GetForegroundWindow(), 87
 gethostbyname(), 184
 GetIpAddrTable(), 277
 GetIpNetTable(), 276, 277
 GetMessage(), 42
 GetNetworkParams(), 228
 GetProcAddress(), 271
 getservbyport(), 249, 250
 GetSystemMetrics(), 69, 89
 GetTcpTable(), 252, 254, 255
 GetUdpTable(), 256
 GetWindow(), 75
 GetWindowPos(), 106
 GetWindowRect(), 103, 113
 GlobalAlloc(), 228
 IcmpCloseHandle(), 270, 272
 IcmpCreateFile(), 270, 271
 IcmpSendEcho(), 270
 InitInstance(), 41, 65
 ioctlsocket(), 214
 IpReleaseAddress(), 257
 IsWindowVisible(), 106
 listen(), 182, 192, 197, 217
 LoadImage(), 65
 LoadLibrary(), 127, 270
 LoadString(), 39
 malloc(), 255
 mciSendCommand(), 85
 MoveWindow(), 103
 NetThread(), 196, 197
 obsługi komunikatów, 42
 OnBnClickedButton(), 236
 OnInitDialog(), 152
 OpenClipboard(), 136
 rand(), 89
 ReadThread(), 237
 recv(), 188, 192
 recvfrom(), 192, 266
 RegisterClassEx(), 41
 RunStopHook(), 124
 select(), 210, 211, 244
 SelectObject(), 67
 send(), 203
 SendMessage(), 73, 83, 90, 99, 125, 129, 132
 sendto(), 193
 SetClipboardData(), 136
 SetCursorPos(), 89
 SetMenu(), 82
 SetParent(), 77, 82
 SetSystemCursor(), 91
 SetWindowLong(), 66
 SetWindowPos(), 69, 78, 104, 106
 SetWindowRect(), 113
 SetWindowRgn(), 108, 109
 SetWindowsHookEx(), 124, 125, 130
 ShellExecute(), 83
 ShowWindow(), 42, 86
 shutdown(), 191
 Sleep(), 73
 socket(), 205
 SysMsgProc(), 125
 SystemParametersInfo(), 87
 TranslateMessage(), 42
 TransmitFile(), 190
 UpdateWindow(), 41, 69, 109
 WaitForMultipleObject(), 221
 WaitForSingleObject(), 69, 102, 221
 WindowFromPoint(), 89
 WinExec(), 94
 WndProc(), 42, 66, 79, 92, 98, 214, 215, 217
 WNetCloseEnum(), 157
 WNetEnumResource(), 157
 WNetOpenEnum(), 157
 WSAAccept(), 183
 WSAAsyncGetHostByName(), 184
 WSAAsyncSelect(), 214, 220, 221
 WSACloseEvent(), 222
 WSACreateEvent(), 185, 186
 WSACreateEvent(), 220, 221
 WSAEventSelect(), 220, 221
 WSAGetLastError(), 188
 WSARecv(), 188, 189
 WSARecvFrom(), 192
 WSAResetEvent(), 222
 WSARecv(), 187, 193
 WSARecvFrom(), 193
 WSASendTo(), 193
 WSASocket(), 179
 WSAStartup(), 175, 264
 WSAWaitForMultipleEvents(), 220, 221
 wywołanie, 39

G

GetAdaptersInfo(), 228
 GetCommState(), 236
 GetCursor(), 91
 GetDesktopWindow(), 93
 GetDlgItemText(), 164
 GetForegroundWindow(), 87
 gethostbyname(), 184
 GetIpAddrTable(), 277
 GetIpNetTable(), 276, 277
 GetMessage(), 42
 GetNetworkParams(), 228
 GetProcAddress(), 271
 getservbyport(), 249, 250

GetSystemMetrics(), 69, 89
 GetTcpTable(), 252, 254, 255
 GetUdpTable(), 256
 GetWindow(), 75
 GetWindowPos(), 106
 GetWindowRect(), 103, 113
 GlobalAlloc(), 228
 gniazda, 147, 165, 171, 209
 asynchroniczne, 209
 Berkeley, 174
 blokujące, 209
 BSD Unix, 147
 CSocket, 158
 ICMP, 271
 komunikaty systemowe, 213
 nieblokujące, 209
 RAW, 270
 synchroiczne, 209
 tryby wymiany danych, 209
 tworzenie, 179
 typy, 180
 Windows, 147
 WinSock, 174

