helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C++. Elementarz hakera

Autor: Michael Flenov
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-7361-801-5
Tytuł oryginału: Hackish C++ Pranks & Tricks
Format: B5, stron: 296

Haker, wbrew utartym poglądom, nie jest osobą, której głównym celem jest niszczenie – haker to ktoś, kto podchodzi do standardowych problemów programistycznych w niestandardowy sposób, tworząc własne rozwiązania, często zaskakujące innych. Opracowywanie takich nietypowych rozwiązań wymaga wszechstronnej wiedzy z zakresu programowania, znajomości systemu operacyjnego i umiejętności wynajdowania i stosowania nieudokumentowanych funkcji języków programowania i platform systemowych.

„C++. Elementarz hakera" to książka przeznaczona dla wszystkich tych, którym „zwykłe" programowanie już nie wystarcza i którzy chcą stworzyć coś wyjątkowego. Przedstawia techniki, dzięki którym programy będą działać szybciej, a efekty ich działania będą zachwycać i zaskakiwać. Czytając ją nauczysz się pisać aplikacje, które rozbawią lub zirytują innych użytkowników, jak tworzyć narzędzia do skanowania portów i jak 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 w stanie zabezpieczyć swoje aplikacje przed atakami tych, którzy swoją wiedzę wykorzystują w niewłaściwy sposób.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

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 gardeł 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. Zacznij 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 tackę 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 drżączka ... 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

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 przeciążać 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.

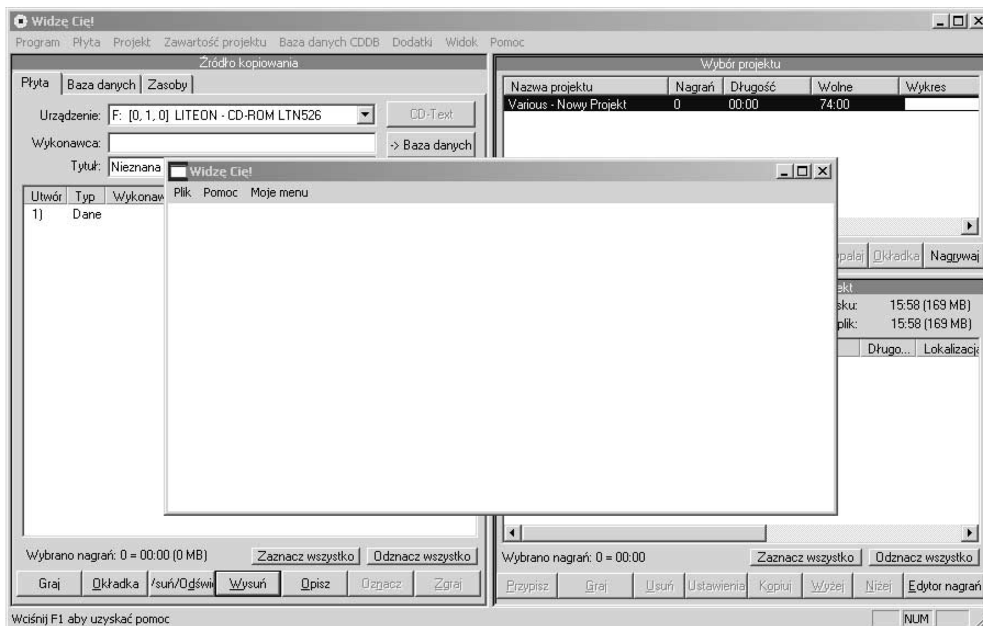
Będę starał się w tym rozdziale stopniować poziom zaawansowania przykładów, tak aby kolejne prezentacje wnosiły coś nowego i Czytelnik nie stracił zainteresowania tematem.

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 proponowana wcześniej metody wyszukiwania. Napiszmy na początek program, który wyszuka 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 `EnumWindows(&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 zostanie działające okno. Drugi parametr to liczba przekazywana do owej funkcji zwrotnej.

W roli funkcji zwrotnej występuje tu `EnumWindowsWnd`. Dla każdego okna, które znajdzie 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 znalezionego 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 nazwę 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 znaleźć w opisie odsyłacz do opisu formatu funkcji zwrotnej.