H

haker, 11, 12
 hasła, 121
 deszyfrowanie, 126
 HBITMAP, 64
 HTTP, 147
 HWND, 64, 72

I

IANA, 181
 ICMP, 260, 264
 icmp.dll, 270
 ICMP_ECHO_REPLY, 271
 IcmpCloseHandle(), 270, 272
 IcmpCreateFile(), 270, 271
 IcmpSendEcho(), 270
 IDD_MFCESENDDTEXT_DIALOG, 165
 ikonki, 54, 132
 animowanie tekstu, 133
 InitInstance(), 37, 41, 65
 interfejs okna głównego, 54
 ioctlsocket(), 214
 IP, 142, 223
 IP_OPTION_INFORMATION, 271
 IPHIpApi.lib, 224
 IpReleaseAddress(), 257
 IPX/SPX, 146, 147
 IsWindowVisible(), 106

J

język
 assembler, 49
 Borland Delphi, 13
 C++, 9
 obiektowy, 24
 XML, 16

K

karty sieciowe, 223
 ustawienia, 225
 klasa okna, 40
 klient, 148, 158, 184
 TCP, 199
 UDP, 205
 klient-serwer, 158, 165, 180
 kod źródłowy programu, 33
 komentarze, 38
 komplikacja projektu, 42
 komponenty projektu, 30
 kompresja plików wykonywalnych, 19
 komputer, 48
 komunikaty, 41, 220
 konsolidacja, 27, 85
 dynamiczna, 28
 kontekst
 obrazka, 67
 okna, 67
 kontrola danych, 208
 kraker, 12

L

latające obiekty, 89
 latający przycisk Start, 62
 liczby losowe, 103
 listen(), 182, 192, 197, 217
 LoadImage(), 65
 LoadLibrary(), 127, 270
 LoadString(), 39
 lpTransmitBuffers, 191
 LR_DEFAULTCOLOR, 65
 LV_ALIGNLEFT, 132
 LV_ALIGNTOP, 132
 LVM_ARRANGE, 132
 LVM_DELETEALLITEMS, 133
 LVM_SETITEMPOSITION, 133
 LVM_SETITEMTEXT, 133
 LVM_UPDATE, 134

M

MAC, 143, 223
 malloc(), 255
 MAX_LOADSTRING, 40
 MCI_OPEN, 85
 mciSendCommand(), 85
 menu, 54, 79
 tworzenie, 80
 MFC, 24
 obsługa sieci, 158
 MIB_IPADDRROW, 278
 MIB_IPADDRTABLE, 278
 MIB_IPNETROW, 277
 MIB_IPNETTABLE, 276
 model OSI, 140
 monitorowanie plików wykonywalnych, 130
 MoveWindow(), 103
 mysz, 88
 ograniczanie zakresu ruchu, 90
 wskaźnik, 91

N

nawiązywanie połączenia, 164
 NET SEND, 94
 NetBEUI, 146
 NetBIOS, 145
 NETRESOURCE, 156
 NetThread(), 196, 197
 niestandardowe okna, 107
 niewidzialne programy, 36
 Novell, 146, 147

O

obiekty zdarzeń, 220
 obrazki, 63
 obsługa
 błędów, 175
 komunikatów, 42
 odbieranie danych, 209, 242
 odnawianie adresu IP, 259
 odrysowanie okna, 66
 odświeżanie pulpitu, 134
 okna, 40
 belka tytułowa, 117
 chwytanie, 119
 dialogowe, 55
 dowolne kształty, 115
 elementy sterujące, 55
 główne, 51, 54
 kształty, 113
 manipulacja, 97

P

pakiety, 141
 ICMP, 262
 IP, 142
 NetBIOS, 146
 śledzenie tras, 267
 pamięć, 39
 pasek
 narzędziowy, 54
 zadań, 71, 75, 76
 pętla
 komunikatów, 101
 nieukończona, 102