W momencie wywołania funkcji zwrotnej mamy do dyspozycji uchwyt następnego znalezionego 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:

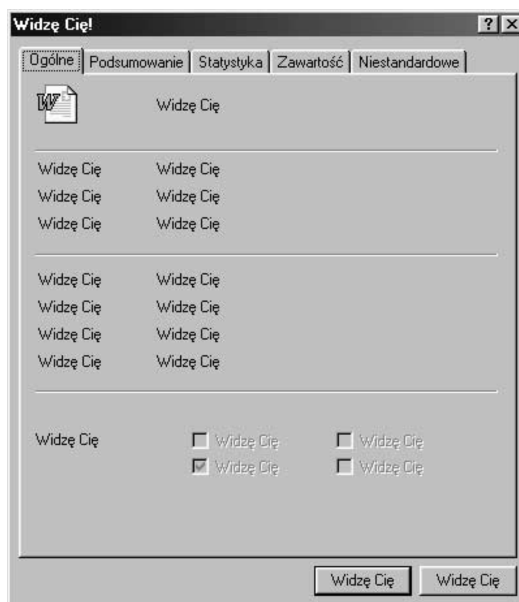
```
BOOL CALLBACK EnumWindowsWnd(
    HWND hwnd,        // uchwyt okna-rodzica
    LPARAM lParam     // parametr własny funkcji zwrotnej
   )
```

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

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

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

Łatwo zauważyć, 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 by uznać za kompletny, ale ma on jeszcze pewną słabość, którą trzeba wyeliminować. Przypuśćmy, że program znajdzie okno i rozpocznie wyliczanie jego okien potomnych, ale użytkownik nagle zamknie okno-rodzica. Program będzie próbował wysłać komunikat do znalezionego 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 płycie 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 sęk, 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,       // uchwyt 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 płycie CD-ROM w podkatalogu *\Przykłady\Rozdział3\RandMinimize*.

3.2. Gorączkowa drżączka

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 zwiększane 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 szerokość okna (rect.bottom - rect.top).

Na koniec funkcja zwrotna zwraca TRUE, umożliwiając dalsze poszukiwania okien.

Po uruchomieniu programu zobaczysz, jak wyświetlone 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 9*x*. 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 zamkniemy programu w ciągu kilku sekund, to również okno menedżera zadań ucieknie z ekranu i trzeba będzie zaczynać zamykanie 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 długo.

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 przenoszący 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 - 10, 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 wewnątrz 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 uchwyt 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 nadrzędnych 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 zaprząc 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 uchwytami 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 znacznych 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 spowolnić 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 zaczną uciekać na lewo. Spróbuj choćby wywołać menu podręczne 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. Zaciekawił mnie na tyle, że nie potrafiłem sobie odmówić takiego marnotrawstwa czasu. Szczególnie spodobał mi się przymus 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 pulpity. 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 dziewięćdziesiątych, 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. Pisałem 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 kwestą kształtów okien.

Na początek stwórzmy okno owalne. 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 owalnego 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.
- `WRect` typu `RECT` służącą do przechowywania rozmiaru i pozycji okna. Określają one obszar, wewnątrz którego zamknięty będzie owal okna.

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

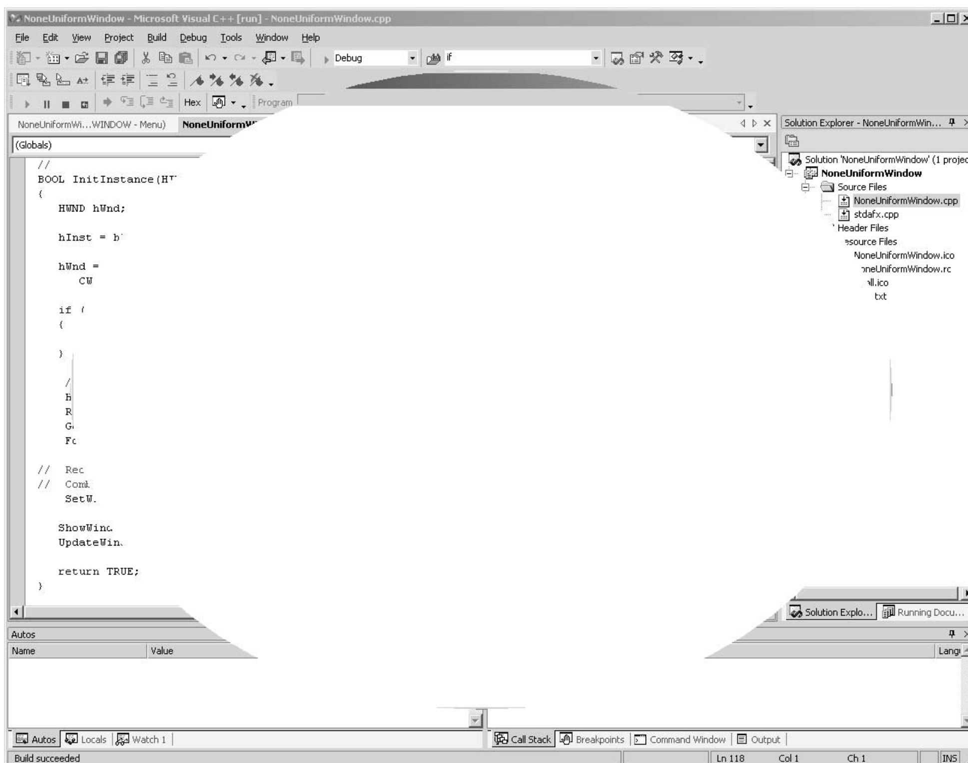
```
HRGN CreateEllipticRgn(
    int nLeftRect,       // współrzędna x górnego lewego narożnika obszaru elipsy
    int nTopRect,        // współrzędna y górnego lewego narożnika obszaru elipsy
    int nRightRect,      // współrzędna x dolnego prawego narożnika obszaru elipsy
    int nBottomRect      //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

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

```
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 ustawieniu regionu wywoływana jest funkcja `UpdateWindow`. Okno zostanie i tak odrysowane — trzeci parametr wywołania `SetWindowRgn` mógłby więc mieć równie dobrze wartość `FALSE`.

Uruchom program, a zobaczysz na ekranie okno owalne, jak na rysunku 3.3.



Rysunek 3.3. *Owalne okno programu*

Pójdźmy nieco dalej i utwórzmy owalne 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 owalny, utworzony funkcją CreateEllipticRgn. Druga ma przechowywać region prostokątny utworzony funkcją CreateRectRgn. Tak jak przy tworzeniu regionu owalnego 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 umieściłem jednolity barwnie podkład, żeby można było na jego tle zobaczyć właściwy kształt okna.

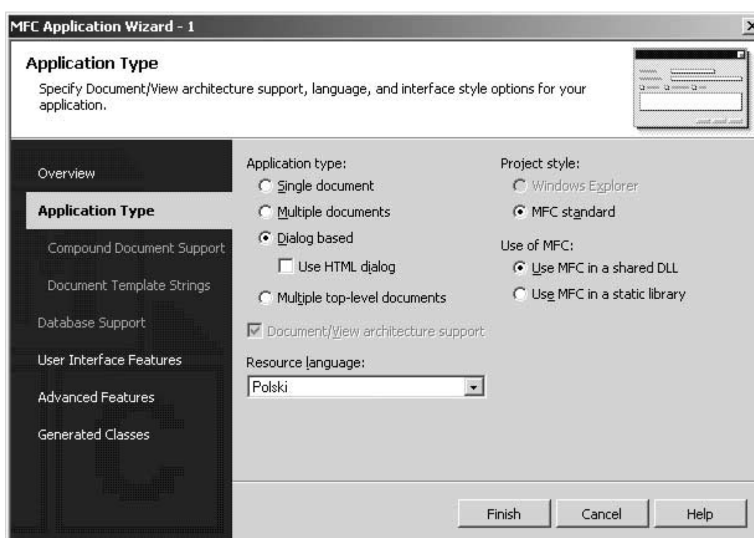
Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płycie CD-ROM w podkatalogu *\Przykłady\Rozdział3\NoneUniformWindow*.