PFIXED_INFO, 228
 piksel, 114
 Pinger, 262
 pliki, 150
 atrybuty, 151
 CPL, 83
 kompresja, 19
 monitorowanie, 130
 nagłówkowe, 123
 wykonywalne, 19, 130
 zawieszające system, 239
 podsistem sieciowy, 223
 połączenia, 164
 MFC, 174
 TCP, 194
 WinSock, 186
 zamykanie, 191
 POP3, 147
 port szeregowy, 234
 komunikacja, 237
 otwieranie, 235
 połaczenie, 236
 wysyłanie danych, 238
 porty, 148, 181
 stan, 252
 poruszanie oknem, 68
 ProgMan, 132
 program, 41
 optymalizacja, 43
 programowanie
 sieciowe, 139
 Windows, 97
 programy-żarty, 19, 61
 bomba sieciowa, 94
 latające obiekty, 89
 latający przycisk Start, 62
 mysz, 88
 pasek zadań, 76
 przycisk Start, 71, 73
 pulpit, 93
 systemowe pliki CPL, 83
 tapeta pulpitu, 87
 ukrywanie okien, 86
 usuwanie zegara z paska zadań, 86
 wskaźnik myszy, 91
 wygaszanie monitora, 83
 wysuwanie tacki napędu CD-ROM, 84
 projekt, 26, 27
 komplikacja, 42
 konsolidacja, 85
 zasoby, 31
 projektowanie okien, 51
 protokoły, 139, 140, 141
 ARP, 143, 273
 bezpołączeniowe, 192

ICMP, 260
 IP, 142
 IPX/SPX, 146, 147
 NetBEUI, 146
 NetBIOS, 145
 RARP, 143
 TCP, 143, 144, 194
 TCP/IP, 141
 transportowe, 143
 UDP, 143, 204
 warstwy aplikacji, 145
 zorientowane na komunikaty, 190
 przeglądarka schowka, 135
 przełączanie ekranów, 103
 przesunięcie bitowe, 48
 w lewo, 48
 w prawo, 48
 przetwarzanie komunikatów, 41
 przycisk
 maksymalizacji, 66
 minimalizacji, 66
 Start, 62, 71, 73
 pulpit, 93
 ikony, 132
 odświeżanie, 134

Q

QoS, 214

R

rand(), 89
 RARP, 143
 ReadThread(), 237
 recv(), 188, 192
 recvfrom(), 192, 266
 RegisterClassEx(), 41
 rozwijana lista, 234
 RS-232, 234
 Rundll32.exe, 84
 RunStopHook(), 124
 rysowanie, 67

S

schowek, 134
 select(), 210, 211, 244
 SelectObject(), 67
 send(), 203
 SendMessage(), 73, 83, 90, 99, 125, 129, 132
 sendto(), 193

serwer, 148, 180
 autoryzacja połączenia, 159
 DHCP, 257
 TCP, 194
 UDP, 204
 współbieżąca obsługa klientów, 242
 SetClipboardData(), 136
 SetCursorPos(), 89
 SetMenu(), 82
 SetParent(), 77, 82
 SetSystemCursor(), 91
 SetWindowLong(), 66
 SetWindowPos(), 69, 78, 104, 106
 SetWindowRect(), 113
 SetWindowRgn(), 108, 109
 SetWindowsHookEx(), 124, 125, 130
 Shell_TrayWnd, 72
 ShellExecute(), 83
 ShowWindow(), 42, 86
 shutdown(), 191
 sieć, 139
 asynchroniczna wymiana danych, 220
 diagramy, 142
 FIDO, 11
 gniazda, 147
 MFC, 158
 model OSI, 140
 NetBEUI, 146
 Novell, 146, 147
 obsługa błędów, 175
 odbieranie danych, 209
 otoczenie sieciowe, 148
 pakiety, 141
 pętla oczekiwania na połączenie, 212
 połączenia, 164
 porty, 148, 181
 protokoły, 139, 141
 przetwarzanie odebranych danych, 207
 skaner portów, 163, 245
 TCP, 194
 TCP/IP, 141
 transfer danych, 141, 165
 UDP, 204
 warstwy OSI, 140
 wysyłanie danych, 209
 wysyłanie większych ilości danych, 202
 zmiana adresu IP, 229
 silnik gry, 44
 skaner portów, 163, 245
 skanowanie portów, 165
 Sleep(), 73
 SM_CXSCREEN, 89
 SM_CYSCREEN, 89
 SMTP, 147

Ś

śledzenie trasy pakietu, 267

T

tabele

ARP, 276
 gotowe wyniki obliczeń, 49
 tapeta pulpitu, 87
 TCHAR, 40
 TCP, 144, 145, 180, 194
 klient, 199
 połączenia, 194
 serwer, 194
 stan portów, 252
 TCP/IP, 141, 146
 testowanie, 50
 TraceRoute, 272
 TranslateMessage(), 42
 transmisja danych, 141, 165, 173
 CSocket, 165
 klient, 172
 połączenie, 173
 serwer, 171
 TransmitFile(), 190
 trasa pakietu, 267
 Tree Control, 152
 TTL, 142, 267
 TVN_ITEMEXPANDING, 153

tworzenie
bibliotek, 122
gniazda, 179
programów, 36
programów-żartów, 19, 61

U

udostępnianie zasobów sieciowych, 149
UDP, 143, 204
klient, 205
serwer, 204
stan portów, 256
transmisja, 204
ujawnianie haseł, 121
ukrywanie okien, 86
UNC, 149
UNICODE, 187
UpdateWindow(), 41, 69, 109
usuwanie
adresu IP, 233
wąskich gardeł, 44
zegara z paska zadań, 86

V

VCL, 24
Visual Basic, 13
Visual C++, 25
biblioteki DLL, 122
Build, 42
edytor menu, 79
kod źródłowy, 33
komponenty projektu, 30
kreator aplikacji, 33
kreator projektu, 26
kreator ustawień aplikacji, 29
MFC, 26, 28
MFC Application, 159
NetNeighbour, 151
projekt, 25
sieć, 139
Solution Explorer, 30
ustawienia konsolidacji, 27
Win32 Application, 30
zasoby, 31, 63

W

WaitForMultipleObject(), 221
WaitForSingleObject(), 69, 102, 221
warstwy OSI, 140
wąskie gardła, 44, 45

wątki, 196
wczytywanie bibliotek, 127
WH_GETMESSAGE, 130
while, 41, 98
WinAMP, 53
WinAPI, 24, 29, 41, 61
WindowFromPoint(), 89
Windows, 61, 97
chwytywanie okna, 119
ikonki, 132
komunikaty, 99
monitorowanie plików, 130
niestandardowe okna, 107
okna, 97
przełączanie ekranów, 103
schowek, 134
ujawnianie haseł, 121
wysyłanie komunikatów, 99
Windows XP, 37
WinExec(), 94
winipcfg, 223
WINS, 224
WinSock, 139, 174, 246
closesocket(), 192
connect(), 192
informacje, 178
klient, 184
lpTransmitBuffers, 191
obsługa błędów, 175
połączenia, 186
protokoły bezpośredniowe, 192
recvfrom(), 192
sendto(), 193
serwer, 180
shutdown(), 191
sockaddr, 181
TransmitFile(), 190
tworzenie gniazda, 179
wczytywanie biblioteki, 175
WSAAccept(), 183
WSAAccept(), 183
WSAAsyncGetHostByName(), 184
WSAAsyncSelect(), 214
WSACleanup(), 178
WSACloseEvent(), 222
WSAConnect(), 185, 186
WSACreateEvent(), 221
WSADATA, 176
WSAEADDRINUSE, 181
WSAEMSGSIZE, 190
WSAENOTSOCK, 192
WSAEEventSelect(), 220, 221
WSAEWOULDBLOCK, 212
WSAGetLastError(), 174, 188
WSARecv(), 188, 189
WSARecvEx(), 174
WSARecvFrom(), 192
WSAResetEvent(), 222
WSASend(), 187, 193
WSASendTo(), 193
WSASocket(), 179
WSASocket(), 174, 175, 264

WSASocket(), 179
WSAStartup(), 176
WSAWaitForMultipleEvents(), 221
wymiana danych, 186
zamykanie połączenia, 191
WinSock2, 174
WinSockInfo, 177
wirus, 11
WM_DESTROY, 42
WM_PAINT, 66, 118
WM_RBUTTONDOWN, 125
WM_SETTEXT, 99
WNDCLASSEX, 40
WndProc(), 36, 42, 66, 79, 92, 98, 214, 215, 217
WNetCloseEnum(), 157
WNetEnumResource(), 157
WNetOpenEnum(), 157
WS_OVERLAPPEDWINDOW, 41
WSAAccept(), 183
WSAAsyncGetHostByName(), 184
WSAAsyncSelect(), 214, 219, 220, 221
WSACleanup(), 174, 178
WSACloseEvent(), 222
WSAConnect(), 185, 186
WSACreateEvent(), 220, 221
WSADATA, 176
WSAEADDRINUSE, 181
WSAEMSGSIZE, 190
WSAENOTSOCK, 192
WSAEEventSelect(), 220, 221
WSAEWOULDBLOCK, 212
WSAGetLastError(), 174, 188
WSARecv(), 188, 189
WSARecvEx(), 174
WSARecvFrom(), 192
WSAResetEvent(), 222
WSASend(), 187, 193
WSASendTo(), 193
WSASocket(), 179
WSASocket(), 174, 175, 264