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 uprościć sobie programowanie, korzystając z biblioteki MFC.

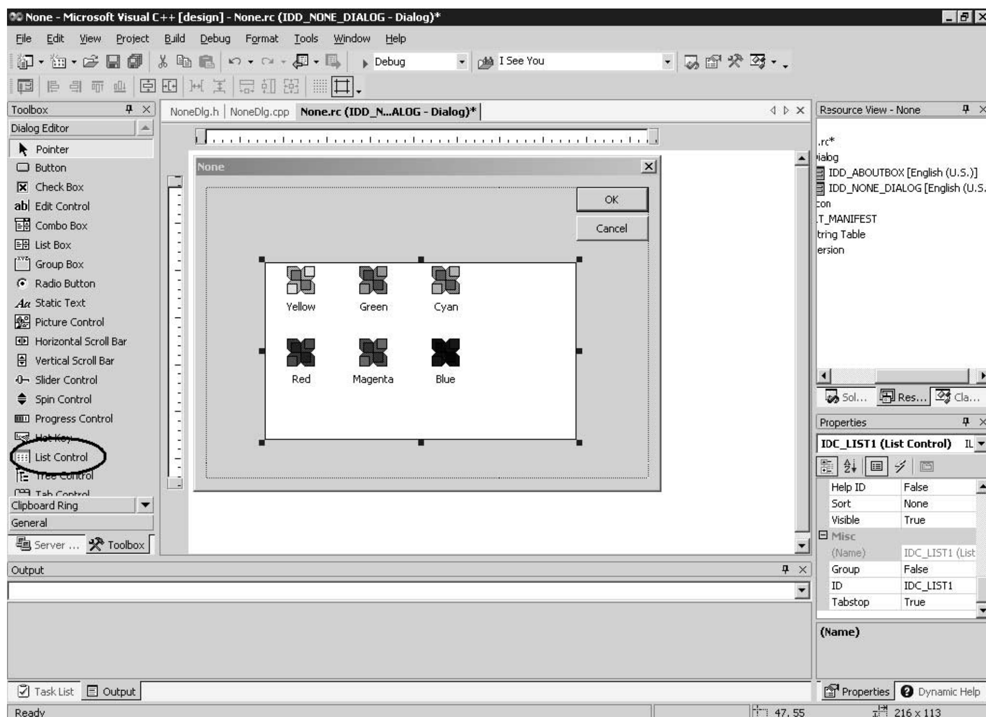
W kreatorze aplikacji przejdź na zakładkę *Application Type* i zaznacz pozycję *Dialog based* (patrz rysunek 3.5). Pozostałe parametry zostaw bez zmian. Ja swój projekt nazwałem *None*.



Rysunek 3.5. *Wybór typu aplikacji w oknie kreatora aplikacji MFC*

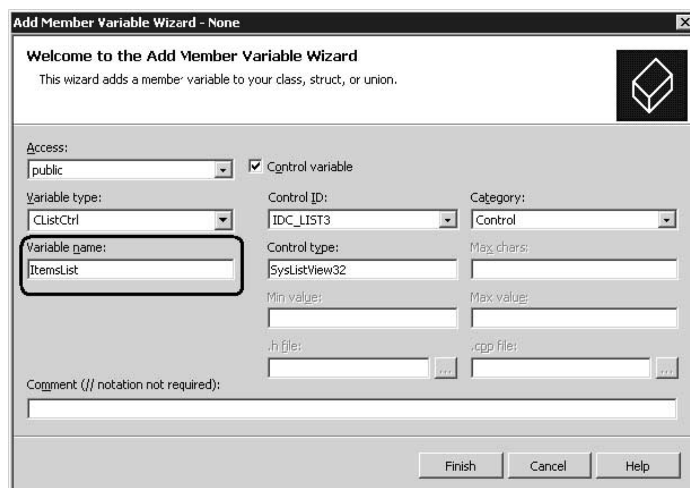
Otwórz przeglądarkę zasobów i kliknij dwukrotnie pozycję *IDD_NONE_DIALOG* w gałęzi *Dialog*. Umieść na formularzu programu jeden komponent *ListControl* (jak na rysunku 3.6).

Aby móc obsługiwać nowy element sterujący, kliknij go prawym przyciskiem myszy i wybierz z menu podręcznego polecenie *Add Variable…*. W oknie, które się pojawi, wpisz w polu *Variable Name* nazwę zmiennej. Nazwijmy ją `ItemsList` (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*



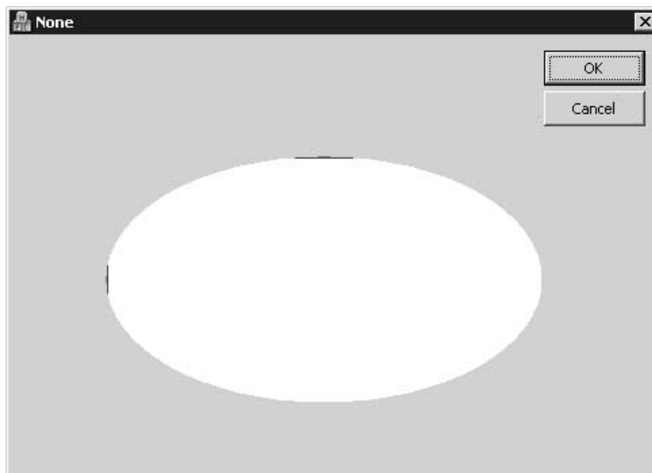
Otwórz teraz plik kodu źródłowego *NoneDlg.cpp* i znajdź w nim funkcję `CNoneDlg::OnInitDialog`. Dodaj do niej poniższy kod, umieszczając go na końcu funkcji za komentarzem *// TODO: Add extra initializacion here*:

```
// 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 `SetWindowRect` są poprzedzane znakami `::` wskazującymi, że funkcje te są wywoływane 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 pikselem tła. Zapamiętaj współrzędną początku prostokątnego regionu w zmiennej `X1`.
2. Przejrzyj resztę wiersza obrazka w poszukiwaniu przeciwległej granicy tła. Pozycję ostatniego nieprzezroczystego piksela zapamiętaj jako `X2`. Jeśli do końca wiersza nie znajdziesz pikseli tła, rozciągnij region do końca wiersza.
3. Ustaw współrzędną `Y1` na numer wiersza, a `Y2` na `Y1+1` (wysokość regionu prostokątnego obejmującego pojedynczy wiersz obrazka to jeden piksel).
4. Skonstruuj region na bazie otrzymanych współrzędnych.
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 przerywany 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 pikselem 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 kompilacji *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 okno 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 powinniśmy 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.

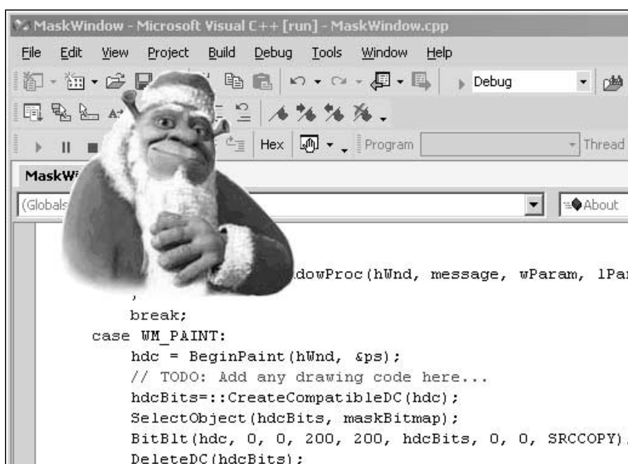
Jeśli uruchomisz przykład, zobaczysz okno takie jak na rysunku 3.10. Okno przyjmie kształt obrazka, ale będzie puste. Utworzyliśmy bowiem tylko kształt okna. Aby wypełnić okno obrazkiem, będziemy musieli uzupełnić program o następujący kod obsługi komunikatu WM_PAINT:

```
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 obraz 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 płycie 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 sygnał 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.