WSAWaitForMultipleEvents(), 220, 221
wskaźnik myszy, 91
wskaźniki, 39
współbieżna obsługa klientów, 242
wygaszanie monitora, 83
wysuwanie tarcii napędu CD-ROM, 84
wysyłanie
danych, 209
komunikatów, 99
wyszukiwanie okna, 75
wywołanie funkcji, 39

X

XML, 16

Z

zabezpieczenia nazw systemu plików, 240
zaczep, 130
zamykanie
okien, 92
połączzeń, 191
zarządzanie ikonami, 132
zasoby, 31
dodawanie, 63
sieciowe, 149
zdarzenia, 69
mysz, 119
TVN_ITEMEXPANDING, 153
tworzenie, 69
zmiana adresu IP, 229
zmienne, 39
deklaracja, 39
globalne, 39
lokalne, 39
zmnieszszanie rozmiaru programu, 24
zwalnianie adresu IP, 258

Zawartość płyty CD-ROM

Zawartość dołączonej do książki płyty CD-ROM przedstawia się następująco:

| Katalog | Zawartość |
|----------------------|---|
| \Przykłady | Kod źródłowy i pliki wykonywalne przykładowych projektów realizowanych w kolejnych podrozdziałach. Choć przykłady są stosunkowo nieliczne, warto rzucić na nie okiem. |
| \Przykłady\Rozdział1 | Kod źródłowy do przykładów z rozdziału 1. |
| \Przykłady\Rozdział2 | Kod źródłowy do przykładów z rozdziału 2. |
| \Przykłady\Rozdział3 | Kod źródłowy do przykładów z rozdziału 3. |
| \Przykłady\Rozdział4 | Kod źródłowy do przykładów z rozdziału 4. |
| \Przykłady\Rozdział5 | Kod źródłowy do przykładów z rozdziału 5. |
| \Przykłady\Rozdział6 | Kod źródłowy do przykładów z rozdziału 6. |
| \Demo | Próbne wersje oprogramowania autorstwa CyD Software Labs. |
| Inne | Prosty, ale przydatny program do kompresowania plików wykonywalnych. |

Zamów tę książkę w dowolnej chwili



Zamówienia książek przez SMS pod numer 0 691 HELION

Wyślij SMS-em pod numer **0 691 HELION (0 691 435 466)** umieszczony na okładce numer katalogowy książki. W ciągu 24 godzin wyślemy przesyłkę z książkami na wskazany adres. Wraz z numerem katalogowym wpisz w treści SMS-a dane wysyłki, np.:

Przykład ➔ **0000 Adam Czerski ul. Kwiatowa 6/12 60-160 Poznań**
zamówienia indywidualne

Przykład ➔ **0000 PPHU Webservice ul. Reymonta 3 60-160 Poznań NIP: 433-093-07-54**
zamówienia dla firm i instytucji

W przypadku gdy planujesz zakup kilku egzemplarzy tego samego tytułu, przed numerem katalogowym dopisz liczbę zamówionych książek wraz ze znakiem *.

Przykład ➔ **2*0000 Adam Czerski ul. Kwiatowa 6/12 60-160 Poznań**

Faktura VAT zostanie wysłana wraz z zamówionym towarem.

Koszty transportu i SMS-a

Domyślnie zamówienie będzie realizowane za pośrednictwem poczty polskiej, za zaliczeniem pocztowym (za książki zapłacisz przy ich odbiorze). W tym przypadku koszty transportu pokrywa wydawnictwo, do ceny książki doliczony zostanie jedynie koszt pobrania pocztowego. Jeśli życzysz sobie, by przesyłkę dostarczyła poczta kurierska, w treści SMS-a dopisz słowo **kurier**.