Możemy 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 płycie 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 zapomnimy 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.

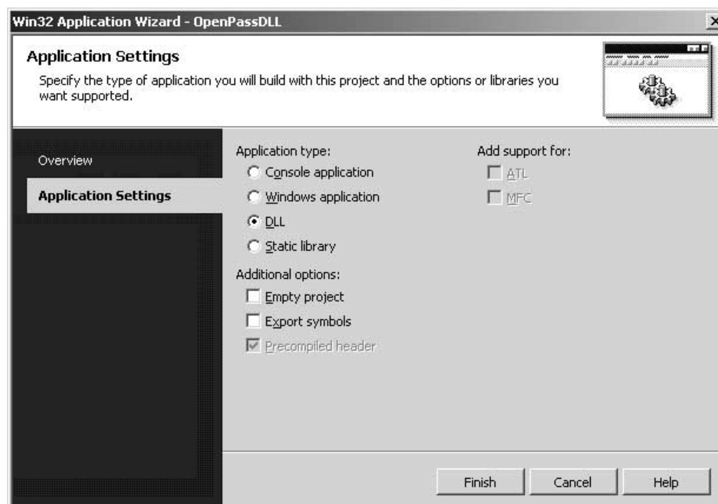
Zamiast iść na łatwiznę, sami napiszemy potrzebny program.

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

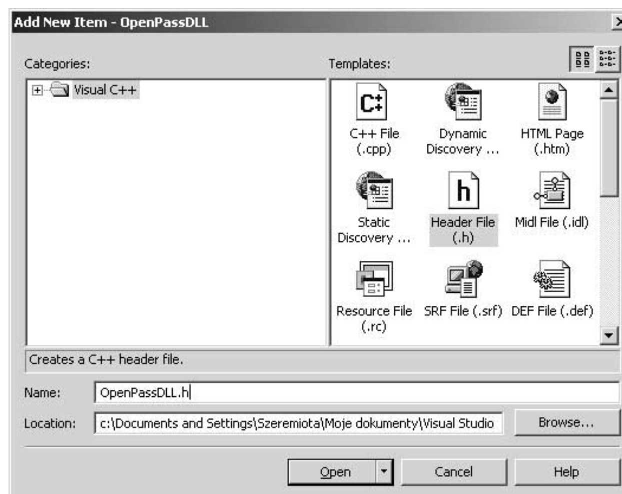
Dla potrzeb tego przykładu oprogramowałem specjalną bibliotekę DLL. Teraz zrobimy to samo wspólnie. Utwórz w Visual C++ nowy projekt *Win32 Project* i nazwij go *OpenPassDLL*. W kreatorze aplikacji wybierz *DLL* jako typ aplikacji (jak na rysunku 3.12).

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 *HeaderFile (.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*.

Przejdźmy 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;
}

LRESULT 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 uchwyt okna, które wygenerowało komunikat
        Wnd = ((tagMSG*)lParam)->hwnd;
```

```
            // Sprawdzenie typu komunikatu.
            // Czy użytkownik nacisnął prawy przycisk myszy?
            if (((tagMSG*)lParam)->message == WM_RBUTTONDOWN)
            {
                SendMessage(Wnd, EM_SETPASSWORDCHAR, 0, 0);
                InvalidateRect(Wnd, 0, true);
            }
        }

        return 0;
    }

    ///////////////////////////////////////////////////////////////////

    DllExport void RunStopHook(bool State, HINSTANCE hInstance)
    {
        if (true)
            SysHook = SetWindowsHookEx(WH_GETMESSAGE, &SysMsgProc, hInst, 0);
        else
            UnhookWindowsHookEx(SysHook);
    }
```

Przyjrzyjmy 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. Przeprowadza się w niej czynności inicjalizacyjne. W naszym przypadku nie mamy czego inicjalizować poza zachowaniem pierwszego parametru wywołania funkcji (uchwytu egzemplarza biblioteki) w zmiennej `hInst`.

Przejdźmy 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 przekazywana 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. Polega 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 kompletnie 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 zasłoniętego 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 płycie 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);
```

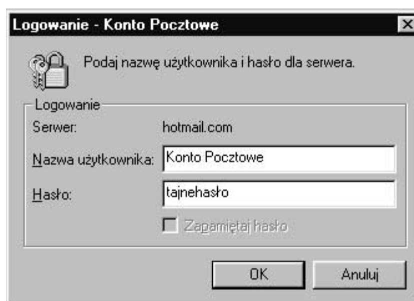
Zaraz po wywołaniu startuje główna pętla komunikatów, w której nie musimy nic zmieniać. Pod koniec programu musimy jedynie zwolnić zaczep i usunąć bibliotekę DLL z pamięci. Realizują to dwa wiersze:

```
(*RunStopHook) (FALSE, hInstance);
FreeLibrary(hModule);
```

Kod źródłowy tego przykładu oraz jego pliki wykonywalne znajdują się na dołączonej do książki płycie 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 jawny 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”*

```
LRESULT 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);
```

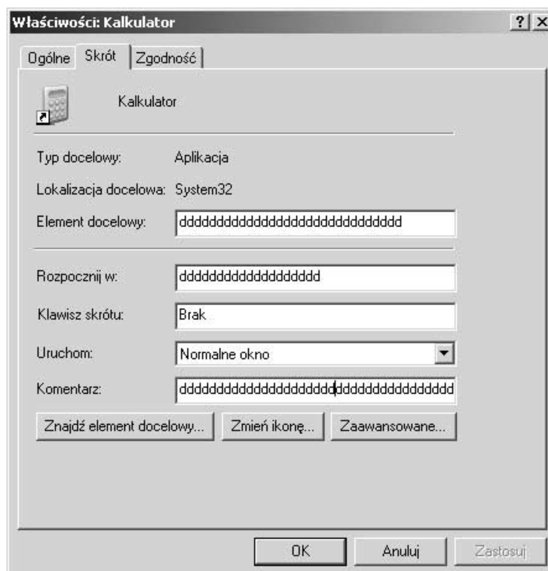
```
    // Sprawdź komunikat
    if (code == HC_ACTION)
    {
        // Pobierz uchwyt okna, które wygenerowało komunikat
        Wnd = ((tagMSG*)lParam)->hwnd;

        // Sprawdzenie typu komunikatu.
        // Czy użytkownik nacisnął 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 płycie 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 równie opiekunom systemów informatycznych i kierownictwu działów.

Haker może, na przykład, oczekiwać 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.

Próbując 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. Powinniśmy bowiem również w tym przypadku założyć zaczep, za pośrednictwem którego monitorowalibyśmy wybrane komunikaty systemowe. Poprzednio zaczep zakładaliśmy wywołaniem `SetWindowsHookEx`, a przechwytywaniu podlegały 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;
}

LRESULT 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 tę 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 uchwyt 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 następnym wierszu mamy wywołanie funkcji pozyskującej uchwyt okna potomnego, a w następnym — okna potomnego okna uzyskanego poprzednio. W ten sposób otrzymujemy uchwyt 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 uchwyt 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 uchwyt 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 uchwyt obiektu zarządzającego ikonami pulpitu).
- Komunikatem. LVM_SETITEMPOSITION wymusza zmianę pozycji ikony.
- Pierwszym parametrem komunikatu — numerem ikony do przesunięcia.
- Drugim parametrem komunikatu — nową pozycją 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 widocznie jedynie po odświeżeniu obrazu pulpitu.

Aby zmienić kolor podpisów pod ikonami, należy przesłać 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 efektowny, 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 wymuszamy jej odrysowanie komunikatem `LVM_UPDATE`. Trzeci parametr wywołania `SendMessage` to numer ikony do odświeżenia. Jeśli zachodziłaby 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*\ *ArrangeIcons* 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 pożyteczny 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 kopiowanie bloków tekstu. Czego użytkownik spodziewa się po schowku? Że wklejone z niego dane będą tymi samymi danymi, które wcześniej do niego skopiował. Zaskoczmy 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_CHANGECBCHAIN` 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_CHANGECBCHAIN()
    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_CHANGECBCHAIN,
                      (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 wklejał, 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.