Przykład ➔ **0000 Adam Czerski ul. Kwiatowa 6/12 60-160 Poznań kurier**

Ceny przesyłek są ustalane przez Pocztę Polską i firmy spedycyjne:

- **4,50 zł** – przesyłka zwykłą pocztą za pobraniem na terenie Polski
- **11,99 zł** – przesyłka pocztą kurierską na terenie Polski

Koszty przesyłki mogą ulec zmianie. Koszt SMS-a określa operator Twojej sieci komórkowej.

Potwierdzenie przyjęcia zamówienia

Po otrzymaniu SMS-a z zamówieniem prześlemy informację zwrotną, potwierdzającą jego przyjęcie. W treści wiadomości znajdziesz numer zamówienia oraz całkowite koszty odbioru przesyłki książek: cena detaliczna książki + koszt pobrania pocztowego (lub koszt przesyłki kurierskiej).

Nie przyjmujemy zamówień wysłanych z internetowych bramek SMS.



Zamówienia telefoniczne:

0 801 33 99 00

Książki możesz zamówić telefonicznie, dzwoniąc pod numer **0 801 33 99 00**. Nasz konsultant doradzi Ci w wyborze książek, błyskawicznie realizując Twoje zamówienie. Koszt połączenia telefonicznego jest zgodny z taryfą operatora Twojej sieci telefonicznej.

НУ ПОГОДИ!

Elementarz hakera

Haker, wbrew utartym poglądom, nie jest osobą, której głównym celem jest niszczenie – haker to ktoś, kto podchodzi do standar-dowych problemów programistycznych w niestandardowy spo-sób, tworząc własne rozwiązania, często zaskakujące innych. Opracowywanie takich nietypowych rozwiązań wymaga wszech-stronnej wiedzy z zakresu programowania, znajomości systemu operacyjnego oraz umiejętności wynajdowania i stosowania nie-udokumentowanych funkcji języków programowania i platform systemowych.

„C++. Elementarz hakera” to książka przeznaczona dla wszyst-kich tych, którym „zwykłe” programowanie już nie wystarcza i którzy chcą stworzyć coś wyjątkowego. Przedstawia techniki, dzięki którym programy będą działać szybciej, a efekty ich dzia-łania będą zachwycić i zaskakiwać. Czytając ją, nauczysz się pisać aplikacje, które rozbawią lub zirytują innych użytkowników, tworzyć narzędzia do skanowania portów oraz wykorzystywać wiedzę o systemach operacyjnych i językach programowania do optymalizacji i przyspieszania działania programów.

Optymalizacja kodu źródłowego i usuwanie wąskich gardeł

Zasady prawidłowego projektowania aplikacji

Tworzenie programów-żartów

Programowanie w systemie Windows

Sieci i protokoły sieciowe

Implementacja obsługi sieci we własnych aplikacjach

Sztuczki ze sprzętem

Techniki hakerskie

Wiedząc, jak działają hakerzy, będziesz mógł zabezpieczyć swoje aplikacje przed atakami tych, którzy swoją wiedzę wykorzystują w niewłaściwy sposób.

na 39,90 zł

Księgarnia internetowa

<http://helion.pl>

Zamów najnowszy katalog:

<http://helion.pl>

Zamów informacje o nowościach:

<http://helion.pl>

Zamów cennik:

<http://helion.pl>

Zamów książkę SMS-em

691 HELION

Nr katalogowy: 2735

Szczegóły na odwrocie

Zamówienia telefoniczne

0 801 339900



ul. Chopina 6, 44-100 Gliwice

44-100 Gliwice, skr. poczt. 462

(32) 230-98-63

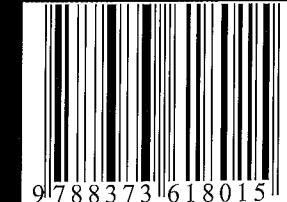
<http://helion.pl>

e-mail: helion@helion.pl

.pl

księgarnia
internetowa

ISBN 83-7361-801-5



9 788373 618015



C++

Elementarz
hakera



Zawiera CD



dft

Michael Flenov

Zawiera CD



ELITA
РОСИЙСКИХ
HAKERÓW
PREZENTUJE

НУ ПОГОДИ!

